
Rubrix

Release 0.18.0.dev0

Recognai

Oct 13, 2022

GETTING STARTED

1	Features	3
2	Quickstart	5
3	Use cases	9
4	Community	11
4.1	Setup and installation	11
4.1.1	1. Install Rubrix	11
4.1.2	2. Launch the web app	11
4.1.3	3. Start logging data	12
4.1.4	Next steps	12
4.2	Concepts	12
4.2.1	Rubrix data model	13
4.2.2	Methods	16
4.3	Basics	16
4.3.1	How to upload data	16
4.3.2	How to label datasets	22
4.3.3	How to prepare your data for training	25
4.3.4	How to train a model	27
4.4	User Management and Workspaces	27
4.4.1	User management model	27
4.4.2	Default user	28
4.4.3	How to add new users and workspaces	29
4.5	Advanced setup guides	30
4.5.1	Setting up Elasticsearch via docker	30
4.5.2	Server configurations	31
4.5.3	Launching the web app via docker	32
4.5.4	Launching the web app via docker-compose	32
4.5.5	Configure Elasticsearch role/users	34
4.5.6	Deploy to aws instance using docker-machine	35
4.5.7	Install from master	36
4.6	Rubrix Cookbook	37
4.6.1	Hugging Face Transformers	37
4.6.2	spaCy	43
4.6.3	Flair	45
4.6.4	Stanza	53
4.7	Tasks Templates	56
4.7.1	Text Classification	56
4.7.2	Token Classification	64

4.7.3	Text2Text	69
4.8	Weak supervision	70
4.8.1	Rubrix weak supervision in a nutshell	70
4.8.2	Built-in label models	72
4.8.3	Detailed Workflow	72
4.8.4	Example dataset	72
4.8.5	1. Create a Rubrix dataset with unlabelled data and test data	73
4.8.6	2. Defining rules	73
4.8.7	3. Building and analyzing weak labels	74
4.8.8	4. Using the weak labels	75
4.9	Monitoring NLP pipelines	83
4.9.1	Using <code>rb.monitor</code>	83
4.9.2	Using <code>rb.log</code> in background mode	84
4.9.3	Using the ASGI middleware	86
4.10	Metrics	86
4.10.1	Install dependencies	86
4.10.2	1. Rubrix Metrics for NER pipelines predictions	86
4.10.3	2. Rubrix Metrics for NER training sets	89
4.10.4	2. Rubrix Metrics for text classification	91
4.11	Datasets	92
4.11.1	Working with a Dataset	92
4.11.2	Importing from other formats	93
4.11.3	Sharing via the Hugging Face Hub	94
4.11.4	Prepare dataset for training	94
4.12	Dataset settings	95
4.12.1	Define a labeling schema	95
4.13	Queries	96
4.13.1	Search fields	96
4.13.2	<code>text</code> and <code>text.exact</code>	96
4.13.3	Words and phrases	97
4.13.4	Metadata fields	97
4.13.5	Filters as query string	97
4.13.6	Combine terms and fields	98
4.13.7	Query string features	98
4.13.8	Field glossary	99
4.14	Introductory	100
4.14.1	Label your data to fine-tune a classifier with Hugging Face	101
4.14.2	Building a news classifier with weak supervision	110
4.14.3	Few-shot classification with SetFit and a custom dataset	120
4.15	Model predictions	123
4.15.1	Explore and analyze <code>spaCy</code> NER pipelines	125
4.15.2	Zero-shot Named Entity Recognition with Flair	131
4.15.3	Analyzing predictions with model explainability methods	134
4.16	Weak supervision	138
4.16.1	Weak supervision in multi-label text classification tasks	139
4.16.2	Weakly supervised NER with <code>skweak</code>	155
4.16.3	Extending weak supervision workflows with sentence embeddings	169
4.17	Active Learning	185
4.17.1	Active learning with <code>ModAL</code> and <code>scikit-learn</code>	186
4.17.2	Active learning for text classification with <code>small-text</code>	194
4.18	Label errors	200
4.18.1	Clean labels using your model loss	201
4.18.2	Find label errors with <code>cleanlab</code>	208
4.19	Monitoring	214

4.19.1	Monitor predictions in HTTP API endpoints	214
4.20	Telemetry	218
4.20.1	How to opt-out	218
4.20.2	Why reporting telemetry	219
4.20.3	Sensitive data	219
4.20.4	Information reported	219
4.21	Python	220
4.21.1	Client	220
4.21.2	Metrics	236
4.21.3	Labeling	242
4.21.4	Listeners	255
4.22	Web App UI	257
4.22.1	Pages	257
4.22.2	Features	264
4.23	Developer documentation	280
4.23.1	Development setup	280
4.23.2	Building the documentation	280
Python Module Index		281
Index		283

Rubrix is a **production-ready framework for building and improving datasets** for NLP projects.

FEATURES

- **Open:** Rubrix is free, open-source, and 100% compatible with major NLP libraries (Hugging Face transformers, spaCy, Stanford Stanza, Flair, etc.). In fact, you can **use and combine your preferred libraries** without implementing any specific interface.
- **End-to-end:** Most annotation tools treat data collection as a one-off activity at the beginning of each project. In real-world projects, data collection is a key activity of the iterative process of ML model development. Once a model goes into production, you want to monitor and analyze its predictions, and collect more data to improve your model over time. Rubrix is designed to close this gap, enabling you to **iterate as much as you need**.
- **User and Developer Experience:** The key to sustainable NLP solutions is to make it easier for everyone to contribute to projects. *Domain experts* should feel comfortable interpreting and annotating data. *Data scientists* should feel free to experiment and iterate. *Engineers* should feel in control of data pipelines. Rubrix optimizes the experience for these core users to **make your teams more productive**.
- **Beyond hand-labeling:** Classical hand labeling workflows are costly and inefficient, but having humans-in-the-loop is essential. Easily combine hand-labeling with active learning, bulk-labeling, zero-shot models, and weak-supervision in **novel data annotation workflows**.

QUICKSTART

Getting started with Rubrix is easy, let's see a quick example using the `transformers` and `datasets` libraries:

```
pip install "rubrix[server]" "transformers[torch]" datasets
```

If you don't have [Elasticsearch \(ES\)](#) running, make sure you have `docker` installed and run:

Note: Check the [setup and installation section](#) for further options and configurations regarding Elasticsearch.

```
docker run -d --name elasticsearch-for-rubrix -p 9200:9200 -p 9300:9300 -e "ES_JAVA_
↳OPTS=-Xms512m -Xmx512m" -e "discovery.type=single-node" docker.elastic.co/
↳elasticsearch/elasticsearch-oss:7.10.2
```

Then simply run:

```
python -m rubrix
```

Afterward, you should be able to access the web app at <http://localhost:6900/>. **The default username and password are rubrix and 1234.**

If you have problems launching Rubrix, you can get direct support from the maintainers and other community member by joining [Rubrix's Slack Community](#)

Now, let's see an example: **Bootstrapping data annotation with a zero-shot classifier**

Why:

- The availability of pre-trained language models with zero-shot capabilities means you can, sometimes, accelerate your data annotation tasks by pre-annotating your corpus with a pre-trained zero-shot model.
- The same workflow can be applied if there is a pre-trained “supervised” model that fits your categories but needs fine-tuning for your own use case. For example, fine-tuning a sentiment classifier for a very specific type of message.

Ingredients:

- A zero-shot classifier from the Hub: `typeform/distilbert-base-uncased-mnli`
- A dataset containing news
- A set of target categories: Business, Sports, etc.

What are we going to do:

1. Make predictions and log them into a Rubrix dataset.
2. Use the Rubrix web app to explore, filter, and annotate some examples.

3. Load the annotated examples and create a training set, which you can then use to train a supervised classifier. Use your favourite editor or a Jupyter notebook to run the following:

```
from transformers import pipeline
from datasets import load_dataset
import rubrix as rb

model = pipeline('zero-shot-classification', model="typeform/squeezebert-mnli")

dataset = load_dataset("ag_news", split='test[0:100]')

labels = ['World', 'Sports', 'Business', 'Sci/Tech']

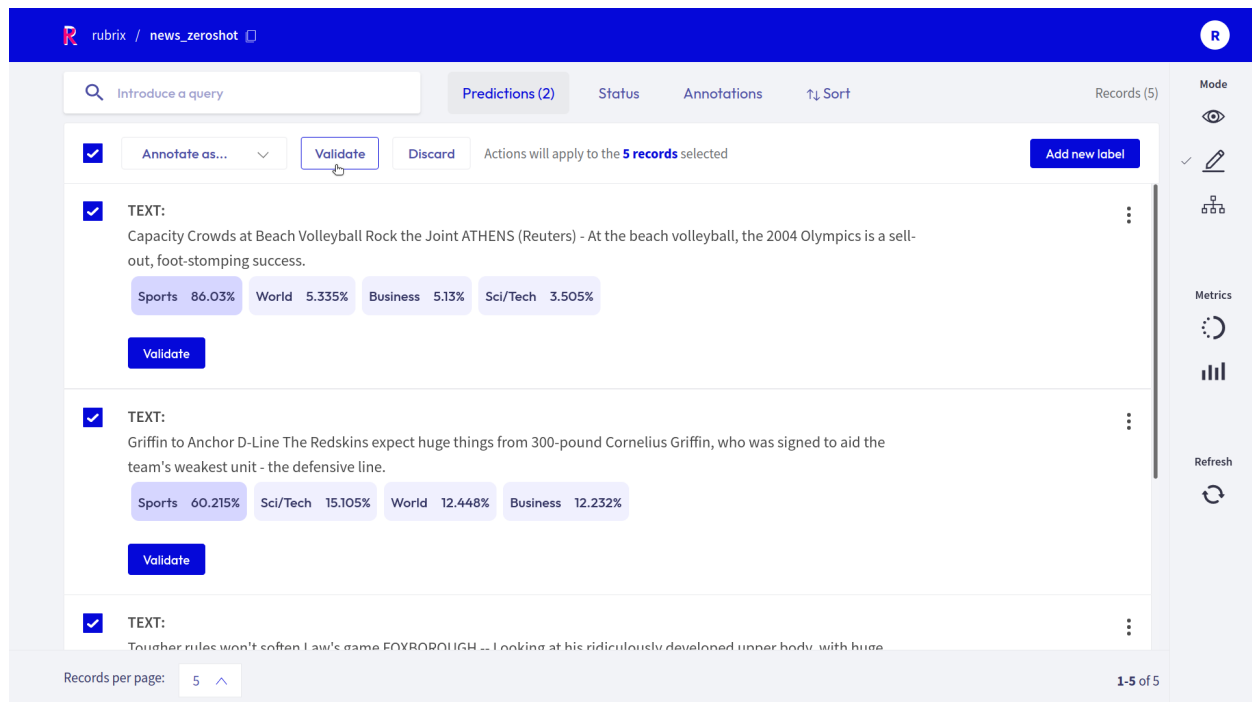
records = []
for record in dataset:
    prediction = model(record['text'], labels)

    records.append(
        rb.TextClassificationRecord(
            text=record["text"],
            prediction=list(zip(prediction['labels'], prediction['scores'])),
        )
    )

rb.log(records, name="news_zeroshot")
```

Now you can explore the records in the Rubrix UI at <http://localhost:6900/>. **The default username and password are rubrix and 1234.**

Let's filter the records predicted as Sports with high probability and use the bulk-labeling feature for labeling 5 records as Sports:



The screenshot shows the Rubrix UI interface for managing records. The top navigation bar includes the Rubrix logo, the current view 'news_zeroshot', and a user icon. Below the navigation bar, there's a search bar and tabs for 'Predictions (2)', 'Status', 'Annotations', and 'Sort'. A 'Records (5)' tab is also present. The main content area displays a list of records, each with a checkbox, a 'TEXT' field, and a table of predicted labels and scores. The first record is selected, and its 'Validate' button is highlighted. The second record is also selected. The third record is partially visible. On the right side, there's a sidebar with icons for 'Mode', 'Metrics', and 'Refresh'. At the bottom, there's a 'Records per page' dropdown set to 5 and a '1-5 of 5' indicator.

Record	Text	Sports	World	Business	Sci/Tech
1	Capacity Crowds at Beach Volleyball Rock the Joint ATHENS (Reuters) - At the beach volleyball, the 2004 Olympics is a sell-out, foot-stomping success.	86.03%	5.335%	5.13%	3.505%
2	Griffin to Anchor D-Line The Redskins expect huge things from 300-pound Cornelius Griffin, who was signed to aid the team's weakest unit - the defensive line.	60.215%	15.105%	12.448%	12.232%
3	Tougher rules won't soften Law's game FOXBOROUGH -- Looking at his ridiculously developed upper body, with huge				

After a few iterations of data annotation, we can load the Rubrix dataset and create a training set to train or fine-tune a supervised model.

```
# load the Rubrix dataset and put it into a pandas DataFrame
rb_df = rb.load(name='news_zeroshot').to_pandas()

# filter annotated records
rb_df = rb_df[rb_df.status == "Validated"]

# select text input and the annotated label
train_df = pd.DataFrame({
    "text": rb_df.text,
    "label": rb_df.annotation,
})
```


USE CASES

- **Data labelling and review:** collect labels to start a project from scratch or from existing live models.
- **Model monitoring and observability:** log and observe predictions of live models.
- **Evaluation:** easily compute “live” metrics from models in production, and slice evaluation datasets to test your system under specific conditions.
- **Model debugging:** log predictions during the development process to visually spot issues.
- **Explainability:** log token attributions to help you interpret model predictions.

COMMUNITY

You can join the conversation on Slack! We are a very friendly and inclusive community:

- [Slack community](#)

4.1 Setup and installation

In this guide, we will help you to get up and running with Rubrix. Basically, you need to:

1. Install Rubrix
2. Launch the web app
3. Start logging data

4.1.1 1. Install Rubrix

First, make sure you have Python 3.7 or above installed.

Then you can install Rubrix with `pip` or `conda`.

with `pip`

```
pip install "rubrix[server]"
```

with `conda`

```
conda install -c conda-forge "rubrix-server"
```

4.1.2 2. Launch the web app

Rubrix uses [Elasticsearch \(ES\)](#) as its main persistent layer. **If you do not have an ES instance running on your machine**, we recommend setting one up via docker:

```
docker run -d --name elasticsearch-for-rubrix -p 9200:9200 -p 9300:9300 -e "ES_JAVA_
↳OPTS=-Xms512m -Xmx512m" -e "discovery.type=single-node" docker.elastic.co/
↳elasticsearch/elasticsearch-oss:7.10.2
```

Note: For more details about setting up ES via docker, check our [advanced setup guide](#).

You can start the Rubrix web app via Python.

```
python -m rubrix
```

Afterward, you should be able to access the web app at <http://localhost:6900/>. **The default username and password are rubrix and 1234** (see the *user management guide* to configure this).

Note: You can also launch the web app via *docker* or *docker-compose*. For the latter you do not need a running ES instance.

4.1.3 3. Start logging data

The following code will log one record into a data set called `example-dataset`:

```
import rubrix as rb

rb.log(
    rb.TextClassificationRecord(text="My first Rubrix example"),
    name='example-dataset'
)
```

If you now go to your Rubrix app at <http://localhost:6900/>, you will find your first data set.

Congratulations! You are ready to start working with Rubrix.

4.1.4 Next steps

Have a look at our *advanced setup guides* if you want to (among other things):

- *setup Elasticsearch (ES) via docker*
- *configure the Rubrix server*
- *share an ES instance with other applications*
- *deploy Rubrix on an AWS instance*

To continue learning we recommend you to:

- Check our **Guides** and **Tutorials**;
- Read about Rubrix's main *concepts*;

4.2 Concepts

In this section, we introduce the core concepts of Rubrix. These concepts are important for understanding how to interact with the tool and its core Python client.

We have two main sections: Rubrix data model and Python client API methods.

4.2.1 Rubrix data model

The Python library and the web app are built around a few simple concepts. This section aims to clarify what those concepts are and to show you the main constructs for using Rubrix with your own models and data. Let's take a look at Rubrix's components and methods:

Dataset

A dataset is a collection of *records* of a common type. You can programmatically *build datasets* with the Rubrix client and *log* them to the web app. In the web app you can *dive into your dataset* to explore and annotate your records. You can also *load* your datasets back to the client and export it into various formats, or *prepare it for training* a model.

Record

A record is a data item composed of **text** inputs and, optionally, **predictions** and **annotations**.

Think of predictions as the classification that your system made over that input (for example: 'Virginia Woolf'), and think of annotations as the ground truth that you manually assign to that input (because you know that, in this case, it would be 'William Shakespeare'). Records are defined by the type of **task** they are related to. Let's see three different examples:

Text classification record

Text classification deals with predicting in which categories a text fits. As if you're shown an image you could quickly tell if there's a dog or a cat in it, we build NLP models to distinguish between a Jane Austen's novel or a Charlotte Bronte's poem. It's all about feeding models with labelled examples and seeing how they start predicting over the very same labels.

Let's see examples of a spam classifier.

```
record = rb.TextClassificationRecord(
    text="Access this link to get free discounts!",

    prediction = [('SPAM', 0.8), ('HAM', 0.2)],
    prediction_agent = "link or reference to agent",

    annotation = "SPAM",
    annotation_agent = "link or reference to annotator",

    # Extra information about this record
    metadata={
        "split": "train"
    },
)
```

Multi-label text classification record

Another similar task to Text Classification, but yet a bit different, is Multi-label Text Classification. Just one key difference: more than one label may be predicted. While in a regular Text Classification task we may decide that the tweet “I can’t wait to travel to Egypt and visit the pyramids” fits into the hashtag #Travel, which is accurate, in Multi-label Text Classification we can classify it as more than one hashtag, like #Travel #History #Africa #Sightseeing #Desert.

```
record = rb.TextClassificationRecord(  
    text="I can't wait to travel to Egypt and visit the pyramids",  
  
    multi_label = True,  
  
    prediction = [('travel', 0.8), ('history', 0.6), ('economy', 0.3), ('sports', 0.2)],  
    prediction_agent = "link or reference to agent",  
  
    annotation = ['travel', 'history'],  
    annotation_agent= "link or reference to annotator",  
)
```

Token classification record

Token classification kind-of-tasks are NLP tasks aimed to divide the input text into words, or syllables, and assign certain values to them. Think about giving each word in a sentence its grammatical category, or highlight which parts of a medical report belong to a certain speciality. There are some popular ones like NER or POS-tagging.

```
record = rb.TokenClassificationRecord(  
    text = "Michael is a professor at Harvard",  
    tokens = ["Micheal", "is", "a", "professor", "at", "Harvard"],  
  
    # Predictions are a list of tuples with all your token labels and its starting and  
→ending positions  
    prediction = [('NAME', 0, 7), ('LOC', 26, 33)],  
    prediction_agent = "link or reference to agent",  
  
    # Annotations are a list of tuples with all your token labels and its starting and  
→ending positions  
    annotation = [('NAME', 0, 7), ('ORG', 26, 33)],  
    annotation_agent = "link or reference to annotator",  
  
    metadata={ # Information about this record  
        "split": "train"  
    },  
)
```

Text2Text record

Text2text tasks, like text generation, are tasks where the model receives and outputs a sequence of tokens. Examples of such tasks are machine translation, text summarization, paraphrase generation, etc.

```
record = rb.Text2TextRecord(
    text = "Michael is a professor at Harvard",

    # The prediction is a list of texts or tuples if you want to add a score to a
    ↪ prediction
    prediction = ["Michael es profesor en Harvard", "Michael es un profesor de Harvard"],
    prediction_agent = "link or reference to agent",

    # The annotation a strings representing the expected output text for the given input
    ↪ text
    annotation = "Michael es profesor en Harvard"
)
```

Task

A task defines the objective and shape of the predictions and annotations inside a record. You can see our supported tasks at [tasks](#).

Settings

The dataset settings. For now, only a set of the predefined labels (labels schema) is configurable. Still, other settings like annotators, and metadata schema, are planned to be supported as part of dataset settings.

Annotation

An annotation is a piece information assigned to a record, a label, token-level tags, or a set of labels, and typically by a human agent.

Prediction

A prediction is a piece information assigned to a record, a label or a set of labels and typically by a machine process.

Metadata

Metadata will hold extra information that you want your record to have: if it belongs to the training or the test dataset, a quick fact about something regarding that specific record... Feel free to use it as you need!

4.2.2 Methods

These methods allow you to programmatically work with Rubrix datasets. To find more information about these methods, please check out the [Client](#).

rb.init

Initialize the python client: `rubrix.init()`

rb.log

Log a dataset from the client to the Rubrix web app: `rubrix.log()`

rb.load

Load a Rubrix dataset from the web app into the client: `rubrix.load()`

rb.delete

Delete a dataset with a given name: `rubrix.delete()`

4.3 Basics

This guide will help you get started with Rubrix to perform basic tasks such as uploading or annotating data.

4.3.1 How to upload data

The main component of the Rubrix data model is called a record. A dataset in Rubrix is a collection of these records. Records can be of different types depending on the currently supported tasks:

1. `TextClassificationRecord`: Records for text classification tasks;
2. `TokenClassificationRecord`: Records for token classification tasks;
3. `Text2TextRecord`: Records for text-to-text tasks;

The most critical attributes of a record that are common to all types are:

- `text`: The input text of the record (Required);
- `annotation`: Annotate your record in a task-specific manner (Optional);
- `prediction`: Add task-specific model predictions to the record (Optional);
- `metadata`: Add some arbitrary metadata to the record (Optional);

In Rubrix, records are created programmatically using the [client library](#) within a Python script, a [Jupyter notebook](#), or another IDE.

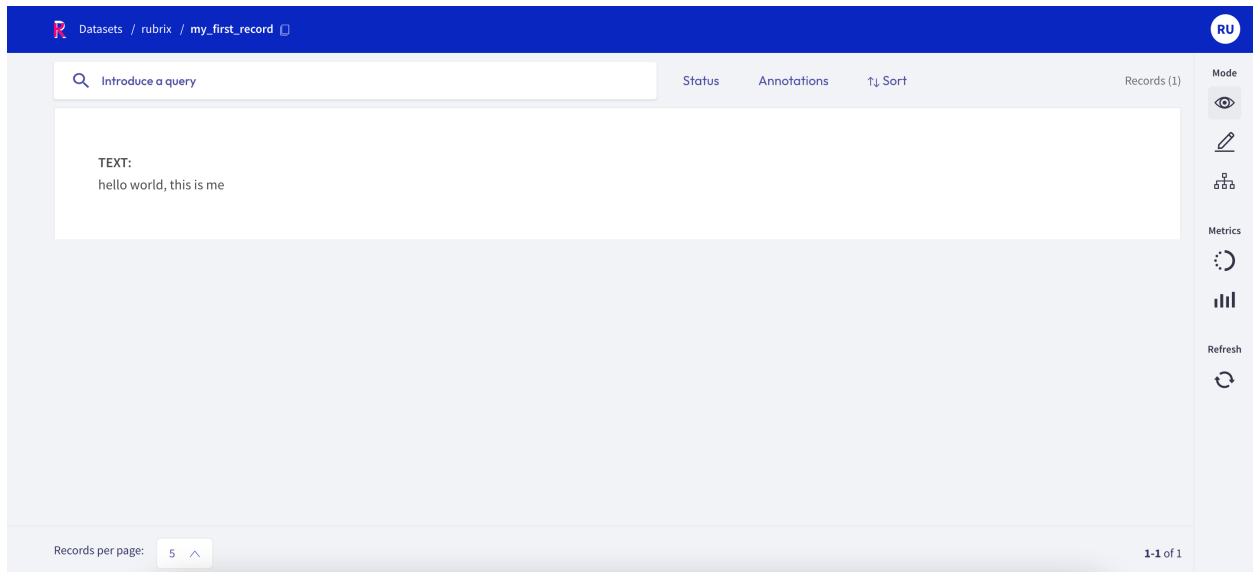
Let's see how to create and upload a basic record to the Rubrix web app (make sure Rubrix is already installed on your machine as described in the [setup guide](#)):

```
[ ]: import rubrix as rb

# Create a basic text classification record
record = rb.TextClassificationRecord(text="Hello world, this is me!")

# Upload (log) the record to the Rubrix web app
rb.log(record, "my_first_record")
```

Now you can access the “*my_first_record*” dataset in the Rubrix web app and look at your first record.



However, most of the time, you will have your data in some file format, like TXT, CSV, or JSON. Rubrix relies on two well-known Python libraries to read these files: `pandas` and `datasets`. After reading the files with one of those libraries, Rubrix provides shortcuts to create your records automatically.

Let us look at a few examples for each of the record types. **As mentioned earlier, you choose the record type depending on the task you want to tackle.**

1. Text classification

In this example, we will read a `CSV` file from a Kaggle competition that contains reviews for the Snapchat app. The underlying task here could be to classify the reviews by their sentiment.

Let us read the file with `pandas`

Note

If the file is too big to fit in memory, try using the `datasets` library with no memory constraints, as shown in the next section.

```
[ ]: import pandas as pd

# Read the CSV file into a pandas DataFrame
dataframe = pd.read_csv("Snapchat_app_store_reviews.csv")
```

and have a quick look at the first three rows of the resulting `pandas DataFrame`:

```
[41]: dataframe.head(3)
```

```
[41]:   Unnamed: 0      userName  rating  \
0          0  Savvanananahhh      4
1          1    Idek 9-101112      3
2          2  William Quintana      3

      review  isEdited      date  \
0  For the most part I quite enjoy Snapchat it's ...   False  10/4/20  6:01
1  I'm sorry to say it, but something is definite...   False  10/14/20  2:13
2  Snapchat update ruined my story organization! ...   False  7/31/20  19:54

      title
0  Performance issues
1  What happened?
2  STORY ORGANIZATION RUINED!
```

We will choose the *review* column as input text for our records. For Rubrix to know, we have to rename the corresponding column to *text*.

```
[ ]: # Rename the 'review' column to 'text',
dataframe = dataframe.rename(columns={"review": "text"})
```

We can now read this DataFrame with Rubrix, which will automatically create the records and put them in a *Rubrix Dataset*.

```
[ ]: import rubrix as rb

# Read DataFrame into a Rubrix Dataset
dataset_rb = rb.read_pandas(dataframe, text="TextClassification")
```

We will upload this dataset to the web app and give it the name *snapchat_reviews*

```
[ ]: # Upload (log) the Dataset to the web app
rb.log(dataset_rb, "snapchat_reviews")
```


Datasets / rubrix / snapchat_reviews

Introduce a query

Status Annotations Sort

Records (9,560)

Mode

Metrics

Refresh

TEXT:

i personally love this app it provides an easy way to communicate with friends more. this app also has good filters fun games to play with friends a large map stories and memories i love this app i like how it provides many things. zodiac signs last year's memories suggested stories and so much more. it is much easier to communicate and show feelings towards i really enjoy this app and all the add on features it has however i do not think it is "fair" that some people get the new dark mode feature and others don't. some people who have been suggesting this feature don't even get it. many many people have been waiting for this filter and it is a disappointment when other people have it and you

Show full record

TEXT:

So snap has been bothering me for about a year now every time I get a snap now I don't get a notification till 4 or 5 minutes later it's also my main app but I may delete it soon because of this especially when you're setting there waiting for a snap 4 minutes later it pops up and you have to ask whoever you're talking to to tell them you forgot what you have said it's very annoying I might delete and return to my regular text messages because it's more reliable. Also I am not the only person having this problem it happens to most of my peers

Records per page: 5

1 2 3 ... 1912 Next

1-5 of 9,560

2. Token classification

We will use German reviews of organic coffees in a [CSV file](#) for this example. The underlying task here could be to extract all attributes of an organic coffee.

This time, let's read the file with [datasets](#).

```
[ ]: from datasets import Dataset

# Read the csv file
dataset = Dataset.from_csv("kaffee_reviews.csv")
```

and have a quick look at the first three rows of the resulting `dataset` `Dataset`:

```
[94]: # The best way to visualize a Dataset is actually via pandas
dataset.select(range(3)).to_pandas()
```

```
[94]:   Unnamed: 0    brand  rating \
0          0  GEPA Kaffee      5
1          1  GEPA Kaffee      5
2          2  GEPA Kaffee      5

      review
0  Wenn ich Bohnenkaffee trinke (auf Arbeit trink...
1  Für mich ist dieser Kaffee ideal. Die Grundvor...
2  Ich persönlich bin insbesondere von dem Geschm...
```

We will choose the `review` column as input text for our records. For Rubrix to know, we have to rename the corresponding column to `text`.

```
[95]: dataset = dataset.rename_column("review", "text")
```

In contrast to the other types, token classification records need the input text **and** the corresponding tokens. So let us tokenize our input text in a small helper function and add the tokens to a new column called *tokens*.

Note

We will use `spaCy` to tokenize the text, but you can use whatever library you prefer.

```
[ ]: import spacy

# Load a german spaCy model to tokenize our text
nlp = spacy.load("de_core_news_sm")

# Define our tokenize function
def tokenize(row):
    tokens = [token.text for token in nlp(row["text"])]
    return {"tokens": tokens}

# Map the tokenize function to our dataset
dataset = dataset.map(tokenize)
```

Let us have a quick look at our extended Dataset:

```
[97]: dataset.select(range(3)).to_pandas()
```

```
[97]:
```

	Unnamed: 0	brand	rating	\
0	0	GEPA Kaffee	5	
1	1	GEPA Kaffee	5	
2	2	GEPA Kaffee	5	

	text	\
0	Wenn ich Bohnenkaffee trinke (auf Arbeit trink...	
1	Für mich ist dieser Kaffee ideal. Die Grundvor...	
2	Ich persönlich bin insbesondere von dem Geschm...	

	tokens
0	[Wenn, ich, Bohnenkaffee, trinke, (, auf, Arbe...
1	[Für, mich, ist, dieser, Kaffee, ideal, ., Die...
2	[Ich, persönlich, bin, insbesondere, von, dem,...

We can now read this Dataset with Rubrix, which will automatically create the records and put them in a *Rubrix Dataset*.

```
[ ]: import rubrix as rb

# Read Dataset into a Rubrix Dataset
dataset_rb = rb.read_dataset(dataset, task="TokenClassification")
```

We will upload this dataset to the web app and give it the name `coffee_reviews`

```
[ ]: # Log the dataset to the Rubrix web app
rb.log(dataset_rb, "coffee-reviews")
```

The screenshot shows the Rubrix Datasets interface for a dataset named 'coffee-reviews'. The top navigation bar is blue with the Rubrix logo and the dataset name. Below the navigation bar, there is a search bar and filters for Status, Annotations, and Sort. The main content area displays a table of records. Two records are visible, each containing a German text snippet. The right sidebar shows icons for Mode, Metrics, and Refresh.

3. Text2Text

In this example, we will use English sentences from the European Center for Disease Prevention and Control available at the [Hugging Face Hub](#). The underlying task here could be to translate the sentences into other European languages.

Let us load the data with `datasets` from the [Hub](#).

```
[ ]: from datasets import load_dataset

# Load the Dataset from the Hugging Face Hub and extract the train split
dataset = load_dataset("europa_ecdc_tm", "en2fr", split="train")
```

and have a quick look at the first row of the resulting dataset `Dataset`:

```
[101]: dataset[0]
[101]: {'translation': {'en': 'Vaccination against hepatitis C is not yet available.',
                        'fr': 'Aucune vaccination contre l'hépatite C n'est encore disponible.'}}
```

We can see that the English sentences are nested in a dictionary inside the `translation` column. To extract the phrases into a new `text` column, we will write a quick helper function and `map` the whole `Dataset` with it.

```
[ ]: # Define our helper extract function
def extract(row):
    return {"text": row["translation"]["en"]}

# Map the extract function to our dataset
dataset = dataset.map(extract)
```

Let us have a quick look at our extended `Dataset`:

```
[103]: dataset[0]
[103]: {'translation': {'en': 'Vaccination against hepatitis C is not yet available.',
  'fr': 'Aucune vaccination contre l'hépatite C n'est encore disponible.'},
  'text': 'Vaccination against hepatitis C is not yet available.'}
```

We can now read this Dataset with Rubrix, which will automatically create the records and put them in a *Rubrix Dataset*.

```
[ ]: import rubrix as rb

# Read Dataset into a Rubrix Dataset
dataset_rb = rb.read_dataset(dataset, tool="Text2Text")
```

We will upload this dataset to the web app and give it the name `ecdc_en`

```
[ ]: # Log the dataset to the Rubrix web app
rb.log(dataset_rb, "ecdc_en")
```

The screenshot shows the Rubrix web application interface. At the top, there's a blue header with the Rubrix logo and the text 'Datasets / rubrix / ecdc_en'. Below the header, there's a search bar with the placeholder 'Introduce a query'. To the right of the search bar are tabs for 'Status', 'Annotations', and 'Sort'. Further right, it says 'Records (2,561)'. On the far right, there's a sidebar with icons for 'Mode' (an eye), 'Metrics' (a bar chart), and 'Refresh' (a circular arrow). The main content area displays a list of records. The first record shows a text snippet: 'The ECDC will organize in 2009 together with WHO-GSS, an advanced workshop for EU MS, EEA/EFTA and candidate countries on detection surveillance and response to foodborne diseases.' The second record shows another text snippet: 'The WHO Regional Office for Europe (WHO/EURO), in particular, has tasks and responsibilities that interlink with those of ECDC.' The third record shows the text 'Recently updated!'. At the bottom, there's a pagination bar that says 'Records per page: 5' with a dropdown arrow, followed by page numbers '1', '2', '3', '...', '513', and a 'Next >' button. On the far right of the pagination bar, it says '1-5 of 2,561'.

4.3.2 How to label datasets

Rubrix provides several ways to label your data. Using Rubrix's UI, you can mix and match the following options:

1. Manually labeling each record using the specialized interface for each task type;
2. Leveraging a user-provided model and validating its predictions;
3. Defining heuristic rules to produce “noisy labels” which can then be combined with weak supervision;

Each way has its pros and cons, and the best match largely depends on your individual use case.

1. Manual labeling

The screenshot shows the Rubrix manual labeling interface. At the top, there's a blue header with the Rubrix logo and the dataset name 'snapchat_reviews'. Below the header, a search bar is present with the placeholder 'Introduce a query'. To the right of the search bar are tabs for 'Status', 'Annotations', and 'Sort'. Further right, it says 'Records (9,560)'. On the far right, there's a 'Mode' dropdown set to 'RU'.

The main content area displays a list of records. The first record is selected, showing a text snippet: 'So snap has been bothering me for about a year now every time I get a snap now I don't get a notification till 4 or 5 minutes later it's also my main app but I may delete it soon because of this especially when you're setting there waiting for a snap 4 minutes later it pops up and you have to ask whoever you're talking to to tell them you forgot what you have said it's very annoying I might delete and return to my regular text messages because it's more reliable. Also I am not the only person having this problem it happens to most of my peers'. Below the text, there are two buttons: 'positive' and 'negative'. The 'negative' button is highlighted, indicating it's the selected label. To the right of the text, there's a 'Validated' button and a three-dot menu.

At the bottom of the interface, there's a pagination bar. It shows 'Records per page: 5' with a dropdown arrow. To the right, there are page numbers '1', '2', '3', and '1912', followed by a 'Next' button. On the far right of the pagination bar, it says '1-5 of 9,560'.

The straightforward approach of manual annotations might be necessary if you do not have a suitable model for your use case or cannot come up with good heuristic rules for your dataset. It can also be a good approach if you dispose of a large annotation workforce or require few but unbiased and high-quality labels.

Rubrix tries to make this relatively cumbersome approach as painless as possible. Via an intuitive and adaptive UI, its exhaustive search and filter functionalities, and bulk annotation capabilities, Rubrix turns the manual annotation process into an efficient option.

Look at our dedicated [feature reference](#) for a detailed and illustrative guide on manually annotating your dataset with Rubrix.

2. Validating predictions

The screenshot shows the Rubrix web app interface for validating predictions. The top navigation bar includes 'Datasets / recognai / concise-concepts'. The main content area is divided into a 'Predictions' tab and a 'Status' column. The 'Predictions' tab shows a list of records with predicted categories (CARBS, DAIRY, FRUIT, HERBS, MEAT, VEGETABLE) and a 'Status' column. The first record is selected, showing a text snippet about smoothies with annotations for 'Peanut Butter', 'Banana', and 'Spinach'. The 'Stats' panel on the right shows counts for each category: DAIRY (1), FRUIT (3), HERBS (4), MEAT (5), and VEGETABLE (6). The 'Mentions' panel shows a list of items with their predicted and annotated counts.

Nowadays, many pre-trained or zero-shot models are available online via model repositories like the Hugging Face Hub. Most of the time, you probably will find a model that already suits your specific dataset task to some degree. In Rubrix, you can pre-annotate your data by including predictions from these models in your records. Assuming that the model works reasonably well on your dataset, you can filter for records with high prediction scores and validate the predictions. In this way, you will rapidly annotate part of your data and alleviate the annotation process.

One downside of this approach is that your annotations will be subject to the possible biases and mistakes of the pre-trained model. When guided by pre-trained models, it is common to see human annotators get influenced by them. Therefore, it is advisable to avoid pre-annotations when building a rigorous test set for the final model evaluation.

Check the [introduction tutorial](#) to learn to add predictions to the records. And our [feature reference](#) includes a detailed guide on validating predictions in the Rubrix web app.

3. Defining rules (weak labeling)

Datasets / rubrix / go_emotions

text:(thanks AND good)

Annotations Status Sort

Records (33)

text:(thanks AND good) Records: 33

admiration annoyance approval curiosity **gratitude** optimism

Save rule

Rule Metrics

Coverage	0.784%	Annotated coverage	1.075%
	33/4,208		4/372
Precision	100.00%	Correct/Incorrect	8/0

Manage rules

TEXT:
 I think you're right. I've been proactive before and it helped. Thanks and hope this year is good to you as well!

Records per page: 5

1 2 3 ... 7 Next

1-5 of 33

Another approach to annotating your data is to define heuristic rules tailored to your dataset. For example, let us assume you want to classify news articles into the categories of *Finance*, *Sports*, and *Culture*. In this case, a reasonable rule would be to label all articles that include the word “stock” as *Finance*.

Rules can get arbitrarily complex and can also include the record’s metadata. The downsides of this approach are that it might be challenging to come up with working heuristic rules for some datasets. Furthermore, rules are rarely 100% precise and often conflict with each other. These noisy labels can be cleaned up using weak supervision and label models, or something as simple as majority voting. It is usually a trade-off between the amount of annotated data and the quality of the labels.

Check [our guide](#) for an extensive introduction to weak supervision with Rubrix. Also, check the [feature reference](#) for the Define rules mode of the web app and our [various tutorials](#) to see practical examples of weak supervision workflows.

4.3.3 How to prepare your data for training

Once you have uploaded and annotated your dataset in Rubrix, you are ready to prepare it for training a model. Most NLP models today are trained via [supervised learning](#) and need input-output pairs to serve as training examples for the model. The input part of such pairs is usually the text itself, while the output is the corresponding annotation.

Manual extraction

The exact data format for training a model depends on your *training framework* and the task you are tackling (text classification, token classification, etc.). Rubrix is framework agnostic; you can always manually extract from the records what you need for the training.

The extraction happens using the *client library* within a Python script, a Jupyter notebook, or another IDE. First, we have to load the annotated dataset from the Rubrix UI:

```
[ ]: import rubrix as rb

dataset = rb.load("my_annotated_dataset")
```

Note

If you follow a weak supervision approach, the steps are slightly different. We refer you to our *weak supervision guide* for a complete workflow.

Then we can iterate over the records and extract our training examples. For example, let's assume you want to train a text classifier with a *sklearn pipeline* that takes as input a text and outputs a label.

```
[ ]: # Save the inputs and labels in Python lists
inputs, labels = [], []

# Iterate over the records in the dataset
for record in dataset:

    # We only want records with annotations
    if record.annotation:
        inputs.append(record.text)
        labels.append(record.annotation)

# Train the model
sklearn_pipeline.fit(inputs, labels)
```

Automatic extraction

For a few frameworks and tasks, Rubrix provides a convenient method to automatically extract training examples in a suitable format from a dataset.

For example: If you want to train a *transformers* model for text classification, you can load an annotated dataset for text classification and call the `prepare_for_training()` method:

```
[ ]: dataset = rb.load("my_annotated_dataset")

dataset_for_training = dataset.prepare_for_training()
```

With the returned `dataset_for_training`, you can continue following the steps to *fine-tune* a *pre-trained model* with the *transformers* library.

Check the dedicated *dataset guide* for more examples of the `prepare_for_training()` method.

4.3.4 How to train a model

Rubrix helps you to create and curate training data. **It is not a framework for training a model.** You can use Rubrix complementary with other excellent open-source frameworks that focus on developing and training NLP models.

Here we list three of the most commonly used open-source libraries, but many more are available and may be more suited for your specific use case:

- **transformers**: This library provides thousands of pre-trained models for various NLP tasks and modalities. Its excellent documentation focuses on fine-tuning those models to your specific use case;
- **spaCy**: This library also comes with pre-trained models built into a pipeline tackling multiple tasks simultaneously. Since its a purely NLP library, it comes with much more NLP features than just model training;
- **scikit-learn**: This de facto standard library is a powerful swiss army knife for machine learning with some NLP support. Usually, their NLP models lack the performance when compared to transformers or spacy, but give it a try if you want to train a lightweight model quickly;

Check our [cookbook](#) for many examples of how to train models using these frameworks together with Rubrix.

4.4 User Management and Workspaces

This guide explains how to setup the users and team workspaces for your Rubrix instance.

Let's first describe Rubrix's user management model:

4.4.1 User management model

User

A Rubrix user is defined by the following fields:

- **username**: The username to use for login into the Webapp.
- **email(optional)**: The user's email.
- **fullname (optional)**: The user's full name
- **disabled(optional)**: Whether this use is enabled (and can interact with Rubrix), this might be useful for disabling user access temporarily.
- **workspaces(optional)**: The team workspaces where the user has read and write access (both from the Webapp and the Python client). If this field is not defined the user will be a super-user and have access to all datasets in the instance. If this field is set to an empty list [] the user will only have access to her user workspace. Read more about workspaces and users below.
- **api_key**: The API key to interact with Rubrix API, mainly through the Python client but also via HTTP for advanced users.

Workspace

A workspace is a Rubrix “space” where users can collaborate, both using the Webapp and the Python client. There are two types of workspace:

- **Team workspace:** Where one or several users have read/write access.
- **User workspace:** Every user gets its own user workspace. This workspace is the default workspace when users log and load data with the Python client. The name of this workspace corresponds to the username.

A user is given access to a workspace by including the name of the workspace in the list of workspaces defined by the `workspaces` field. **Users with no defined workspaces field are super-users** and have access and right to all datasets.

Python client methods and workspaces

The Python client gives developers the ability to log, load, and copy datasets from and to different workspace. Check out the *Python Reference* for the parameter and methods related to workspaces. Some examples are:

```
import rubrix as rb

# After this init, all logging and loading will use the specified workspace
rb.init(workspace="my_shared_workspace")

# Setting the workspace explicitly will also affect all logging and loading
rb.set_workspace("my_private_workspace")
```

users.yml

The above user management model is configured using a YAML file which server maintainers can define before launching a Rubrix instance. This can be done when launching Rubrix from Python or with the provided `docker-compose.yml`. Read below for more details on the different options.

4.4.2 Default user

By default, if you don’t configure a `users.yml` file, your Rubrix instance is pre-configured with the following default user:

- `username: rubrix`
- `password: 1234`
- `api_key: rubrix.apikey`

for security reasons we recommend changing at least the password and API key.

How to override the default API key

To override the default API key you can set the following environment variable before launching the server:

```
export RUBRIX_LOCAL_AUTH_DEFAULT_APIKEY=new-apikey
```

How to override the default user password

To override the password, you must set an environment variable that contains an already hashed password. You can use `htpasswd` to generate a hashed password:

```
htpasswd -nbB "" my-new-password
:$apr1$n0C4S20a$noG.3yWxH10IKfFITgz130
```

Afterwards, set the environment variable omitting the first `:` character (in our case `$apr1$n0C4...`):

```
export RUBRIX_LOCAL_AUTH_DEFAULT_PASSWORD="<generated_user_password>"
```

Alternatively, you can also generate the hash using `passlib`'s `CryptContext`: E.g., by running the following in a python console:

```
from passlib.context import CryptContext
print(CryptContext(schemes=["bcrypt"], deprecated="auto").hash('password'))
```

4.4.3 How to add new users and workspaces

To configure your Rubrix instance for various users, you just need to create a yaml file as follows:

```
#.users.yaml
# Users are provided as a list
- username: user1
  hashed_password: <generated-hashed-password> # See the previous section above
  api_key: "ThisIsTheUser1APIKEY"
  workspaces: [] # This user will only have her user workspace available
- username: user2
  hashed_password: <generated-hashed-password> # See the previous section above
  api_key: "ThisIsTheUser2APIKEY"
  workspaces: ['client_projects'] # access to her user workspace and the client_projects_
↳workspace
- username: user3
  hashed_password: <generated-hashed-password> # See the previous section above
  api_key: "ThisIsTheUser2APIKEY" # this user can access all workspaces (including
- ...
```

Then point the following environment variable to this yaml file before launching the server:

```
export RUBRIX_LOCAL_AUTH_USERS_DB_FILE=/path/to/.users.yaml
```

If everything went well, the configured users can now log in and their annotations will be tracked with their usernames.

Using docker-compose

Make sure you create the yaml file above in the same folder as your `docker-compose.yaml`. You can download the `docker-compose` from this [URL](#):

Then open the provided `docker-compose.yaml` and configure your Rubrix instance as follows:

```
# docker-compose.yaml
services:
  rubrix:
    image: recognai/rubrix:master
    ports:
      - "6900:80"
    environment:
      ELASTICSEARCH: http://elasticsearch:9200
      RUBRIX_LOCAL_AUTH_USERS_DB_FILE: /config/.users.yaml

  volumes:
    # We mount the local file .users.yaml in remote container in path /config/.users.
    ↪yaml
    - $PWD/.users.yaml:/config/.users.yaml
    ...
```

You can reload the *Rubrix* service to refresh the container:

```
docker-compose up -d rubrix
```

If everything went well, the configured users can now log in, their annotations will be tracked with their usernames, and they'll have access to the defined workspaces.

4.5 Advanced setup guides

Here we provide some setup guides for an advanced usage of Rubrix.

4.5.1 Setting up Elasticsearch via docker

Setting up Elasticsearch (ES) via docker is straightforward. Simply run the following command:

```
docker run -d --name elasticsearch-for-rubrix -p 9200:9200 -p 9300:9300 -e "ES_JAVA_
↪OPTS=-Xms512m -Xmx512m" -e "discovery.type=single-node" docker.elastic.co/
↪elasticsearch/elasticsearch-oss:7.10.2
```

This will create an ES docker container named “*elasticsearch-for-rubrix*” that will run in the background.

To see the logs of the container, you can run:

```
docker logs elasticsearch-for-rubrix
```

Or you can stop/start the container via:

```
docker stop elasticsearch-for-rubrix
docker start elasticsearch-for-rubrix
```

Warning: Keep in mind, if you remove your container with

```
docker rm elasticsearch-for-rubrix
```

you will lose all your datasets in Rubrix!

For more details about the ES installation with docker, see their [official documentation](#). For MacOS and Windows, Elasticsearch also provides [homebrew formulae](#) and a [msi package](#), respectively. We recommend ES version 7.10 to work with Rubrix.

4.5.2 Server configurations

By default, the Rubrix server will look for your ES endpoint at `http://localhost:9200`. But you can customize this by setting the `ELASTICSEARCH` environment variable. Have a look at the list of available [environment variables](#) to further configure the Rubrix server.

Since the Rubrix server is built on fastapi, you can launch it using **uvicorn** directly:

```
uvicorn rubrix:app
```

Note: For Rubrix versions below 0.9 you can launch the server via

```
uvicorn rubrix.server.server:app
```

For more details about fastapi and uvicorn, see [here](#).

Fastapi also provides beautiful REST API docs that you can check at <http://localhost:6900/api/docs>.

Environment variables

You can set following environment variables to further configure your server and client.

Server

- `ELASTICSEARCH`: URL of the connection endpoint of the Elasticsearch instance (Default: `http://localhost:9200`).
- `RUBRIX_ELASTICSEARCH_SSL_VERIFY`: If “False”, disables SSL certificate verification when connection to the Elasticsearch backend.
- `RUBRIX_ELASTICSEARCH_CA_PATH`: Path to CA cert for ES host. For example: `/full/path/to/root-ca.pem` (Optional)
- `RUBRIX_NAMESPACE`: A prefix used to manage Elasticsearch indices. You can use this namespace to use the same Elasticsearch instance for several independent Rubrix instances.
- `RUBRIX_DEFAULT_ES_SEARCH_ANALYZER`: Default analyzer for textual fields excluding the metadata (Default: “standard”).
- `RUBRIX_EXACT_ES_SEARCH_ANALYZER`: Default analyzer for `*.exact` fields in textual information (Default: “whitespace”).
- `METADATA_FIELDS_LIMIT`: Max number of fields in the metadata (Default: 50, max: 100).

- `CORS_ORIGINS`: List of host patterns for CORS origin access.
- `DOCS_ENABLED`: If False, disables openapi docs endpoint at `/api/docs`.

Client

- `RUBRIX_API_URL`: The default API URL when calling `rubrix.init()`.
- `RUBRIX_API_KEY`: The default API key when calling `rubrix.init()`.
- `RUBRIX_WORKSPACE`: The default workspace when calling `rubrix.init()`.

4.5.3 Launching the web app via docker

You can use vanilla docker to run our image of the web app. First, pull the image from the [Docker Hub](#):

```
docker pull recognai/rubrix
```

Then simply run it. Keep in mind that you need a running Elasticsearch instance for Rubrix to work. By default, the Rubrix server will look for your Elasticsearch endpoint at `http://localhost:9200`. But you can customize this by setting the `ELASTICSEARCH` environment variable.

```
docker run -p 6900:6900 -e "ELASTICSEARCH=<your-elasticsearch-endpoint>" --name rubrix_
↪recognai/rubrix
```

To find running instances of the Rubrix server, you can list all the running containers on your machine:

```
docker ps
```

To stop the Rubrix server, just stop the container:

```
docker stop rubrix
```

If you want to deploy your own Elasticsearch cluster via docker, we refer you to the excellent guide on the [Elasticsearch homepage](#).

4.5.4 Launching the web app via docker-compose

For this method you first need to install [Docker Compose](#).

Then, create a folder:

```
mkdir rubrix && cd rubrix
```

and launch the docker-contained web app with the following command:

```
wget -O docker-compose.yml https://raw.githubusercontent.com/recognai/rubrix/master/
↪docker-compose.yml && docker-compose up -d
```

Warning: Latest versions of docker should be executed without the dash '-', e.g:

```
docker compose up -d
```

This is a convenient way because it automatically includes an [Elasticsearch](#) instance, Rubrix's main persistent layer.

Warning: Keep in mind, if you execute

```
docker-compose down
```

you will lose all your datasets in Rubrix!

Persisting Elasticsearch data

To avoid losing all the data when the docker-compose/server goes down, you can add some persistence by mounting a volume in the docker compose.

To this end, **modify the elasticsearch service and create a new volume** in the docker-compse.yml file:

```
services:
  elasticsearch:
    image: docker.elastic.co/elasticsearch/elasticsearch:7.11.1
    container_name: elasticsearch
    environment:
      - node.name=elasticsearch
      - cluster.name=es-rubrix-local
      - discovery.type=single-node
      - "ES_JAVA_OPTS=-Xms512m -Xmx512m"
    ulimits:
      memlock:
        soft: -1
        hard: -1
    networks:
      - rubrix
    # Add the volume to the elasticsearch service
    volumes:
      - elasticdata:/usr/share/elasticsearch/data
  rubrix:
    # ... here goes the rest of the docker-compose.yml

# ...

# At the end of the file create a volume for ElasticSearch
volumes:
  elasticdata:
```

Then, even if the ElasticSearch service goes down the data will be persisted in the elasticdata volume. To check it you can execute the command:

```
docker volume ls
```

Note that if you want to apply these changes, and you already have a previous docker-compose instance running, you need to execute the **up** command again:

```
docker-compose up -d
```

4.5.5 Configure Elasticsearch role/users

If you have an Elasticsearch instance and want to share resources with other applications, you can easily configure it for Rubrix.

All you need to take into account is:

- Rubrix will create its ES indices with the following pattern `.rubrix*`. It's recommended to create a new role (e.g., `rubrix`) and provide it with all privileges for this index pattern.
- Rubrix creates an index template for these indices, so you may provide related template privileges to this ES role.

Rubrix uses the `ELASTICSEARCH` environment variable to set the ES connection.

You can provide the credentials using the following scheme:

```
http(s)://user:passwd@elastichost
```

Below you can see a screenshot for setting up a new *rubrix* Role and its permissions:

The screenshot shows the Elasticsearch Kibana interface for configuring a new role. The role name is set to 'rubrix'. Below this, there are three main sections for configuring permissions:

- Cluster privileges:** A dropdown menu shows 'manage_index_templates'.
- Run As privileges:** A dropdown menu shows 'Add a user...'.
- Index privileges:** A dropdown menu shows '.rubrix*' for indices and 'all' for privileges. There are also checkboxes for 'Grant access to specific fields' and 'Grant read privileges to specific documents', both of which are currently unchecked.

At the bottom, there is a button labeled 'Add index privilege'.

Change elasticsearch index analyzers

By default, for indexing text fields, Rubrix uses the `standard` analyzer for general search and the `whitespace` analyzer for more exact queries (required by certain rules in the weak supervision module). If those analyzers don't fit your use case, you can change them using the following environment variables: `RUBRIX_DEFAULT_ES_SEARCH_ANALYZER` and `RUBRIX_EXACT_ES_SEARCH_ANALYZER`.

Note that provided analyzers names should be defined as built-in ones. If you want to use a customized analyzer, you should create it inside an index_template matching Rubrix index names (`.rubrix*.records-v0`), and then provide the analyzer name using the specific environment variable.

4.5.6 Deploy to aws instance using docker-machine

Setup an AWS profile

The aws command cli must be installed. Then, type:

```
aws configure --profile rubrix
```

and follow command instructions. For more details, visit [AWS official documentation](#).

Once the profile is created (a new entry should appear in file `~/.aws/config`), you can activate it via setting environment variable:

```
export AWS_PROFILE=rubrix
```

Create docker machine (aws)

```
docker-machine create --driver amazec2 \
--amazec2-root-size 60 \
--amazec2-instance-type t2.large \
--amazec2-open-port 80 \
--amazec2-ami ami-0b541372 \
--amazec2-region eu-west-1 \
rubrix-aws
```

Available ami depends on region. The provided ami is available for eu-west regions

Verify machine creation

```
$>docker-machine ls
```

NAME	ERRORS	ACTIVE	DRIVER	STATE	URL	SWARM
↪ DOCKER						
rubrix-aws		-	amazec2	Running	tcp://52.213.178.33:2376	
↪ v20.10.7						

Save assigned machine ip

In our case, the assigned ip is 52.213.178.33

Connect to remote docker machine

To enable the connection between the local docker client and the remote daemon, we must type following command:

```
eval $(docker-machine env rubrix-aws)
```

Define a docker-compose.yaml

```
# docker-compose.yaml
version: "3"

services:
  rubrix:
    image: recognai/rubrix:master
    ports:
      - "80:80"
    environment:
      ELASTICSEARCH: <elasticsearch-host_and_port>
    restart: unless-stopped
```

Pull image

```
docker-compose pull
```

Launch docker container

```
docker-compose up -d
```

Accessing Rubrix

In our case `http://52.213.178.33`

4.5.7 Install from master

If you want the cutting-edge version of *Rubrix* with the latest changes and experimental features, follow the steps below in your terminal. **Be aware that this version might be unstable!**

First, you need to install the master version of our python client:

```
pip install -U git+https://github.com/recognai/rubrix.git
```

Then, the easiest way to get the master version of our web app up and running is via docker-compose:

Note: For now, we only provide the master version of our web app via docker. If you want to run the web app of the master branch **without** docker, we refer you to our [development setup](#).

```
# get the docker-compose yaml file
mkdir rubrix && cd rubrix
wget -O docker-compose.yml https://raw.githubusercontent.com/recognai/rubrix/master/
↪ docker-compose.yml
# use the master image of the rubrix container instead of the latest
sed -i 's/rubrix:latest/rubrix:master/' docker-compose.yml
# start all services
docker-compose up
```

If you want to use vanilla docker (and have your own Elasticsearch instance running), you can just use our master image:

```
docker run -p 6900:6900 -e "ELASTICSEARCH=<your-elasticsearch-endpoint>" --name rubrix_
↪recognai/rubrix:master
```

4.6 Rubrix Cookbook

This guide is a collection of recipes. It shows examples for using Rubrix with some of the most popular NLP Python libraries.

Rubrix can be used with any library or framework inside your favourite IDE, be it VS Code, or Jupyter Lab.

With these examples, you'll be able to start exploring and annotating data with these libraries and get some inspiration if your library of choice is not in this guide.

If you miss a library in this guide that, leave a message in the [Rubrix Discussion forum](#) or open an issue or PR, we'll be very happy to receive contributions.

4.6.1 Hugging Face Transformers

[Hugging Face](#) has made working with NLP easier than ever before. With a few lines of code we can take a pretrained Transformer model from the [Hub](#), start making some predictions and log them into Rubrix.

```
[ ]: %pip install torch transformers datasets -qqq
```

Text Classification

For text and zeroshot classification pipelines, Rubrix's `rb.monitor` method makes it really easy to store data in Rubrix. Let's see some examples.

Zero-shot classification pipelines

Let's load a zero-shot-classification pipeline:

```
[ ]: import rubrix as rb
from transformers import pipeline

nlp = pipeline("zero-shot-classification", model="typeform/distilbert-base-uncased-mnli")
```

Let's use the `rb.monitor` method, which will asynchronously log our pipeline predictions. Now every time we predict with this pipeline the records will be logged in Rubrix. For example the code:

```
[ ]: # sample rate = 1 means we'll be logging every prediction
# for monitoring production models a lower rate might be preferable
nlp = rb.monitor(nlp, dataset="zeroshot_example", sample_rate=1)
nlp("this is a test", candidate_labels=['World', 'Sports', 'Business', 'Sci/Tech'])
```

It will create the following dataset:

zeroshot_example

name: typeform/distilbert-base-uncased-mnli

transformers_version: 4.12.3

model_type: distilbert

task: zero-shot-classification

TextClassification

29 minutes ago

10 minutes ago

which contains the following record:

rubrix

zeroshot_example

Text Classification records (2)

Search records

Predictions

Status

Metadata

Sort

TEXT:

this is a test

Sci/Tech 56.50%

World 21.38%

Business 12.58%

Sports 9.54%

Now if we want to log a larger dataset we can use the batch prediction method from pipelines in a similar way. Let's load a dataset from the Hugging Face Hub and use the `dataset.map` method to parallelize the inference. The following will log the predictions for the first 20 records in the `ag_news` test dataset. You can use the same idea for any custom dataset, using `pandas.read_csv` for example.

```
[ ]: from datasets import load_dataset

dataset = load_dataset("ag_news", split="test[0:20]")

dataset.map(
    lambda examples: {"predictions": nlp(examples["text"], candidate_labels=['World',
↪ 'Sports', 'Business', 'Sci/Tech'])},
    batch_size=5,
    batches=True
)
```

Text classification pipelines

For text classification pipelines it will work in the same way as above. Let's see an example, this time using `pandas`.

Let's read a dataset with tweets:

```
[2]: import pandas as pd

# a url to a dataset containing tweets
url = "https://raw.githubusercontent.com/ajayshevale/Sentiment-Analysis-of-Text-Data-
↪ Tweets-/master/data/test.csv"
df = pd.read_csv(url)
df.head()

[2]:
```

	Id	Category
0	6.289494e+17	dear @Microsoft the newOffice for Mac is grea...
1	6.289766e+17	@Microsoft how about you make a system that do...

(continues on next page)

(continued from previous page)


```
2 6.290232e+17 Not Available
3 6.291792e+17 Not Available
4 6.291863e+17 If I make a game as a #windows10 Universal App...
```



And use a sentiment analysis pipeline with the `rb.monitor` method:

```
[ ]: nlp = pipeline("sentiment-analysis")
nlp = rb.monitor(nlp, dataset="text_classification_example", sample_rate=1)

for i, example in df.iterrows():
    nlp(example.Category)
```

which will create the following dataset:

text_classification_example 

name: distilbert-base-uncased-finetuned-sst-2-english	TextClassification	2 minutes ago	a few seconds ago		
transformers_version: 4.12.3					
model_type: distilbert	task: sentiment-analysis				

Training

The above examples have shown how to store data in Rubrix, using pre-trained models. You can use Rubrix for storing datasets without predictions and without annotations, or a combination of both annotations and predictions.

One of the main features of Rubrix is data annotation, which lets you rapidly create training sets. In this example, let's see how we can take labelled dataset from Rubrix to fine-tune a Hugging Face transformers text classifier.

Let's read a Rubrix dataset, prepare a training set and use the Trainer API for fine-tuning a `distilbert-base-uncased` model.

Take into account that a `zeroshot_example` contains annotations. You can go to this dataset (if you have run the previous example) and do some manual annotation using the Annotation mode.

```
[8]: from datasets import Dataset
import rubrix as rb

# load rubrix dataset
dataset_rb = rb.load('zeroshot_example')

# create dataset with text and labels as numeric ids
train_ds = dataset_rb.prepare_for_training()

[ ]: from transformers import AutoModelForSequenceClassification
from transformers import AutoTokenizer
from transformers import Trainer

# from here, it's just regular fine-tuning with transformers
tokenizer = AutoTokenizer.from_pretrained("distilbert-base-uncased")
model = AutoModelForSequenceClassification.from_pretrained("distilbert-base-uncased",
    num_labels=4)
```

(continues on next page)

(continued from previous page)

```
def tokenize_function(examples):
    return tokenizer(examples=["text"], padding="max_length", truncation=True)

train_dataset = train_ds.map(tokenize_function, batched=True).shuffle(seed=42)

trainer = Trainer(model=model, train_dataset=train_dataset)

trainer.train()
```

Token Classification

We will explore a DistilBERT NER classifier fine-tuned for NER using the conll03 English dataset.

```
[ ]: import rubrix as rb
      from transformers import pipeline

      input_text = "My name is Sarah and I live in London"

      # We define our HuggingFace Pipeline
      classifier = pipeline(
          "ner",
          model="elastic/distilbert-base-cased-finetuned-conll03-english",
      )

      # Making the prediction
      predictions = classifier(input_text, aggregation_strategy="first")

      # Creating the predicted entities as a list of tuples (entity, start_char, end_char)
      prediction = [(pred["entity_group"], pred["start"], pred["end"]) for pred in predictions]

      # Create word tokens
      batch_encoding = classifier.tokenizer(input_text)
      word_ids = sorted(set(batch_encoding.word_ids()) - {None})
      words = []
      for word_id in word_ids:
          char_span = batch_encoding.word_to_chars(word_id)
          words.append(input_text[char_span.start:char_span.end])

      # Building a TokenClassificationRecord
      record = rb.TokenClassificationRecord(
          text=input_text,
          token=words,
          prediction=prediction,
          prediction_agent="distilbert-base-cased-finetuned-conll03-english",
      )

      # Logging into Rubrix
      rb.log(records=record, name="zeroshot-ner")
```

Fine-tuning with custom annotations

Let's see how we can use this model to pre-annotate a custom dataset and fine-tune a small transformer model. As custom dataset, we will take the well known **AG news** dataset that is normally used for benchmarking text classification models.

```
[ ]: from datasets import load_dataset
      from transformers import pipeline

      import rubrix as rb
      from tqdm.auto import tqdm

      # Our custom dataset will consist of the first 'n' examples of the AG news dataset.
      n = 1000
      ag_news = load_dataset("ag_news", split="train", streaming=True)

      # Our pre-annotations will come from a fine-tuned distilbert model
      classifier = pipeline("ner", "elastic/distilbert-base-cased-finetuned-conll03-english")

      # For our annotation process we want word tokens, NOT subword tokens
      def make_tokens(examples):
          batch_encoding = classifier.tokenizer(examples["text"])
          examples["tokens"] = []
          for text, encoding in zip(examples["text"], batch_encoding.encodings):
              word_ids = sorted(set(encoding.word_ids) - {None})
              words = []
              for word_id in word_ids:
                  start, end = encoding.word_to_chars(word_id)
                  words.append(text[start:end])
              examples["tokens"].append(words)
          return examples

      # Get the prediction from our fine-tuned distilbert
      def make_predictions(examples):
          examples["prediction"] = classifier(examples["text"], aggregation_strategy="first")
          return examples

      # Add tokens and predictions
      ag_news_prepared = ag_news.take(n)\
          .map(make_tokens, batched=True)\
          .map(make_predictions, batched=True, batch_size=32)

      # Make Rubrix records
      records = []
      for idx, example in tqdm(enumerate(ag_news_prepared), total=n, desc="Making records"):
          record = rb.TokenClassificationRecord(
              text=example["text"],
              tokens=example["tokens"],
              prediction=[(p["entity_group"], p["start"], p["end"]) for p in example[
→ "prediction"]],
              prediction_agent="elastic/distilbert-base-cased-finetuned-conll03-english",
              id=idx,
```

(continues on next page)

(continued from previous page)

```

    )
    records.append(record)

# Upload records to Rubrix
rb.log(records, "ag_news_ner")

```

After adding the records to Rubrix, we can now annotate them in the web app with the help of the predictions. Afterward, we load them back into the notebook and fine-tune our model. We chose the very efficient and light-weight **ELECTRA** model to illustrate the procedure.

Following training steps are a copy&paste of the excellent [Hugging Face How-to guides](#). Check them for more information about the training process.

```

[ ]: from transformers import AutoTokenizer, DataCollatorForTokenClassification

# Load the dataset from the web app and prepare it for training a Hugging Face
↳ transformer
agnews_ds = rb.load("ag_news_ner").prepare_for_training()

# Split it into a train and test set
agnews = agnews_ds.train_test_split()

# We need to properly tokenize our data and align our labels
tokenizer = AutoTokenizer.from_pretrained("google/electra-small-discriminator")
data_collator = DataCollatorForTokenClassification(tokenizer=tokenizer)

def tokenize_and_align_labels(examples):
    tokenized_inputs = tokenizer(examples["tokens"], truncation=True, is_split_into_
↳ words=True)

    labels = []
    for i, label in enumerate(examples["ner_tags"]):
        word_ids = tokenized_inputs.word_ids(batch_index=i) # Map tokens to their
↳ respective word.
        previous_word_idx = None
        label_ids = []
        for word_idx in word_ids: # Set the special tokens to -100.
            if word_idx is None:
                label_ids.append(-100)
            elif word_idx != previous_word_idx: # Only label the first token of a given
↳ word.
                label_ids.append(label[word_idx])
            else:
                label_ids.append(-100)
            previous_word_idx = word_idx
        labels.append(label_ids)

    tokenized_inputs["labels"] = labels
    return tokenized_inputs

tokenized_agnews = agnews.map(tokenize_and_align_labels, batched=True)

```



```
[ ]: from transformers import AutoModelForTokenClassification, TrainingArguments, Trainer

# Load the pre-trained transformer and provide the dimensions of your token_
↳classification head
model = AutoModelForTokenClassification.from_pretrained(
    "google/electra-small-discriminator",
    num_labels=len(agnews_ds.features["ner_tags"][0].names)
)

# Define your training arguments
training_args = TrainingArguments(
    output_dir="./results",
    evaluation_strategy="epoch",
    learning_rate=2e-5,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    num_train_epochs=3,
    weight_decay=0.01,
)

# Instantiate the trainer
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=tokenized_agnews["train"],
    eval_dataset=tokenized_agnews["test"],
    tokenizer=tokenizer,
    data_collator=data_collator,
)

# Train the model
trainer.train()
```

4.6.2 spaCy

spaCy offers industrial-strength Natural Language Processing, with support for 64+ languages, trained pipelines, multi-task learning with pretrained Transformers, pretrained word vectors and much more.

```
[ ]: %pip install spacy
```

Token Classification

We will focus our spaCy recipes into Token Classification tasks, showing you how to log data from NER and POS tagging.

NER

For this recipe, we are going to try the French language model to extract NER entities from some sentences.

```
[ ]: !python -m spacy download fr_core_news_sm

[ ]: import rubrix as rb
import spacy

input_text = "Paris a un enfant et la for^et a un oiseau ; l'oiseau s'appelle le moineau_
↔; l'enfant s'appelle le gamin"

# Loading spaCy model
nlp = spacy.load("fr_core_news_sm")

# Creating spaCy doc
doc = nlp(input_text)

# Creating the prediction entity as a list of tuples (entity, start_char, end_char)
prediction = [(ent.label, ent.start_char, ent.end_char) for ent in doc.ents]

# Building TokenClassificationRecord
record = rb.TokenClassificationRecord(
    text=input_text,
    token=[token.text for token in doc],
    prediction=prediction,
    prediction_agent="spacy.fr_core_news_sm",
)

# Logging into Rubrix
rb.log(records=record, name="lesmiserables-ner")
```

POS tagging

Changing very few parameters, we can make a POS tagging experiment, instead of NER. Let's try it out with the same input sentence.

```
[ ]: import rubrix as rb
import spacy

input_text = "Paris a un enfant et la for^et a un oiseau ; l'oiseau s'appelle le moineau_
↔; l'enfant s'appelle le gamin"

# Loading spaCy model
nlp = spacy.load("fr_core_news_sm")
```

(continues on next page)

(continued from previous page)

```
# Creating spaCy doc
doc = nlp(input_text)

# Creating the prediction entity as a list of tuples (tag, start_char, end_char)
prediction = [(token.pos_, token.idx, token.idx + len(token)) for token in doc]

# Building TokenClassificationRecord
record = rb.TokenClassificationRecord(
    text=input_text,
    token=[token.text for token in doc],
    prediction=prediction,
    prediction_agent="spacy.fr_core_news_sm",
)

# Logging into Rubrix
rb.log(records=record, name="lesmiserables-pos")
```

Train a spaCy model by exporting to Docbin

With the examples above, we have been able to store from spaCy models into Rubrix.

In order to train models with spaCy, Rubrix provides you with an easy util to prepare a dataset `spaCy Docbin` format. With the example below, you can export your Rubrix dataset into a Docbin, save it to disk, and then use this file with the `spacy train` command.

```
[ ]: import spacy
import rubrix as rb

from datasets import load_dataset

# Dataset loading to export it as a Docbin
dataset_raw = load_dataset("conll2003", split="train")
dataset_rubrix = rb.DatasetForTokenClassification.from_datasets(dataset_raw, tag="ner_
→tags")

# Loading an spaCy blank language model to create the Docbin, as it works faster
nlp = spacy.blank("en")

# After this line, the file will be stored in disk
dataset_rubrix.prepare_for_training(framework="spacy", lang=nlp).to_disk("train.spacy")
```

4.6.3 Flair

It's a framework that provides a state-of-the-art NLP library, a text embedding library and a PyTorch framework for NLP. `Flair` offers sequence tagging language models in English, Spanish, Dutch, German and many more, and they are also hosted on [HuggingFace Model Hub](#).

```
[ ]: %pip install flair
```

Token Classification (NER)

Inference

```
[ ]: import rubrix as rb

from flair.data import Sentence
from flair.models import SequenceTagger

# load tagger
tagger = rb.monitor(SequenceTagger.load("flair/ner-english"), dataset="flair-example",
↳ sample_rate=1.0)

# make example sentence
sentence = Sentence("George Washington went to Washington")

# predict NER tags. This will log the prediction in Rubrix
tagger.predict(sentence)
```

Training

Let's read a Rubrix dataset, prepare a training set, save to .txt for loading with flair ColumnCorpus and train with flair SequenceTagger

```
[ ]: import pandas as pd
from difflib import SequenceMatcher

from flair.data import Corpus
from flair.datasets import ColumnCorpus
from flair.embeddings import WordEmbeddings, FlairEmbeddings, StackedEmbeddings
from flair.models import SequenceTagger
from flair.trainers import ModelTrainer

import rubrix as rb

# 1. Load the dataset from Rubrix (your own NER/token classification task)
# Note: we initiate the 'tars_ner_wnut_17' from " Zero-shot Named Entity Recognition_
↳ with Flair" tutorial
# (reference: https://rubrix.readthedocs.io/en/stable/tutorials/08-zeroshot\_ner.html)
train_dataset = rb.load("tars_ner_wnut_17").to_pandas()
```

```
[ ]: # 2. Pre-processing to BIO scheme before saving as .txt file

# Use original predictions as annotations for demonstration purposes, in a real use case_
↳ you would use the `annotations` instead
prediction_list = train_dataset.prediction
text_list = train_dataset.text

annotation_list = []
```

(continues on next page)

(continued from previous page)

```

idx = 0
for ner_list in prediction_list:
    new_ner_list = []
    for val in ner_list:
        new_ner_list.append((text_list[idx][val[1]:val[2]], val[0]))
    annotation_list.append(new_ner_list)
    idx += 1

ready_data = pd.DataFrame()
ready_data['text'] = text_list
ready_data['annotation'] = annotation_list

def matcher(string, pattern):
    """
    Return the start and end index of any pattern present in the text.
    """
    match_list = []
    pattern = pattern.strip()
    seqMatch = SequenceMatcher(None, string, pattern, autojunk=False)
    match = seqMatch.find_longest_match(0, len(string), 0, len(pattern))
    if (match.size == len(pattern)):
        start = match.a
        end = match.a + match.size
        match_tup = (start, end)
        string = string.replace(pattern, "X" * len(pattern), 1)
        match_list.append(match_tup)
    return match_list, string

def mark_sentence(s, match_list):
    """
    Marks all the entities in the sentence as per the BIO scheme.
    """
    word_dict = {}
    for word in s.split():
        word_dict[word] = 'O'
    for start, end, e_type in match_list:
        temp_str = s[start:end]
        tmp_list = temp_str.split()
        if len(tmp_list) > 1:
            word_dict[tmp_list[0]] = 'B-' + e_type
            for w in tmp_list[1:]:
                word_dict[w] = 'I-' + e_type
        else:
            word_dict[temp_str] = 'B-' + e_type
    return word_dict

def create_data(df, filepath):
    """

```

(continues on next page)

(continued from previous page)

The function responsible for the creation of data in the said format.

```
"""
with open(filepath, 'w') as f:
    for text, annotation in zip(df.text, df.annotation):
        text_ = text
        match_list = []
        for i in annotation:
            a, text_ = matcher(text, i[0])
            match_list.append((i[0][0], a[0][1], i[1]))
        d = mark_sentence(text, match_list)
        for i in d.keys():
            f.writelines(i + ' ' + d[i] + '\n')
        f.writelines('d')

# path to save the txt file.
filepath = 'train.txt'

# creating the file.
create_data(ready_data, filepath)
```

```
[ ]: # 3. Load to Flair ColumnCorpus
# define columns
columns = {0: 'text', 1: 'ner'}

# directory where the data resides
data_folder = './'

# initializing the corpus
corpus: Corpus = ColumnCorpus(data_folder, columns,
                              train_file='train.txt',
                              test_file=None,
                              dev_file=None)

# 4. Define training parameters

# tag to predict
label_type = 'ner'

# make tag dictionary from the corpus
label_dict = corpus.make_label_dictionary(label_type=label_type)

# initialize embeddings
embedding_types = [
    WordEmbeddings('glove'),
    FlairEmbeddings('news-forward'),
    FlairEmbeddings('news-backward'),
]

embeddings: StackedEmbeddings = StackedEmbeddings(
```

(continues on next page)

(continued from previous page)

```

        embeddings=embedding_types)

# 5. initialize sequence tagger
tagger = SequenceTagger(hidden_size=256,
                        embedding=embeddings,
                        tag_dictionary=label_dict,
                        tag_type=label_type,
                        use_crf=True)

# 6. initialize trainer
trainer = ModelTrainer(tagger, corpus)

# 7. start training
trainer.train('token-classification',
             learning_rate=0.1,
             mini_batch_size=32,
             max_epochs=15)

```

Text Classification

Training

Let's read a Rubrix dataset, prepare a training set, save to .csv for loading with flair CSVClassificationCorpus and train with flair ModelTrainer

```

[ ]: import pandas as pd
import torch
from torch.optim.lr_scheduler import OneCycleLR

from flair.datasets import CSVClassificationCorpus
from flair.embeddings import TransformerDocumentEmbeddings
from flair.models import TextClassifier
from flair.trainers import ModelTrainer

import rubrix as rb

# 1. Load the dataset from Rubrix
limit_num = 2048
train_dataset = rb.load("tweet_eval_emojis", limit=limit_num).to_pandas()

# 2. Pre-processing training pandas dataframe
train_df = pd.DataFrame()
train_df['text'] = train_dataset['text']
train_df['label'] = train_dataset['annotation']

# 3. Save as csv with tab delimiter
train_df.to_csv('train.csv', sep='|')

```

```
[ ]: # 4. Read the with CSVClassificationCorpus
data_folder = './'

# column format indicating which columns hold the text and label(s)
label_type = "label"
column_name_map = {1: "text", 2: "label"}

corpus = CSVClassificationCorpus(
    data_folder, column_name_map, skip_header=True, delimiter='\\t', label_type=label_
    ↪type)

# 5. create the label dictionary
label_dict = corpus.make_label_dictionary(label_type=label_type)

# 6. initialize transformer document embeddings (many models are available)
document_embeddings = TransformerDocumentEmbeddings(
    'distilbert-base-uncased', fine_tune=True)

# 7. create the text classifier
classifier = TextClassifier(
    document_embeddings, label_dictionary=label_dict, label_type=label_type)

# 8. initialize trainer with AdamW optimizer
trainer = ModelTrainer(classifier, corpus, optimizer=torch.optim.AdamW)

# 9. run training with fine-tuning
trainer.train('./emojis-classification',
    learning_rate=5.0e-5,
    mini_batch_size=4,
    max_epochs=4,
    scheduler=OneCycleLR,
    embeddings_storage_mode='none',
    weight_decay=0.,
    )
```

Let's make a prediction with flair TextClassifier

```
[ ]: from flair.data import Sentence
from flair.models import TextClassifier

classifier = TextClassifier.load('./emojis-classification/best-model.pt')

# create example sentence
sentence = Sentence('Farewell, Charleston! The memories are sweet #mimosa #dontwannago @_
    ↪Virginia on King')

# predict class and print
classifier.predict(sentence)

print(sentence.labels)
```


Zero-shot and Few-shot classifiers

Flair enables you to use few-shot and zero-shot learning for text classification with Task-aware representation of sentences (TARS), introduced by Halder et al. (2020), see [Flair's documentation](#) for more details.

Let's see an example of the base zero-shot TARS model:

```
[ ]: import rubrix as rb
      from flair.models import TARSClassifier
      from flair.data import Sentence

      # Load our pre-trained TARS model for English
      tars = TARSClassifier.load('tars-base')

      # Define labels
      labels = ["happy", "sad"]

      # Create a sentence
      text = "I am so glad you liked it!"
      sentence = Sentence(text)

      # Predict for these labels
      tars.predict_zero_shot(sentence, labels)

      # Creating the prediction entity as a list of tuples (label, probability)
      prediction = [(pred.value, pred.score) for pred in sentence.labels]

      # Building a TextClassificationRecord
      record = rb.TextClassificationRecord(
          text=text,
          prediction=prediction,
          prediction_agent="tars-base",
      )

      # Logging into Rubrix
      rb.log(records=record, name="en-emotion-zeroshot")
```

Custom and pre-trained classifiers

Let's see an example with the German offensive language classifier

```
[ ]: import rubrix as rb
      from flair.models import TextClassifier
      from flair.data import Sentence

      text = "Du erzählst immer Quatsch."

      # Load our pre-trained classifier
      classifier = TextClassifier.load("de-offensive-language")

      # Creating Sentence object
```

(continues on next page)

(continued from previous page)

```

sentence = Sentence(text)

# Make the prediction
classifier.predict(sentence, return_probabilities_for_all_classes=True)

# Creating the prediction entity as a list of tuples (label, probability)
prediction = [(pred.value, pred.score) for pred in sentence.label_]

# Building a TextClassificationRecord
record = rb.TextClassificationRecord(
    text=text,
    prediction=prediction,
    prediction_agent="de-offensive-language",
)

# Logging into Rubrix
rb.log(records=record, name="german-offensive-language")

```

POS tagging

In the following snippet we will use the multilingual POS tagging model from Flair.

```

[ ]: import rubrix as rb
      from flair.data import Sentence
      from flair.models import SequenceTagger

      input_text = "George Washington went to Washington. Dort kaufte er einen Hut."

      # Loading our POS tagging model from flair
      tagger = SequenceTagger.load("flair/upos-multi")

      # Creating Sentence object
      sentence = Sentence(input_text)

      # run NER over sentence
      tagger.predict(sentence)

      # Creating the prediction entity as a list of tuples (entity, start_char, end_char)
      prediction = [
          (entity.get_labels()[0].value, entity.start_pos, entity.end_pos)
          for entity in sentence.get_spans()
      ]

      # Building a TokenClassificationRecord
      record = rb.TokenClassificationRecord(
          text=input_text,
          tokens=[token.text for token in sentence],
          prediction=prediction,
          prediction_agent="flair/upos-multi",
      )

```

(continues on next page)

(continued from previous page)

```
# Logging into Rubrix
rb.log(records=record, name="flair-pos-tagging")
```

4.6.4 Stanza

Stanza is a collection of efficient tools for many NLP tasks and processes, all in one library. It's maintained by the Stanford NLP Group. We are going to take a look at a few interactions that can be done with Rubrix.

```
[ ]: %pip install stanza
```

Text Classification

Let's start by using a Sentiment Analysis model to log some TextClassificationRecords.

```
[ ]: import rubrix as rb
import stanza

text = (
    "There are so many NLP libraries available, I don't know which one to choose!"
)

# Downloading our model, in case we don't have it cached
stanza.download("en")

# Creating the pipeline
nlp = stanza.Pipeline(lang="en", processors="tokenize,sentiment")

# Analyzing the input text
doc = nlp(text)

# This model returns 0 for negative, 1 for neutral and 2 for positive outcome.
# We are going to log them into Rubrix using a dictionary to translate numbers to labels.
num_to_labels = {0: "negative", 1: "neutral", 2: "positive"}

# Build a prediction entities list
# Stanza, at the moment, only output the most likely label without probability.
# So we will suppose Stanza predicts the most likely label with 1.0 probability, and
# the rest with 0.
entities = []

for _, sentence in enumerate(doc.sentences):
    for key in num_to_labels:
        if key == sentence.sentiment:
            entities.append((num_to_labels[key], 1))
        else:
            entities.append((num_to_labels[key], 0))

# Building a TextClassificationRecord
record = rb.TextClassificationRecord(
```

(continues on next page)

(continued from previous page)

```

    text=text,
    prediction=entities,
    prediction_agent="stanza/en",
)

# Logging into Rubrix
rb.log(records=record, name="stanza-sentiment")

```

Token Classification

Stanza offers so many different pretrained language models for Token Classification Tasks, and the list does not stop growing.

POS tagging

We can use one of the many UD models, used for POS tags, morphological features and syntactic relations. UD stands for [Universal Dependencies](#), the framework where these models has been trained. For this example, let's try to extract POS tags of some Catalan lyrics.

```

[ ]: import rubrix as rb
import stanza

# Loading a cool Obrint Pas lyric
input_text = "Viure sempre corrent, avançant amb la gent, rellevant contra el vent,
↳transportant sentiments."

# Downloading our model, in case we don't have it cached
stanza.download("ca")

# Creating the pipeline
nlp = stanza.Pipeline(lang="ca", processors="tokenize,mwt,pos")

# Analyzing the input text
doc = nlp(input_text)

# Creating the prediction entity as a list of tuples (tag, start_char, end_char)
prediction = [
    (word.pos, token.start_char, token.end_char)
    for sent in doc.sentences
    for token in sent.tokens
    for word in token.words
]

# Building a TokenClassificationRecord
record = rb.TokenClassificationRecord(
    text=input_text,
    token=[word.text for sent in doc.sentences for word in sent.words],
    prediction=prediction,
    prediction_agent="stanza/catalan",
)

```

(continues on next page)

(continued from previous page)

```
# Logging into Rubrix
rb.log(records=record, name="stanza-catalan-pos")
```

NER

Stanza also offers a list of available pretrained models for NER tasks. So, let's try Russian

```
[ ]: import rubrix as rb
import stanza

input_text = (
    "-- -- " # War and Peace is one my favourite books
)

# Downloading our model, in case we don't have it cached
stanza.download("ru")

# Creating the pipeline
nlp = stanza.Pipeline(lang="ru", processors="tokenize,ner")

# Analyzing the input text
doc = nlp(input_text)

# Creating the prediction entity as a list of tuples (entity, start_char, end_char)
prediction = [
    (token.ner, token.start_char, token.end_char)
    for sent in doc.sentences
    for token in sent.tokens
]

# Building a TokenClassificationRecord
record = rb.TokenClassificationRecord(
    text=input_text,
    token=[word.text for sent in doc.sentences for word in sent.words],
    prediction=prediction,
    prediction_agent="flair/russian",
)

# Logging into Rubrix
rb.log(records=record, name="stanza-russian-ner")
```

4.7 Tasks Templates

Hi there! In this article we wanted to share some examples of our supported tasks, so you can go from zero to hero as fast as possible. We are going to cover those tasks present in our supported tasks list, so don't forget to stop by and take a look.

The tasks are divided into their different category, from text classification to token classification. We will update this article, as well as the supported task list when a new task gets added to Rubrix.

4.7.1 Text Classification

Text classification deals with predicting in which categories a text fits. As if you're shown an image you could quickly tell if there's a dog or a cat in it, we build NLP models to distinguish between a Jane Austen's novel or a Charlotte Bronte's poem. It's all about feeding models with labelled examples and seeing how they start predicting over the very same labels.

Text Categorization

This is a general example of the Text Classification family of tasks. Here, we will try to assign pre-defined categories to sentences and texts. The possibilities are endless! Topic categorization, spam detection, and a vast etcétera.

For our example, we are using the [SqueezeBERT](#) zero-shot classifier for predicting the topic of a given text, in three different labels: politics, sports and technology. We are also using [AG](#), a collection of news, as our dataset.

```
[ ]: import rubrix as rb
      from transformers import pipeline
      from datasets import load_dataset

      # Loading our dataset
      dataset = load_dataset("ag_news", split="train[0:20]")

      # Define our HuggingFace Pipeline
      classifier = pipeline(
          "zero-shot-classification",
          model="typeform/squeezebert-mnli",
          framework="pt",
      )

      records = []

      for record in dataset:

          # Making the prediction
          prediction = classifier(
              record["text"],
              candidate_labels=[
                  "politics",
                  "sports",
                  "technology",
              ],
          )
```

(continues on next page)

(continued from previous page)

```

# Creating the prediction entity as a list of tuples (label, probability)
prediction = list(zip(prediction["labels"], prediction["scores"]))

# Appending to the record list
records.append(
    rb.TextClassificationRecord(
        text=record["text"],
        prediction=prediction,
        prediction_agent="https://huggingface.co/typeform/squeezebert-mnli",
        metadata={"split": "train"},
    )
)

# Logging into Rubrix
rb.log(
    record=records,
    name="text-categorization",
    tags={
        "task": "text-categorization",
        "phase": "data-analysis",
        "family": "text-classification",
        "dataset": "ag_news",
    },
)

```

Sentiment Analysis

In this kind of project, we want our models to be able to detect the polarity of the input. Categories like *positive*, *negative* or *neutral* are often used.

For this example, we are going to use an [Amazon review polarity dataset](#), and a sentiment analysis [roBERTa model](#), which returns LABEL 0 for positive, LABEL 1 for neutral and LABEL 2 for negative. We will handle that in the code.

```

[ ]: import rubrix as rb
      from transformers import pipeline
      from datasets import load_dataset

# Loading our dataset
dataset = load_dataset("amazon_polarity", split="train[0:20]")

# Define our HuggingFace Pipeline
classifier = pipeline(
    "text-classification",
    model="cardiffnlp/twitter-roberta-base-sentiment",
    framework="pt",
    return_all_scores=True,
)

# Make a dictionary to translate labels to a friendly-language
translate_labels = {
    "LABEL_0": "positive",

```

(continues on next page)

(continued from previous page)

```
"LABEL_1": "neutral",
"LABEL_2": "negative",
}

records = []

for record in dataset:

    # Making the prediction
    predictions = classifier(
        record["content"],
    )

    # Creating the prediction entity as a list of tuples (label, probability)
    prediction = [
        (translate_labels[prediction["label"]], prediction["score"])
        for prediction in predictions[0]
    ]

    # Appending to the record list
    records.append(
        rb.TextClassificationRecord(
            text=record["content"],
            prediction=prediction,
            prediction_agent="https://huggingface.co/cardiffnlp/twitter-roberta-base-
↪sentiment",
            metadata={"split": "train"},
        )
    )

# Logging into Rubrix
rb.log(
    record=records,
    name="sentiment-analysis",
    tags={
        "task": "sentiment-analysis",
        "phase": "data-annotation",
        "family": "text-classification",
        "dataset": "amazon-polarity",
    },
)
```


Semantic Textual Similarity

This task is all about how close or far a given text is from any other. We want models that output a value of closeness between two inputs.

For our example, we will be using [MRPC dataset](#), a corpus consisting of 5,801 sentence pairs collected from newswire articles. These pairs could (or could not) be paraphrases. Our model will be a [sentence Transformer](#), trained specifically for this task.

As HuggingFace Transformers does not support natively this task, we will be using the [Sentence Transformer](#) framework. For more information about how to make these predictions with HuggingFace Transformer, please visit this [link](#).

```
[ ]: import rubrix as rb
from sentence_transformers import SentenceTransformer, util
from datasets import load_dataset

# Loading our dataset
dataset = load_dataset("glue", "mrpc", split="train[0:20]")

# Loading the model
model = SentenceTransformer("paraphrase-MiniLM-L6-v2")

records = []

for record in dataset:

    # Creating a sentence list
    sentences = [record["sentence1"], record["sentence2"]]

    # Obtaining similarity
    paraphrases = util.paraphrase_mining(model, sentences)

    for paraphrase in paraphrases:
        score, _, _ = paraphrase

    # Building up the prediction tuples
    prediction = [("similar", score), ("not similar", 1 - score)]

    # Appending to the record list
    records.append(
        rb.TextClassificationRecord(
            input={
                "sentence 1": record["sentence1"],
                "sentence 2": record["sentence2"],
            },
            prediction=prediction,
            prediction_agent="https://huggingface.co/sentence-transformers/paraphrase-
↳ MiniLM-L12-v2",
            metadata={"split": "train"},
        )
    )
```

(continues on next page)

(continued from previous page)

```
# Logging into Rubrix
rb.log(
    record=records,
    name="semantic-textual-similarity",
    tags={
        "task": "similarity",
        "type": "paraphrasing",
        "family": "text-classification",
        "dataset": "mrpc",
    },
)
```

Natural Language Inference

Natural language inference is the task of determining whether a hypothesis is true (which will mean entailment), false (contradiction), or undetermined (neutral) given a premise. This task also works with pair of sentences.

Our dataset will be the famous [SNLI](#), a collection of 570k human-written English sentence pairs; and our model will be a zero-shot, cross encoder for inference.

```
[ ]: import rubrix as rb
from transformers import pipeline
from datasets import load_dataset

# Loading our dataset
dataset = load_dataset("snli", split="train[0:20]")

# Define our HuggingFace Pipeline
classifier = pipeline(
    "zero-shot-classification",
    model="cross-encoder/nli-MiniLM2-L6-H768",
    framework="pt",
)

records = []

for record in dataset:

    # Making the prediction
    prediction = classifier(
        record["premise"] + record["hypothesis"],
        candidate_labels=[
            "entailment",
            "contradiction",
            "neutral",
        ],
    )

    # Creating the prediction entity as a list of tuples (label, probability)
    prediction = list(zip(prediction["labels"], prediction["scores"]))
```

(continues on next page)

(continued from previous page)

```

# Appending to the record list
records.append(
    rb.TextClassificationRecord(
        input={"premise": record["premise"], "hypothesis": record["hypothesis"]},
        prediction=prediction,
        prediction_agent="https://huggingface.co/cross-encoder/nli-MiniLM2-L6-H768",
        metadata={"split": "train"},
    )
)

# Logging into Rubrix
rb.log(
    record=records,
    name="natural-language-inference",
    tags={
        "task": "nli",
        "family": "text-classification",
        "dataset": "snli",
    },
)

```

Stance Detection

Stance detection is the NLP task which seeks to extract from a subject's reaction to a claim made by a primary actor. It is a core part of a set of approaches to fake news assessment. For example:

- **Source:** *"Apples are the most delicious fruit in existence"*
- **Reply:** *"Obviously not, because that is a reuben from Katz's"*
- **Stance:** deny

But it can be done in many different ways. In the search of fake news, there is usually one source of text.

We will be using the [LIAR dataset](#), a fake news detection dataset with 12.8K human labeled short statements from politifact.com's API, and each statement is evaluated by a politifact.com editor for its truthfulness, and a zero-shot distilbart model.

```

[ ]: import rubrix as rb
      from transformers import pipeline
      from datasets import load_dataset

# Loading our dataset
dataset = load_dataset("liar", split="train[0:20]")

# Define our HuggingFace Pipeline
classifier = pipeline(
    "zero-shot-classification",
    model="valhalla/distilbart-mnli-12-3",
    framework="pt",
)

```

(continues on next page)

(continued from previous page)

```

records = []

for record in dataset:

    # Making the prediction
    prediction = Classifier(
        record["statement"],
        candidate_labels=[
            "false",
            "half-true",
            "mostly-true",
            "true",
            "barely-true",
            "pants-fire",
        ],
    )

    # Creating the prediction entity as a list of tuples (label, probability)
    prediction = list(zip(prediction["labels"], prediction["scores"]))

    # Appending to the record list
    records.append(
        rb.TextClassificationRecord(
            text=record["statement"],
            prediction=prediction,
            prediction_agent="https://huggingface.co/typeform/squeezebert-mnli",
            metadata={"split": "train"},
        )
    )

# Logging into Rubrix
rb.log(
    record=records,
    name="stance-detection",
    tags={
        "task": "stance detection",
        "family": "text-classification",
        "dataset": "liar",
    },
)

```

Multilabel Text Classification

A variation of the text classification basic problem, in this task we want to categorize a given input into one or more categories. The labels or categories are not mutually exclusive.

For this example, we will be using the `go emotions` dataset, with Reddit comments categorized in 27 different emotions. Alongside the dataset, we've chosen a `DistilBERT` model, distilled from a zero-shot classification pipeline.

```

[ ]: import rubrix as rb
    from transformers import pipeline

```

(continues on next page)

(continued from previous page)

```

from datasets import load_dataset

# Loading our dataset
dataset = load_dataset("go_emotions", split="train[0:20]")

# Define our HuggingFace Pipeline
classifier = pipeline(
    "text-classification",
    model="joeddav/distilbert-base-uncased-go-emotions-student",
    framework="pt",
    return_all_scores=True,
)

records = []

for record in dataset:

    # Making the prediction
    prediction = classifier(record["text"], multi_label=True)

    # Creating the prediction entity as a list of tuples (label, probability)
    prediction = [(pred["label"], pred["score"]) for pred in prediction[0]]

    # Appending to the record list
    records.append(
        rb.TextClassificationRecord(
            text=record["text"],
            prediction=prediction,
            prediction_agent="https://huggingface.co/typeform/squeezebert-mnli",
            metadata={"split": "train"},
            multi_label=True, # we also need to set the multi_label option in Rubrix
        )
    )

# Logging into Rubrix
rb.log(
    record=records,
    name="multilabel-text-classification",
    tags={
        "task": "multilabel-text-classification",
        "family": "text-classification",
        "dataset": "go_emotions",
    },
)

```

Node Classification

The node classification task is the one where the model has to determine the labelling of samples (represented as nodes) by looking at the labels of their neighbours, in a Graph Neural Network. If you want to know more about GNNs, we've made a [tutorial](#) about them using Kglab and PyTorch Geometric, which integrates Rubrix into the pipeline.

4.7.2 Token Classification

Token classification kind-of-tasks are NLP tasks aimed to divide the input text into words, or syllables, and assign certain values to them. Think about giving each word in a sentence its grammatical category, or highlight which parts of a medical report belong to a certain speciality. There are some popular ones like NER or POS-tagging. For this part of the article, we will use [spaCy](#) with Rubrix to track and monitor Token Classification tasks.

Remember to install spaCy and datasets, or running the following cell.

```
[ ]: %pip install datasets -qqq
      %pip install -U spacy -qqq
      %pip install protobuf
```

NER

Named entity recognition (NER) is the task of tagging entities in text with their corresponding type. Approaches typically use *BIO* notation, which differentiates the beginning (**B**) and the inside (**I**) of entities. **O** is used for non-entity tokens.

For this tutorial, we're going to use the [Gutenberg Time](#) dataset from the Hugging Face Hub. It contains all explicit time references in a dataset of 52,183 novels whose full text is available via Project Gutenberg. From extracts of novels, we are surely going to find some NER entities. We will also use the `en_core_web_trf` pretrained English model, a Roberta-based spaCy model. If you do not have them installed, run:

```
[ ]: !python -m spacy download en_core_web_trf #Download the model
```

```
[ ]: import rubrix as rb
      import spacy
      from datasets import load_dataset

      # Load our dataset
      dataset = load_dataset("gutenberg_time", split="train[0:20]")

      # Load the spaCy model
      nlp = spacy.load("en_core_web_trf")

      records = []

      for record in dataset:

          # We only need the text of each instance
          text = record["tok_context"]

          # spaCy Doc creation
          doc = nlp(text)
```

(continues on next page)

(continued from previous page)

```

# Prediction entities with the tuples (label, start character, end character)
entities = [(ent.label_, ent.start_char, ent.end_char) for ent in doc.ents]

# Pre-tokenized input text
tokens = [token.text for token in doc]

# Rubrix TokenClassificationRecord list
records.append(
    rb.TokenClassificationRecord(
        text=text,
        token=tokens,
        prediction=entities,
        prediction_agent="en_core_web_trf",
    )
)

# Logging into Rubrix
rb.log(
    record=records,
    name="ner",
    tags={
        "task": "NER",
        "family": "token-classification",
        "dataset": "guttenberg-time",
    },
)

```

POS tagging

A POS tag (or part-of-speech tag) is a special label assigned to each word in a text corpus to indicate the part of speech and often also other grammatical categories such as tense, number, case etc. POS tags are used in corpus searches and in-text analysis tools and algorithms.

We will be repeating duo for this second spaCy example, with the [Gutenberg Time](#) dataset from the Hugging Face Hub and the `en_core_web_trf` pretrained English model.

```

[ ]: import rubrix as rb
import spacy
from datasets import load_dataset

# Load our dataset
dataset = load_dataset("gutenberg_time", split="train[0:10]")

# Load the spaCy model
nlp = spacy.load("en_core_web_trf")

records = []

for record in dataset:

    # We only need the text of each instance

```

(continues on next page)

(continued from previous page)

```

text = record["tok_context"]

# spaCy Doc creation
doc = nlp(text)

# Creating the prediction entity as a list of tuples (tag, start_char, end_char)
prediction = [(token.pos_, token.idx, token.idx + len(token)) for token in doc]

# Rubrix TokenClassificationRecord list
records.append(
    rb.TokenClassificationRecord(
        text=text,
        token=[token.text for token in doc],
        prediction=prediction,
        prediction_agent="en_core_web_trf",
    )
)

# Logging into Rubrix
rb.log(
    record=records,
    name="pos-tagging",
    tags={
        "task": "pos-tagging",
        "family": "token-classification",
        "dataset": "gutenberg-time",
    },
)

```

Slot Filling

The goal of Slot Filling is to identify, from a running dialog different slots, which one correspond to different parameters of the user's query. For instance, when a user queries for nearby restaurants, key slots for location and preferred food are required for a dialog system to retrieve the appropriate information. Thus, the goal is to look for specific pieces of information in the request and tag the corresponding tokens accordingly.

We made a tutorial on this matter for our open-source NLP library, [biome.text](#). We will use similar procedures here, focusing on the logging of the information. If you want to see in-depth explanations on how the pipelines are made, please visit [the tutorial](#).

Let's start by downloading `biome.text` and importing it alongside Rubrix.

```
[ ]: %pip install -U biome-text
exit(0) # Force restart of the runtime
```

```
[ ]: import rubrix as rb

from biome.text import Pipeline, Dataset, PipelineConfiguration, VocabularyConfiguration,
↳ Trainer
from biome.text.configuration import FeaturesConfiguration, WordFeatures, CharFeatures
from biome.text.modules.configuration import Seq2SeqEncoderConfiguration
from biome.text.modules.heads import TokenClassificationConfiguration

```


For this tutorial we will use the SNIPS data set adapted by Su Zhu.

```
[ ]: !curl -O https://biome-tutorials-data.s3-eu-west-1.amazonaws.com/token_classifier/train.
      ↪ json
!curl -O https://biome-tutorials-data.s3-eu-west-1.amazonaws.com/token_classifier/valid.
      ↪ json
!curl -O https://biome-tutorials-data.s3-eu-west-1.amazonaws.com/token_classifier/test.
      ↪ json

train_ds = Dataset.from_json("train.json")
valid_ds = Dataset.from_json("valid.json")
test_ds = Dataset.from_json("test.json")
```

Afterwards, we need to configure our biome.text Pipeline. More information on this configuration [here](#).

```
[ ]: word_feature = WordFeatures(
    embedding_dim=300,
    weights_file="https://dl.fbaipublicfiles.com/fasttext/vectors-english/wiki-news-300d-
    ↪ 1M.vec.zip",
)

char_feature = CharFeatures(
    embedding_dim=32,
    encoder={
        "type": "gru",
        "bidirectional": True,
        "num_layers": 1,
        "hidden_size": 32,
    },
    dropout=0.1
)

features_config = FeaturesConfiguration(
    word=word_feature,
    char=char_feature
)

encoder_config = Seq2SeqEncoderConfiguration(
    type="gru",
    bidirectional=True,
    num_layers=1,
    hidden_size=128,
)

labels = {tag[2:] for tags in train_ds["labels"] for tag in tags if tag != "0"}

for ds in [train_ds, valid_ds, test_ds]:
    ds.rename_column("labels", "tags")

head_config = TokenClassificationConfiguration(
    label=list(labels),
    label_encoding="BIO",
    top_k=1,
    feedforward={
```

(continues on next page)

(continued from previous page)

```

        "num_layers": 1,
        "hidden_dims": [128],
        "activations": ["relu"],
        "dropout": [0.1],
    },
)

```

And now, let's train our model!

```

[ ]: pipeline_config = PipelineConfiguration(
    name="slot_filling_tutorial",
    features=features_config,
    encoder=encoder_config,
    head=head_config,
)

pl = Pipeline.from_config(pipeline_config)

vocab_config = VocabularyConfiguration(min_count={"word": 2}, include_valid_data=True)

trainer = Trainer(
    pipeline=pl,
    train_dataset=train_ds,
    valid_dataset=valid_ds,
    vocab_config=vocab_config,
    trainer_config=None,
)

trainer.fit()

```

Having trained our model, we can go ahead and log the predictions to Rubrix.

```

[ ]: dataset = Dataset.from_json("test.json")

records = []

for record in dataset[0:10]["text"]:

    # We only need the text of each instance
    text = " ".join(word for word in record)

    # Predicting tags and entities given the input text
    prediction = pl.predict(text=text)

    # Creating the prediction entity as a list of tuples (tag, start_char, end_char)
    prediction = [
        (token["label"], token["start"], token["end"])
        for token in prediction["entities"][0]
    ]

    # Rubrix TokenClassificationRecord list
    records.append(

```

(continues on next page)

(continued from previous page)

```

        rb.tokenClassificationRecord(
            text=text,
            token=record,
            prediction=prediction,
            prediction_agent="biome_slot_filling_tutorial",
        )
    )

# Logging into Rubrix
rb.log(
    record=records,
    name="slot-filling",
    tags={
        "task": "slot-filling",
        "family": "token-classification",
        "dataset": "SNIPS",
    },
)

```

4.7.3 Text2Text

The expression *Text2Text* encompasses text generation tasks where the model receives and outputs a sequence of tokens. Examples of such tasks are machine translation, text summarization, paraphrase generation, etc.

Machine translation

Machine translation is the task of translating text from one language to another. It is arguably one of the oldest NLP tasks, but human parity remains an [open challenge](#) especially for low resource languages and domains.

In the following small example we will showcase how *Rubrix* can help you to fine-tune an English-to-Spanish translation model. Let us assume we want to translate “Sesame Street” related content. If you have been to Spain before you probably noticed that named entities (like character or band names) are often translated quite literally or are very different from the original ones.

We will use a pre-trained transformers model to get a few suggestions for the translation, and then correct them in *Rubrix* to obtain a training set for the fine-tuning.

```

[ ]: #!pip install transformers

from transformers import pipeline
import rubrix as rb

# Instantiate the translator
translator = pipeline("translation_en_to_es", model="Helsinki-NLP/opus-mt-en-es")

# 'Sesame Street' related phrase
en_phrase = "Sesame Street is an American educational children's television series_
↳starring the muppets Ernie and Bert."

```

(continues on next page)

(continued from previous page)

```
# Get two predictions from the translator
es_predictions = [output["translation_text"] for output in translator(en_phrase, num_
↳return_sequence=2)]

# Log the record to Rubrix and correct them
record = rb.text2textrecord(
    text=en_phrase,
    prediction=es_predictions,
)
rb.log(record, name="sesame_street_en-es")

# For a real training set you probably would need more than just one 'Sesame Street'
↳related phrase.
```

In the *Rubrix* web app we can now easily browse the predictions and annotate the records with a corrected prediction of our choice. The predictions for our example phrase are: 1. Sesame Street es una serie de televisión infantil estadounidense protagonizada por los muppets Ernie y Bert. 2. Sesame Street es una serie de televisión infantil y educativa estadounidense protagonizada por los muppets Ernie y Bert.

We probably would choose the second one and correct it in the following way:

2. *Barrio Sésamo* es una serie de televisión infantil y educativa estadounidense protagonizada por los *teleñecos* *Epi* y *Blas*.*

After correcting a substantial number of example phrases, we can load the corrected data set as a DataFrame to use it for the fine-tuning of the model.

```
[ ]: # load corrected translations for the fine-tuning of the translation model
df = rb.load("sesame_street_en-es")
```

4.8 Weak supervision

This guide gives you a brief introduction to weak supervision with Rubrix.

Rubrix currently supports weak supervision for multi-class text classification use cases, but we'll be adding support for multilabel text classification and token classification (e.g., Named Entity Recognition) soon.

4.8.1 Rubrix weak supervision in a nutshell

The recommended workflow for weak supervision is:

- Log an unlabelled dataset into Rubrix
- Use the **Annotate** mode for hand- and/or bulk-labelling a test set. This test is key to measure the quality and performance of your rules.
- Use the **Define rules** mode for testing and defining rules. Rules are defined with search queries (using ES query string DSL).
- Use the Python client for reading rules, defining additional rules if needed, and train a label (for building a training set) or a downstream model (for building an end classifier).

The next sections cover the main components of this workflow. If you want to jump into a practical tutorial, check the [news classification tutorial](#).

Weak labeling using the UI

Since version 0.8.0 you can find and define rules directly in the UI. The *Define rules mode* is found in the right side bar of the *Dataset page*. The video below shows how you can interactively find and save rules with the UI. For a full example check the *Weak supervision tutorial*.

Weak supervision from Python

Doing weak supervision with Rubrix should be straightforward. Keeping the same spirit as other parts of the library, you can virtually use any weak supervision library or method, such as Snorkel or Flyingsquid.

Rubrix weak supervision support is built around two basic abstractions:

Rule

A rule encodes an heuristic for labeling a record.

Heuristics can be defined using *Elasticsearch's queries*:

```
plz = Rule(query="plz OR please", label="SPAM")
```

or with Python functions (similar to Snorkel's labeling functions, which you can use as well):

```
def contains_http(record: rb.TextClassificationRecord) -> Optional[str]:
    if "http" in record.inputs["text"]:
        return "SPAM"
```

Besides textual features, Python labeling functions can exploit metadata features:

```
def author_channel(record: rb.TextClassificationRecord) -> Optional[str]:
    # the word channel appears in the comment author name
    if "channel" in record.metadata["author"]:
        return "SPAM"
```

A rule should either return a string value, that is a weak label, or a None type in case of abstention.

Weak Labels

Weak Labels objects bundle and apply a set of rules to the records of a Rubrix dataset. Applying a rule to a record means assigning a weak label or abstaining.

This abstraction provides you with the building blocks for training and testing weak supervision “denoising”, “label” or even “end” models:

```
rules = [contains_http, author_channel]
weak_labels = WeakLabels(
    rules=rules,
    dataset="weak_supervision_yt"
)
```

(continues on next page)

(continued from previous page)

```
# returns a summary of the applied rules
weak_labels.summary()
```

More information about these abstractions can be found in *the Python Labeling module docs*.

4.8.2 Built-in label models

To make things even easier for you, we provide wrapper classes around the most common label models, that directly consume a `WeakLabels` object. This makes working with those models a breeze. Take a look at the list of built-in models in the *labeling module docs*.

4.8.3 Detailed Workflow

A typical workflow to use weak supervision is:

1. Create a Rubrix dataset with your raw dataset. If you actually have some labelled data you can log it into the the same dataset.
2. Define a set of weak labeling rules with the Rules definition mode in the UI.
3. Create a `WeakLabels` object and apply the rules. You can load the rules from your dataset and add additional rules and labeling functions using Python. Typically, you'll iterate between this step and step 2.
4. Once you are satisfied with your weak labels, use the matrix of the `WeakLabels` instance with your library/method of choice to build a training set or even train a downstream text classification model.

This guide shows you an end-to-end example using Snorkel, Flyingsquid and Weasel. Let's get started!

4.8.4 Example dataset

We'll be using a well-known dataset for weak supervision examples, the [YouTube Spam Collection](#) dataset, which is a binary classification task for detecting spam comments in Youtube videos.

```
[1]: import pandas as pd

# load data
train_df = pd.read_csv('../tutorials/data/yt_comments_train.csv')
test_df = pd.read_csv('../tutorials/data/yt_comments_test.csv')

# preview data
train_df.head()
```

```
[1]:
```

	Unnamed: 0	author	date	\
0	0	Alessandro leite	2014-11-05T22:21:36	
1	1	Salim Tayara	2014-11-02T14:33:30	
2	2	Phuc Ly	2014-01-20T15:27:47	
3	3	DropShotSk8r	2014-01-19T04:27:18	
4	4	css403	2014-11-07T14:25:48	

		text	label	video
0	pls http://www10.vakinha.com.br/VaquinhaE.aspx...	-1.0	1	
1	if your like drones, plz subscribe to Kamal Ta...	-1.0	1	

(continues on next page)

(continued from previous page)

2	go here to check the views :3	-1.0	1
3	Came here to check the views, goodbye.	-1.0	1
4	i am 2,126,492,636 viewer :D	-1.0	1

4.8.5 1. Create a Rubrix dataset with unlabelled data and test data

Let's load the train (non-labelled) and the test (containing labels) dataset.

```
[ ]: import rubrix as rb

# build records from the train dataset
records = [
    rb.TextClassificationRecord(
        text=row.text,
        metadata={"video":row.video, "author": row.author}
    )
    for i,row in train_df.iterrows()
]

# build records from the test dataset with annotation
labels = ["HAM", "SPAM"]
records += [
    rb.TextClassificationRecord(
        text=row.text,
        annotation=labels[row.label],
        metadata={"video":row.video, "author": row.author}
    )
    for i,row in test_df.iterrows()
]

# log records to Rubrix
rb.log(records, name="weak_supervision_yt")
```

After this step, you have a fully browsable dataset available that you can access via the [Rubrix web app](#).

4.8.6 2. Defining rules

Let's now define some of the rules proposed in the tutorial [Snorkel Intro Tutorial: Data Labeling](#). Most of these rules can be defined directly with our web app in the *Define rules mode* and *Elasticsearch's query strings*. Afterward, you can conveniently load them into your notebook with the *load_rules* function.

Rules can also be defined programmatically as shown below. Depending on your use case and team structure you can mix and match both interfaces (UI or Python).

Let's see here some programmatic rules:

```
[ ]: from rubrix.labeling.text_classification import Rule, WeakLabels

# rules defined as Elasticsearch queries
check_out = Rule(query="check out", label="SPAM")
plz = Rule(query="plz OR please", label="SPAM")
```

(continues on next page)

(continued from previous page)

```
subscribe = Rule(query="subscribe", label="SPAM")
my = Rule(query="my", label="SPAM")
song = Rule(query="song", label="HAM")
love = Rule(query="love", label="HAM")
```

You can also define plain Python labeling functions:

```
[ ]: import re

# rules defined as Python labeling functions
def contains_http(record: rb.TextClassificationRecord):
    if "http" in record.inputs["text"]:
        return "SPAM"

def short_comment(record: rb.TextClassificationRecord):
    return "HAM" if len(record.inputs["text"].split()) < 5 else None

def regex_check_out(record: rb.TextClassificationRecord):
    return "SPAM" if re.search(r"check.*out", record.inputs["text"], flags=re.I) else_
↳ None
```

4.8.7 3. Building and analyzing weak labels

```
[ ]: from rubrix.labeling.text_classification import load_rules

# bundle our rules in a list
rules = [check_out, plz, subscribe, my, song, love, contains_http, short_comment, regex_
↳ check_out]

# optionally add the rules defined in the web app UI
rules += load_rules(dataset="weak_supervision_yt")

# apply the rules to a dataset to obtain the weak labels
weak_labels = WeakLabels(
    rules=rules,
    dataset="weak_supervision_yt"
)
```

```
[6]: # show some stats about the rules, see the `summary()` docstring for details
weak_labels.summary()
```

```
[6]:
```

	label	coverage	annotated_coverage	overlaps	\
check out	{SPAM}	0.242919	0.180	0.235839	
plz OR please	{SPAM}	0.090414	0.080	0.081155	
subscribe	{SPAM}	0.106754	0.120	0.083878	
my	{SPAM}	0.190632	0.188	0.166667	
song	{HAM}	0.132898	0.192	0.079521	
love	{HAM}	0.092048	0.140	0.070261	
contains_http	{SPAM}	0.106209	0.024	0.073529	
short_comment	{HAM}	0.245098	0.368	0.110566	
regex_check_out	{SPAM}	0.226580	0.180	0.226035	

(continues on next page)

(continued from previous page)

total	{SPAM, HAM}	0.754902	0.836	0.448802
	conflicts	correct	incorrect	precision
check out	0.029956	45	0	1.000000
plz OR please	0.019608	20	0	1.000000
subscribe	0.028867	30	0	1.000000
my	0.049564	41	6	0.872340
song	0.033769	39	9	0.812500
love	0.031590	28	7	0.800000
contains_http	0.049564	6	0	1.000000
short_comment	0.064270	84	8	0.913043
regex_check_out	0.027778	45	0	1.000000
total	0.120915	338	30	0.918478

4.8.8 4. Using the weak labels

At this step you have at least two options:

1. Use the weak labels for training a “denoising” or label model to build a less noisy training set. Highly popular options for this are [Snorkel](#) or [Flyingsquid](#). After this step, you can train a downstream model with the “clean” labels.
2. Use the weak labels directly with recent “end-to-end” (e.g., [Weasel](#)) or joint models (e.g., [COSINE](#)).

Let’s see some examples:

A simple majority vote

As a first example we will show you, how to use the `WeakLabels` object together with a simple majority vote model, which is arguably the most straightforward label model. On a per-record basis, it simply counts the votes for each label returned by the rules, and takes the majority vote. Rubrix provides a neat implementation of this logic in its `MajorityVoter` class.

```
[17]: from rubrix.labeling.text_classification import MajorityVoter

# instantiate the majority vote label model by simply providing the weak labels object
majority_model = MajorityVoter(weak_labels)
```

In contrast to the other label models we will discuss further down, the majority voter does not need to be fitted. You can directly check its performance by simply calling its `score()` method.

```
[20]: # check its performance
print(majority_model.score(output_size=True))
```

	precision	recall	f1-score	support
SPAM	0.99	0.92	0.95	89
HAM	0.94	0.99	0.96	111
accuracy			0.96	200
macro avg	0.96	0.96	0.96	200
weighted avg	0.96	0.96	0.96	200

An accuracy of 0.96 seems surprisingly high, but you need to keep in mind that we simply excluded the records from the evaluation, for which the model abstained (that is a tie in the votes or no votes at all). So let's account for this and correct the accuracy by assuming the model performs like a random classifier for these abstained records:

$$accuracy_c = frac_{non} \times accuracy + frac_{abs} \times accuracy_{random}$$

where $frac_{non}$ is the fraction of non-abstained records and $frac_{abs}$ the fraction of abstained records.

```
[ ]: # calculate fractions using the support metric (see above)
frac_non = 200 / len(weak_labels.annotation())
frac_abs = 1 - (200 / len(weak_labels.annotation()))

# accuracy without abstentions: 0.96; accuracy of random classifier: 0.5
print("accuracy_c:", frac_non * 0.96 + frac_abs * 0.5)
# accuracy_c: 0.868
```

As we will see further down, **an accuracy of 0.868** is still a very decent baseline.

Note

To get a noisy estimate of the corrected accuracy, you can also set the `"tie_break_policy"` argument: `majority_model.score(..., tie_break_policy="random")`.

When predicting weak labels to train a down-stream model, however, you probably want to discard the abstentions. Calling the `predict()` method on the majority voter, excludes the abstentions by default and only returns records without annotations. These are normally used to build a training set for a downstream model.

You can quickly explore the predicted records with Rubrix, before building a training set for training a downstream text classifier. This step is useful for validation, manual revision, or defining score thresholds for accepting labels from your label model (for example, only considering labels with a score greater than 0.8.)

```
[ ]: # get your training records with the predictions of the label model
records_for_training = majority_model.predict()

# optional: log the records to a new dataset in Rubrix
rh.log(records_for_training, name="majority_voter_results")

# extract training data
training_data = pd.DataFrame(
    [
        {"text": rec.text, "label": rec.prediction[0][0]}
        for rec in records_for_training
    ]
)
```

```
[36]: # preview training data
training_data
```

```
[36]:
```

	text	label
0	Hi I'm lil m !!! Check out love the way yo...	SPAM
1	LADIES!!! ----->> If you have a broken h...	SPAM
2	Love these guys, love the song!	HAM
3	She's awesome XD	HAM
4	go check out our video	SPAM
...

(continues on next page)

(continued from previous page)

```

1050                               Nice    HAM
1051  all u should go check out j rants vi about eminem  SPAM
1052                               Check out this playlist on YouTube:  SPAM
1053                               just came to check the view count  SPAM
1054                               Fantastic!!!    HAM

```

```
[1055 rows x 2 columns]
```

Label model with Snorkel

Snorkel's label model is by far the most popular option for using weak supervision, and Rubrix provides built-in support for it. Using Snorkel with Rubrix's `WeakLabels` is as simple as:

```

[ ]: %pip install snorkel -qqq

[ ]: from rubrix.labeling.text_classification import Snorkel

# we pass our WeakLabels instance to our Snorkel label model
snorkel_model = Snorkel(weak_labels)

# we fit the model
snorkel_model.fit(lr=0.001, n_epochs=50)

```

Note

The `Snorkel` label model is not suited for multi-label classification tasks and does not support them.

When fitting the snorkel model, we recommend performing a quick grid search for the learning rate `lr` and the number of epochs `n_epochs`.

```

[38]: # we check its performance
print(snorkel_model.score(output_str=True))

```

	precision	recall	f1-score	support
SPAM	0.96	0.93	0.94	95
HAM	0.94	0.96	0.95	114
accuracy			0.95	209
macro avg	0.95	0.95	0.95	209
weighted avg	0.95	0.95	0.95	209

At first sight, the model seems to perform worse than the majority vote baseline. However, let's again correct the accuracy for the abstentions.

```

[ ]: # calculate fractions using the support metric (see above)
frac_non = 209 / len(weak_labels.annotation())
frac_abs = 1 - (209 / len(weak_labels.annotation()))

# accuracy without abstentions: 0.95; accuracy of random classifier: 0.5

```

(continues on next page)

(continued from previous page)

```
print("accuracy_c:", frac_pos * 0.95 + frac_neg * 0.5)
# accuracy_c: 0.8761999999999999
```

Now we can see that with **an accuracy of 0.876**, its performance over the whole test set is actually slightly better.

After fitting your label model, you can quickly explore its predictions, before building a training set for training a downstream text classifier. This step is useful for validation, manual revision, or defining score thresholds for accepting labels from your label model (for example, only considering labels with a score greater than 0.8.)

```
[ ]: # get your training records with the predictions of the label model
records_for_training = snorkel_model.predict()

# optional: log the records to a new dataset in Rubrix
rh.log(records_for_training, name="snorkel_results")

# extract training data
training_data = pd.DataFrame(
    [
        {"text": rec.text, "label": rec.prediction[0][0]}
        for rec in records_for_training
    ]
)
```

```
[49]: # preview training data
training_data
```

```
[49]:
```

	text	label
0	Check out Melbourne shuffle, everybody!	SPAM
1	I love this song	HAM
2	I fuckin love this song! Afte...	HAM
3	Check out this video on YouTube:	SPAM
4	Who's watching in 2015 Subscribe for me !	SPAM
...
1172	Hey guys! Im a 12 yr old music producer. I mak...	SPAM
1173	Hey, check out my new website!! This site is a...	SPAM
1174	:3	HAM
1175	Hey! I'm NERDY PEACH and I'm a new youtuber an...	SPAM
1176	Are those real animals	HAM

[1177 rows x 2 columns]

Note

For an example of how to use the `WeakLabels` object with Snorkel's raw `LabelModel` class, you can check out the [WeakLabels reference](#).

Label model with FlyingSquid

FlyingSquid is a powerful method developed by [Hazy Research](#), a research group from Stanford behind ground-breaking work on programmatic data labeling, including Snorkel. FlyingSquid uses a closed-form solution for fitting the label model with great speed gains and similar performance. Just like for Snorkel, Rubrix provides built-in support for FlyingSquid, too.

```
[ ]: %pip install flyingsquid pgmpy -qqq

[ ]: from rubrix.labeling.text_classification import FlyingSquid

# we pass our WeakLabels instance to our FlyingSquid label model
flyingsquid_model = FlyingSquid(weak_labels)

# we fit the model
flyingsquid_model.fit()
```

Note

The FlyingSquid label model is not suited for multi-label classification tasks and does not support them.

```
[51]: # we check its performance
print(flyingsquid_model.score(support_score=True))
```

	precision	recall	f1-score	support
SPAM	0.93	0.91	0.92	95
HAM	0.92	0.95	0.94	114
accuracy			0.93	209
macro avg	0.93	0.93	0.93	209
weighted avg	0.93	0.93	0.93	209

Again, let's correct the accuracy for the abstentions.

```
[ ]: # calculate fractions using the support metric (see above)
frac_non = 209 / len(weak_labels.annotation())
frac_abs = 1 - (209 / len(weak_labels.annotation()))

# accuracy without abstentions: 0.93; accuracy of random classifier: 0.5
print("accuracy_c:", frac_non * 0.93 + frac_abs * 0.5)
# accuracy_c: 0.85948
```

Here, it really seems that with **an accuracy of 0.859**, the performance over the whole test set is actually slightly worse than the baseline of the majority vote.

After fitting your label model, you can quickly explore its predictions, before building a training set for training a downstream text classifier. This step is useful for validation, manual revision, or defining score thresholds for accepting labels from your label model (for example, only considering labels with a score greater than 0.8.)

```
[ ]: # get your training records with the predictions of the label model
records_for_training = flyingsquid_model.predict()
```

(continues on next page)

(continued from previous page)

```
# log the records to a new dataset in Rubrix
rx.log(records_for_training, name="flyingsquid_results")

# extract training data
training_data = pd.DataFrame(
    [
        {"text": rec.text, "label": rec.prediction[0][0]}
        for rec in records_for_training
    ]
)
```

```
[231]: # preview training data
training_data
```

```
[231]:
```

	text	label
0	Hey I'm a British youtuber!! I upload...	SPAM
1	NOKIA spotted	HAM
2	Dance :)	HAM
3	You guys should check out this EXTRAORDINARY w...	SPAM
4	Need money ? check my channel and subscribe,so...	SPAM
...
1172	Please check out my acoustic cover channel :)	SPAM
1173	PLEASE SUBSCRIBE ME!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!	SPAM
1174	<a href="http://www.gofundme.com/Helpmypitbull..."	SPAM
1175	I love this song so much!:-D I've heard it so ...	HAM
1176	Check out this video on YouTube:	SPAM

```
[1177 rows x 2 columns]
```

Joint Model with Weasel

Weasel lets you train downstream models end-to-end using directly weak labels. In contrast to Snorkel or FlyingSquid, which are two-stage approaches, Weasel is a one-stage method that jointly trains the label and the end model at the same time. For more details check out the [End-to-End Weak Supervision paper](#) presented at NeurIPS 2021.

In this guide we will show you, how you can **train a Hugging Face transformers** model directly **with weak labels using Weasel**. Since Weasel uses **PyTorch Lightning** for the training, some basic knowledge of PyTorch is helpful, but not strictly necessary.

Let's start with installing the Weasel python package:

```
[ ]: !python -m pip install git+https://github.com/autonlab/weasel#egg=weasel[all]
```

The first step is to obtain our weak labels. For this we use the same rules and data set as in the examples above (Snorkel and FlyingSquid).

```
[ ]: # obtain our weak labels
weak_labels = WeakLabels(
    rules=rules,
    dataset="weak_supervision_yt"
)
```

In a second step we instantiate our end model, which in our case will be a pre-trained transformer from the Hugging Face Hub. Here we choose the small ELECTRA model by Google that shows excellent performance given its moderate number of parameters. Due to its size, you can fine-tune it on your CPU within a reasonable amount of time.

```
[ ]: from weasel.models.downstream_models.transformers import Transformers

# instantiate our transformers end model
end_model = Transformers("google/electra-small-discriminator", num_labels=2)
```

With our end-model at hand, we can now instantiate the Weasel model. Apart from the end-model, it also includes a neural encoder that tries to estimate latent labels.

```
[ ]: from weasel.models import Weasel

# instantiate our weasel end-to-end model
weasel = Weasel(
    end_model=end_model,
    num_LF=len(weak_labels.rules),
    n_classes=2,
    encoder={'hidden_dims': [32, 10]},
    optim_encoder={'name': 'adam', 'lr': 1e-4},
    optim_end_model={'name': 'adam', 'lr': 5e-5},
)
```

Afterwards, we wrap our data in the TransformersDataModule, so that Weasel and PyTorch Lightning can work with it. In this step we also tokenize the data. Here we need to be careful to use the corresponding tokenizer to our end model.

```
[ ]: from transformers import AutoTokenizer
from weasel.datamodules.transformers_datamodule import TransformersDataModule, \
↳ TransformersCollator

# tokenizer for our transformers end model
tokenizer = AutoTokenizer.from_pretrained("google/electra-small-discriminator")

# tokenize train and test data
X_train = [
    tokenizer(rec.text, truncation=True)
    for rec in weak_labels.records(has_annotation=False)
]
X_test = [
    tokenizer(rec.text, truncation=True)
    for rec in weak_labels.records(has_annotation=True)
]

# instantiate data module
datamodule = TransformersDataModule(
    label_matrix=weak_labels.matrix(has_annotation=False),
    X_train=X_train,
    collator=TransformersCollator(tokenizer),
    X_test=X_test,
    Y_test=weak_labels.annotation(),
    batch_size=8
)
```

Now we have everything ready to start the training of our Weasel model. For the training process, Weasel relies on the excellent [PyTorch Lightning Trainer](#). It provides tons of options and features to optimize the training process, but the defaults below should give you reasonable results. Keep in mind that you are fine-tuning a full-blown transformer model, albeit a small one.

```
[ ]: import pytorch_lightning as pl

# instantiate the pytorch-lightning trainer
trainer = pl.Trainer(
    gpus=0, # >= 1 to use GPU(s)
    max_epochs=2,
    logger=None,
    callbacks=[pl.callbacks.ModelCheckpoint(monitor="Val/accuracy", mode="max")]
)

# fit the model end-to-end
trainer.fit(
    model=weasel,
    datamodule=datamodule,
)
```

After the training we can call the `Trainer.test` method to check the final performance. The model should achieve a test accuracy of around 0.94.

```
[ ]: trainer.test()
# {'accuracy': 0.94, ...}
```

To use the model for inference, you can either use its *predict* method:

```
[ ]: # Example text for the inference
text = "In my head this is like 2 years ago.. Time FLIES"

# Get predictions for the example text
predicted_probs, predicted_label = weasel.predict(
    tokenizer(text, return_tensors="pt")
)

# Map predicted int to label
weak_labels.int2label[int(predicted_label)] # HAM
```

Or you can instantiate one of the popular transformers pipelines, providing directly the end-model and the tokenizer:

```
[ ]: from transformers import pipeline

# modify the id2label mapping of the model
weasel.end_model.model.config.id2label = weak_labels.int2label

# create transformers pipeline
classifier = pipeline("text-classification", model=weasel.end_model.model,
    ↳tokenizer=tokenizer)

# use pipeline for predictions
classifier(text) # [{'label': 'HAM', 'score': 0.6110987663269043}]
```


4.9 Monitoring NLP pipelines

Rubrix currently gives users several ways to monitor and observe model predictions.

This brief guide introduces the different methods and expected usages.

4.9.1 Using `rb.monitor`

For widely-used libraries Rubrix includes an “auto-monitoring” option via the `rb.monitor` method. Currently supported libraries are Hugging Face Transformers and spaCy, if you’d like to see another library supported feel free to add a discussion or issue on GitHub.

`rb.monitor` will wrap HF and spaCy pipelines so every time you call them, the output of these calls will be logged into the dataset of your choice, as a background process, in a non-blocking way. Additionally, `rb.monitor` will add several tags to your dataset such as the library build version, the model name, the language, etc. This should also work for custom (private) pipelines, not only the Hub’s or official spaCy models.

It is worth noting that this feature is useful beyond monitoring, and can be used for data collection (e.g., bootstrapping data annotation with pre-trained pipelines), model development (e.g., error analysis), and model evaluation (e.g., combined with data annotation to obtain evaluation metrics).

Let’s see it in action using the IMDB dataset:

```
[ ]: from datasets import load_dataset

dataset = load_dataset("imdb", split="test[0:1000]")
```

Hugging Face Transformer Pipelines

Rubrix currently supports monitoring `text-classification` and `zero-shot-classification` pipelines, but `token-classification` and `text2text` pipelines will be added in coming releases.

```
[ ]: from transformers import pipeline
import rubrix as rb

nlp = pipeline("sentiment-analysis", return_all_scores=True, padding=True,
↳ truncation=True)
nlp = rb.monitor(nlp, dataset="nlp_monitoring")

dataset.map(lambda example: {"prediction": nlp(example["text"])}))
```

Once the `map` operation starts, you can start browsing the predictions in the Web-app:

The default Rubrix installation comes with **Elastic’s Kibana** pre-configured, so you can easily build custom monitoring dashboards and alerts (for your team and other stakeholders):

Record-level metadata is a key element of Rubrix datasets, enabling users to do fine-grained analysis and dataset slicing. Let’s see how we can log metadata while using `rb.monitor`. Let’s use the label in `ag_news` to add a `news_category` field for each record.

```
[ ]: dataset

[ ]: dataset.map(lambda example: {"prediction": nlp(example["text"]), metadata={"news_category":
↳ example["label"]}}))
```

spaCy

Rubrix currently supports monitoring the NER pipeline component, but textcat will be added soon.

```
[ ]: import spacy
import rubrix as rb

nlp = spacy.load("en_core_web_sm")
nlp = rb.monitor(nlp, dataset="nlp_monitoring_spacy")

dataset.map(lambda example: {"prediction": nlp(example["text"])})
```

Once the map operation starts, you can start browsing the predictions in the Web-app:

Flair

Rubrix currently supports monitoring Flair NER pipelines component.

```
[ ]: import rubrix as rb

from flair.data import Sentence
from flair.models import SequenceTagger

# load tagger
tagger = rb.monitor(SequenceTagger.load("flair/ner-english"), dataset="flair-example",
↪ sample_rate=1.0)

# make example sentence
sentence = Sentence("George Washington went to Washington")

# predict NER tags. This will log the prediction in Rubrix
tagger.predict(sentence)
```

The following logs the predictions over the IMDB dataset:

```
[ ]: def make_prediction(example):
    tagger.predict(Sentence(example["text"]))
    return {"prediction": True}

dataset.map(make_prediction)
```

4.9.2 Using rb.log in background mode

You can monitor your own models without adding a response delay by using the background param in rb.log

Let's see an example using [BentoML](#) with a spaCy NER pipeline:

```
[ ]: import spacy

nlp = spacy.load("en_core_web_sm")
```

```
[ ]: %%writefile spacy_model.py

from bentoml import BentoService, api, artifacts, env
from bentoml.adapters import JsonInput
from bentoml.frameworks.spacy import SpacyModelArtifact

import rubrix as rb

@env(infer_pip_packages=True)
@artifacts([SpacyModelArtifact("nlp")])
class SpacyNERService(BentoService):

    @api(input=JsonInput(), batch=True)
    def predict(self, parsed_json_list):
        result, rb_records = ([], [])
        for index, parsed_json in enumerate(parsed_json_list):
            doc = self.artifacts.nlp(parsed_json["text"])
            prediction = [{"entity": ent.text, "label": ent.label_} for ent in doc.ents]
            rb_records.append(
                rb.TokenClassificationRecord(
                    text=doc.text,
                    tokens=[t.text for t in doc],
                    prediction=[
                        (ent.label_, ent.start_char, ent.end_char) for ent in doc.ents
                    ],
                )
            )
            result.append(prediction)

        rb.log(
            name="monitor-for-spacy-ner",
            records=rb_records,
            tags={"framework": "bentoml"},
            background=True,
            verbose=False
        ) # By using the background=True, the model latency won't be affected

        return result
```

```
[ ]: from spacy_model import SpacyNERService

svc = SpacyNERService()
svc.pack('nlp', nlp)

saved_path = svc.save()
```

You can predict some data without serving the model. Just launch following command:

```
[ ]: !bentoml run SpacyNERService:latest predict --input "{\"text\": \"I am driving BMW\"}"
```

If you're running Rubrix in local, go to <http://localhost:6900/datasets/rubrix/monitor-for-spacy-ner> and see that the new dataset `monitor-for-spacy-ner` contains your data

4.9.3 Using the ASGI middleware

For using the ASGI middleware, see this [tutorial](#)

4.10 Metrics

This guide gives you a brief introduction to Rubrix Metrics. Rubrix Metrics enable you to perform fine-grained analyses of your models and training datasets. Rubrix Metrics are inspired by a number of seminal works such as [Explain-aboard](#).

The main goal is to make it easier to build more robust models and training data, going beyond single-number metrics (e.g., F1).

This guide gives a brief overview of currently supported metrics. For the full API documentation see the [Python API reference](#)

This feature is experimental, you can expect some changes in the Python API. Please report on Github any issue you encounter.

4.10.1 Install dependencies

Verify you have already installed Jupyter Widgets in order to properly visualize the plots. See https://ipywidgets.readthedocs.io/en/latest/user_install.html

For running this guide you need to install the following dependencies:

```
[ ]: %pip install datasets spacy plotly -qqq
```

and the spacy model:

```
[ ]: !python -m spacy download en_core_web_sm
```

4.10.2 1. Rubrix Metrics for NER pipelines predictions

Load dataset and spaCy model

We'll be using spaCy for this guide, but all the metrics we'll see are computed for any other framework (Flair, Stanza, Hugging Face, etc.). As an example will use the WNUT17 NER dataset.

```
[ ]: import rubrix as rb
import spacy
from datasets import load_dataset
```

(continues on next page)

(continued from previous page)

```
nlp = spacy.load("en_core_web_sm")
dataset = load_dataset("wnut_17", split="train")
```

Log records into a Rubrix dataset

Let's log spaCy predictions using the built-in `rb.monitor` method:

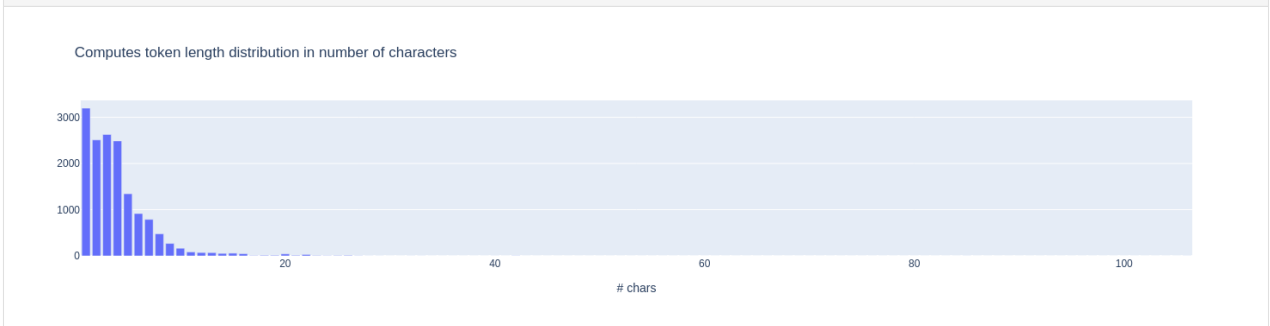
```
[ ]: nlp = rb.monitor(nlp, dataset="spacy_sm_wnut17")

def predict(record):
    doc = nlp(" ".join(record["tokens"]))
    return {"predicted": True}

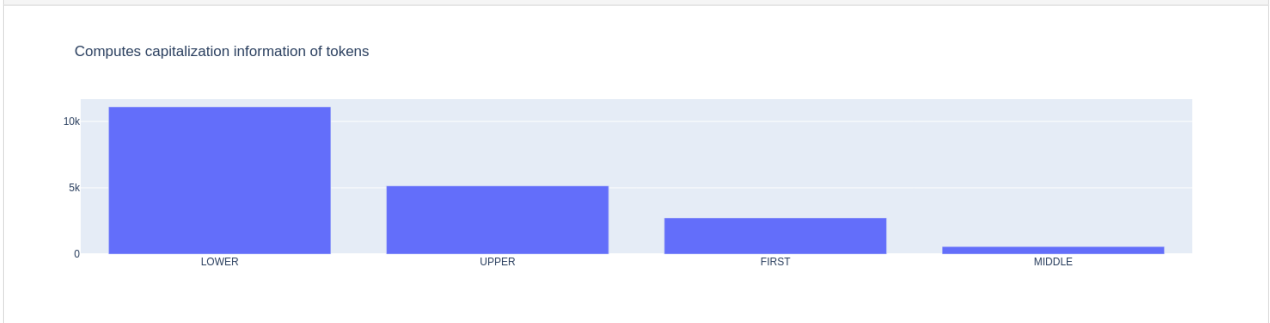
dataset.map(predict)
```

Explore some metrics for this pipeline

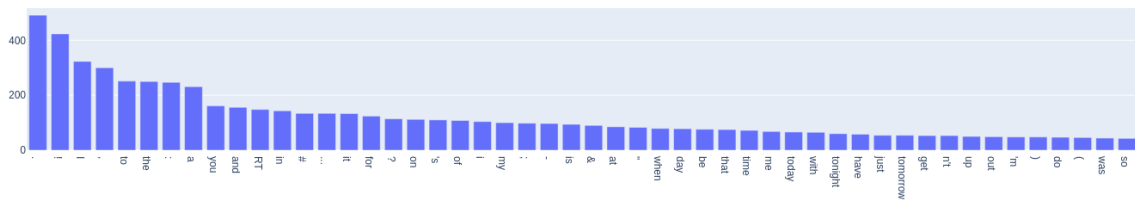
```
[17]: from rubrix.metrics.token_classification import token_length
token_length(name="spacy_sm_wnut17").visualize()
```



```
[7]: from rubrix.metrics.token_classification import token_capitalness
token_capitalness(name="spacy_sm_wnut17").visualize()
```

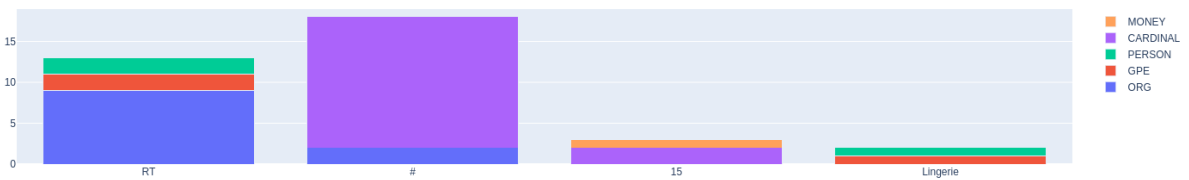


```
[20]: from rubrix.metrics.token_classification import token_frequency
token_frequency(name="spacy_sm_wnut17", tokens=50).visualize()
```



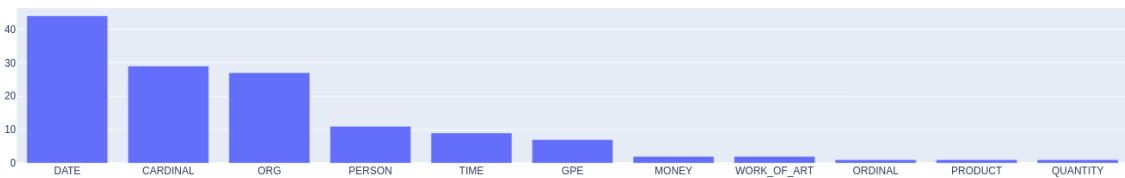
```
[21]: from rubrix.metrics.token_classification import entity_consistency
entity_consistency(name="spacy_sm_wnut17", mentions=5000, threshold=2).visualize()
```

Computes entity label variability for top-k predicted entity mentions



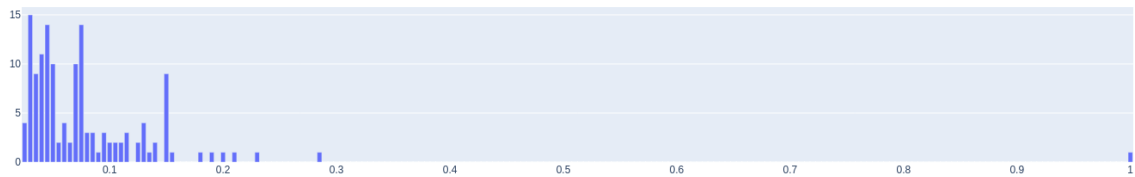
```
[5]: from rubrix.metrics.token_classification import entity_labels
entity_labels(name="spacy_sm_wnut17").visualize()
```

Predicted entity labels distribution



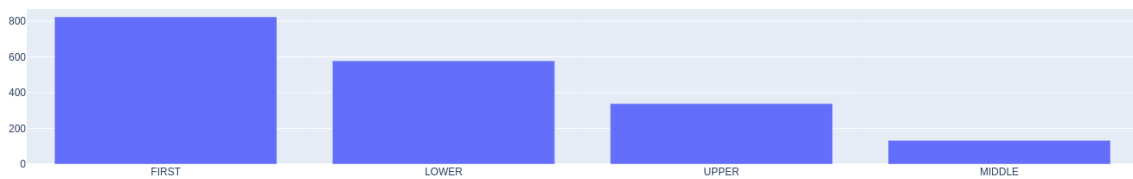
```
[6]: from rubrix.metrics.token_classification import entity_density
entity_density(name="spacy_sm_wnut17").visualize()
```

Computes the ratio between the number of all entity tokens and tokens in the text



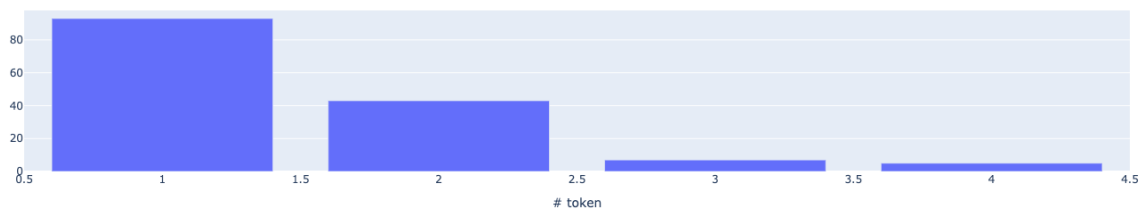
```
[8]: from rubrix.metrics.token_classification import entity_capitalness
entity_capitalness(name="spacy_sm_wnut17").visualize()
```

Computes capitalization information of predicted entity mentions



```
[8]: from rubrix.metrics.token_classification import mention_length
mention_length(name="spacy_sm_wnut17").visualize()
```

Computes the length of the predicted entity mention measured in number of tokens



4.10.3 2. Rubrix Metrics for NER training sets

Analyzing tags

Let's analyze the conll2002 dataset at the tag level.

```
[ ]: dataset = load_dataset("conll2002", "es", split="train[0:5000]")
```

```
[24]: def parse_entities(record):
        entities = []
        counter = 0
        for i in range(len(record['ner_tags'])):
```

(continues on next page)

(continued from previous page)

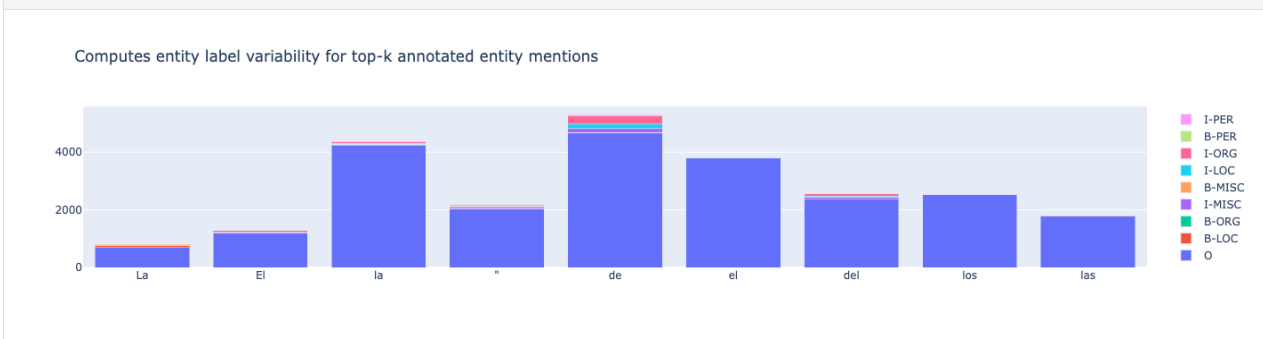
```
entity = (dataset.features["ner_tags"].feature.names[record["ner_tags"]],
↪ counter, counter + len(record["tokens"][-1]))
entity.append(entity)
counter += len(record["tokens"][-1]) + 1
return entities
```

```
[30]: records = [
    rb.TokenClassificationRecord(
        text=" ".join(example["tokens"]),
        tokens=example["tokens"],
        annotation=parse_entities(example)
    )
    for example in dataset
]
```

```
[ ]: rb.log(records, "conll2002_es")
```

```
[51]: from rubrix.metrics.token_classification import entity_consistency
from rubrix.metrics.token_classification.metrics import Annotations
```

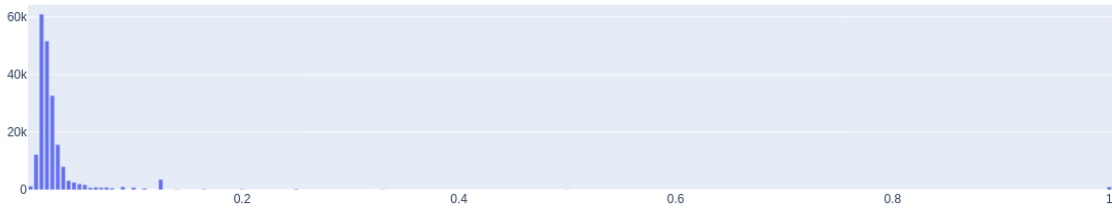
```
entity_consistency(name="conll2002_es", mention=30, threshold=4, compute_
↪ for=Annotations).visualize()
```



From the above we see we can quickly detect an annotation issue: double quotes " are most of the time tagged as O (no entity) but in some cases (~60 examples) are tagged as beginning of entities like ORG or MISC, which is likely a hand-labelling error, including the quotes inside the entity span.

```
[54]: from rubrix.metrics.token_classification import *
entity_density(name="conll2002_es", compute_for=Annotations).visualize()
```


Computes the ratio between the number of all entity tokens and tokens in the text



4.10.4 2. Rubrix Metrics for text classification

```
[ ]: from datasets import load_dataset
      from transformers import pipeline

      import rubrix as rb

      sst2 = load_dataset("glue", "sst2", split="validation")
      labels = sst2.features["label"].names
      nlp = pipeline("sentiment-analysis")
```

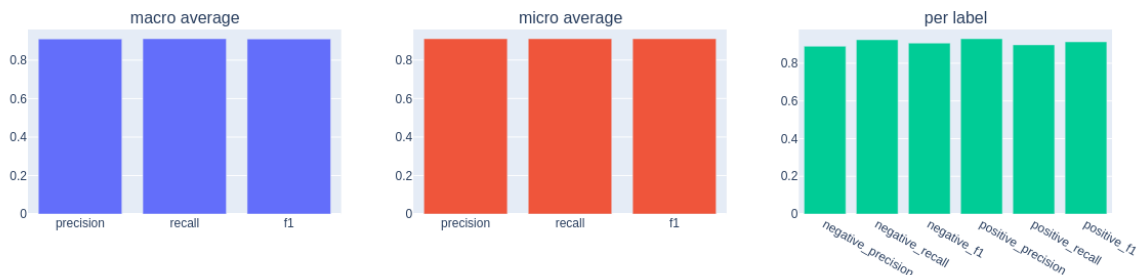
```
[11]: records = [
        rb.TextClassificationRecord(
            text=record["sentence"],
            annotation=labels[record["label"]],
            prediction=[(pred["label"].lower(), pred["score"]) for pred in nlp(record[
↪ "sentence"])]
        )
        for record in sst2
    ]
```

```
[ ]: rb.log(records, name="sst2")
```

```
[13]: from rubrix.metrics.text_classification import f1

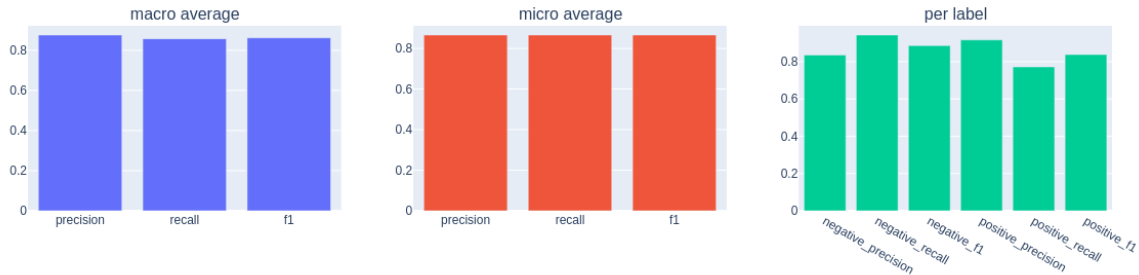
      f1(name="sst2").visualize()
```

F1 Metrics for single-label (averaged and per label)



```
[20]: # now compute metrics for negation ( -> negative precision and positive recall go down)
      f1(name="sst2", query="n't OR not").visualize()
```

F1 Metrics for single-label (averaged and per label)



4.11 Datasets

This guide showcases some features of the `Dataset` classes in the Rubrix client. The `Dataset` classes are lightweight containers for Rubrix records. These classes facilitate importing from and exporting to different formats (e.g., `pandas.DataFrame`, `datasets.Dataset`) as well as sharing and versioning Rubrix datasets using the Hugging Face Hub.

For each record type there's a corresponding `Dataset` class called `DatasetFor<RecordType>`. You can look up their API in the [reference section](#)

4.11.1 Working with a Dataset

Under the hood the `Dataset` classes store the records in a simple Python list. Therefore, working with a `Dataset` class is not very different to working with a simple list of records:

```
[ ]: import rubrix as rb

# Start with a list of Rubrix records
dataset_rb = rb.DatasetForTextClassification(my_records)

# Loop over the dataset
for record in dataset_rb:
    print(record)

# Index into the dataset
dataset_rb[0] = rb.TextClassificationRecord(text="replace record")

# log a dataset to the Rubrix web app
rb.log(dataset_rb, "my_dataset")
```

The `Dataset` classes do some extra checks for you, to make sure you do not mix record types when appending or indexing into a dataset.

4.11.2 Importing from other formats

When you have your data in a `pandas DataFrame` or a `datasets Dataset`, we provide some neat shortcuts to import this data into a Rubrix Dataset. You have to make sure that the data follows the record model of a specific task, otherwise you will get validation errors. Columns in your `DataFrame/Dataset` that are not supported or recognized, will simply be ignored.

The record models of the tasks are explained in the [reference section](#).

Note

Due to its pyarrow nature, data in a `datasets.Dataset` has to follow a slightly different model, that you can look up in the examples of the `Dataset*.from_datasets` [docstrings](#).

```
[ ]: import rubrix as rb

# import data from a pandas DataFrame
dataset_rb = rb.read_pandas(my_dataframe, task="TextClassification")
# or
dataset_rb = rb.DatasetForTextClassification.from_pandas(my_dataframe)

# import data from a datasets Dataset
dataset_rb = rb.read_datasets(my_dataset, task="TextClassification")
# or
dataset_rb = rb.DatasetForTextClassification.from_datasets(my_dataset)
```

We also provide helper arguments you can use to read almost arbitrary datasets for a given task from the [Hugging Face Hub](#). They map certain input arguments of the Rubrix records to columns of the given dataset. Let's have a look at a few examples:

```
[ ]: import rubrix as rb
from datasets import load_dataset

# the "poem_sentiment" dataset has columns "verse_text" and "label"
dataset_rb = rb.DatasetForTextClassification.from_datasets(
    dataset=load_dataset("poem_sentiment", split="test"),
    text="verse_text",
    annotation="label",
)

# the "snli" dataset has the columns "premise", "hypothesis" and "label"
dataset_rb = rb.DatasetForTextClassification.from_datasets(
    dataset=load_dataset("snli", split="test"),
    input=["premise", "hypothesis"],
    annotation="label",
)

# the "conll2003" dataset has the columns "id", "tokens", "pos_tags", "chunk_tags" and
# ↪ "ner_tags"
rb.DatasetForTokenClassification.from_datasets(
    dataset=load_dataset("conll2003", split="test"),
    tags="ner_tags",
)
```

(continues on next page)

(continued from previous page)

```
# the "xsum" dataset has the columns "id", "document" and "summary"
rb.DatasetForTextText.from_datasets(
    dataset=load_dataset("xsum", split="test"),
    text="document",
    annotation="summary",
)
```

You can also use the shortcut `rb.read_datasets(dataset=..., task=..., **kwargs)` where the keyword arguments are passed on to the corresponding `from_datasets()` method.

4.11.3 Sharing via the Hugging Face Hub

You can easily share your Rubrix dataset with your community via the Hugging Face Hub. For this you just need to export your Rubrix Dataset to a `datasets.Dataset` and [push it to the hub](#):

```
[ ]: import rubrix as rb

# load your annotated dataset from the Rubrix web app
dataset_rb = rb.load("my_dataset")

# export your Rubrix Dataset to a datasets Dataset
dataset_ds = dataset_rb.to_datasets()

# push the dataset to the Hugging Face Hub
dataset_ds.push_to_hub("my_dataset")
```

Afterward, your community can easily access your annotated dataset and log it directly to the Rubrix web app:

```
[ ]: from datasets import load_dataset

# download the dataset from the Hugging Face Hub
dataset_ds = load_dataset("user/my_dataset", split="train")

# read in dataset, assuming its a dataset for text classification
dataset_rb = rb.read_datasets(dataset_ds, task="TextClassification")

# log the dataset to the Rubrix web app
rb.log(dataset_rb, "dataset_by_user")
```

4.11.4 Prepare dataset for training

If you want to train a Hugging Face transformer with your dataset, we provide a handy method to prepare your dataset: `DatasetFor*.prepare_for_training()`. It will return a Hugging Face dataset, optimized for the training process with the Hugging Face Trainer.

For text classification tasks, it flattens the inputs into separate columns of the returned dataset and converts the annotations of your records into integers and writes them in a label column:

```
[ ]: dataset_rb = rb.DatasetForTextClassification([
    rb.TextClassificationRecord(inputs={"title": "My title", "content": "My content"},
    ↪ annotation="news")
])
```

(continues on next page)

(continued from previous page)

```

])

dataset_rb.prepare_for_training()[0]
# Output:
# {'title': 'My title', 'content': 'My content', 'label': 0}

```

For token classification tasks, it converts the annotations of a record into integers representing BIO tags and writes them in a `ner_tags` column:

```

[ ]: dataset_rb = rb.DatasetForTokenClassification([
    rb.TokenClassificationRecord(text="I live in Madrid", token=["I", "live", "in",
↪ "Madrid"], annotation=[("LOC", 10, 15)])
])

dataset_rb.prepare_for_training()[0]
# Output:
# {..., 'tokens': ['I', 'live', 'in', 'Madrid'], 'ner_tags': [0, 0, 0, 1], ...}

```

4.12 Dataset settings

Rubrix datasets have certain *settings* that you can configure via the `rb.*Settings` classes, for example `rb.TextClassificationSettings`.

4.12.1 Define a labeling schema

You can define a labeling schema for your Rubrix dataset, which fixes the allowed labels for your predictions and annotations. Once you set a labeling schema, each time you log to the corresponding dataset, Rubrix will perform validations of the added predictions and annotations to make sure they comply with the schema.

```

[7]: import rubrix as rb

# Define labeling schema
settings = rb.TextClassificationSettings(label_schema=["A", "B", "C"])

# Apply settings to a new or already existing dataset
rb.configure_dataset(name="my_dataset", settings=settings)

# Logging to the newly created dataset triggers the validation checks
rb.log(rb.TextClassificationRecord(text="text", annotation="D"), "my_dataset")
#BadRequestApiError: Rubrix server returned an error with http status: 400

```

4.13 Queries

The search in Rubrix is driven by Elasticsearch’s powerful [query string syntax](#). It allows you to perform simple fuzzy searches of words and phrases, or complex queries taking full advantage of Rubrix’s data model.

These queries can be used in the search bar of the Rubrix web app, or with the Python client as optional arguments.

4.13.1 Search fields

An important concept when searching with Elasticsearch is the *field* concept. Every search term in Rubrix is directed to a specific field of the record’s underlying data model. For example, writing `text:fox` in the search bar will search for records with the word `fox` in the field `text`.

If you do not provide any fields in your query string, by default Rubrix will search in the `text` field. For a complete list of available fields and their content, have a look at the field glossary below.

Note: The default behavior when not specifying any fields in the query string changed in version `>=0.16.0`. Before this version, Rubrix searched in a mixture of the deprecated `word` and `word.extended` fields that allowed searches for special characters like `!` and `..`. If you want to search for special characters now, you have to specify the `text.exact` field. For example, this is the query if you want to search for words with an exclamation mark in the end: `text.exact:*\!`

4.13.2 `text` and `text.exact`

The (arguably) most important fields are the `text` and `text.exact` fields. They both contain the text of the records, however in two different forms:

- the `text` field uses Elasticsearch’s [standard analyzer](#) that ignores capitalization and removes most of the punctuation;
- the `text.exact` field uses the [whitespace analyzer](#) that differentiates between lower and upper case, and does take into account punctuation;

Let’s have a look at a few examples. Suppose we have 2 records with the following text:

1. *The quick brown fox jumped over the lazy dog.*
2. *THE LAZY DOG HATED THE QUICK BROWN FOX!*

Now consider these queries:

- `text:dog.` or `text:fox`: matches both of the records.
- `text.exact:dog` or `text.exact:FOX`: matches none of the records.
- `text.exact:dog.` or `text.exact:fox`: matches only the first record.
- `text.exact:DOG` or `text.exact:FOX\!`: matches only the second record.

You can see how the `text.exact` field can be used to search in a more fine-grained manner.

TextClassificationRecord's inputs

For `text classification records` you can take advantage of the multiple `inputs` when performing a search. For example, if we uploaded records with `inputs={"subject": ..., "body": ...}`, you can direct your searches to only one of those inputs by specifying the `inputs.subject` or `inputs.body` field in your query. So to look for records in which the *subject* contains the word *news*, you would search for

- `inputs.subject:news`

Again, as with the `text` field, you can also use the white space analyzer to perform more fine-grained searches by specifying the `exact` field:

- `inputs.subject.exact:NEWS`

4.13.3 Words and phrases

Apart from single words you can also search for *phrases* by surrounding multiples words with double quotes. This searches for all the words in the phrase, in the same order.

If we take the two examples from above, then following query will only return the second example:

- `text:"lazy dog hated"`

4.13.4 Metadata fields

You also have the metadata of your records available when performing a search. Imagine you provided the `split` to which the record belongs to as metadata, that is `metadata={"split": "train"}` or `metadata={"split": "test"}`. Then you could only search your training data by specifying the corresponding field in your query:

- `metadata.split:train`

Metadata are indexed as keywords. This means you cannot search for single words in them, and capitalization and punctuations are taken into account. You can, however, use wild cards.

4.13.5 Filters as query string

Just like the metadata, you can also use the filter fields in you query. A few examples to emulate the filters in the query string are:

- `status:Validated`
- `annotated_as:HAM`
- `predicted_by:Model A`

The field values are treated as keywords, that is you cannot search for single words in them, and capitalization and punctuations are taken into account. You can, however, use wild cards.

4.13.6 Combine terms and fields

You can combine an arbitrary amount of terms and fields in your search using the familiar boolean operators AND, OR and NOT. Following examples showcase the power of these operators:

- `text:(quick AND fox)`: Returns records that contain the word *quick* and *fox*. The AND operator is the default operator, so `text:(quick fox)` is equivalent.
- `text:(quick OR brown)`: Returns records that contain either the word *quick* or *brown*.
- `text:(quick AND fox AND NOT news)`: Returns records that contain the words *quick* and *fox*, **and do not** contain *news*.
- `metadata.split:train AND text:fox`: Returns records that contain the word *fox* and that have a metadata “*split: train*”.
- `NOT _exists_:metadata.split`: Returns records that don’t have a metadata *split*.

4.13.7 Query string features

The query string syntax has many powerful features that you can use to create complex searches. Following is just a hand selected subset of the many features you can look up on the official [Elasticsearch documentation](#).

Wildcards

Wildcard searches can be run on individual search terms, using ? to replace a single character, and * to replace zero or more characters:

- `text:(qu?ck bro*)`
- `text.exact:"Lazy Dog*"`: Matches, for example, “*Lazy Dog*”, “*Lazy Dog.*”, or “*Lazy Dogs*”.
- `inputs.*:news`: Searches all input fields for the word *news*.

Regular expressions

Regular expression patterns can be embedded in the query string by wrapping them in forward slashes “/”:

- `text:/joh?n(ath[oa]n)/`: Matches *jonathon*, *jonathan*, *johnathon*, and *johnathan*.

The supported regular expression syntax is explained on the official [Elasticsearch documentation](#).

Fuzziness

You can search for terms that are similar to, but not exactly like the search terms, using the *fuzzy* operator. This is useful to cover human misspellings:

- `text:quikc~`: Matches *quick* and *quikc*.

Ranges

Ranges can be specified for date, numeric or string fields. Inclusive ranges are specified with square brackets and exclusive ranges with curly brackets:

- `score:[0.5 TO 0.6]`
- `score:{0.9 TO *}`

Escaping special characters

The query string syntax has some reserved characters that you need to escape if you want to search for them. The reserved characters are: `+ - = & | > < ! () { } [] ^ " ~ * ? : \ /` For instance, to search for `“(I+I)=2”` you need to write:

- `text:\(1\+1\)\=2`

4.13.8 Field glossary

This is a table with available fields that you can use in your query string:

Field name	Description	TextClass.	TokenClass.	Text2Text
annotated_as	annotation	✓	✓	✓
annotated_by	annotation agent	✓	✓	✓
event_timestamp	timestamp	✓	✓	✓
id	id	✓	✓	✓
inputs.*	inputs	✓		
metadata.*	metadata	✓	✓	✓
last_updated	date of the last update	✓	✓	✓
predicted_as	prediction	✓	✓	✓
predicted_by	prediction agent	✓	✓	✓
score	prediction score	✓		
status	status	✓	✓	✓
text	text, standard analyzer	✓	✓	✓
text.exact	text, whitespace analyzer	✓	✓	✓
tokens	tokens		✓	
-	-	-	-	-
metrics.text_len	Input text length	✓	✓	✓
metrics.tokens.idx	Token idx in record		✓	
metrics.tokens.value	Text of the token		✓	
metrics.tokens.char_start	Start char idx of token		✓	
metrics.tokens.char_end	End char idx of token		✓	
metrics.annotated.mentions.value	Text of the mention (annotation)		✓	
metrics.annotated.mentions.label	Label of the mention (annotation)		✓	
metrics.annotated.mentions.score	Score of the mention (annotation)		✓	
metrics.annotated.mentions.capitalness	Mention capitalness (annotation)		✓	
metrics.annotated.mentions.density	Local mention density (annotation)		✓	
metrics.annotated.mentions.tokens_length	Mention length in tokens (annotation)		✓	
metrics.annotated.mentions.chars_length	Mention length in chars (annotation)		✓	
metrics.annotated.tags.value	Text of the token (annotation)		✓	
metrics.annotated.tags.tag	IOB tag (annotation)		✓	

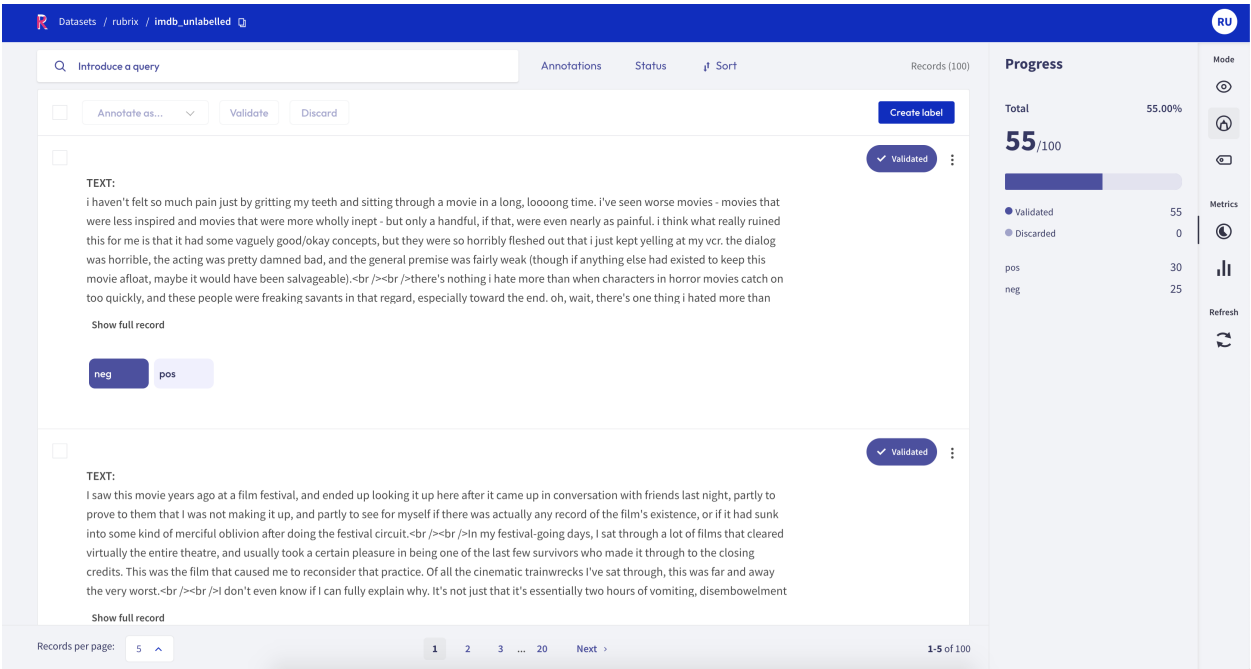
continues on next page

Table 1 – continued from previous page

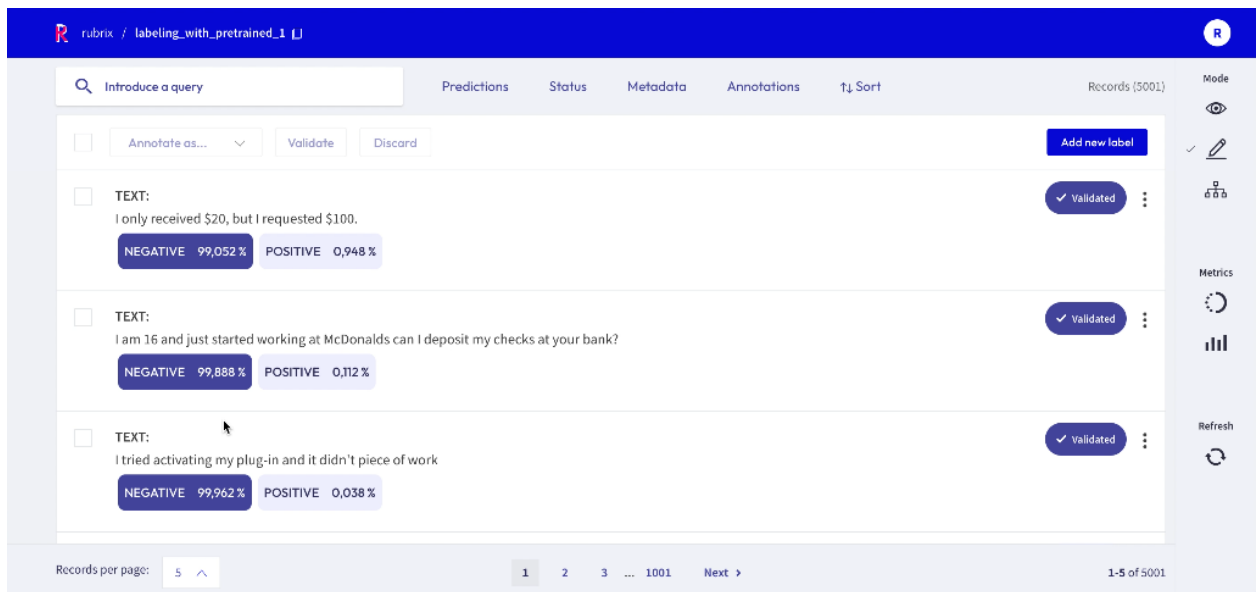
Field name	Description	TextClass.	TokenClass.	Text2Text
metrics.predicted.mentions.value	Text of the mention (prediction)		✓	
metrics.predicted.mentions.label	Label of the mention (prediction)		✓	
metrics.predicted.mentions.score	Score of the mention (prediction)		✓	
metrics.predicted.mentions.capitalness	Mention capitalness (prediction)		✓	
metrics.predicted.mentions.density	Local mention density (prediction)		✓	
metrics.predicted.mentions.tokens_length	Mention length in tokens (prediction)		✓	
metrics.predicted.mentions.chars_length	Mention length in chars (prediction)		✓	
metrics.predicted.tags.value	Text of the token (prediction)		✓	
metrics.predicted.tags.tag	IOB tag (prediction)		✓	

4.14 Introductory

These tutorials are a good starting point and introduce you to various topics covered by Rubrix.



Few-shot classification with SetFit and a custom dataset Build a custom text classifier using SetFit, few-shot classification with Sentence Transformers



Building a news classifier with weak supervision Build a sentiment classifier for user requests in the banking domain by leveraging a pre-trained model from the Hugging Face Hub.

Building a news classifier with weak supervision Build a news classifier using rules and weak supervision, and how Rubrix makes this process very interactive.

4.14.1 Label your data to fine-tune a classifier with Hugging Face

In this tutorial, we'll build a sentiment classifier for user requests in the banking domain as follows:

- Start with the most popular sentiment classifier on the Hugging Face Hub (almost 4 million monthly downloads as of December 2021) which has been fine-tuned on the SST2 sentiment dataset.
- Label a training dataset with banking user requests starting with the pre-trained sentiment classifier predictions.
- Fine-tune the pre-trained classifier with your training dataset.
- Label more data by correcting the predictions of the fine-tuned model.
- Fine-tune the pre-trained classifier with the extended training dataset.

Introduction

This tutorial will show you how to fine-tune a sentiment classifier for your own domain, starting with no labeled data.

Most online tutorials about fine-tuning models assume you already have a training dataset. You'll find many tutorials for fine-tuning a pre-trained model with widely-used datasets, such as IMDB for sentiment analysis.

However, very often **what you want is to fine-tune a model for your use case**. It's well-known that NLP model performance usually degrades with "out-of-domain" data. For example, a sentiment classifier pre-trained on movie reviews (e.g., IMDB) will not perform very well with customer requests.

This is an overview of the workflow we'll be following:

Let's get started!

Setup

Rubrix, is a free and open-source tool to explore, annotate, and monitor data for NLP projects.

If you are new to Rubrix, check out the [Github repository](#).

If you have not installed and launched Rubrix, check the *Setup and Installation guide*.

In this tutorial, we'll use the `transformers`, `datasets` and `sklearn` libraries. We'll also install `ipywidgets` for training progress bars.

```
[ ]: %pip install transformers[torch] datasets sklearn ipywidgets -qq
```

Preliminaries

For building our fine-tuned classifier we'll be using two main resources, both available in the Hub :

1. A **dataset** in the banking domain: `banking77`
2. A **pre-trained sentiment classifier**: `distilbert-base-uncased-finetuned-sst-2-english`

Dataset: Banking 77

This dataset contains online banking user queries annotated with their corresponding intents.

In our case, **we'll label the sentiment of these queries**. This might be useful for digital assistants and customer service analytics.

Let's load the dataset directly from the hub and split the dataset into two 50% subsets. We'll start with the `to_label1` split for data exploration and annotation, and keep `to_label2` for further iterations.

```
[ ]: from datasets import load_dataset

banking_db = load_dataset("banking77")

to_label1, to_label2 = banking_db['train'].train_test_split(test_size=0.5, seed=42).
↪ values()
```

Model: sentiment distilbert fine-tuned on sst-2

As of December 2021, the `distilbert-base-uncased-finetuned-sst-2-english` is in the top five of the most popular text-classification models in the [Hugging Face Hub](#).

This model is a **distilbert model** fine-tuned on SST-2 (Stanford Sentiment Treebank), a highly popular sentiment classification benchmark.

As we will see later, this is a general-purpose sentiment classifier, which will need further fine-tuning for specific use cases and styles of text. In our case, **we'll explore its quality on banking user queries and build a training set for adapting it to this domain**.

Let's load the model and test it with an example from our dataset:

```
[3]: from transformers import pipeline

sentiment_classifier = pipeline(
```

(continues on next page)

(continued from previous page)

```

model="distilbert-base-uncased-finetuned-sst-2-english",
task="sentiment-analysis",
return_all_scores=True,
)

to_label([3]['text'], sentiment_classifier(to_label([3]['text']))

```

```

[3]: ('Hi, Last week I have contacted the seller for a refund as directed by you, but i have.
↳not received the money yet. Please look into this issue with seller and help me in.
↳getting the refund.',
[[{'label': 'NEGATIVE', 'score': 0.9934700727462769},
 {'label': 'POSITIVE', 'score': 0.0065299225971102715}])

```

The model assigns more probability to the NEGATIVE class. Following our annotation policy (read more below), we'll label examples like this as POSITIVE as they are general questions, not related to issues or problems with the banking application. The ultimate goal will be to fine-tune the model to predict POSITIVE for these cases.

A note on sentiment analysis and data annotation

Sentiment analysis is one of the most subjective tasks in NLP. What we understand by sentiment will vary from one application to another and depend on the business objectives of the project. Also, sentiment can be modeled in different ways, leading to different **labeling schemes**. For example, sentiment can be modeled as real value (going from -1 to 1, from 0 to 1.0, etc.) or with 2 or more labels (including different degrees such as positive, negative, neutral, etc.)

For this tutorial, we'll use the **original labeling scheme** defined by the pre-trained model which is composed of two labels: POSITIVE and NEGATIVE. We could have added the NEUTRAL label, but let's keep it simple.

Another important issue when approaching a data annotation project are the **annotation guidelines**, which explain how to assign the labels to specific examples. As we'll see later, the messages we'll be labeling are mostly questions with a neutral sentiment, which we'll label with the POSITIVE label, and some other are negative questions which we'll label with the NEGATIVE label. Later on, we'll show some examples of each label.

1. Run the pre-trained model over the dataset and log the predictions

As a first step, let's use the pre-trained model for predicting over our raw dataset. For this, we will use the handy `dataset.map` method from the `datasets` library.

The following steps could be simplified by using the auto-monitor support for Hugging Face pipelines. You can find more details in the [Monitoring guide](#).

Predict

```

[ ]: def predict(examples):
    return {"predictions": sentiment_classifier(examples['text'], truncation=True)}

# add .select(range(10)) before map if you just want to test this quickly with 10
↳examples
to_label = to_label.map(predict, batched=True, batch_size=4)

```

Note

If you don't want to run the predictions yourself, you can also load the records with the predictions directly from the Hugging Face Hub: `load_dataset("rubrix/sentiment-banking", split="train")`, see below for more details.

Create records

The following code builds a list of Rubrix records with the predictions.

```
[ ]: import rubrix as rb

records = []
for example in to_label.shuffle():
    record = rb.TextClassificationRecord(
        text=example["text"],
        metadata={'category': example['label']}, # log the intents for exploration of
        ↪specific intents
        prediction=[(pred['label'], pred['score']) for pred in example['predictions']],
        prediction_agent="distilbert-base-uncased-finetuned-sst-2-english"
    )
    records.append(record)
```

Before logging the records to Rubrix, we will upload them to the [Hugging Face Hub](#). In this way we save a version of them with the predictions, so the next time we do this tutorial, we don't have to run the pre-trained model again. You can do the same, once you annotated the dataset to effectively version your complete records.

```
[ ]: dataset_rb = rb.DatasetForTextClassification(records)
dataset_ds = dataset_rb.to_dataset()

dataset_ds.push_to_hub("rubrix/sentiment-banking")
```

After pushing the dataset to the hub, you can simply retrieve it via `load_dataset` and `rb.read_datasets`.

```
[ ]: dataset_ds = load_dataset("rubrix/sentiment-banking", split="train")
dataset_rb = rb.read_datasets(dataset_ds, task="TextClassification")
```

Log

Now let's log the records to Rubrix to explore the dataset and label our first training set.

```
[ ]: rb.log(name='labeling_with_pretrained', records=dataset_1)
```

2. Explore and label data with the pretrained model

In this step, we'll start by exploring how the pre-trained model is performing with our dataset.

At first sight:

- The pre-trained sentiment classifier tends to label most of the examples as **NEGATIVE** (4.835 of 5.001 records). You can see this yourself using the `Predictions / Predicted as:` filter
- Using this filter and filtering by predicted as **POSITIVE**, we see that examples like *"I didn't withdraw the amount of cash that is showing up in the app."* are not predicted as expected (according to our basic "annotation policy" described in the preliminaries).

Taking into account this analysis, we can start labeling our data.

Rubrix provides you with a search-driven UI to annotated data, using **free-text search**, **search filters** and **the Elasticsearch query DSL** for advanced queries. This is especially useful for sparse datasets, tasks with a high number of labels, or unbalanced classes. In the standard case, we recommend you to follow the workflow below:

1. **Start labeling examples sequentially**, without using search features. This way you will annotate a fraction of your data which will be aligned with the dataset distribution.
2. Once you have a sense of the data, you can **start using filters and search features to annotate examples with specific labels**. In our case, we'll label examples predicted as **POSITIVE** by our pre-trained model, and then a few examples predicted as **NEGATIVE**.

Labeling random examples

Labeling POSITIVE examples

After some minutes, we've labelled almost **5% of our raw dataset with more than 200 annotated examples**, which is a small dataset but should be enough for a first fine-tuning of our banking sentiment classifier:

3. Fine-tune the pre-trained model

In this step, we'll load our training set from Rubrix and fine-tune using the `Trainer` API from Hugging Face `transformers`. For this, we closely follow the guide [Fine-tuning a pre-trained model](#) from the `transformers` docs.

First, let's load the annotations from our dataset using the `query` parameter from the `load` method. The `Validated` status corresponds to annotated records.

```
[47]: rb_dataset = rb.load(name='labeling_with_pretrained', query="status:Validated")
      rb_dataset.to_pandas().head(3)
```

```
[47]:          inputs \
0      {'text': 'I would like to cancel a purchase.'}
1      {'text': 'What's up with the extra fee I got?'}
```

(continues on next page)

(continued from previous page)

```

2 {'text': 'Do you have an age requirement when ...

                                prediction \
0 [(NEGATIVE, 0.9997695088386536), (POSITIVE, 0...
1 [(NEGATIVE, 0.9968097805976868), (POSITIVE, 0...
2 [(NEGATIVE, 0.9825802445411682), (POSITIVE, 0...

                                prediction_agent annotation \
0 distilbert-base-uncased-finetuned-sst-2-english POSITIVE
1 distilbert-base-uncased-finetuned-sst-2-english NEGATIVE
2 distilbert-base-uncased-finetuned-sst-2-english POSITIVE

annotation_agent multi_label explanation \
0 rubrix False None
1 rubrix False None
2 rubrix False None

                                id metadata status \
0 0002cbd9-b687-462a-bbd2-3130f4c88d8d {'category': 52} Validated
1 0009f445-4844-4ccd-9ea8-207a1fb0e239 {'category': 19} Validated
2 0012e385-643c-4660-ad66-5b4339bb3999 {'category': 1} Validated

event_timestamp metrics search_keywords
0 None None None
1 None None None
2 None None None

```

Prepare training and test datasets

Let's now prepare our dataset for training and testing our sentiment classifier, using the datasets library:

```

[ ]: # create dataset with labels as numeric ids
train_ds = rh.dataset.prepare_for_training()

[ ]: from transformers import AutoTokenizer

# tokenize our datasets
tokenizer = AutoTokenizer.from_pretrained("distilbert-base-uncased-finetuned-sst-2-
→english")

def tokenize_function(examples):
    return tokenizer(examples["text"], padding="max_length", truncation=True)

tokenized_train_ds = train_ds.map(tokenize_function, batched=True)

[ ]: # split the data into a training and evaluation set
train_dataset, eval_dataset = tokenized_train_ds.train_test_split(test_size=0.2,
→seed=42).values()

```


Train our sentiment classifier

As we mentioned before, we're going to fine-tune the `distilbert-base-uncased-finetuned-sst-2-english` model. Another option will be fine-tuning a **distilbert masked language model** from scratch, but we leave this experiment to you.

Let's load the model:

```
[ ]: from transformers import AutoModelForSequenceClassification

model = AutoModelForSequenceClassification.from_pretrained("distilbert-base-uncased-
↳ finetuned-sst-2-english")
```

Let's configure the **Trainer**:

```
[ ]: import numpy as np
from transformers import Trainer
from datasets import load_metric
from transformers import TrainingArguments

training_args = TrainingArguments(
    "distilbert-base-uncased-sentiment-banking",
    evaluation_strategy="epoch",
    logging_step=30,
)

metric = load_metric("accuracy")

def compute_metrics(eval_pred):
    logits, labels = eval_pred
    predictions = np.argmax(logits, axis=-1)
    return metric.compute(predictions=predictions, references=labels)

trainer = Trainer(
    args=training_args,
    model=model,
    train_dataset=train_dataset,
    eval_dataset=eval_dataset,
    compute_metrics=compute_metrics,
)
```

And finally, we can train our first model!

```
[ ]: trainer.train()
```

4. Testing the fine-tuned model

In this step, let's first test the model we have just trained.

Let's create a new pipeline with our model:

```
[ ]: finetuned_sentiment_classifier = pipeline(  
    model=model.to("cpu"),  
    tokenizer=tokenizer,  
    task="sentiment-analysis",  
    return_all_scores=True  
)
```

Then, we can compare its predictions with the pre-trained model and an example:

```
[ ]: finetuned_sentiment_classifier(  
    'I need to deposit my virtual card, how do i do that.'  
) , sentiment_classifier(  
    'I need to deposit my virtual card, how do i do that.'  
)
```

As you can see, our fine-tuned model now classifies this general questions (not related to issues or problems) as POSITIVE, while the pre-trained model still classifies this as NEGATIVE.

Let's check now an example related to an issue where both models work as expected:

```
[ ]: finetuned_sentiment_classifier(  
    'Why is my payment still pending?'  
) , sentiment_classifier(  
    'Why is my payment still pending?'  
)
```

5. Run our fine-tuned model over the dataset and log the predictions

Let's now create a dataset from the remaining records (those which we haven't annotated in the first annotation session).

We'll do this using the Default status, which means the record hasn't been assigned a label.

```
[ ]: rb_dataset = rb.load(name='labeling_with_pretrained', query="status:Default")
```

From here, this is basically the same as step 1, in this case using our fine-tuned model:

Let's take advantage of the datasets map feature, to make batched predictions.

```
[ ]: def predict(examples):  
    texts = [example["text"] for example in examples["inputs"]]  
    return {  
        "prediction": finetuned_sentiment_classifier(texts),  
        "prediction_agent": ["distilbert-base-uncased-banking77-sentiment"]*len(texts)  
    }  
  
ds_dataset = rb_dataset.to_datasets().map(predict, batched=True, batch_size=8)
```

Afterward, we can convert the dataset directly to Rubrix records again and log them to the web app.

```
[ ]: records = rh.read_datasets(ds_dataset, tool="TextClassification")

rh.log(records=records, name='labeling_with_finetuned')
```

6. Explore and label data with the fine-tuned model

In this step, we'll start by exploring how the fine-tuned model is performing with our dataset.

At first sight, using the predicted as filter by POSITIVE and then by NEGATIVE, we can observe that the fine-tuned model predictions are more aligned with our "annotation policy".

Now that the model is performing better for our use case, we'll extend our training set with highly informative examples. A typical workflow for doing this is as follows:

1. Use the prediction score filter for labeling uncertain examples.
2. Label examples predicted by our fine-tuned model as POSITIVE and then predicted as NEGATIVE to correct the predictions.

After spending some minutes, we labelled almost **2% of our raw dataset with around 80 annotated examples**, which is a small dataset but hopefully with highly informative examples.

7. Fine-tuning with the extended training dataset

In this step, we'll add the new examples to our training set and fine-tune a new version of our banking sentiment classifier.

Adding labeled examples to our previous training set

Let's add our new examples to our previous training set.

```
[ ]: rh_dataset = rh.load("labeling_with_finetuned")

train_ds = rh_dataset.prepare_for_training()
tokenized_train_ds = train_ds.map(tokenize_function, batched=True)

[ ]: from datasets import concatenate_datasets

train_dataset = concatenate_datasets([train_dataset, tokenized_train_ds])
```

Training our sentiment classifier

As we want to measure the effect of adding examples to our training set we will:

- Fine-tune from the pre-trained sentiment weights (as we did before)
- Use the previous test set and the extended train set (obtaining a metric we use to compare this new version with our previous model)

```
[ ]: from transformers import AutoModelForSequenceClassification
model = AutoModelForSequenceClassification.from_pretrained("distilbert-base-uncased-
↳ finetuned-sst-2-english")
```

```
[ ]: train_ds = train_dataset.shuffle(seed=42)

trainer = Trainer(
    args=training_args,
    model=model,
    train_dataset=train_dataset,
    eval_dataset=eval_dataset,
    compute_metrics=compute_metrics,
)

trainer.train()

[ ]: model.save_pretrained("distilbert-base-uncased-sentiment-banking")
```

Summary

In this tutorial, you learned how to build a training set from scratch with the help of a pre-trained model, performing two iterations of `predict > log > label`.

Although this is somehow a toy example, you will be able to apply this workflow to your own projects to adapt existing models or building them from scratch.

In this tutorial, we've covered one way of building training sets: **hand labeling**. If you are interested in other methods, which could be combined with hand labeling, checkout the following:

- *Building a news classifier with weak supervision*
- *Active learning with ModAL and scikit-learn*

Next steps

Star Rubrix [Github repo](#) to stay updated.

[Rubrix documentation](#) for more guides and tutorials.

Join the Rubrix community! A good place to start is the [discussion forum](#).

4.14.2 Building a news classifier with weak supervision

In this tutorial, we will build a news classifier using rules and weak supervision:

- For this example, we use the AG News dataset but you can follow this process to programmatically label any dataset.
- The train split without labels is used to build a training set with rules, Rubrix and Snorkel's Label model.
- The test set is used for evaluating our weak labels, label model and downstream news classifier.
- We achieve 0.82 macro avg. f1-score without using a single example from the original dataset and using a pretty lightweight model (scikit-learn's `MultinomialNB`).

The following diagram shows the overall process for using Weak supervision with Rubrix:

Introduction

Weak supervision is a branch of machine learning where noisy, limited, or imprecise sources are used to provide supervision signal for labeling large amounts of training data in a supervised learning setting. This approach alleviates the burden of obtaining hand-labeled data sets, which can be costly or impractical. Instead, inexpensive weak labels are employed with the understanding that they are imperfect, but can nonetheless be used to create a strong predictive model. [\[Wikipedia\]](#)

For a broader introduction to weak supervision, as well as further references, we recommend the excellent [overview](#) by Alex Ratner et al..

This tutorial aims to be a practical introduction to weak supervision and will walk you through its entire process. First we will generate weak labels with *Rubrix*, combine these labels with *Snorkel*, and finally train a classifier with *Scikit Learn*.

Setup

Rubrix, is a free and open-source tool to explore, annotate, and monitor data for NLP projects.

If you are new to Rubrix, check out the [Github repository](#).

If you have not installed and launched Rubrix yet, check the [Setup and Installation guide](#).

For this tutorial we also need some third party libraries that can be installed via pip:

```
[ ]: %pip install snorkel datasets sklearn -qqq
```

Note

If you want to skip the first three sections of this tutorial, and only prepare the training set and train a downstream model, you can load the records directly from the [Hugging Face Hub](#):

```
import rubrix as rb
from datasets import load_dataset

# this replaces the `records = label_model.predict()` line of section 4
records = rb.read_datasets(
    load_dataset("rubrix/news", split="train"),
    task="TextClassification",
)
```

1. Load test and unlabelled datasets into Rubrix

First, let's download the `ag_news` data set and have a quick look at it.

```
[ ]: from datasets import load_dataset

# load our data
dataset = load_dataset("ag_news")

# get the index to label mapping
labels = dataset["test"].features["label"].names
```

```
[10]: import pandas as pd

# quick look at our data
with pd.option_context('display.max_colwidth', None):
    display(dataset["test"].to_pandas().head())
```

	text \	label
0	Fears for T... N pension after talks Unions representing workers at Turner Newall say they are 'disappointed' after talks with stricken parent firm Federal Mogul.	2
1	The Race is On: Second Private Team Sets Launch Date for Human Spaceflight (SPACE. com) SPACE.com - TORONTO, Canada -- A second\team of rocketeers competing for the #36; 10 million Ansari X Prize, a contest for\privately funded suborbital space flight, has officially announced the first\launch date for its manned rocket.	3
2	Ky. Company Wins Grant... to Study Peptides (AP) AP - A company founded by a chemistry researcher at the University of Louisville won a grant to develop a method of producing better peptides, which are short chains of amino acids, the building blocks of proteins.	3
3	Prediction Unit Helps Forecast Wildfires (AP) AP - It's barely dawn when Mike Fitzpatrick starts his shift with a blur of colorful maps, figures and endless charts, but already he knows what the day will bring. Lightning will strike in places he expects. Winds will pick up, moist places will dry and flames will roar.	3
4	Calif. Aims to Limit Farm-Related Smog (AP) AP - Southern California's smog-fighting agency went after emissions of the bovine variety Friday, adopting the nation's first rules to reduce air pollution from dairy cow manure.	3

Now we will log the test split of our data set to *Rubrix*, which we will be using for testing our label and downstream models.

```
[ ]: import rubrix as rb

# build our test records
records = [
    rb.TextClassificationRecord(
        text=record["text"],
        metadata={"split": "test"},
        annotation=labels[record["label"]]
    )
    for record in dataset["test"]
]
```

(continues on next page)

(continued from previous page)

```
# log the records to Rubrix
rb.log(records, name="news")
```

In a second step we log the train split without labels. Remember, our goal is to programmatically build a training set using rules and weak supervision.

```
[ ]: # build our training records without labels
records = [
    rb.TextClassificationRecord(
        text=record["text"],
        metadata={"split": "unlabelled"},
    )
    for record in dataset["train"]
]

# log the records to Rubrix
rb.log(records, name="news")
```

The result of the above is the following dataset in Rubrix, with **127,600 records** (120,000 unlabelled and 7,600 for testing).

You can use the web app to find good rules for programmatic labeling!

2. Interactive weak labeling: Finding and defining rules

After logging the dataset, you can find and save rules directly with the UI. Then, you can read the rules with Python to train a label or downstream model, as we'll see in the next step.

3. Denoise weak labels with Snorkel's Label Model

The goal at this step is to **denoise** the weak labels we've just created using rules. There are several approaches to this problem using different statistical methods.

In this tutorial, we're going to use Snorkel but you can actually use any other Label model or weak supervision method, such as FlyingSquid for example (see the [Weak supervision guide](#) for more details). For convenience, Rubrix defines a simple wrapper over Snorkel's Label Model so it's easier to use with Rubrix weak labels and datasets

Let's first read the rules defined in our dataset and create our weak labels:

```
[ ]: from rubrix.labeling.text_classification import WeakLabels

weak_labels = WeakLabels(dataset="news")
```

```
[22]: weak_labels.summary()
```

```
[22]:
```

	label	coverage	annotated_coverage \
money	{Business}	0.008276	0.008816
financ*	{Business}	0.019655	0.017763
dollar*	{Business}	0.016591	0.016316
war	{World}	0.011779	0.013289
gov*	{World}	0.045078	0.045263
minister*	{World}	0.030031	0.028289
conflict	{World}	0.003041	0.002895

(continues on next page)

(continued from previous page)

football*	{Sports}	0.013166	0.015000
sport*	{Sports}	0.021191	0.021316
game	{Sports}	0.038879	0.037763
play*	{Sports}	0.052453	0.050000
sci*	{Sci/Tech}	0.016552	0.018421
techno*	{Sci/Tech}	0.027218	0.028289
computer*	{Sci/Tech}	0.027320	0.028026
software	{Sci/Tech}	0.030243	0.029605
web	{Sci/Tech}	0.015376	0.013289
total	{World, Sports, Business, Sci/Tech}	0.317022	0.311447

	overlaps	conflicts	correct	incorrect	precision
money	0.002437	0.001936	30	37	0.447761
financ*	0.005893	0.005188	80	55	0.592593
dollar*	0.003542	0.002908	87	37	0.701613
war	0.003213	0.001348	75	26	0.742574
gov*	0.010878	0.006270	170	174	0.494186
minister*	0.007531	0.002821	193	22	0.897674
conflict	0.001003	0.000102	18	4	0.818182
football*	0.004945	0.000439	107	7	0.938596
sport*	0.007045	0.001223	139	23	0.858025
game	0.014083	0.002375	216	71	0.752613
play*	0.016889	0.005063	268	112	0.705263
sci*	0.002735	0.001309	114	26	0.814286
techno*	0.008433	0.003174	155	60	0.720930
computer*	0.011058	0.004459	159	54	0.746479
software	0.009655	0.003346	184	41	0.817778
web	0.004067	0.001607	76	25	0.752475
total	0.053582	0.019561	2071	774	0.727944

```
[ ]: from rubrix.labeling.text_classification import Snorkel
```

```
# create the label model
label_model = Snorkel(weak_label)

# fit the model
label_model.fit()
```

```
[28]: print(label_model.score(output_sci=True))
```

	precision	recall	f1-score	support
Sports	0.79	0.96	0.87	632
Sci/Tech	0.77	0.77	0.77	773
World	0.70	0.80	0.74	509
Business	0.65	0.36	0.46	453
accuracy			0.75	2367
macro avg	0.73	0.72	0.71	2367
weighted avg	0.74	0.75	0.73	2367

4. Prepare our training set

Now, we already have a “denoised” training set, which we can prepare for training a downstream model. The label model predict returns TextClassificationRecord objects with the predictions from the label model.

We can either refine and review these records using the Rubrix web app, use them as is, or filter them by score, for example.

In this case, we assume the predictions are precise enough and use them without any revision. Our training set has ~38,000 records, which corresponds to all records where the label model has not abstained.

```
[ ]: import pandas as pd

# get records with the predictions from the label model
records = label_model.predict()
# you can replace this line with
# records = rb.read_datasets(
#     load_dataset("rubrix/news", split="train"),
#     task="TextClassification",
# )

# we could also use the `weak_labels.label2int` dict
label2int = {'Sports': 0, 'Sci/Tech': 1, 'World': 2, 'Business': 3}

# extract training data
x_train = [rec.text for rec in records]
y_train = [label2int[rec.prediction[0][0]] for rec in records]
```

```
[16]: # quick look at our training data with the weak labels from our label model
with pd.option_context('display.max_colwidth', None):
    display(pd.DataFrame({"text": x_train, "label": y_train}))
```

```

→
→
→
→
text label
0
→FA Cup: Third round draw - joy for Yeading and Exeter The great big fat bullies from
→across the playground enter the FA Cup arena at the third round stage, for which the
→draw was made yesterday (December 5). 0
1
→ Rats May Help Unravel Human Drug Addiction Mysteries By LAURAN NEERGAARD
→WASHINGTON (AP) -- Rats can become drug addicts. That's important to know, scientists
→say, and has taken a long time to prove... 1
2
→Palmer Passes Test Bengals quarterback Carson Palmer enjoyed his breakthrough game at
→the expense of the Super Bowl champion Patriots, racking up 179 yards on 12-of-19
→passing in a 31-3 triumph on Saturday night. 0
3
→Compromises urged amid deadlock in Darfur talks
→ABUJA, Nigeria -- Peace talks on Sudan's violence-torn Darfur region are deadlocked, a
→mediator said yesterday, as the chief of the African Union appealed to the Sudanese
→government and rebels to compromise. 2
4
→CAPELLO FED UP WITH
→FEIGNING Juventus coach Fabio Capello has ordered his players not to kick the ball out
→of play when an opponent falls to the ground apparently hurt because he believes some
→players fake injury to stop the match. 0
(continues on next page)
```

(continued from previous page)

```

...
↪
↪
↪
...
38080
↪ Apple ships Mac OS X update The 26MB upgrade
↪ to version 10.3.6 is available now via the OS #39; Software Update control panel and
↪ from Apple #39;s support web site. 1
38081 Bob Evans, mainframe pioneer, dies at 77 Bob Evans, an IBM computer scientist who
↪ helped to develop the modern mainframe computer, died Thursday. He was 77. Evans died
↪ of heart failure at his at his home in the San Francisco suburb of Hillsborough, his
↪ son Evan told the Associated Press. 1
38082
↪ For Brazil's Economy, the Doctor Is In Antonio Palocci, a doctor from Brazil's farm
↪ belt, has found himself presiding as the country's finance minister during the most
↪ robust economic expansion in a decade. 2
38083 UbiSoft Get Ready With The Next
↪ Rainbox Six Installment Ubisoft today announced its plans to launch the next
↪ installment in the Tom Clancys Rainbow Six franchise in Spring 2005. The next Rainbow
↪ Six game follows Team Rainbow, the worlds most 0
38084 PM #39;s visit to focus on reconstruction of Kashmir New Delhi: The two-
↪ day visit of Prime Minister Manmohan Singh to Jammu and Kashmir, starting on Wednesday
↪ will focus more on reconstruction and development of the state, Parliamentary Affairs
↪ Minister Ghulam Nabi Azad has said. 2

[38085 rows x 2 columns]

```

5. Train a downstream model with scikit-learn

Now, let's train our final model using scikit-learn:

```

[ ]: from sklearn.feature_extraction.text import TfidfTransformer, CountVecorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.pipeline import Pipeline

# define our final classifier
classifier = Pipeline([
    ('vect', CountVecorizer()),
    ('clf', MultinomialNB())
])

# fit the classifier
classifier.fit(
    X=train,
    y=train,
)

```

To test our trained model, we use the records with validated annotations, that is the original ag_news test set.

```

[ ]: # retrieve records with annotations
test_ds = weak_labels.records(has_annotation=True)

```

(continues on next page)

(continued from previous page)

```
# you can replace this line with
# test_ds = rb.read_datasets(
#     load_dataset("rubrix/news_test", split="train"),
#     task="TextClassification",
# )

# extract text and labels
X_test = [rec.text for rec in test_ds]
y_test = [label2int[rec.annotation] for rec in test_ds]
```

```
[77]: # compute the test accuracy
accuracy = classifier.score(
    X=X_test,
    y=y_test,
)

print(f"Test accuracy: {accuracy}")

Test accuracy: 0.8182894736842106
```

Not too bad!

We have achieved around **0.82 accuracy** without even using a single example from the original `ag_news` train set and with a small set of 16 rules. Also, we've improved over the 0.75 accuracy of our Label Model.

Finally, let's take a look at more detailed metrics:

```
[76]: from sklearn import metrics

# get predictions for the test set
predicted = classifier.predict(X_test)

print(metrics.classification_report(X_test, predicted, target_names=label2int.keys()))
```

	precision	recall	f1-score	support
Sports	0.86	0.98	0.91	1900
Sci/Tech	0.76	0.84	0.80	1900
World	0.80	0.89	0.84	1900
Business	0.88	0.57	0.69	1900
accuracy			0.82	7600
macro avg	0.82	0.82	0.81	7600
weighted avg	0.82	0.82	0.81	7600

At this point, we could go back to the UI to define more rules for those labels with less performance. Looking at the above table, we might want to add some more rules for increasing the recall of the **Business** label.

Summary

In this tutorial, we saw how you can leverage weak supervision to quickly build up a large training data set, and use it for the training of a first lightweight model.

Rubrix is a very handy tool to start the weak supervision process by making it easy to find a good set of starting rules, and to iterate on them dynamically. Since *Rubrix* also provides built-in support for the most common label models, you can get from rules to weak labels in a few straight forward steps. For more suggestions on how to leverage weak labels, you can checkout our [weak supervision guide](#) where we describe an *interesting approach* to jointly train the label and a transformers downstream model.

Next steps

If you are interested in the topic of weak supervision check our [weak supervision guide](#).

Rubrix [Github repo](#) to stay updated.

[Rubrix documentation](#) for more guides and tutorials.

Join the Rubrix community on [Slack](#)

Appendix I: Create rules and weak labels from Python

For some use cases, you might want to use Python for defining labeling rules and generating weak labels. Rubrix provides you with the ability to define and test rules and labeling functions directly using Python. This might be useful for combining it with rules defined in the UI, and for leveraging structured resources such as lexicons and gazeteers which are easier to use directly a programmatic environment.

In this section, we define the rules we've defined in the UI, this time directly using Python:

```
[ ]: from rubrix.labeling.text_classification import Rule

# define queries and patterns for each category (using ES DSL)
queries = [
    (["money", "financ*", "dollar*"], "Business"),
    (["war", "gov*", "minister*", "conflict"], "World"),
    (["footbal*", "sport*", "game", "play*"], "Sports"),
    (["sci*", "techno*", "computer*", "software", "web"], "Sci/Tech")
]

# define rules
rules = [
    Rule(query=term, label=label)
    for term, label in queries
    for term in term
]
```

```
[ ]: from rubrix.labeling.text_classification import WeakLabels

# generate the weak labels
weak_labels = WeakLabels(
    rules=rules,
    dataset="news"
)
```

On our machine it took around 24 seconds to apply the rules and to generate weak labels for the 127,600 examples.

Typically, you want to iterate on the rules and check their statistics. For this, you can use `weak_labels.summary` method:

[78]: `weak_labels.summary` 

```
[78]:
```

	label	coverage	annotated_coverage	\
money	{Business}	0.008276	0.008816	
financ*	{Business}	0.019655	0.017763	
dollar*	{Business}	0.016591	0.016316	
war	{World}	0.011779	0.013289	
gov*	{World}	0.045078	0.045263	
minister*	{World}	0.030031	0.028289	
conflict	{World}	0.003041	0.002895	
football*	{Sports}	0.013166	0.015000	
sport*	{Sports}	0.021191	0.021316	
game	{Sports}	0.038879	0.037763	
play*	{Sports}	0.052453	0.050000	
sci*	{Sci/Tech}	0.016552	0.018421	
techno*	{Sci/Tech}	0.027218	0.028289	
computer*	{Sci/Tech}	0.027320	0.028026	
software	{Sci/Tech}	0.030243	0.029605	
web	{Sci/Tech}	0.015376	0.013289	
total	{World, Sports, Business, Sci/Tech}	0.317022	0.311447	

	overlaps	conflicts	correct	incorrect	precision
money	0.002437	0.001936	30	37	0.447761
financ*	0.005893	0.005188	80	55	0.592593
dollar*	0.003542	0.002908	87	37	0.701613
war	0.003213	0.001348	75	26	0.742574
gov*	0.010878	0.006270	170	174	0.494186
minister*	0.007531	0.002821	193	22	0.897674
conflict	0.001003	0.000102	18	4	0.818182
football*	0.004945	0.000439	107	7	0.938596
sport*	0.007045	0.001223	139	23	0.858025
game	0.014083	0.002375	216	71	0.752613
play*	0.016889	0.005063	268	112	0.705263
sci*	0.002735	0.001309	114	26	0.814286
techno*	0.008433	0.003174	155	60	0.720930
computer*	0.011058	0.004459	159	54	0.746479
software	0.009655	0.003346	184	41	0.817778
web	0.004067	0.001607	76	25	0.752475
total	0.053582	0.019561	2071	774	0.727944

From the above, we see that our rules cover around **30% of the original training set** with an **average precision of 0.73**. Our hope is that the label and downstream models will improve both the recall and the precision of the final classifier.

Appendix II: Log datasets to the Hugging Face Hub

Here we will show you how we pushed our Rubrix datasets (records) to the [Hugging Face Hub](#). In this way you can effectively version any of your Rubrix datasets.

```
[ ]: train_rb = rb.DatasetForTextClassification(label_model.predict())
train_rb.to_dataset().push_to_hub("rubrix/news")
```

```
[ ]: test_rb = rb.load("news", query="status:Validated")
test_rb.to_dataset().push_to_hub("rubrix/news_test")
```

4.14.3 Few-shot classification with SetFit and a custom dataset

SetFit is an exciting open-source package for few-shot classification developed by teams at Hugging Face and Intel Labs. You can read all about it on the [project repository](#).

To showcase how powerful is **the combination of SetFit and Rubrix**:

- We manually **label 55 examples** from the unlabelled split of the imdb dataset,
- we train a model in **5 min**,
- and without using a single example from the original imdb training set, we achieve **0.9 accuracy on the full test set!**

Summary

In this tutorial, you'll learn to:

1. **Load a unlabelled dataset** in Rubrix. We'll be using the unlabelled split from the imdb movie reviews sentiment dataset. This same workflow can be applied to any custom dataset, problem, and language!
2. Manually **label a FEW examples** using the UI.
3. **Train a SetFit model** to get highly competitive results. For this example, with **only 55 examples**, we get **0.9 accuracy** on the test set which is comparable to models fine-tuned on 3K examples. That means similar performance with 50x less examples .

For reference see the [Hugging Face Hub](#) and [PapersWithCode](#) leaderboards.

Let's get started!

Setup Rubrix

Rubrix is a free and **open-source data labeling framework for NLP**.

To get started on your local machine, you just need three steps:

1. Install the library:

```
[19]: !pip install rubrix[server]
```

2. Install and launch [Elasticsearch](#).
3. Launch the server and the UI from your terminal or notebook:

```
python -m rubrix
```

If everything went well, you can go to <https://localhost:6900> and login using the default user/password: rubrix/1234.

If you need help you can join our [Slack channel](#) to get immediate support.

Setup SetFit and datasets libraries

```
[ ]: !pip install setfit datasets -qqq

[ ]: from datasets import load_dataset
    from sentence_transformers.losses import CosineSimilarityLoss

    from setfit import SetFitModel, SetFitTrainer

    import rubrix as rb
```

Load unlabelled dataset in Rubrix

First, we load the unsupervised split from the imdb dataset and create a new Rubrix dataset with 100 random examples:

```
[ ]: unlabelled = load_dataset("imdb", split="unsupervised").shuffle(seed=42).
    ↪ select(range(100))

    unlabelled = rb.DatasetForTextClassification.from_datasets(unlabelled)

    rb.log(unlabelled, "imdb_unlabelled")
```

Manual labelling

In this step, we create the labels pos and neg using the same label scheme as the original dataset. Then we use the UI to sequentially label a few examples. For the example, we spent literally 15 minutes.

Watch the video below to get a sense of the steps and time you need to replicate the results.

Before training, you can easily share the dataset using the `push_to_hub` method. This might be useful if you don't have a GPU on your machine and want to use a training service or Colab for example.

```
[ ]: rb.load("imdb_unlabelled").prepare_for_training().push_to_hub("mini-imdb")
```

The dataset is available on the [HF hub](#). You can see the summary in the UI below:

The screenshot shows the Rubrix dataset interface. At the top, there's a header with 'Datasets / rubrix / imdb_unlabelled'. Below this, there's a search bar and tabs for 'Annotations', 'Status', and 'Sort'. A 'Records (100)' indicator is present. On the right, a 'Progress' section shows 'Total 55/100' and '55.00%'. Below the progress bar, there's a legend for 'Validated' (blue dot) and 'Discarded' (grey dot). The main area displays two movie reviews with their full text and a 'Show full record' link. At the bottom, there's a 'Records per page' dropdown set to 5, and a pagination bar showing '1 2 3 ... 20 Next' and '1-5 of 100'.

Train and evaluate SetFit model

Finally, we are ready to test SetFit!

Thanks to Rubrix's integration with datasets and the Hub, if you don't have a local GPU you can use this [Google Colab](#) to reproduce the training process with the labelled dataset. If you use a GPU runtime, it literally takes 5 minutes to train.

Below we load the dataset from Rubrix, format it for training with transformers, load the full imdb test dataset, load a pre-trained sentence transformers model, train the SetFit model, and evaluate it!

```
[ ]: # Load the handlabelled dataset from Rubrix
train_ds = rb.load("imdb_unlabelled").prepare_for_training()

# Load the full imdb test dataset
test_ds = load_dataset("imdb", split="test")

# Load SetFit model from Hub
model = SetFitModel.from_pretrained("sentence-transformers/paraphrase-mpnet-base-v2")

# Create trainer
trainer = SetFitTrainer(
    model=model,
    train_dataset=train_ds,
    eval_dataset=test_ds,
    loss_class=CosineSimilarityLoss,
    batch_size=16,
    num_iteration=20, # The number of text pairs to generate
)

# Train and evaluate
```

(continues on next page)

(continued from previous page)

```
trainer.train()
metrics = trainer.evaluate()
```

Optionally, you can share your amazing model with the world!

```
[ ]: trainer.push_to_hub("setfit-mini-imdb")
```

Conclusion

The metrics object should give you around 0.9 accuracy on the full test set

And remember:

- We have manually labelled 55 examples,
- We haven't used a single example from the original training set,
- and we've trained the model in 5 min!

Now, I don't think you have any more excuses to not invest some time labeling a few good quality examples!

4.15 Model predictions

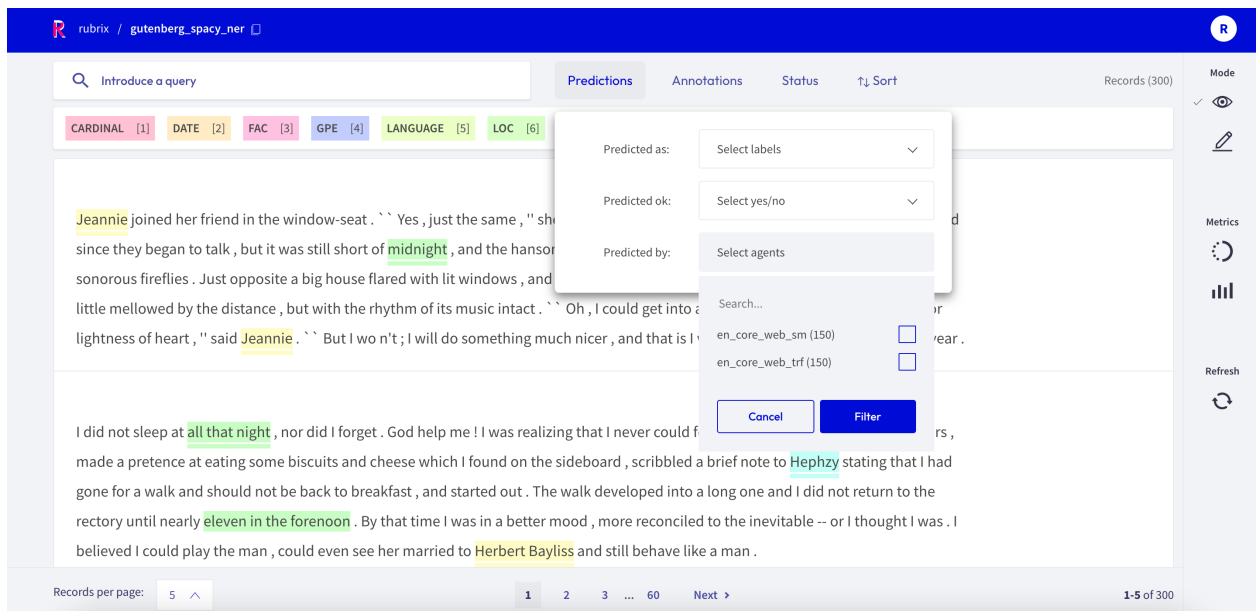
These tutorials show you how you can leverage model predictions to analyze and evaluate your model, or to pre-annotate your data in Rubrix.

The screenshot displays the Rubrix web interface for a sentiment classification task. The header shows the project name 'rubrix / labeling_with_pretrained_1'. The main table lists three text samples with their predicted sentiment and confidence scores. Each row has a 'Validate' button and a 'Discard' button. The right sidebar contains icons for 'Add new label', 'Metrics', and 'Refresh'.

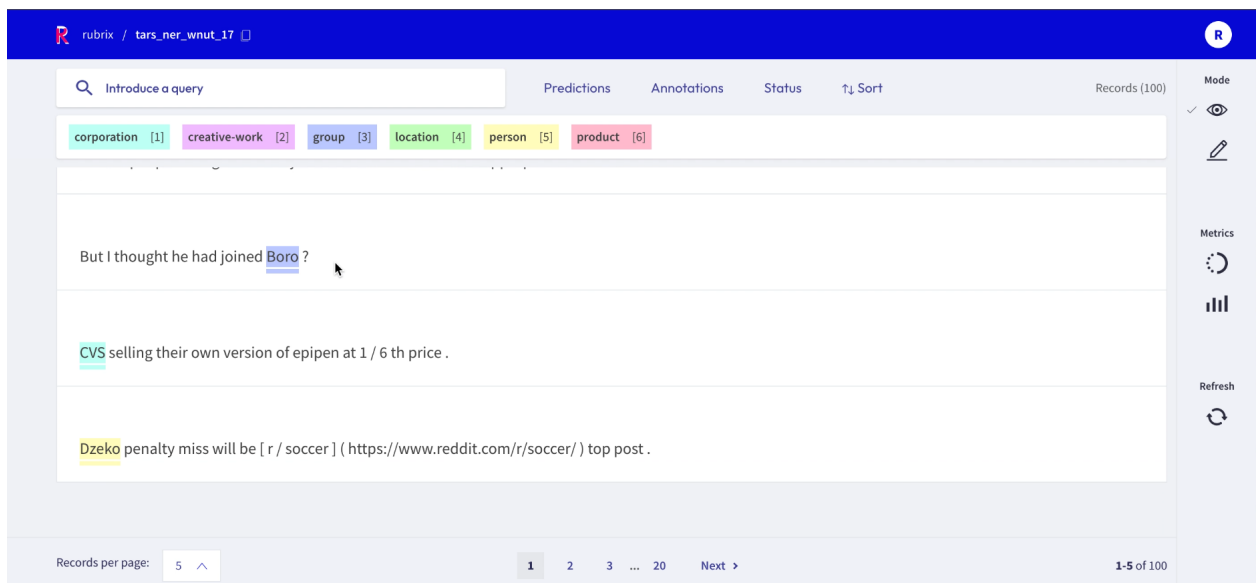
Text	Negative	Positive	Action
TEXT: I only received \$20, but I requested \$100.	NEGATIVE 99,052 %	POSITIVE 0,948 %	Validate, Discard, Add new label
TEXT: I am 16 and just started working at McDonalds can I deposit my checks at your bank?	NEGATIVE 99,888 %	POSITIVE 0,112 %	Validate, Discard, Add new label
TEXT: I tried activating my plug-in and it didn't piece of work	NEGATIVE 99,962 %	POSITIVE 0,038 %	Validate, Discard, Add new label

Records per page: 5 | 1 2 3 ... 1001 Next > | 1-5 of 5001

Label your data to fine-tune a classifier with Hugging Face Build a sentiment classifier for user requests in the banking domain by leveraging a pre-trained model from the Hugging Face Hub.



Explore and analyze spaCy NER pipelines Learn to log and analyze spaCy Name Entity Recognition (NER) predictions by exploring two well known datasets from the Hugging Face Hub.



Zero-shot Named Entity Recognition with Flair Learn how to analyze and validate NER predictions from Flair's zero-shot model using the WNUT 17 dataset from the Hugging Face Hub.

The screenshot shows the Rubrix web interface for the dataset 'rubrix / transformers_interpret_example'. The interface includes a search bar, tabs for 'Predictions (1)', 'Status', 'Annotations', and 'Sort (1)', and a 'Records (32)' indicator. On the right, there are icons for 'Mode', 'Help', 'Metrics', and 'Refresh'.

Three text samples are displayed, each with a 'NEGATIVE' sentiment score:

- TEXT:** i do n't know precisely what to make of steven so ##der ##berg ##h i s full frontal i though that did n't stop me from enjoying much of it
NEGATIVE 70.581%
- TEXT:** with the film i s striking ending , one realizes that we have a long way to go before we fully understand all the sexual per ##mut ##ations involved .
NEGATIVE 84.247%
- TEXT:** between the drama of cube
NEGATIVE 70.581%

At the bottom, there is a pagination bar showing 'Records per page: 5' and '1-5 of 32'.

Analyzing predictions with model explainability methods Learn to log and analyze model explanations using Transformers Interpret and Shap.

4.15.1 Explore and analyze spaCy NER pipelines

In this tutorial, we will learn to log `spaCy` Name Entity Recognition (NER) predictions.

This is useful for:

- Evaluating pre-trained models.
- Spotting frequent errors both during development and production.
- Improving your pipelines over time using Rubrix annotation mode.
- Monitoring your model predictions using Rubrix integration with Kibana

Let's get started!

Introduction

In this tutorial we will learn how to explore and analyze `spaCy` NER pipelines in an easy way.

We will load the `Gutenberg Time` dataset from the Hugging Face Hub and use a transformer-based `spaCy` model for detecting entities in this dataset and log the detected entities into a Rubrix dataset. This dataset can be used for exploring the quality of predictions and for creating a new training set, by correcting, adding and validating entities.

Then, we will use a smaller `spaCy` model for detecting entities and log the detected entities into the same Rubrix dataset for comparing its predictions with the previous model. And, as a bonus, we will use Rubrix and `spaCy` on a more challenging dataset: IMDB.

Setup

Rubrix is a free and open-source tool to explore, annotate, and monitor data for NLP projects.

If you are new to Rubrix, visit and star Rubrix for more materials like and detailed docs: [Github repo](#)

If you have not installed and launched Rubrix yet, check the [Setup and Installation guide](#).

For this tutorial we also need the third party libraries datasets and of course spaCy together with pytorch, which can be installed via git:

```
[ ]: %pip install torch -qqq
      %pip install datasets "spacy[transformers]~=3.0" protobuf -qqq
```

Note

If you want to skip running the spaCy pipelines, you can also load the resulting Rubrix records directly from the [Hugging Face Hub](#), and continue the tutorial logging them to the Rubrix web app. For example:

```
import rubrix as rb
from datasets import load_dataset

records = rb.read_datasets(
    load_dataset("rubrix/gutenberg_spacy_ner", split="train"),
    task="TokenClassification",
)
```

The Rubrix records of this tutorial are available under the names “*rubrix/gutenberg_spacy_ner*” and “*rubrix/imdb_spacy_ner*”.

Our dataset

For this tutorial, we’re going to use the [Gutenberg Time](#) dataset from the [Hugging Face Hub](#). It contains all explicit time references in a dataset of 52,183 novels whose full text is available via Project Gutenberg. From extracts of novels, we are surely going to find some NER entities.

```
[ ]: from datasets import load_dataset

dataset = load_dataset("gutenberg_time", split="train", streaming=True)
```

Let’s have a look at the first 5 examples of the train set.

```
[2]: import pandas as pd

      pd.DataFrame(dataset.take(5))
```

	guten_id	hour_reference	time_phrase	is_ambiguous	\
0	4447	5	five o'clock	True	
1	4447	12	the fall of the winter noon	True	
2	28999	12	midday	True	
3	28999	12	midday	True	
4	28999	0	midnight	True	
	time_pos_start	time_pos_end	\		

(continues on next page)

(continued from previous page)

0	145	147
1	68	74
2	46	47
3	133	134
4	43	44

	tok_context
0	I crossed the ground she had traversed , notin...
1	So profoundly penetrated with thoughtfulness w...
2	And here is Hendon , and it is time for us to ...
3	Sorrows and trials she had had in plenty in he...
4	Jeannie joined her friend in the window-seat ...

Logging spaCy NER entities into Rubrix

Using a Transformer-based pipeline

Let's download our Roberta-based pretrained pipeline and instantiate a spaCy nlp pipeline with it.

```
[ ]: !python -m spacy download en_core_web_trf
```

```
[ ]: import spacy

nlp = spacy.load("en_core_web_trf")
```

Now let's apply the nlp pipeline to the first 50 examples in our dataset, collecting the **tokens** and **NER entities**.

```
[ ]: import rubrix as rb
from tqdm.auto import tqdm

# Creating an empty record list to save all the records
records = []

# Iterate over the first 50 examples of the Gutenberg dataset
for record in tqdm(list(dataset.take(50))):

    # We only need the text of each instance
    text = record["tok_context"]

    # spaCy Doc creation
    doc = nlp(text)

    # Entity annotations
    entities = [
        (ent.label_, ent.start_char, ent.end_char)
        for ent in doc.ents
    ]

    # Pre-tokenized input text
    tokens = [token.text for token in doc]
```

(continues on next page)

(continued from previous page)

```
# Rubrix TokenClassificationRecord list
records.append(
    rb.TokenClassificationRecord(
        text=text,
        tokens=tokens,
        prediction=entities,
        prediction_agent="en_core_web_trf",
    )
)
```

```
[ ]: rb.log(records=records, name="gutenberg_spacy_ner")
```

If you go to the `gutenberg_spacy_ner` dataset in Rubrix you can explore the predictions of this model.

You can:

- Filter records containing specific entity types,
- See the most frequent “mentions” or surface forms for each entity. Mentions are the string values of specific entity types, such as for example “1 month” can be the mention of a duration entity. This is useful for error analysis, to quickly see potential issues and problematic entity types,
- Use the free-text search to find records containing specific words,
- And validate, include or reject specific entity annotations to build a new training set.

Using a smaller but more efficient pipeline

Now let’s compare with a smaller, but more efficient pre-trained model.

Let’s first download it:

```
[ ]: !python -m spacy download en_core_web_sm
```

```
[ ]: import spacy

nlp = spacy.load("en_core_web_sm")
```

```
[ ]: # Creating an empty record list to save all the records
records = []

# Iterate over the first 50 examples of the Gutenberg dataset
for record in tqdm(list(dataset.take(50))):

    # We only need the text of each instance
    text = record["tok_context"]

    # spaCy Doc creation
    doc = nlp(text)

    # Entity annotations
    entities = [
        (ent.label_, ent.start_char, ent.end_char)
```

(continues on next page)

(continued from previous page)

```

    for ent in doc.ents:
    ]

    # Pre-tokenized input text
    tokens = [token.text for token in doc]

    # Rubrix TokenClassificationRecord list
    records.append(
        rh.TokenClassificationRecord(
            text=text,
            token=tokens,
            prediction=entities,
            prediction_agent="en_core_web_sm",
        )
    )

```

```
[ ]: rh.log(records=records, name="guttenberg_spacy_ner")
```

Exploring and comparing en_core_web_sm and en_core_web_trf models

If you go to your `guttenberg_spacy_ner` dataset, you can explore and compare the results of both models.

To only see predictions of a specific model, you can use the `predicted by` filter, which comes from the `prediction_agent` parameter of your `TextClassificationRecord`.

The screenshot shows the Rubrix web interface for the `guttenberg_spacy_ner` dataset. The 'Predictions' tab is active, showing a list of records. A modal window is open, allowing the user to filter predictions by 'Predicted by' (agent). The modal shows two options: `en_core_web_sm (150)` and `en_core_web_trf (150)`. The `en_core_web_trf` option is selected. The background shows a sample text record with entity predictions highlighted in green.

Explore the IMDB dataset

So far, both **spaCy pretrained models** seem to work pretty well. Let's try with a more challenging dataset, which is more dissimilar to the original training data these models have been trained on.

```
[ ]: imdb = load_dataset("imdb", split="test")

[ ]: records = []
for record in tqdm(imdb.select(range(50))):
    # We only need the text of each instance
    text = record["text"]

    # spaCy Doc creation
    doc = nlp(text)

    # Entity annotations
    entities = [
        (ent.label_, ent.start_char, ent.end_char)
        for ent in doc.ents
    ]

    # Pre-tokenized input text
    tokens = [token.text for token in doc]

    # Rubrix TokenClassificationRecord list
    records.append(
        rh.TokenClassificationRecord(
            text=text,
            token=tokens,
            prediction=entities,
            prediction_agent="en_core_web_sm",
        )
    )

[ ]: rh.log(records=records, name="imdb_spacy_ner")
```

Exploring this dataset highlights **the need of fine-tuning for specific domains**.

For example, if we check the most frequent mentions for Person, we find two highly frequent missclassified entities: **gore** (the film genre) and **Oscar** (the prize).

You can easily check every example by using the filters and search-box.

Summary

In this tutorial, you learned how to log and explore different spaCy NER models with Rubrix. Now you can:

- Build custom dashboards using Kibana to monitor and visualize spaCy models.
- Build training sets using pre-trained spaCy models.

Next steps

Rubrix [documentation](#) for more guides and tutorials.

Join the Rubrix community! A good place to start is the [discussion forum](#).

Rubrix [Github repo](#) to stay updated.

Appendix: Log datasets to the Hugging Face Hub

Here we will show you an example of how you can push a Rubrix dataset (records) to the [Hugging Face Hub](#). In this way you can effectively version any of your Rubrix datasets.

```
[ ]: records = rt.load("gutenberg_spacy_ner")
records.to_dataset().push_to_hub("<name of the dataset on the HF Hub>")
```

4.15.2 Zero-shot Named Entity Recognition with Flair

In this tutorial you will learn how to analyze and validate NER predictions from the new zero-shot model provided by the Flair NLP library with Rubrix.

- Useful for quickly bootstrapping a training set (using Rubrix *Annotation Mode*) as well as integrating with weak-supervision workflows.
- We will use a challenging, exciting dataset: wnut_17 (more info below).
- You will be able to see and work with the obtained predictions.

Introduction

This tutorial will show you how to work with Named Entity Recognition (NER), Flair and Rubrix. But, what is NER?

According to [Analytics Vidhya](#), “NER is a **natural language processing technique** that can automatically scan entire articles and pull out some fundamental entities in a text and classify them into predefined categories”. These entities can be names, quantities, dates and times, amounts of money/currencies, and much more.

On the other hand, [Flair](#) is a library which facilitates the application of NLP models to NER and other NLP techniques in many different languages. It is not only a powerful library, but also intuitive.

Thanks to these resources and the *Annotation Mode* of *Rubrix*, we can quickly build up a data set to train a domain-specific model.

Setup

Rubrix, is a free and open-source tool to explore, annotate, and monitor data for NLP projects.

If you are new to Rubrix, check out the [Github repository](#) .

If you have not installed and launched Rubrix yet, check the *Setup and Installation guide*.

For this tutorial we also need the third party libraries datasets and flair, which can be installed via pip:

```
[ ]: %pip install datasets flair -qqq
```

1. Load the wnut_17 dataset

In this example, we'll use a challenging NER dataset, the “**WNUT 17: Emerging and Rare entity recognition**”, which focuses on unusual, previously-unseen entities in the context of emerging discussions. This dataset is useful for getting a sense of the quality of our zero-shot predictions.

Let's load the test set from the [Hugging Face Hub](#):

```
[ ]: from datasets import load_dataset

# download data set
dataset = load_dataset("wnut_17", split="test")

[2]: # define labels
labels = ['corporation', 'creative-work', 'group', 'location', 'person', 'product']
```

2. Configure Flair TARSTagger

Now let's configure our NER model, following [Flair's documentation](#):

```
[ ]: from flair.models import TARSTagger

# load zero-shot NER tagger
tars = TARSTagger.load('tars-ner')

# define labels for named entities using wnut labels
tars.add_and_switch_to_new_task('task 1', labels, label_type='ner')

Let's test it with one example!

[ ]: from flair.data import Sentence

# wrap our tokens in a flair Sentence
sentence = Sentence(" ".join(dataset[0]['tokens']))

[6]: # add predictions to our sentence
tars.predict(sentence)

# extract predicted entities into a list of tuples (entity, start_char, end_char)
[
    (entity.get_labels()[0].value, entity.start_pos, entity.end_pos)
    for entity in sentence.get_spans("ner")
]

[6]: [('location', 100, 107)]
```

3. Predict over wnut_17 and log into rubrix

Now, let's log the predictions in Rubrix:

```
[ ]: import rubrix as rb

# build records for the first 100 examples
records = []
for record in dataset.select(range(100)):
    input_text = " ".join(record["tokens"])

    sentence = Sentence(input_text)
    tars.predict(sentence)
    prediction = [
        (entity.get_label()[0].value, entity.start_pos, entity.end_pos)
        for entity in sentence.get_spans("ner")
    ]

    # building TokenClassificationRecord
    records.append(
        rb.tokenClassificationRecord(
            text=input_text,
            tokens=[token.text for token in sentence],
            prediction=prediction,
            prediction_agent="tars-ner",
        )
    )

# log the records to Rubrix
rb.log(records, name='tars_ner_wnut_17', metadata={"split": "test"})
```

Now you can see the results obtained! With the annotation mode, you can change, add, validate or discard your results. Statistics are also available, to better monitor your records!

Summary

Getting predictions with a zero-shot approach can be very helpful to guide humans in their annotation process. Especially for NER tasks, Rubrix makes it very easy to explore and correct those predictions thanks to its **Annotation Mode**.

Next steps

Star Rubrix Github repo to stay updated.

Rubrix documentation for more guides and tutorials.

Join the Rubrix community! A good place to start is the discussion forum.

```
[ ]:
```

4.15.3 Analyzing predictions with model explainability methods

In this tutorial you will learn to log and explore NLP model explanations using Transformers and the following libraries:

- Transformers Interpret
- Shap

Interpretability and explanation information in Rubrix is not limited to these two libraries. You can populate this information using your method of choice to highlight specific tokens.

This tutorial is useful to get started and understand the underlying structure of explanation information in Rubrix records.

Beyond browsing examples during model development and evaluation, storing explainability information in Rubrix can be really useful for monitoring and assessing production models (more tutorials on this soon!)

Let's get started!

Setup

Rubrix, is a free and open-source framework for data-centric NLP. If you are new to Rubrix, check out the [Github repository](#).

If you have not installed and launched Rubrix yet, check the *Setup and Installation guide*.

Token attributions and what do highlight colors mean?

Rubrix enables you to register token attributions as part of the dataset records. For getting token attributions, you can use methods such as Integrated Gradients or SHAP. These methods try to provide a mechanism to interpret model predictions. The attributions work as follows:

- **[0,1] Positive attributions (in blue)** reflect those tokens that are making the model predict the specific predicted label.
- **[-1, 0] Negative attributions (in red)** reflect those tokens that can influence the model to predict a label other than the specific predicted label.

Using Transformers Interpret

In this example, we will use the `sst` sentiment dataset and a `distilbert`-based sentiment classifier. For getting model explanation information, we will use the excellent [Transformers Interpret](#) library by [Charles Pierse](#).

Install dependencies

```
[ ]: !pip install transformers-interpret==0.5.2 datasets transformers
```

Create a fully searchable dataset with model predictions and explanations

```
[ ]: from transformers import AutoModelForSequenceClassification, AutoTokenizer
from transformers_interpret import SequenceClassificationExplainer
from datasets import load_dataset

import rubrix as rb
from rubrix import TokenAttributions

# Load Stanford sentiment treebank test set
dataset = load_dataset("sst", "default", split="test")

# Let's use a sentiment classifier fine-tuned on sst
model_name = "distilbert-base-uncased-finetuned-sst-2-english"
model = AutoModelForSequenceClassification.from_pretrained(model_name)
tokenizer = AutoTokenizer.from_pretrained(model_name)

# Define the explainer using transformers_interpret
cls_explainer = SequenceClassificationExplainer(model, tokenizer)

records = []
for example in dataset:

    # Build Token attributions objects
    word_attributions = cls_explainer(example["sentence"])
    token_attributions = [
        TokenAttributions(
            token=token,
            attributions={cls_explainer.predicted_class_name: score}
        ) # ignore first (CLS) and last (SEP) tokens
        for token, score in word_attributions[1:-1]
    ]
    # Build Text classification records
    record = rb.TextClassificationRecord(
        text=example["sentence"],
        prediction=[(cls_explainer.predicted_class_name, cls_explainer.pred_prob)],
        explanation={"text": token_attributions},
    )
    records.append(record)

# Build Rubrix dataset with interpretations for each record
rb.log(records, name="transformers_interpret_example")
```

Example: *Predicted as negative sorted by descending score*

If you go to http://localhost:6900/datasets/rubrix/transformers_interpret_example (assuming you are running Rubrix on your local machine you get a fully-searchable dataset). For example, let's drill down to look at examples predicted as negative with a low score:

Using Shap

In this example, we will use the widely-used [Shap](#) library by [Scott Lundberg](#).

Install dependencies

```
[ ]: !pip install shap==0.40.0 numba==0.53.1
```

Create a fully searchable dataset with model predictions and explanations

This example is very similar to the one above. The main difference is that we need to scale the values from Shap to match the range required by Rubrix UI. This restriction is for visualization purposes. If you are more interested in monitoring use cases you might not need to rescale.

```
[ ]: import transformers
from datasets import load_dataset

from sklearn.preprocessing import MinMaxScaler

import shap

from rubrix import TextClassificationRecord, TokenAttributions
```

(continues on next page)

(continued from previous page)

```

import rubrix as rb

# Transformers pipeline model
model = transformers.pipeline("sentiment-analysis", return_all_scores=True)

# Load Stanford treebank dataset only the first 5 records for testing
sst = load_dataset("sst", split="test[0:5]")

# Use shap's library text explainer
explainer = shap.Explainer(model)
shap_values = explainer(sst['sentence'])

# Instantiate the scaler
scaler = MinMaxScaler(feature_range=[-1, 1])

predictions = model(sst["sentence"])

for i in range(0, len(shap_values.values)):

    # Scale shap values between -1 and 1 (using e.g., scikit-learn MinMaxScaler)
    scaled = scaler.fit_transform(shap_values.values[i])

    # get prediction label idx for indexing attributions and shap_values
    # sorts by score to get the max score prediction
    sorted_predictions = sorted(predictions[i], key=lambda d: d["score"], reverse=True)
    label_idx = 0 if sorted_predictions[0]["label"] == "NEGATIVE" else 1

    # Build token attributions
    token_attributions = [
        TokenAttributions(
            token=token, attributions={shap_values.output_names[label_idx]: score}
        )
        for token, score in zip(shap_values.data[i], [row[label_idx] for row in scaled])
    ]

    # Build Rubrix record
    record = TextClassificationRecord(
        inputs=sst["sentence"][i],
        prediction=[(pred["label"], pred["score"]) for pred in predictions[i]],
        explanation={"text": token_attributions},
    )

    # Log record
    rb.log(record, name="rubrix_shap_example")

```

4.16 Weak supervision

These tutorials get you started with weak supervision using Rubrix.

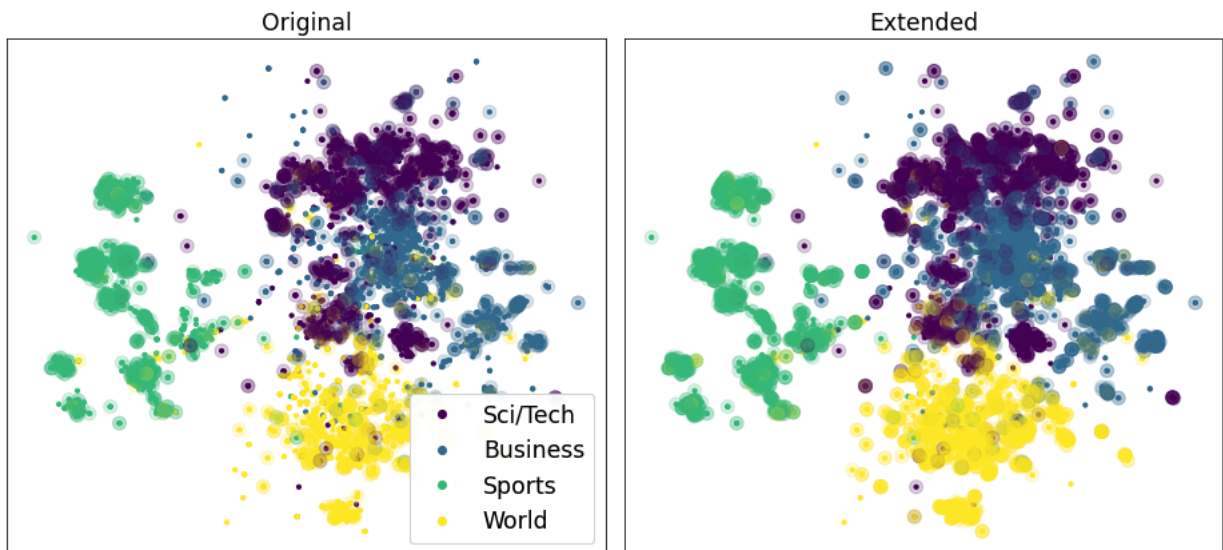
Building a news classifier with weak supervision Build a news classifier using rules and weak supervision, and how Rubrix makes this process very interactive.

The screenshot shows the Rubrix web interface for the 'go_emotions' dataset. At the top, there's a search bar with the query 'text:(thanks AND good)'. Below it, a list of emotion tags is displayed: admiration, annoyance, approval, curiosity, gratitude, and optimism. A 'Save rule' button is visible. To the right, a 'Rule Metrics' panel shows the following data:

Rule Metrics	
Coverage	0.784%
Total:	20.556%
Annotations	1.075%
Total:	20.161%
Precision	100.00%
Avg:	98.81%
Correct/incorrect	8/0
Total:	83/1

Below the metrics, there's a 'Manage rules' button. At the bottom, a text snippet is shown: 'I think you're right. I've been proactive before and it helped. Thanks and hope this year is good to you as well!'.

Weak supervision in multi-label text classification tasks Learn how to tackle multi-label classification tasks with weak supervision with the example of two challenging datasets.



Extending weak supervision workflows with sentence embeddings Extend your weak supervision workflows in Rubrix with sentence embeddings to increase the coverage and obtain more training data.

The screenshot displays the Rubrix web interface for the 'conll2003_dev' dataset. The top header shows the Rubrix logo and the dataset name. The sidebar on the left contains filters for 'LOC' (1), 'MISC' (2), 'ORG' (3), and 'PER' (4), along with a 'Mode' dropdown. The main table lists records with text snippets and their corresponding NER labels and counts. For instance, the first record is 'Wigan 4 Chester 2', where 'Wigan' is labeled 'LOC' with a count of 4 and 'Chester' is labeled 'PER' with a count of 2. Other records include 'Port Vale 4 1 2 1 4 4 5', 'West Bromwich 3 0 2 1 2 3 2', 'Livingston 3 3 0 0 6 2 9', and 'Hajduk 4 2 0 2 5 3 6'. The bottom of the interface shows pagination controls, including 'Records per page' (set to 5) and a page indicator '1-5 of 160'.

Weakly supervised NER with skweak Use Rubrix to improve weak supervision and data programming workflows for Named Entity Recognition tasks with the skweak library.

4.16.1 Weak supervision in multi-label text classification tasks

In this tutorial we use Rubrix and weak supervision to tackle two multi-label classification datasets:

- The first dataset is a curated version of [GoEmotions](#), a dataset intended for **multi-label emotion classification**.
- We inspect the dataset in Rubrix, come up with good heuristics, and combine them with a label model to train a **weakly supervised Hugging Face transformer**.
- In the second dataset, we [categorize research papers](#) by topic based on their titles, which is a **multi-label topic classification** problem.
- We repeat the process of finding good heuristics, combine them with a label model and train a **lightweight downstream model using sklearn** in the end.

Note

If you are new to weak supervision, check out our [weak supervision guide](#) and our first [weak supervision tutorial](#).

Note

The Snorkel and FlyingSquid label models are not suited for multi-label classification tasks and do not support them.

Setup

Rubrix, is a free and open-source tool to explore, annotate, and monitor data for NLP projects.

If you are new to Rubrix, check out the [Github repository](#) .

If you have not installed and launched Rubrix yet, check the [Setup and Installation guide](#).

For this tutorial we also need some third party libraries that can be installed via pip:

```
[ ]: %pip install datasets "transformers[torch]" scikit-multilearn ipywidgets -qqq
```

GoEmotions

The original [GoEmotions](#) is a challenging dataset intended for multi-label emotion classification. For this tutorial, we simplify it a bit by selecting only 6 out of the 28 emotions: *admiration*, *annoyance*, *approval*, *curiosity*, *gratitude*, *optimism*. We also try to accentuate the multi-label part of the dataset by down-sampling the examples that are classified with only one label. See Appendix A for all the details of this preprocessing step.

Define rules

Let us start by downloading our curated version of the dataset from the Hugging Face Hub, and log it to Rubrix:

```
[ ]: import rubrix as rb
from datasets import load_dataset

# Download preprocessed dataset
ds_rb = rb.read_dataset(
    load_dataset("rubrix/go_emotions_multi-label", split="train"),
    task="TextClassification"
)
```

```
[ ]: # Log dataset to Rubrix to find good heuristics
rb.log(ds_rb, name="go_emotions")
```

After uploading the dataset, we can explore and inspect it to find good heuristic rules. For this we highly recommend the dedicated *Define rules mode* of the Rubrix web app, that allows you to quickly iterate over heuristic rules, compute their metrics and save them.

Here we copy our rules found via the web app to the notebook for you to easily follow along the tutorial.

```
[ ]: from rubrix.labeling.text_classification import Rule

# Define our heuristic rules, they can surely be improved
rules = [
    Rule("thank*", "gratitude"),
    Rule("appreciate", "gratitude"),
    Rule("text:(thanks AND good)", ["admiration", "gratitude"]),
    Rule("advice", "admiration"),
    Rule("amazing", "admiration"),
    Rule("awesome", "admiration"),
    Rule("impressed", "admiration"),
    Rule("text:(good AND (point OR call OR idea OR job))", "admiration"),
```

(continues on next page)

(continued from previous page)

```

Rule("legend", "admiration"),
Rule("exactly", "approval"),
Rule("agree", "approval"),
Rule("yeah", "approval"),
Rule("suck", "annoyance"),
Rule("pissed", "annoyance"),
Rule("annoying", "annoyance"),
Rule("ruined", "annoyance"),
Rule("hoping", "optimism"),
Rule('text:("good luck")', "optimism"),
Rule("nice day", "optimism"),
Rule("what is", "curiosity"),
Rule("can you", "curiosity"),
Rule("would you", "curiosity"),
]

```

We go on and apply these heuristic rules to our dataset creating our weak label matrix. Since we are dealing with a multi-label classification task, the weak label matrix will have 3 dimensions.

Dimensions of the weak multi label matrix: *number of records x number of rules x number of labels*

It will be filled with 0 and 1, depending on if the rule voted for the respective label or not. If the rule abstained for a given record, the matrix will be filled with -1.

```

[ ]: from rubrix.labeling.text_classification import WeakMultiLabels

# Compute the weak labels for our dataset given the rules.
# If your dataset already contains rules you can omit the rules argument.
weak_labels = WeakMultiLabels("go_emotions", rules=rules)

```

We can call the `weak_labels.summary()` method to check the precision of each rule as well as our total coverage of the dataset.

```

[15]: # Check coverage/precision of our rules
weak_labels.summary()

[15]:
↪ label \
thank*
↪ {gratitude}
appreciate
↪ {gratitude}
text:(thanks AND good) {admiration,
↪ gratitude}
advice
↪ {admiration}
amazing
↪ {admiration}
awesome
↪ {admiration}
impressed
↪ {admiration}
text:(good AND (point OR call OR idea OR job))
↪ {admiration}

```

(continues on next page)

(continued from previous page)

```

legend
  ↳{admiration}
exactly
  ↳{approval}
agree
  ↳{approval}
yeah
  ↳{approval}
suck
  ↳{annoyance}
pissed
  ↳{annoyance}
annoying
  ↳{annoyance}
ruined
  ↳{annoyance}
hoping
  ↳{optimism}
text:("good luck")
  ↳{optimism}
"nice day"
  ↳{optimism}
"what is"
  ↳{curiosity}
"can you"
  ↳{curiosity}
"would you"
  ↳{curiosity}
total                                {curiosity, annoyance, admiration,
  ↳approval, o...

```

	coverage	annotated_coverage \
thank*	0.196768	0.196237
appreciate	0.016160	0.021505
text:(thanks AND good)	0.007842	0.010753
advice	0.008317	0.008065
amazing	0.025428	0.021505
awesome	0.025190	0.034946
impressed	0.002139	0.005376
text:(good AND (point OR call OR idea OR job))	0.008555	0.018817
legend	0.001901	0.002688
exactly	0.004278	0.002688
agree	0.016873	0.021505
yeah	0.024952	0.021505
suck	0.002139	0.008065
pissed	0.002139	0.008065
annoying	0.003327	0.018817
ruined	0.000713	0.002688
hoping	0.003565	0.005376
text:("good luck")	0.015209	0.018817
"nice day"	0.000713	0.005376
"what is"	0.004040	0.005376

(continues on next page)

(continued from previous page)

"can you"	0.004278	0.008065
"would you"	0.000951	0.005376
total	0.327234	0.384409

	overlaps	correct	incorrect	\
thank*	0.037785	73	0	
appreciate	0.009506	7	1	
text:(thanks AND good)	0.007605	8	0	
advice	0.006654	3	0	
amazing	0.003565	8	0	
awesome	0.006179	12	1	
impressed	0.000000	2	0	
text:(good AND (point OR call OR idea OR job))	0.002376	7	0	
legend	0.000475	1	0	
exactly	0.001188	1	0	
agree	0.003089	6	2	
yeah	0.004990	5	3	
suck	0.000475	3	0	
pissed	0.000475	2	1	
annoying	0.000951	7	0	
ruined	0.000238	1	0	
hoping	0.000713	2	0	
text:("good luck")	0.002139	4	3	
"nice day"	0.000000	2	0	
"what is"	0.000951	2	0	
"can you"	0.000713	3	0	
"would you"	0.000000	2	0	
total	0.041825	161	11	

	precision
thank*	1.000000
appreciate	0.875000
text:(thanks AND good)	1.000000
advice	1.000000
amazing	1.000000
awesome	0.923077
impressed	1.000000
text:(good AND (point OR call OR idea OR job))	1.000000
legend	1.000000
exactly	1.000000
agree	0.750000
yeah	0.625000
suck	1.000000
pissed	0.666667
annoying	1.000000
ruined	1.000000
hoping	1.000000
text:("good luck")	0.571429
"nice day"	1.000000
"what is"	1.000000
"can you"	1.000000
"would you"	1.000000

(continues on next page)

(continued from previous page)

total	0.936047
-------	----------

Create training set

When we are happy with our heuristics, it is time to combine them and compute weak labels for the training of our downstream model. For this we will use the `MajorityVoter`. In the multi-label case, it sets the probability of a label to 0 or 1 depending on whether at least one non-abstaining rule voted for the respective label or not.

```
[ ]: from rubrix.labeling.text_classification import MajorityVoter

# Use the majority voter as the label model
label_model = MajorityVoter(weak_labels)
```

From our label model we get the training records together with its weak labels and probabilities. We will use the weak labels with a probability greater than 0.5 as labels for our training, and hence copy them to the annotation property of our records.

```
[ ]: # Get records with the predictions from the label model to train a down-stream model
train_rb = label_model.predict()

# Copy label model predictions to annotation with a threshold of 0.5
for rec in train_rb:
    rec.annotation = [pred[0] for pred in rec.prediction if pred[1] > 0.5]
```

We extract the test set with manual annotations from our `WeakMultiLabels` object:

```
[ ]: # Get records with manual annotations to use as test set for the down-stream model
test_rb = rb.DatasetForTextClassification(weak_labels.records(max_annotation=True))
```

We will use the convenient `DatasetForTextClassification.prepare_for_training()` method to create datasets optimized for training with the Hugging Face transformers library:

```
[ ]: from datasets import DatasetDict

# Create dataset dictionary and shuffle training set
ds = DatasetDict(
    train=train_rb.prepare_for_training().shuffle(seed=42),
    test=test_rb.prepare_for_training(),
)
```

Let us push the dataset to the Hub to share it with our colleagues. It is also an easy way to outsource the training of the model to an environment with an accelerator, like Google Colab for example.

```
[ ]: # Push dataset for training our down-stream model to the HF hub
ds.push_to_hub("rubrix/go_emotions_training")
```

Train a transformer downstream model

The following steps are basically a copy&paste from the amazing documentation of the [Hugging Face transformers](#) library.

First, we will load the tokenizer corresponding to our model, which we choose to be the [distilled version](#) of the infamous BERT.

Note

Since we will use a full-blown transformer as a downstream model (albeit a distilled one), we recommend executing the following code on a machine with a GPU, or in a Google Colab with a GPU backend enabled.

```
[ ]: from transformers import AutoTokenizer

# Initialize tokenizer
tokenizer = AutoTokenizer.from_pretrained("distilbert-base-uncased")
```

Afterward, we tokenize our data:

```
[ ]: def tokenize_func(example):
    return tokenizer(example["text"], padding="max_length", truncation=True)

# Tokenize the data
tokenized_ds = ds.map(tokenize_func, batched=True)
```

The transformer model expects our labels to follow a common multi-label format of binaries, so let us use [sklearn](#) for this transformation.

```
[ ]: from sklearn.preprocessing import MultiLabelBinarizer

# Turn labels into multi-label format
mb = MultiLabelBinarizer()
mb.fit(ds["test"]["label"])

def binarize_labels(example):
    return {"label": mb.transform(example["label"])}

binarized_tokenized_ds = tokenized_ds.map(binarize_labels, batched=True)
```

Before we start the training, it is important to define our metric for the evaluation. Here we settle on the commonly used micro averaged *F1* metric, but we will also keep track of the *F1 per label*, for a more in-depth error analysis afterward.

```
[ ]: from datasets import load_metric
import numpy as np

# Define our metrics
metric = load_metric("f1", config_name="multilabel")
def compute_metrics(eval_pred):
    logits, labels = eval_pred
    # apply sigmoid
    predictions = ( 1. / (1 + np.exp(-logits)) ) > 0.5
```

(continues on next page)

(continued from previous page)

```

# f1 micro averaged
metrics = metric.compute(prediction=predictions, reference=labels, average="micro")
# f1 per label
per_label_metric = metric.compute(prediction=predictions, reference=labels,
↪ average=None)
for label, f1 in zip(0, ["train"].features["label"][0].names, per_label_metric["f1"]):
    metrics[f"f1_{label}"] = f1

return metrics

```

Now we are ready to load our pretrained transformer model and prepare it for our task: multi-label text classification with 6 labels.

```

[ ]: from transformers import AutoModelForSequenceClassification

# Init our down-stream model
model = AutoModelForSequenceClassification.from_pretrained(
    "distilbert-base-uncased",
    problem_type="multi_label_classification",
    num_labels=6
)

```

The only thing missing for the training is the `Trainer` and its `TrainingArguments`. To keep it simple, we mostly rely on the default arguments, that often work out of the box, but tweak a bit the batch size to train faster. We also checked that 2 epochs are enough for our rather small dataset.

```

[ ]: from transformers import TrainingArguments

# Set our training arguments
training_args = TrainingArgument(
    output_dir="test_trainer",
    evaluation_strategy="epoch",
    num_train_epochs=2,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
)

```

```

[ ]: from transformers import Trainer

# Init the trainer
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=binarized_tokenized_ds["train"],
    eval_dataset=binarized_tokenized_ds["test"],
    compute_metric=compute_metrics,
)

```

```

[ ]: # Train the down-stream model
trainer.train()

```

We achieved an micro averaged *F1* of about 0.54, which is not perfect, but a good baseline for this challenging dataset. When inspecting the *F1s per label*, we clearly see that the worst performing labels are the ones with the poorest

heuristics in terms of accuracy and coverage, which comes to no surprise.

Research topic dataset

After covering a multi-label emotion classification task, we will try to do the same for a multi-label classification task related to topic modeling. In this dataset, research papers were classified with 6 non-exclusive labels based on their title and abstract.

We will try to classify the papers only based on the title, which is considerably harder, but allows us to quickly scan through the data and come up with heuristics. See Appendix B for all the details of the minimal data preprocessing.

Define rules

Let us start by downloading our preprocessed dataset from the Hugging Face Hub, and log it to Rubrix:

```
[ ]: import rubrix as rb
      from datasets import load_dataset

      # Download preprocessed dataset
      ds_rb = rb.read_dataset(
          load_dataset("rubrix/research_titles_multi-label", split="train"),
          task="TextClassification"
      )
```

```
[ ]: # Log dataset to Rubrix to find good heuristics
      rb.log(ds_rb, "research_titles")
```

After uploading the dataset, we can explore and inspect it to find good heuristic rules. For this we highly recommend the dedicated *Define rules mode* of the Rubrix web app, that allows you to quickly iterate over heuristic rules, compute their metrics and save them.

Here we copy our rules found via the web app to the notebook for you to easily follow along the tutorial.

```
[ ]: from rubrix.labeling.text_classification import Rule

      # Define our heuristic rules (can probably be improved)
      rules = [
          Rule("stock*", "Quantitative Finance"),
          Rule("*asset*", "Quantitative Finance"),
          Rule("trading", "Quantitative Finance"),
          Rule("finance", "Quantitative Finance"),
          Rule("pric*", "Quantitative Finance"),
          Rule("economy", "Quantitative Finance"),
          Rule("deep AND neural AND network*", "Computer Science"),
          Rule("convolutional", "Computer Science"),
          Rule("memor* AND (design* OR network*)", "Computer Science"),
          Rule("system* AND design*", "Computer Science"),
          Rule("allocat* AND *net*", "Computer Science"),
          Rule("program", "Computer Science"),
          Rule("classification* AND (label* OR deep)", "Computer Science"),
          Rule("scattering", "Physics"),
          Rule("astro*", "Physics"),
          Rule("material*", "Physics"),
```

(continues on next page)

(continued from previous page)

```
Rule("spin", "Physics"),
Rule("magnetic", "Physics"),
Rule("optical", "Physics"),
Rule("ray", "Physics"),
Rule("entangle*", "Physics"),
Rule("*algebra*", "Mathematics"),
Rule("manifold* AND (NOT learn*)", "Mathematics"),
Rule("equation", "Mathematics"),
Rule("spaces", "Mathematics"),
Rule("operators", "Mathematics"),
Rule("regression", "Statistics"),
Rule("bayes*", "Statistics"),
Rule("estimation", "Statistics"),
Rule("mixture", "Statistics"),
Rule("gaussian", "Statistics"),
Rule("gene", "Quantitative Biology"),
]
```

We go on and apply these heuristic rules to our dataset creating our weak label matrix. As mentioned in the *GoEmotions* section, the weak label matrix will have 3 dimensions and values of -1, 0 and 1.

```
[ ]: from rubrix.labeling.text_classification import WeakMultilabels

# Compute the weak labels for our dataset given the rules
# If your dataset already contains rules you can omit the rules argument.
weak_labels = WeakMultilabels("research_titles", rules=rules)
```

Let us get an overview of the our heuristics and how they perform:

```
[5]: # Check coverage/precision of our rules
weak_labels.summary()
```

	label
↪ \	
stock*	{Quantitative Finance}
asset	{Quantitative Finance}
trading	{Quantitative Finance}
finance	{Quantitative Finance}
pric*	{Quantitative Finance}
economy	{Quantitative Finance}
deep AND neural AND network*	{Computer Science}
convolutional	{Computer Science}
memor* AND (design* OR network*)	{Computer Science}
system* AND design*	{Computer Science}
allocat* AND *net*	{Computer Science}
program	{Computer Science}
classification* AND (label* OR deep)	{Computer Science}
scattering	{Physics}
astro*	{Physics}
material*	{Physics}
spin	{Physics}
magnetic	{Physics}
optical	{Physics}

(continues on next page)

(continued from previous page)

ray	{Physics}
entangle*	{Physics}
algebra	{Mathematics}
manifold* AND (NOT learn*)	{Mathematics}
equation	{Mathematics}
spaces	{Mathematics}
operators	{Mathematics}
regression	{Statistics}
bayes*	{Statistics}
estimation	{Statistics}
mixture	{Statistics}
gaussian	{Statistics}
gene	{Quantitative Biology}
total	{Physics, Quantitative Biology, Mathematics, C...

	coverage	annotated_coverage	overlaps \
stock*	0.000954	0.000715	0.000334
asset	0.000477	0.000715	0.000286
trading	0.000954	0.000238	0.000191
finance	0.000048	0.000238	0.000000
pric*	0.003433	0.003337	0.000715
economy	0.000238	0.000238	0.000000
deep AND neural AND network*	0.009155	0.010250	0.002909
convolutional	0.010109	0.009297	0.002241
memor* AND (design* OR network*)	0.001383	0.002145	0.000286
system* AND design*	0.001144	0.002384	0.000238
allocat* AND *net*	0.000763	0.000715	0.000000
program	0.002623	0.003099	0.000143
classification* AND (label* OR deep)	0.003338	0.004052	0.001335
scattering	0.004053	0.002861	0.001001
astro*	0.003099	0.004052	0.000620
material*	0.004148	0.003099	0.000238
spin	0.013542	0.015018	0.002146
magnetic	0.011301	0.012872	0.002432
optical	0.007105	0.006913	0.001097
ray	0.005865	0.007390	0.001192
entangle*	0.002623	0.002861	0.000095
algebra	0.014829	0.018355	0.000620
manifold* AND (NOT learn*)	0.007057	0.008343	0.000858
equation	0.010681	0.007867	0.000954
spaces	0.010586	0.009774	0.001860
operators	0.006151	0.005959	0.001526
regression	0.009393	0.009058	0.002575
bayes*	0.015306	0.014779	0.003147
estimation	0.021266	0.021216	0.003385
mixture	0.003290	0.003099	0.001287
gaussian	0.009250	0.011204	0.002766
gene	0.001287	0.001669	0.000191
total	0.176616	0.185936	0.017833

	correct	incorrect	precision
stock*	3	0	1.000000

(continues on next page)

(continued from previous page)

asset	3	0	1.000000
trading	1	0	1.000000
finance	1	0	1.000000
pric*	9	5	0.642857
economy	1	0	1.000000
deep AND neural AND network*	32	11	0.744186
convolutional	32	7	0.820513
memor* AND (design* OR network*)	9	0	1.000000
system* AND design*	9	1	0.900000
allocat* AND *net*	3	0	1.000000
program	11	2	0.846154
classification* AND (label* OR deep)	14	3	0.823529
scattering	10	2	0.833333
astro*	17	0	1.000000
material*	10	3	0.769231
spin	60	3	0.952381
magnetic	49	5	0.907407
optical	27	2	0.931034
ray	27	4	0.870968
entangle*	11	1	0.916667
algebra	70	7	0.909091
manifold* AND (NOT learn*)	28	7	0.800000
equation	24	9	0.727273
spaces	38	3	0.926829
operators	22	3	0.880000
regression	33	5	0.868421
bayes*	49	13	0.790323
estimation	65	24	0.730337
mixture	10	3	0.769231
gaussian	36	11	0.765957
gene	6	1	0.857143
total	720	135	0.842105

Create training set

When we are happy with our heuristics, it is time to combine them and compute weak labels for the training of our downstream model. As for the “GoEmotions” dataset, we will use the simple MajorityVoter.

```
[ ]: from rubrix.labeling.text_classification import MajorityVoter

# Use the majority voter as the label model
label_model = MajorityVoter(weak_labels)
```

From our label model we get the training records together with its weak labels and probabilities. Since we are going to train an sklearn model, we will put the records in a pandas DataFrame that generally has a good integration with the sklearn ecosystem.

```
[ ]: train_df = label_model.predict().to_pandas()
```

Before training our model, we need to extract the training labels from the label model predictions and transform them into a multi-label compatible format.

```
[ ]: # Create labels in multi-label format, we will use a threshold of 0.5 for the probability
def multi_label_binarizer(predictions, threshold=0.5):
    predicted_labels = [label for label, prob in predictions if prob > threshold]
    binary_labels = [1 if label in predicted_labels else 0 for label in weak_labels.
    ↪ labels]
    return binary_labels

train_df["label"] = train_df.prediction.map(multi_label_binarizer)
```

Now, let us define our downstream model and train it.

We will use the `scikit-multilearn` library to wrap a multinomial **Naive Bayes classifier** that is suitable for classification with discrete features (e.g., word counts for text classification). The `BinaryRelevance` class transforms the multi-label problem with L labels into L single-label binary classification problems, so in the end we will automatically fit L naive bayes classifiers to our data.

The features for our classifier will be the counts of different word **n-grams**: that is, for each example we count the number of contiguous sequences of n words, where n goes from 1 to 5. We extract these features with the `CountVectorizer`.

Finally, we will put our feature extractor and multi-label classifier in a sklearn pipeline that makes fitting and scoring the model a breeze.

```
[ ]: from skmultilearn.problem_transform import BinaryRelevance
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.pipeline import Pipeline

# Define our down-stream model
classifier = Pipeline([
    ('vect', CountVectorizer()),
    ('clf', BinaryRelevance(MultinomialNB()))
])
```

Training the model is as easy as calling the `fit` method on the our pipeline, and provide our training text and training labels.

```
[ ]: import numpy as np

# Fit the down-stream classifier
classifier.fit(
    X=train_df.text,
    y=np.array(train_df.label.tolist()),
)
```

To score our trained model, we retrieve its predictions of the test set and use sklearn's `classification_report` to get all important classification metrics in a nicely formatted string.

```
[ ]: # Get predictions for test set
predictions = classifier.predict(
    X=[rec.text for rec in weak_labels.records(has_annotation=True)]
)
```

```
[77]: from sklearn.metrics import classification_report
```

(continues on next page)

(continued from previous page)

```
# Compute metrics
print(classification_report(
    weak_labels.annotation(),
    predictions,
    target_names=weak_labels.labels
))
```

	precision	recall	f1-score	support
Computer Science	0.82	0.26	0.40	1740
Mathematics	0.71	0.64	0.67	1141
Physics	0.81	0.70	0.75	1186
Quantitative Biology	1.00	0.01	0.02	109
Quantitative Finance	0.44	0.09	0.15	45
Statistics	0.48	0.71	0.57	1069
micro avg	0.66	0.53	0.59	5290
macro avg	0.71	0.40	0.43	5290
weighted avg	0.73	0.53	0.56	5290
samples avg	0.66	0.56	0.59	5290

We obtain a micro averaged F1 score of around 0.59, which again is not perfect but can serve as a decent baseline for future improvements. Looking at the F1 per label, we see that the main problem is the recall of our heuristics and we should either define more of them, or try to find more general ones.

Summary

In this tutorial we saw how you can use *Rubrix* to tackle multi-label text classification problems with weak supervision. We showed you how to train two downstream models on two different multi-label datasets using the discovered heuristics.

For the emotion classification task, we trained a full-blown transformer model with Hugging Face, while for the topic classification task, we relied on a more lightweight Bayes classifier from sklearn. Although the results are not perfect, they can serve as a good baseline for future improvements.

So the next time you encounter a multi-label classification problem, maybe try out weak supervision with *Rubrix* and save some time for your annotation team .

Next steps

Star Rubrix [Github repo](#) to stay updated.

[Rubrix documentation](#) for more guides and tutorials.

Join the Rubrix community! A good place to start is the [discussion forum](#).

Appendix A

This appendix summarizes the preprocessing steps for our curated *GoEmotions* dataset. The goal was to limit the labels, and down-sample single-label annotations to move the focus to multi-label outputs.

```
[ ]: # load original dataset and check label frequencies

import pandas as pd
import datasets

go_emotions = datasets.load_dataset("go_emotions")
df = go_emotions["test"].to_pandas()

def int2str():
    #return int(i)
    return go_emotions["train"].features["labels"].feature.int2str(int())

label_freq = []
idx_multi = df.labels.map(lambda x: len(x) > 1)
df["is_single"] = df.labels.map(lambda x: 0 if len(x) > 1 else 1)
df[idx_multi].labels.map(lambda x: [label_freq.append(int(x)) for i in x])
pd.Series(label_freq).value_counts()

[ ]: # limit labels, down-sample single-label annotations and create Rubrix records

import rubrix as rb

def create(split: str) -> pd.DataFrame:
    df = go_emotions[split].to_pandas()
    df["is_single"] = df.labels.map(lambda x: 0 if len(x) > 1 else 1)

    #['admiration', 'approval', 'annoyance', 'gratitude', 'curiosity', 'optimism', 'amusement']
    idx_most_common = df.labels.map(lambda x: all([int(label) in [0, 4, 3, 15, 7, 15, 15, 20] for label in x]))
    df_multi = df[(df.is_single == 0) & idx_most_common]
    df_single = df[idx_most_common].sample(3*len(df_multi), weights="is_single", axis=0, random_state=42)
    return pd.concat([df_multi, df_single]).sample(frac=1, random_state=42)

def make_records(row, is_train: bool) -> rb.TextClassificationRecord:
    annotation = [int2str(x) for i in row.labels] if not is_train else None
    return rb.TextClassificationRecord(
        inputs=row.text,
        annotation=annotation,
        multi_label=True,
        id=row.id,
    )

train_recs = create("train").apply(make_records, axis=1, is_train=True)
test_recs = create("test").apply(make_records, axis=1, is_train=False)

records = train_recs.tolist() + test_recs.tolist()
```

```
[ ]: # publish dataset in the Hub

ds_rb = rb.DatasetForTextClassification(records).to_datasets()

ds_rb.push_to_hub("rubrix/go_emotions_multi-label", private=True)
```

Appendix B

This appendix summarizes the minimal preprocessing done to [this multi-label classification dataset](#) from Kaggle. You can download the original data (`train.csv`) following the Kaggle link.

The preprocessing consists of extracting only the title from the research paper, and split the data into a train and validation set.

```
[ ]: # Extract the title and split the data

import pandas as pd
import rubrix as rb
from sklearn.model_selection import train_test_split

df = pd.read_csv("train.csv")

_, test_id = train_test_split(df.ID, test_size=0.2, random_state=42)

labels = ["Computer Science", "Physics", "Mathematics", "Statistics", "Quantitative_
↪Biology", "Quantitative Finance"]
def make_record(row):
    annotation = [label for label in labels if row[label] == 1]
    return rb.TextClassificationRecord(
        inputs=row.TITLE,
        # inputs={"title": row.TITLE, "abstract": row.ABSTRACT},
        annotation=annotation if row.ID in test_id else None,
        multi_label=True,
        id=row.ID,
    )

records = df.apply(make_record, axis=1)
```

```
[ ]: # publish the dataset in the Hub

dataset_rb = rb.DatasetForTextClassification(records.tolist())

dataset_rb.to_datasets().push_to_hub("rubrix/research_titles_multi-label")
```


4.16.2 Weakly supervised NER with skweak

This tutorial will walk you through the process of using Rubrix to improve weak supervision and data programming workflows with the [skweak library](#).

- Using Rubrix, *skweak* and *spaCy*, we define heuristic rules for the [CoNLL 2003](#) dataset.
- We then log the labelled documents to Rubrix and visualize the results via its web app.
- After aggregating the noisy labels, we fine-tune and evaluate a *spaCy* NER model.

Introduction

Our goal is to show you how you can incorporate Rubrix into data programming workflows to programmatically build training data with a human-in-the-loop approach. We will use the [skweak](#) library.

What is weak supervision? and skweak?

Weak supervision is a branch of machine learning based on getting lower quality labels more efficiently. We can achieve this by using [skweak](#), a library for programmatically building and managing training datasets without manual labeling.

This tutorial

In this tutorial, we will show you how to extend weak supervision workflows in *skweak* with Rubrix.

We will take records from the CoNLL 2003 dataset and build our own annotations with *skweak*. Then we are going to evaluate NER models trained on our annotations on the development set of CoNLL 2003.

Setup

Rubrix, is a free and open-source tool to explore, annotate, and monitor data for NLP projects.

If you are new to Rubrix, check out the [Github repository](#).

If you have not installed and launched Rubrix yet, check the [Setup and Installation guide](#).

For this tutorial we also need some third party libraries that can be installed via pip:

```
[ ]: %pip install -U spacy -qqq
      %pip install --user git+https://github.com/NorskRegnesentral/skweak -qqq
      !python -m spacy download en_core_web_lg
```

1. Log the dataset into Rubrix

Rubrix allows you to log and track data for different NLP tasks (such as Token Classification or Text Classification).

In this tutorial, we will use the English portion of the [CoNLL 2003](#) dataset, a standard Named Entity Recognition benchmark.

The dataset

We will use `skweak`'s data programming methods to annotate our training set, with the help of Rubrix for analyzing and reviewing the data. We will then train a model on this training set.

Although the gold labels for the training set of CoNLL 2003 are already known, we will purposefully ignore them, as our goal in this tutorial is to build our own annotations and see how well they perform on the development set.

And to simplify our tutorial, only the `ORG` label will be taken into account, both in training and evaluation. Other labels present on the dataset will be ignored (`LOC`, `PER` and `MISC`).

We will load the CoNLL 2003 dataset with the help of the `datasets` library.

```
[ ]: from datasets import load_dataset

conll2003 = load_dataset("conll2003")
```

Logging

Before we log the development data, we define a utility function that will convert our NER tags from the `datasets` format to Rubrix annotations.

```
[ ]: from spacy.tokens import Doc
from spacy.vocab import Vocab
from spacy.training.iob_utils import iob_to_biluo, biluo_tags_to_offsets

def tags_to_entities(row):
    doc = Doc(Vocab(), words=row["tokens"])
    ner_tags = conll2003["train"].features["ner_tags"].feature.int1str(row["ner_tags"])
    offsets = biluo_tags_to_offsets(doc, iob_to_biluo(ner_tags))

    return [(entity, start, stop) for start, stop, entity in offsets]
```

We define a generator that will yield each row of our dataset as a `TokenClassificationRecord` object.

```
[ ]: import rubrix as rb
from tqdm.auto import tqdm

def dataset_to_records(dataset):
    for row in tqdm(dataset):

        text = " ".join(row["tokens"])

        # seems like we have "empty" rows
        if not text.strip():
            continue

        yield rb.TokenClassificationRecord(
            text=text,
            tokens=row["tokens"],
            annotation=tags_to_entities(row)
        )
```

Now we upload our records through the Rubrix API for a first inspection. Although we are uploading all annotations, we can filter for ORG entities on the web app.

```
[ ]: rb.log(dataset_by_record (conll2003 ["validation"]), "conll2003_dev")
```

2. Use Rubrix to write skweak heuristic rules

Heuristic rules in `skweak` are applied through labelling functions. Each of these functions must yield the start and end index of the annotated `span` followed by its assigned label.

Annotating a specific case: sports teams

We define our first heuristic rules to match records related to sports teams.

After inspecting the dataset on Rubrix, we are able to notice that several records start with the name of a sports team followed by its game scores.

We also notice that other group of records feature the names of two sports teams and their scores after a match against each other.

We write two rules to capture these sports team names as ORG entities.

The screenshot shows the Rubrix web application interface. At the top, there is a blue navigation bar with the Rubrix logo and the text 'rubrix / conll2003_dev'. Below this is a filter bar with tabs for 'LOC [1]', 'MISC [2]', 'ORG [3]', and 'PER [4]'. The main content area displays a list of records. The first record is 'Wigan 4 Chester 2' with an 'annot. ORG' label. The second record is 'Port Vale 4 1 2 1 4 4 5' with an 'annot. ORG' label. The third record is 'West Bromwich 3 0 2 1 2 3 2'. The fourth record is 'Livingston 3 3 0 0 6 2 9'. The fifth record is 'Hajduk 4 2 0 2 5 3 6'. On the right side, there is a sidebar with icons for 'Mode', 'Metrics', and 'Refresh'. At the bottom, there is a pagination bar showing 'Records per page: 5' and '1 2 3 ... 32 Next >'.

```
[ ]: def sports_results_detector(loc):
    """
    Captures a sports team name followed by its game scores.
    Labels the sports team as an ORG.
    Examples:
        Loznica 4 2 0 2 7 4 6
```

(continues on next page)

(continued from previous page)

```

    Berwick 3 0 0 3 1 14 0
    """
    # Label first word as ORG if it is followed only by numbers and punctuation.
    if len(doc) < 2:
        return
    has_digits = False
    for idx, token in enumerate(doc):
        if not idx and token.text.isalpha() and token.text.istitle():
            continue
        elif idx and token.text.isdigit():
            continue
        else:
            break
    else:
        yield 0, 1, "ORG"

def sports_match_detector(doc):
    """
    Captures a sports match.
    Labels both sports teams as ORG.
    Examples:
        Bournemouth 1 Peterborough 2
        Dumbarton 1 Brechin 1
    """
    if len(doc) != 4:
        return

    if (
        doc[0].text.istitle()
        and doc[1].text.isdigit()
        and doc[2].text.istitle()
        and doc[3].text.isdigit()
    ):
        yield 0, 1, "ORG"
        yield 2, 3, "ORG"

```

Let's encapsulate our heuristic rules as labelling functions.

Labelling functions are defined as `FunctionAnnotator` objects, and multiple functions can be grouped inside a single `CombinedAnnotator`.

```

[ ]: from skweak.heuristics import FunctionAnnotator

sports_results_annotator = FunctionAnnotator("sports_results", sports_results_detector)
sports_match_annotator = FunctionAnnotator("sports_match", sports_match_detector)

```

Although it is possible to call each one of these annotators independently, if we are going to call several annotators at the same time, it is more convenient to group them under a single combined annotator.

We add each one of them to our combined annotator through a `add_annotator` method.

```

[ ]: from skweak.base import CombinedAnnotator

```

(continues on next page)

(continued from previous page)

```
rule_based_annotator = CombinedAnnotator()

for annotator in [sports_results_annotator, sports_match_annotator]:
    rule_based_annotator.add_annotator(annotator)
```

Annotating with generic rules

We can also write rules that are a little bit more generic.

For instance, organizations often are presented as a series of capitalized words that either start or end with a certain keyword. We write a generator called `title_detector` to capture them.

```
[ ]: def title_detector(loc, keyword=None, label="ORG", reverse=False):
    """
    Captures a sequence of capitalized words that either start or end with a certain
    keyword.
    Labels the sequence, including the keyword, with the ORG label.
    Examples:

        The following examples start with the keyword "U.S.":
        - U.S. Treasury Department
        - U.S. Treasuries
        - U.S. Agriculture Department

        The following examples end with the keyword "Corp":
        - First of Michigan Corp
```

(continues on next page)

(continued from previous page)

```

        - Caltex Petroleum Corp
        - Kia Motor Corp
    """
    start = None
    end = None

    if reverse:
        len_doc = len(doc)
        doc = reversed(doc)

    for idx, token in enumerate(doc):
        if token.text == keyword:
            start = idx
        elif start:
            if token.text.istitle():
                continue
            else:
                if start + 2 != idx:
                    end = idx

                if reverse:
                    start, end = len_doc - end, len_doc - start

                yield start, end, label

    start = None
    end = None

```

We take a small list of keywords that appear at the start of capitalized ORG entities, and initialize an annotator for each one of these keywords. All annotators are added to our combined annotator, `rule_based_annotator`.

```

[ ]: from functools import partial

title_start = [ "Federal", "National", "New", "United", "First", "U.N." ]

for keyword in title_start:
    func = partial(title_detector, keyword=keyword, reverse=False)
    annotator = FunctionAnnotator(keyword + " (start)", func)
    rule_based_annotator.add_annotator(annotator)

```

We repeat the same process, but this time for keywords that appear at the end of capitalized ORG entities.

```

[ ]: title_ending = [
    "Office", "Department", "Association",
    "Corporation", "Army", "Party",
    "Exchange", "Council", "University",
    "Newsroom", "Bureau", "Organisation",
    "Council", "Group", "Inc",
    "Corp", "Ltd"
]

for keyword in title_ending:

```

(continues on next page)

(continued from previous page)

```
func = partial(title_detector, keyword=keyword, reverse=True)
annotator = FunctionAnnotator(keyword + " (end)", func)
rule_based_annotator.add_annotator(annotator)
```

If you have large lists of keywords that must be labelled as entities on every occurrence (e.g. a list of the names of all Fortune 500 companies), you may be interested in utilizing a [GazetteerAnnotator](#). The [Step by step NER tutorial](#) on skweak's documentation shows how you can utilize gazetteers to annotate your data.

Annotating with regex

Until now, all of our rules have manipulated spaCy Doc objects to capture the start and end index of a matching span. However, it is also possible to capture entities by applying regex patterns directly over the text.

Rubrix has some support for regex operators. If we search for `*shire` and filter for records annotated as `ORG`, we will notice that many sports team names end with `-shire`.

The screenshot shows the Rubrix web interface with a search bar containing '*shire'. The results are filtered by the 'ORG' entity type. The interface displays a list of records with highlighted text and entity labels. The records include:

- Middlesex 20 points, Hampshire 5.
- Leicestershire 22 points, Somerset 4.
- CRICKET - LEICESTERSHIRE TAKE OVER AT TOP AFTER INNINGS VICTORY.
- At Hove : Sussex 363 and 144, Lancashire 218 and 53-0.
- Chesterfield : Worcestershire 238 and 133-5, Derbyshire 471 (J. Adams 123, T.O'Gorman 109 not out, K. Barnett 87 ; T. Moody 6-82)

The interface also shows a sidebar with options for Mode, Metrics, and Refresh, and a bottom section for Records per page and pagination.

We can write a rule to capture these entities. This rule can be added to our combined annotator in the same way as all the heuristic rules we have defined so far.

```
[ ]: import re

def shire_detector(doc):
    """
    Captures sports team names ending with -shire.
    Examples:
    - Derbyshire
```

(continues on next page)

(continued from previous page)

```

- Hampshire
- Worcestershire
"""
for match in re.finditer("[A-Z][a-z]*shire", doc.text):
    char_start, char_end = match.span()
    span = doc.char_span(char_start, char_end)
    if span:
        yield span.start, span.end, "ORG"

```

```
[ ]: shire_annotator = FunctionAnnotator("shire_team", shire_detector)
rule_based_annotator.add_annotator(shire_annotator)
```

As long as we return the start, end and label for a span, we are allowed to capture entities in a Doc object in any way we like.

Beyond regex, another way to detect such entities would be to utilize a [Matcher object](#), as defined on spaCy's documentation.

Logging to Rubrix

After defining our labelling functions, it's time to effectively annotate our documents.

First we annotate the development set with gold labels, and add the weak labels of our labelling functions.

```
[ ]: from spacy.tokens import Span

def annotate_dataset(dataset, tokens_field="tokens", label_field="ner_tags", gold_field=
    ↪ "gold"):
    for row in tqdm(dataset):
        doc = Doc(Vocab(), words=row[tokens_field])
        ner_tags = dataset.features[label_field].feature.int2str(row[label_field])
        offsets = biluo_tags_to_offsets(doc, job_to_biluo(ner_tags))
        spans = [ doc.char_span(x[0], x[1], label=x[2]) for x in offsets ]
        doc.spans[gold_field] = spans
        yield doc

dev_docs = list(annotate_dataset(conll2003["validation"]))
dev_docs = list(rule_based_annotator.pipe(dev_docs))

```

Then we will log records to Rubrix, for which any of the labelling functions triggered a weak label, or for which we have a gold annotation. In this way we will be able to quickly visualize any bugs or missing edge cases which may not yet be covered by our labelling functions.

We also add a metadata doc_index that will allow us to group distinct labelling functions for the same document.

```
[ ]: def spans_logger(docs, dataset="conll_2003_spans"):
    def unroll_spans(span_list):
        return [ (span.label_, span.start_char, span.end_char) for span in span_list ]

    for idx, doc in enumerate(tqdm(docs)):
        tokens = [token.text for token in doc]

        if tokens == []:

```

(continues on next page)

(continued from previous page)

```

        continue

    predictions, annotations = {}, None
    for labelling_function, span_list in doc.spans.items():
        if labelling_function == "gold":
            annotations = unroll_spans(span_list)
        else:
            predictions[labelling_function] = unroll_spans(span_list)

    # add records for each labelling function, if they made a prediction
    for agent, prediction in predictions.items():
        if prediction:
            yield rb.TokenClassificationRecord(
                text=" ".join(tokens),
                token=tokens,
                prediction=prediction,
                prediction_agent=agent,
                annotation=annotations,
                metadata={"doc_index": idx}
            )

    # add records with annotations, for which no labelling function triggered
    if not any(predictions.values()) and annotations:
        yield rb.TokenClassificationRecord(
            text=" ".join(tokens),
            tokens=tokens,
            annotation=annotations,
            metadata={"doc_index": idx}
        )

rb.log(records=spans_logger(dev_docs), name="conll_2003_dev_spans")

```

The screenshot shows the Rubrix web interface for a document titled 'conll_2003_dev_spans'. The interface includes a search bar, tabs for Predictions (1), Annotations, Status, Metadata, and Sort, and a Records (11) count. A modal is open for editing a prediction, with fields for 'Predicted as:', 'Predicted ok:', and 'Predicted by:'. The document text is displayed with various spans highlighted in different colors (yellow, blue, green, red) representing different entity types. A specific span is highlighted in red and labeled 'pred. ORG'.

3. Evaluate the precision of our rules

After getting a bird's-eye view of our annotations with Rubrix, we can use `skweak`'s `LFAAnalysis` to numerically evaluate the precision of our rules.

We want to eliminate rules from our combined annotator that have very low precision scores, as this may negatively affect the performance of a model trained on our annotated data.

[38]: *# We evaluate the precision of our heuristic rules*

```
from skweak.analysis import LFAAnalysis
import pandas as pd

lf_analysis = LFAAnalysis(
    dev_docs,
    ["ORG"]
)

scores = lf_analysis.lf_empirical_scores(
    dev_docs,
    gold_span_name="gold",
    gold_labels=["ORG", "MISC", "PER", "LOC", "O"]
)

def scores_to_df(scores):
    for annotator, label_dict in scores.items():
        for label, metrics_dict in label_dict.items():
            row = {
```

(continues on next page)

(continued from previous page)

```

        "annotator": annotator,
        "label": label,
        "precision": metrics_dict["precision"],
        "recall": metrics_dict["recall"],
        "f1": metrics_dict["f1"]
    }
    yield row

evaluation_df = pd.DataFrame(list(scores_to_df(scores)))\
    .round(3)\
    .sort_values(["label", "precision"], ascending=False)\
    .reset_index(drop=True)
evaluation_df[["annotator", "label", "precision"]]

```

```
[38]:
```

	annotator	label	precision
0	Corp (end)	ORG	1.000
1	Organisation (end)	ORG	1.000
2	Group (end)	ORG	1.000
3	Council (end)	ORG	1.000
4	Department (end)	ORG	1.000
5	Exchange (end)	ORG	1.000
6	Bureau (end)	ORG	1.000
7	Corporation (end)	ORG	1.000
8	Ltd (end)	ORG	1.000
9	sports_results	ORG	1.000
10	gold	ORG	1.000
11	sports_match	ORG	1.000
12	Party (end)	ORG	1.000
13	Newsroom (end)	ORG	1.000
14	Army (end)	ORG	1.000
15	Inc (end)	ORG	1.000
16	shire_team	ORG	0.982
17	New (start)	ORG	0.909
18	U.N. (start)	ORG	0.882
19	Association (end)	ORG	0.800
20	First (start)	ORG	0.800
21	United (start)	ORG	0.800
22	Federal (start)	ORG	0.714
23	National (start)	ORG	0.640

4. Annotate the training data and aggregate the weak labels

Aggregation

After carefully considering which rules are appropriate for our dataset, we will annotate the training data and then aggregate our annotations into a single layer.

skweak includes an aggregation model called **majority voter**. It considers each labelling function as a voter and outputs the most frequent label. We will utilize this majority voter to produce a single set of annotations for our documents, and then we will log the results to Rubrix.

The majority voter is particularly useful when annotating for multiple labels, as in this case the annotations produced by the heuristic rules may not only overlap and but also conflict with each other. However, as we are annotating only

for the ORG label, we won't need the majority voter to resolve any conflicts: it will simply merge the labels from each annotator into the `maj_voter` field.

```
[ ]: # Create the training docs and annotate them with heuristic rules

train_docs = [ Doc(locat(), words=row["tokens"]) for row in conll2003["train"] ]
train_docs = list(rule_based_annotator.pipe(train_docs))

[ ]: # Perform majority voting over the training data

from skweak.aggregation import MajorityVoter
voter = MajorityVoter("maj_voter", label=["ORG"], sequence_labelling=True)
train_docs = list(voter.pipe(train_docs))

[ ]: # Log to Rubrix

rh.log(records=spans_logger(train_docs), name="conll_2003_train")
```

Although here we are using the majority voter in a rather simple way to vote for a single ORG label, it is possible to attribute weights to the vote of each labelling function and even define complex hierarchies between labels. These details are explained in the majority voter [documentation](#) and [code](#) on the skweak repository.

The screenshot displays the Rubrix web interface for the 'conll_2003_train' dataset. The interface includes a search bar, tabs for 'Predictions (2)', 'Status', 'Metadata', 'Annotations', and 'Sort', and a 'Records (999)' indicator. A filter for 'ORG [1]' is active. A modal window shows the prediction details for a selected record: 'Predicted as: ORG' and 'Predicted by: maj_voter'. The table lists several records, including 'Brentford 3 2 1 0 6 3 7', 'Kongsvinger 20 7 4 9 26 38 25', 'Swansea 1 Lincoln 2', 'At Trent Bridge : Nottinghamshire 392-6 (G. Archer 143 not', and 'Vojvodina 2 2 0 0 4 1 6'. A tooltip shows 'pred. ORG' for the 'At Trent Bridge' record. The bottom of the interface shows 'Records per page: 5' and '1-5 of 999'.

Generating the training data

Our final annotations should be set to the field `ents` of our spaCy Doc objects.

We set the labels defined by our majority voter for the training set, and the gold labels for the development set.

```
[ ]: for doc in train_docs:
    doc.set_ents(doc.spans.get("maj_voter", []))

for doc in dev_docs:
    org_ents = filter(lambda token: token.label_ == "ORG", doc.spans.get("gold", []))
    doc.set_ents(org_ents)
```

In order to avoid training on an unbalanced dataset, we make sure that we have the same amount of annotated and blank records in our training data.

```
[ ]: import random
    random.seed(42)
    annotated_docs = [ doc for doc in train_docs if doc.ents ]
    empty_docs = random.sample([ doc for doc in train_docs if not doc.ents ], len(annotated_docs))
    train_docs_sample = annotated_docs + empty_docs
```

Finally, we use skweak's `docbin_writer` to write our training and development sets to a binary file format that is compatible with spaCy's command line tools.

```
[ ]: # Save the training and development data.

from skweak.utils import docbin_writer

docbin_writer(train_docs_sample, "/tmp/train.spacy")
docbin_writer(dev_docs, "/tmp/dev.spacy")
```

5. Evaluate the baseline

Before we train and evaluate our own solution, let's test a simple model to see what is possible to achieve without weak supervision. In this way we can see if our solution is able to improve on this baseline.

We evaluate the `en_core_web_lg` spaCy model on CoNLL 2003. The model has been trained on a distinct dataset, OntoNotes 5.0. We do not perform any sort of adaptation on the model and evaluate its zero-shot performance on the development set.

As it can be seen below, our baseline was able to achieve an F-score of **37,3%**.

```
[49]: from spacy.training import Example
    from spacy.scorer import Scorer
    import spacy

    nlp = spacy.load("en_core_web_lg")
    dev_eval_docs = [ nlp(" ".join(row["tokens"])) for row in conll2003["validation"] ]

    for doc in dev_eval_docs:
        doc.set_ents(list(filter(lambda x: x.label_ == "ORG", doc.ents)))
```

(continues on next page)

(continued from previous page)

```

scorer_object = Scorer()
scores = scorer_object.score([ Example(dev_eval_docs[i], dev_docs[i]) for i in range(0,
→len(dev_docs)) ])

pd.DataFrame([{'k': v for k, v in scores.items() if k in ["ents_p", "ents_r", "ents_f"]}).
→round(3)

```

```

[49]:      ents_p  ents_r  ents_f
0      0.453   0.317   0.373

```

6. Train and evaluate our model

Here we train and evaluate a spaCy model on the training data annotated by our heuristic rules.

We initialize our NER model with vectors from our baseline model, `en_core_web_lg`, and train it for 400 steps.

Our model was able to achieve a F-score of 43.67%, which is a **6,37% improvement** over our baseline.

```

[20]: # After training our model on data annotated with heuristic rules, we reach an F score
→of 44%, which is 7% above the baseline.

```

```

!spacy init config --lang en --pipeline ner --optimize accuracy | \
spacy train - \
--training.max_steps 400 \
--system.seed 42 \
--paths.train /tmp/train.spacy \
--paths.dev /tmp/dev.spacy \
--initialize.vectors en_core_web_lg \
--output /tmp/model

```

```

Saving to output directory: /tmp/model
Using CPU

```

```

===== Initializing pipeline =====
[2022-01-27 12:38:58,091] [INFO] Set up nlp object from config
[2022-01-27 12:38:58,102] [INFO] Pipeline: ['tok2vec', 'ner']
[2022-01-27 12:38:58,107] [INFO] Created vocabulary
[2022-01-27 12:38:59,423] [INFO] Added vectors: en_core_web_lg
[2022-01-27 12:39:00,729] [INFO] Finished initializing nlp object
[2022-01-27 12:39:23,582] [INFO] Initialized pipeline components: ['tok2vec', 'ner']
  Initialized pipeline

```

```

===== Training pipeline =====
Pipeline: ['tok2vec', 'ner']
Initial learn rate: 0.001

```

E	#	LOSS TOK2VEC	LOSS NER	ENTS_F	ENTS_P	ENTS_R	SCORE
0	0	0.00	39.17	2.97	2.26	4.33	0.03
0	200	69.40	1239.88	36.93	91.72	23.12	0.37
1	400	15.18	280.61	43.67	78.79	30.20	0.44

```

  Saved pipeline to output directory
/tmp/model/model-last

```

Summary

Writing precise heuristic rules is a key component to weak supervision workflows with skweak. Rubrix makes it easier for us to identify patterns in our data, create new rules and then debug our labelling functions. In this way we are able to accelerate our data annotation pipelines and quickly train new models that score above common zero-shot baselines.

Next steps

Rubrix [Github repo](#) to stay updated.

[Rubrix documentation](#) for more guides and tutorials.

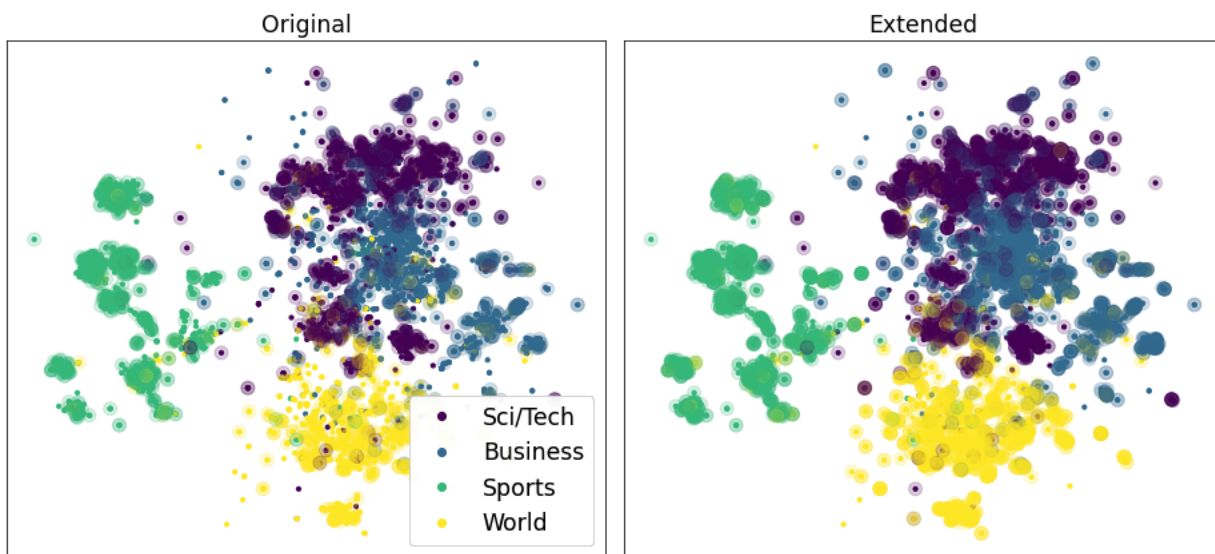
Join the Rubrix community! A good place to start is the [discussion forum](#).

[]:

4.16.3 Extending weak supervision workflows with sentence embeddings

In this tutorial, we show how weak supervision workflows in Rubrix can be extended with sentence embeddings. We start from the weak supervision workflow presented in our *Weak supervision with Rubrix tutorial* and improve on its results by extending the coverage of its rules.

- We define rules and generate weak labels for the `ag_news` data set.
- We extend our weak labels with sentence embeddings from the [Sentence Transformers](#) library.
- Finally, we use a label model to generate data for training a downstream model as a news classifier.
- We achieve a **4% improvement** in accuracy over the original workflow simply by extending our weak labels.



The two plots above show the coverage of the weak labels before and after extending them with embeddings. Each point corresponds to an example in the ag news test set. The color indicates the corresponding class of the example. Points in a transparent circle are covered by at least one rule.

Introduction

Labelling functions normally have high precision, but low coverage. Only records that strictly match the conditions determined by a given function will be labelled, while other potential candidates will be left out.

Building on [the findings of the Hazy Research group](#), we present a way to solve this problem by extending the weak labels produced by our labelling functions with sentence embeddings.

We extend the coverage of our labelling functions by giving unlabelled records the same label as their nearest labelled neighbor in the embedding space if the cosine similarity between them scores above a certain threshold.

We will show in this tutorial that, by adjusting these similarity thresholds and selecting proper sentence embeddings, we are able to significantly improve the accuracy of the downstream classifiers produced by our weak supervision workflows.

Detailed Workflow

A typical workflow to perform weak supervision with sentence embeddings is:

1. Create a Rubrix dataset with your raw dataset. If you have some labelled data, you can log it into the the same dataset.
2. Define a set of weak labeling rules with the Rules definition mode in the UI.
3. Create a `WeakLabels` object and apply the rules. You can load the rules from your dataset and add additional rules and labeling functions using Python. Typically, you'll iterate between this step and step 2.
4. Extend the `WeakLabels` object by giving sentence embeddings for each record (the rows of the matrix) and a similarity threshold for each rule (the columns of the matrix).
5. Once you are satisfied with your extended weak labels, use the extended matrix of the `WeakLabels` instance with your library/method of choice to build a training set or even train a downstream text classification model. You can iterate between this step and step 4 to try several thresholds and embeddings possibilities until you achieve a satisfactory result.

This guide shows you an end-to-end example using Snorkel. You could alternatively use any other label model available in Rubrix. If you are interested in learning about other options, please check our [weak supervision guide](#).

Setup

Rubrix, is a free and open-source tool to explore, annotate, and monitor data for NLP projects.

If you are new to Rubrix, check out the [Github repository](#).

If you have not installed and launched Rubrix yet, check the [Setup and Installation guide](#).

For this tutorial we also need some third party libraries that can be installed via pip:

```
[ ]: %pip install faiss-cpu sentence_transformers transformers datasets
```


The dataset

Since this tutorial is an extension of our *Weak supervision with Rubrix tutorial*, we will also use the `ag_news` dataset, a well-known benchmark text classification models.

However, to guarantee a fair comparison, we will optimize the thresholds on a validation split, and leave the test split for the final evaluation.

```
[ ]: from datasets import load_dataset

agnews = load_dataset("ag_news")

agnews_train, agnews_valid = agnews["train"].train_test_split(test_size=7600, seed=43).
↪ values()
```

1. Create a Rubrix dataset with unlabelled data and test data

Just like in the *first tutorial*, let's load a labelled and unlabelled set of records into Rubrix.

```
[ ]: import rubrix as rb

# build our labelled records to evaluate our heuristic rules and optimize the thresholds
records = [
    rb.TextClassificationRecord(
        text=record["text"],
        metadata={"split": "labelled"},
        annotation=agnews_valid.features["label"].int2str(record["label"]),
        id=f"valid_{idx}",
    )
    for idx, record in enumerate(agnews_valid)
]

# build our unlabelled records
records += [
    rb.TextClassificationRecord(
        text=record["text"],
        metadata={"split": "unlabelled"},
        id=f"train_{idx}",
    )
    for idx, record in enumerate(agnews_train)
]

# log the records to Rubrix
rb.log(records, name="news2")
```

After this step, you have a fully browsable dataset available that you can access via the [Rubrix web app](#).

2. Defining rules

We will use the same rules as found in the *previous tutorial*.

```
[ ]: from rubrix.labeling.text_classification import Rule

# define queries and patterns for each category (using ES DSL)
queries = [
    (["money", "financ*", "dollar*"], "Business"),
    (["war", "gov*", "minister*", "conflict"], "World"),
    (["footbal*", "sport*", "game", "play*"], "Sports"),
    (["sci*", "techno*", "computer*", "software", "web"], "Sci/Tech")
]

# define rules
rules = [
    Rule(query=term, label=label)
    for terms, label in queries
    for term in terms
]
```

3. Building and analyzing weak labels

After building weak labels from our rules, their summary reveals that our rules have, in total, 31% coverage while achieving 74% precision.

```
[ ]: from rubrix.labeling.text_classification import WeakLabels

# apply the rules to the dataset to obtain the weak labels
weak_labels = WeakLabels(
    rules=rules,
    dataset="news2"
)
```

```
[5]: weak_labels.summary()
```

```
[5]:
```

	label	coverage	annotated_coverage \
money	{Business}	0.008242	0.008816
financ*	{Business}	0.019775	0.021184
dollar*	{Business}	0.016608	0.016974
war	{World}	0.011683	0.008816
gov*	{World}	0.045067	0.043158
minister*	{World}	0.030142	0.030263
conflict	{World}	0.003050	0.003684
footbal*	{Sports}	0.013050	0.015132
sport*	{Sports}	0.021183	0.021711
game	{Sports}	0.038950	0.043026
play*	{Sports}	0.052608	0.057632
sci*	{Sci/Tech}	0.016433	0.015658
techno*	{Sci/Tech}	0.027150	0.028816
computer*	{Sci/Tech}	0.027275	0.026447
software	{Sci/Tech}	0.030283	0.032763
web	{Sci/Tech}	0.015508	0.016316

(continues on next page)

(continued from previous page)

total	{Sci/Tech, World, Sports, Business}		0.317375		0.327895
	overlaps	conflicts	correct	incorrect	precision
money	0.002450	0.001925	31	36	0.462687
financ*	0.005892	0.005183	115	46	0.714286
dollar*	0.003492	0.002850	98	31	0.759690
war	0.003242	0.001367	44	23	0.656716
gov*	0.010800	0.006225	156	172	0.475610
minister*	0.007508	0.002825	207	23	0.900000
conflict	0.001025	0.000092	20	8	0.714286
footbal*	0.004875	0.000408	105	10	0.913043
sport*	0.007033	0.001225	146	19	0.884848
game	0.014067	0.002375	253	74	0.773700
play*	0.016767	0.004992	312	126	0.712329
sci*	0.002742	0.001275	101	18	0.848739
techno*	0.008325	0.003108	153	66	0.698630
computer*	0.011100	0.004483	167	34	0.830846
software	0.009625	0.003308	202	47	0.811245
web	0.004100	0.001608	111	13	0.895161
total	0.053408	0.019425	2221	746	0.748568

In the next steps, we will try to extend our weak labels matrix through sentence embeddings. In this way, we will increase the coverage of our rules, while maintaining an acceptable precision.

4. Using the weak labels

Label model with Snorkel

Snorkel's label model is by far the most popular option for using weak supervision, and Rubrix provides built-in support for it. Here we fit our weak labels to the Snorkel label model, and then we check the performance on the records that have been covered by the rules.

```
[6]: from rubrix.labeling.text_classification import Snorkel

# create the the Snorkel label model
label_model = Snorkel(weak_labels)

# fit the model, for the learning rate and epochs we ran a quick grid search
label_model.fit(lr=0.002, n_epochs=10, progress_bar=False)

# evaluate the label model
print(label_model.score(output_cls=True))
```

	precision	recall	f1-score	support
Business	0.73	0.41	0.53	493
Sports	0.77	0.97	0.86	703
World	0.69	0.83	0.75	462
Sci/Tech	0.80	0.74	0.77	833
accuracy			0.76	2491
macro avg	0.75	0.74	0.73	2491

(continues on next page)

(continued from previous page)

weighted avg	0.76	0.76	0.74	2491
--------------	------	------	------	------

5. Extending the weak labels

Let's extend our weak labels and see how that impacts the evaluation of the Snorkel label model.

Generate sentence embeddings

Let's generate sentence embeddings for each record of our weak labels matrix. Best results will be achieved through powerful general-purpose pretrained embeddings, or by embeddings specifically pretrained for the domain of the task at hand.

Here we choose the `all-mpnet-base-v2` embeddings from the well-known [Sentence Transformers library](#). Rubrix allows us to experiment with embeddings from any source, as long as they are provided to the weak labels matrix as a two-dimensional array.

For instance, instead of Sentence Transformers, we could have utilized GPT-3 [similarity embeddings](#) from the OpenAI Embeddings API, or text embeddings from the [Tensorflow Hub](#), or we could even have trained our own embeddings from scratch.

```
[ ]: from sentence_transformers import SentenceTransformer
    from tqdm.auto import tqdm

    # instantiate the model for the sentence embeddings
    # we strongly recommend using a GPU for the computation of the embeddings
    model = SentenceTransformer('all-mpnet-base-v2', device='cuda')

    # compute the embeddings and store them in a list
    embeddings = []
    for rec in tqdm(weak_labels.records()):
        embeddings.append(model.encode(rec.text))
```

Set the thresholds

We start by making an educated guess on which thresholds will work for this particular weak labels matrix. We set the thresholds for all rules to 0.60. This means that, for each rule, the label of a record will be extended to its nearest unlabelled neighbor if their cosine similarity is above this value.

```
[ ]: thresholds = [0.6] * len(rules)
```

Extend the weak labels matrix

We call the `extend_matrix` method by providing the thresholds and the sentence embeddings.

```
[ ]: weak_labels.extend_matrix(thresholds, embedding)
```

With the weak label matrix extended, we can check that our coverage went up significantly (from 0.32 to 0.79).

```
[10]: weak_labels.summary()
```

```
[10]:
```

	label	coverage	annotated_coverage	\
money	{Business}	0.079342	0.083158	
financ*	{Business}	0.122692	0.130526	
dollar*	{Business}	0.104917	0.110789	
war	{World}	0.083958	0.081053	
gov*	{World}	0.218817	0.216447	
minister*	{World}	0.124367	0.121974	
conflict	{World}	0.038975	0.036579	
football*	{Sports}	0.040083	0.043289	
sport*	{Sports}	0.114292	0.111053	
game	{Sports}	0.125608	0.130526	
play*	{Sports}	0.205158	0.206053	
sci*	{Sci/Tech}	0.058375	0.056053	
techno*	{Sci/Tech}	0.117850	0.124079	
computer*	{Sci/Tech}	0.100017	0.097632	
software	{Sci/Tech}	0.088967	0.088816	
web	{Sci/Tech}	0.084800	0.081579	
total	{Sci/Tech, Sports, Business, World}	0.793542	0.793553	

	overlaps	conflicts	correct	incorrect	precision
money	0.068042	0.061908	342	290	0.541139
financ*	0.096475	0.087525	596	396	0.600806
dollar*	0.084467	0.077458	552	290	0.655582
war	0.069425	0.050867	353	263	0.573052
gov*	0.157083	0.118283	843	802	0.512462
minister*	0.091817	0.055133	752	175	0.811219
conflict	0.035533	0.023192	199	79	0.715827
football*	0.029475	0.007708	308	21	0.936170
sport*	0.084200	0.031992	745	99	0.882701
game	0.094600	0.035858	745	247	0.751008
play*	0.156583	0.091692	900	666	0.574713
sci*	0.035192	0.028675	257	169	0.603286
techno*	0.092108	0.074517	558	385	0.591729
computer*	0.079608	0.060317	553	189	0.745283
software	0.065025	0.046550	517	158	0.765926
web	0.067950	0.054550	415	205	0.669355
total	0.392908	0.228892	8635	4434	0.660724

We also see that the average precision of our rules went down (from 0.75 to 0.66). This drop, however, can be partially compensated by our label model. If we fit our weak labels to a Snorkel label model again, we can see that the support went up significantly, as expected, while the drop in accuracy is minor.

```
[12]: label_model = Snorkel(weak_labels)
label_model.fit(l=0.002, n_epoch=10, progress_bar=False)
```

(continues on next page)

(continued from previous page)

```
print(label_model.score(output_scs=True))
```

	precision	recall	f1-score	support
Sci/Tech	0.76	0.74	0.75	1636
World	0.67	0.86	0.76	1421
Sports	0.79	0.96	0.87	1544
Business	0.78	0.39	0.52	1430
accuracy			0.74	6031
macro avg	0.75	0.74	0.72	6031
weighted avg	0.75	0.74	0.73	6031

You can have a look at the *Appendix* to have a detailed explanation about how the weak label matrix is extended under the hood.

Instead of using generic fixed thresholds, we recommend to optimize them in some way to get the highest performance gains. Our optimization described in detail in the *Appendix* yielded following thresholds:

```
[ ]: optimized_thresholds = [0.4, 0.4, 0.6, 0.4, 0.5, 0.8, 1., 0.4, 0.4, 0.5, 0.6, 0.4, 0.4,
                             ↪0.6, 0.6, 0.8]
```

Each call to `extend_matrix` with thresholds and embeddings will build a `faiss` index that will be cached inside the weak labels object.

If we do not provide embeddings in our next calls to `extend_matrix`, this index will be reutilized, and a new extended matrix will replace the current extended matrix. So extending the matrix with new threshold is very cheap.

```
[28]: weak_labels.extend_matrix(optimized_thresholds)
label_model = SparkML(weak_labels)
label_model.fit(l=0.002, n_epochs=10, progress_bar=False)
print(label_model.score(output_scs=True))
```

	precision	recall	f1-score	support
Sci/Tech	0.74	0.67	0.70	1906
World	0.78	0.64	0.70	1789
Sports	0.82	0.90	0.86	1875
Business	0.60	0.69	0.64	1877
accuracy			0.73	7447
macro avg	0.73	0.73	0.73	7447
weighted avg	0.73	0.73	0.73	7447

The optimized thresholds seem to further reduce the accuracy of the label model, but also increase the coverage significantly.

6. Training a downstream model

Now we will train the same downstream model as in the *previous tutorial*, but on the data produced by a label model from our extended weak labels.

Let us first define a helper function that is basically a copy&paste from the previous tutorial.

```
[ ]: import pandas as pd
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.pipeline import Pipeline
from sklearn import metrics

def train_and_evaluate_downstream_model(label_model):
    """
    Train a downstream model with the predictions of a label model and
    evaluate it with the test split of the ag news dataset
    """
    # get records with the predictions from the label model
    records = label_model.predict()

    # turn str labels into integers
    label2int = label_model.weak_labels.label2int

    # extract training data
    X_train = [rec.text for rec in records]
    y_train = [label2int[rec.prediction[0][0]] for rec in records]

    # define our final classifier
    classifier = Pipeline([
        ('vect', CountVectorizer()),
        ('clf', MultinomialNB())
    ])

    # fit the classifier
    classifier.fit(
        X_train,
        y_train,
    )

    # extract text and labels
    X_test = [rec["text"] for rec in agnews["test"]]
    y_test = [label2int[agnews["test"].features["label"].int2str(rec["label"])] for rec in agnews["test"]]

    # get predictions for the test set
    predicted = classifier.predict(X_test)

    return metrics.classification_report(y_test, predicted, target_name=[k for k in label2int.keys() if k])
```

Now let's see how our downstream model compares with the original model from the *previous tutorial*. Remember we achieved an accuracy of around **82%**.

```
[35]: print(train_and_evaluate_downstream_model(label_model))
```

	precision	recall	f1-score	support
Sci/Tech	0.85	0.82	0.83	1900
World	0.90	0.84	0.87	1900
Sports	0.90	0.97	0.93	1900
Business	0.81	0.82	0.81	1900
accuracy			0.86	7600
macro avg	0.86	0.86	0.86	7600
weighted avg	0.86	0.86	0.86	7600

Now, with our extended weak label matrix, we were able to achieve an accuracy of **86%**, a 4% improvement over our original approach.

Summary

In this tutorial you have seen how to improve your weak supervision workflows in Rubrix using word embeddings. With very small changes to the original workflow, we were able to significantly increase the accuracy of our downstream models. This shows that Rubrix can greatly reduce the amount of effort that human annotators need to put into writing rules before they can achieve exceptional results.

Next steps

Rubrix [Github repo](#) to stay updated.

[Rubrix documentation](#) for more guides and tutorials.

Join the Rubrix community! A good place to start is the [discussion forum](#).

Appendix: Visualize changes

Let's visualize how the weak labels matrix is being extended in a single row.

```
[ ]: import pandas as pd

def get_transitions(weak_labels, id):
    transitions = list(list(zip(row[0], row[1])) for row in zip(weak_labels._matrix,
    ↪ weak_labels._extended_matrix))
    transitions = transitions[id]
    label_dict = weak_labels.int2label
    rule_labels = weak_labels.summary().reset_index()['index'].values.tolist()[:-1]
    transitions_df = []
    for rule_idx, rule in enumerate(rule_labels):
        old_label = transitions[rule_idx][0]
        new_label = transitions[rule_idx][1]
        transitions_df.append({
            "rule": rule,
            "old label": label_dict[old_label],
            "new label": label_dict[new_label],
```

(continues on next page)

(continued from previous page)

```

    })
    transitions_df = pd.DataFrame(transitions_df)
    text = weak_labels.records[0][0].text
    return transitions_df, text

transitions, text = get_transitions(weak_labels, 15)

```

By reading the selected record, we can clearly notice that it is a news article about world politics, and therefore should be classified as World.

[79]: text

```

[79]: 'Israel #39;determined to complete Gaza plan #39; Israel is determined to go ahead with
      ↳ its unilateral withdrawal from the Gaza Strip - regardless of the death of Yasser
      ↳ Arafat and even if settlers resist - a top Israeli general who helped design the plan
      ↳ says.'

```

Let's put side by side the row of the original weak labels matrix for this record (the "old label" row) and the same row after extension (the "new label" row).

We see that this news article was not labelled in the original matrix by any of our rules.

However, it was the nearest unlabelled neighbor of two Business articles, matched by the rules `financ*` and `dollar*`, and its similarity with them scored above our selected thresholds. The same happened for two World articles, matched by the rules `war` and `minister*`, and for a Sci/Tech article matched by the rule `sci*`.

[80]: transitions.transpose()

```

[80]:
      0      1      2      3      4      5      6  \
rule   money  financ*  dollar*  war  gov*  minister*  conflict
old label  None      None      None  None  None      None      None
new label  None  Business  Business  World  None      World      None

      7      8      9     10     11     12     13  \
rule  footbal*  sport*  game  play*     sci*  techno*  computer*
old label      None      None  None  None      None      None      None
new label      None      None  None  None  Sci/Tech      None      None

      14     15
rule   software  web
old label      None  None
new label      None  None

```

Appendix: Optimizing the thresholds

Each call to `extend_matrix` with thresholds and embeddings will build a `faiss` index that will be cached inside the weak labels object.

If we do not provide embeddings in our next calls to `extend_matrix`, this index will be reutilized, and a new extended matrix will replace the current extended matrix. This new matrix is an extension of the original weak labels matrix made according to our new similarity thresholds.

```

[ ]: # Let's try to set all thresholds to 0.8 instead of 0.6.
     thresholds = [0.8] * len(rules)

```

(continues on next page)

(continued from previous page)

```
# As we have already generated the index in our first call, we just need to provide the
↳ thresholds.
weak_labels.extend_matrix(thresholds)
```

There are a few different approaches to find the best similarity thresholds for extending a weak labels matrix: we will list them from the least to the most computationally expensive.

1. Block the extension of low overlap rules

After setting all similarity thresholds to a reasonable value, a good way to optimize the similarity thresholds on an individual level is to block the extension of rules with low overlap, as they are more likely to produce inaccurate results after extension.

```
[111]: summary = weak_labels.summary(normalize_by_coverage=True).reset_index().head(len(rules))
summary = summary.rename(columns={"index": "rule"})
summary = summary.sort_values(by="overlaps", ascending=True)[["rule", "overlaps"]]
summary = summary.reset_index()
summary
```

```
[111]:
```

	index	rule	overlaps
0	4	gov*	0.239645
1	15	web	0.264374
2	14	software	0.317832
3	10	play*	0.318707
4	6	conflict	0.336066
5	9	game	0.361147
6	8	sport*	0.393875
7	11	sci*	0.483512
8	5	minister*	0.514205
9	7	footbal*	0.567152
10	13	computer*	0.703716
11	12	techno*	0.710083
12	1	financ*	0.737078
13	2	dollar*	0.742097
14	3	war	0.744417
15	0	money	0.788047

```
[112]: thresholds = [0.6] * len(rules)

# Let's block the extension of the top 5 rules with the least overlap.
turn_off_index = summary['index'][0:6]

# We block the extension of a rule by setting its similarity threshold to 1.0.
for rule_index in turn_off_index:
    thresholds[rule_index] = 1.0

weak_labels.extend_matrix(thresholds)
label_model = Snorkel(weak_labels)
label_model.fit(l=0.002, n_epochs=10, progress_bar=False)
print(train_and_evaluate_downstream_model(label_model))
```

	precision	recall	f1-score	support
Sci/Tech	0.81	0.84	0.82	1900
World	0.90	0.87	0.88	1900
Sports	0.91	0.96	0.93	1900
Business	0.81	0.76	0.78	1900
accuracy			0.86	7600
macro avg	0.85	0.86	0.85	7600
weighted avg	0.85	0.86	0.85	7600

2. Brute force: Grid search over the label model

In this approach, we set all thresholds to an initial value, and then grid search for the best value for each one of them individually. Then we optimize for the harmonic mean between the coverage and the accuracy of the label model on the development set. This will ensure that we choose the thresholds with the best trade-off between both metrics.

We arrive at the same improvement as the previous approach, with a final accuracy of 86% over the test set.

```
[ ]: def train_eval_labelmodel(th):
    weak_labels.extend_matrix(th)

    label_model = Snorkel(weak_labels)
    label_model.fit(lr=0.002, n_epochs=10, progress_bar=False)

    metrics = label_model.score()
    acc, sup, n = metrics["accuracy"], metrics["macro avg"]["support"], len(weak_labels.
    ↳ annotation())
    coverage = sup / n
    return 2 * acc * coverage / (acc + coverage)
```

```
[ ]: import copy
    from tqdm.auto import tqdm

    ths_range = np.arange(1, 0.3, -0.1)
    n_ths = len(weak_labels.rules)

    best_thresholds = [1.0] * n_ths
    best_acc = 0.0
    for i in tqdm(range(n_ths), total=n_ths):
        thresholds = best_thresholds.copy()
        for threshold in ths_range:
            thresholds[i] = threshold
            acc = train_eval_labelmodel(thresholds)
            if acc > best_acc:
                best_acc = acc
                best_thresholds = thresholds.copy()
```

```
[121]: np.array(best_thresholds)
```

```
[121]: array([0.4, 0.4, 0.6, 0.4, 0.5, 0.8, 1. , 0.4, 0.4, 0.5, 0.6, 0.4, 0.4,
        0.6, 0.6, 0.8])
```

```
[122]: weak_labels.extend_matrix(best_threshold)
label_model = Snorkel(weak_labels)
label_model.fit(lr=0.002, n_epochs=10, progress_bar=False)
print(train_and_evaluate_downstream_model(label_model))
```

	precision	recall	f1-score	support
Sci/Tech	0.83	0.83	0.83	1900
World	0.89	0.85	0.87	1900
Sports	0.90	0.97	0.93	1900
Business	0.82	0.79	0.80	1900
accuracy			0.86	7600
macro avg	0.86	0.86	0.86	7600
weighted avg	0.86	0.86	0.86	7600

3. Brute force: Grid search over the downstream model

Here again we set all thresholds to an initial value and grid search for the best value for each individual threshold, but now we optimize for the accuracy of the downstream model on the development set. We arrive at a final accuracy of 85% on the test set, which is slightly less than what we achieved through the previous approaches.

```
[126]: # retrieve records with annotations
test_ds = weak_labels.records(has_annotation=True)

# extract text and labels
X_test_for_grid_search = [rec.text for rec in test_ds]
y_test_for_grid_search = [weak_labels.label2int[rec.annotation] for rec in test_ds]

def train_eval_downstream(th):
    weak_labels.extend_matrix(th)

    label_model = Snorkel(weak_labels)
    label_model.fit(lr=0.002, n_epochs=10, progress_bar=False)

    records = label_model.predict()

    X_train = [rec.text for rec in records]
    y_train = [weak_labels.label2int[rec.prediction[0][0]] for rec in records]

    classifier = Pipeline([
        ('vect', CountVectorizer()),
        ('clf', MultinomialNB())
    ])

    classifier.fit(
        X=X_train,
        y=y_train,
```

(continues on next page)

(continued from previous page)

```

    )

    accuracy = classifier.score(
        X=X_test_for_grid_search,
        y=y_test_for_grid_search,
    )

    return accuracy

```

```

[ ]: from copy import copy
    from tqdm.auto import tqdm

    best_thresholds, best_acc = [1.0] * len(weak_labels.rules), 0
    ths_range = np.arange(1, 0.3, -0.1)
    n_ths = len(weak_labels.rules)

    for i in tqdm(range(n_ths), total=n_ths):
        thresholds = best_thresholds.copy()
        for threshold in ths_range:
            thresholds[ ] = threshold
            acc = train_eval_downstream(thresholds)
            if acc > best_acc:
                best_acc = acc
                best_thresholds = thresholds.copy()

```

```
[128]: np.array(best_thresholds)
```

```
[128]: array([0.6, 0.7, 0.9, 1. , 1. , 0.8, 0.7, 1. , 0.6, 1. , 1. , 0.7, 0.8,
           0.7, 0.9, 0.8])
```

```

[129]: weak_labels.extend_matrix(best_thresholds)
    label_model = Snorkel(weak_labels)
    label_model.fit(lr=0.002, n_epoch=10, progress_bar=False)
    print(train_and_evaluate_downstream_model(label_model))

```

	precision	recall	f1-score	support
Sci/Tech	0.81	0.82	0.82	1900
World	0.89	0.85	0.87	1900
Sports	0.88	0.98	0.93	1900
Business	0.82	0.75	0.78	1900
accuracy			0.85	7600
macro avg	0.85	0.85	0.85	7600
weighted avg	0.85	0.85	0.85	7600

Tips on threshold optimization

Grid search with large downstream models, such as transformers, can be very expensive. In this scenario, we can consider to optimize only a subset of the thresholds, or to optimize all thresholds on a small sample of the development set.

Although in this tutorial we perform grid search sequentially, there is no impediment to speed it up by performing it in parallel, as long as we make deep copies of the weak labels object for each process or thread.

Appendix: Plot extension

```
[ ]: import umap
import matplotlib.pyplot as plt

umap_data = umap.UMAP(n_neighbors=15, n_components=2, min_dist=0.0, metric='cosine').fit_
↳ transform(embeddings)

df = rb.DatasetForTextClassification(weak_labels.records()).to_pandas()
df["x"], df["y"] = umap_data[:, 0], umap_data[:, 1]
df["wl"] = [em for em in weak_labels.matrix]
df["wl_ext"] = [em for em in weak_labels.extended_matrix]

cov_idx = df["wl"].map(lambda x: x.sum() != -16)
cov_ext_idx = df["wl_ext"].map(lambda x: x.sum() != -16)
test_idx = ~(df.annotation.isna())

df_test = df[test_idx]
df_cov, df_cov_ext = df[cov_idx & test_idx], df[cov_ext_idx & test_idx]

label2int = {label: i for i, label in enumerate(df_test.annotation.value_counts().index)}

fig, ax = plt.subplots(1, 2, figsize=(13, 6), )

ax[0].scatter(df_test.x, df_test.y, c=df_test.annotation.map(lambda x: label2int[x]),
↳ =10)
↳ =10)
ax[0].scatter(df_cov.x, df_cov.y, c=df_cov.annotation.map(lambda x: label2int[x]),
↳ =100,
↳ alpha=0.2)

scatter = ax[1].scatter(df_test.x, df_test.y, c=df_test.annotation.map(lambda x:
↳ label2int[x]),
↳ =10)
↳ =10)
ax[1].scatter(df_cov_ext.x, df_cov_ext.y, c=df_cov_ext.annotation.map(lambda x:
↳ label2int[x]),
↳ =100, alpha=0.2)

ax[0].set_title("Original", {"fontsize": "xx-large"})
ax[0].set_xticks([]), ax[0].set_yticks([])

ax[1].set_title("Extended", {"fontsize": "xx-large"})
ax[1].set_xticks([]), ax[1].set_yticks([])

labels = list(scatter.legend_elements())
labels[1] = list(label2int.keys())
legend1 = ax[0].legend(*labels, loc="lower right", fontsize="xx-large")
```

(continues on next page)

(continued from previous page)

```
ax[0].add_artist(legend)

fig.tight_layout()
plt.savefig("extend_weak_labels.png", facecolor='white', transparent=False)
```

4.17 Active Learning

These tutorials show you how to create Active Learning workflows with Rubrix.

The screenshot displays the Rubrix web interface for an active learning tutorial. The top navigation bar is blue with the Rubrix logo and the text 'rubrix / active_learning_tutorial'. Below the navigation bar, there's a search bar labeled 'Introduce a query' and tabs for 'Predictions', 'Annotations', and 'Status'. The 'Status' tab is selected, showing a list of records. Each record has a checkbox, a text description, and two buttons labeled 'HAM' and 'SPAM' with '50,00 %' next to them. A dropdown menu is open over the 'Status' tab, showing 'Select options', a search bar, and two options: 'Default (4)' and 'Validated (6)', each with a checkbox. The 'Validated (6)' option is selected. The bottom of the interface shows 'Records per page: 10' and '1-10 of 10'.

Active learning with ModAL and scikit-learn Build an active learning prototype with Rubrix, ModAL and scikit-learn to filter spam from the YouTube Spam Collection dataset.

The screenshot shows the Rubrix UI interface for an active learning task. The top navigation bar includes 'Datasets / rubrix / trec_with_active_learning'. The main interface has tabs for 'Annotations', 'Status (1)', 'Metadata', and 'Sort'. A search bar is present with the text 'Introduce a query'. Below this, there are three text prompts for classification:

- TEXT: What is SVHS ? (Buttons: NUM, LOC, HUM, ENTY, DESC, ABBR)
- TEXT: What was the name of the horse that fell on Queen Elizabeth , Prince Albert 's wife ? (Buttons: NUM, LOC, HUM, ENTY, DESC, ABBR)
- TEXT: What major league baseball team compiled the best won-lost record between 1957 and 1983 ? (Buttons: NUM, LOC, HUM, ENTY, DESC, ABBR)

On the right, the 'Progress' section shows a total of 35/40 records with 87.50% completion. A table lists the distribution of labels:

Label	Count
DESC	11
ENTY	9
LOC	6
HUM	4
NUM	3
ABBR	2

The bottom of the interface shows 'Records per page: 20' and '1-5 of 5'.

Active learning for text classification with small-text Set up a complete active learning loop in the Rubrix UI with small-text and a Hugging Face transformer using a Rubrix listener.

4.17.1 Active learning with ModAL and scikit-learn

In this tutorial, we will walk through the process of building an active learning prototype with *Rubrix*, *ModAL* and *scikit-learn*.

- We train a spam filter using the YouTube Spam Collection data set.
- For this we embed a lightweight scikit-learn classifier in an active learner via ModAL.
- We design an active learning loop around *Rubrix*, to quickly build up a training data set from scratch.

Introduction

Active learning is a special case of machine learning in which a learning algorithm can interactively query a user (or some other information source) to label new data points with the desired outputs. In statistics literature, it is sometimes also called optimal experimental design. The information source is also called teacher or oracle. [Wikipedia]

In this tutorial **our goal is to show you how to incorporate Rubrix into an active learning workflow involving a human in the loop**. We will build a simple text classifier by combining **scikit-learn**, the active learning framework **ModAL** and **Rubrix**. Scikit-learn will provide the model that we will embed in an active learner from ModAL, and you and *Rubrix* will serve as the information source that teach the model to become a sample efficient classifier.

This tutorial is only a proof of concept for educational purposes and to inspire you with some ideas involving interactive learning processes, and how they can help to quickly build a training data set from scratch.

Setup

Rubrix, is a free and open-source tool to explore, annotate, and monitor data for NLP projects.

If you are new to Rubrix, check out the [Github repository](#).

If you have not installed and launched Rubrix yet, check the [Setup and Installation guide](#).

For this tutorial we also need the third party libraries modAL, scikit-learn and matplotlib (optional), which can be installed via pip:

```
[ ]: %pip install modAL scikit-learn matplotlib -qqq # matplotlib is optional
```

1. Loading and preparing data

Rubrix allows you to log and track data for different NLP tasks, such as Token Classification or Text Classification.

In this tutorial, we will use the [YouTube Spam Collection](#) dataset, which is a binary classification task for detecting spam comments in YouTube videos.

Let's load the data and have a look at it:

```
[1]: import pandas as pd

train_df = pd.read_csv("data/active_learning/train.csv")
test_df = pd.read_csv("data/active_learning/test.csv")
```

```
[2]: test_df
```

```
[2]:
```

	COMMENT_ID \	
0	z120djlhizeksdulo23mj5z52vjmxlhrk04	
1	z133ibkihkmaj3bfq22rilaxmp2yt54nb	
2	z12gxdortqzwhhqas04cfjrwituzghb5tvk0k	
3	_2viQ_Qnc6_ZYkMn1fs805Z6oy8Ime06pSjMLAlwYfM	
4	z120slagtmmtler404cifqbzxvd15idtw0k	
..	...	
387	z13pup2w2k3rz1lx104cf1a5qzavgvv51vg0k	
388	z13psdarpuzbjplhh04cjfwgzonextlhflw	
389	z131xnwierifxxkj204cgvjxyo3oydb42r40k	
390	z12pwxj0kfrwnxye04cxtqntyacd1yia44	
391	z13oxvzqrzvyit00322jwjtjo2tzqylhof04	

	AUTHOR	DATE \
0	Murlock Nightcrawler	2015-05-24T07:04:29.844000
1	Debra Favacho (Debra Sparkle)	2015-05-21T14:08:41.338000
2	Muhammad Asim Mansha	NaN
3	mile panika	2013-11-03T14:39:42.248000
4	Sheila Cenabre	2014-08-19T12:33:11
..
387	geraldine lopez	2015-05-20T23:44:25.920000
388	bilal bilo	2015-05-22T20:36:36.926000
389	YULIOR ZAMORA	2014-09-10T01:35:54
390		2015-05-15T19:46:53.719000
391	Octavia W	2015-05-22T02:33:26.041000

(continues on next page)

(continued from previous page)

	CONTENT	CLASS	VIDEO
0	Charlie from LOST?	0	3
1	BEST SONG EVER X3333333333	0	4
2	Aslamu Lykum... From Pakistan	1	3
3	I absolutely adore watching football plus I've...	1	4
4	I really love this video.. http://www.bubblews...	1	1
..
387	love the you lie the good	0	3
388	I liked 	0	4
389	I loved it so much ...	0	1
390	good party	0	2
391	Waka waka	0	4

[392 rows x 6 columns]

As we can see, the data contains the comment id, the author of the comment, the date, the content (the comment itself) and a class column that indicates if a comment is spam or ham. We will use the class column only in the test dataset to illustrate the effectiveness of the active learning approach with Rubrix. For the training dataset, we will simply ignore the column and assume that we are gathering training data from scratch.

2. Defining our classifier and Active Learner

In this tutorial, we will use a multinomial **Naive Bayes classifier** that is suitable for classification with discrete features (e.g., word counts for text classification):

```
[3]: from sklearn.naive_bayes import MultinomialNB

# Define our classification model
classifier = MultinomialNB()
```

Then, we will define our active learner, which uses the classifier as an estimator of the most uncertain predictions:

```
[4]: from modAL.models import ActiveLearner

# Define active learner
learner = ActiveLearner(
    estimator=classifier,
)
```

The features for our classifier will be the counts of different word **n-grams**: that is, for each example we count the number of contiguous sequences of n words, where n goes from 1 to 5.

The output of this operation will be matrices of n -gram counts for our train and test data set, where each element in a row equals the counts of a specific word n -gram found in the example:

```
[5]: from sklearn.feature_extraction.text import CountVectorizer

# The resulting matrices will have the shape of ('nr of examples', 'nr of word n-grams')
vectorizer = CountVectorizer(ngram_range=(1, 5))

X_train = vectorizer.fit_transform(train_df.CONTENT)
X_test = vectorizer.transform(test_df.CONTENT)
```

3. Active Learning loop

Now we can start our active learning loop that consists of iterating over following steps:

1. Annotate samples
2. Teach the active learner
3. Plot the improvement (optional)

Before starting the learning loop, let us define two variables:

- the number of instances we want to annotate per iteration,
- a variable to keep track of our improvements by recording the achieved accuracy after each iteration.

```
[6]: # Number of instances we want to annotate per iteration
n_instances = 10

# Accuracies after each iteration to keep track of our improvement
accuracies = []
```

Step 1: Annotate samples

The first step of the training loop is about annotating n examples having the most uncertain prediction. In the first iteration, these will be just random examples, since the classifier is still not trained and we do not have predictions yet.

```
[7]: from sklearn.exceptions import NotFittedError

# query examples from our training pool with the most uncertain prediction
query_idx, query_inst = learner.query(X_train, n_instance=n_instances)

# get predictions for the queried examples
try:
    probabilities = learner.predict_proba(X_train[query_idx])
# For the very first query we do not have any predictions
except NotFittedError:
    probabilities = [[0.5, 0.5]]*n_instances
```

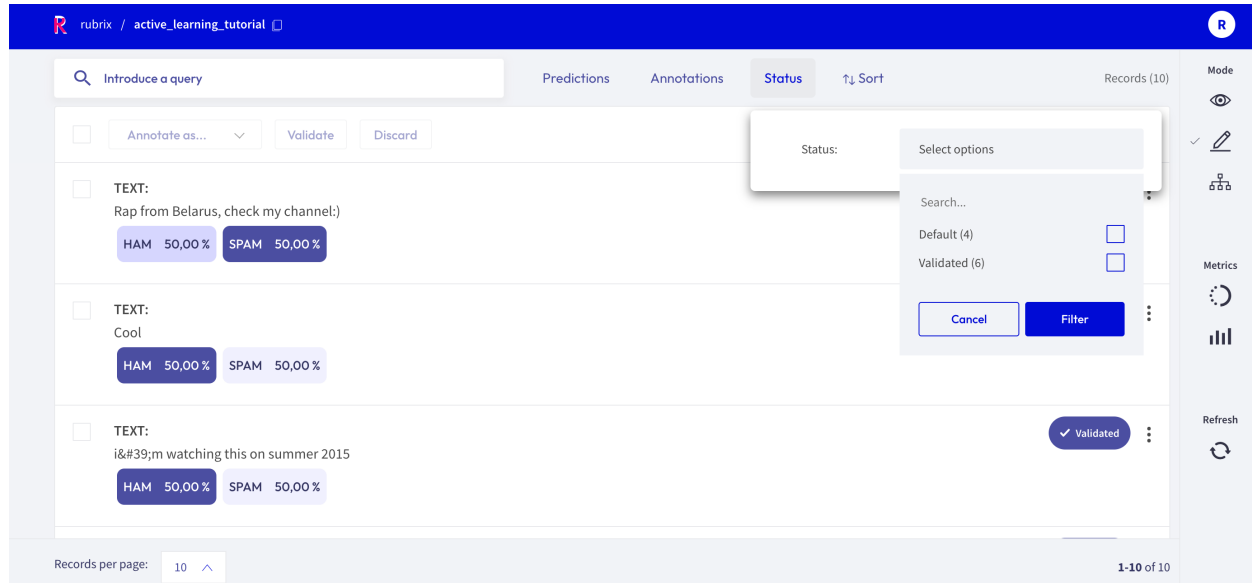
```
[ ]: import rubrix as rb

# Build the Rubrix records
records = [
    rb.TextClassificationRecord(
        id=idx,
        text=train_df.CONTENT.iloc[idx],
        prediction=list(zip(["HAM", "SPAM"], probs)),
        prediction_agent="MultinomialNB",
    )
    for idx, probs in zip(query_idx, probabilities)
]

# Log the records
rb.log(records, name="active_learning_tutorial")
```

After logging the records to Rubrix, we switch over to the UI, where we can find the newly logged examples in the `active_learning_tutorial` dataset.

To only show the examples that are still missing an annotation, you can select **“Default”** in the **Status** filter as shown in the screenshot below. After annotating a few examples you can press the **Refresh** button in the left side bar, to update the view with respect to the filters.



Once you are done annotating the examples, you can continue with the active learning loop. If your annotations only contained one class, consider increasing the `n_instances` parameter.

Step 2: Teach the learner

The second step in the loop is to teach the learner. Once we have trained our classifier with the newly annotated examples, we can apply the classifier to the test data and record the accuracy to keep track of our improvement.

```
[ ]: # Load the annotated records into a pandas DataFrame
records_df = rh.load("active_learning_tutorial", id=query_idx.tolist()).to_pandas()

# check if all examples were annotated
if any(records_df.announcement.isna()):
    raise UserWarning("Please annotate first all your samples before teaching the model")

# train the classifier with the newly annotated examples
y_train = records_df.announcement.map(lambda x: int(x == "SPAM"))
learner.train(X=X_train[query_idx], y=y_train.tolist())

# Keep track of our improvement
accuracies.append(learner.score(X=X_test, y=test_df.CLASS))
```

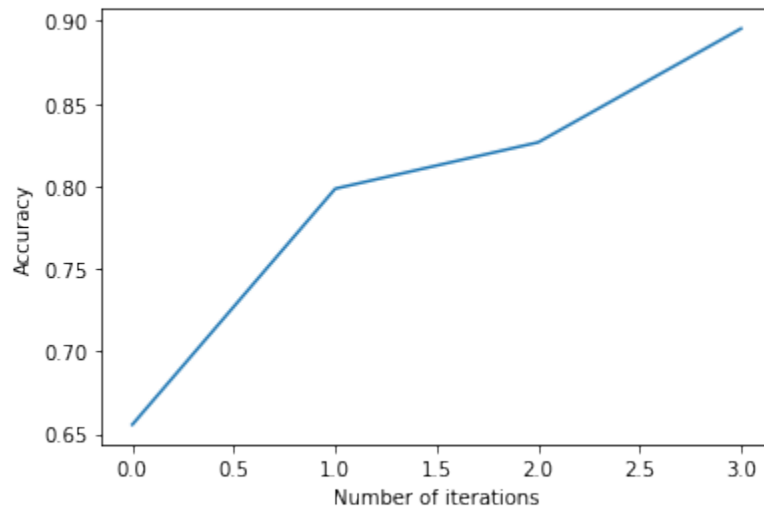
Now go back to step 1 and repeat both steps a couple of times.

Step 3. Plot the improvement (optional)

After a few iterations, we can check the current performance of our classifier by plotting the accuracies. If you think the performance can still be improved, you can **repeat step 1 and 2** and check the accuracy again.

```
[39]: import matplotlib.pyplot as plt

# Plot the accuracy versus the iteration number
plt.plot(accuracies)
plt.xlabel("Number of iterations")
plt.ylabel("Accuracy");
```



Summary

In this tutorial we saw how to embed *Rubrix* in an active learning loop, and also how it can help you to **gather a sample efficient data set by annotating only the most decisive examples**. We created a rather minimalist active learning loop, but *Rubrix* does not really care about the complexity of the loop. It will always help you to record and annotate data examples with their model predictions, allowing you to **quickly build up a data set from scratch**.

Next steps

Rubrix Github repo to stay updated.

Rubrix documentation for more guides and tutorials.

Join the Rubrix community! A good place to start is the discussion forum.

Bonus: Compare query strategies, random vs max uncertainty

In this bonus, we quickly demonstrate the effectiveness of annotating only the most uncertain predictions compared to random annotations. So the next time you want to build a data set from scratch, keep this strategy in mind and maybe use *Rubrix* for the annotation process!

```
[ ]: import numpy as np

n_iterations = 150
n_instances = 10
random_samples = 50

# max uncertainty strategy
accuracies_max = []
for i in range(random_samples):
    train_rnd_df = train_df.sample(frac=1)
    test_rnd_df = test_df.sample(frac=1)
    X_rnd_train = vectorizer.transform(train_rnd_df.CONTENT)
    X_rnd_test = vectorizer.transform(test_rnd_df.CONTENT)

    accuracies, learner = [], ActiveLearner(estimator=MultinomialNB())

    for i in range(n_iterations):
        query_idx, _ = learner.query(X_rnd_train, n_instances=n_instances)
        learner.teach(X=X_rnd_train[query_idx], y=train_rnd_df.CLASS.iloc[query_idx].to_
↳ list())
        accuracies.append(learner.score(X=X_rnd_test, y=test_rnd_df.CLASS))
    accuracies_max.append(accuracies)

# random strategy
accuracies_rnd = []
for i in range(random_samples):
    accuracies, learner = [], ActiveLearner(estimator=MultinomialNB())

    for random_idx in np.random.choice(X_train.shape[0], size=(n_iterations, n_
↳ instances), replace=False):
        learner.teach(X=X_train[random_idx], y=train_df.CLASS.iloc[random_idx].to_list())
        accuracies.append(learner.score(X=X_test, y=test_df.CLASS))
    accuracies_rnd.append(accuracies)

arr_max, arr_rnd = np.array(accuracies_max), np.array(accuracies_rnd)

[ ]: plt.plot(range(n_iterations), arr_max.mean(0))
plt.fill_between(range(n_iterations), arr_max.mean(0)-arr_max.std(0), arr_max.
↳ mean(0)+arr_max.std(0), alpha=0.2)
```

(continues on next page)

(continued from previous page)

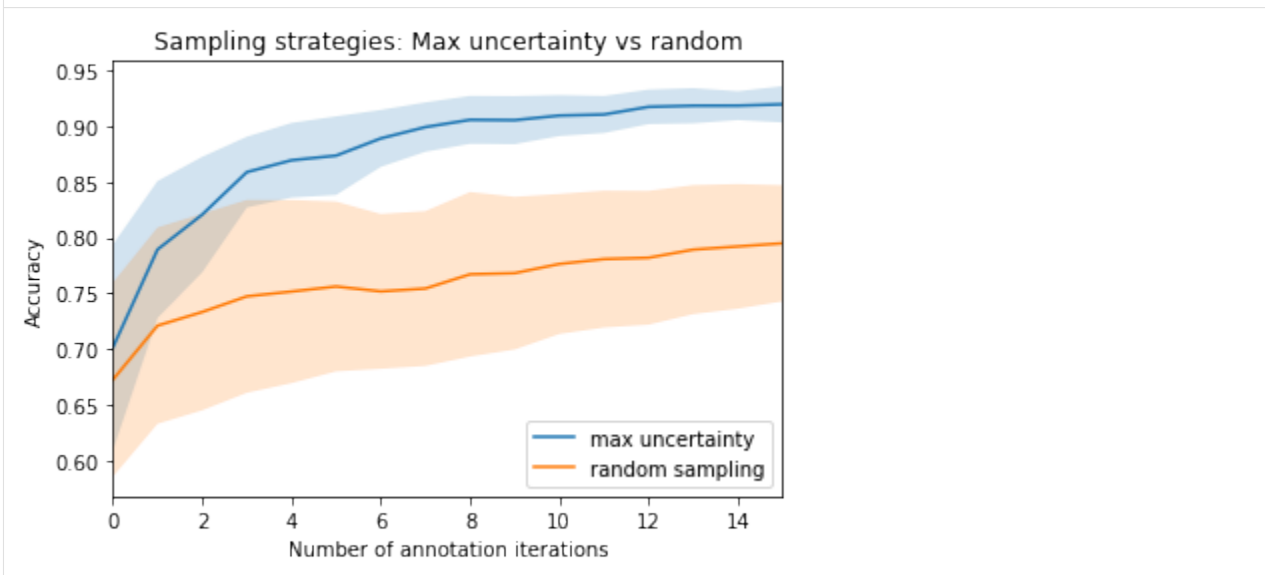
```

plt.plot(range(n_illustrations), acc_rnd.mean(0))
plt.fill_between(range(n_illustrations), acc_rnd.mean(0)-acc_rnd.std(0), acc_rnd.
    ↳ mean(0)+acc_rnd.std(0), alpha=0.2)

plt.xlim(0,15)
plt.title("Sampling strategies: Max uncertainty vs random")
plt.xlabel("Number of annotation iterations")
plt.ylabel("Accuracy")
plt.legend(["max uncertainty", "random sampling"], loc=4)

```

<matplotlib.legend.Legend at 0x7fa38aaaab20>



Appendix: How did we obtain the train/test data?

```

[ ]: import pandas as pd
    from urllib import request
    from sklearn.model_selection import train_test_split
    from pathlib import Path
    from tempfile import TemporaryDirectory

    def load_data() -> pd.DataFrame:
        """
        Downloads the [YouTube Spam Collection](http://www.dt.fee.unicamp.br/~tiago//
        ↳ youtubespamcollection/)
        and returns the data as a tuple with a train and test DataFrame.
        """
        links, data_df = [
            "http://lasid.sor.ufscar.br/labeling/datasets/9/download/",
            "http://lasid.sor.ufscar.br/labeling/datasets/10/download/",
            "http://lasid.sor.ufscar.br/labeling/datasets/11/download/",
            "http://lasid.sor.ufscar.br/labeling/datasets/12/download/",
            "http://lasid.sor.ufscar.br/labeling/datasets/13/download/",

```

(continues on next page)

(continued from previous page)

```

], None

with TemporaryDirectory() as tmpdirname:
    dfs = []
    for i, link in enumerate(links):
        file = Path(tmpdirname) / f"{i}.csv"
        request.urlretrieve(link, file)
        df = pd.read_csv(file)
        df["VIDEO"] = i
        dfs.append(df)
    data_df = pd.concat(dfs).reset_index(drop=True)

    train_df, test_df = train_test_split(data_df, test_size=0.2, random_state=42)

    return train_df, test_df

train_df, test_df = load_data()
train_df.to_csv("data/active_learning/train.csv", index=False)
test_df.to_csv("data/active_learning/test.csv", index=False)

```

4.17.2 Active learning for text classification with small-text

In this tutorial, you will learn how to set up a complete active learning loop with a [Hugging Face transformer](#):

- Use the excellent [small-text](#) library to set up your active learner;
- Use a *Rubrix listener* to build and start an active learning loop;
- Use the *Rubrix UI* to annotate examples and learn actively;

The screenshot displays the Rubrix UI interface for active learning. The top navigation bar shows the path 'Datasets / rubrix / trec_with_active_learning'. The main area contains a list of text examples for annotation. Each example has a checkbox, a text input, and a set of classification buttons (NUM, LOC, HUM, ENTY, DESC, ABBR). The first example is 'What is SVHS?', the second is 'What was the name of the horse that fell on Queen Elizabeth, Prince Albert's wife?', and the third is 'What major league baseball team compiled the best won-lost record between 1957 and 1983?'. The right sidebar shows the 'Progress' section with a total of 35/40 records, 87.50% validated, and a table of counts for each classification category.

Category	Count
Validated	35
Discarded	0
DESC	11
ENTY	9
LOC	6
HUM	4
NUM	3
ABBR	2

Introduction

Active learning is a special case of machine learning in which a learning algorithm can interactively query a user (or some other information source) to label new data points with the desired outputs. [Wikipedia](#)

Supervised machine learning often requires large amounts of labeled data that are expensive to generate. *Active Learning* (AL) systems attempt to overcome this labeling bottleneck. The underlying idea is that not all data points are equally important for training the model. The AL system tries to query only the most relevant data from a pool of unlabeled data to be labeled by a so-called *oracle*, which is often a human annotator. Therefore, AL systems are usually much more sample efficient and need far less training data than traditional supervised systems.

This tutorial will show you how to incorporate [Rubrix](#) into an active learning workflow involving a human in the loop. We will build a simple text classifier by combining the active learning framework [small-text](#) and Rubrix. Hugging Face's [transformers](#) will provide the classifier we will embed in an active learner from small-text. Rubrix excels in making **you** the oracle that conveniently teaches the model via an intuitive UI.

Setup

Rubrix is a free and open-source tool to explore, annotate, and monitor data for NLP projects.

If you are new to Rubrix, check out the [Github repository](#).

If you have not installed and launched Rubrix yet, check the [Setup and Installation Guide](#).

For this tutorial, we also need some optional and third-party libraries that you can install via pip:

```
[ ]: %pip install "rubrix[listeners]" "datasets~=2.5.0" "small-text>=1.1.0"
      ↪ "transformers[torch]"
```

The TREC dataset

For this tutorial, we will use the well-known [TREC dataset](#) containing 6000 labeled questions; 5500 in the training and 500 in the test split. This dataset can be turned into a text classification task, in which a model must predict one of the six coarse labels given the question. The labels consist of ABBREVIATION (ABBR), ENTITY (ENTY), DESCRIPTION (DESC), HUMAN (HUM), LOCATION (LOC), and NUMERIC VALUE (NUM).

Let us load the dataset from the [Hugging Face Hub](#):

```
[ ]: import datasets

trec = datasets.load_dataset('trec', version=datasets.Version("2.0.0"))
```

Preprocessing the dataset

We first need to wrap the dataset in a specific data class provided by [small-text](#), the excellent active learning framework we will use in this tutorial. Since we will choose a [Hugging Face transformer](#) in the active learner, small-text will expect a `TransformersDataset` object that already contains the tokenized input text.

In order to build a `TransformersDataset` object, we first need a tokenizer:

```
[ ]: import torch
      from transformers import AutoTokenizer
```

(continues on next page)

(continued from previous page)

```
# Choose transformer model: In non-gpu environments we use a tiny model to reduce the
↪runtime
if not torch.cuda.is_available():
    transformer_model = "prajjwal1/bert-tiny"
else:
    transformer_model = "bert-base-uncased"

# Init tokenizer
tokenizer = AutoTokenizer.from_pretrained(transformer_model)
```

With this, we can create a `TransformersDataset` by calling `TransformersDataset.from_arrays()` which expects a list of texts, a numpy array (which indicates single-label classification), a tokenizer, and lastly the possible (integer values) of target labels within this dataset.

```
[ ]: import numpy as np
from small_text import TransformersDataset

num_classes = tree["train"].features["coarse_label"].num_classes
target_labels = np.arange(num_classes)

train_text = [row["text"] for row in tree["train"]]
train_labels = np.array([row["coarse_label"] for row in tree["train"]])

# Create the dataset for small-text
dataset = TransformersDataset.from_arrays(train_text, train_labels, tokenizer, target_
↪labels=target_labels)
```

We will also create a test dataset to track the performance of the transformer model during the active learning loop.

```
[ ]: # Create test dataset
test_text = [row["text"] for row in tree["test"]]
test_labels = np.array([row["coarse_label"] for row in tree["test"]])

dataset_test = TransformersDataset.from_arrays(test_text, test_labels, tokenizer, target_
↪labels=np.arange(num_classes))
```

Setting up the active learner

Now that we have our datasets ready let's set up the active learner. For this, we need two components:

- the **classifier** to be trained;
- the **query strategy** to obtain the most relevant data;

In our case, we choose a [Hugging Face transformer](#) as the classifier and a [tie-breaker](#) as the query strategy. The latter selects instances of the data pool with a small margin between the two most likely predicted labels.

```
[ ]: from small_text import (
    BreakingLies,
    PoolBasedActiveLearner,
    TransformerBasedClassificationFactory,
    TransformerModelArguments
)
```

(continues on next page)

(continued from previous page)

```
# Define our classifier
cli_factory = TransformerBasedClassificationFactory(
    TransformerModelArguments(transformer_model),
    num_classes=6,
    # If you have a cuda device, specify it here.
    # Otherwise, just remove the following line.
    # kwargs={"device": "cuda"}
)

# Define our query strategy
query_strategy = BreakingLine()

# Use the active learner with a pool containing all unlabeled data
active_learner = PoolBasedActiveLearner(cli_factory, query_strategy, dataset)
```

Since most query strategies, including ours, require a trained model, we randomly draw a subset from the data pool to initialize our AL system. After obtaining the labels for this batch of instances, the active learner will use them to create the first classifier.

```
[ ]: from small_text import random_initialization
import numpy as np
# Fix seed for reproducibility
np.random.seed(42)

# Number of samples in our queried batches
NUM_SAMPLES = 20

# Randomly draw an initial subset from the data pool
initial_indices = random_initialization(dataset, NUM_SAMPLES)
```

Rubrix and you: the perfect oracle

With our active learner ready, it is time to teach it. We first create a *Rubrix dataset* to log and label the initial random batch queried by the active learner.

```
[ ]: import rubrix as rb

# Choose a name for the dataset
DATASET_NAME = "trec_with_active_learning"

# Define labeling schema
labels = trec["train"].features["coarse_label"].names
settings = rb.TextClassificationSettings(label_schema=labels)

# Create dataset with a label schema
rb.configure_dataset(name=DATASET_NAME, setting=settings)

# Create records from the initial batch
records = [
```

(continues on next page)

(continued from previous page)

```

    rb.TextClassificationRecord(
        text=rec["train"]["text"][idx],
        metadata={"batch_id": 0},
        id=idx,
    )
    for idx in initial_indices
]

# Log initial records to Rubrix
rb.log(records, DATASET_NAME)

```

Before switching to the Rubrix UI to label the records, we will set up the **active learning loop**. For this, we will use the *listener decorator* from Rubrix. The loop will run automatically once all records of a batch are labeled (see the *query* and *condition* argument of the decorator). It will trigger the classifier's training, query a new batch from the active learner and log it to Rubrix. We will also keep track of the accuracy of the current classifier by evaluating it on our test set.

```

[ ]: from rubrix.listeners import listener
    from sklearn.metrics import accuracy_score

# Define some helper variables
LABEL2INT = {rec["train"].features["coarse_label"].str2int}
ACCURACIES = []

# Set up the active learning loop with the listener decorator
@listener(
    dataset=DATASET_NAME,
    query="status:Validated AND metadata.batch_id:{batch_id}",
    condition=lambda search: search.total==NUM_SAMPLES,
    execution_interval_in_second=3,
    batch_id=0
)
def active_learning_loop(records, ctx):

    # 1. Update active learner
    print(f"Updating with batch_id {ctx.query_params['batch_id']} ...")
    y = np.array([LABEL2INT(rec.annotation) for rec in records])

    # initial update
    if ctx.query_params["batch_id"] == 0:
        indices = np.array([rec.id for rec in records])
        active_learner.initialize_data(indices, y)
    # update with the prior queried indices
    else:
        active_learner.update(y)
    print("Done!")

    # 2. Query active learner
    print("Querying new data points ...")
    queried_indices = active_learner.query(num_samples=NUM_SAMPLES)
    new_batch = ctx.query_params["batch_id"] + 1
    new_records = [

```

(continues on next page)

(continued from previous page)

```

    rh.TextClassificationRecord(
        text=trec["train"]["text"][idx],
        metadata={"batch_id": new_batch},
        id=idx,
    )
    for idx in queried_indices
]

# 3. Log the batch to Rubrix
rh.log(new_records, DATASET_NAME)

# 4. Evaluate current classifier on the test set
print("Evaluating current classifier ...")
accuracy = accuracy_score(
    dataset_test.y,
    active_learner.classifier.predict(dataset_test),
)

ACCURACIES.append(accuracy)
ctx.set_params["batch_id"] = new_batch
print("Done!")

print("Waiting for annotations ...")

```

Starting the active learning loop

Now we can start the loop and switch to the Rubrix UI.

```
[ ]: active_learning_loop.start()
```

In the Rubrix UI, we will set the number of records per page to 20 since it is also our chosen batch size. Furthermore, we will use the *Status filter* to filter out already annotated records. Now, all we have to do is to annotate the displayed records. Once annotating everything, the classifier's training will be automatically triggered.

After a few seconds, you should see the newly queried batch when pressing the *Refresh button*. The training can take longer depending on your machine and whether you have a CUDA device. You can always check the status of the active learning loop from your notebook.

Can we stop?

After a few iterations, we can check the accuracy of the current classifier and plot its evaluation.

```
[ ]: import pandas as pd

pd.Series(ACCURACIES).plot(xlabel="Iteration", ylabel="Accuracy");

```

We should achieve an accuracy of at least **0.8 after around 12 iterations**, corresponding to roughly 260 annotated records. The stopping criterion is ultimately up to you, and you can choose more sophisticated criteria like the `Kap-paAverage` implemented in `small-text`.

```
[ ]: active_learning_loop.stop()
```

Summary

In this tutorial, we saw how you could **embed Rubrix in an active learning loop involving a human in the loop**. We relied on **small-text to use a Hugging Face transformer within an active learning setup**. In the end, we gathered a **sample-efficient data set by annotating only the most informative records** for the model.

Rubrix makes it very easy to use a dedicated annotation team or subject matter experts as an oracle for your active learning system. They will only interact with the Rubrix UI and do not have to worry about training or querying the system. We encourage you to try out active learning in your next project and make your and your annotator's life a little easier.

Next steps

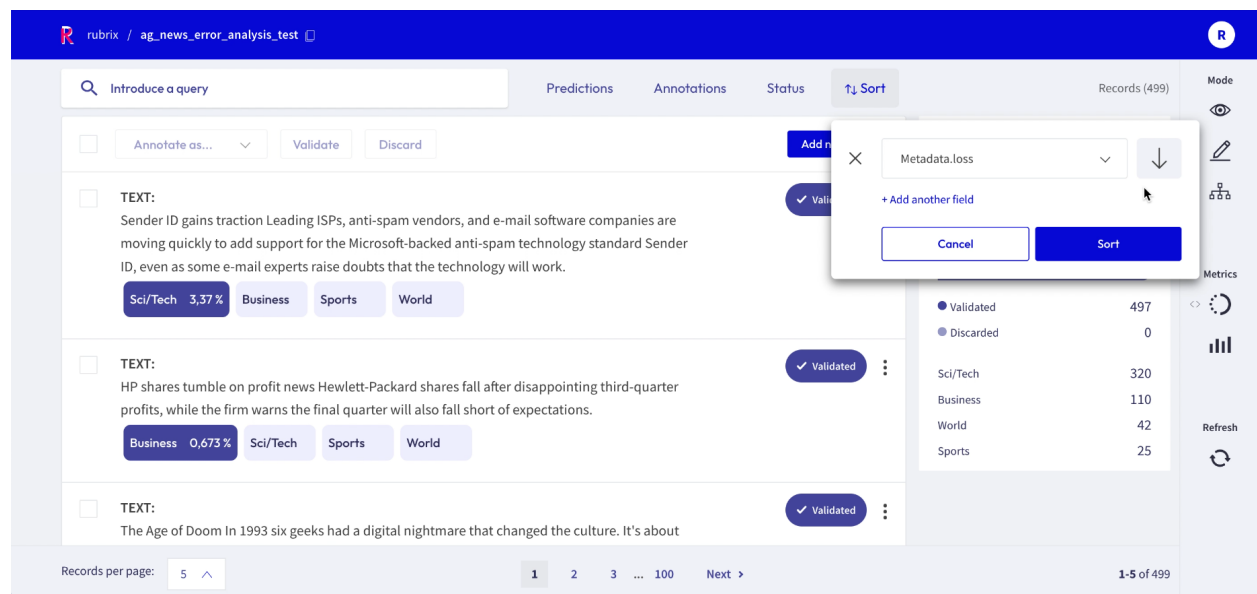
Rubrix [Github repo](#) to stay updated.

Rubrix [documentation](#) for more guides and tutorials.

Join the Rubrix community! A good place to start is our [slack channel](#).

4.18 Label errors

These tutorials show you how to find label errors in your data using Rubrix.



The screenshot shows the Rubrix interface for a dataset named 'ag_news_error_analysis_test'. The main view displays a list of records with their text, predicted labels, and validation status. A modal is open, allowing the user to sort the records by 'Metadata.loss'. The modal includes a search bar, a dropdown menu for the field to sort by, and buttons for 'Cancel' and 'Sort'.

Field	Count
Validated	497
Discarded	0
Sci/Tech	320
Business	110
World	42
Sports	25

Clean labels using your model loss Learn a simple technique for error analysis, using model loss to find potential training data errors.

The screenshot shows the Rubrix web interface for managing a dataset. At the top, there's a search bar and navigation tabs: Predictions, Annotations, Status, Metadata, and Sort. A modal is open for sorting, showing a dropdown menu with 'Metadata.label_error_candidate' selected. Below the modal, there's a list of text samples. Each sample has a checkbox, a text description, and a set of labels with their respective percentages. For example, the first sample is labeled 'Sci/Tech' with 99.988%, 'Business' with 0.012%, 'Sports' with 0.00%, and 'World' with 0.00%. The second sample is labeled 'Business' with 100.00%, 'Sci/Tech' with 0.00%, 'World' with 0.00%, and 'Sports' with 0.00%. The third sample is labeled 'Business' with 100.00%, 'Sci/Tech' with 0.00%, 'World' with 0.00%, and 'Sports' with 0.00%. At the bottom, there's a pagination bar showing 'Records per page: 5' and '1 2 3 ... 370 Next >'.

Find label errors with cleanlab Leverage Rubrix and cleanlab to find, uncover and correct potential label errors in the AGNews dataset.

4.18.1 Clean labels using your model loss

In this tutorial, we will learn to introduce a simple technique for error analysis, **using model loss to find potential training data errors**.

- This technique is shown using a **fine-tuned text classifier from the Hugging Face Hub** on the **AG News dataset**.
- Using Rubrix, we will verify **more than 50 mislabelled examples on the training set** of this well-known NLP benchmark.
- This trick is useful for **model training with small and noisy datasets**.
- This trick is complementary with other “data-centric” ML methods such as **cleanlab** (see this [Rubrix tutorial](#)).

Introduction

This tutorial explains a simple trick you can leverage with Rubrix for finding potential errors in training data: *using your model loss to identify label errors or ambiguous examples*. This trick is not new (those who’ve worked with fastai know how useful the `plot_top_losses` method is). Even Andrej Karpathy [tweeted](#) about this some time ago:

When you sort your dataset descending by loss you are guaranteed to find something unexpected, strange and helpful.

— Andrej Karpathy (@karpathy) October 2, 2020

The technique is really simple: if you are training a model with a training set, train your model, and you apply your model to the training set to **compute the loss for each example in the training set**. If you sort your dataset examples by loss, examples with the highest loss are the most ambiguous and difficult to learn.

This technique can be used for **error analysis during model development** (e.g., identifying tokenization problems), but it turns out is also a really simple technique for **cleaning up your training data, during model development or after training data collection activities**.

In this tutorial, we'll use this technique with a well-known text classification benchmark, the [AG News dataset](#). After computing the losses, we'll use Rubrix to analyse the highest loss examples. In less than 5 minutes, we manually check and relabel the first 50 examples. In fact, the first 50 examples with the highest loss, are all incorrect in the original training set. If we visually inspect further examples, we still find label errors in the top 500 examples.

Why it's important

1. **Machine learning models are only as good as the data they're trained on.** Almost all training data source can be considered “noisy” (e.g., crowd-workers, annotator errors, weak supervision sources, data augmentation, etc.)
2. With this simple technique **we're able to find more than 50 label errors on a widely-used benchmark in less than 5 minutes** (your dataset will probably be noisier!).
3. With advanced model architectures widely-available, **managing, cleaning, and curating data is becoming a key step for making robust ML applications.** A good summary of the current situation can be found in the website of the [Data-centric AI NeurIPS Workshop](#).
4. This simple trick **can be used across the whole ML lifecycle** and not only for finding label errors. With this trick you can improve data preprocessing, tokenization, and even your model architecture.

Setup

Rubrix, is a free and open-source tool to explore, annotate, and monitor data for NLP projects.

If you are new to Rubrix, check out the [Github repository](#).

If you have not installed and launched Rubrix yet, check the [Setup and Installation guide](#).

For this tutorial we will also need the third party libraries [transformers](#) and [datasets](#), as well as [PyTorch](#), which can be installed via pip:

```
[ ]: %pip install transformers datasets torch -qqq
```

Preliminaries

1. A model fine-tuned with the AG News dataset (you could train your own model if you wish).
2. The AG News train split (the same trick could and should be applied to validation and test splits).
3. Rubrix for logging, exploring, and relabeling wrong examples (we provide a pre-computed datasets so feel free to skip to this step)

1. Load the fine-tuned model and the training dataset

Now, we will load the [AG News dataset](#). But first, we need to define and set the device, the model and the tokenizer:

```
[ ]: import torch

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```



```
[ ]: from transformers import AutoTokenizer, AutoModelForSequenceClassification

# load model and tokenizer
tokenizer = AutoTokenizer.from_pretrained("andi611/distilbert-base-uncased-ner-agnews")
model = AutoModelForSequenceClassification.from_pretrained("andi611/distilbert-base-
↳uncased-ner-agnews")

[ ]: from datasets import load_dataset

# load the training split
ds = load_dataset('ag_news', split='train')

[ ]: # tokenize and encode the training set
def tokenize_and_encode(batch):
    return tokenizer(batch['text'], truncation=True)

ds_enc = ds.map(tokenize_and_encode, batched=True)
```

2. Computing the loss

The following code will compute the loss for each example using our trained model. This process is taken from the very well-explained blog post by Lewis Tunstall: “Using data collators for training and error analysis”, where he explains this process for error analysis during model training.

In our case, we instantiate a data collator directly, while he uses the Data Collator from the Trainer directly:

```
[ ]: from transformers.data.data_collator import DataCollatorWithPadding

# create the data collator for inference
data_collator = DataCollatorWithPadding(tokenizer, padding=True)

[ ]: # function to compute the loss example-wise
def loss_per_example(batch):
    batch = data_collator(batch)
    input_ids = torch.tensor(batch["input_ids"], device=device)
    attention_mask = torch.tensor(batch["attention_mask"], device=device)
    labels = torch.tensor(batch["labels"], device=device)

    with torch.no_grad():
        output = model(input_ids, attention_mask)
        batch["predicted_label"] = torch.argmax(output.logits, axis=1)
        # compute the probabilities for logging them into Rubrix
        batch["predicted_probas"] = torch.nn.functional.softmax(output.logits, dim=0)

    # don't reduce the loss (return the loss for each example)
    loss = torch.nn.functional.cross_entropy(output.logits, labels, reduction="none")
    batch["loss"] = loss

    # datasets complains with numpy dtypes, let's use Python lists
    for k, v in batch.items():
        batch[k] = v.cpu().numpy().tolist()
```

(continues on next page)

(continued from previous page)

```
return batch
```

Now, it is time to turn the dataset into a Pandas dataframe and sort this dataset by descending loss:

```
[ ]: import pandas as pd

losses_ds = ds_enc.remove_column("text").map(loss_per_example, batched=True, batch_
↳ size=32)

# turn the dataset into a Pandas dataframe, sort by descending loss and visualize the_
↳ top examples.
pd.set_option("display.max_colwidth", None)

losses_ds.set_format('pandas')
losses_df = losses_ds[:][['label', 'predicted_label', 'loss', 'predicted_probas']]

# add the text column removed by the trainer
losses_df['text'] = ds_enc['text']
losses_df.sort_values("loss", ascending=False).head(10)
```

	label	...	text
44984	1	...	Baghdad blasts kills at least 16 Insurgents have detonated two_ bombs near a convoy of US military vehicles in southern Baghdad, killing at least 16_ people, Iraqi police say.
101562	1	...	Immoral, unjust, oppressive_ dictatorship. . . and then there #39;s ... ROBERT MUGABES_ Government is pushing through legislation designed to prevent human rights_ organisations from operating in Zimbabwe.
31564	1	...	Ford to Cut 1,150 Jobs At British Jaguar Unit Ford Motor Co._ announced Friday that it would eliminate 1,150 jobs in England to streamline its_ Jaguar Cars Ltd. unit, where weak sales have failed to offset spending on new products_ and other parts of the business.
41247	1	...	Palestinian gunmen kidnap_ CNN producer GAZA CITY, Gaza Strip -- Palestinian gunmen abducted a CNN producer in_ Gaza City on Monday, the network said. The network said Riyadh Ali was taken away at_ gunpoint from a CNN van.
44961	1	...	Bomb Blasts in Baghdad Kill at Least 35, Wound 120_ Insurgents detonated three car bombs near a US military convoy in southern Baghdad on_ Thursday, killing at least 35 people and wounding around 120, many of them children,_ officials and doctors said.
75216	1	...	Marine Wives_ Rally A group of Marine wives are running for the family of a Marine Corps officer who_ was killed in Iraq.
31229	1	...	Auto Stocks Fall Despite Ford_ Outlook Despite a strong profit outlook from Ford Motor Co., shares of automotive_ stocks moved mostly lower Friday on concerns sales for the industry might not be as_ strong as previously expected.

(continues on next page)

(continued from previous page)

```

19737      3 ...
→ Mladin Release From Road Atlanta Australia #39;s Mat Mladin completed a winning_
→double at the penultimate round of this year #39;s American AMA Chevrolet Superbike_
→Championship after taking
60726      2 ...                               Suicide Bombings_
→Kill 10 in Green Zone Insurgents hand-carried explosives into the most fortified_
→section of Baghdad Thursday and detonated them within seconds of each other, killing_
→10 people and wounding 20.
28307      3 ... Lightning Strike Injures 40 on Texas Field (AP) AP - About 40_
→players and coaches with the Grapeland High School football team in East Texas were_
→injured, two of them critically, when lightning struck near their practice field_
→Tuesday evening, authorities said.

[10 rows x 5 columns]
```

```

[2]: # save this to a file for further analysis
#losses_df.to_json("agnews_train_loss.json", orient="records", lines=True)
```

While using Pandas and Jupyter notebooks is useful for initial inspection, and programmatic analysis. If you want to quickly explore the examples, relabel them, and share them with other project members, Rubrix provides you with a straight-forward way for doing this. Let's see how.

3. Log high loss examples into Rubrix

Using the amazing Hugging Face Hub we've shared the resulting dataset, which you can find [here](#) and load directly using the datasets library

Now, we log the first 500 examples into a Rubrix dataset:

```

[ ]: # if you have skipped the first two steps you can load the dataset here:
import pandas as pd
from datasets import load_dataset

dataset = load_dataset("dvilasuelro/ag_news_training_set_losses", split='train')
losses_df = dataset.to_pandas()

ds = load_dataset('ag_news', split='test') # only for getting the label names
```

```

[7]: import rubrix as rb
# creates a Text classification record for logging into Rubrix
def make_record(row):

    return rb.TextClassificationRecord(
        text=row.text,
        # this is the "gold" label in the original dataset
        annotation=[(ds.features['label'].names[row.label])],
        # this is the prediction together with its probability
        prediction=[(ds.features['label'].names[row.predicted_label], row.predicted_
→ proba[row.predicted_label])],
        # metadata fields can be used for sorting and filtering, here we log the loss
        metadata={"loss": row.loss},
```

(continues on next page)

(continued from previous page)

```
# who makes the prediction
prediction_agent="andi611/distilbert-base-uncased-ner-agnews",
# source of the gold label
annotation_agent="ag_news_benchmark"

)
```

```
[8]: # if you want to log the full dataset remove the indexing
top_losses = losses_df.sort_values("loss", ascending=False)[0:499]

# build Rubrix records
records = top_losses.apply(make_record, axis=1)
```

```
[ ]: rb.log(records, name="ag_news_error_analysis")
```

4. Using Rubrix Webapp for inspection and relabeling

In this step, we have a Rubrix Dataset available for exploration and annotation. A useful feature for this use case is **Sorting**. With Rubrix you can sort your examples by combining different fields, both from the standard fields (such as score) and custom fields (via the metadata fields). In this case, we've logged the loss so we can order our training examples by loss in descending order (showing higher loss examples first).

For preparing this tutorial, we have manually checked and relabelled the first 50 examples. Moreover, we've shared this re-annotated dataset in the Hugging Face Hub. In the next section, we show you how easy is to share Rubrix Datasets in the Hub.

5. Sharing the dataset in the Hugging Face Hub

Let's first load the re-annotated examples. Re-labelled examples are marked as `annotated_by` the user `rubrix`, which is the default user when launching Rubrix with Docker. We can retrieve only these records using the `query` param as follows:

```
[11]: import rubrix as rb
dataset = rb.load("ag_news_error_analysis", query="annotated_by:rubrix").to_pandas()

# let's do some transformations before uploading the dataset
dataset['loss'] = dataset.metadata.transform(lambda x: x['loss'])
dataset = dataset.rename(columns={"annotation": "corrected_label"})

dataset.head()
```

```
[11]:              inputs \
0  {'text': 'Top nuclear official briefs Majlis c...
1  {'text': 'Fischer Delivers Strong Message in S...
2  {'text': 'The Politics of Time and Dispossessi...
3  {'text': 'Hadash Party joins prisoners #39; st...
4  {'text': 'China May Join $10Bln Sakhalin-2 Ru...

              prediction corrected_label \
0  [(World, 0.1832696944)]              World
1  [(World, 0.0695228428)]              World
2  [(Sci/Tech, 0.100481838)]            Sci/Tech
```

(continues on next page)

(continued from previous page)

```

3      [(World, 0.1749624908)]      World
4      [(Business, 0.1370282918)]    Business

      prediction_agent annotation_agent multi_label \
0 andi611/distilbert-base-uncased-ner-agnews      rubrix      False
1 andi611/distilbert-base-uncased-ner-agnews      rubrix      False
2 andi611/distilbert-base-uncased-ner-agnews      rubrix      False
3 andi611/distilbert-base-uncased-ner-agnews      rubrix      False
4 andi611/distilbert-base-uncased-ner-agnews      rubrix      False

      explanation      id      metadata \
0      None      071a1014-71e7-41f4-83e4-553ba47610cf      {'loss': 7.6656146049}
1      None      07c8c4f6-3288-46f4-a618-3da4a537e605      {'loss': 7.9892320633}
2      None      0965a0d1-4886-432a-826a-58e99dfd9972      {'loss': 7.133708477}
3      None      09fc7065-a2c8-4041-adf8-34e029a7fde0      {'loss': 7.339015007}
4      None      1ef97c49-2f0f-43be-9b28-80a291cb3b1d      {'loss': 7.321100235}

      status event_timestamp metrics \
0 Validated      None      {}
1 Validated      None      {}
2 Validated      None      {}
3 Validated      None      {}
4 Validated      None      {}

      text      loss
0 Top nuclear official briefs Majlis committee T... 7.665615
1 Fischer Delivers Strong Message in Syria Germa... 7.989232
2 The Politics of Time and Dispossession Make a ... 7.133708
3 Hadash Party joins prisoners #39; strike for 2... 7.339015
4 China May Join \ $10Bln Sakhalin-2 Russia said ... 7.321100

```

```

[12]: # let's add the original dataset labels to share them together with the corrected ones
      # we sort by ascending loss our corrected dataset
      dataset = dataset.sort_values("loss", ascending=False)

      # we add original labels in string form
      id_label = list(dataset.corrected_label.unique())
      original_labels = [id_label[i] for i in top_losses[0:50].label.values]
      dataset["original_label"] = original_labels

```

Now let's transform this into a Dataset and define the features schema:

```

[13]: from datasets import Dataset, Features, Value, ClassLabel

      ds = dataset[['text', 'corrected_label', 'original_label']].to_dict(orient='list')

      hf_ds = Dataset.from_dict(
          ds,
          features=Features({
              "text": Value("string"),
              "corrected_label": ClassLabel(names=list(dataset.corrected_label.unique())),
              "original_label": ClassLabel(names=list(dataset.corrected_label.unique()))
          })
      )

```

(continues on next page)

(continued from previous page)

```
}
)
```

```
[19]: hf_ds.features
```

```
[19]: {'text': Value(dtype='string', id=None),
      'corrected_label': ClassLabel(num_classes=4, names=['World', 'Business', 'Sports', 'Sci/
      ↪Tech'], names_file=None, id=None),
      'original_label': ClassLabel(num_classes=4, names=['World', 'Business', 'Sports', 'Sci/
      ↪Tech'], names_file=None, id=None)}
```

Uploading the dataset with the `push_to_hub` method is as easy as:

```
[ ]: hf_ds.push_to_hub("Recognai/ag_news_corrected_labels")
```

Now the dataset is publicly [available](#) at the Hub!

Summary

In this tutorial we saw how you can leverage the **model loss** to find potential label errors in your training data set. The *Rubrix* web app makes it very convenient to sort your data by loss, inspect single records by eye, and allows you to easily correct label errors on the fly.

Next steps

If you are interested in the topic of training data curation and denoising datasets, check out the tutorial for using *Rubrix* with *cleanlab*.

Rubrix [Github repo](#) to stay updated.

[Rubrix documentation](#) for more guides and tutorials.

Join the Rubrix community! A good place to start is the [discussion forum](#).

```
[ ]:
```

4.18.2 Find label errors with cleanlab

In this tutorial we will leverage *Rubrix* and [cleanlab](#) to find, uncover and correct potential label errors. You can do this following 4 basic steps:

- load a dataset with potential label errors, here we use the [ag_news](#) dataset;
- train a model to make predictions for a test set, here we use a lightweight [sklearn](#) model;
- use *cleanlab* via *Rubrix* and get potential label error candidates in the test set;
- uncover and correct label errors quickly and comfortably with the *Rubrix* web app;

Introduction

As shown recently by [Curtis G. Northcutt et al.](#) label errors are pervasive even in the most-cited test sets used to benchmark the progress of the field of machine learning. They introduce a new principled framework to “identify label errors, characterize label noise, and learn with noisy labels” called **confident learning**. It is open-sourced as the [cleanlab Python package](#) that supports finding, quantifying, and learning with label errors in data sets.

Rubrix provides built-in support for *cleanlab* and makes it a breeze to find potential label errors in your dataset. In this tutorial we will try to uncover and correct label errors in the well-known [ag_news](#) dataset that is often used to benchmark classification models in NLP.

Setup

Rubrix, is a free and open-source tool to explore, annotate, and monitor data for NLP projects.

If you are new to Rubrix, check out the [Github repository](#).

If you have not installed and launched Rubrix yet, check the [Setup and Installation guide](#).

For this tutorial we also need the third party libraries `datasets`, `sklearn`, and `cleanlab`, which can be installed via `pip`:

```
[ ]: %pip install datasets scikit-learn cleanlab -qqq
```

Note

If you want to skip the first three sections of this tutorial, and only uncover and correct the label errors in the Rubrix web app, you can load the records directly from the [Hugging Face Hub](#):

```
import rubrix as rb
from datasets import load_dataset

records_with_label_errors = rb.read_datasets(
    load_dataset("rubrix/cleanlab-label_errors", split="train"),
    task="TextClassification",
)
```

1. Load datasets

We start by downloading the `ag_news` dataset via the very convenient `datasets` library.

```
[ ]: from datasets import load_dataset

# download data
dataset = load_dataset('ag_news')
```

We then extract the train and test set, as well as the labels of this classification task. We also shuffle the train set, since by default it is ordered by the classification label.

```
[ ]: # get train set and shuffle
ds_train = dataset["train"].shuffle(seed=43)

# get test set
```

(continues on next page)

(continued from previous page)

```
ds_test = dataset["test"]

# get classification labels
labels = ds_train.features["label"].names
```

2. Train model

For this tutorial we will use a multinomial **Naive Bayes classifier**, a lightweight and easy to train `sklearn` model. However, you can use any model of your choice as long as it includes the probabilities for all labels in its predictions.

The features for our classifier will be simply the token counts of our input text.

```
[ ]: from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.pipeline import Pipeline

# define our classifier as a pipeline of token counts + naive bayes model
classifier = Pipeline([
    ('vect', CountVectorizer()),
    ('clf', MultinomialNB())
])
```

After defining our classifier, we can fit it with our train set. Since we are using a rather lightweight model, this should not take too long.

```
[ ]: # fit the classifier
classifier.fit(
    x=ds_train["text"],
    y=ds_train["label"]
)
```

Let us check how our model performs on the test set.

```
[ ]: # compute test accuracy
classifier.score(
    x=ds_test["text"],
    y=ds_test["label"],
)
```

We should obtain a decent accuracy of 0.90, especially considering the fact that we only used the token counts as input feature.

3. Get label error candidates

As a first step to get label error candidates in our test set, we have to predict the probabilities for all labels.

```
[ ]: # get predicted probabilities for all labels
probabilities = classifier.predict_proba(ds_test["text"])
```

With the predictions at hand, we create Rubrix records that contain the text input, the prediction of the model, the potential erroneous annotation, and some metadata of your choice.


```
[ ]: import rubrix as rb

# create records for the test set
records = [
    rb.TextClassificationRecord(
        text=data["text"],
        prediction=list(zip(labels, prediction)),
        annotation=labels[data["label"]],
        metadata={"split": "test"}
    )
    for data, prediction in zip(ds_test, probabilities)
]
```

We could log these records directly to Rubrix and conveniently inspect them by eye, checking the annotation of each text input. But here we will use a quicker way by leveraging Rubrix’s built-in support for [cleanlab](#). You simply import the `find_label_errors` function from *Rubrix* and pass in the list of records. That’s it.

```
[ ]: from rubrix.labeling.text_classification import find_label_errors

# get records with potential label errors
records_with_label_error = find_label_errors(records)
```

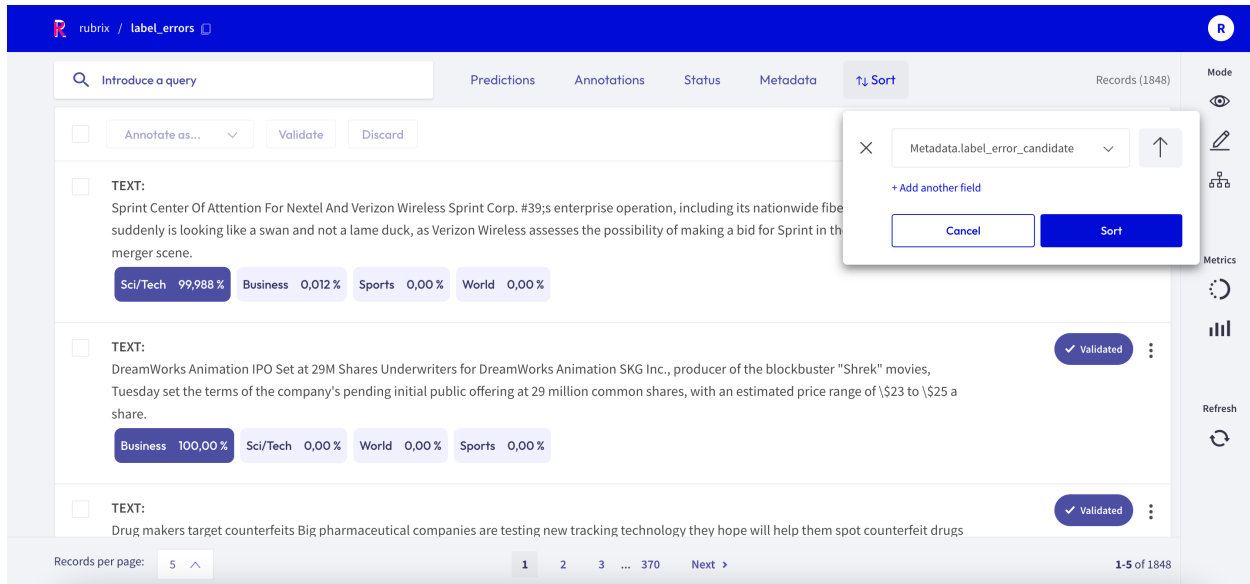
The `records_with_label_error` list contains around 600 candidates for potential label errors, which is more than 8% of our test data.

4. Uncover and correct label errors

Now let us log those records to the *Rubrix* web app to conveniently check them by eye, and to quickly correct potential label errors at the same time.

```
[ ]: # uncover label errors in the Rubrix web app
rb.log(records_with_label_error, "label_errors")
```

By default the records in the `records_with_label_error` list are ordered by their likelihood of containing a label error. They will also contain a metadata called “*label_error_candidate*” by default, which reflects the order in the list. You can use this field in the *Rubrix* web app to sort the records as shown in the screenshot below.



We can confirm that the most likely candidates are indeed clear label errors. Towards the end of the candidate list, the examples get more ambiguous, and it is not immediately obvious if the gold annotations are in fact erroneous.

Summary

With *Rubrix* you can quickly and conveniently find label errors in your data. The built-in support for [cleanlab](#), together with the optimized user experience of the Rubrix web app, makes the process a breeze, and allows you to efficiently correct label errors on the fly.

In just a few steps you can quickly check if your test data set is seriously affected by label errors and if your benchmarks are really meaningful in practice. Maybe your less complex models turn out to beat your resource hungry super model, and the deployment process just got a little bit easier.

Although we only used a sklearn model in this tutorial, Rubrix does not care about the model architecture or the framework you are working with. It just cares about the underlying data and allows you to put more humans in the loop of your AI Lifecycle.

Next steps

Rubrix [Github repo](#) to stay updated.

[Rubrix documentation](#) for more guides and tutorials.

Join the Rubrix community on [Slack](#)

Appendix I: Find label errors in your train data using cross-validation

In order to check your training data for label errors, you can fall back to the cross-validation technique to get out-of-sample predictions. With a classifier from sklearn, cross-validation is really easy and you can do it conveniently in one line of code. Afterwards, the steps of creating *Rubrix* records, finding label error candidates, and uncovering them are the same as shown in the tutorial above.

```
[ ]: from sklearn.model_selection import cross_val_predict
```

(continues on next page)

(continued from previous page)

```
# get predicted probabilities for the whole dataset via cross validation
cv_probs = cross_val_predict(
    classifier,
    X=ds_train["text"] + ds_test["text"],
    y=ds_train["label"] + ds_test["label"],
    cv=int(len(ds_train) / len(ds_test)),
    method="predict_proba",
    n_jobs=-1
)
```

```
[ ]: # create records for the training set
records = [
    rb.TextClassificationRecord(
        text=data["text"],
        prediction=list(zip(labels, prediction)),
        annotation=labels[data["label"]],
        metadata={"split": "train"}
    )
    for data, prediction in zip(ds_train, cv_probs)
]
```

```
[ ]: # uncover label errors for the train set in the Rubrix web app
rb.log(find_label_errors(records), "label_errors_in_train")
```

Here we find around 9400 records with potential label errors, which is also around 8% with respect to the train data.

Appendix II: Log datasets to the Hugging Face Hub

Here we will show you an example of how you can push a Rubrix dataset (records) to the [Hugging Face Hub](#). In this way you can effectively version any of your Rubrix datasets.

```
[ ]: records = rb.load("label_errors")
records.to_dataset().push_to_hub("<name of the dataset on the HF Hub>")
```

4.19 Monitoring

These tutorials show you how Rubrix can help you to monitor your model predictions.

The screenshot shows the Rubrix interface for monitoring predictions. The interface is titled 'default' and has a 'POST' method for 'Predict Transformers'. The 'Parameters' section is empty. The 'Request body' is required and set to 'application/json'. The request body content is '["string"]'. There is a green 'G' icon next to the request body. An 'Execute' button is at the bottom. The 'Responses' section is at the bottom.

Monitor predictions in HTTP API endpoints Learn to monitor the predictions of a FastAPI inference endpoint and log model predictions in a Rubrix dataset.

4.19.1 Monitor predictions in HTTP API endpoints

In this tutorial, you'll learn to monitor the predictions of a FastAPI inference endpoint and log model predictions in a Rubrix dataset. It will walk you through 4 basic steps:

- Load the model you want to use.
- Convert model output to Rubrix format.
- Create a FastAPI endpoint.
- Add middleware to automate logging to Rubrix

Introduction

Models are often deployed via an HTTP API endpoint that is called by a client to obtain the model's predictions. With [FastAPI](#) and *Rubrix* you can easily monitor those predictions and log them to a *Rubrix* dataset. Due to its human-centric UX, *Rubrix* datasets can be comfortably viewed and explored by any team member of your organization. But *Rubrix* also provides automatically computed metrics, both of which help you to keep track of your predictor and spot potential issues early on.

FastAPI and *Rubrix* allow you to deploy and monitor any model you like, but in this tutorial we will focus on the two most common frameworks in the NLP space: [spaCy](#) and [transformers](#). Let's get started!

Setup

Rubrix is a free and open-source tool to explore, annotate, and monitor data for NLP projects.

If you are new to Rubrix, check out the [Github repository](#).

If you have not installed and launched Rubrix yet, check the [Setup and Installation guide](#).

Apart from Rubrix, we'll need a few third party libraries that can be installed via pip:

```
[ ]: %pip install fastapi uvicorn[standard] spacy transformers[torch] -qqq
```

1. Loading models

As a first step, let's load our models. For spacy we need to first download the model before we can instantiate a spacy pipeline with it. Here we use the small English model `en_core_web_sm`, but you can choose any available model on their [hub](#).

```
[ ]: !python -m spacy download en_core_web_sm
```

```
[ ]: import spacy

spacy_pipeline = spacy.load("en_core_web_sm")
```

The “text-classification” pipeline by transformers download's the model for you and by default it will use the `distilbert-base-uncased-finetuned-sst-2-english` model. But you can instantiate the pipeline with any compatible model on their [hub](#).

```
[ ]: from transformers import pipeline

transformers_pipeline = pipeline("text-classification", return_all_scores=True)
```

For more information about using the transformers library with Rubrix, check the tutorial [How to label your data and fine-tune a sentiment classifier](#)

Model output

Let's try the transformer's pipeline in this example:

```
[5]: from pprint import pprint

batch = ['I really like rubrix!']
predictions = transformers_pipeline(batch)
pprint(predictions)

[[{'label': 'NEGATIVE', 'score': 0.0003226407279726118},
 {'label': 'POSITIVE', 'score': 0.9996774196624756}]]
```

Looks like the `predictions` is a list containing lists of two elements : - The first dictionary containing the NEGATIVE sentiment label and its score. - The second dictionary containing the same data but for POSITIVE sentiment.

2. Convert output to Rubrix format

To log the output to Rubrix, we should supply a list of dictionaries, each dictionary containing two keys: - `labels` : value is a list of strings, each string being the label of the sentiment. - `scores` : value is a list of floats, each float being the probability of the sentiment.

```
[6]: rubrix_format = [
    {
        "labels": [ ["label"] for p in prediction],
        "scores": [ ["score"] for p in prediction],
    }
    for prediction in predictions
]
print(rubrix_format)

[{'labels': ['NEGATIVE', 'POSITIVE'],
  'scores': [0.0003226407279726118, 0.9996774196624756]}]
```

3. Create prediction endpoint

```
[ ]: from fastapi import FastAPI
from typing import List

app_transformers = FastAPI()

# prediction endpoint using transformers pipeline
@app_transformers.post("/")
def predict_transformers(batch: List[str]):
    predictions = transformers_pipeline(batch)
    return [
        {
            "labels": [p["label"] for p in prediction],
            "scores": [p["score"] for p in prediction],
        }
        for prediction in predictions
    ]
```

4. Add Rubrix logging middleware to the application

```
[ ]: from rubrix.monitoring.asgi import RubrixLogHTTPMiddleware

app_transformers.add_middleware(
    RubrixLogHTTPMiddleware,
    api_endpoint="/transformers/", #the endpoint that will be logged
    dataset="monitoring_transformers", #your dataset name
    # you could post-process the predict output with a custom record_mapper function
    # record_mapper=custom_text_classification_mapper,
)
```

5. Do the same for spaCy

We'll add a custom mapper to convert spaCy's output to TokenClassificationRecord format

FastAPI application

```
[ ]: from rubrix.monitoring.asgi import RubrixLogHTTPMiddleware, token_classification_mapper

app_spacy = FastAPI()

app_spacy.add_middleware(
    RubrixLogHTTPMiddleware,
    api_endpoint="/spacy/",
    dataset="monitoring_spacy",
    records_mapper=token_classification_mapper
)

# prediction endpoint using spacy pipeline
@app_spacy.post("/")
def predict_spacy(batch: List[str]):
    predictions = []
    for text in batch:
        doc = spacy.pipeline(text) # spaCy Doc creation
        # Entity annotations
        entities = [
            {"label": ent.label_, "start": ent.start_char, "end": ent.end_char}
            for ent in doc.ents
        ]

        prediction = {
            "text": text,
            "entities": entities,
        }
        predictions.append(prediction)
    return predictions
```

6. Putting it all together

Now we can combine everything in order to see our results!

```
[ ]: app = FastAPI()

@app.get("/")
def root():
    return {"message": "alive"}

app.mount("/transformers", app_transformers)
app.mount("/spacy", app_spacy)
```

Launch the application

To launch the application, copy the whole code into a file named `main.py` and run the following command:

```
[ ]: !uvicorn main:app
```

Transformers demo

spaCy demo

Summary

In this tutorial, we learned to automatically log model outputs into Rubrix. This can be used to continuously and transparently monitor HTTP inference endpoints.

Next steps

Rubrix [Github repo](#) to stay updated.

[Rubrix documentation](#) for more guides and tutorials.

Join the Rubrix community! A good place to start is the [discussion forum](#).

```
[ ]:
```

4.20 Telemetry

Rubrix uses telemetry to report anonymous usage and error information. As an open-source software, this type of information is important to improve and understand how the product is used.

4.20.1 How to opt-out

You can opt-out of telemetry reporting using the ENV variable `RUBRIX_ENABLE_TELEMETRY` before launching the server. Setting this variable to `0` will completely disable telemetry reporting.

If you are a Linux/MacOs users you should run:

```
export RUBRIX_ENABLE_TELEMETRY=0
```

If you are Windows users you should run:

```
set RUBRIX_ENABLE_TELEMETRY=0
```

To opt-in again, you can set the variable to `1`.

4.20.2 Why reporting telemetry

Anonymous telemetry information enable us to continuously improve the product and detect recurring problems to better serve all users. We collect aggregated information about general usage and errors. We do NOT collect any information of users' data records, datasets, or metadata information.

4.20.3 Sensitive data

We do not collect any piece of information related to the source data you store in Rubrix. We don't identify individual users. Your data does not leave your server at any time:

- No dataset record is collected.
- No dataset names or metadata are collected.

4.20.4 Information reported

The following usage and error information is reported:

- The code of the raised error
- The user-agent and accept-language http headers
- Task name and number of records for bulk operations
- An anonymous generated user uuid
- The rubrix version running the server
- The python version, e.g. 3.8.13
- The system/OS name, such as Linux, Darwin, Windows
- The system's release version, e.g. Darwin Kernel Version 21.5.0: Tue Apr 26 21:08:22 PDT 2022; root:xnu-8020
- The machine type, e.g. AMD64
- The underlying platform spec with as much useful information as possible. (ej. macOS-10.16-x86_64-i386-64bit)

This is performed by registering information from the following API methods:

- /api/me
- /api/dataset/{name}/{task}:bulk
- Raised server API errors

For transparency, you can inspect the source code where this is performed here (add link to the source).

If you have any doubts, don't hesitate to join our [Slack channel](#) or open a GitHub issue. We'd be very happy to discuss about how we can improve this.

4.21 Python

The python reference guide for Rubrix. This section contains:

- *Client*: The base client module
- *Metrics*: The module for dataset metrics
- *Labeling*: A toolbox to enhance your labeling workflow (weak labels, noisy labels, etc.)
- *Listeners*: This module contains all you need to define and configure dataset rubrix listeners

4.21.1 Client

Here we describe the Python client of Rubrix that we divide into three basic modules:

- *Methods*: These methods make up the interface to interact with Rubrix's REST API.
- *Records*: You need to wrap your data in these *Records* for Rubrix to understand it.
- *Datasets*: Datasets: You can wrap your records around these *Datasets* for extra functionality.

Methods

`rubrix.copy(dataset, name_of_copy, workspace=None)`

Creates a copy of a dataset including its tags and metadata

Parameters

- **dataset** (*str*) – Name of the source dataset
- **name_of_copy** (*str*) – Name of the copied dataset
- **workspace** (*str*) – If provided, dataset will be copied to that workspace

Examples

```
>>> import rubrix as rb
>>> rb.copy("my_dataset", name_of_copy="new_dataset")
>>> rb.load("new_dataset")
```

`rubrix.delete(name)`

Deletes a dataset.

Parameters **name** (*str*) – The dataset name.

Return type None

Examples

```
>>> import rubrix as rb
>>> rb.delete(name="example-dataset")
```

rubrix.delete_records(name, query=None, ids=None, discard_only=False, discard_when_forbidden=True)

Delete records from a Rubrix dataset.

Parameters

- **name** (str) – The dataset name.
- **query** (Optional[str]) – An ElasticSearch query with the [query string syntax](#)
- **ids** (Optional[List[Union[str, int]]]) – If provided, deletes dataset records with given ids.
- **discard_only** (bool) – If *True*, matched records won't be deleted. Instead, they will be marked as *Discarded*
- **discard_when_forbidden** (bool) – Only super-user or dataset creator can delete records from a dataset. So, running “hard” deletion for other users will raise an *ForbiddenApiError* error. If this parameter is *True*, the client API will automatically try to mark as *Discarded* records instead. Default, *True*

Returns The total of matched records and real number of processed errors. These numbers could not be the same if some data conflicts are found during operations (some matched records change during deletion).

Return type *Tuple*[int, int]

Examples

```
>>> ## Delete by id
>>> import rubrix as rb
>>> rb.delete_records(name="example-dataset", ids=[1,3,5])
>>> ## Discard records by query
>>> import rubrix as rb
>>> rb.delete_records(name="example-dataset", query="metadata.code=33", discard_
↳ only=True)
```

rubrix.get_workspace()

Returns the name of the active workspace.

Returns The name of the active workspace as a string.

Return type str

rubrix.init(api_url=None, api_key=None, workspace=None, timeout=60, extra_headers=None)

Init the Python client.

We will automatically init a default client for you when calling other client methods. The arguments provided here will overwrite your corresponding environment variables.

Parameters

- **api_url** (Optional[str]) – Address of the REST API. If *None* (default) and the env variable RUBRIX_API_URL is not set, it will default to *http://localhost:6900*.

- **api_key** (*Optional[str]*) – Authentication key for the REST API. If *None* (default) and the env variable RUBRIX_API_KEY is not set, it will default to *rubrix.apikey*.
- **workspace** (*Optional[str]*) – The workspace to which records will be logged/loaded. If *None* (default) and the env variable RUBRIX_WORKSPACE is not set, it will default to the private user workspace.
- **timeout** (*int*) – Wait *timeout* seconds for the connection to timeout. Default: 60.
- **extra_headers** (*Optional[Dict[str, str]]*) – Extra HTTP headers sent to the server. You can use this to customize the headers of Rubrix client requests, like additional security restrictions. Default: *None*.

Examples

```
>>> import rubrix as rb
>>> rb.init(api_url="http://localhost:9090", api_key="4AkeAPIk3Y")
>>> # Customizing request headers
>>> headers = {"X-Client-id":"id","X-Secret":"secret"}
>>> rb.init(api_url="http://localhost:9090", api_key="4AkeAPIk3Y", extra_
↳ headers=headers)
```

rubrix.load(*name, query=None, ids=None, limit=None, id_from=None, as_pandas=None*)

Loads a Rubrix dataset.

name: The dataset name.

query: An ElasticSearch query with the *query string syntax*

ids: If provided, load dataset records with given ids.

limit: The number of records to retrieve.

id_from: If provided, starts gathering the records starting from that Record. As the Records returned with the load method are sorted by ID, 'id_from' can be used to load using batches.

as_pandas: DEPRECATED! To get a pandas DataFrame do `rb.load('my_dataset').to_pandas()`.

A Rubrix dataset.

Examples

Basic Loading: load the samples sorted by their ID `>>> import rubrix as rb >>> dataset = rb.load(name="example-dataset")`

Iterate over a large dataset: When dealing with a large dataset you might want to load it in batches to optimize memory consumption and avoid network timeouts. To that end, a simple batch-iteration over the whole database can be done employing the *from_id* parameter. This parameter will act as a delimiter, retrieving the N items after the given id, where N is determined by the *limit* parameter. **NOTE** If no *limit* is given the whole dataset after that ID will be retrieved.

```
>>> import rubrix as rb
>>> dataset_batch_1 = rb.load(name="example-dataset", limit=1000)
>>> dataset_batch_2 = rb.load(name="example-dataset", limit=1000, id_from=dataset_
↳ batch_1[-1].id)
```

Parameters

- **name** (*str*) –
- **query** (*Optional[str]*) –
- **ids** (*Optional[List[Union[str, int]]]*) –
- **limit** (*Optional[int]*) –
- **id_from** (*Optional[str]*) –

Return type *Union[rubrix.client.datasets.DatasetForTextClassification, rubrix.client.datasets.DatasetForTokenClassification, rubrix.client.datasets.DatasetForText2Text]*

`rubrix.log(records, name, tags=None, metadata=None, chunk_size=500, verbose=True, background=False)`

Logs Records to Rubrix.

The logging happens asynchronously in a background thread.

Parameters

- **records** (*Union[rubrix.client.models.TextClassificationRecord, rubrix.client.models.TokenClassificationRecord, rubrix.client.models.Text2TextRecord, Iterable[Union[rubrix.client.models.TextClassificationRecord, rubrix.client.models.TokenClassificationRecord, rubrix.client.models.Text2TextRecord]], rubrix.client.datasets.DatasetForTextClassification, rubrix.client.datasets.DatasetForTokenClassification, rubrix.client.datasets.DatasetForText2Text]*) – The record, an iterable of records, or a dataset to log.
- **name** (*str*) – The dataset name.
- **tags** (*Optional[Dict[str, str]]*) – A dictionary of tags related to the dataset.
- **metadata** (*Optional[Dict[str, Any]]*) – A dictionary of extra info for the dataset.
- **chunk_size** (*int*) – The chunk size for a data bulk.
- **verbose** (*bool*) – If True, shows a progress bar and prints out a quick summary at the end.
- **background** (*bool*) – If True, we will NOT wait for the logging process to finish and return an `asyncio.Future` object. You probably want to set `verbose` to `False` in that case.

Returns Summary of the response from the REST API. If the `background` argument is set to `True`, an `asyncio.Future` will be returned instead.

Return type *Union[rubrix.client.models.BulkResponse, asyncio.Future]*

Examples

```
>>> import rubrix as rb
>>> record = rb.TextClassificationRecord(
...     text="my first rubrix example",
...     prediction=[('spam', 0.8), ('ham', 0.2)]
... )
>>> rb.log(record, name="example-dataset")
1 records logged to http://localhost:6900/datasets/rubrix/example-dataset
BulkResponse(dataset='example-dataset', processed=1, failed=0)
>>>
```

(continues on next page)

(continued from previous page)

```
>>> # Logging records in the background
>>> rb.log(record, name="example-dataset", background=True, verbose=False)
<Future at 0x7f675a1ffa0 state=pending>
```

`rubrix.set_workspace(workspace)`

Sets the active workspace.

Parameters `workspace` (*str*) – The new workspace

Records

This module contains the data models for the interface

```
class rubrix.client.models.Text2TextRecord(*, text, prediction=None, prediction_agent=None,
                                           annotation=None, annotation_agent=None, id=None,
                                           metadata=None, status=None, event_timestamp=None,
                                           metrics=None, search_keywords=None)
```

Record for a text to text task

Parameters

- **text** (*str*) – The input of the record
- **prediction** (*Optional[List[Union[str, Tuple[str, float]]]*) – A list of strings or tuples containing predictions for the input text. If tuples, the first entry is the predicted text, the second entry is its corresponding score.
- **prediction_agent** (*Optional[str]*) – Name of the prediction agent. By default, this is set to the hostname of your machine.
- **annotation** (*Optional[str]*) – A string representing the expected output text for the given input text.
- **annotation_agent** (*Optional[str]*) – Name of the prediction agent. By default, this is set to the hostname of your machine.
- **id** (*Optional[Union[int, str]]*) – The id of the record. By default (None), we will generate a unique ID for you.
- **metadata** (*Optional[Dict[str, Any]]*) – Meta data for the record. Defaults to {}.
- **status** (*Optional[str]*) – The status of the record. Options: ‘Default’, ‘Edited’, ‘Discarded’, ‘Validated’. If an annotation is provided, this defaults to ‘Validated’, otherwise ‘Default’.
- **event_timestamp** (*Optional[datetime.datetime]*) – The timestamp of the record.
- **metrics** (*Optional[Dict[str, Any]]*) – READ ONLY! Metrics at record level provided by the server when using *rb.load*. This attribute will be ignored when using *rb.log*.
- **search_keywords** (*Optional[List[str]]*) – READ ONLY! Relevant record keywords/terms for provided query when using *rb.load*. This attribute will be ignored when using *rb.log*.

Return type None

Examples

```
>>> import rubrix as rb
>>> record = rb.Text2TextRecord(
...     text="My name is Sarah and I love my dog.",
...     prediction=["Je m'appelle Sarah et j'aime mon chien."]
... )
```

classmethod `prediction_as_tuples(prediction)`

Preprocess the predictions and wraps them in a tuple if needed

Parameters `prediction` (*Optional[List[Union[str, Tuple[str, float]]]*) –

class `rubrix.client.models.TextClassificationRecord`(*, *text=None, inputs=None, prediction=None, prediction_agent=None, annotation=None, annotation_agent=None, multi_label=False, explanation=None, id=None, metadata=None, status=None, event_timestamp=None, metrics=None, search_keywords=None*)

Record for text classification

Parameters

- **text** (*Optional[str]*) – The input of the record. Provide either ‘text’ or ‘inputs’.
- **inputs** (*Optional[Union[str, List[str], Dict[str, Union[str, List[str]]]]*) – Various inputs of the record (see examples below). Provide either ‘text’ or ‘inputs’.
- **prediction** (*Optional[List[Tuple[str, float]]]*) – A list of tuples containing the predictions for the record. The first entry of the tuple is the predicted label, the second entry is its corresponding score.
- **prediction_agent** (*Optional[str]*) – Name of the prediction agent. By default, this is set to the hostname of your machine.
- **annotation** (*Optional[Union[str, List[str]]]*) – A string or a list of strings (multi-label) corresponding to the annotation (gold label) for the record.
- **annotation_agent** (*Optional[str]*) – Name of the prediction agent. By default, this is set to the hostname of your machine.
- **multi_label** (*bool*) – Is the prediction/annotation for a multi label classification task? Defaults to *False*.
- **explanation** (*Optional[Dict[str, List[rubrix.client.models.TokenAttributions]]]*) – A dictionary containing the attributions of each token to the prediction. The keys map the input of the record (see *inputs*) to the *TokenAttributions*.
- **id** (*Optional[Union[int, str]]*) – The id of the record. By default (*None*), we will generate a unique ID for you.
- **metadata** (*Optional[Dict[str, Any]]*) – Meta data for the record. Defaults to *{}*.
- **status** (*Optional[str]*) – The status of the record. Options: ‘Default’, ‘Edited’, ‘Discarded’, ‘Validated’. If an annotation is provided, this defaults to ‘Validated’, otherwise ‘Default’.
- **event_timestamp** (*Optional[datetime.datetime]*) – The timestamp of the record.

- **metrics** (*Optional[Dict[str, Any]]*) – READ ONLY! Metrics at record level provided by the server when using *rb.load*. This attribute will be ignored when using *rb.log*.
- **search_keywords** (*Optional[List[str]]*) – READ ONLY! Relevant record keywords/terms for provided query when using *rb.load*. This attribute will be ignored when using *rb.log*.

Return type None

Examples

```
>>> # Single text input
>>> import rubrix as rb
>>> record = rb.TextClassificationRecord(
...     text="My first rubrix example",
...     prediction=[('eng', 0.9), ('esp', 0.1)]
... )
>>>
>>> # Various inputs
>>> record = rb.TextClassificationRecord(
...     inputs={
...         "subject": "Has ganado 1 million!",
...         "body": "Por usar Rubrix te ha tocado este premio: <link>"
...     },
...     prediction=[('spam', 0.99), ('ham', 0.01)],
...     annotation="spam"
... )
```

class rubrix.client.models.TokenAttributions(*, token, attributions=None)

Attribution of the token to the predicted label.

In the Rubrix app this is only supported for TextClassificationRecord and the multi_label=False case.

Parameters

- **token** (*str*) – The input token.
- **attributions** (*Dict[str, float]*) – A dictionary containing label-attribution pairs.

Return type None

class rubrix.client.models.TokenClassificationRecord(text=None, tokens=None, tags=None, *, prediction=None, prediction_agent=None, annotation=None, annotation_agent=None, id=None, metadata=None, status=None, event_timestamp=None, metrics=None, search_keywords=None)

Record for a token classification task

Parameters

- **text** (*Optional[str]*) – The input of the record
- **tokens** (*Optional[Union[List[str], Tuple[str, ...]]]*) – The tokenized input of the record. We use this to guide the annotation process and to cross-check the spans of your *prediction/annotation*.

- **prediction** (*Optional[List[Union[Tuple[str, int, int], Tuple[str, int, int, Optional[float]]]]*) – A list of tuples containing the predictions for the record. The first entry of the tuple is the name of predicted entity, the second and third entry correspond to the start and stop character index of the entity. The fourth entry is optional and corresponds to the score of the entity (a float number between 0 and 1).
- **prediction_agent** (*Optional[str]*) – Name of the prediction agent. By default, this is set to the hostname of your machine.
- **annotation** (*Optional[List[Tuple[str, int, int]]]*) – A list of tuples containing annotations (gold labels) for the record. The first entry of the tuple is the name of the entity, the second and third entry correspond to the start and stop char index of the entity.
- **annotation_agent** (*Optional[str]*) – Name of the prediction agent. By default, this is set to the hostname of your machine.
- **id** (*Optional[Union[int, str]]*) – The id of the record. By default (None), we will generate a unique ID for you.
- **metadata** (*Optional[Dict[str, Any]]*) – Meta data for the record. Defaults to {}.
- **status** (*Optional[str]*) – The status of the record. Options: 'Default', 'Edited', 'Discarded', 'Validated'. If an annotation is provided, this defaults to 'Validated', otherwise 'Default'.
- **event_timestamp** (*Optional[datetime.datetime]*) – The timestamp of the record.
- **metrics** (*Optional[Dict[str, Any]]*) – READ ONLY! Metrics at record level provided by the server when using *rb.load*. This attribute will be ignored when using *rb.log*.
- **search_keywords** (*Optional[List[str]]*) – READ ONLY! Relevant record keywords/terms for provided query when using *rb.load*. This attribute will be ignored when using *rb.log*.
- **tags** (*Optional[List[str]]*) –

Return type None

Examples

```
>>> import rubrix as rb
>>> record = rb.TokenClassificationRecord(
...     text = "Michael is a professor at Harvard",
...     tokens = ["Michael", "is", "a", "professor", "at", "Harvard"],
...     prediction = [('NAME', 0, 7), ('LOC', 26, 33)]
... )
```

char_id2token_id(*char_idx*)

DEPRECATED, please use the `rubrix.utisl.SpanUtils.SpanUtils.char_to_token_idx` dict instead.

Parameters **char_idx** (*int*) –

Return type *Optional[int]*

spans2iob(*spans=None*)

DEPRECATED, please use the `rubrix.utils.SpanUtils.to_tags()` method.

Parameters **spans** (*Optional[List[Tuple[str, int, int]]]*) –

Return type *Optional[List[str]]*

token_span(*token_idx*)

DEPRECATED, please use the `rubrix.utisl.span_utils.SpanUtils.token_to_char_idx` dict instead.

Parameters **token_idx**(*int*) –

Return type *Tuple[int, int]*

Datasets

class `rubrix.client.datasets.DatasetForText2Text`(*records=None*)

This Dataset contains `Text2TextRecord` records.

It allows you to export/import records into/from different formats, loop over the records, and access them by index.

Parameters **records** (*Optional[List[rubrix.client.models.Text2TextRecord]]*) – A list of `Text2TextRecord`'s.

Raises **WrongRecordTypeError** – When the record type in the provided list does not correspond to the dataset type.

Examples

```
>>> # Import/export records:
>>> import rubrix as rb
>>> dataset = rb.DatasetForText2Text.from_pandas(my_dataframe)
>>> dataset.to_datasets()
>>>
>>> # Passing in a list of records:
>>> records = [
...     rb.Text2TextRecord(text="example"),
...     rb.Text2TextRecord(text="another example"),
... ]
>>> dataset = rb.DatasetForText2Text(records)
>>> assert len(dataset) == 2
>>>
>>> # Looping over the dataset:
>>> for record in dataset:
...     print(record)
>>>
>>> # Indexing into the dataset:
>>> dataset[0]
... rb.Text2TextRecord(text="example")
>>> dataset[0] = rb.Text2TextRecord(text="replaced example")
```

classmethod `from_datasets`(*dataset, text=None, annotation=None, metadata=None, id=None*)

Imports records from a `datasets.Dataset`.

Columns that are not supported are ignored.

Parameters

- **dataset** (`datasets.Dataset`) – A datasets Dataset from which to import the records.

- **text** (*Optional[str]*) – The field name used as record text. Default: *None*
- **annotation** (*Optional[str]*) – The field name used as record annotation. Default: *None*
- **metadata** (*Optional[Union[str, List[str]]]*) – The field name used as record metadata. Default: *None*
- **id** (*Optional[str]*) –

Returns The imported records in a Rubrix Dataset.

Return type *DatasetForText2Text*

Examples

```
>>> import datasets
>>> ds = datasets.Dataset.from_dict({
...     "text": ["my example"],
...     "prediction": ["mi ejemplo", "ejemplo mio"]
... })
>>> # or
>>> ds = datasets.Dataset.from_dict({
...     "text": ["my example"],
...     "prediction": [{"text": "mi ejemplo", "score": 0.9}]
... })
>>> DatasetForText2Text.from_datasets(ds)
```

classmethod **from_pandas**(*dataframe*)

Imports records from a *pandas.DataFrame*.

Columns that are not supported are ignored.

Parameters **dataframe** (*pandas.core.frame.DataFrame*) – A pandas DataFrame from which to import the records.

Returns The imported records in a Rubrix Dataset.

Return type *rubrix.client.datasets.DatasetForText2Text*

class **rubrix.client.datasets.DatasetForTextClassification**(*records=None*)

This Dataset contains TextClassificationRecord records.

It allows you to export/import records into/from different formats, loop over the records, and access them by index.

Parameters **records** (*Optional[List[rubrix.client.models.TextClassificationRecord]]*) – A list of `TextClassificationRecord`'s.

Raises **WrongRecordTypeError** – When the record type in the provided list does not correspond to the dataset type.

Examples

```
>>> # Import/export records:
>>> import rubrix as rb
>>> dataset = rb.DatasetForTextClassification.from_pandas(my_dataframe)
>>> dataset.to_datasets()
>>>
>>> # Looping over the dataset:
>>> for record in dataset:
...     print(record)
>>>
>>> # Passing in a list of records:
>>> records = [
...     rb.TextClassificationRecord(text="example"),
...     rb.TextClassificationRecord(text="another example"),
... ]
>>> dataset = rb.DatasetForTextClassification(records)
>>> assert len(dataset) == 2
>>>
>>> # Indexing into the dataset:
>>> dataset[0]
... rb.TextClassificationRecord(text="example")
>>> dataset[0] = rb.TextClassificationRecord(text="replaced example")
```

classmethod `from_datasets`(dataset, text=None, id=None, inputs=None, annotation=None, metadata=None)

Imports records from a `datasets.Dataset`.

Columns that are not supported are ignored.

Parameters

- **dataset** (`datasets.Dataset`) – A datasets Dataset from which to import the records.
- **text** (`Optional[str]`) – The field name used as record text. Default: *None*
- **id** (`Optional[str]`) – The field name used as record id. Default: *None*
- **inputs** (`Optional[Union[str, List[str]]]`) – A list of field names used for record inputs. Default: *None*
- **annotation** (`Optional[str]`) – The field name used as record annotation. Default: *None*
- **metadata** (`Optional[Union[str, List[str]]]`) – The field name used as record metadata. Default: *None*

Returns The imported records in a Rubrix Dataset.

Return type `DatasetForTextClassification`

Examples

```
>>> import datasets
>>> ds = datasets.Dataset.from_dict({
...     "inputs": ["example"],
...     "prediction": [
...         [{"label": "LABEL1", "score": 0.9}, {"label": "LABEL2", "score": 0.
↵1}]
...     ]
... })
>>> DatasetForTextClassification.from_datasets(ds)
```

classmethod `from_pandas(dataframe)`

Imports records from a *pandas.DataFrame*.

Columns that are not supported are ignored.

Parameters `dataframe` (*pandas.core.frame.DataFrame*) – A pandas DataFrame from which to import the records.

Returns The imported records in a Rubrix Dataset.

Return type *rubrix.client.datasets.DatasetForTextClassification*

prepare_for_training()

Prepares the dataset for training.

This will return a *datasets.Dataset* with a *label* column, and one column for each key in the *inputs* dictionary of the records:

- Records without an annotation are removed.
- The *label* column corresponds to the annotations of the records.
- Labels are transformed to integers.

Returns A *datasets.Dataset* with a *label* column and several *inputs* columns.

Return type *datasets.Dataset*

Examples

```
>>> import rubrix as rb
>>> rb_dataset = rb.DatasetForTextClassification([
...     rb.TextClassificationRecord(
...         inputs={"header": "my header", "content": "my content"},
...         annotation="SPAM",
...     )
... ])
>>> rb_dataset.prepare_for_training().features
{'header': Value(dtype='string'),
 'content': Value(dtype='string'),
 'label': ClassLabel(num_classes=1, names=['SPAM'])}
```

class `rubrix.client.datasets.DatasetForTokenClassification(records=None)`

This Dataset contains *TokenClassificationRecord* records.

It allows you to export/import records into/from different formats, loop over the records, and access them by index.

Parameters **records** (*Optional[List[rubrix.client.models.TokenClassificationRecord]]*) – A list of `TokenClassificationRecord`s`.

Raises **WrongRecordTypeError** – When the record type in the provided list does not correspond to the dataset type.

Examples

```
>>> # Import/export records:
>>> import rubrix as rb
>>> dataset = rb.DatasetForTokenClassification.from_pandas(my_dataframe)
>>> dataset.to_datasets()
>>>
>>> # Looping over the dataset:
>>> assert len(dataset) == 2
>>> for record in dataset:
...     print(record)
>>>
>>> # Passing in a list of records:
>>> import rubrix as rb
>>> records = [
...     rb.TokenClassificationRecord(text="example", tokens=["example"]),
...     rb.TokenClassificationRecord(text="another example", tokens=["another",
↪ "example"]),
... ]
>>> dataset = rb.DatasetForTokenClassification(records)
>>>
>>> # Indexing into the dataset:
>>> dataset[0]
... rb.TokenClassificationRecord(text="example", tokens=["example"])
>>> dataset[0] = rb.TokenClassificationRecord(text="replace example", tokens=[
↪ "replace", "example"])
```

classmethod **from_datasets**(*dataset, text=None, id=None, tokens=None, tags=None, metadata=None*)

Imports records from a `datasets.Dataset`.

Columns that are not supported are ignored.

Parameters

- **dataset** (*datasets.Dataset*) – A datasets Dataset from which to import the records.
- **text** (*Optional[str]*) – The field name used as record text. Default: *None*
- **id** (*Optional[str]*) – The field name used as record id. Default: *None*
- **tokens** (*Optional[str]*) – The field name used as record tokens. Default: *None*
- **tags** (*Optional[str]*) – The field name used as record tags. Default: *None*
- **metadata** (*Optional[Union[str, List[str]]]*) – The field name used as record metadata. Default: *None*

Returns The imported records in a Rubrix Dataset.

Return type *DatasetForTokenClassification*

Examples

```
>>> import datasets
>>> ds = datasets.Dataset.from_dict({
...     "text": ["my example"],
...     "tokens": [["my", "example"]],
...     "prediction": [
...         [{"label": "LABEL1", "start": 3, "end": 10, "score": 1.0}]
...     ])
>>> DatasetForTokenClassification.from_datasets(ds)
```

classmethod `from_pandas(dataframe)`

Imports records from a *pandas.DataFrame*.

Columns that are not supported are ignored.

Parameters `dataframe` (*pandas.core.frame.DataFrame*) – A pandas DataFrame from which to import the records.

Returns The imported records in a Rubrix Dataset.

Return type *rubrix.client.datasets.DatasetForTokenClassification*

prepare_for_training(*framework='transformers', lang=None*)

Prepares the dataset for training.

This will return a *datasets.Dataset* with all columns returned by *to_datasets* method and an additional *ner_tags* column:

- Records without an annotation are removed.
- The *ner_tags* column corresponds to the iob tags sequences for annotations of the records
- The iob tags are transformed to integers.

Parameters

- **framework** (*Union[rubrix.client.datasets.Framework, str]*) – A stringenum specifying the framework for the training. “transformers” and “spacy” are currently supported. Default: *transformers*
- **lang** (*Optional[spacy.Language]*) – The spacy nlp Language pipeline used to process the dataset. (Only for spacy framework)

Returns A *datasets.Dataset* with a *ner_tags* column and all columns returned by *to_datasets* for “transformers” framework. A spacy DocBin ready to use for training a spacy NER model for “spacy” framework.

Return type *Union[datasets.Dataset, spacy.tokens.DocBin]*

Examples

```
>>> import rubrix as rb
>>> rb_dataset = rb.DatasetForTokenClassification([
...     rb.TokenClassificationRecord(
...         text="The text",
...         tokens=["The", "text"],
...         annotation=[("TAG", 0, 2)],
...     )
... ])
>>> rb_dataset.prepare_for_training().features
{'text': Value(dtype='string'),
 'tokens': Sequence(feature=Value(dtype='string'), length=-1),
 'prediction': Value(dtype='null'),
 'prediction_agent': Value(dtype='null'),
 'annotation': [{'end': Value(dtype='int64'),
                  'label': Value(dtype='string'),
                  'start': Value(dtype='int64')}],
 'annotation_agent': Value(dtype='null'),
 'id': Value(dtype='null'),
 'metadata': Value(dtype='null'),
 'status': Value(dtype='string'),
 'event_timestamp': Value(dtype='null'),
 'metrics': Value(dtype='null'),
 'ner_tags': [ClassLabel(num_classes=3, names=['O', 'B-TAG', 'I-TAG'])]}
```

`rubrix.client.datasets.read_datasets(dataset, task, **kwargs)`

Reads a datasets Dataset and returns a Rubrix Dataset

Parameters

- **dataset** (`datasets.Dataset`) – Dataset to be read in.
- **task** (`Union[str, rubrix.client.sdk.datasets.models.TaskType]`) – Task for the dataset, one of: ["TextClassification", "TokenClassification", "Text2Text"].
- ****kwargs** – Passed on to the task-specific `DatasetFor*.from_datasets()` method.

Returns A Rubrix dataset for the given task.

Return type `Union[rubrix.client.datasets.DatasetForTextClassification, rubrix.client.datasets.DatasetForTokenClassification, rubrix.client.datasets.DatasetForText2Text]`

Examples

```
>>> # Read text classification records from a datasets Dataset
>>> import datasets
>>> ds = datasets.Dataset.from_dict({
...     "inputs": ["example"],
...     "prediction": [
...         [{"label": "LABEL1", "score": 0.9}, {"label": "LABEL2", "score": 0.1}]
...     ]
... })
>>> read_datasets(ds, task="TextClassification")
>>>
```

(continues on next page)

(continued from previous page)

```
>>> # Read token classification records from a datasets Dataset
>>> ds = datasets.Dataset.from_dict({
...     "text": ["my example"],
...     "tokens": [["my", "example"]],
...     "prediction": [
...         [{"label": "LABEL1", "start": 3, "end": 10}]
...     ]
... })
>>> read_datasets(ds, task="TokenClassification")
>>>
>>> # Read text2text records from a datasets Dataset
>>> ds = datasets.Dataset.from_dict({
...     "text": ["my example"],
...     "prediction": [{"mi ejemplo", "ejemplo mio"}]
... })
>>> # or
>>> ds = datasets.Dataset.from_dict({
...     "text": ["my example"],
...     "prediction": [{"text": "mi ejemplo", "score": 0.9}]
... })
>>> read_datasets(ds, task="Text2Text")
```

`rubrix.client.datasets.read_pandas(dataframe, task)`

Reads a pandas DataFrame and returns a Rubrix Dataset

Parameters

- **dataframe** (`pandas.core.frame.DataFrame`) – Dataframe to be read in.
- **task** (`Union[str, rubrix.client.sdk.datasets.models.TaskType]`) – Task for the dataset, one of: ["TextClassification", "TokenClassification", "Text2Text"]

Returns A Rubrix dataset for the given task.

Return type `Union[rubrix.client.datasets.DatasetForTextClassification, rubrix.client.datasets.DatasetForTokenClassification, rubrix.client.datasets.DatasetForText2Text]`

Examples

```
>>> # Read text classification records from a pandas DataFrame
>>> import pandas as pd
>>> df = pd.DataFrame({
...     "inputs": ["example"],
...     "prediction": [
...         [("LABEL1", 0.9), ("LABEL2", 0.1)]
...     ]
... })
>>> read_pandas(df, task="TextClassification")
>>>
>>> # Read token classification records from a datasets Dataset
>>> df = pd.DataFrame({
...     "text": ["my example"],
...     "tokens": [["my", "example"]],
```

(continues on next page)

(continued from previous page)

```
...     "prediction": [
...         ["LABEL1", 3, 10]]
...     ]
... })
>>> read_pandas(df, task="TokenClassification")
>>>
>>> # Read text2text records from a datasets Dataset
>>> df = pd.DataFrame({
...     "text": ["my example"],
...     "prediction": [["mi ejemplo", "ejemplo mio"]]
... })
>>> # or
>>> ds = pd.DataFrame({
...     "text": ["my example"],
...     "prediction": [["mi ejemplo", 0.9]]
... })
>>> read_pandas(df, task="Text2Text")
```

4.21.2 Metrics

Here we describe the available metrics in Rubrix:

- Text classification: Metrics for text classification
- Token classification: Metrics for token classification

Text classification

`rubrix.metrics.text_classification.metrics.f1(name, query=None)`

Computes the single label f1 metric for a dataset

Parameters

- **name** (*str*) – The dataset name.
- **query** (*Optional[str]*) – An Elasticsearch query with the [query string syntax](<https://rubrix.readthedocs.io/en/stable/guides/queries.html>)

Returns The f1 metric summary

Return type `rubrix.metrics.models.MetricSummary`

Examples

```
>>> from rubrix.metrics.text_classification import f1
>>> summary = f1(name="example-dataset")
>>> summary.visualize() # will plot a bar chart with results
>>> summary.data # returns the raw result data
```

`rubrix.metrics.text_classification.metrics.f1_multilabel(name, query=None)`

Computes the multi-label label f1 metric for a dataset

Parameters

- **name** (*str*) – The dataset name.
- **query** (*Optional[str]*) – An ElasticSearch query with the [query string syntax](<https://rubrix.readthedocs.io/en/stable/guides/queries.html>)

Returns The f1 metric summary

Return type `rubrix.metrics.models.MetricSummary`

Examples

```
>>> from rubrix.metrics.text_classification import f1_multilabel
>>> summary = f1_multilabel(name="example-dataset")
>>> summary.visualize() # will plot a bar chart with results
>>> summary.data # returns the raw result data
```

Token classification

class `rubrix.metrics.token_classification.metrics.ComputeFor`(*value*)

An enumeration.

`rubrix.metrics.token_classification.metrics.entity_capitalness`(*name*, *query=None*, *compute_for=ComputeFor.PREDICTIONS*)

Computes the entity capitalness. The entity capitalness splits the entity mention shape in 4 groups:

UPPER: All characters in entity mention are upper case.

LOWER: All characters in entity mention are lower case.

FIRST: The first character in the mention is upper case.

MIDDLE: First character in the mention is lower case and at least one other character is upper case.

Parameters

- **name** (*str*) – The dataset name.
- **query** (*Optional[str]*) – An ElasticSearch query with the `query string syntax`
- **compute_for** (*Union[str, rubrix.metrics.token_classification.metrics.ComputeFor]*) – Metric can be computed for annotations or predictions. Accepted values are Annotations and Predictions. Default to Predictions.

Returns The summary entity capitalness distribution

Return type `rubrix.metrics.models.MetricSummary`

Examples

```
>>> from rubrix.metrics.token_classification import entity_capitalness
>>> summary = entity_capitalness(name="example-dataset")
>>> summary.visualize()
```

`rubrix.metrics.token_classification.metrics.entity_consistency`(*name*, *query=None*, *compute_for=ComputeFor.PREDICTIONS*, *mentions=100*, *threshold=2*)

Computes the consistency for top entity mentions in the dataset.

Entity consistency defines the label variability for a given mention. For example, a mention *first* identified in the whole dataset as *Cardinal*, *Person* and *Time* is less consistent than a mention *Peter* identified as *Person* in the dataset.

Parameters

- **name** (*str*) – The dataset name.
- **query** (*Optional[str]*) – An ElasticSearch query with the [query string syntax](#)
- **compute_for** (*Union[str, rubrix.metrics.token_classification.metrics.ComputeFor]*) – Metric can be computed for annotations or predictions. Accepted values are Annotations and Predictions. Default to Predictions
- **mentions** (*int*) – The number of top mentions to retrieve.
- **threshold** (*int*) – The entity variability threshold (must be greater or equal to 2).

Returns The summary entity capitalness distribution

Examples

```
>>> from rubrix.metrics.token_classification import entity_consistency
>>> summary = entity_consistency(name="example-dataset")
>>> summary.visualize()
```

```
rubrix.metrics.token_classification.metrics.entity_density(name, query=None, compute_for=ComputeFor.PREDICTIONS, interval=0.005)
```

Computes the entity density distribution. Then entity density is calculated at record level for each mention as $\text{mention_length} / \text{tokens_length}$

Parameters

- **name** (*str*) – The dataset name.
- **query** (*Optional[str]*) – An ElasticSearch query with the [query string syntax](#)
- **compute_for** (*Union[str, rubrix.metrics.token_classification.metrics.ComputeFor]*) – Metric can be computed for annotations or predictions. Accepted values are Annotations and Predictions. Default to Predictions.
- **interval** (*float*) – The interval for histogram. The entity density is defined in the range 0-1.

Returns The summary entity density distribution

Return type `rubrix.metrics.models.MetricSummary`

Examples

```
>>> from rubrix.metrics.token_classification import entity_density
>>> summary = entity_density(name="example-dataset")
>>> summary.visualize()
```

```
rubrix.metrics.token_classification.metrics.entity_labels(name, query=None, compute_for=ComputeFor.PREDICTIONS, labels=50)
```

Computes the entity labels distribution

Parameters

- **name** (*str*) – The dataset name.
- **query** (*Optional[str]*) – An ElasticSearch query with the [query string syntax](#)
- **compute_for** (*Union[str, rubrix.metrics.token_classification.metrics.ComputeFor]*) – Metric can be computed for annotations or predictions. Accepted values are Annotations and Predictions. Default to Predictions
- **labels** (*int*) – The number of top entities to retrieve. Lower numbers will be better performers

Returns The summary for entity tags distribution

Return type `rubrix.metrics.models.MetricSummary`

Examples

```
>>> from rubrix.metrics.token_classification import entity_labels
>>> summary = entity_labels(name="example-dataset", labels=20)
>>> summary.visualize() # will plot a bar chart with results
>>> summary.data # The top-20 entity tags
```

```
rubrix.metrics.token_classification.metrics.f1(name, query=None)
```

Computes F1 metrics for a dataset based on entity-level.

Parameters

- **name** (*str*) – The dataset name.
- **query** (*Optional[str]*) – An ElasticSearch query with the [query string syntax](#)

Returns The F1 metric summary containing precision, recall and the F1 score (averaged and per label).

Return type `rubrix.metrics.models.MetricSummary`

Examples

```
>>> from rubrix.metrics.token_classification import fl
>>> summary = fl(name="example-dataset")
>>> summary.visualize() # will plot three bar charts with the results
>>> summary.data # returns the raw result data
```

To display the results as a table:

```
>>> import pandas as pd
>>> pd.DataFrame(summary.data.values(), index=summary.data.keys())
```

`rubrix.metrics.token_classification.metrics.mention_length(name, query=None, level='token', compute_for=ComputeFor.PREDICTIONS, interval=1)`

Computes mentions length distribution (in number of tokens).

Parameters

- **name** (*str*) – The dataset name.
- **query** (*Optional[str]*) – An ElasticSearch query with the `query string` syntax
- **level** (*str*) – The mention length level. Accepted values are “token” and “char”
- **compute_for** (*Union[str, rubrix.metrics.token_classification.metrics.ComputeFor]*) – Metric can be computed for annotations or predictions. Accepted values are Annotations and Predictions. Defaults to Predictions.
- **interval** (*int*) – The bins or bucket for result histogram

Returns The summary for mention token distribution

Return type `rubrix.metrics.models.MetricSummary`

Examples

```
>>> from rubrix.metrics.token_classification import mention_length
>>> summary = mention_length(name="example-dataset", interval=2)
>>> summary.visualize() # will plot a histogram chart with results
>>> summary.data # the raw histogram data with bins of size 2
```

`rubrix.metrics.token_classification.metrics.token_capitalness(name, query=None)`

Computes the token capitalness distribution

UPPER: All characters in the token are upper case.

LOWER: All characters in the token are lower case.

FIRST: The first character in the token is upper case.

MIDDLE: First character in the token is lower case and at least one other character is upper case.

Parameters

- **name** (*str*) – The dataset name.
- **query** (*Optional[str]*) – An ElasticSearch query with the `query string` syntax

Returns The summary for token length distribution

Return type rubrix.metrics.models.MetricSummary

Examples

```
>>> from rubrix.metrics.token_classification import token_capitalness
>>> summary = token_capitalness(name="example-dataset")
>>> summary.visualize() # will plot a histogram with results
>>> summary.data # The token capitalness distribution
```

rubrix.metrics.token_classification.metrics.**token_frequency**(name, query=None, tokens=1000)

Computes the token frequency distribution for a numbe of tokens.

Parameters

- **name** (str) – The dataset name.
- **query** (Optional[str]) – An ElasticSearch query with the query string syntax
- **tokens** (int) – The top-k number of tokens to retrieve

Returns The summary for token frequency distribution

Return type rubrix.metrics.models.MetricSummary

Examples

```
>>> from rubrix.metrics.token_classification import token_frequency
>>> summary = token_frequency(name="example-dataset", token=50)
>>> summary.visualize() # will plot a histogram with results
>>> summary.data # the top-50 tokens frequency
```

rubrix.metrics.token_classification.metrics.**token_length**(name, query=None)

Computes the token size distribution in terms of number of characters

Parameters

- **name** (str) – The dataset name.
- **query** (Optional[str]) – An ElasticSearch query with the query string syntax

Returns The summary for token length distribution

Return type rubrix.metrics.models.MetricSummary

Examples

```
>>> from rubrix.metrics.token_classification import token_length
>>> summary = token_length(name="example-dataset")
>>> summary.visualize() # will plot a histogram with results
>>> summary.data # The token length distribution
```

rubrix.metrics.token_classification.metrics.**tokens_length**(name, query=None, interval=1)

Computes the text length distribution measured in number of tokens.

Parameters

- **name** (str) – The dataset name.

- **query** (*Optional[str]*) – An Elasticsearch query with the `query` string syntax
- **interval** (*int*) – The bins or bucket for result histogram

Returns The summary for token distribution

Return type `rubrix.metrics.models.MetricSummary`

Examples

```
>>> from rubrix.metrics.token_classification import tokens_length
>>> summary = tokens_length(name="example-dataset", interval=5)
>>> summary.visualize() # will plot a histogram with results
>>> summary.data # the raw histogram data with bins of size 5
```

4.21.3 Labeling

The `rubrix.labeling` module aims at providing tools to enhance your labeling workflow.

Text classification

Labeling tools for the text classification task.

class `rubrix.labeling.text_classification.rule.Rule(query, label, name=None, author=None)`

A rule (labeling function) in form of an Elasticsearch query.

Parameters

- **query** (*str*) – An Elasticsearch query with the `query` string syntax.
- **label** (*Union[str, List[str]]*) – The label associated to the query. Can also be a list of labels.
- **name** (*Optional[str]*) – An optional name for the rule to be used as identifier in the `rubrix.labeling.text_classification.WeakLabels` class. By default, we will use the `query` string.
- **author** (*Optional[str]*) –

Examples

```
>>> import rubrix as rb
>>> urgent_rule = Rule(query="inputs.text:(urgent AND immediately)", label="urgent",
↳ name="urgent_rule")
>>> not_urgent_rule = Rule(query="inputs.text:(NOT urgent) AND metadata.title_
↳ length>20", label="not urgent")
>>> not_urgent_rule.apply("my_dataset")
>>> my_dataset_records = rb.load(name="my_dataset")
>>> not_urgent_rule(my_dataset_records[0])
"not urgent"
```


__call__(*record*)

Check if the given record is among the matching ids from the `self.apply` call.

Parameters **record** (`rubrix.client.models.TextClassificationRecord`) – The record to be labelled.

Returns A label or list of labels if the record id is among the matching ids, otherwise `None`.

Raises **RuleNotAppliedError** – If the rule was not applied to the dataset before.

Return type *Optional[Union[str, List[str]]]*

apply(*dataset*)

Apply the rule to a dataset and save matching ids of the records.

Parameters **dataset** (*str*) – The name of the dataset.

property **author**

Who authored the rule.

property **label**: `Union[str, List[str]]`

The rule label

metrics(*dataset*)

Compute the rule metrics for a given dataset:

- **coverage**: Fraction of the records labeled by the rule.
- **annotated_coverage**: Fraction of annotated records labeled by the rule.
- **correct**: Number of records the rule labeled correctly (if annotations are available).
- **incorrect**: Number of records the rule labeled incorrectly (if annotations are available).
- **precision**: Fraction of correct labels given by the rule (if annotations are available). The precision does not penalize the rule for abstains.

Parameters **dataset** (*str*) – Name of the dataset for which to compute the rule metrics.

Returns The rule metrics.

Return type *Dict[str, Union[int, float]]*

property **name**

The name of the rule.

property **query**: `str`

The rule query

`rubrix.labeling.text_classification.rule.load_rules(dataset)`

load the rules defined in a given dataset.

Parameters **dataset** (*str*) – Name of the dataset.

Returns A list of rules defined in the given dataset.

Return type *List[rubrix.labeling.text_classification.rule.Rule]*

class `rubrix.labeling.text_classification.weak_labels.WeakLabels(dataset, rules=None, ids=None, query=None, label2int=None)`

Computes the weak labels of a single-label text classification dataset by applying a given list of rules.

Parameters

- **dataset** (*str*) – Name of the dataset to which the rules will be applied.
- **rules** (*Optional[List[Callable]]*) – A list of rules (labeling functions). They must return a string, or None in case of abstention. If None, we will use the rules of the dataset (Default).
- **ids** (*Optional[List[Union[str, int]]]*) – An optional list of record ids to filter the dataset before applying the rules.
- **query** (*Optional[str]*) – An optional Elasticsearch query with the [query string syntax](#) to filter the dataset before applying the rules.
- **label2int** (*Optional[Dict[Optional[str], int]]*) – An optional dict, mapping the labels to integers. Remember that the return type None means abstention (e.g. {None: -1}). By default, we will build a mapping on the fly when applying the rules.

Raises

- **NoRulesFoundError** – When you do not provide rules, and the dataset has no rules either.
- **DuplicatedRuleNameError** – When you provided multiple rules with the same name.
- **NoRecordsFoundError** – When the filtered dataset is empty.
- **MultiLabelError** – When trying to get weak labels for a multi-label text classification task.
- **MissingLabelError** – When provided with a label2int dict, and a weak label or annotation label is not present in its keys.

Examples

```
>>> # Get the weak label matrix from a dataset with rules:
>>> weak_labels = WeakLabels(dataset="my_dataset")
>>> weak_labels.matrix()
>>> weak_labels.summary()
>>>
>>> # Get the weak label matrix from rules defined in Python:
>>> def awesome_rule(record: TextClassificationRecord) -> str:
...     return "Positive" if "awesome" in record.text else None
>>> another_rule = Rule(query="good OR best", label="Positive")
>>> weak_labels = WeakLabels(dataset="my_dataset", rules=[awesome_rule, another_
↳ rule])
>>> weak_labels.matrix()
>>> weak_labels.summary()
>>>
>>> # Use the WeakLabels object with snorkel's LabelModel:
>>> from snorkel.labeling.model import LabelModel
>>> label_model = LabelModel()
>>> label_model.fit(_train=weak_labels.matrix(has_annotation=False))
>>> label_model.score(l=weak_labels.matrix(has_annotation=True), y=weak_labels.
↳ annotation())
>>> label_model.predict(l=weak_labels.matrix(has_annotation=False))
>>>
>>> # For a builtin integration with Snorkel, see `rubrix.labeling.text_
↳ classification.Snorkel`.
```

annotation(*include_missing=False, exclude_missing_annotations=None*)

Returns the annotation labels as an array of integers.

Parameters

- **include_missing** (*bool*) – If True, returns an array of the length of the record list (`self.records()`). For this, we will fill the array with the `self.label2int[None]` integer for records without an annotation.
- **exclude_missing_annotations** (*Optional[bool]*) – DEPRECATED

Returns The annotation array of integers.

Return type `numpy.ndarray`

property cardinality: int

The number of labels.

change_mapping(label2int)

Allows you to change the mapping between labels and integers.

This will update the `self.matrix` as well as the `self.annotation`.

Parameters **label2int** (*Dict[str, int]*) – New label to integer mapping. Must cover all previous labels.

extend_matrix(thresholds, embeddings=None, gpu=False)

Extends the weak label matrix through embeddings according to the similarity thresholds for each rule.

Implementation based on [Epoxy](#).

Parameters

- **thresholds** (*Union[List[float], numpy.ndarray]*) – An array of thresholds between 0.0 and 1.0, one for each column of the weak labels matrix. Each one stands for the minimum cosine similarity between two sentences for a rule to be extended.
- **embeddings** (*Optional[numpy.ndarray]*) – Embeddings for each row of the weak label matrix. If not provided, we will use the ones from the last `WeakLabels.extend_matrix()` call.
- **gpu** (*bool*) – If True, perform FAISS similarity queries on GPU.

Examples

```
>>> # Choose any model to generate the embeddings.
>>> from sentence_transformers import SentenceTransformer
>>> model = SentenceTransformer('all-mpnet-base-v2', device='cuda')
>>>
>>> # Generate the embeddings and set the thresholds.
>>> weak_labels = WeakLabels(dataset="my_dataset")
>>> embeddings = np.array([ model.encode(rec.text) for rec in weak_labels.
↳ records() ])
>>> thresholds = [0.6] * len(weak_labels.rules)
>>>
>>> # Extend the weak labels matrix.
>>> weak_labels.extend_matrix(thresholds, embeddings)
>>>
>>> # Calling the method below will now retrieve the extended matrix.
>>> weak_labels.matrix()
>>>
```

(continues on next page)

(continued from previous page)

```

>>> # Subsequent calls without the embeddings parameter will reuse the
↳ faiss index built on the first call.
>>> thresholds = [0.75] * len(weak_labels.rules)
>>> weak_labels.extend_matrix(thresholds)
>>> weak_labels.matrix()

```

property int2label: Dict[int, Optional[str]]

The dictionary that maps integers to weak/annotation labels.

property label2int: Dict[Optional[str], int]

The dictionary that maps weak/annotation labels to integers.

property labels: List[str]

The list of labels.

matrix(has_annotation=None)

Returns the weak label matrix, or optionally just a part of it.

Parameters **has_annotation** (Optional[bool]) – If True, return only the part of the matrix that has a corresponding annotation. If False, return only the part of the matrix that has NOT a corresponding annotation. By default, we return the whole weak label matrix.

Returns The weak label matrix, or optionally just a part of it.

Return type numpy.ndarray

show_records(labels=None, rules=None)

Shows records in a pandas DataFrame, optionally filtered by weak labels and non-abstaining rules.

If you provide both labels and rules, we take the intersection of both filters.

Parameters

- **labels** (Optional[List[str]]) – All of these labels are in the record’s weak labels. If None, do not filter by labels.
- **rules** (Optional[List[Union[str, int]]]) – All of these rules did not abstain for the record. If None, do not filter by rules. You can refer to the rules by their (function) name or by their index in the `self.rules` list.

Returns The optionally filtered records as a pandas DataFrame.

Return type pandas.core.frame.DataFrame

summary(normalize_by_coverage=False, annotation=None)

Returns following summary statistics for each rule:

- **label:** Set of unique labels returned by the rule, excluding “None” (abstain).
- **coverage:** Fraction of the records labeled by the rule.
- **annotated_coverage:** Fraction of annotated records labeled by the rule (if annotations are available).
- **overlaps:** Fraction of the records labeled by the rule together with at least one other rule.
- **conflicts:** Fraction of the records where the rule disagrees with at least one other rule.
- **correct:** Number of labels the rule predicted correctly (if annotations are available).
- **incorrect:** Number of labels the rule predicted incorrectly (if annotations are available).

- **precision:** Fraction of correct labels given by the rule (if annotations are available). The precision does not penalize the rule for abstains.

Parameters

- **normalize_by_coverage** (*bool*) – Normalize the overlaps and conflicts by the respective coverage.
- **annotation** (*Optional[numpy.ndarray]*) – An optional array with ints holding the annotations. By default, we will use `self.annotation(include_missing=True)`.

Returns The summary statistics for each rule in a pandas DataFrame.

Return type `pandas.core.frame.DataFrame`

```
class rubrix.labeling.text_classification.weak_labels.WeakMultiLabels(dataset, rules=None,
                                                                    ids=None, query=None)
```

Computes the weak labels of a multi-label text classification dataset by applying a given list of rules.

Parameters

- **dataset** (*str*) – Name of the dataset to which the rules will be applied.
- **rules** (*Optional[List[Callable]]*) – A list of rules (labeling functions). They must return a string, list of strings, or `None` in case of abstention. If `None`, we will use the rules of the dataset (Default).
- **ids** (*Optional[List[Union[str, int]]]*) – An optional list of record ids to filter the dataset before applying the rules.
- **query** (*Optional[str]*) – An optional Elasticsearch query with the [query string syntax](#) to filter the dataset before applying the rules.

Raises

- **NoRulesFoundError** – When you do not provide rules, and the dataset has no rules either.
- **DuplicatedRuleNameError** – When you provided multiple rules with the same name.
- **NoRecordsFoundError** – When the filtered dataset is empty.

Examples

```
>>> # Get the 3 dimensional weak label matrix from a multi-label classification_
↳ dataset with rules:
>>> weak_labels = WeakMultiLabels(dataset="my_dataset")
>>> weak_labels.matrix()
>>> weak_labels.summary()
>>>
>>> # Get the 3 dimensional weak label matrix from rules defined in Python:
>>> def awesome_rule(record: TextClassificationRecord) -> str:
...     return ["Positive", "Slang"] if "next level" in record.text else None
>>> another_rule = Rule(query="amped OR psyched", label=["Positive", "Slang"])
>>> weak_labels = WeakMultiLabels(dataset="my_dataset", rules=[awesome_rule,
↳ another_rule])
>>> weak_labels.matrix()
>>> weak_labels.summary()
```

annotation(*include_missing=False*)

Returns the annotation labels as a matrix of integers.

It has the dimensions (“nr of record” x “nr of labels”). It holds a 1 or 0 to indicate if the record is annotated with the corresponding label. In case there is no annotation for the record, it holds a -1 for each label.

Parameters **include_missing** (*bool*) – If True, returns a matrix of the length of the record list (`self.records()`). For this, we will fill the matrix with -1 for records without an annotation.

Returns The annotation labels as a matrix of integers.

Return type `numpy.ndarray`

property cardinality: int

The number of labels.

extend_matrix(*thresholds, embeddings=None, gpu=False*)

Extends the weak label matrix through embeddings according to the similarity thresholds for each rule.

Implementation based on [Epoxy](#).

Parameters

- **thresholds** (*Union[List[float], numpy.ndarray]*) – An array of thresholds between 0.0 and 1.0, one for each column of the weak labels matrix. Each one stands for the minimum cosine similarity between two sentences for a rule to be extended.
- **embeddings** (*Optional[numpy.ndarray]*) – Embeddings for each row of the weak label matrix. If not provided, we will use the ones from the last `WeakLabels.extend_matrix()` call.
- **gpu** (*bool*) – If True, perform FAISS similarity queries on GPU.

Examples

```
>>> # Choose any model to generate the embeddings.
>>> from sentence_transformers import SentenceTransformer
>>> model = SentenceTransformer('all-mpnet-base-v2', device='cuda')
>>>
>>> # Generate the embeddings and set the thresholds.
>>> weak_labels = WeakMultiLabels(dataset="my_dataset")
>>> embeddings = np.array([ model.encode(rec.text) for rec in weak_labels.
↳ records() ])
>>> thresholds = [0.6] * len(weak_labels.rules)
>>>
>>> # Extend the weak labels matrix.
>>> weak_labels.extend_matrix(thresholds, embeddings)
>>>
>>> # Calling the method below will now retrieve the extended matrix.
>>> weak_labels.matrix()
>>>
>>> # Subsequent calls without the embeddings parameter will reuse the
↳ faiss index built on the first call.
>>> thresholds = [0.75] * len(weak_labels.rules)
>>> weak_labels.extend_matrix(thresholds)
>>> weak_labels.matrix()
```

property labels: `List[str]`

The labels of the multi-label text classification dataset.

matrix(*has_annotation=None*)

Returns the 3 dimensional weak label matrix, or optionally just a part of it.

It has the dimensions (“nr of record” x “nr of rules” x “nr of labels”). It holds a 1 or 0 in case a rule votes for a label or not. If the rule abstains, it holds a -1 for each label.

Parameters *has_annotation* (*Optional[bool]*) – If True, return only the part of the matrix that has a corresponding annotation. If False, return only the part of the matrix that has NOT a corresponding annotation. By default, we return the whole weak label matrix.

Returns The 3 dimensional weak label matrix, or optionally just a part of it.

Return type `numpy.ndarray`

show_records(*labels=None, rules=None*)

Shows records in a pandas DataFrame, optionally filtered by weak labels and non-abstaining rules.

If you provide both *labels* and *rules*, we take the intersection of both filters.

Parameters

- **labels** (*Optional[List[str]]*) – All of these labels are in the record’s weak labels. If None, do not filter by labels.
- **rules** (*Optional[List[Union[str, int]]]*) – All of these rules did not abstain for the record. If None, do not filter by rules. You can refer to the rules by their (function) name or by their index in the `self.rules` list.

Returns The optionally filtered records as a pandas DataFrame.

Return type `pandas.core.frame.DataFrame`

summary(*normalize_by_coverage=False, annotation=None*)

Returns following summary statistics for each rule:

- **label**: Set of unique labels returned by the rule, excluding “None” (abstain).
- **coverage**: Fraction of the records labeled by the rule.
- **annotated_coverage**: Fraction of annotated records labeled by the rule (if annotations are available).
- **overlaps**: Fraction of the records labeled by the rule together with at least one other rule.
- **correct**: Number of labels the rule predicted correctly (if annotations are available).
- **incorrect**: Number of labels the rule predicted incorrectly or missed (if annotations are available).
- **precision**: Fraction of correct labels given by the rule (if annotations are available). The precision does not penalize the rule for abstains.

Parameters

- **normalize_by_coverage** (*bool*) – Normalize the overlaps by the respective coverage.
- **annotation** (*Optional[numpy.ndarray]*) – An optional matrix with ints holding the annotations (see `self.annotation`). By default, we will use `self.annotation(include_missing=True)`.

Returns The summary statistics for each rule in a pandas DataFrame.

Return type `pandas.core.frame.DataFrame`

class rubrix.labeling.text_classification.label_models.**FlyingSquid**(*weak_labels*, ***kwargs*)
 The label model by FlyingSquid.

Note: It is not suited for multi-label classification and does not support it!

Parameters

- **weak_labels** (rubrix.labeling.text_classification.weak_labels.WeakLabels) – A *WeakLabels* object containing the weak labels and records.
- ****kwargs** – Passed on to the init of the FlyingSquid’s *LabelModel*.

Examples

```
>>> from rubrix.labeling.text_classification import WeakLabels
>>> weak_labels = WeakLabels(dataset="my_dataset")
>>> label_model = FlyingSquid(weak_labels)
>>> label_model.fit()
>>> records = label_model.predict()
```

fit(*include_annotated_records=False*, ***kwargs*)

Fits the label model.

Parameters

- **include_annotated_records** (*bool*) – Whether to include annotated records in the fitting.
- ****kwargs** – Passed on to the FlyingSquid’s *LabelModel.fit()* method.

predict(*include_annotated_records=False*, *include_abstentions=False*, *prediction_agent='FlyingSquid'*, *verbose=True*, *tie_break_policy='abstain'*)

Applies the label model.

Parameters

- **include_annotated_records** (*bool*) – Whether to include annotated records.
- **include_abstentions** (*bool*) – Whether to include records in the output, for which the label model abstained.
- **prediction_agent** (*str*) – String used for the *prediction_agent* in the returned records.
- **verbose** (*bool*) – If True, print out messages of the progress to stderr.
- **tie_break_policy** (*Union[rubrix.labeling.text_classification.label_models.TieBreakPolicy, str]*) – Policy to break ties. You can choose among two policies:
 - *abstain*: Do not provide any prediction
 - *random*: randomly choose among tied option using deterministic hash

The last policy can introduce quite a bit of noise, especially when the tie is among many labels, as is the case when all the labeling functions (rules) abstained.

Returns A dataset of records that include the predictions of the label model.

Raises **NotFittedError** – If the label model was still not fitted.

Return type `rubrix.client.datasets.DatasetForTextClassification`

score(*tie_break_policy='abstain', verbose=False, output_str=False*)

Returns some scores/metrics of the label model with respect to the annotated records.

The metrics are:

- accuracy
- micro/macro averages for precision, recall and f1
- precision, recall, f1 and support for each label

For more details about the metrics, check out the [sklearn docs](#).

Note: Metrics are only calculated over non-abstained predictions!

Parameters

- **tie_break_policy** (`Union[rubrix.labeling.text_classification.label_models.TieBreakPolicy, str]`) – Policy to break ties. You can choose among two policies:
 - *abstain*: Do not provide any prediction
 - *random*: randomly choose among tied option using deterministic hash

The last policy can introduce quite a bit of noise, especially when the tie is among many labels, as is the case when all the labeling functions (rules) abstained.

- **verbose** (*bool*) – If True, print out messages of the progress to stderr.
- **output_str** (*bool*) – If True, return output as nicely formatted string.

Returns The scores/metrics in a dictionary or as a nicely formatted str.

Raises

- **NotFittedError** – If the label model was still not fitted.
- **MissingAnnotationError** – If the weak_labels do not contain annotated records.

Return type `Union[Dict[str, float], str]`

class `rubrix.labeling.text_classification.label_models.MajorityVoter`(*weak_labels*)

A basic label model that computes the majority vote across all rules.

For single-label classification, it will predict the label with the most votes. For multi-label classification, it will predict all labels that got at least one vote by the rules.

Parameters **weak_labels** (`Union[rubrix.labeling.text_classification.weak_labels.WeakLabels, rubrix.labeling.text_classification.weak_labels.WeakMultiLabels]`) – The weak labels object.

fit(**args, **kwargs*)

Raises a `NotImplementedError`.

No need to call fit on the MajorityVoter!

predict(*include_annotated_records=False, include_abstentions=False, prediction_agent='MajorityVoter', tie_break_policy='abstain'*)

Applies the label model.

Parameters

- **include_annotated_records** (*bool*) – Whether to include annotated records.
- **include_abstentions** (*bool*) – Whether to include records in the output, for which the label model abstained.
- **prediction_agent** (*str*) – String used for the `prediction_agent` in the returned records.
- **tie_break_policy** (*Union[rubrix.labeling.text_classification.label_models.TieBreakPolicy, str]*) – Policy to break ties (IGNORED FOR MULTI-LABEL!). You can choose among two policies:
 - *abstain*: Do not provide any prediction
 - *random*: randomly choose among tied option using deterministic hash

The last policy can introduce quite a bit of noise, especially when the tie is among many labels, as is the case when all the labeling functions (rules) abstained.

Returns A dataset of records that include the predictions of the label model.

Return type *rubrix.client.datasets.DatasetForTextClassification*

score(*tie_break_policy='abstain', output_str=False*)

Returns some scores/metrics of the label model with respect to the annotated records.

The metrics are:

- accuracy
- micro/macro averages for precision, recall and f1
- precision, recall, f1 and support for each label

For more details about the metrics, check out the [sklearn docs](#).

Note: Metrics are only calculated over non-abstained predictions!

Parameters

- **tie_break_policy** (*Union[rubrix.labeling.text_classification.label_models.TieBreakPolicy, str]*) – Policy to break ties (IGNORED FOR MULTI-LABEL). You can choose among two policies:
 - *abstain*: Do not provide any prediction
 - *random*: randomly choose among tied option using deterministic hash

The last policy can introduce quite a bit of noise, especially when the tie is among many labels, as is the case when all the labeling functions (rules) abstained.
- **output_str** (*bool*) – If True, return output as nicely formatted string.

Returns The scores/metrics in a dictionary or as a nicely formatted str.

Raises **MissingAnnotationError** – If the `weak_labels` do not contain annotated records.

Return type *Union[Dict[str, float], str]*

```
class rubrix.labeling.text_classification.label_models.Snorkel(weak_labels, verbose=True,  
                                                             device='cpu')
```

The label model by [Snorkel](#).

Note: It is not suited for multi-label classification and does not support it!

Parameters

- **weak_labels** (`rubrix.labeling.text_classification.weak_labels.WeakLabels`) – A *WeakLabels* object containing the weak labels and records.
- **verbose** (*bool*) – Whether to show print statements
- **device** (*str*) – What device to place the model on ('cpu' or 'cuda:0', for example). Passed on to the *torch.Tensor.to()* calls.

Examples

```
>>> from rubrix.labeling.text_classification import WeakLabels
>>> weak_labels = WeakLabels(dataset="my_dataset")
>>> label_model = Snorkel(weak_labels)
>>> label_model.fit()
>>> records = label_model.predict()
```

fit(*include_annotated_records=False*, ***kwargs*)

Fits the label model.

Parameters

- **include_annotated_records** (*bool*) – Whether to include annotated records in the fitting.
- ****kwargs** – Additional kwargs are passed on to Snorkel's [fit method](#). They must not contain `L_train`, the label matrix is provided automatically.

predict(*include_annotated_records=False*, *include_abstentions=False*, *prediction_agent='Snorkel'*, *tie_break_policy='abstain'*)

Returns a list of records that contain the predictions of the label model

Parameters

- **include_annotated_records** (*bool*) – Whether to include annotated records.
- **include_abstentions** (*bool*) – Whether to include records in the output, for which the label model abstained.
- **prediction_agent** (*str*) – String used for the *prediction_agent* in the returned records.
- **tie_break_policy** (`Union[rubrix.labeling.text_classification.label_models.TieBreakPolicy, str]`) – Policy to break ties. You can choose among three policies:
 - *abstain*: Do not provide any prediction
 - *random*: randomly choose among tied option using deterministic hash

- *true-random*: randomly choose among the tied options. NOTE: repeated runs may have slightly different results due to differences in broken ties

The last two policies can introduce quite a bit of noise, especially when the tie is among many labels, as is the case when all the labeling functions (rules) abstained.

Returns A dataset of records that include the predictions of the label model.

Return type `rubrix.client.datasets.DatasetForTextClassification`

score(*tie_break_policy*='abstain', *output_str*=False)

Returns some scores/metrics of the label model with respect to the annotated records.

The metrics are:

- accuracy
- micro/macro averages for precision, recall and f1
- precision, recall, f1 and support for each label

For more details about the metrics, check out the [sklearn docs](#).

Note: Metrics are only calculated over non-abstained predictions!

Parameters

- **tie_break_policy** (`Union[rubrix.labeling.text_classification.label_models.TieBreakPolicy, str]`) – Policy to break ties. You can choose among three policies:

- *abstain*: Do not provide any prediction
- *random*: randomly choose among tied option using deterministic hash
- *true-random*: randomly choose among the tied options. NOTE: repeated runs may have slightly different results due to differences in broken ties

The last two policies can introduce quite a bit of noise, especially when the tie is among many labels, as is the case when all the labeling functions (rules) abstained.

- **output_str** (`bool`) – If True, return output as nicely formatted string.

Returns The scores/metrics in a dictionary or as a nicely formatted str.

Raises **MissingAnnotationError** – If the weak_labels do not contain annotated records.

Return type `Union[Dict[str, float], str]`

`rubrix.labeling.text_classification.label_errors.find_label_errors(records, sort_by='likelihood', meta-data_key='label_error_candidate', n_jobs=1, **kwargs)`

Finds potential annotation/label errors in your records using [cleanlab](<https://github.com/cleanlab/cleanlab>).

We will consider all records for which a prediction AND annotation is available. Make sure the predictions were made in a holdout manner, that is you should only include records that were not used in the training of the predictor.

Parameters

- **records** (*Union[List[rubrix.client.models.TextClassificationRecord], rubrix.client.datasets.DatasetForTextClassification]*) – A list of text classification records
- **sort_by** (*Union[str, rubrix.labeling.text_classification.label_errors.SortBy]*) – One of the three options - “likelihood”: sort the returned records by likelihood of containing a label error (most likely first) - “prediction”: sort the returned records by the probability of the prediction (highest probability first) - “none”: do not sort the returned records
- **metadata_key** (*str*) – The key added to the record’s metadata that holds the order, if `sort_by` is not “none”.
- **n_jobs** (*Optional[int]*) – Number of processing threads used by multiprocessing. If None, uses the number of threads on your CPU. Defaults to 1, which removes parallel processing.
- ****kwargs** – Passed on to `cleanlab.pruning.get_noise_indices` (cleanlab < 2.0) or `cleanlab.filter.find_label_issues` (cleanlab >= 2.0)

Returns A list of records containing potential annotation/label errors

Raises

- **NoRecordsError** – If none of the records has a prediction AND annotation.
- **MissingPredictionError** – If a prediction is missing for one of the labels.
- **ValueError** – If not supported kwargs are passed on, e.g. ‘sorted_index_method’.

Return type *List[rubrix.client.models.TextClassificationRecord]*

Examples

```
>>> import rubrix as rb
>>> records = rb.load("my_dataset")
>>> records_with_label_errors = find_label_errors(records)
```

4.21.4 Listeners

Here we describe the Rubrix listeners capabilities

class `rubrix.listeners.Metrics(*args, **kwargs)`

Metrics results for a single listener execution.

The metrics object exposes the metrics configured for the listener as property values. For example, if you define a listener including the metric “F1”, the results will be accessible as `metrics.F1`

class `rubrix.listeners.RBDatasetListener(dataset, action, metrics=None, query=None, query_params=None, condition=None, query_records=True, interval_in_seconds=30)`

The Rubrix dataset listener class

Parameters

- **dataset** (*str*) – The dataset over which listener is created

- **action** (*Union[Callable[[List[Union[rubrix.client.models.TextClassificationRecord, rubrix.client.models.TokenClassificationRecord, rubrix.client.models.Text2TextRecord]], rubrix.listeners.models.RBListenerContext], bool], Callable[[rubrix.listeners.models.RBListenerContext], bool]]*) – The action to execute when condition is satisfied
- **metrics** (*Optional[List[str]]*) – A list of metrics ids that will be required in condition
- **query** (*Optional[str]*) – The query string to apply
- **query_params** (*Optional[Dict[str, Any]]*) – Defined parameters used dynamically in the provided query
- **condition** (*Optional[Callable[[rubrix.listeners.models.Search, Optional[rubrix.listeners.models.RBListenerContext]], bool]]*) – The condition to satisfy to execute the action
- **query_records** (*bool*) – If False, the records won't be passed as argument to the action. Default: True
- **interval_in_seconds** (*int*) – How often the listener is executed. Default to 30 seconds

Return type None

property formatted_query: `Optional[str]`

Formatted query using defined query params, if any

is_running()

True if listener is running

start(**action_args*, ***action_kwargs*)

Start listen to changes in the dataset. Additionally, args and kwargs can be passed to action by using the *action_** arguments

If the listener is already started, a `ValueError` will be raised

stop()

Stops listener if it's still running.

If listener is already stopped, a `ValueError` will be raised

class `rubrix.listeners.RBListenerContext`(*listener*, *search=None*, *metrics=None*, *query_params=None*)

The Rubrix listener execution context. This class keeps the context components related to a listener

Parameters

- **listener** (`RBDatasetListener`) – The rubrix listener instance
- **search** (*Optional[rubrix.listeners.models.Search]*) – Search results for current execution
- **metrics** (*Optional[rubrix.listeners.models.Metrics]*) – Metrics results for current execution
- **query_params** (*Optional[Dict[str, Any]]*) – Dynamic parameters used in the listener query

Return type None

property dataset: `str`

Computed property that returns the configured listener dataset name

property query: `Optional[str]`

Computed property that returns the configured listener query string

class `rubrix.listeners.Search`(*total*, *query_params=None*)

Search results for a single listener execution

Parameters

- **total** (*int*) – The total number of records affected by the listener query
- **query_params** (*Optional[Dict[str, Any]]*) – The query parameters applied to the executed search

Return type `None`

`rubrix.listeners.listener`(*dataset*, *query=None*, *metrics=None*, *condition=None*, *with_records=True*, *execution_interval_in_seconds=30*, ***query_params*)

Configures the decorated function as a Rubrix listener.

Parameters

- **dataset** (*str*) – The dataset name.
- **query** (*Optional[str]*) – The query string.
- **metrics** (*Optional[List[str]]*) – Required metrics for listener condition.
- **condition** (*Optional[Callable[[rubrix.listeners.models.Search, Optional[rubrix.listeners.models.RBListenerContext]], bool]]*) – Defines condition over search and metrics that launch action when is satisfied.
- **with_records** (*bool*) – Include records as part or action arguments. If False, only the listener context `RBListenerContext` will be passed. Default: True.
- **execution_interval_in_seconds** (*int*) – Define the execution interval in seconds when listener iteration will be executed.
- ****query_params** – Dynamic parameters used in the query. These parameters will be available via the listener context and can be updated for subsequent queries.

4.22 Web App UI

Here we provide a comprehensible overview of the Rubrix web app's User Interface (UI). To launch the web app, please have a look at our [setup and installation guide](#).

4.22.1 Pages

- **Home page**: Search, access and share your datasets in a unified place
- **Dataset**: Dive into your dataset to explore and annotate your records

Home page

Name ↑↓	Workspace ▾	Task ▾	Tags ↑↓	Created at ↑↓	Updated at ↑↓		
med7_trf_datashift <input type="checkbox"/>	recognai	TokenClassification		a day ago	a day ago		
med7_trf_test <input type="checkbox"/>	recognai	TokenClassification		a day ago	a day ago		
labeling_with_pretrained_2test <input type="checkbox"/>	recognai	TextClassification		a month ago	a day ago		
clean_imdb <input type="checkbox"/>	recognai	TokenClassification		a month ago	4 days ago		
go_emotion_multi_label <input type="checkbox"/>	recognai	TextClassification		20 days ago	4 days ago		
ner-without-labels <input type="checkbox"/>	recognai	TokenClassification		7 days ago	7 days ago		
veganuary_v2 <input type="checkbox"/>	recognai	TokenClassification		11 days ago	11 days ago		

The *Home page* is mainly a **filterable, searchable and sortable list** of **datasets**. It is the **entry point** to the Rubrix web app and is composed of the following three components.

Search bar

The “*Search datasets*” bar on the top allows you to search for a specific dataset by its name.

Dataset list

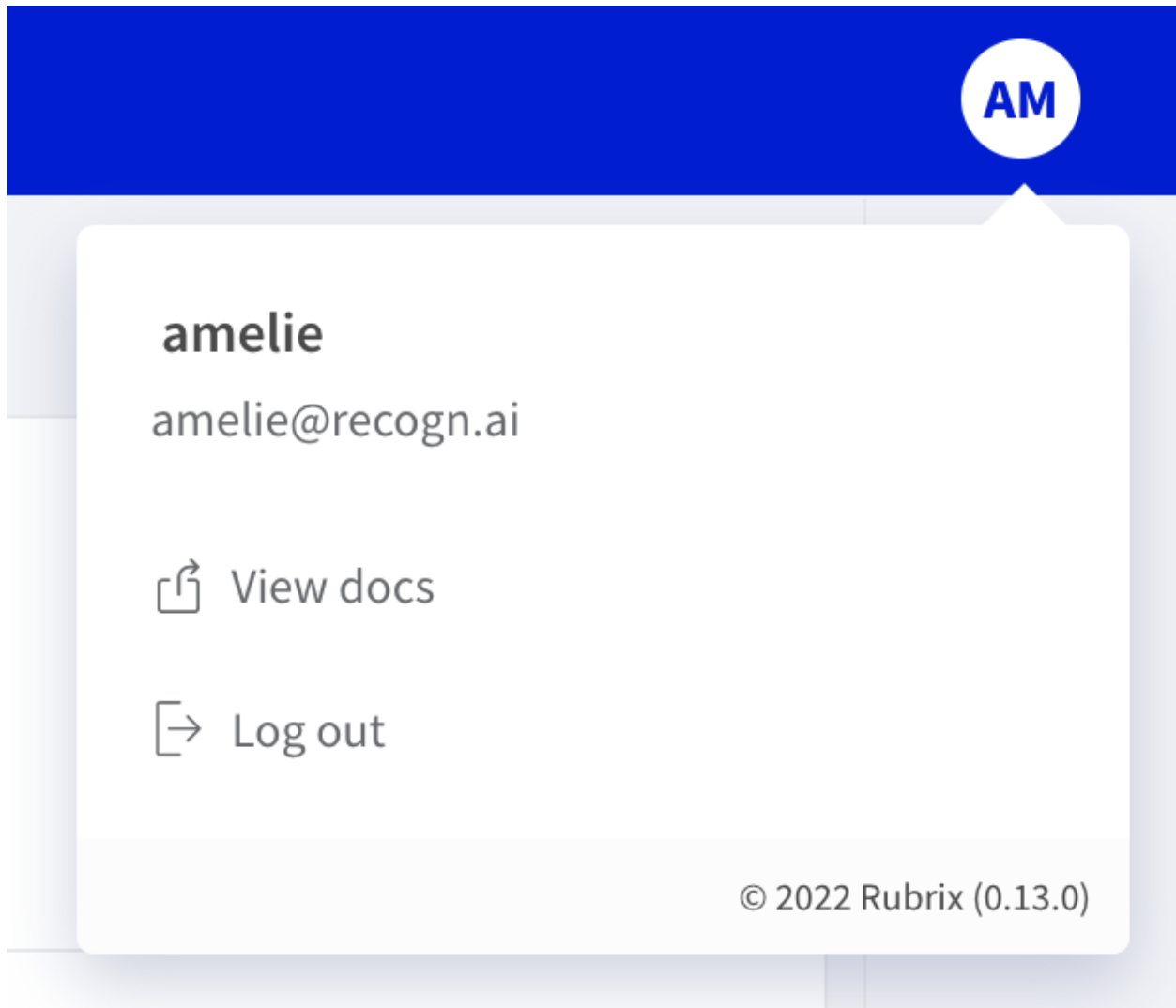
In the center of the page you see the list of datasets available for **your user account**. The list consists of following columns:

- **Name:** The name of the dataset, can be sorted alphabetically.
- **Workspace:** A filterable column showing the workspace to which the dataset belongs.
- **Task:** The *task* of the dataset. This is a filterable column.
- **Tags:** User defined tags for the dataset.
- **Created at:** When was the dataset first logged by the client.
- **Updated at:** When was the dataset last modified, either via the Rubrix web app or the client.

Side bar

You can find a user icon and a refresh button on the top right:

- **User icon:** This icon shows the initials of your username and allows you to **view the documentation**, view your **current Rubrix version**, and **log out**.
- **Refresh:** This button updates the list of datasets in case you just logged new data from the client.



Dataset

The screenshot shows the Rubrix Dataset page for the dataset 'conscience-concepts'. The interface includes a search bar, tabs for Predictions, Annotations, and Status (1), and a 'Records (151)' indicator. A filter bar at the top shows categories: CARBS [1], DAIRY [2], FRUIT [3], HERBS [4], MEAT [5], and VEGETABLE [6]. Below this, there are buttons for 'Validate' and 'Discard', with a note that actions will apply to the 1 record selected. The main content area displays a text record with various words highlighted and annotated. For example, 'Peanut Butter' is annotated as 'DAIRY', 'Banana' as 'FRUIT', and 'Spinach' as 'VEGETABLE'. There are also buttons for 'Save' and 'Clear annotations'. On the right side, there is a 'Stats' sidebar showing 'Mentions' for different categories: DAIRY (cheese, eggs, cream cheese), FRUIT (berries, Berry, Strawberry), and HERBS (Pork Picadillo, cumin, fennel). Each category has a count of 0. At the bottom, there is a pagination bar showing 'Records per page: 5' and a range from 1 to 31, with the current page being 11-15 of 151.

The *Dataset* page is the main page of the Rubrix web app. From here you can access most of Rubrix’s features, like **exploring and annotating** the records of your dataset.

The page is composed of 4 major components:

Search bar

The screenshot shows the Rubrix Search bar interface. At the top, there is a search bar with the query 'predicted_as:(NOT person)'. Below the search bar, there are tabs for Predictions, Annotations, Status, and Sort. A filter bar at the bottom shows categories: corporation [1], creative-work [2], group [3], location [4], person [5], and product [6]. The main content area is currently empty, showing only the search bar and filter bar.

Rubrix’s *search bar* is a powerful tool that allows you to thoroughly explore your dataset, and quickly navigate through the records. You can either fuzzy search the contents of your records, or use the more advanced *query string syntax* of Elasticsearch to take full advantage of Rubrix’s *data models*. You can find more information about how to use the search bar in our detailed *search guide*.

Filters



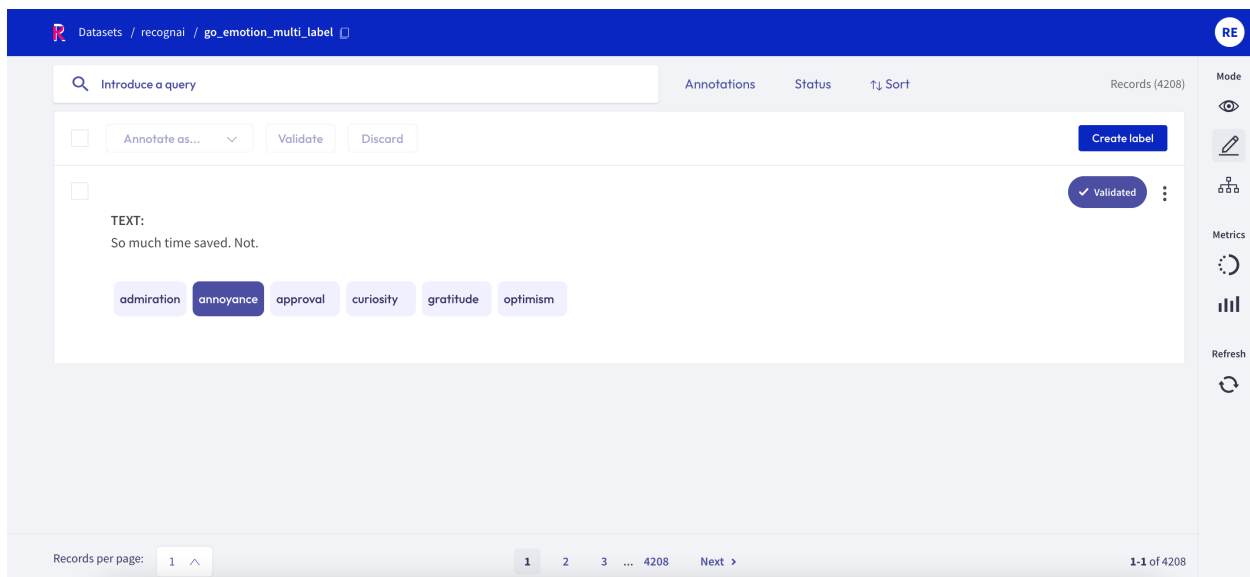
The *filters* provide you a quick and intuitive way to filter and sort your records with respect to various parameters. You can find more information about how to use the filters in our detailed [filter guide](#).

Record cards

The record cards are at the heart of the *Dataset* page and contain your data. There are three different flavors of record cards depending on the *task* of your dataset. All of them share the same basic structure showing the input text and a vertical ellipsis (or “kebab menu”) on the top right that lets you access the record’s metadata. Predictions and annotations are shown depending on the current *mode* and *task* of the dataset.

Check out our [exploration](#) and [annotation](#) guides to see how the record cards work in the different *modes*.

Text classification



In this task the predictions are given as tags below the input text. They contain the label as well as a percentage score. When in *Explore mode* annotations are shown as tags on the right together with a symbol indicating if the predictions match the annotations or not. When in *Annotate mode* predictions and annotations share the same labels (annotation labels are darker).

A text classification dataset can support either single-label or multi-label classification - in other words, records are either annotated with one single label or various.

Token classification

RE

Introduce a query

Predictions Annotations Status Sort

Records (185)

Mode

Metrics

Refresh

Records per page: 5

< Prev 1 2 3 4 ... 37 Next >

6-10 of 185

Layers of cream cheese, berries, and buttery pretzels add up to the Strawberry Pretzel Salad recipe that is loved by adults and kids alike. The pretzel crust adds a salty-sweet layer to this dessert that makes it irresistible. Strawberry. Pretzel. Salad. It's not really a salad and it sure isn't my mama's typical Sunday dinner...

The post Strawberry Pretzel Salad appeared first on Barefeet in the Kitchen.

Of the many ways you can utilize ground pork, pork picadillo is one of the easiest—and even, perhaps, most delicious. After all, few dishes manage to evoke the same comfort and vibrance the way this dish does! With its uniquely Latin American and Filipino roots, pork picadillo breathes life into every kitchen with its assortment...

Read On →

The post Pork Picadillo appeared first on Panlasang Pinoy.

In this task annotations are given as colored highlights in the input text, while predictions are indicated by underlines. At the top of the record list you will find a legend that connects the colors to the respective labels. When in *Annotate mode* you can remove annotations or add new ones by simply selecting the desired text.

Hint: When using the *score filter*, the predictions that do **not** fall in the selected range will be missing the solid thin line.

Text2Text

RE

Introduce a query

Predictions Annotations Status Metadata Sort

Records (25)

Mode

Metrics

Refresh

Records per page: 1

1 2 3 ... 25 Next >

1-1 of 25

Resumption of the session

Prediction

Reanudación del período de sesiones

Score: 58,714 %

1 of 1 predictions

View annotation

In this task predictions and the annotation are given in a text field below the input text. You can switch between prediction and annotation via the “*View annotation*”/“*View predictions*” buttons. For the predictions you can find an associated score in the lower left corner. If you have multiple predictions you can toggle between them using the arrows on the button of the record card.

Sidebar

The right sidebar is divided into three sections.

Modes

This section of the sidebar lets you switch between the different Rubrix modes that are covered extensively in their respective guides:

- **Explore:** this mode is for *exploring your dataset* and gain valuable insights
- **Annotate:** this mode lets you conveniently *annotate your data*
- **Define rules:** this mode helps you to *define rules* to automatically label your data

Note: Not all modes are available for all *tasks*.

Metrics

In this section you find several “metrics” that can provide valuable insights to your dataset. They also provide some support while annotating your records, or defining heuristic rules. There are three different kind of *metrics*:

- **Progress:** see metrics of your annotation process, like its progress and the label distribution (only visible in the *Explore* and *Annotate* mode)
- **Overall rule metrics:** see aggregated metrics about your defined rules (only visible in the *Define rules* mode)
- **Stats:** check the keywords of your dataset (text classification, text2text) or the mentions of your annotations and predictions (token classification)

You can find more information about each metric in our dedicated *metrics guide*.

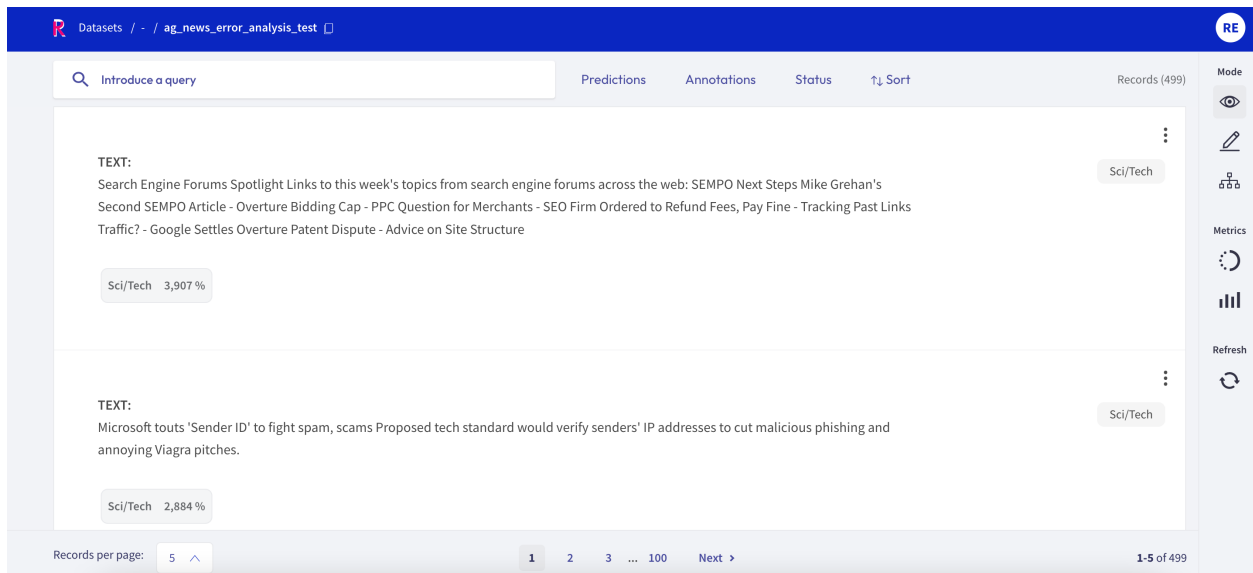
Refresh

This button allows you to refresh the list of the record cards with respect to the activated filters. For example, if you are annotating and use the *Status filter* to filter out annotated records, you can press the *Refresh* button to hide the latest annotated records.

4.22.2 Features

- *Explore records*: Explore your data and predictions, as well as your annotations
- *Annotate records*: Annotate your records for different *tasks*
- *Define rules*: Define rules to weakly supervise your data
- *Search records*: Search your records with the powerful elasticsearch query syntax
- *Filter records*: Quickly filter your records by predictions, annotations or their metadata
- *View dataset metrics*: View insightful metrics of your dataset

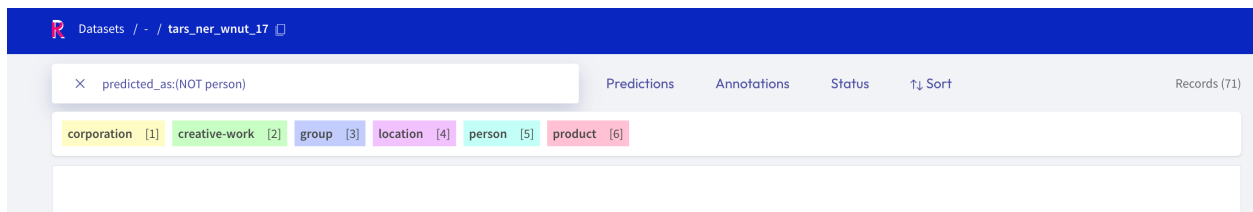
Explore records



If you want to explore your dataset or analyze the predictions of a model, the Rubrix web app offers a dedicated Explore mode. The powerful search functionality and intuitive filters allow you to quickly navigate through your records and dive deep into your dataset. At the same time, you can view the predictions and compare them to gold annotations.

You can access the *Explore mode* via the sidebar of the *Dataset page*.

Search and filter



The powerful search bar allows you to do simple, quick searches, as well as complex queries that take full advantage of Rubrix's *data models*. In addition, the *filters* provide you a quick and intuitive way to filter and sort your records with respect to various parameters, including predictions and annotations. Both of the components can be used together to dissect in-depth your dataset, validate hunches, and find specific records.

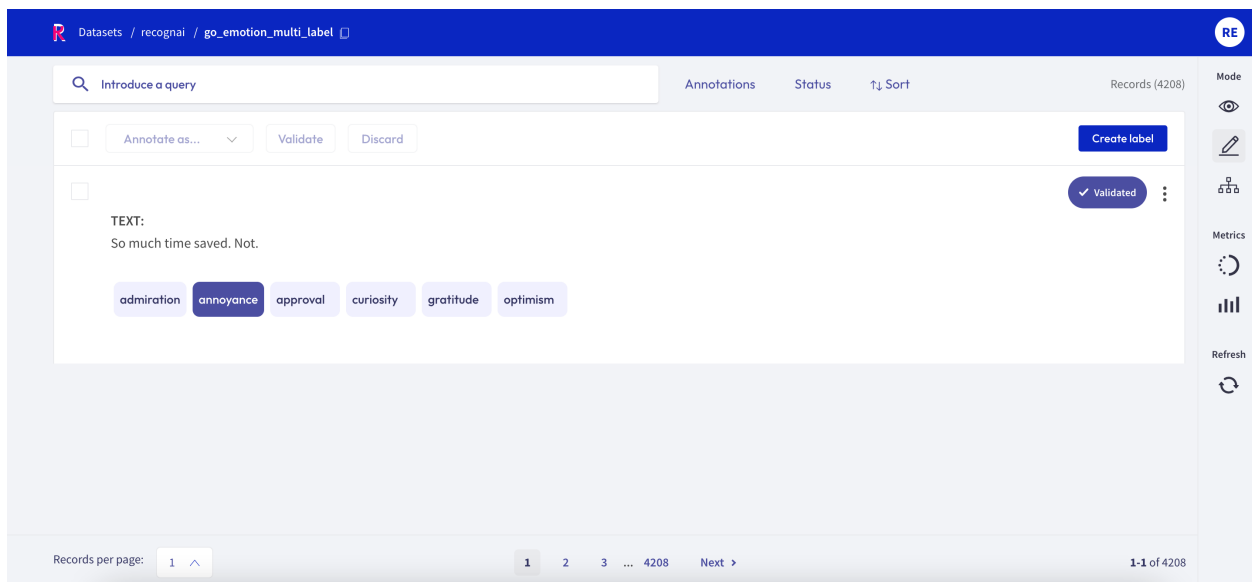
You can find more information about how to use the search bar and the filters in our detailed [search guide](#) and [filter guide](#).

Note: Not all filters are available for all [tasks](#).

Predictions and annotations

Predictions and annotations are an integral part of Rubrix's [data models](#). The way they are presented in the Rubrix web app depends on the [task](#) of the dataset.

Text classification



In this task the predictions are given as tags below the input text. They contain the label as well as a percentage score. Annotations are shown as tags on the right together with a symbol indicating if the predictions match the annotations or not.

Token classification

Datasets / recognal / concise-concepts

Introduce a query

Predictions Annotations Status Sort

Records (185)

Mode

Metrics

Refresh

Records per page: 5

< Prev 1 2 3 4 ... 37 Next >

6-10 of 185

In this task, predictions and annotations are displayed as highlights in the input text. To easily identify them at a glance, **annotations** are highlighted with the color of their corresponding label, while **predictions** are underlined with a solid line (see picture).

For datasets with available score, the solid line for **predictions** disappears when the **score filter** (in **Predictions filter** section) is applied.

Text2Text

Datasets / david / text2text_translation

Introduce a query

Predictions Annotations Status Metadata Sort

Records (25)

Mode

Metrics

Refresh

Records per page: 1

1 2 3 ... 25 Next >

1-1 of 25

In this task predictions and the annotation are given in a text field below the input text. You can switch between prediction and annotation via the “*View annotation*”/“*View predictions*” buttons. For the predictions you can find an

associated score in the lower left corner. If you have multiple predictions you can toggle between them using the arrows on the button of the record card.

Metrics

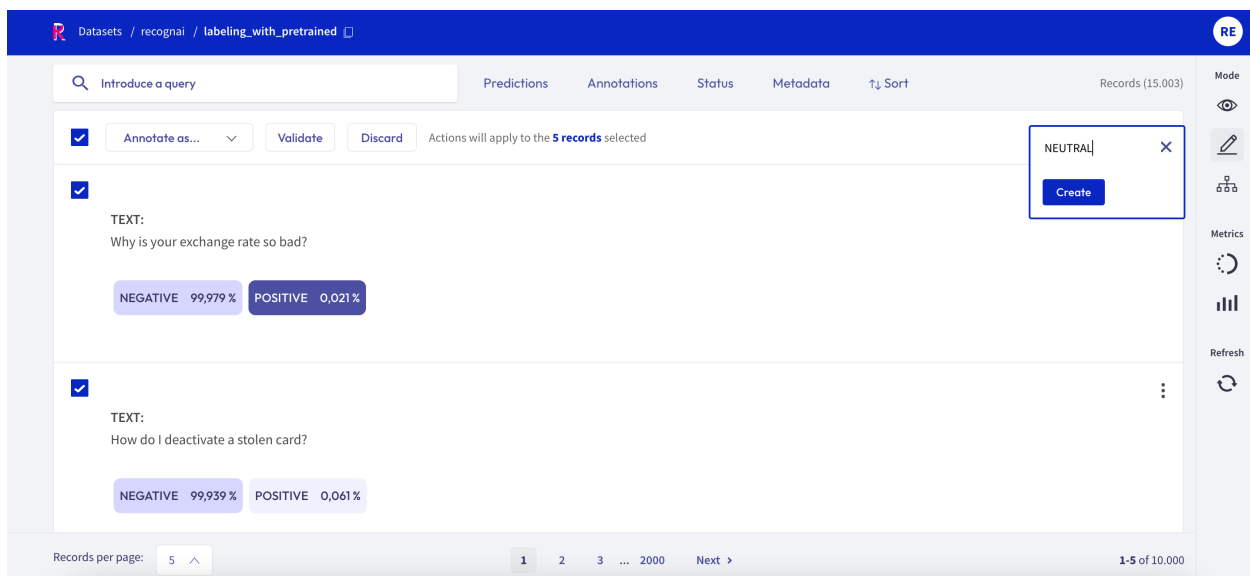
From the side bar you can access the *Stats metrics* that provide support for your analysis of the dataset.

Annotate records

The Rubrix web app has a dedicated mode to quickly label your data in a very intuitive way, or revise previous gold labels and correct them. Rubrix’s powerful search and filter functionalities, together with potential model predictions, can guide the annotation process and support the annotator.

You can access the *Annotate mode* via the sidebar of the *Dataset page*.

Create labels



For the text and token classification tasks, you can create new labels within the *Annotate mode*. On the right side of the bulk validation bar, you will find a “+ Create new label” button that lets you add new labels to your dataset.

Annotate

To annotate the records, the Rubrix web app provides a simple and intuitive interface that tries to follow the same interaction pattern as in the *Explore mode*. As in the *Explore mode*, the record cards in the *Annotate mode* are also customized depending on the *task* of the dataset.

Text Classification

When switching in the *Annotate mode* for a text classification dataset, the labels in the record cards become clickable and you can annotate the records by simply clicking on them. For multi-label classification tasks, you can also annotate a record with no labels by either validating an empty selection or deselecting all labels.

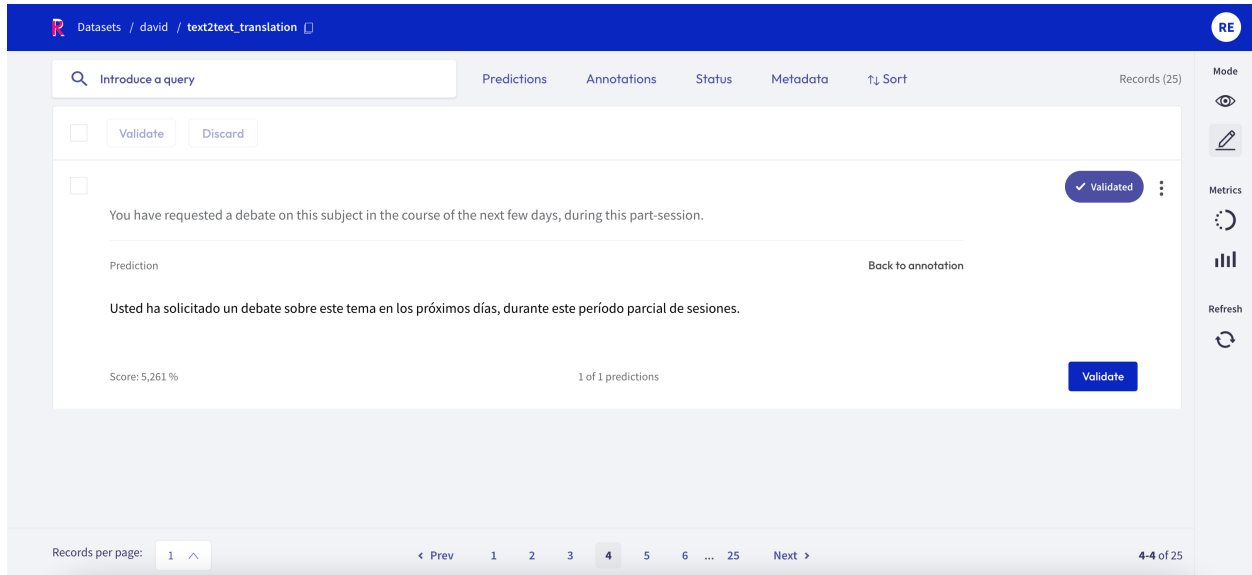
Once a record is annotated, it will be marked as *Validated* in the upper right corner of the record card.

Token Classification

For token classification datasets, you can highlight words (tokens) in the text and annotate them with a label. Under the hood, the highlighting takes advantage of the *tokens* information in the *Token Classification data model*. You can also remove annotations by hovering over the highlights and pressing the *X* button.

After modifying a record, either by adding or removing annotations, its status will change to *Pending* and a *Save* button will appear. Once a record is saved, its status will change to *Validated*.

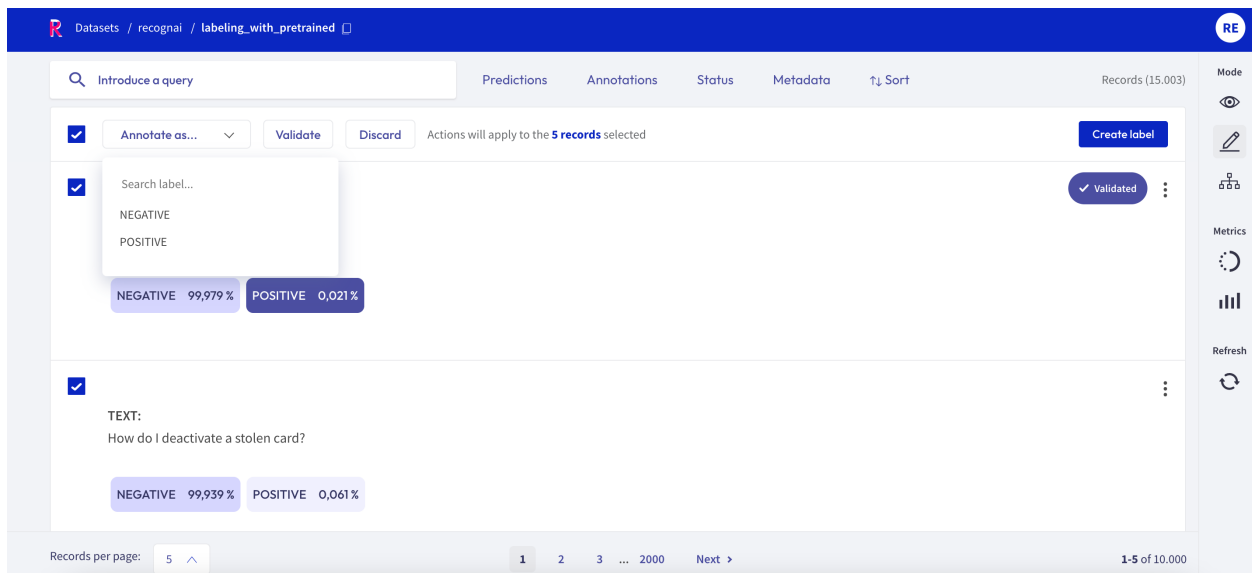
Text2Text



The screenshot shows the Rubrix interface for a Text2Text dataset. The top navigation bar includes 'Datasets / david / text2text_translation'. The main content area displays a record with a prediction. The prediction text is: 'You have requested a debate on this subject in the course of the next few days, during this part-session.' Below the prediction, there is a 'Back to annotation' link. The record is marked as 'Validated' with a green checkmark. At the bottom, there is a 'Score: 5,261 %' and '1 of 1 predictions'. A 'Validate' button is visible in the bottom right corner of the record card. The bottom of the interface shows a pagination bar with 'Records per page: 1' and a range of '4-4 of 25'.

For text2text datasets, you have a text box available, in which you can draft or edit an annotation. After editing or drafting your annotation, don't forget to save your changes.

Bulk annotate



The screenshot shows the Rubrix interface for a dataset named 'recognai / labeling_with_pretrained'. The main content area displays a record with a prediction. The prediction text is: 'How do I deactivate a stolen card?'. Below the prediction, there is a 'Back to annotation' link. The record is marked as 'Validated' with a green checkmark. At the bottom, there is a 'Score: 5,261 %' and '1 of 1 predictions'. A 'Validate' button is visible in the bottom right corner of the record card. The bottom of the interface shows a pagination bar with 'Records per page: 5' and a range of '1-5 of 10.000'.

For all *tasks*, you can **bulk validate** the predictions of the records. You can either select the records one by one with the selection box on the upper left of each card, or you can use the global selection box below the search bar, which will select all records shown on the page. Then you can either *Validate* or *Discard* the selected records.

For the text classification task, you can additionally **bulk annotate** the selected records with a specific label, by simply selecting the label from the “*Annotate as ...*” list.

Validate predictions

In Rubrix you can pre-annotate your data by including model predictions in your records. Assuming that the model works reasonably well on your dataset, you can filter for records with high prediction scores, and simply *validate* their predictions to quickly annotate records.

Text Classification

For this task, model predictions are shown as percentages in the label tags. You can validate the predictions shown in a slightly darker tone by pressing the *Validate* button:

- for a **single label** classification task, this will be the prediction with the highest percentage
- for a **multi label** classification task, this will be the predictions with a percentage above 50%

Token Classification

For this task, predictions are shown as underlines. You can also validate the predictions (or the absence of them) by pressing the *Validate* button.

Once the record is saved or validated, its status will change to *Validated*.

Text2Text

You can validate or edit a prediction, by first clicking on the *view predictions* button, and then the *Edit* or *Validate* button. After editing or drafting your annotation, don’t forget to save your changes.

Search and filter



The powerful search bar allows you to do simple, quick searches, as well as complex queries that take full advantage of Rubrix’s *data models*. In addition, the *filters* provide you a quick and intuitive way to filter and sort your records with respect to various parameters, including the metadata of your records. For example, you can use the **Status filter** to hide already annotated records (*Status: Default*), or to only show annotated records when revising previous annotations (*Status: Validated*).

You can find more information about how to use the search bar and the filters in our detailed *search guide* and *filter guide*.

Note: Not all filters are available for all *tasks*.

Progress metric

From the sidebar you can access the *Progress metrics*. There you will find the progress of your annotation session, the distribution of validated and discarded records, and the label distribution of your annotations.

You can find more information about the metrics in our dedicated [metrics guide](#).

Define rules

The screenshot shows the 'Define rules' interface in the Rubrix web app. The top navigation bar indicates the current dataset is 'go_emotion_multi_label'. The main area is divided into several sections:

- Annotations:** A search bar with 'I appreciate' and a list of emotion labels: admiration, annoyance, approval, curiosity, gratitude, and optimism. A 'Save rule' button is at the bottom.
- Rule Metrics:** A blue box showing performance metrics for the current rule:

Metric	Value
Coverage	1.331% (56/4,208)
Annotated coverage	2.639% (10/379)
Precision	40.00%
Correct/incorrect	8/12
- Overall rule metrics:** A sidebar on the right showing aggregated metrics for all rules:

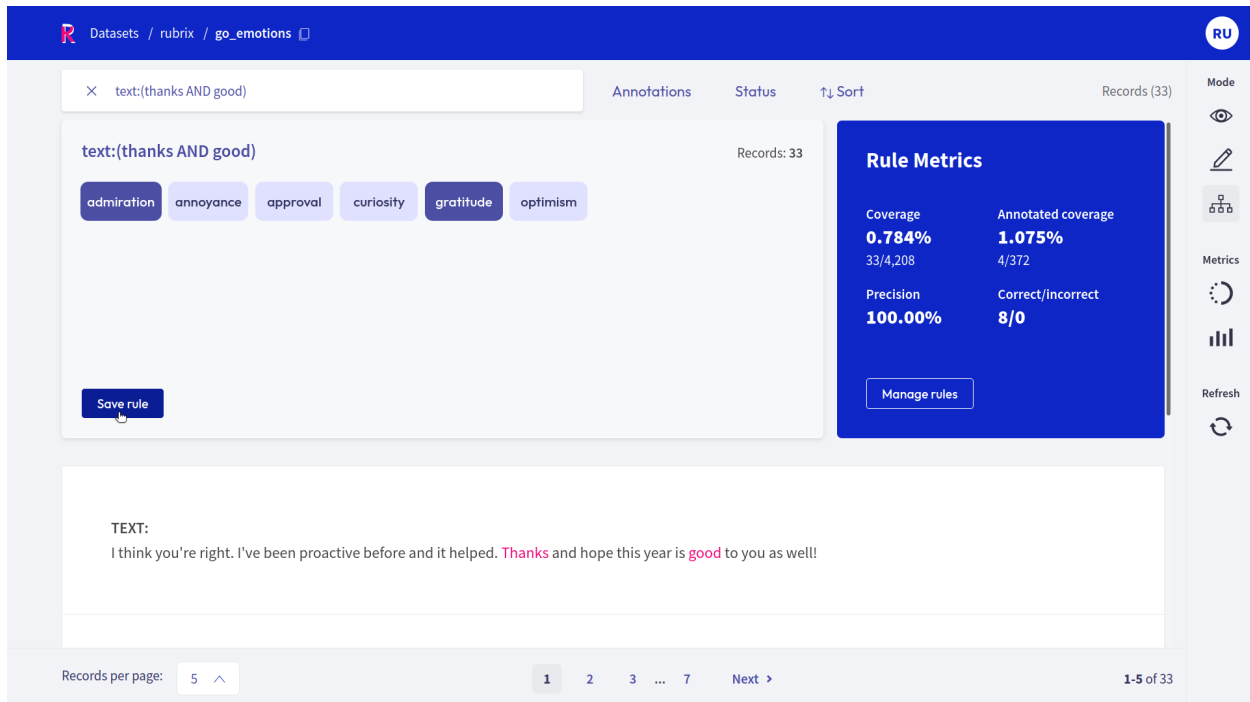
Metric	Value
Coverage	32.01%
Annotated Coverage	34.037%
Precision average	71.493%
Correct/Incorrect	158/63
Total rules	11
gratitude	4
optimism	4
approval	4
admiration	3
curiosity	2
annoyance	2
- Text Examples:** Two text snippets are shown with their predicted labels:
 - TEXT: 'I really appreciate you saying that And yes he is the cutest' with label 'annoyance'.
 - TEXT: (empty) with label 'curiosity'.
- Footer:** A pagination bar showing 'Records per page: 5' and '1-5 of 56'.

The Rubrix web app has a dedicated mode to find good **heuristic rules**, also often referred to as *labeling functions*, for a **weak supervision** workflow. As shown in our [guide](#) and [tutorial](#), these rules allow you to quickly annotate your data with noisy labels in a semiautomatic way.

You can access the *Define rules* mode via the sidebar of the [Dataset page](#).

Note: The *Define rules* mode is only available for text classification datasets.

Query plus labels



The screenshot shows the Rubrix interface for creating a rule. The top bar indicates the dataset is 'rubrix / go_emotions'. The search bar contains the query 'text:(thanks AND good)'. Below the search bar, a list of labels is shown: admiration, annoyance, approval, curiosity, gratitude, and optimism. The 'gratitude' label is selected. The 'Save rule' button is visible. On the right, the 'Rule Metrics' panel displays the following data:

Rule Metrics	
Coverage	Annotated coverage
0.784%	1.075%
33/4,208	4/372
Precision	Correct/incorrect
100.00%	8/0

The 'Manage rules' button is located at the bottom of the metrics panel. Below the rule configuration, the 'TEXT:' section shows a sample record: 'I think you're right. I've been proactive before and it helped. Thanks and hope this year is good to you as well!'. The bottom of the interface shows the 'Records per page' set to 5 and a pagination bar with '1 2 3 ... 7 Next >' and '1-5 of 33'.

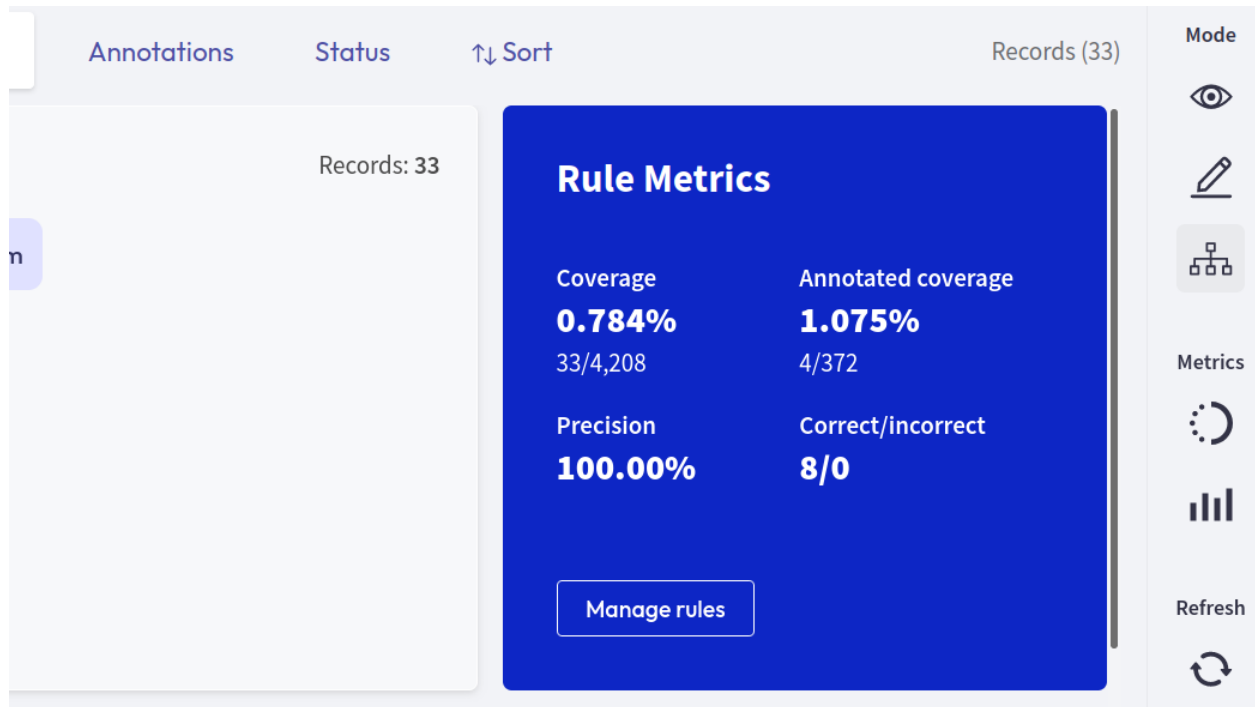
A rule in Rubrix basically applies a chosen set of labels to a list of records that match a given *query*, so all you need is a query plus labels. After entering a query in the search bar and selecting one or multiple labels, you will see some *metrics* for the rule on the right and the matches of your query in the record list below.

Warning: Filters are not part of the rule, but are applied to the record list. This means, if you have filters set, the record list does not necessarily correspond to the records affected by the rule.

If you are happy with the metrics and/or the matching record list, you can save the rule by clicking on “Save rule”. In this way it will be stored as part of the current dataset and can be accessed via the *manage rules* button.

Hint: If you want to add labels to the available list of labels, you can switch to the *Annotation mode* and create labels there.

Rule Metrics

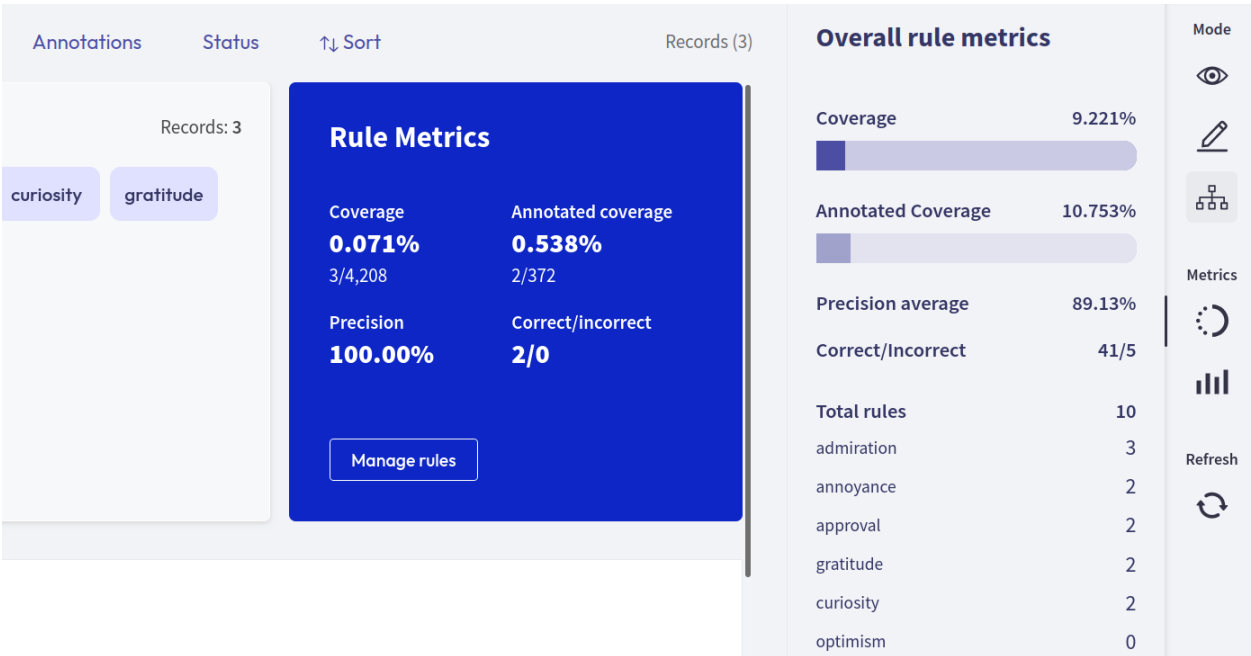


After entering a query and selecting labels, Rubrix provides you with some key metrics about the rule. Some metrics are only available if your dataset has also annotated records.

- **Coverage:** Percentage of records labeled by the rule.
- **Annotated coverage:** Percentage of annotated records labeled by the rule.
- **Correct/incorrect:** Number of labels the rule predicted correctly/incorrectly with respect to the annotations.
- **Precision:** Percentage of correct labels given by the rule with respect to the annotations.

Note: For multi-label classification tasks, we only count wrongly predicted labels as incorrect, not labels that the rule misses.

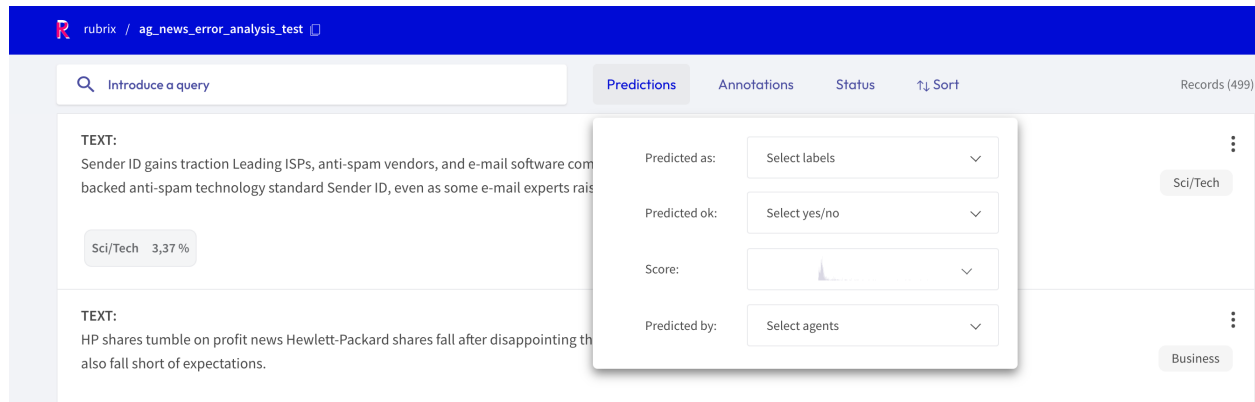
Overall rule metrics



From the *right sidebar* you can access the **Overall rule metrics**. Here you will find the aggregated metrics, such as the coverages, the average precision and the total number of correctly/incorrectly predicted labels. You can also find an overview about how many rules you saved and how they are distributed with respect to their labels.

Hint: If you struggle to increase the overall coverage, try to filter for the records that are not covered by your rules via the *Annotation filter*.

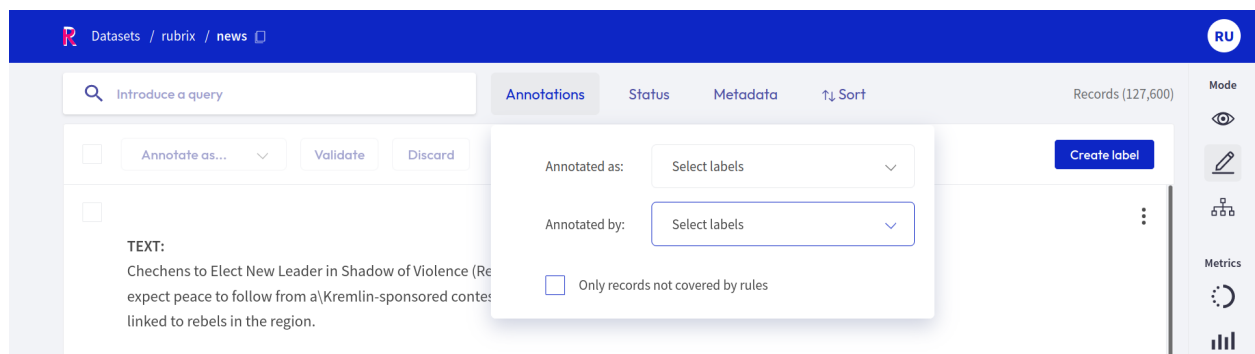
Predictions filter



This filter allows you to filter records with respect of their predictions:

- **Predicted as:** filter records by their predicted labels
- **Predicted ok:** filter records whose predictions do, or do not, match the annotations
- **Score:** filter records with respect to the score of their prediction
- **Predicted by:** filter records by the *prediction agent*

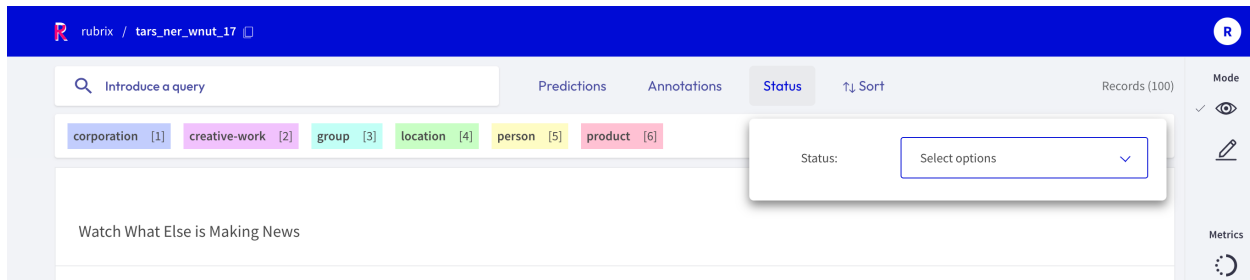
Annotations filter



This filter allows you to filter records with respect to their annotations:

- **Annotated as:** filter records with respect to their annotated labels
- **Annotated by:** filter records by the *annotation agent*
- **Only records not covered by rules:** this option only appears if you *defined rules* for your dataset. It allows you to show only records that are **not** covered by your rules.

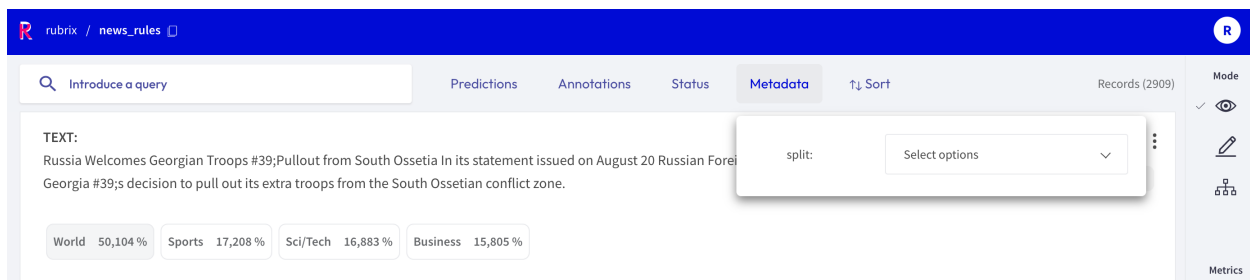
Status filter



This filter allows you to filter records with respect to their status:

- **Default:** records without any annotation or edition
- **Validated:** records with validated annotations
- **Edited:** records with annotations but still not validated

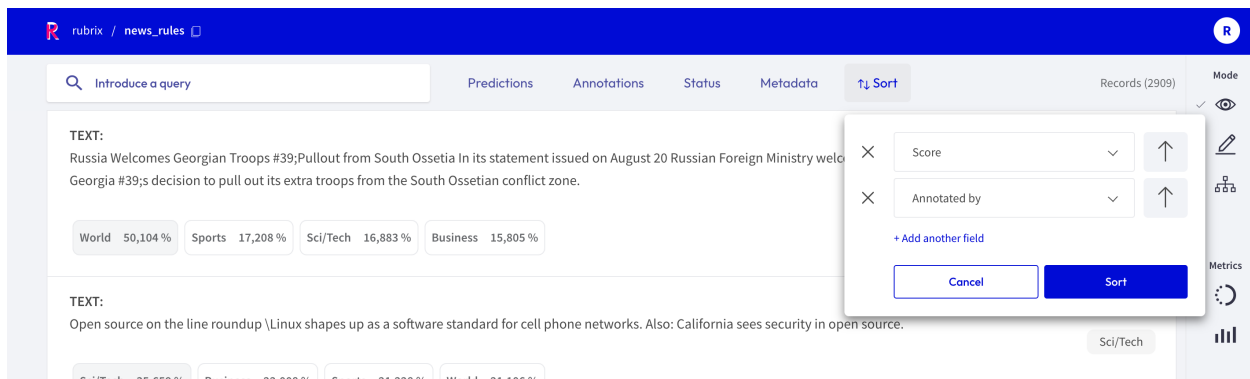
Metadata filter



This filter allows you to filter records with respect to their metadata.

Hint: Nested metadata will be flattened and the keys will be joint by a dot.

Sort records



With this component you can sort the records by various parameters, such as the predictions, annotations or their metadata.

Examples

Here we will provide a few examples how you can take advantage of the filters for different use cases.

Missing annotations

If you are annotating records and want to display only records that do not have an annotation yet, you can set the *status filter* to **Default**.

Low scores

If you uploaded model predictions and want to check for which records the model still struggles, you can use the *score filter* to filter records with a low score.

High loss

If you logged the *model loss* as a metadata for each record, you can *sort the records* by this loss in descending order to see records for which the model disagrees with the annotations (see this *tutorial* for an example).

View dataset metrics

The **Dataset Metrics** are part of the **Sidebar** placed on the right side of **Rubrix datasets**. To know more about this component, click *here*.

Rubrix metrics are very convenient in terms of assessing the status of the dataset, and to extract valuable information.

How to use Metrics

Metrics are composed of two submenus: **Progress** and **Stats**. These submenus might be different for **Token** and **Text Classification** tasks, as well as for the different modes (especially the **Define rules mode**).

Progress

This submenu is useful when users need to know how many records have been annotated, validated and/or discarded.

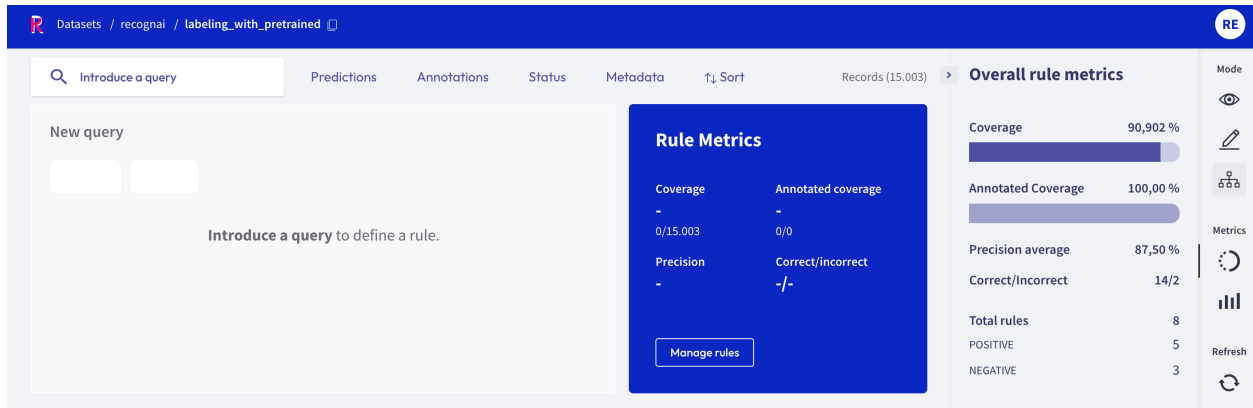
Annotation and Explore modes

When clicking on this menu, not only the progress is shown. The number of records is also displayed, as well as the number of labeled records or entities that are **validated** or **discarded**.

Define rules mode

In this mode, **progress** is related to the coverage of the rules. It shows the **model coverage** and the **annotated coverage**, and also the **precision average** and the number of correct and incorrect results.

In the **total rules** section, users can find the number of rules related to the different categories.



Stats

This submenu allows users to know more about the keywords of the dataset.

Explore and Define Rules mode

In both modes, the **Keywords** list displays a list of relevant words and the number of occurrences.

Annotation mode

In the **annotation mode**, the stats show the **mentions** (this is, the entities) which are present in the records.

This submenu has the **Predicted as** (for predictions) and the **Annotated as** (for annotation) sections, and here users can see the number of entities predicted or annotated with a specific label. The number of occurrences is ordered from highest to lowest, and the labels are also ordered in that way.

Refresh button

Users should click this button whenever they wanted to see the page updated. If any change is made, this button displays the updated page.

4.23 Developer documentation

Here we provide some guides for the development of *Rubrix*.

4.23.1 Development setup

To set up your system for *Rubrix* development, you first of all have to [fork](#) our [repository](#) and clone the fork to your computer:

```
git clone https://github.com/[your-github-username]/rubrix.git
cd rubrix
```

To keep your fork's master branch up to date with our repo you should add it as an [upstream remote branch](#):

```
git remote add upstream https://github.com/recognai/rubrix.git
```

Now go ahead and create a new conda environment in which the development will take place and activate it:

```
conda env create -f environment_dev.yml
conda activate rubrix
```

In the new environment *Rubrix* will already be installed in [editable mode](#) with all its server dependencies.

To keep a consistent code format, we use [pre-commit](#) hooks. You can install them by simply running:

```
pre-commit install
```

Install the *commit-msg* hook if you want to check your commit messages in your contributions:

```
pre-commit install --hook-type commit-msg
```

The last step is to build the static UI files in case you want to work on the UI:

```
bash scripts/build_frontend.sh
```

If you want to run the web app now, simply execute:

```
python -m rubrix
```

Congrats, you are ready to take *Rubrix* to the next level

4.23.2 Building the documentation

To build the documentation, make sure you set up your system for *Rubrix* development. Then go to the *docs* folder in your cloned repo and execute the `make` command:

```
cd docs
make html
```

This will create a `_build/html` folder in which you can find the `index.html` file of the documentation.

PYTHON MODULE INDEX

r

- `rubrix`, [220](#)
- `rubrix.client.datasets`, [228](#)
- `rubrix.client.models`, [224](#)
- `rubrix.labeling.text_classification.label_errors`,
[254](#)
- `rubrix.labeling.text_classification.label_models`,
[249](#)
- `rubrix.labeling.text_classification.rule`, [242](#)
- `rubrix.labeling.text_classification.weak_labels`,
[243](#)
- `rubrix.listeners`, [255](#)
- `rubrix.metrics.text_classification.metrics`,
[236](#)
- `rubrix.metrics.token_classification.metrics`,
[237](#)

Symbols

`__call__()` (*rubrix.labeling.text_classification.rule.Rule* method), 242

A

`annotation()` (*rubrix.labeling.text_classification.weak_labels.WeakLabels* method), 244

`annotation()` (*rubrix.labeling.text_classification.weak_labels.WeakMultiLabels* method), 247

`apply()` (*rubrix.labeling.text_classification.rule.Rule* method), 243

`author` (*rubrix.labeling.text_classification.rule.Rule* property), 243

C

`cardinality` (*rubrix.labeling.text_classification.weak_labels.WeakLabels* property), 245

`cardinality` (*rubrix.labeling.text_classification.weak_labels.WeakMultiLabels* property), 248

`change_mapping()` (*rubrix.labeling.text_classification.weak_labels.WeakLabels* method), 245

`char_id2token_id()` (*rubrix.client.models.TokenClassificationRecord* method), 227

`ComputeFor` (*class in rubrix.metrics.token_classification.metrics*), 237

`copy()` (*in module rubrix*), 220

D

`dataset` (*rubrix.listeners.RBListenerContext* property), 256

`DatasetForText2Text` (*class in rubrix.client.datasets*), 228

`DatasetForTextClassification` (*class in rubrix.client.datasets*), 229

`DatasetForTokenClassification` (*class in rubrix.client.datasets*), 231

`delete()` (*in module rubrix*), 220

`delete_records()` (*in module rubrix*), 221

E

`entity_capitalness()` (*in module*

rubrix.metrics.token_classification.metrics), 237

`entity_consistency()` (*in module rubrix.metrics.token_classification.metrics*), 237

`entity_density()` (*in module rubrix.metrics.token_classification.metrics*), 238

`entity_labels()` (*in module rubrix.metrics.token_classification.metrics*), 239

`extend_matrix()` (*rubrix.labeling.text_classification.weak_labels.WeakLabels* method), 245

`extend_matrix()` (*rubrix.labeling.text_classification.weak_labels.WeakMultiLabels* method), 248

`f1` (*in module rubrix.metrics*

rubrix.metrics.text_classification.metrics), 236

`f1` (*in module rubrix.metrics.token_classification.metrics*), 239

`f1_multilabel()` (*in module rubrix.metrics.text_classification.metrics*), 236

`find_label_errors()` (*in module rubrix.labeling.text_classification.label_errors*), 254

`fit()` (*rubrix.labeling.text_classification.label_models.FlyingSquid* method), 250

`fit()` (*rubrix.labeling.text_classification.label_models.MajorityVoter* method), 251

`fit()` (*rubrix.labeling.text_classification.label_models.Snorkel* method), 253

`FlyingSquid` (*class in rubrix.labeling.text_classification.label_models*), 249

`formatted_query` (*rubrix.listeners.RBDatasetListener* property), 256

`from_datasets()` (*rubrix.client.datasets.DatasetForText2Text* class method), 228

`from_datasets()` (*rubrix.client.datasets.DatasetForTextClassification* class method), 230

`from_datasets()` (*rubrix.client.datasets.DatasetForTokenClassification* class method), 232

`from_pandas()` (*rubrix.client.datasets.DatasetForText2Text* class method), 229

`from_pandas()` (*rubrix.client.datasets.DatasetForTextClassification* class method), 231

`from_pandas()` (*rubrix.client.datasets.DatasetForTokenClassification* class method), 233

`rubrix` labeling.text_classification.label_models, 249

`rubrix` labeling.text_classification.rule, 242

`rubrix` labeling.text_classification.weak_labels, 243

`rubrix` listeners, 255

`rubrix` metrics.text_classification.metrics, 236

`rubrix` metrics.token_classification.metrics, 237

G

`get_workspace()` (*in module rubrix*), 221

I

`init()` (*in module rubrix*), 221

`int2label` (*rubrix.labeling.text_classification.weak_labels.WeakLabels* property), 246

`is_running()` (*rubrix.listeners.RBDatasetListener* method), 256

L

`label` (*rubrix.labeling.text_classification.rule.Rule* property), 243

`label2int` (*rubrix.labeling.text_classification.weak_labels.WeakLabels* property), 246

`labels` (*rubrix.labeling.text_classification.weak_labels.WeakLabels* property), 246

`labels` (*rubrix.labeling.text_classification.weak_labels.WeakMultiLabels* property), 248

`listener()` (*in module rubrix.listeners*), 257

`load()` (*in module rubrix*), 222

`load_rules()` (*in module rubrix.labeling.text_classification.rule*), 243

`log()` (*in module rubrix*), 223

M

`MajorityVoter` (class *in rubrix.labeling.text_classification.label_models*), 251

`matrix()` (*rubrix.labeling.text_classification.weak_labels.WeakLabels* method), 246

`matrix()` (*rubrix.labeling.text_classification.weak_labels.WeakMultiLabels* method), 249

`mention_length()` (*in module rubrix.metrics.token_classification.metrics*), 240

`Metrics` (class *in rubrix.listeners*), 255

`metrics()` (*rubrix.labeling.text_classification.rule.Rule* method), 243

`module`

`rubrix`, 220

`rubrix.client.datasets`, 228

`rubrix.client.models`, 224

`rubrix.labeling.text_classification.label_errors`, 254

N

`name` (*rubrix.labeling.text_classification.rule.Rule* property), 243

P

`predict()` (*rubrix.labeling.text_classification.label_models.FlyingSquid* method), 250

`predict()` (*rubrix.labeling.text_classification.label_models.MajorityVoter* method), 251

`predict()` (*rubrix.labeling.text_classification.label_models.Snorkel* method), 253

`prediction_as_tuples()`

`prepare_for_training()` (*rubrix.client.models.Text2TextRecord* class method), 225

`prepare_for_training()` (*rubrix.client.datasets.DatasetForTextClassification* method), 231

`prepare_for_training()` (*rubrix.client.datasets.DatasetForTokenClassification* method), 233

Q

`query` (*rubrix.labeling.text_classification.rule.Rule* property), 243

`query` (*rubrix.listeners.RBListenerContext* property), 256

R

`RBDatasetListener` (class *in rubrix.listeners*), 255

`RBListenerContext` (class *in rubrix.listeners*), 256

`read_datasets()` (*in module rubrix.client.datasets*), 234

`read_pandas()` (*in module rubrix.client.datasets*), 235

`rubrix`

`module`, 220

`rubrix.client.datasets`

`module`, 228

`rubrix.client.models`

`module`, 224

`rubrix.labeling.text_classification.label_errors`

`module`, 254

`rubrix.labeling.text_classification.label_models`

module, 249
 rubrix.labeling.text_classification.rule
 module, 242
 rubrix.labeling.text_classification.weak_labels
 module, 243
 rubrix.listeners
 module, 255
 rubrix.metrics.text_classification.metrics
 module, 236
 rubrix.metrics.token_classification.metrics
 module, 237
 Rule (class in rubrix.labeling.text_classification.rule),
 242

S
 score() (rubrix.labeling.text_classification.label_models.FlyingSquid
 method), 251
 score() (rubrix.labeling.text_classification.label_models.MajorityVoter
 method), 252
 score() (rubrix.labeling.text_classification.label_models.Snorkel
 method), 254
 Search (class in rubrix.listeners), 257
 set_workspace() (in module rubrix), 224
 show_records() (rubrix.labeling.text_classification.weak_labels.WeakLabels
 method), 246
 show_records() (rubrix.labeling.text_classification.weak_labels.WeakMultiLabels
 method), 249
 Snorkel (class in rubrix.labeling.text_classification.label_models),
 253
 spans2io() (rubrix.client.models.TokenClassificationRecord
 method), 227
 start() (rubrix.listeners.RBDatasetListener method),
 256
 stop() (rubrix.listeners.RBDatasetListener method),
 256
 summary() (rubrix.labeling.text_classification.weak_labels.WeakLabels
 method), 246
 summary() (rubrix.labeling.text_classification.weak_labels.WeakMultiLabels
 method), 249

T
 Text2TextRecord (class in rubrix.client.models), 224
 TextClassificationRecord (class in
 rubrix.client.models), 225
 token_capitalness() (in module
 rubrix.metrics.token_classification.metrics),
 240
 token_frequency() (in module
 rubrix.metrics.token_classification.metrics),
 241
 token_length() (in module
 rubrix.metrics.token_classification.metrics),
 241
 token_span() (rubrix.client.models.TokenClassificationRecord
 method), 228
 TokenAttributions (class in rubrix.client.models), 226
 TokenClassificationRecord (class in
 rubrix.client.models), 226
 tokens_length() (in module
 rubrix.metrics.token_classification.metrics),
 241

W
 WeakLabels (class in rubrix.labeling.text_classification.weak_labels),
 243
 WeakMultiLabels (class in
 rubrix.labeling.text_classification.weak_labels),
 247