Rubrix

Release 0.7

Recognai

Nov 30, 2021

GETTING STARTED

CHAPTER

ONE

WHAT'S RUBRIX?

Rubrix is a **production-ready Python framework for exploring, annotating, and managing data** in NLP projects.

Key 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**.

Rubrix currently supports several `natural language processing` and `knowledge graph` use cases but we'll be adding support for speech recognition and computer vision soon.

CHAPTER

TWO

QUICKSTART

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

Make sure you have `Docker` installed and run (check the *setup and installation section* for a more detailed installation process):

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

And then run:

```
wget -O docker-compose.yml https://git.io/rb-docker && docker-compose up
```

Install Rubrix python library (and `transformers`, `pytorch` and `datasets` libraries for this example):

```
pip install rubrix==0.7.0 transformers datasets torch
```

Now, let's see an example: **Bootstraping 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 zeroshot 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
```

(continues on next page)

(continued from previous page)

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

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

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

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

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

    rb.log(item, 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`.

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 as a pandas DataFrame
rb_df = rb.load(name='news_zeroshot')

# 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.inputs.transform(lambda r: r["text"]),
   "label": rb_df.annotation,
})
```

CHAPTER THREE

USE CASES

- **Model monitoring and observability:** log and observe predictions of live models.
- **Ground-truth data collection**: collect labels to start a project from scratch or from existing 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 things like token attributions to understand your model predictions.

CHAPTER

FOUR

NEXT STEPS

The documentation is divided into different sections, which explore different aspects of Rubrix:

- *Setup and installation*
- *Concepts*
- **Tutorials**
- **Guides**
- **Reference**

CHAPTER

FIVE

COMMUNITY

You can join the conversation on our Github page and our Github forum.

- Github page
- Github forum

5.1 Setup and installation

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

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

5.1.1 1. Install the Rubrix Python client

First, make sure you have Python 3.6 or above installed.

Then you can install Rubrix with `pip`:

```
pip install rubrix==0.7.0
```

5.1.2 2. Launch the web app

There are two ways to launch the webapp:

a. Using docker-compose (**recommended**).

b. Executing the server code manually

a) Using `docker-compose` (recommended)

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.yaml && docker-compose up
```

This is the recommended way because it automatically includes an Elasticsearch instance, Rubrix's main persistent layer.

b) Executing the server code manually

When executing the server code manually you need to provide an Elasticsearch instance yourself. This method may be preferred if you (1) want to avoid or cannot use `Docker`, (2) have an existing Elasticsearch service, or (3) want to have full control over your Elasticsearch configuration.

1. First you need to install Elasticsearch (we recommend version 7.10) and launch an Elasticsearch instance. For MacOS and Windows there are Homebrew formulae and a msi package, respectively.
2. Install the Rubrix Python library together with its server dependencies:

```
pip install rubrix[server]==0.7.0
```

3. Launch a local instance of the Rubrix web app

```
python -m rubrix.server
```

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.

If you are already running an Elasticsearch instance for other applications and want to share it with Rubrix, please refer to our *advanced setup guide*.

5.1.3 3. Start logging data

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

```python
import rubrix as rb

rb.log(
    rb.TextClassificationRecord(inputs="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. **The default username and password are `rubrix` and `1234`** (see the user management guide to configure this). You can also check the REST API docs at http://localhost:6900/api/docs.

Congratulations! You are ready to start working with Rubrix.

Please refer to our *advanced setup guides* if you want to:

- setup Rubrix using docker
- share the Elasticsearch instance with other applications
- deploy Rubrix on an AWS instance
- manage users in Rubrix

5.1.4 Next steps

To continue learning we recommend you to:

- Check our **Guides** and **Tutorials.**
- Read about Rubrix's main *Concepts*

5.2 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.

5.2.1 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 stored in Rubrix. The main things you can do with a `Dataset` are to `log` records and to `load` the records of a `Dataset` into a `Pandas.Dataframe` from a Python app, script, or a Jupyter/Colab notebook.

Record

A record is a data item composed of `inputs` and, optionally, `predictions` and `annotations`. Usually, inputs are the information your model receives (for example: 'Macbeth').

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(
    inputs={
        "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",

    metadata={  # Information about this record
        "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 Egypts and visit the pyramids" fits into the hastag #Travel, which is accurate, in Multi-label Text Classification we can classify it as more than one hastag, like #Travel #History #Africa #Sightseeing #Desert.

```
record = rb.TextClassificationRecord(
    inputs={
        "text": "I can't wait to travel to Egypts 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",

    # When annotated, scores are suppoused to be 1
    annotation = ['travel', 'history'],   # list of all annotated labels,
    annotation_agent= "link or reference to annotator",

    metadata={  # Information about this record
        "split": "train"
    },

)
```

Token classification record

Token classification kind-of-tasks are NLP tasks aimed to divide the input text into words, or syllabes, and assign certain values to them. Think about giving each word in a sentence its gramatical 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 = token_list,

    # 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"
        },
    )
```

Task

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

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

Metada 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!

5.2.2 Methods

To find more information about these methods, please check out the *Client*.

rb.init

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

rb.log

Register a set of logs into Rubrix: `rubrix.log()`

rb.load

Load a dataset as a pandas DataFrame: `rubrix.load()`

rb.delete

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

5.3 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:

5.3.1 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 in and log and load data with the Python client. The name of this workspace corresponds to the username.

A user is given access to 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.

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.

5.3.2 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
:$2y$05$T5mHt/TfRHPPYwbeN2.q7e11QqhgvsHbhvQQ1c/pdap.xPZM2axje
```

Then set the environment variable omitting the first `:` character (in our case `$2y$05$T5...`):

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

5.3.3 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:latest
    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.

5.4 Advanced setup guides

Here we provide some advanced setup guides, in case you want to use docker, configure your own Elasticsearch instance or install the cutting-edge master version.

5.4.1 Using docker

You can use vanilla docker to run our image of the server. 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

5.4.2 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:

5.4.3 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 be 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 amazonec2 \
--amazonec2-root-size 60 \
--amazonec2-instance-type t2.large \
--amazonec2-open-port 80 \
--amazonec2-ami ami-0b541372 \
--amazonec2-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                    ACTIVE   DRIVER      STATE     URL                          SWARM   
↪DOCKER      ERRORS
rubrix-aws              -        amazonec2   Running   tcp://52.213.178.33:2376             
↪v20.10.7
```

Save asigned 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:v0.7.0
    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

5.4.4 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:

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

(continues on next page)

(continued from previous page)

```
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
```

If you want to execute the server code of the master branch manually, we refer you to our *Development setup*.

5.5 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 is *agnostic*, it can be used with any library or framework, no need to implement any interface or modify your existing toolbox and workflows.

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

If you miss a library in this guide, leave a message at the Rubrix Github forum.

5.5.1 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
%pip install transformers
%pip install datasets
```

Text Classification

Inference

Let's try a zero-shot classifier using `typeform/distilbert-base-uncased-mnli` for predicting the topic of a sentence.

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

input_text = "I love watching rock climbing competitions!"

# We define our HuggingFace Pipeline
classifier = pipeline(
    "zero-shot-classification",
    model="typeform/distilbert-base-uncased-mnli",
    framework="pt",
```

(continues on next page)

(continued from previous page)

```
)

# Making the prediction
prediction = classifier(
    input_text,
    candidate_labels=['World', 'Sports', 'Business', 'Sci/Tech'],
    hypothesis_template="This text is about {}.",
)

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

# Building a TextClassificationRecord
record = rb.TextClassificationRecord(
    inputs=input_text,
    prediction=prediction,
    prediction_agent="typeform/distilbert-base-uncased-mnli",
)

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

Training

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 `labelled_dataset` is expected to be found in your Rubrix client.

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

# load rubrix dataset
df = rb.load('labelled_dataset')

# inputs can be dicts to support multifield classifiers, we just use the text here.
df['text'] = df.inputs.transform(lambda r: r['text'])

# we create a dict for turning our annotations (labels) into numeric ids
label2id = {label: id for id, label in enumerate(df.annotation.unique())}


# create  dataset from pandas with labels as numeric ids
dataset = Dataset.from_pandas(df[['text', 'annotation']])
dataset = dataset.map(lambda example: {'labels': label2id[example['annotation']]})
```

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

# from here, it's just regular fine-tuning with  transformers
```

(continues on next page)

(continued from previous page)

```
tokenizer = AutoTokenizer.from_pretrained("distilbert-base-uncased")
model = AutoModelForSequenceClassification.from_pretrained("distilbert-base-uncased",␣
↪num_labels=4)

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

train_dataset = dataset.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",
    framework="pt",
)

# Making the prediction
predictions = classifier(
    input_text,
)

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

# Building a TokenClassificationRecord
record = rb.TokenClassificationRecord(
    text=input_text,
    tokens=input_text.split(),
    prediction=prediction,
    prediction_agent="https://huggingface.co/elastic/distilbert-base-cased-finetuned-
↪conll03-english",
)

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

5.5.2 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,
    tokens=[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")

# 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,
    tokens=[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")
```

5.5.3 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
```

If you get an error message when trying to import flair due to issues for downloading the wordnet_ic package try running the following and manually download the `wordnet_ic` package (available under the All Packages tab). Otherwise you can skip this cell.

[ ]:
```
import nltk
import ssl

try:
    _create_unverified_https_context = ssl._create_unverified_context
except AttributeError:
    pass
else:
    ssl._create_default_https_context = _create_unverified_https_context
```

(continues on next page)

(continued from previous page)

```
nltk.download()
```

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)

# 2. Pre-processing training pandas dataframe
ready_input = [row['text'] for row in train_dataset.inputs]

train_df = pd.DataFrame()
train_df['text'] = ready_input
train_df['label'] = train_dataset['annotation']

# 3. Save as csv with tab delimiter
train_df.to_csv('train.csv', sep='\t')
```

```
[ ]: # 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)
```

(continues on next page)

(continued from previous page)

```
# 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.,
              )
```

Inference

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)
```

Text Classification

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:

[ ]:
```python
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
input_text = "I am so glad you liked it!"
sentence = Sentence(input_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(
    inputs=input_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 Deutch offensive language model.

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

input_text = "Du erzählst immer Quatsch."  # something like: "You are always narrating␣
↪silliness."

# Load our pre-trained classifier
```

(continues on next page)

(continued from previous page)

```
classifier = TextClassifier.load("de-offensive-language")

# Creating Sentence object
sentence = Sentence(input_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.labels]

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

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

Training

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

[ ]:
```
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)

# 2. Pre-processing training pandas dataframe
ready_input = [row['text'] for row in train_dataset.inputs]

train_df = pd.DataFrame()
train_df['text'] = ready_input
train_df['label'] = train_dataset['annotation']

# 3. Save as csv with tab delimiter
train_df.to_csv('train.csv', sep='\t')
```

(continues on next page)

(continued from previous page)

```
[ ]: # 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.,
              )
```

Inference

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
```

(continues on next page)

(continued from previous page)

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

# predict class and print
classifier.predict(sentence)

print(sentence.labels)
```

Token Classification

Flair offers a lot of tools for Token Classification, supporting tasks like named entity recognition (NER), part-of-speech tagging (POS), special support for biomedical data, etc. with a growing number of supported languages.

Let's see some examples for NER and POS tagging.

NER

In this example, we will try the pretrained Dutch NER model from Flair.

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

input_text = "De Nachtwacht is in het Rijksmuseum"

# Loading our NER model from flair
tagger = SequenceTagger.load("flair/ner-dutch")

# 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("ner")
]

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

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

POS tagging

In the following snippet we will use de 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",
)

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

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
```

(continues on next page)

(continued from previous page)

```
# 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")
```

[ ]:
```
# 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 = []
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 = {}
```

(continues on next page)

(continued from previous page)

```
    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):
    '''
    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((a[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('\n')


# 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)
```

(continues on next page)

(continued from previous page)

```
# 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(
    embeddings=embedding_types)

# 5. initialize sequence tagger
tagger = SequenceTagger(hidden_size=256,
                        embeddings=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)
```

Inference

Let's make a prediction with flair SequenceTagger

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

# load the trained model
model = SequenceTagger.load('./token-classification/best-model.pt')

# create example sentence
sentence = Sentence('I want to fly from Barcelona to Paris next month')

# predict the tags
model.predict(sentence)
```

(continues on next page)

(continued from previous page)

```
print(sentence.to_tagged_string())
```

5.5.4 Stanza

Stanza is a collection of efficient tools for many NLP tasks and processes, all in one library. It's maintained by the Standford 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

input_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")

# Analizing the input text
doc = nlp(input_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 suppouse 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)

```
    inputs=input_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 syntantic 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")

# Analizing 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,
    tokens=[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")

# Analizing 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,
    tokens=[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")
```

5.6 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.

5.6.1 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 SequeezeBERT 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(
            inputs=record["text"],
            prediction=prediction,
            prediction_agent="https://huggingface.co/typeform/squeezebert-mnli",
            metadata={"split": "train"},
        )
    )

# Logging into Rubrix
rb.log(
    records=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(
            inputs=record["content"],
            prediction=prediction,
            prediction_agent="https://huggingface.co/cardiffnlp/twitter-roberta-base-
↪sentiment",
            metadata={"split": "train"},
        )
    )

# Logging into Rubrix
rb.log(
    records=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(
            inputs={
                "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(
    records=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(
            inputs={"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(
    records=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 datastet, 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(
            inputs=record["statement"],
            prediction=prediction,
            prediction_agent="https://huggingface.co/typeform/squeezebert-mnli",
            metadata={"split": "train"},
        )
    )

# Logging into Rubrix
rb.log(
    records=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)

```python
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(
            inputs=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(
    records=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.

5.6.2 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,
            tokens=tokens,
            prediction=entities,
            prediction_agent="en_core_web_trf",
        )
    )

# Logging into Rubrix
rb.log(
    records=records,
    name="ner",
    tags={
        "task": "NER",
        "family": "token-classification",
        "dataset": "gutenberg-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,
            tokens=[token.text for token in doc],
            prediction=prediction,
            prediction_agent="en_core_web_trf",
        )
    )

# Logging into Rubrix
rb.log(
    records=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 != "O"}

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

     head_config = TokenClassificationConfiguration(
         labels=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,
            tokens=record,
            prediction=prediction,
            prediction_agent="biome_slot_filling_tutorial",
        )
    )

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

5.6.3 Text2Text (Experimental)

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_sequences=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 to a DataFrame for the fine-tuning of the translation model
df = rb.load("sesame_street_en-es")
```

5.7 Weak supervision

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

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

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

5.7.1 Rubrix weak supervision in a nutshell

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"
)

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

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

5.7.2 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*.

5.7.3 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 rules, exploring and trying out different things directly in the Rubrix web app.
3. Create a `WeakLabels` object and apply the rules. 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 and Flyingsquid. Let's get started!

5.7.4 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
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
```

5.7.5 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(
        inputs=row.text,
        metadata={"video":row.video, "author": row.author}
    )
    for i,row in train_df.iterrows()
]

# build records from the test dataset
labels = ["HAM", "SPAM"]
records += [
    rb.TextClassificationRecord(
        inputs=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 at `http://localhost:6900/weak_supervision_yt` (or the base URL where your Rubrix instance is hosted).

5.7.6 2. Defining rules

Let's now define some of the rules proposed in the tutorial Snorkel Intro Tutorial: Data Labeling.

Remember you can use Elasticsearch's query string DSL and test your queries directly in the web app. Available fields in the query are described in *the Rubrix web app reference*.

[4]:
```
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")
subscribe = Rule(query="subscribe", label="SPAM")
my = Rule(query="my", label="SPAM")
song = Rule(query="song", label="HAM")
love = Rule(query="love", label="HAM")
```

Besides using the UI, if you want to quickly see the effect of a rule, you can do:

[10]:
```
# display full length text
pd.set_option('display.max_colwidth', None)
```

(continues on next page)

(continued from previous page)

```python
# get the subset for the rule query
rb.load(name="weak_supervision_yt", query="plz OR please")[['inputs']]
```

[10]:

```
                                                                                                     ␣
↪                                                                                                    ␣
↪                                                                                                    ␣
↪                                                                                                    ␣
↪                                                                                                    ␣
↪                                                                                                    ␣
↪      inputs
0                                                                                                    ␣
↪                                                                                                    ␣
↪                                                                                                    ␣
↪                                                                                                    ␣
↪                                                                                                    ␣
↪                                                    {'text': 'Thank you. Please give␣
↪your email. '}
1                                                                                                    ␣
↪                                                                                                    ␣
↪                                                                                                    ␣
↪                                                              {'text': 'HUH HYUCK HYUCK␣
↪IM SPECIAL WHO&#39;S WATCHING THIS IN 2015 IM FROM AUSTRALIA OR SOMETHING GIVE ME␣
↪ATTENTION PLEASE IM JUST A RAPPER WITH A DREAM IM GONNA SHARE THIS ON GOOGLE PLUS␣
↪BECAUSE IM SO COOL.'}
2                       {'text': 'Media is Evil! Please see and share: W W W. THE FARRELL␣
↪REPORT. NET  Top Ex UK Police Intelligence Analyst turned Whistleblower Tony Farrell␣
↪exposes a horrific monstrous cover-up perpetrated by criminals operating crimes from␣
↪inside Mainstream Entertainment and Media Law firms. Beware protect your children!!␣
↪These devils brutally target innocent people. These are the real criminals linked to␣
↪London&#39;s 7/7 attacks 2005.  MUST SEE AND MAKE VIRAL!!! Also see UK Column video on␣
↪31st January 2013.'}
3                   {'text': 'hey guys if you guys can please SUBSCRIBE to my channel ,i&
↪#39;m a young rapper really dedicated i post a video everyday ,i post a verse (16␣
↪bars)(part of a song)everyday to improve i&#39;m doing this for 365 days ,right now i&
↪#39;m on day 41  i&#39;m doing it for a whole year without missing one day if you guys␣
↪can please SUBSCRIBE and follow me on my journey to my dream watch me improve, it␣
↪really means a lot to me  thank you (:, i won&#39;t let you down i promise(: i&#39;m␣
↪lyrical i keep it real!'}
4                                                                                                    ␣
↪                                                                                                    ␣
↪                                                                                                    ␣
↪                                                                                                    ␣
↪      {'text': 'Please do buy these new Christmas shirts! You can buy at any time␣
↪before  December 4th and they are sold worldwide! Don't miss out:  http://teespring.
↪com/treechristmas'}
..                                                                                                   ␣
↪                                                                                                    ␣
↪                                                                                                    ␣
↪                                                                                                    ␣
↪                                                                                                    ␣
↪                                                                                                    ␣
↪         ...
```

(continues on next page)

(continued from previous page)

```
181                                                                                    ␣
↪                                                                                      ␣
↪                                                                                      ␣
↪                                                                                      ␣
↪                                                                                      ␣
↪                                                      {'text': 'Please subscribe to us and␣
↪thank you'}
182  {'text': 'My honest opinion. It's a very mediocre song. Nothing unique or special ␣
↪about her music, lyrics or voice. Nothing memorable like Billie Jean or  Beat It.␣
↪Before her millions of fans reply with hate comments, i know this  is a democracy and␣
↪people are free to see what they want. But then don't I  have the right to express my␣
↪opinion? Please don't reply with dumb comments  lie "if you don't like it don't watch␣
↪it". I just came here to see what's  the buzz about(661 million views??) and didn't␣
↪like what i saw. OK?'}
183                                                                                    ␣
↪                                                                                      ␣
↪                                                                                      ␣
↪                                                                                      ␣
↪                                                                                      ␣
↪                  {'text': 'EVERYONE PLEASE GO SUBSCRIBE TO MY CHANNEL OR JUST LOON AT␣
↪MY VIDEOS'}
184                                                                                    ␣
↪                                                                                      ␣
↪                                                                                      ␣
↪                                                                                      ␣
↪                                                                                      ␣
↪                      {'text': 'please suscribe i am bored of 5 subscribers try to get␣
↪it to 20!'}
185                                                                                    ␣
↪                                                                                      ␣
↪                                                                                      ␣
↪                                                                                      ␣
↪           {'text': 'https://www.facebook.com/eeccon/posts/733949243353321?comment_
↪id=734237113324534&amp;offset=0&amp;total_comments=74   please like frigea marius␣
↪gabriel comment :D'}

[186 rows x 1 columns]
```

You can also define plain Python labeling functions:

[ ]:
```python
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
```

(continues on next page)

(continued from previous page)

5.7.7 3. Building and analizing weak labels

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

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

[26]:
```
# show some stats about the rules, see the `summary()` docstring for details
weak_labels.summary()
```

[26]:
```
                    polarity  coverage  overlaps  conflicts  correct  \
check out             {SPAM}  0.235379  0.229147   0.028763       90
plz OR please         {SPAM}  0.089166  0.079099   0.019175       40
subscribe             {SPAM}  0.108341  0.084372   0.028763       60
my                    {SPAM}  0.190316  0.167306   0.050815       82
song                   {HAM}  0.139981  0.085331   0.034995       78
love                   {HAM}  0.097795  0.075743   0.032119       56
contains_http         {SPAM}  0.096357  0.066155   0.045062       12
short_comment          {HAM}  0.259827  0.113135   0.058965      168
regex_check_out       {SPAM}  0.220997  0.220518   0.026846       90
total            {SPAM, HAM}  0.764621  0.447267   0.116970      676

                 incorrect  precision
check out                0   1.000000
plz OR please            0   1.000000
subscribe                0   1.000000
my                      12   0.872340
song                    18   0.812500
love                    14   0.800000
contains_http            0   1.000000
short_comment           16   0.913043
regex_check_out          0   1.000000
total                   60   0.918478
```

5.7.8 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:

Label model with Snorkel

Snorkel 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
label_model = Snorkel(weak_labels)

# we train the model
label_model.fit()

# we check its performance
label_model.score()
```

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 then 0.8.)

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

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

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.

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

By default, the `WeakLabels` class uses `-1` as value for an abstention. FlyingSquid, though, expects a value of `0`. With Rubrix you can define a custom `label2int` mapping like this:

```
[ ]: weak_labels = WeakLabels(rules=rules, dataset="weak_supervision_yt", label2int={None: 0,
     ↪'SPAM': -1, 'HAM': 1})
```

```
[ ]: from flyingsquid.label_model import LabelModel

     # train our label model
     label_model = LabelModel(len(weak_labels.rules))
     label_model.fit(L_train=weak_labels.matrix(has_annotation=False),verbose=True)
```

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 then 0.8.)

```
[ ]: # get the part of the weak label matrix that has no corresponding annotation
     train_matrix = weak_labels.matrix(has_annotation=False)

     # get predictions from our label model
     predictions = label_model.predict_proba(L_matrix=train_matrix)
     predicted_labels = label_model.predict(L_matrix=train_matrix)
     preds = [[('SPAM', pred[0]), ('HAM', pred[1])] for pred in predictions]

     # get the records that do not have an annotation
     train_records = weak_labels.records(has_annotation=False)
```

```
[ ]: # add the predictions to the records
     def add_prediction(record, prediction):
         record.prediction = prediction
         return record

     train_records_with_lm_prediction = [
         add_prediction(rec, pred)
         for rec, pred, label in zip(train_records, preds, predicted_labels)
         if label != weak_labels.label2int[None] # exclude records where the label model␣
     ↪abstains
     ]

     # log a new dataset to Rubrix
     rb.log(train_records_with_lm_prediction, name="flyingsquid_results")
```

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.

First, we need to install the Weasel python package:

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

Before we get started, we need to define some classes, that wrap our data and our end model in a way Weasel can work with them.

[ ]:
```
from weasel.datamodules.base_datamodule import AbstractWeaselDataset,␣
↪AbstractDownstreamDataset
from weasel.models.downstream_models.base_model import DownstreamBaseModel
from transformers import AutoModelForSequenceClassification, AutoTokenizer
from torch.utils.data import DataLoader
import torch


class TrainDataset(AbstractWeaselDataset):
    def __init__(self, L, inputs):
        super().__init__(L, None)
        self.inputs = inputs

        if self.L.shape[0] != len(self.inputs):
            raise ValueError("L and inputs have different number of samples")

    def __getitem__(self, item):
        return self.L[item], self.inputs[item]


class TestDataset(AbstractDownstreamDataset):
    def __init__(self, inputs, Y):
        super().__init__(None, Y)
        self.inputs = inputs

        if len(self.Y) != len(self.inputs):
            raise ValueError("inputs and Y have different number of samples")

    def __getitem__(self, item):
        return self.inputs[item], self.Y[item]

class TrainCollator:
    def __init__(self, tokenizer):
        self._tokenizer = tokenizer
    def __call__(self, batch):
        L = torch.stack([b[0] for b in batch])
        inputs = {key: [b[1][key] for b in batch] for key in batch[0][1]}
        return L, self._tokenizer.pad(inputs, return_tensors="pt")


class TestCollator:
    def __init__(self, tokenizer):
        self._tokenizer = tokenizer
    def __call__(self, batch):
        Y = torch.stack([b[1] for b in batch])
        inputs = {key: [b[0][key] for b in batch] for key in batch[0][0]}
        return self._tokenizer.pad(inputs, return_tensors="pt"), Y
```

(continues on next page)

(continued from previous page)

```
class TransformersEndModel(DownstreamBaseModel):
    def __init__(self, name: str, num_labels: int = 2):
        super().__init__()
        self.out_dim = num_labels
        self.model = AutoModelForSequenceClassification.from_pretrained(name, num_
↪labels=num_labels)

    def forward(self, kwargs):
        model_output = self.model(**kwargs)
        return model_output["logits"]
```

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.

[ ]:
```
# instantiate our transformers end model
end_model = TransformersEndModel("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_LFs=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 torch `Datasets` and `DataLoaders`, 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.

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

# torch data set of our training data
train_ds = TrainDataset(
    L=weak_labels.matrix(has_annotation=False),
```

(continues on next page)

(continued from previous page)

```
    inputs=[tokenizer(rec.inputs["text"], truncation=True)
            for rec in weak_labels.records(has_annotation=False)],
)

# torch data set of our test data
test_ds = TestDataset(
    inputs=[tokenizer(rec.inputs["text"], truncation=True)
            for rec in weak_labels.records(has_annotation=True)],
    Y=weak_labels.annotation(),
)

# torch data loader for our training data
train_loader = DataLoader(
    dataset=train_ds,
    collate_fn=TrainCollator(tokenizer),
    batch_size=8,
)

# torch data loader for our test data
test_loader = DataLoader(
    dataset=test_ds,
    collate_fn=TestCollator(tokenizer),
    batch_size=16,
)
```

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,
    train_dataloaders=train_loader,
    val_dataloaders=test_loader
)
```

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

```
[ ]: trainer.test(dataloaders=test_loader)  # List of test metrics
```

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

[ ]:
```python
# 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:

[ ]:
```python
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}]
```

5.8 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.

5.8.1 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:

[ ]:
```python
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 Kibana configured, so you can easily explore your model predictions and build custom dashboards (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:

5.8.2 Using the ASGI middleware

For using the ASGI middleware, see this *tutorial*

5.9 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 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.

5.9.1 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
```

5.9.2 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

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

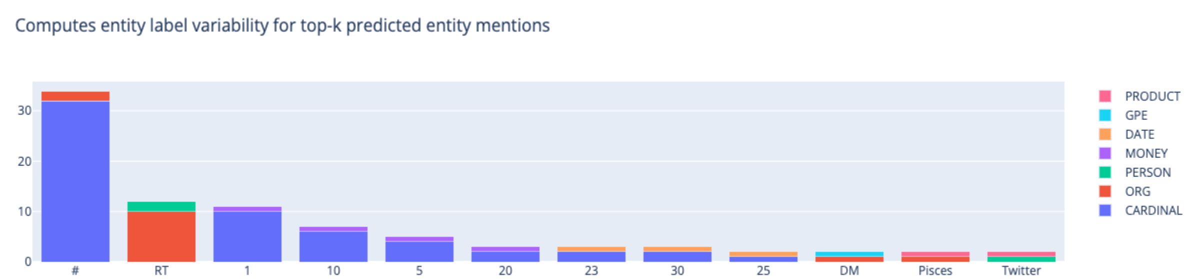
Log records into a Rubrix dataset

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

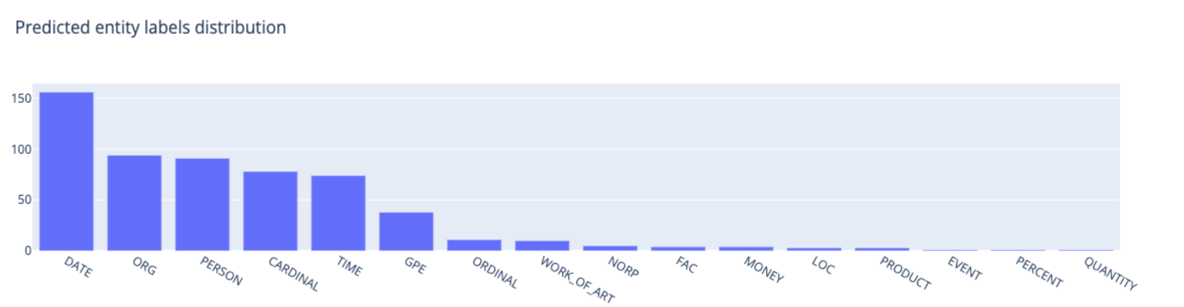
```
[ ]: nlp = rb.monitor(nlp, dataset="spacy_sm_wnut17")

def predict_batch(records):
    docs = nlp(" ".join(records["tokens"]))
    return {"predicted": [True for _ in docs]}

dataset.map(predict_batch)
```

Explore the metrics for this pipeline

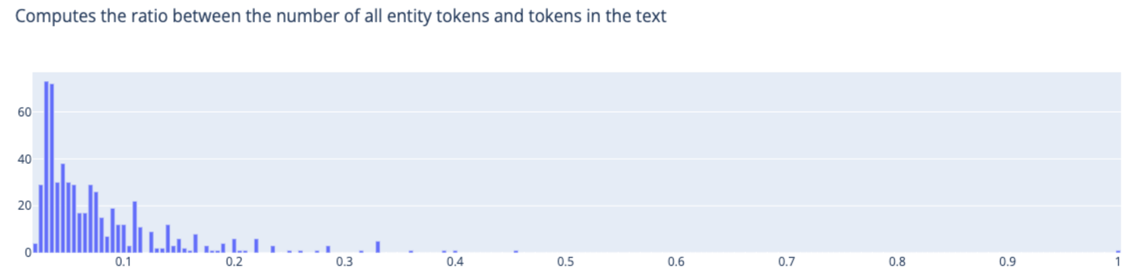
```
[35]: from rubrix.metrics.token_classification import entity_consistency

entity_consistency(name="spacy_sm_wnut17", mentions=5000, threshold=2).visualize()
```



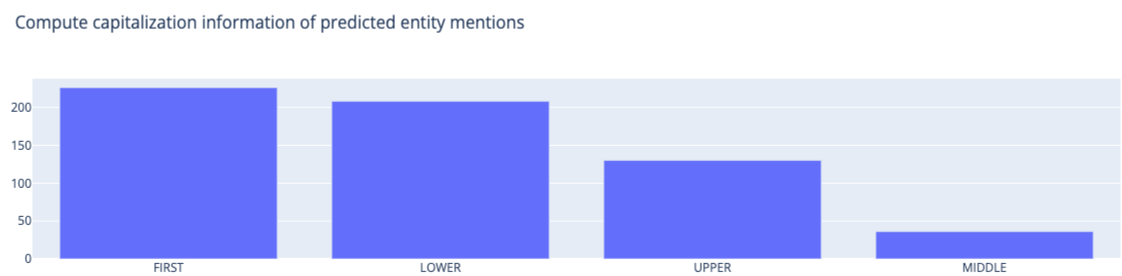

```
[15]: from rubrix.metrics.token_classification import entity_labels

entity_labels(name="spacy_sm_wnut17").visualize()
```



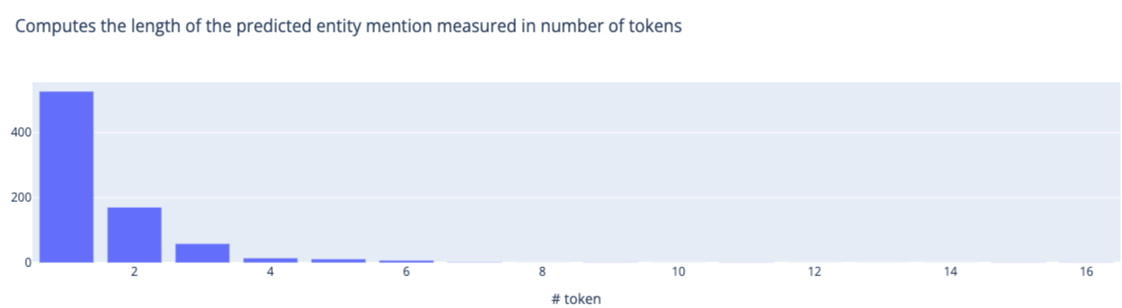

```
[16]: from rubrix.metrics.token_classification import entity_density

entity_density(name="spacy_sm_wnut17").visualize()
```



[17]:
```
from rubrix.metrics.token_classification import entity_capitalness

entity_capitalness(name="spacy_sm_wnut17").visualize()
```



[19]:
```
from rubrix.metrics.token_classification import mention_length
mention_length(name="spacy_sm_wnut17").visualize()
```



5.9.3 2. Rubrix Metrics for training sets

Analyzing tags

[20]:
```
dataset = load_dataset("conll2002", "es", split="train[0:5000]")
```

```
Downloading:   0%|          | 0.00/2.63k [00:00<?, ?B/s]
```

```
Downloading:   0%|          | 0.00/2.01k [00:00<?, ?B/s]
```

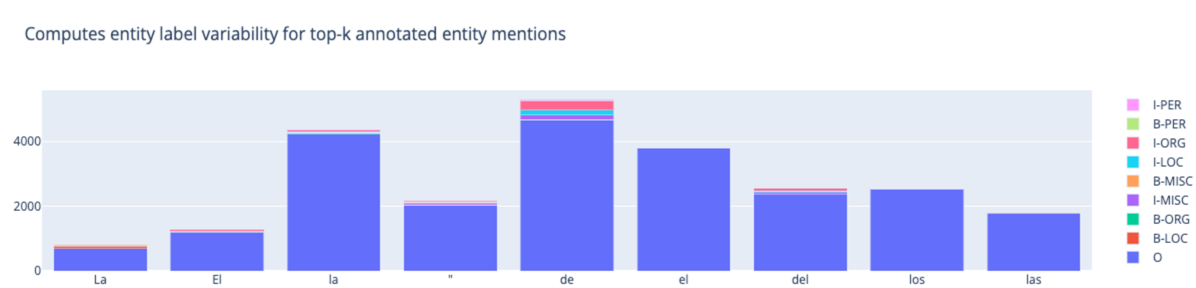
```
Reusing dataset conll2002 (/Users/dani/.cache/huggingface/datasets/conll2002/es/1.0.0/
↪a3a8a8612caf57271f5b35c5ae1dd25f99ddb9efb9c1667abaa70ede33e863e5)
```

[24]:
```
def parse_entities(record):
    entities = []
    counter = 0
    for i in range(len(record['ner_tags'])):
        entity = (dataset.features["ner_tags"].feature.names[record["ner_tags"][i]],␣
↪counter, counter + len(record["tokens"][i]))
        entities.append(entity)
        counter += len(record["tokens"][i]) + 1
    return entities
```

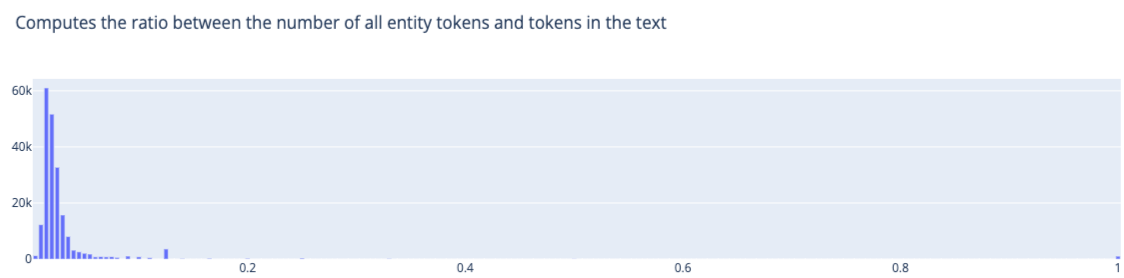
[30]:
```
records = [
    rb.TokenClassificationRecord(
        text=" ".join(example["tokens"]),
        tokens=example["tokens"],
        annotation=parse_entities(example)
    )
    for example in dataset
]
```

[ ]:
```
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", mentions=30, threshold=4, compute_
↪for=Annotations).visualize()
```



[54]:
```
from rubrix.metrics.token_classification import *

entity_density(name="conll2002_es", compute_for=Annotations).visualize()
```



5.10 How to label your data and fine-tune a sentiment classifier

5.10.1 TL;DR

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 (2.3 million monthly downloads as of July 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.

5.10.2 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 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!

5.10.3 Setup Rubrix

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*.

Once installed, you only need to import Rubrix:

[1]:
```
import rubrix as rb
```

5.10.4 Install tutorial dependencies

In this tutorial, we'll use the `transformers` and `datasets` libraries.

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

5.10.5 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**, which might be useful for digital assistants and customer service analytics.

Let's load the dataset directly from the hub:

[ ]:
```
from datasets import load_dataset

banking_ds = load_dataset("banking77")
```

For this tutoral, let's split the dataset into two 50% splits. We'll start with the `to_label1` split for data exploration and annotation and keep `to_label2` for further iterations.

[ ]:
```
to_label1, to_label2 = banking_ds['train'].train_test_split(test_size=0.5, seed=42).
↪values()
```

Model: sentiment `distilbert` fine-tuned on sst-2

As of July 2021, the `distilbert-base-uncased-finetuned-sst-2-english` is the most popular text-classification model in the Hugging Face Hub.

This model is a distilbert model fine-tuned on the highly popular sentiment classification benchmark SST-2 (Stanford Sentiment Treebank).

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**.

[6]:
```python
from transformers import pipeline

sentiment_classifier = pipeline(
    model="distilbert-base-uncased-finetuned-sst-2-english",
    task="sentiment-analysis",
    return_all_scores=True,
)
```

Now let's test this pipeline with an example of our dataset:

[15]:
```python
to_label1[3]['text'], sentiment_classifier(to_label1[3]['text'])
```

[15]:
```
('I just have one additional card from the USA. Do you support that?',
 [[{'label': 'NEGATIVE', 'score': 0.5619744062423706},
   {'label': 'POSITIVE', 'score': 0.43802565336227417}]])
```

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 annotaion 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.

5.10.6 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 will use the handy `dataset.map` method from the `datasets` library.

Predict

[16]:
```
def predict(examples):
    return {"predictions": sentiment_classifier(examples['text'], truncation=True)}
```

[ ]:
```
to_label1 = to_label1.map(predict, batched=True, batch_size=4)
```

Log

The following code builds a list of Rubrix records with the predictions and logs them into a Rubrix Dataset. We'll use this dataset to explore and label our first training set.

[18]:
```
records = []
for example in to_label1.shuffle():
    record = rb.TextClassificationRecord(
        inputs=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)
```

[ ]:
```
rb.log(name='labeling_with_pretrained', records=records)
```

5.10.7 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.

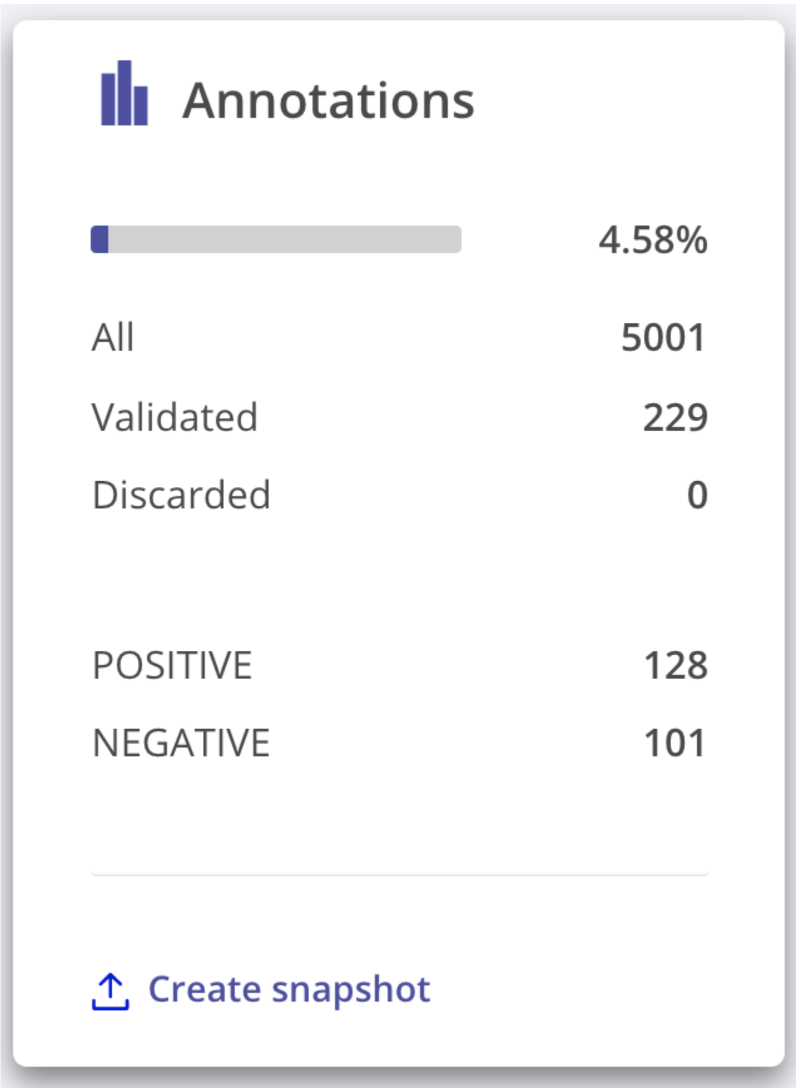
Rubrix provides you with a search-driven UI to annotated data, using free-text search, search filters and the Elasticsearch query DSL for advanced queries. This is most useful for sparse datasets, tasks with a high number of labels or unbalanced classes. In the standard case, we recommend you to follow the workflow below:

1. **Start labeling examples sequentially**, without using search features. This way you'll 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 spending 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:



5.10.8 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 our dataset:

[2]:
```python
rb_df = rb.load(name='labeling_with_pretrained')
```

This dataset contains all records, let's filter only our annotations using the status column. The `Validated` status corresponds to annotated records. You can read more about how record status is defined in Rubrix.

[3]:
```python
rb_df = rb_df[rb_df.status == "Validated"]
```

[4]:
```python
rb_df.head()
```

[4]:
```
                                                 inputs  \
4771  {'text': 'I saw there is a cash withdrawal fro...
4772  {'text': 'Why is it showing that my account ha...
4773  {'text': 'I thought I lost my card but I found...
4774  {'text': 'I wanted to top up my account and it...
4775  {'text': 'I need to deposit my virtual card, h...

                                             prediction  annotation  \
4771  [(NEGATIVE, 0.9997006654739381), (POSITIVE, 0...  [NEGATIVE]
4772  [(NEGATIVE, 0.9991878271102901), (POSITIVE, 0...  [NEGATIVE]
4773  [(POSITIVE, 0.9842885732650751), (NEGATIVE, 0...  [POSITIVE]
4774  [(NEGATIVE, 0.999732434749603), (POSITIVE, 0.0...  [NEGATIVE]
4775  [(NEGATIVE, 0.9992493987083431), (POSITIVE, 0...  [POSITIVE]

                                     prediction_agent annotation_agent  \
4771  distilbert-base-uncased-finetuned-sst-2-english    .local-Rubrix
4772  distilbert-base-uncased-finetuned-sst-2-english    .local-Rubrix
4773  distilbert-base-uncased-finetuned-sst-2-english    .local-Rubrix
4774  distilbert-base-uncased-finetuned-sst-2-english    .local-Rubrix
4775  distilbert-base-uncased-finetuned-sst-2-english    .local-Rubrix

      multi_label explanation                                    id  \
4771        False        None  0001e324-3247-4716-addc-d9d9c83fd8f9
4772        False        None  0017e5c9-c135-44b9-8efb-a17ffecdbe68
4773        False        None  0048ccce-8c9f-453d-81b1-a966695e579c
4774        False        None  0046aadc-2344-40d2-a930-81f00687bf44
4775        False        None  00071745-741d-4555-82b3-54d25db44c38

              metadata     status event_timestamp
4771  {'category': 20}  Validated            None
4772  {'category': 34}  Validated            None
4773  {'category': 13}  Validated            None
4774  {'category': 59}  Validated            None
4775  {'category': 37}  Validated            None
```

Prepare training and test datasets

Let's now prepare our dataset for training and testing our sentiment classifier, using the `datasets` library:

[ ]:
```
from datasets import Dataset

# select text input and the annotated label
rb_df['text'] = rb_df.inputs.transform(lambda r: r['text'])
# keep in mind that `rb_df.annotation` can be a list of labels
# to support multi-label text classifiers
rb_df['labels'] = rb_df.annotation


# create  dataset from pandas with labels as numeric ids
label2id = {"NEGATIVE": 0, "POSITIVE": 1}
train_ds = Dataset.from_pandas(rb_df[['text', 'labels']])
train_ds = train_ds.map(lambda example: {'labels': label2id[example['labels']]})
```

[6]:
```
train_ds = train_ds.train_test_split(test_size=0.2) ; train_ds
```

[6]:
```
DatasetDict({
    train: Dataset({
        features: ['__index_level_0__', 'labels', 'text'],
        num_rows: 183
    })
    test: Dataset({
        features: ['__index_level_0__', 'labels', 'text'],
        num_rows: 46
    })
})
```

[ ]:
```
from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained("distilbert-base-uncased-finetuned-sst-2-
↪english")

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

train_dataset = train_ds['train'].map(tokenize_function, batched=True).shuffle(seed=42)
eval_dataset = train_ds['test'].map(tokenize_function, batched=True).shuffle(seed=42)
```

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, we leave this experiment to you.

Let's load the model:

[1]:
```
from transformers import AutoModelForSequenceClassification

model = AutoModelForSequenceClassification.from_pretrained("distilbert-base-uncased-
↪finetuned-sst-2-english")
```

(continues on next page)

(continued from previous page)

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_steps=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 train our first model!

[ ]:
```
trainer.train()
```

5.10.9 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:

[33]:
```
finetuned_sentiment_classifier = pipeline(
    model=model.to("cpu"),
    tokenizer=tokenizer,
    task="sentiment-analysis",
    return_all_scores=True
)
```

And compare its predictions with the pre-trained model with an example:

[34]:
```
finetuned_sentiment_classifier(
    'I need to deposit my virtual card, how do i do that.'
), sentiment_classifier(
```

(continues on next page)

(continued from previous page)

```
    'I need to deposit my virtual card, how do i do that.'
)
```

[34]:
```
([[{'label': 'NEGATIVE', 'score': 0.0002401248930254951},
   {'label': 'POSITIVE', 'score': 0.9997599124908447}]],
 [[{'label': 'NEGATIVE', 'score': 0.9992493987083435},
   {'label': 'POSITIVE', 'score': 0.0007506058318540454}]])
```

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:

[35]:
```
finetuned_sentiment_classifier(
    'Why is my payment still pending?'
), sentiment_classifier(
    'Why is my payment still pending?'
)
```

[35]:
```
([[{'label': 'NEGATIVE', 'score': 0.9988037347793579},
   {'label': 'POSITIVE', 'score': 0.0011962745338678360}]],
 [[{'label': 'NEGATIVE', 'score': 0.9983781576156616},
   {'label': 'POSITIVE', 'score': 0.0016218466917052865}]])
```

5.10.10 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_df = rb.load(name='labeling_with_pretrained')
rb_df = rb_df[rb_df.status == "Default"]
rb_df['text'] = rb_df.inputs.transform(lambda r: r['text'])
```

From here, this is basically the same as step 1, in this case using our fine-tuned model:

[64]:
```
ds = Dataset.from_pandas(rb_df[['text']])
```

[65]:
```
def predict(examples):
    return {"predictions": finetuned_sentiment_classifier(examples['text'])}
```

[ ]:
```
ds = ds.map(predict, batched=True, batch_size=8)
```

[67]:
```
records = []
for example in ds.shuffle():
    record = rb.TextClassificationRecord(
        inputs=example["text"],
        prediction=[(pred['label'], pred['score']) for pred in example['predictions']],
        prediction_agent="distilbert-base-uncased-banking77-sentiment"
    )
    records.append(record)
```

```
[ ]: rb.log(name='labeling_with_finetuned', records=records)
```

5.10.11 6. Explore and label data with the fine-tuned model

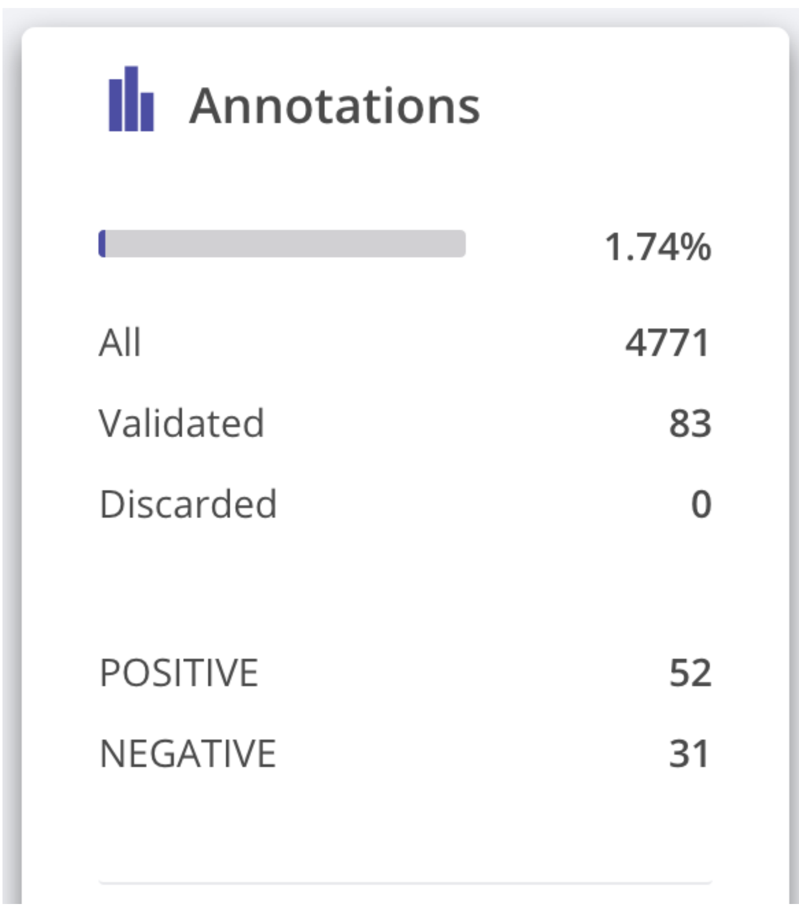
In this step, we'll start by exploring how the fine-tuned model is performing with our dataset.

At first sight, using the predicted as filter by `POSITIVE` and then by `NEGATIVE`, we see 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. Below you can see how to use this filter for labeling examples withing the range from 0 to 0.6.
2. Label examples predicted as `POSITIVE` by our fine-tuned model, and then predicted as `NEGATIVE` to correct the predictions.

After spending some minutes, we've labelled almost **2% of our raw dataset with around 80 annotated examples**, which is a small dataset but hopefully with highly informative examples.



5.10.12 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.

Add labeled examples to our previous training set

Let's add our new examples to our previous training set.

[11]:
```
def prepare_train_df(dataset_name):
    rb_df = rb.load(name=dataset_name)
    rb_df = rb_df[rb_df.status == "Validated"] ; len(rb_df)
    rb_df['text'] = rb_df.inputs.transform(lambda r: r['text'])
    rb_df['labels'] = rb_df.annotation.transform(lambda r: r[0])
    return rb_df
```

[12]:
```
df = prepare_train_df('labeling_with_finetuned') ; len(df)
```

[12]:
```
83
```

[13]:
```
train_dataset = train_dataset.remove_columns('__index_level_0__')
```

We'll use the .add_item method from the `datasets` library to add our examples:

[14]:
```
for i,r in df.iterrows():
    tokenization = tokenizer(r["text"], padding="max_length", truncation=True)
    train_dataset = train_dataset.add_item({
        "attention_mask": tokenization["attention_mask"],
        "input_ids": tokenization["input_ids"],
        "labels": label2id[r['labels']],
        "text": r['text'],
    })
```

[15]:
```
train_dataset
```

[15]:
```
Dataset({
    features: ['attention_mask', 'input_ids', 'labels', 'text'],
    num_rows: 266
})
```

Train 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)

[17]:
```
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", push_to_hub=True)
```

5.10.13 Wrap-up

In this tutorial, you've learnt 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 could 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 witth hand labeling, checkout the following tutorials:

- Active learning with modAL
- Weak supervision with Snorkel

5.10.14 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.

5.11 Building a news classifier with weak supervision

5.11.1 TL;DR

1. We build a news classifier using rules and weak supervision
2. For this example, we use the AG News dataset but you can follow this process to programatically label any dataset.
3. The train split without labels is used to build a training set with rules, Rubrix and Snorkel's Label model.
4. The test set is used for evaluating our weak labels, label model and downstream news classifier.
5. We achieve 0.81 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:

5.11.2 Setup Rubrix

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.

You can install Rubrix on your local machine, on a server, or using a cloud provider. If you have not installed and launched Rubrix, check the *Setup and Installation guide*.

Once installed, you only need to import Rubrix and some other libraries we'll be using for this tutorial:

[2]:
```
import rubrix as rb
from rubrix.labeling.text_classification import *

from datasets import load_dataset
import pandas as pd
```

5.11.3 1. Load test and unlabelled datasets into Rubrix

Let's load the test split from the `ag_news` dataset, which we'll be using for testing our label and downstream models.

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

labels = dataset.features["label"].names

records = [
    rb.TextClassificationRecord(
        inputs=record["text"],
        metadata={"split": "test"},
        annotation=labels[record["label"]]
    )
    for record in dataset
]

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

Let's load the train split from the `ag_news` dataset without labels. Our goal will be to programmatically build a training set using rules and weak supervision.

[ ]:
```
dataset = load_dataset("ag_news", split="train")

records = [
    rb.TextClassificationRecord(
        inputs=record["text"],
        metadata={"split": "unlabelled"},
    )
    for record in dataset
]

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 webapp for finding good rules for programmatic labeling.

5.11.4 2. Create rules and weak labels

Let's define some rules for each category, here you can use the expressive power of Elasticsearch's query string DSL.

[3]:
```
# 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")
]

rules = [
    Rule(query=term, label=label)
    for terms,label in queries
    for term in terms
]
```

[ ]:
```
weak_labels = WeakLabels(
    rules=rules,
    dataset="news"
)
```

It takes around 24 seconds to apply the rules and get the 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:

[5]:
```
weak_labels.summary()
```

[5]:
```
                                     polarity  coverage  overlaps  conflicts  \
money                              {Business}  0.008276  0.002437   0.001936
financ*                            {Business}  0.019655  0.005893   0.005188
dollar*                            {Business}  0.016591  0.003542   0.002908
war                                   {World}  0.011779  0.003213   0.001348
gov*                                  {World}  0.045078  0.010878   0.006270
minister*                             {World}  0.030031  0.007531   0.002821
conflict                              {World}  0.003041  0.001003   0.000102
footbal*                             {Sports}  0.013166  0.004945   0.000439
sport*                               {Sports}  0.021191  0.007045   0.001223
game                                 {Sports}  0.038879  0.014083   0.002375
play*                                {Sports}  0.052453  0.016889   0.005063
sci*                               {Sci/Tech}  0.016552  0.002735   0.001309
techno*                            {Sci/Tech}  0.027218  0.008433   0.003174
computer*                          {Sci/Tech}  0.027320  0.011058   0.004459
software                           {Sci/Tech}  0.030243  0.009655   0.003346
web                                {Sci/Tech}  0.015376  0.004067   0.001607
total      {Sci/Tech, Business, Sports, World}  0.317022  0.053582   0.019561

           correct  incorrect  precision
money           30         37   0.447761
```

(continues on next page)

(continued from previous page)

```
financ*         80         55   0.592593
dollar*         87         37   0.701613
war             75         26   0.742574
gov*           170        174   0.494186
minister*      193         22   0.897674
conflict        18          4   0.818182
footbal*       107          7   0.938596
sport*         139         23   0.858025
game           216         71   0.752613
play*          268        112   0.705263
sci*           114         26   0.814286
techno*        155         60   0.720930
computer*      159         54   0.746479
software       184         41   0.817778
web             76         25   0.752475
total         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 0.72**, our hope is that the label and downstream models will improve both the recall and the precision of the final classifier.

5.11.5 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 (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:

[6]:
```python
# If Snorkel is not installed on your machine !pip install snorkel

label_model = Snorkel(weak_labels)

# Fit Label Model
label_model.fit()

# Test with labeled test set
label_model.score()
```

```
WARNING:rubrix.labeling.text_classification.label_models:Metrics are only calculated␣
↪over non-abstained predictions!
```

[6]:
```
{'accuracy': 0.7448246725813266}
```

5.11.6 3. 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 Webapp, 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.

[20]:
```
records = label_model.predict()

# build a simple dataframe with text and the prediction with the highest score
df_train = pd.DataFrame([
    {"text": record.inputs["text"], "label": label_model.weak_labels.label2int[record.
↪prediction[0][0]]}
    for record in records
])
df_train
```

[20]:
```
                                                    text  label
0      Jan Baan launches Web services firm com Septem...      0
1      Molson Indy Vancouver gets black flag  quot;Th...      1
2      The football gods were on our side #39; Jason ...      1
3      Jags get offense clicking in second half Fred ...      1
4      Puzzle Over Low Galaxy Count Scientists from t...      0
...                                                  ...    ...
38080  Football legend Maradona rushed to hospital Fo...      1
38081  Head of British charity expelled from Sudan Th...      3
38082  From SANs to SATAs, storage vendors continue p...      0
38083  Billups Sits Out Because of Ankle Sprain (AP) ...      1
38084  Judge Rules for Oracle in PeopleSoft Bid (Reut...      0

[38085 rows x 2 columns]
```

[19]:
```
# for the test set, we can retrieve the records with validated annotations (the original␣
↪ag_news test set)
df_test = rb.load("news", query="status:Validated")

df_test['text'] = df_test.inputs.transform(lambda r: r['text'])
df_test['annotation'] = df_test['annotation'].apply(
    lambda r:label_model.weak_labels.label2int[r]
)
```

5.11.7 4. Train a downstream model with scikit-learn

Now, let's train our final model using `scikit-learn`

```
[ ]: from sklearn.feature_extraction.text import TfidfTransformer, CountVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.pipeline import Pipeline

classifier = Pipeline([
    ('vect', CountVectorizer()),
    ('clf', MultinomialNB())
])

classifier.fit(
    X=df_train.text.tolist(),
    y=df_train.label.values
)
```

```
[18]: accuracy = classifier.score(
    X=df_test.text.tolist(),
    y=label_model.weak_labels.annotation()
)

f"Test accuracy: {accuracy}"
```

```
[18]: 'Test accuracy: 0.8177631578947369'
```

Not too bad!

We have achieved around **0.81 accuracy** without even using a single example from the original `ag_news` train set and with a small set of rules (less than 30). Also, we've largely improved over the 0.74 accuracy of our Label Model.

Finally, let's take a look at more detailed metrics:

```
[82]: from sklearn import metrics

labels = list(label_model.weak_labels.label2int.keys())[1:] # removes "abstain" label
predicted = classifier.predict(df_test.text.tolist())

print(metrics.classification_report(label_model.weak_labels.annotation(), predicted,␣
↪target_names=labels))
```

```
              precision    recall  f1-score   support

    Sci/Tech       0.76      0.83      0.80      1900
      Sports       0.86      0.98      0.91      1900
    Business       0.89      0.56      0.69      1900
       World       0.79      0.89      0.84      1900

    accuracy                           0.82      7600
   macro avg       0.82      0.82      0.81      7600
weighted avg       0.82      0.82      0.81      7600
```

5.11.8 Next steps

If you are interested in the topic of weak supervision check the *Weak supervision guide*.

Rubrix documentation for more guides and tutorials.

Join the Rubrix community on Slack

Rubrix Github repo to stay updated.

5.12 Explore and analyze spaCy NER pipelines

In this tutorial, you'll 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.
- Improve your pipelines over time using Rubrix annotation mode.
- Monitor your model predictions using Rubrix integration with Kibana

Let's get started!

5.12.1 Introduction

In this tutorial we will:

- Load the Gutenberg Time dataset from the Hugging Face Hub.
- 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.
- 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.
- As a bonus, we will use Rubrix and spaCy on a more challenging dataset: IMDB.

5.12.2 Setup Rubrix

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, check the Setup and Installation guide.

Once installed, you only need to import Rubrix:

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

5.12.3 Install tutorial dependencies

In this tutorial, we'll use the `datasets` and `spaCy` libraries and the `en_core_web_trf` pretrained English model, a Roberta-based spaCy model . If you do not have them installed, run:

```
[ ]: %pip install torch datasets "spacy[transformers]~=3.0" protobuf -qqq
```

5.12.4 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")
```

Let's take a look at our dataset!

```
[ ]: train, test = dataset.train_test_split(test_size=0.002, seed=42).values() ; test
```

5.12.5 Logging spaCy NER entities into Rubrix

Using a Transformer-based pipeline

Let's install and load our roberta-based pretrained pipeline and apply it to one of our dataset records:

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

```
[ ]: import spacy

nlp = spacy.load("en_core_web_trf")
doc = nlp(dataset[0]["tok_context"])
doc
```

Now let's apply the nlp pipeline to our dataset records, collecting the tokens and NER entities.

```
[ ]: from tqdm.auto import tqdm

records = []

for record in tqdm(test, total=len(test)):
    # 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(
        rb.TokenClassificationRecord(
            text=text,
            tokens=tokens,
            prediction=entities,
            prediction_agent="en_core_web_trf",
        )
    )
```

[ ]:
```
records[0]
```

[ ]:
```
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.
- You can 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.
- You can use the free-text search to find records containing specific words.
- You could 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")
doc = nlp(dataset[0]["tok_context"])
```

[ ]:
```
records = []    # Creating and empty record list to save all the records

for record in tqdm(test, total=len(test)):

    text = record["tok_context"]  # We only need the text of each instance
    doc = nlp(text)    # spaCy Doc creation

    # Entity annotations
    entities = [
```

(continues on next page)

(continued from previous page)

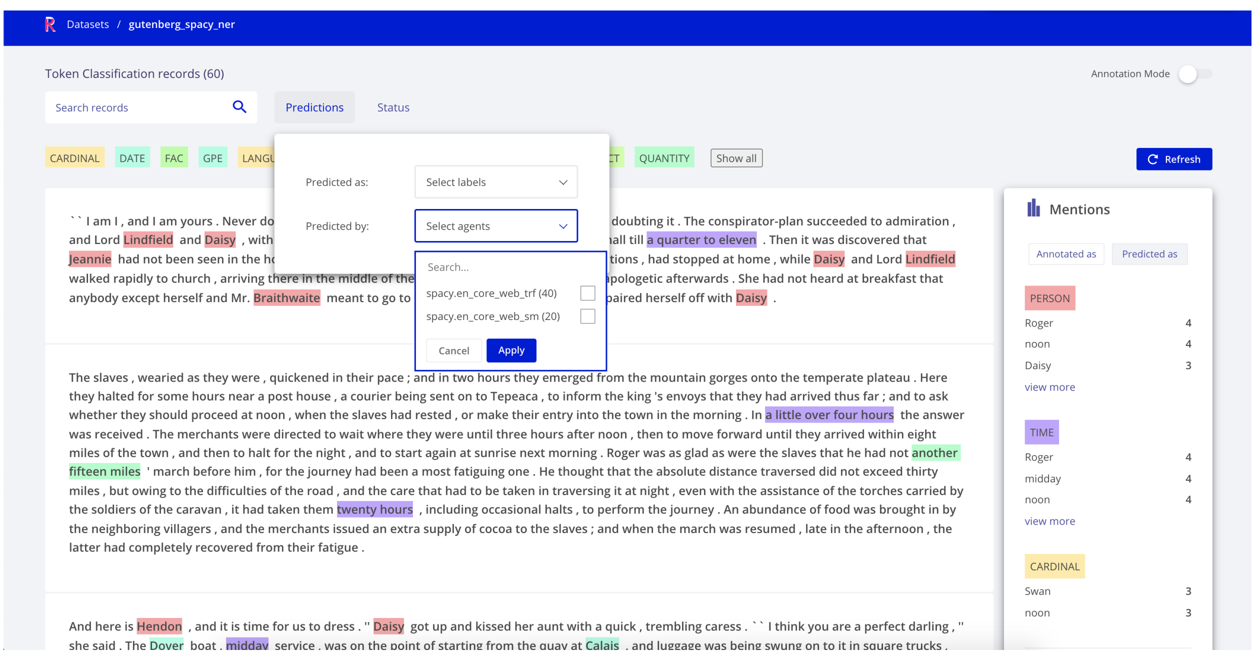
```
        (ent.label_, ent.start_char, ent.end_char)
        for ent in doc.ents
    ]

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


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

[ ]:
```
rb.log(records=records, name="gutenberg_spacy_ner")
```

5.12.6 Exploring and comparing `en_core_web_sm` and `en_core_web_trf` models

If you go to your `gutenberg_spacy_ner` you can explore and compare the results of both models.

You can use the `predicted by` filter, which comes from the `prediction_agent` parameter of your `TextClassificationRecord` to only see predictions of a specific model:



5.12.7 Extra: 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[0:5000]")
```

```
[ ]: records = []
for record in tqdm(imdb, total=len(imdb)):
    # 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(
        rb.TokenClassificationRecord(
            text=text,
            tokens=tokens,
            prediction=entities,
            prediction_agent="en_core_web_sm",
        )
    )
```

```
[ ]: rb.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 check yourself each an every example by using the filters and search-box.

5.12.8 Summary

In this tutorial, we have learnt to log and explore diffrent `spaCy` NER models with Rubrix. Using what we´ve learnt here you can:

- Build custom dashboards using Kibana to monitor and visualize spaCy models.
- Build training sets using pre-trained spaCy models.

5.12.9 Next steps

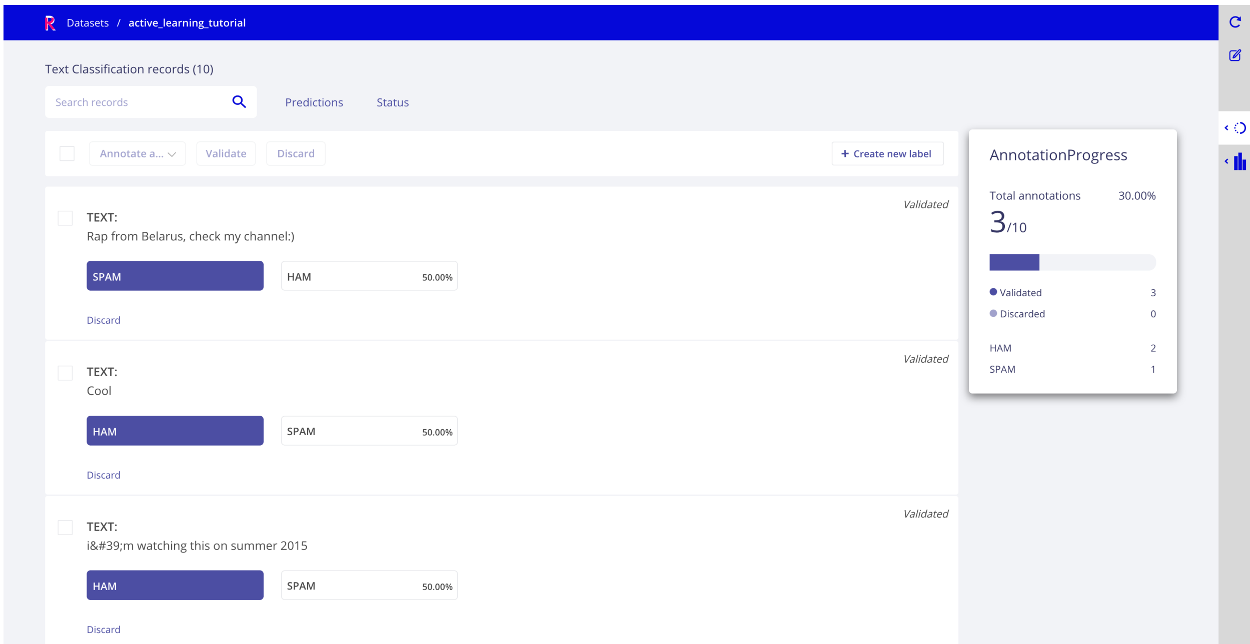
Rubrix documentation for more guides and tutorials.

Join the Rubrix community! A good place to start is the discussion forum.

Rubrix Github repo to stay updated.

5.13 Active learning with ModAL and scikit-learn

In this tutorial, we will walk through the process of building an active learning prototype with *Rubrix*, the active learning framework ModAL and scikit-learn



5.13.1 Introduction

Our goal is to show you how to incorporate Rubrix into interactive workflows involving a human in the loop. This 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. There are several great tools which focus on active learning, being Prodi.gy the most prominent.

What is active learning?

> *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]

This tutorial

In this tutorial, we will build a simple text classifier by combining scikit-learn, ModAL and *Rubrix*. Scikit-learn will provide the model that we 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.

The tutorial is organized into:

1. **Loading the data**: Quick look at the data
2. **Create the active learner**: Create the model and embed it in the active learner
3. **Active learning loop**: Annotate samples and teach the model

But first things first, let's install our extra dependencies and setup *Rubrix*.

5.13.2 Setup Rubrix

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, check the Setup and Installation guide.

Once installed, you only need to import Rubrix:

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

5.13.3 Setup

Install scikit-learn and ModAL

Apart from the two required dependencies we will also install matplotlib to plot our improvement for each active learning loop. However, this is of course optional and you can simply ignore this dependency.

[ ]:
```
%pip install modAL scikit-learn matplotlib -qqq
```

Imports

Let us import all the necessary stuff in the beginning.

[ ]:
```python
import rubrix as rb
import pandas as pd
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.exceptions import NotFittedError
from modAL.models import ActiveLearner
import matplotlib.pyplot as plt
```

5.13.4 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 data set which is a binary classification task for detecting spam comments in YouTube videos. Let's load the data and have a look at it.

```
[ ]: train_df = pd.read_csv("data/active_learning/train.csv")
     test_df = pd.read_csv("data/active_learning/test.csv")
```

```
[ ]: test_df
```

```
                                        COMMENT_ID  \
0            z120djlhizeksdulo23mj5z52vjmxlhrk04
1              z133ibkihkmaj3bfq22rilaxmp2yt54nb
2          z12gxdortqzwhhqas04cfjrwituzghb5tvk0k
3     _2viQ_Qnc6_ZYkMn1fS805Z6oy8ImeO6pSjMLAlwYfM
4          z120s1agtmmetler404cifqbxzvdx15idtw0k
..                                           ...
387        z13pup2w2k3rz1lxl04cf1a5qzavgvv51vg0k
388          z13psdarpuzbjp1hh04cjfwgzonextlhf1w
389        z131xnwierifxxkj204cgvjxyo3oydb42r40k
390          z12pwrxj0kfrwnxye04cjxtqntycd1yia44
391          z13oxvzqrzvyit00322jwtjo2tzqylhof04

                              AUTHOR                        DATE  \
0                Murlock Nightcrawler  2015-05-24T07:04:29.844000
1     Debora Favacho (Debora 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

                                              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<br />      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 data set to illustrate the effectiveness of the active learning approach with *Rubrix*. For the training data set we simply ignore the column and assume that we are gathering training data from scratch.

5.13.5 2. Defining our classifier and Active Learner

For 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).

[ ]:
```
# Define our classification model
classifier = MultinomialNB()
```

Then we define our active learner that uses the classifier as an estimator of the most uncertain predictions.

[ ]:
```
# 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.

[ ]:
```
# 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)
```

5.13.6 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
- and a variable to keep track of our improvements by recording the achieved accuracy after each iteration

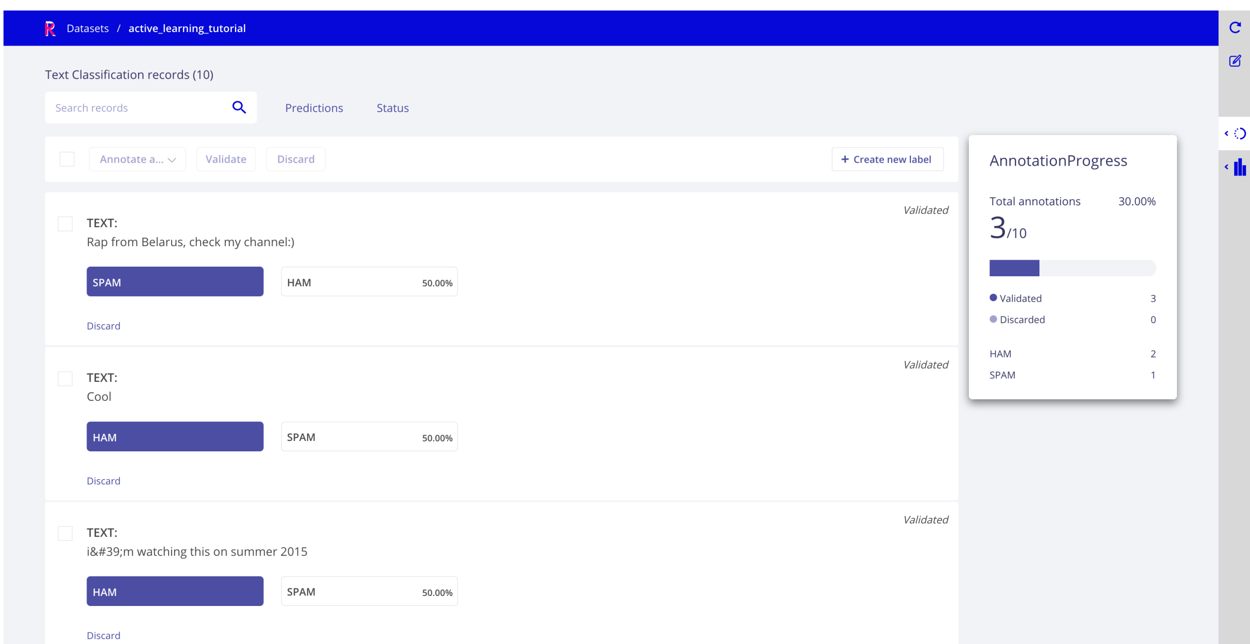
[ ]:
```
# Number of instances we want to annotate per iteration
n_instances = 10

# Accuracies after each iteration to keep track of our improvement
accuracies = []
```

1. Annotate samples

The first step of the training loop is about annotating *n* examples that have 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.

[ ]:
```python
# query examples from our training pool with the most uncertain prediction
query_idx, query_inst = learner.query(X_train, n_instances=n_instances)

# get predictions for the queried examples
try:
    probs = learner.predict_proba(X_train[query_idx])
# For the very first query we do not have any predictions
except NotFittedError:
    probs = [[0.5, 0.5]]*n_instances

# Build the Rubrix records
records = [
    rb.TextClassificationRecord(
        id=idx,
        inputs=train_df.CONTENT.iloc[idx],
        prediction=list(zip(["HAM", "SPAM"], [0.5, 0.5])),
        prediction_agent="MultinomialNB",
    )
    for idx in query_idx
]

# 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 upper right corner to update the view with respect to the filters.



Once you are done annotating the examples, you can continue with the active learning loop.

2. Teach the learner

The second step in the loop is to teach the learner. Once we trained our classifier with the newly annotated examples, we will apply the classifier to the test data and record the accuracy to keep track of our improvement.

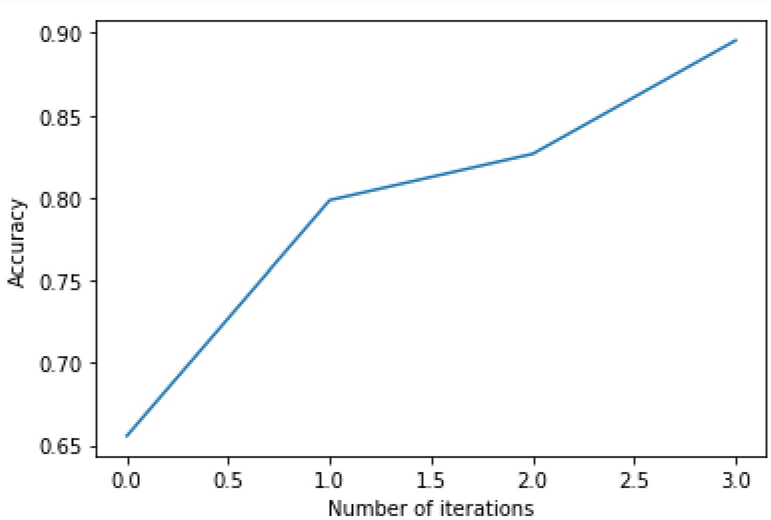
[ ]:
```
# Load the annotated records into a pandas DataFrame
records_df = rb.load("active_learning_tutorial")

# filter examples from the last annotation session
idx = records_df.id.isin(query_idx)

# check if all examples were annotated
if any(records_df[idx].annotation.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[idx].annotation.map(lambda x: int(x[0] == "SPAM"))
learner.teach(X=X_train[query_idx], y=y_train.to_list())

# 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.

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]:
```
# Plot the accuracy versus the iteration number
plt.plot(accuracies)
plt.xlabel("Number of iterations")
plt.ylabel("Accuracy");
```



5.13.7 Summary

In this tutorial we saw how to embed *Rubrix* in an active learning loop and how it can help you to gather a sample efficient data set by annotating only the most decisive examples. Here 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.

5.13.8 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.

5.13.9 Appendix: Compare query strategies, random vs max uncertainty

In this appendix 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 .

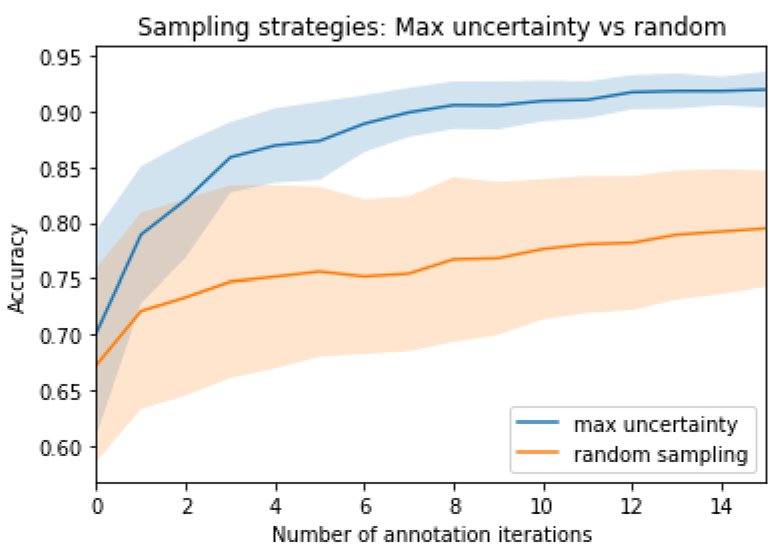
[ ]:
```
import numpy as np

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):
```
(continues on next page)

(continued from previous page)

```
        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)
plt.plot(range(n_iterations), arr_rnd.mean(0))
plt.fill_between(range(n_iterations), arr_rnd.mean(0)-arr_rnd.std(0), arr_rnd.
↪mean(0)+arr_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>
```



5.13.10 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/)
```

(continues on next page)

(continued from previous page)

```
    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/",
    ], 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)
```

5.14 Find label errors with cleanlab

In this tutorial, we will show you how you can find possible labeling errors in your data set with the help of cleanlab and *Rubrix*.

5.14.1 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. In the worst-case scenario, these label errors can destabilize benchmarks and tend to favor more complex models with a higher capacity over lower capacity models.

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.

This tutorial walks you through 5 basic steps to find and correct label errors in your data set:

1. Load the data set you want to check, and a model trained on it;
2. Make predictions for the test split of your data set;
3. Get label error candidates with *cleanlab*;
4. Uncover label errors with *Rubrix*;

5. Correct label errors and load the corrected data set;

5.14.2 Setup Rubrix

If you are new to Rubrix, visit and star Rubrix for updates: Github repository

If you have not installed and launched Rubrix, check the *Setup and Installation guide*.

Once installed, you only need to import Rubrix:

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

Install tutorial dependencies

Apart from cleanlab, we will also install the Hugging Face libraries transformers and datasets, as well as PyTorch, that provide us with the model and the data set we are going to investigate.

[2]:
```
%pip install cleanlab torch transformers datasets -qqq
```

Imports

Let us import all the necessary stuff in the beginning.

[1]:
```
import rubrix as rb
from cleanlab.pruning import get_noise_indices

import torch
import datasets
from transformers import AutoTokenizer, AutoModelForSequenceClassification
```

5.14.3 1. Load model and data set

For this tutorial we will use the well studied Microsoft Research Paraphrase Corpus (MRPC) data set that forms part of the GLUE benchmark, and a pre-trained model from the Hugging Face Hub that was fine-tuned on this specific data set.

Let us first get the model and its corresponding tokenizer to be able to make predictions. For a detailed guide on how to use the *transformers* library, please refer to their excellent documentation.

[ ]:
```
model_name = "textattack/roberta-base-MRPC"

tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForSequenceClassification.from_pretrained(model_name)
```

We then get the test split of the MRPC data set, that we will scan for label errors.

[ ]:
```
dataset = datasets.load_dataset("glue", "mrpc", split="test")
```

Let us have a quick look at the format of the data set. Label `1` means that both `sentence1` and `sentence2` are *semantically equivalent*, a `0` as label implies that the sentence pair is *not equivalent*.

[185]:
```
dataset.to_pandas().head()
```

[185]:
```
                                           sentence1  \
0  PCCW 's chief operating officer , Mike Butcher...
1  The world 's two largest automakers said their...
2  According to the federal Centers for Disease C...
3  A tropical storm rapidly developed in the Gulf...
4  The company didn 't detail the costs of the re...

                                           sentence2  label  idx
0  Current Chief Operating Officer Mike Butcher a...      1    0
1  Domestic sales at both GM and No. 2 Ford Motor...      1    1
2  The Centers for Disease Control and Prevention...      1    2
3  A tropical storm rapidly developed in the Gulf...      0    3
4  But company officials expect the costs of the ...      0    4
```

5.14.4 2. Make predictions

Now let us use the model to get predictions for our data set, and add those to our dataset instance. We will use the `.map` functionality of the *datasets* library to process our data batch-wise.

[ ]:
```
def get_model_predictions(batch):
    # batch is a dictionary of lists
    tokenized_input = tokenizer(
        batch["sentence1"], batch["sentence2"], padding=True, return_tensors="pt"
    )
    # get logits of the model prediction
    logits = model(**tokenized_input).logits
    # convert logits to probabilities
    probabilities = torch.softmax(logits, dim=1).detach().numpy()

    return {"probabilities": probabilities}

# Apply predictions batch-wise
dataset = dataset.map(
    get_model_predictions,
    batched=True,
    batch_size=16,
)
```

5.14.5 3. Get label error candidates

To identify label error candidates the cleanlab framework simply needs the probability matrix of our predictions (`n x m`, where `n` is the number of examples and `m` the number of labels), and the potentially noisy labels.

[154]:
```
# Output the data as numpy arrays
dataset.set_format("numpy")

# Get a boolean array of label error candidates
label_error_candidates = get_noise_indices(
    s=dataset["label"],
```

(continues on next page)

(continued from previous page)

```
    psx=dataset["probabilities"],
)
```

This one line of code provides us with a boolean array of label error candidates that we can investigate further. Out of the **1725 sentence pairs** present in the test data set we obtain **129 candidates** (7.5%) for possible label errors.

[164]:
```
frac = label_error_candidates.sum()/len(dataset)
print(
    f"Total: {len(dataset)}\n"
    f"Candidates: {label_error_candidates.sum()} ({100*frac:0.1f}%)"
)
```

```
Total: 1725
Candidates: 129 (7.5%)
```

5.14.6 4. Uncover label errors in Rubrix

Now that we have a list of potential candidates, let us log them to *Rubrix* to uncover and correct the label errors. First we switch to a pandas DataFrame to filter out our candidates.

[165]:
```
candidates = dataset.to_pandas()[label_error_candidates]
```

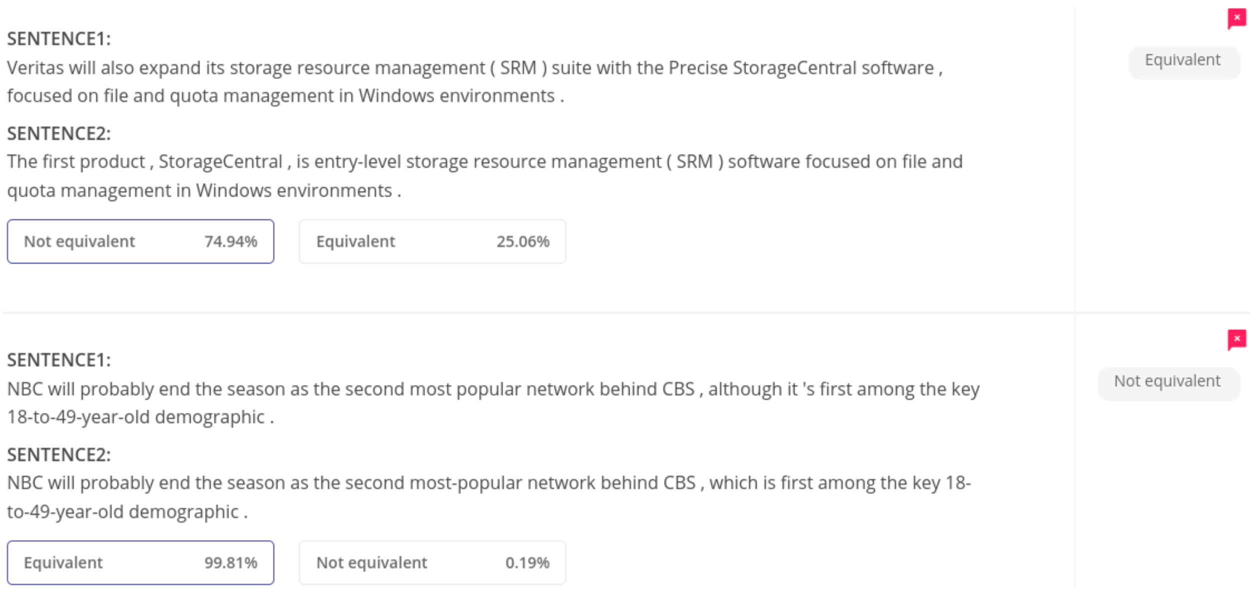
Then we will turn those candidates into *TextClassificationRecords* that we will log to *Rubrix*.

[166]:
```
def make_record(row):
    prediction = list(zip(["Not equivalent", "Equivalent"], row.probabilities))
    annotation = "Not equivalent"
    if row.label == 1:
        annotation = "Equivalent"

    return rb.TextClassificationRecord(
        inputs={"sentence1": row.sentence1, "sentence2": row.sentence2},
        prediction=prediction,
        prediction_agent="textattack/roberta-base-MRPC",
        annotation=annotation,
        annotation_agent="MRPC"
    )

records = candidates.apply(make_record, axis=1)
```

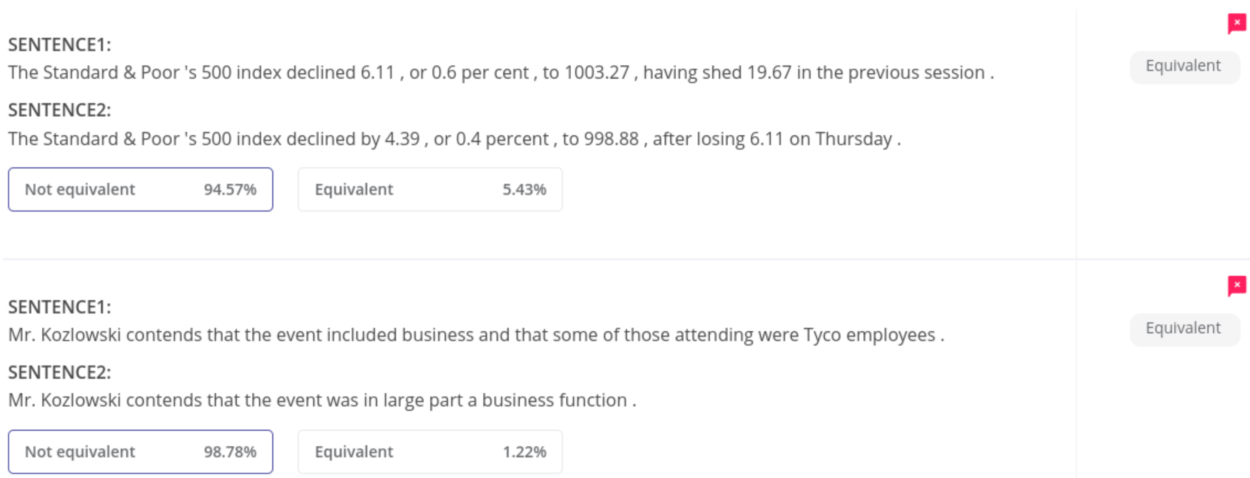
Having our records at hand we can now log them to *Rubrix* and save them in a dataset that we call `"mrpc_label_error"`.

[ ]:
```
rb.log(records, name="mrpc_label_error")
```

Scanning through the records in the *Explore Mode* of *Rubrix*, we were able to find at least **30 clear cases** of label errors. A couple of examples are shown below, in which the noisy labels are shown in the upper right corner of each example. The predictions of the model together with their probabilities are shown below each sentence pair.



If your model is not terribly over-fitted, you can also try to run the candidate search over your training data to find very obvious label errors. If we repeat the steps above on the training split of the MRPC data set (3668 examples), we obtain **9 candidates** (this low number is expected) out of which **5 examples** were clear cases of label errors. A couple of examples are shown below.



5.14.7 5. Correct label errors

With *Rubrix* it is very easy to correct those label errors. Just switch on the *Annotation Mode*, correct the noisy labels and load the dataset back into your notebook.

[181]:
```python
# Load the dataset into a pandas DataFrame
dataset_with_corrected_labels = rb.load("mrpc_label_error")

dataset_with_corrected_labels.head()
```

[181]:
```
                                              inputs  \
0  {'sentence1': 'Deaths in rollover crashes acco...
1  {'sentence1': 'Mr. Kozlowski contends that the...
2  {'sentence1': 'Larger rivals , including Tesco...
```

(continues on next page)

(continued from previous page)

```
3  {'sentence1': 'The Standard & Poor 's 500 inde...
4  {'sentence1': 'Defense lawyers had said a chan...

                                          prediction        annotation  \
0  [(Equivalent, 0.9751904606819153), (Not equiva...  [Not equivalent]
1  [(Not equivalent, 0.9878258109092712), (Equiva...      [Equivalent]
2  [(Equivalent, 0.986499547958374), (Not equival...  [Not equivalent]
3  [(Not equivalent, 0.9457013010978699), (Equiva...      [Equivalent]
4  [(Equivalent, 0.9974484443664551), (Not equiva...  [Not equivalent]

                prediction_agent annotation_agent  multi_label explanation  \
0  textattack/roberta-base-MRPC             MRPC        False        None
1  textattack/roberta-base-MRPC             MRPC        False        None
2  textattack/roberta-base-MRPC             MRPC        False        None
3  textattack/roberta-base-MRPC             MRPC        False        None
4  textattack/roberta-base-MRPC             MRPC        False        None

                                     id metadata     status event_timestamp
0  bad3f616-46e3-43ca-8ba3-f2370d421fd2       {}  Validated            None
1  50ca41c9-a147-411f-8682-1e3880a522f9       {}  Validated            None
2  6c06250f-7953-475a-934f-7eb35fc9dc4d       {}  Validated            None
3  39f37fcc-ac22-4871-90f1-3766cf73f575       {}  Validated            None
4  080c6d5c-46de-4670-9e0a-98e0c7592b11       {}  Validated            None
```

Now you can use the corrected data set to repeat your benchmarks and measure your model's "real-word performance" you care about in practice.

5.14.8 Summary

In this tutorial we saw how to leverage *cleanlab* and *Rubrix* to uncover label errors in your data set. 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 turns out to beat your resource hungry super model, and the deployment process just got a little bit easier .

Cleanlab and *Rubrix* do not care about the model architecture or the framework you are working with. They just care about the underlying data and allow you to put more humans in the loop of your AI Lifecycle.

5.14.9 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.

5.15 Zero-shot Named Entity Recognition with Flair

5.15.1 TL;DR:

You can use Rubrix for analizing and validating the NER predictions from the new zero-shot model provided by the Flair NLP library.

This is useful for quickly bootstrapping a training set (using Rubrix *Annotation Mode*) as well as integrating with weak-supervision workflows.

5.15.2 Install dependencies

[ ]:
```
%pip install datasets flair -qqq
```

5.15.3 Setup Rubrix

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*.

Once installed, you only need to import Rubrix:

[1]:
```
import rubrix as rb
```

5.15.4 Load the `wnut_17` dataset

In this example, we'll use a challenging NER dataset, the "WNUT 17: Emerging and Rare entity recognition" dataset, 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

dataset = load_dataset("wnut_17", split="test")
```

[7]:
```
wnut_labels = ['corporation', 'creative-work', 'group', 'location', 'person', 'product']
```

5.15.5 Configure Flair TARSTagger

Now let's configure our NER model, following Flair's documentation.

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

# Load zero-shot NER tagger
tars = TARSTagger.load('tars-ner')

# Define labels for named entities using wnut labels
labels = wnut_labels
tars.add_and_switch_to_new_task('task 1', labels, label_type='ner')
```

Let's test it with one example!

[9]:
```
sentence = Sentence(" ".join(dataset[0]['tokens']))
```

[10]:
```
tars.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("ner")
]
prediction
```

[10]:
```
[('location', 100, 107)]
```

5.15.6 Predict over wnut_17 and log into rubrix

Now, let's log the predictions in rubrix

[ ]:
```
records = []
for record in dataset.select(range(100)):
    input_text = " ".join(record["tokens"])

    sentence = Sentence(input_text)
    tars.predict(sentence)
    prediction = [
        (entity.get_labels()[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",
        )
    )
```

(continues on next page)

(continued from previous page)

```
rb.log(records, name='tars_ner_wnut_17', metadata={"split": "test"})
```

5.16 Clean labels using your model loss

5.16.1 TL;DR

1. A simple technique for error analysis is introduced: **using model loss to find potential training data errors**.
2. The technique is shown using a **fine-tuned text classifier from the Hugging Face Hub** on the **AG News dataset**.
3. Using Rubrix, **we verify more than 100 mislabelled examples on the training set** of this well-known NLP benchmark.
4. This trick is useful during **model training with small and noisy datasets**.
5. This trick is complementary with other "data-centric" ML methods such as `cleanlab` (see the Rubrix tutorial on cleanlab).

5.16.2 Introduction

This tutorial explains a simple trick for finding potential errors in training data: *using your model loss to identify label errors or ambiguous examples*. This trick is inspired by the following tweet:

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 very simple 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 10 minutes, we manually check and relabel the first 100 examples. In fact, the first 100 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.

5.16.3 Ingredients

- A model fine-tuned with the AG News dataset (you could train your own model if you wish).
- The AG News train split (the same trick could and should be applied to validation and test splits).
- Rubrix for logging, exploring, and relabeling wrong examples.

5.16.4 Steps

1. Load the fine-tuned model and the AG News train split.
2. Compute the loss for each example and sort examples by descending loss.
3. Log the first 500 example into a Rubrix dataset. We provide you with the processed dataset if you want to skip the first two steps.
4. Use Rubrix webapp for inspecting the examples ordered by loss. In the following video, we show you the full process for manually correcting the first 100 examples (all incorrect in the original dataset, the original video is 8 minutes long):

5.16.5 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 100 label errors on a widely-used benchmark in less than 10 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 accross the whole ML life-cyle** and not only for finding label errors. With this trick you can improve data preprocessing, tokenization, and even your model architecture.

5.16.6 Setup Rubrix

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*.

Once installed, you only need to import Rubrix:

[3]:
```
import rubrix as rb
```

5.16.7 Tutorial dependencies

We'll install the Hugging Face libraries transformers and datasets, as well as PyTorch, for the model and data set we'll use in the next steps.

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

5.16.8 1. Load the fine-tuned model and the training dataset

[ ]:
```
import torch

from transformers import AutoTokenizer, AutoModelForSequenceClassification
from transformers.data.data_collator import DataCollatorWithPadding

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

[ ]:
```
# load model and tokenizer
tokenizer = AutoTokenizer.from_pretrained("andi611/distilbert-base-uncased-ner-agnews")
model = AutoModelForSequenceClassification.from_pretrained("andi611/distilbert-base-
↪uncased-ner-agnews")

# load the training split
from datasets import load_dataset
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)
```

5.16.9 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.

[ ]:
```
# 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 = batch["input_ids"].to(device)
    attention_mask = batch["attention_mask"].to(device)
    labels = batch["labels"].to(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
```

(continues on next page)

(continued from previous page)

```
    # datasets complains with numpy dtypes, let's use Python lists
    for k, v in batch.items():
        batch[k] = v.cpu().numpy().tolist()

    return batch
```

[ ]:
```
import pandas as pd

losses_ds = ds_enc.remove_columns("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 &lt;b&gt;...&lt;/b&gt; 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.

5.16.10 3. Log high loss examples into Rubrix

Using the amazing Hugging Face Hub we've shared the resulting dataset, which you can find here.

[7]:
```
# if you have skipped the first two steps you can load the dataset here:
#losses_df = pd.read_json("agnews_train_loss.jsonl", lines=True, orient="records")
```

[ ]:
```
# creates a Text classification record for logging into Rubrix
def make_record(row):

    return rb.TextClassificationRecord(
        inputs={"text": row.text},
        # this is the "gold" label in the original dataset
        annotation=[(ds_enc.features['label'].names[row.label])],
        # this is the prediction together with its probability
        prediction=[(ds_enc.features['label'].names[row.predicted_label], row.predicted_
↪probas[row.predicted_label])],
        # metadata fields can be used for sorting and filtering, here we log the loss
        metadata={"loss": row.loss},
        # who makes the prediction
        prediction_agent="andi611/distilbert-base-uncased-agnews",
        # source of the gold label
        annotation_agent="ag_news_benchmark"
    )
```

[ ]:
```
# if you want to log the full dataset remove the indexing
top_losses = losses_df.sort_values("loss", ascending=False)[0:499]

```

(continues on next page)

(continued from previous page)

```
# build Rubrix records
records = top_losses.apply(make_record, axis=1)
```

[ ]:
```
rb.log(records, name="ag_news_error_analysis")
```

5.16.11 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 100 examples. You can watch the full session (with high-speed during the last part) here. In the video we use Rubrix annotation mode to change the label of mislabelled examples (the first label correspond to the original "gold" label and the second corresponds to the predictions of the model).

We've also randomly checked the next 400 examples finding many potential errors. If you are interested you can repeat our experiment or even help validate the next 100 examples, we'd love to know about your results! We plan to share the 100 relabeled examples with the community in the Hugging Face Hub.

5.16.12 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 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.

[ ]:
```

```

5.17 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.

This tutorial walks 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

Let's get started!

5.17.1 Setup Rubrix

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*.

Once installed, you only need to import Rubrix:

5.17.2 Install tutorial dependencies

Apart from Rubrix, we'll need the following libraries: - `transformers` - `spaCy` - `uvicorn` - `FastAPI`

And the following models: - `distilbert-base-uncased-finetuned-sst-2-english` : a sentiment-analysis model - `en_core_web_sm` : spaCy's trained pipeline for English

To install all requirements, run the following commands :

```
[ ]: # spaCy
!pip install spacy
# spaCy pipeline
!python -m spacy download en_core_web_sm
# FastAPI
!pip install fastapi
# transformers
!pip install transformers
# uvicorn
!pip install uvicorn[standard]
```

The transformer's pipeline will be downloaded in the next step.

5.17.3 Loading models

Let's get and load our model pretrained pipeline and apply it to one of our dataset records:

```
[ ]: from transformers import pipeline
import spacy

transformers_pipeline = pipeline("sentiment-analysis", return_all_scores=True)
spacy_pipeline = spacy.load("en_core_web_sm")
```

For more informations 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:

[ ]:
```
from pprint import pprint

batch = ['I really like rubrix!']
predictions = transformers_pipeline(batch)
pprint(predictions)
```

Looks like the `predictions` is a list containing lists of two elements : - The first dictionnary containing the `NEGATIVE` sentiment label and its score. - The second dictionnary containing the same data but for `POSITIVE` sentiment.

5.17.4 Convert output to Rubrix format

To log the output to rubrix we should supply a list of dictionnaries, each dictonnary 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.

[ ]:
```
rubrix_format = [
    {
        "labels": [p["label"] for p in prediction],
        "scores": [p["score"] for p in prediction],
    }
    for prediction in predictions
]
pprint(rubrix_format)
```

5.17.5 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
    ]
```

5.17.6 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,
)
```

5.17.7 Do the same for spaCy

We'll add a custom mapper to convert spaCy's output to `TokenClassificationRecord` format

Mapper

[ ]:
```
import re
import datetime

from rubrix.client.models import TokenClassificationRecord

def custom_mapper(inputs, outputs):
    spaces_regex = re.compile(r"\s+")
    text = inputs
    return TokenClassificationRecord(
            text=text,
            tokens=spaces_regex.split(text),
            prediction=[
                    (entity["label"], entity["start"], entity["end"])
                    for entity in (
                            outputs.get("entities") if isinstance(outputs, dict) else␣
↪outputs
                    )
            ],
            event_timestamp=datetime.datetime.now(),
    )
```

FastAPI application

[ ]:
```
app_spacy = FastAPI()

app_spacy.add_middleware(
    RubrixLogHTTPMiddleware,
    api_endpoint="/spacy/",
    dataset="monitoring_spacy",
    records_mapper=custom_mapper
)
```

(continues on next page)

(continued from previous page)

```
# 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
```

5.17.8 Putting it all together

[ ]:
```
app = FastAPI()

@app.get("/")
def root():
    return {"message": "alive"}

app.mount("/transformers", app_transformers)
app.mount("/spacy", app_spacy)
```

Launch the appplication

To launch the application, copy the whole code into a file named `main.py` and run the following command:

[ ]:
```
!uvicorn main:app
```

5.17.9 Transformers demo

5.17.10 spaCy demo

5.17.11 Summary

In this tutorial, we have learnt to automatically log model outputs into Rubrix, this can be used to continuously and transparently monitor HTTP inference endpoints.

5.17.12 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.

5.18 Faster data annotation with a zero-shot text classifier

5.18.1 TL;DR

1. A simple example for data annotation with Rubrix is shown: **using a zero-shot classification model to pre-annotate and hand-label data more efficiently**.
2. We use the new **SELECTRA zero-shot classifier** and the Spanish part of the **MLSum**, a multilingual dataset for text summarization.
3. Two data annotation rounds are performed: (1) **labeling random examples**, and (2) **bulk labeling high score examples**.
4. Besides boosting the labeling process, this workflow lets you **evaluate the performance of zero-shot classification for a specific use case**. In this example use case, we observe the pre-trained zero-shot classifier provides pretty decent results, which might be enough for general news categorization.

5.18.2 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 zeroshot 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.

5.18.3 Setup Rubrix

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*.

Once installed, you only need to import Rubrix:

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

5.18.4 Install dependencies

For this tutorial we only need to install a few additional dependencies:

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

5.18.5 1. Load the Spanish zero-shot classifier: `Selectra`

We will use the recently released SELECTRA zero-shot classifier model, a zero-shot classifier for Spanish language.

```
[ ]: from transformers import pipeline

classifier = pipeline("zero-shot-classification",
                      model="Recognai/zeroshot_selectra_medium")
```

5.18.6 2. Loading the `MLSum` dataset

MLSUM, is a large scale multilingual text summarization dataset. Obtained from online newspapers, it contains 1.5M+ article/summary pairs in five different languages – namely, French, German, Spanish, Russian and Turkish. To illustrate the labeling process, in this tutorial we will only use the first 500 examples of its Spanish test set.

```
[ ]: from datasets import load_dataset

mlsum = load_dataset("mlsum", "es", split="test[0:500]")
```

5.18.7 3. Making zero-shot predictions

The zero-shot classifier allows you to provide arbitrary candidate labels, which it will use for its predictions. Since under the hood, this zero-shot classifier is based on natural language inference (NLI), we need to convert the candidate labels into a "hypothesis". For this we use a *hypothesis_template*, in which the `{}` will be replaced by each one of our candidate label. This template can have a big effect on the scores of your predictions and should be adopted to your use case.

```
[ ]: # We adopted the hypothesis to our use case of predicting the topic of news articles
hypothesis_template = "Esta noticia habla de {}."
# The candidate labels for our zero-shot classifier
candidate_labels = ["política", "cultura", "sociedad", "economia", "deportes", "ciencia␣
↪y tecnología"]
```

(continues on next page)

(continued from previous page)

```python
# Make predictions batch-wise
def make_prediction(rows):
    predictions = classifier(
        rows["summary"],
        candidate_labels=candidate_labels,
        hypothesis_template=hypothesis_template
    )
    return {key: [pred[key] for pred in predictions] for key in predictions[0]}

mlsum_with_predictions = mlsum.map(make_prediction, batched=True, batch_size=8)
```

5.18.8 4. Logging predictions in Rubrix

Let us log the examples to Rubrix and start our hand-labeling session, which will hopefully become more efficient with the zero-shot predictions.

[ ]:
```python
records = []

for row in mlsum_with_predictions:
    records.append(
        rb.TextClassificationRecord(
            inputs=row["summary"],
            prediction=list(zip(row['labels'], row['scores'])),
            prediction_agent="zeroshot_selectra_medium",
            metadata={"topic": row["topic"]}
        )
    )
```

[ ]:
```python
rb.log(records, name="zeroshot_noticias", metadata={"tags": "data-annotation"})
```

5.18.9 5. Hand-labeling session

Let's do two data annotation sessions.

Label first 20 random examples

Labeling random or sequential examples is always recommended to get a sense of the data distribution, the usefulness of zero-shot predictions, and the suitability of the labeling scheme (the target labels). Typically, this is how you will build your first test set, which you can then use to validate the downstream supervised model.

Label records with high score predictions

In this case, we will use bulk-labeling (labeling a set of records with a few clicks) after quickly reviewing high score predictions from our zero-shot model. The main idea is that above a certain score, the predictions from this model are more likely to be correct.

5.18.10 Next steps

If you are interested in the topic of zero-shot models, check out the tutorial for using *Rubrix with Flair's zero-shot NER*.

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.

[ ]:
```
```

5.19 Python

The python reference guide for Rubrix. This section contains:

- *Client*: The base client module
- *Metrics (Experimental)*: The module for dataset metrics
- *Labeling (Experimental)*: A toolbox to enhance your labeling workflow (weak labels, noisy labels, etc.)

5.19.1 Client

Here we describe the Python client of Rubrix that we divide into two basic modules:

- Methods: These methods make up the interface to interact with Rubrix's REST API.
- Models: You need to wrap your data in these data models for Rubrix to understand it.

Methods

This module contains the interface to access Rubrix's REST API.

`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`** (*`Optional[str]`*) – If provided, dataset will be copied to that workspace

Examples

```
>>> import rubrix as rb
>>> rb.copy("my_dataset", name_of_copy="new_dataset")
>>> dataframe = rb.load("new_dataset")
```

`rubrix.delete`(*name*)

Delete a dataset.

Parameters **name** (*str*) – The dataset name.

Return type None

Examples

```
>>> import rubrix as rb
>>> rb.delete(name="example-dataset")
```

`rubrix.get_workspace`()

Returns the name of the active workspace for the current client session.

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*)

Init the python client.

Passing an api_url disables environment variable reading, which will provide default values.

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]*) – Authentification 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.

Return type None

Examples

```
>>> import rubrix as rb
>>> rb.init(api_url="http://localhost:9090", api_key="4AkeAPIk3Y")
```

`rubrix.load`(*name*, *query=None*, *ids=None*, *limit=None*, *as_pandas=True*)

Loads a dataset as a pandas DataFrame or a list of records.

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, load dataset records with given ids.
- **limit** (*Optional[int]*) – The number of records to retrieve.
- **as_pandas** (*bool*) – If True, return a pandas DataFrame. If False, return a list of records.

Returns The dataset as a pandas Dataframe or a list of records.

Return type Union[pandas.core.frame.DataFrame, List[Union[*rubrix.client.models.TextClassificationRecord*, *rubrix.client.models.TokenClassificationRecord*, *rubrix.client.models.Text2TextRecord*]]]

Examples

```
>>> import rubrix as rb
>>> dataframe = rb.load(name="example-dataset")
```

rubrix.**log**(*records*, *name*, *tags=None*, *metadata=None*, *chunk_size=500*, *verbose=True*)

Log Records to Rubrix.

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]]]*) – The record or an iterable of records.
- **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.

Returns Summary of the response from the REST API

Return type *rubrix.client.models.BulkResponse*

Examples

```
>>> import rubrix as rb
>>> record = rb.TextClassificationRecord(
...     inputs={"text": "my first rubrix example"},
...     prediction=[('spam', 0.8), ('ham', 0.2)]
... )
>>> response = rb.log(record, name="example-dataset")
```

rubrix.**set_workspace**(*ws*)

Sets the active workspace for the current client session.

Parameters **ws** (*str*) – The new workspace

Return type None

Models

This module contains the data models for the interface

class rubrix.client.models.**BulkResponse**(*, *dataset*, *processed*, *failed=0*)
Summary response when logging records to the Rubrix server.

Parameters

- **dataset** (*str*) – The dataset name.
- **processed** (*int*) – Number of records in bulk.
- **failed** (*Optional[int]*) – Number of failed records.

Return type None

class rubrix.client.models.**Text2TextRecord**(**args*, *text*, *prediction=None*, *annotation=None*, *prediction_agent=None*, *annotation_agent=None*, *id=None*, *metadata=None*, *status=None*, *event_timestamp=None*, *metrics=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.
- **annotation** (*Optional[str]*) – A string representing the expected output text for the given input text.
- **prediction_agent** (*Optional[str]*) – Name of the prediction agent. By default, this is set to the hostname of your machine.
- **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** (*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*.

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**(**args*, *inputs*, *prediction=None*, *annotation=None*, *prediction_agent=None*, *annotation_agent=None*, *multi_label=False*, *explanation=None*, *id=None*, *metadata=None*, *status=None*, *event_timestamp=None*, *metrics=None*)

Record for text classification

Parameters

- **inputs** (*Union[str, List[str], Dict[str, Union[str, List[str]]]]*) – The inputs of the record
- **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.
- **annotation** (*Optional[Union[str, List[str]]]*) – A string or a list of strings (multilabel) corresponding to the annotation (gold label) for the record.
- **prediction_agent** (*Optional[str]*) – Name of the prediction agent. By default, this is set to the hostname of your machine.
- **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** (*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*.

Return type None

Examples

```
>>> import rubrix as rb
>>> record = rb.TextClassificationRecord(
...     inputs={"text": "my first rubrix example"},
...     prediction=[('spam', 0.8), ('ham', 0.2)]
... )
```

classmethod input_as_dict(*inputs*)
Preprocess record inputs and wraps as dictionary if needed

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**(**args*, *text*, *tokens*, *prediction=None*, *annotation=None*, *prediction_agent=None*, *annotation_agent=None*, *id=None*, *metadata=None*, *status=None*, *event_timestamp=None*, *metrics=None*)
Record for a token classification task

Parameters

- **text** (*str*) – The input of the record
- **tokens** (*List[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, 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. EXPERIMENTAL: The fourth entry is optional and corresponds to the score of the entity.
- **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.
- **prediction_agent** (*Optional[str]*) – Name of the prediction agent. By default, this is set to the hostname of your machine.
- **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** (*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*.

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)]
... )
```

5.19.2 Metrics (Experimental)

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/reference/rubrix_webapp_reference.html#search-input)

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/reference/rubrix_webapp_reference.html#search-input)

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 charactes in entity mention are upper case

`LOWER`: All charactes in entity mention are lower case

`FIRST`: The mention is capitalized

`MIDDLE`: Some character in mention between first and last is capitalized

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=10, 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 `mention_length/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 performants

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=10)
>>> 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 f1
>>> summary = f1(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.`**`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
```

5.19.3 Labeling (Experimental)

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*)

A rule (labeling function) in form of an ElasticSearch query.

Parameters

- **query** (*str*) – An ElasticSearch query with the query string syntax.
- **label** (*str*) – The label associated to the query.

- **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.

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", as_pandas=False)
>>> 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 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[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 name
The name of the rule.

exception rubrix.labeling.text_classification.weak_labels.**DuplicatedRuleNameError**

class rubrix.labeling.text_classification.weak_labels.**WeakLabels**(*rules*, *dataset*, *ids=None*, *query=None*, *label2int=None*)
Computes the weak labels of a dataset by applying a given list of rules.

Parameters

- **rules** (*List[Callable]*) – A list of rules (labeling functions). They must return a string, or None in case of abstention.
- **dataset** (*str*) – Name of the dataset to which the rules will be applied.
- **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

- *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 and a summary of the applied rules:

```
>>> def awesome_rule(record: TextClassificationRecord) -> str:
...     return "Positive" if "awesome" in record.inputs["text"] else None
>>> another_rule = Rule(query="good OR best", label="Positive")
>>> weak_labels = WeakLabels(rules=[awesome_rule, another_rule], dataset="my_dataset
↪")
>>> weak_labels.matrix()
>>> weak_labels.summary()
```

Use snorkel's LabelModel:

```
>>> from snorkel.labeling.model import LabelModel
>>> label_model = LabelModel()
>>> label_model.fit(L_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))
```

annotation(*exclude_missing_annotations=True*)

Returns the annotation labels as an array of integers.

Parameters **exclude_missing_annotations** (*bool*) – If True, excludes missing annotations, that is all entries with the `self.label2int[None]` integer.

Returns The annotation array of integers.

Return type numpy.ndarray

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.

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.

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

`records`(*has_annotation=None*)

Returns the records corresponding to the weak label matrix.

Parameters `has_annotation` (*Optional[bool]*) – If True, return only the records that have an annotation. If False, return only the records that have NO annotation. By default, we return all the records.

Returns A list of records, or optionally just a part of them.

Return type List[*rubrix.client.models.TextClassificationRecord*]

`property rules: List[Callable]`

The rules (labeling functions) that were used to produce the weak labels.

`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:

- **polarity**: Set of unique labels returned by the rule, excluding "None" (abstain).
- **coverage**: Fraction of the records labeled by the rule.
- **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 records the rule labeled correctly (if annotations are available).
- **incorrect**: Number of records the rule labels 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(exclude_missing_annotations=False)`.

Returns The summary statistics for each rule in a pandas DataFrame.

Return type pandas.core.frame.DataFrame

class rubrix.labeling.text_classification.label_models.**LabelModel**(*weak_labels*)
Abstract base class for a label model implementation.

Parameters **weak_labels** (rubrix.labeling.text_classification.weak_labels.WeakLabels) – Every label model implementation needs at least a *WeakLabels* instance.

fit(*include_annotated_records=False*, **args*, ***kwargs*)
Fits the label model.

Parameters **include_annotated_records** (*bool*) – Whether or not to include annotated records in the training.

predict(*include_annotated_records=False*, *include_abstentions=False*, ***kwargs*)
Applies the label model.

Parameters

- **include_annotated_records** (*bool*) – Whether or not to include annotated records.
- **include_abstentions** (*bool*) – Whether or not to include records in the output, for which the label model abstained.

Returns A list of records that include the predictions of the label model.

Return type List[*rubrix.client.models.TextClassificationRecord*]

score(**args*, ***kwargs*)
Evaluates the label model.

Return type Dict

property weak_labels: rubrix.labeling.text_classification.weak_labels.WeakLabels
The underlying *WeakLabels* object, containing the weak labels and records.

class rubrix.labeling.text_classification.label_models.**Snorkel**(*weak_labels*, *verbose=True*, *device='cpu'*)
The label model by Snorkel.

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 Rule, WeakLabels
>>> rule = Rule(query="good OR best", label="Positive")
>>> weak_labels = WeakLabels(rules=[rule], 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 or not to include annotated records in the training.
- ****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*, *tie_break_policy='abstain'*)
Returns a list of records that contain the predictions of the label model

Parameters

- **include_annotated_records** (*bool*) – Whether or not to include annotated records.
- **include_abstentions** (*bool*) – Whether or not to include records in the output, for which the label model abstained.
- **tie_break_policy** (*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 of the labeling functions abstained.

Returns A list of records that include the predictions of the label model.

Return type List[*rubrix.client.models.TextClassificationRecord*]

score(*tie_break_policy='abstain'*)
Returns some scores of the label model with respect to the annotated records.

Parameters **tie_break_policy** – 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 of the labeling functions abstained.

Returns A list of records that include the predictions of the label model.

Raises **MissingAnnotationError** – If the weak_labels do not contain annotated records.

Return type Dict[str, float]

5.20 Web App UI

This section contains a quick overview of Rubrix web-app's User Interface (UI).

The web-app has two main pages: the **Home** page and the **Dataset** page.

5.20.1 Home page

The **Home page** is the entry point to Rubrix Datasets. It's a searchable and sortable list of datasets with the following attributes:

- **Name**
- **Tags**, which displays the `tags` passed to the `rubrix.log` method. Tags are useful to organize your datasets by project, model, status and any other dataset attribute you can think of.
- **Task**, which is defined by the type of `Records` logged into the dataset.
- **Created at**, which corresponds to the timestamp of the Dataset creation. Datasets in Rubrix are created by directly using `rb.log` to log a collection of records.
- **Updated at**, which corresponds to the timestamp of the last update to this dataset, either by adding/changing/removing some annotations with the UI or via the Python client or the REST API.

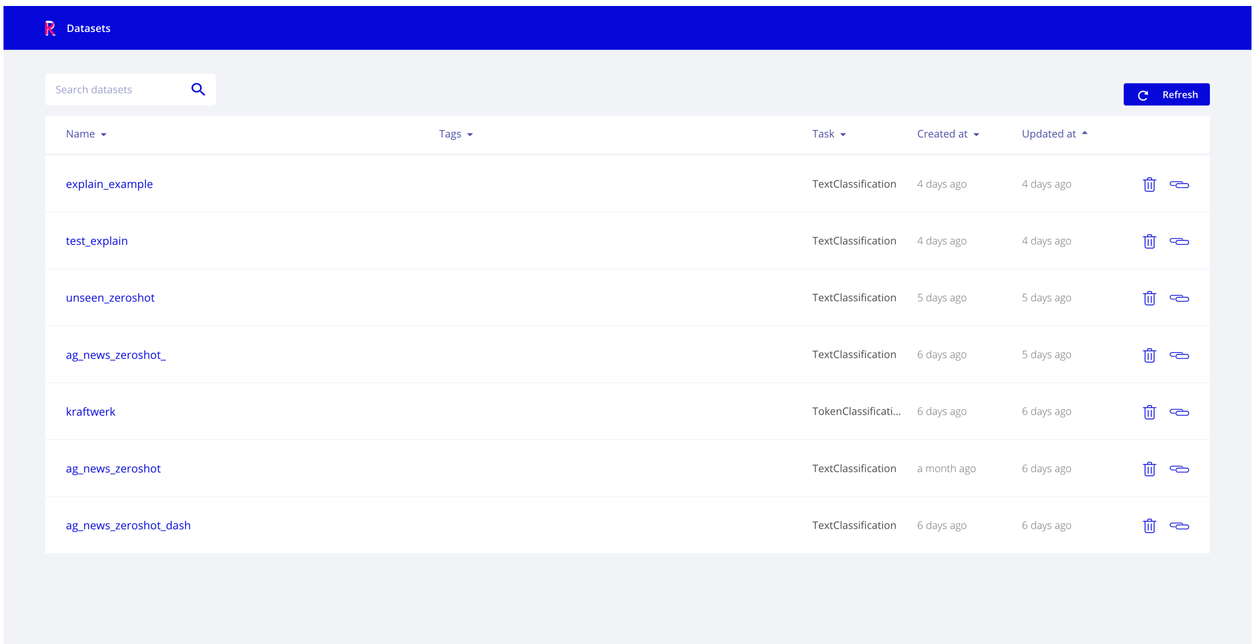


Fig. 1: Rubrix Home page view

5.20.2 Dataset page

The **Dataset page** is the workspace for exploring and annotating records in a Rubrix Dataset. Every task has its own specialized components, while keeping a similar layout and structure.

Here we describe the search components and the two modes of operation (Explore and Annotation).

The Rubrix Dataset page is driven by search features. The search bar gives users quick filters for easily exploring and selecting data subsets. The main sections of the search bar are following:

Search input

This component enables:

Full-text queries over all record `inputs`.

Queries using Elasticsearch's query DSL with the query string syntax, which enables powerful queries for advanced users, using the Rubrix data model. Some examples are:

`inputs.text:(women AND feminists)` : records containing the words "women" AND "feminist" in the inputs.text field.

`inputs.text:(NOT women)` : records NOT containing women in the inputs.text field.

`inputs.hypothesis:(not OR don't)` : records containing the word "not" or the phrase "don't" in the inputs.hypothesis field.

`metadata.format:pdf AND metadata.page_number>1` : records with metadata.format equals pdf and with metadata.page_number greater than 1.

`NOT(_exists_:metadata.format)` : records that don't have a value for metadata.format.

`predicted_as:(NOT Sports)` : records which are not predicted with the label `Sports`, this is useful when you have many target labels and want to exclude only some of them.

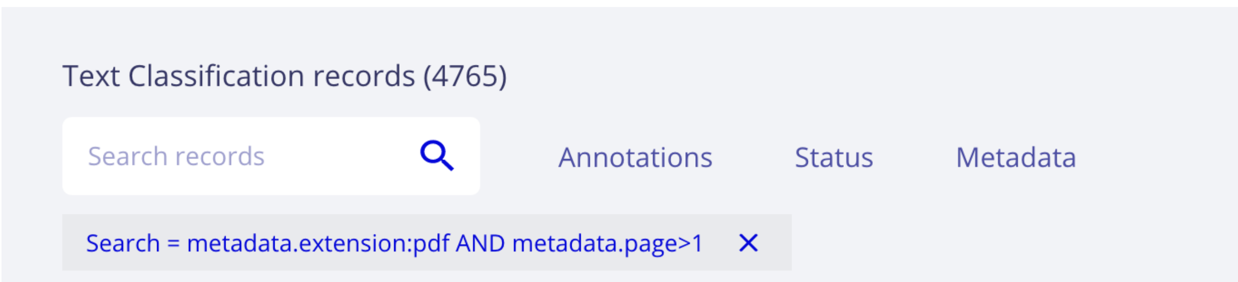


Fig. 2: Rubrix search input with Elasticsearch DSL query string

Elasticsearch's query DSL supports **escaping special characters** that are part of the query syntax. The current list special characters are

`+ - && || ! ( ) { } [ ] ^ " ~ * ? : \`

To escape these character use the \ before the character. For example to search for (1+1):2 use the query:

`\(1\+1\)\:2`

Elasticsearch fields

Below you can find a summary of available fields which can be used for the query DSL as well as for building Kibana Dashboards: common fields to all record types, and those specific to certain record types:

Common fields
annotated_as
annotated_by
event_timestamp
id
last_updated
metadata.*
multi_label
predicted
predicted_as
predicted_by
status
words

Text classification fields
inputs.*
score

Tokens classification fields
tokens

Predictions filters

This component allows filtering by aspects related to predictions, such as:

- predicted as, for filtering records by predicted labels,
- predicted by, for filtering by prediction_agent (e.g., different versions of a model)
- predicted ok or ko, for filtering records whose predictions are (or not) correct with respect to the annotations.

Annotations filters

This component allows filtering by aspects related to annotations, such as:

- annotated as, for filtering records by annotated labels,
- annotated by, for filtering by annotation_agent (e.g., different human users or dataset versions)

Status filter

This component allows filtering by record status:

- **Default**: records without any annotation or edition.
- **Validated**: records with validated annotations.
- **Edited**: records with annotations but not yet validated.

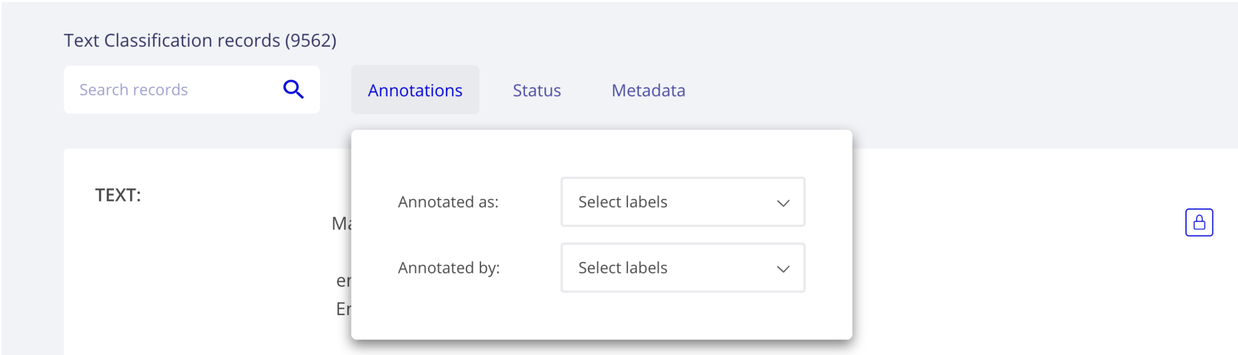


Fig. 3: Rubrix annotation filters

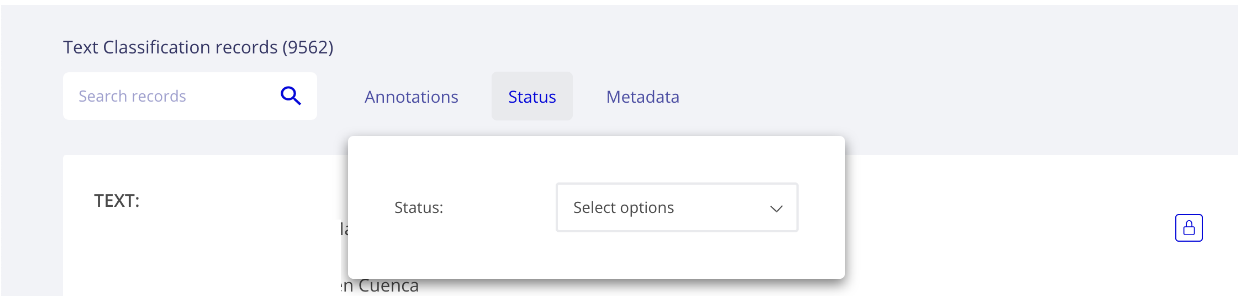


Fig. 4: Rubrix status filters

Metadata filters

This component allows filtering by metadata fields. The list of filters is dynamic and it's created with the aggregations of metadata fields included in any of the logged records.

Active query parameters

This component show the current active search params, it allows removing each individual param as well as all params at once.

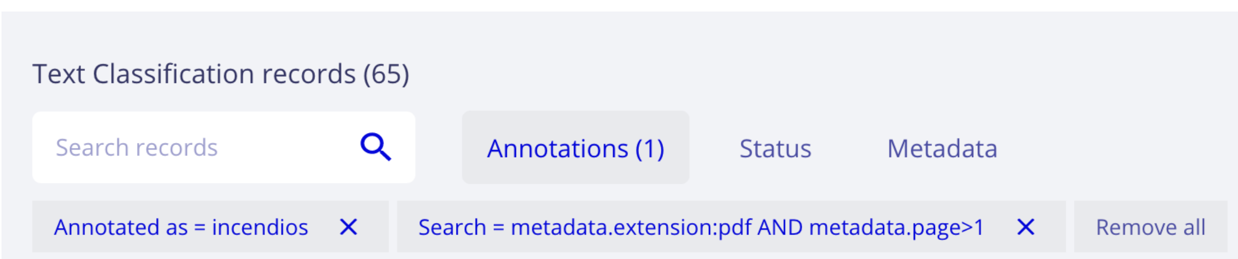


Fig. 5: Active query params module

Explore mode

This mode enables users to explore a records in a dataset. Different tasks provide different visualizations tailored for the task.

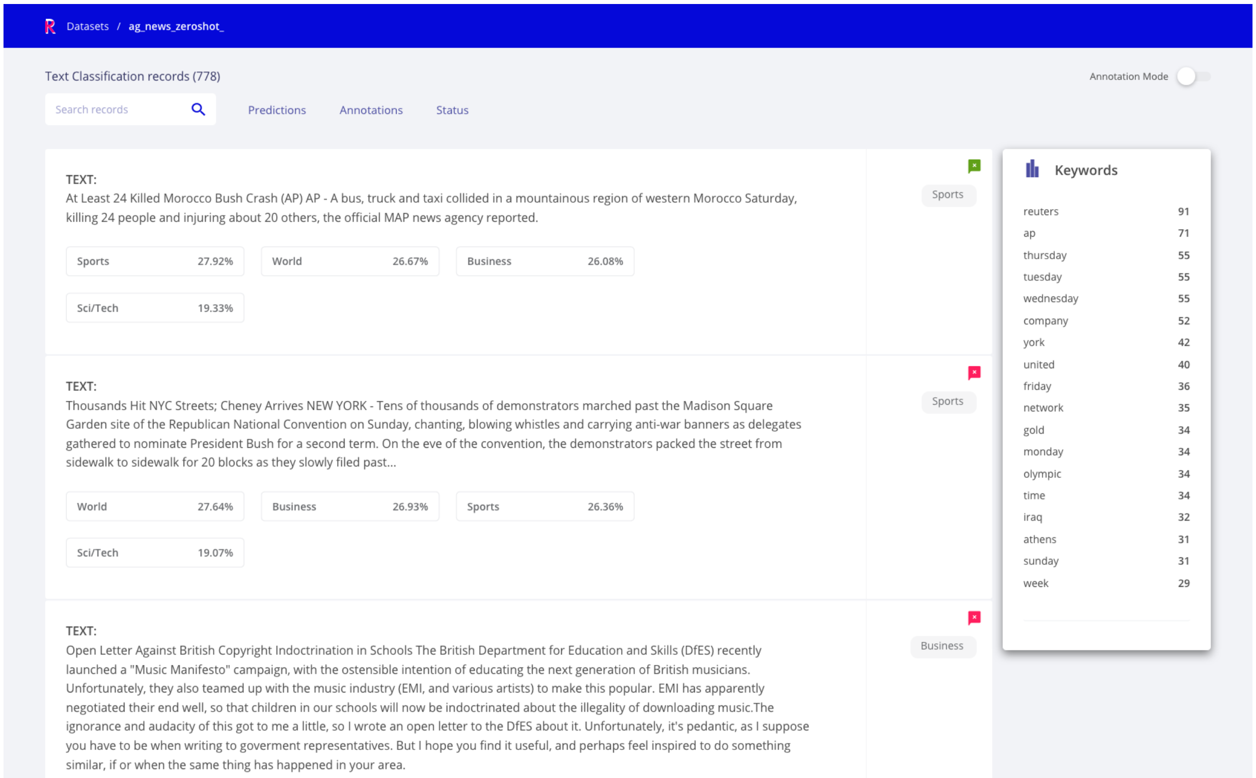


Fig. 6: Rubrix Text Classification Explore mode

Annotation mode

This mode enables users to add and modify annotations, while following the same interaction patterns as in the explore mode (e.g., using filters and advanced search), as well as novel features such as bulk annotation for a given set of search params.

Annotation by different users will be saved with different annotation agents. To setup various users in your Rubrix server, please refer to our user management guide.

5.21 Developer documentation

Here we provide some guides for the development of *Rubrix*.

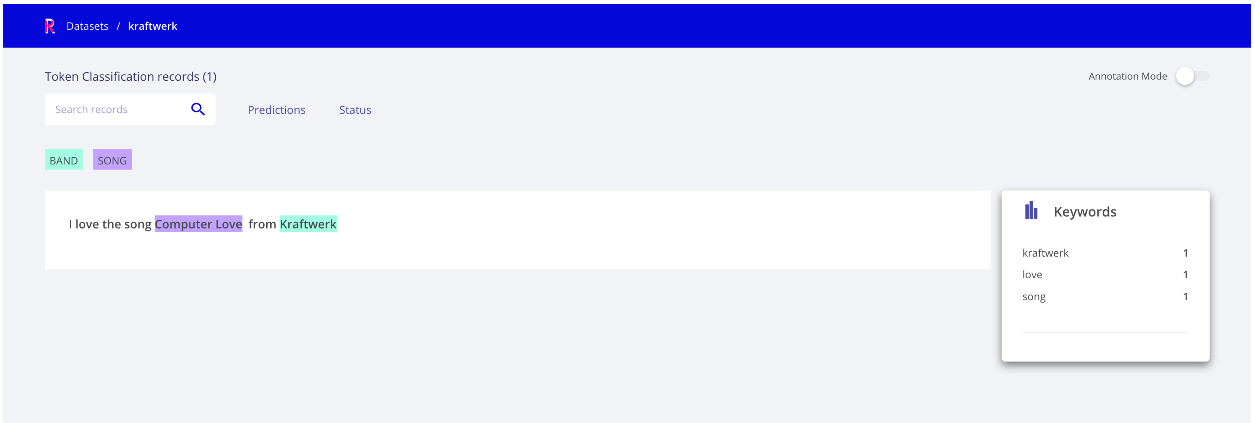


Fig. 7: Rubrix Token Classification (NER) Explore mode

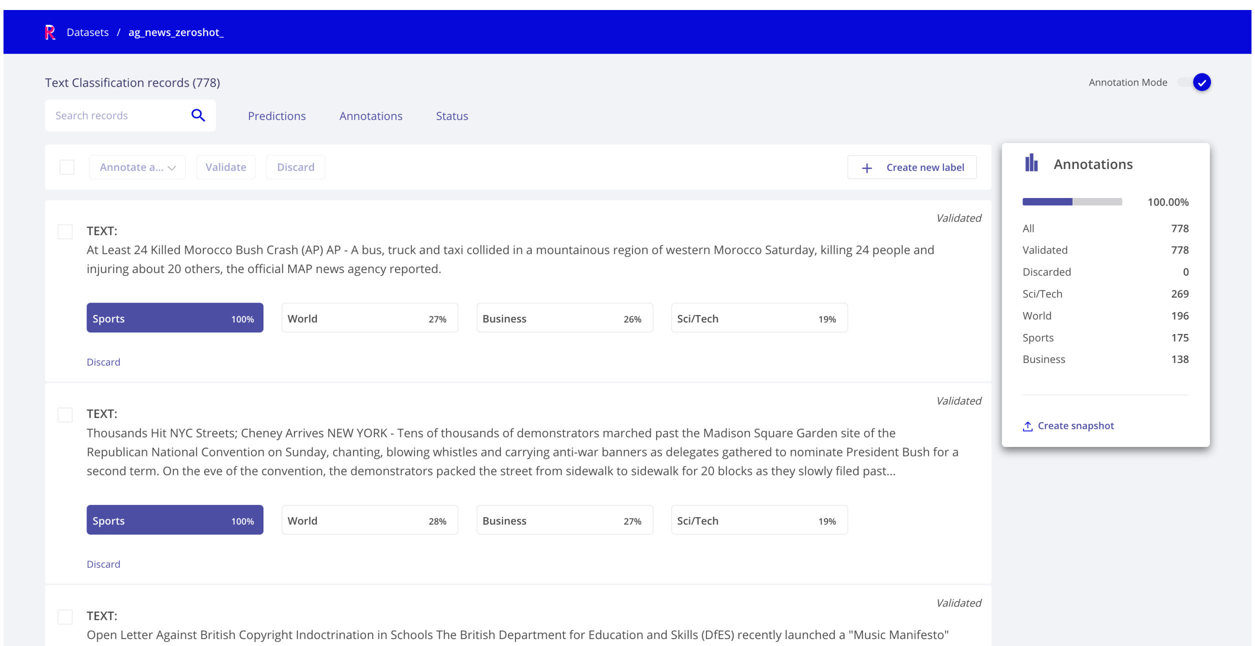


Fig. 8: Rubrix Text Classification Annotation mode

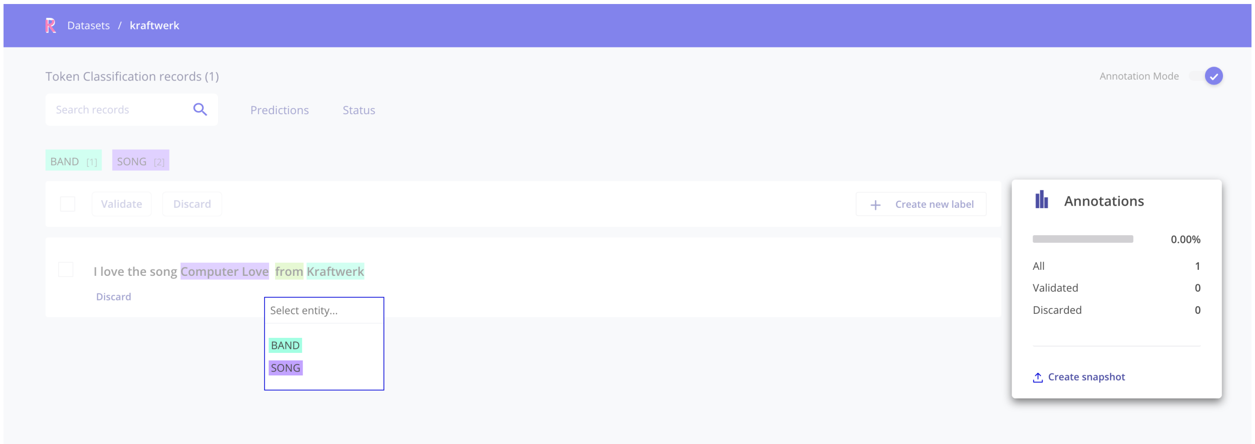


Fig. 9: Rubrix Token Classification (NER) Annotation mode

5.21.1 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
```

The last step is to build the static UI files in case you want to work on the UI:

```
bash scripts/build_frontend.sh
```

Now you are ready to take *Rubrix* to the next level

5.21.2 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

INDEX