# Rubrix

*Release 0.9*

**Recognai**

**Feb 03, 2022**

# 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:

```
pip install rubrix[server]==0.9.0 transformers[torch] datasets
```

If you don't have Elasticsearch (ES) running, make sure you have *Docker* installed and run:

**Note:** Check the *setup and installation section* for further options and configurations regarding Elasticsearch.

```
docker run -d \
  --name elasticsearch-for-rubrix \
  -p 9200:9200 -p 9300:9300 \
  -e "ES_JAVA_OPTS=-Xms512m -Xmx512m" \
  -e "discovery.type=single-node" \
  docker.elastic.co/elasticsearch/elasticsearch-oss:7.10.2
pip install "elasticsearch<7.14.0"
```

Then simply run:

```
python -m rubrix
```

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

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

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

### 5.1.1 1. Install Rubrix

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

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

**with pip**

```
pip install rubrix[server]==0.9.0
```

**with conda**

```
conda install -c conda-forge rubrix
```

**Note:** Conda for now only installs the Python client of Rubrix. This means, you have to launch the web app via *docker* or *docker-compose*.

### 5.1.2 2. Launch the web app

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

You can start the Rubrix web app via Python.

```
python -m rubrix
```

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

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

- *configure the Rubrix server*
- *share an ES instance with other applications*
- deploy Rubrix on an AWS instance

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

### 5.1.3 3. Start logging data

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

```
import rubrix as rb

rb.log(
    rb.TextClassificationRecord(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.

**Congratulations! You are ready to start working with 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"
```

(continues on next page)

(continued from previous page)

```
    },

)
```

### 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)],
```

(continues on next page)

(continued from previous page)

```
    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"
```

(continues on next page)

(continued from previous page)

```
  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:v0.9.0
    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:

### 5.4.1 Setting up Elasticsearch via docker

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

```
docker run -d \
  --name elasticsearch-for-rubrix \
  -p 9200:9200 -p 9300:9300 \
  -e "ES_JAVA_OPTS=-Xms512m -Xmx512m" \
  -e "discovery.type=single-node" \
  docker.elastic.co/elasticsearch/elasticsearch-oss:7.10.2
pip install "elasticsearch<7.14.0"
```

This will create an ES docker container named *"elasticsearch-for-rubrix"* that will run in the background, and it will also install the appropriate client.

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

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

Or you can stop/start the container via:

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

**Warning:** Keep in mind, if you remove your container with `docker rm elasticsearch-for-rubrix`, you will loose all your datasets in Rubrix!

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

### 5.4.2 Server configurations

By default, the Rubrix server will look for your ES endpoint at `http://localhost:9200`. But you can customize this by setting the `ELASTICSEARCH` environment variable.

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

```
uvicorn rubrix:app
```

*(for Rubrix versions below 0.9 you can launch the server via)*

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

For more details about fastapi and uvicorn, see here

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

### 5.4.3 Launching the web app via docker

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

```
docker pull recognai/rubrix
```

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

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

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

```
docker ps
```

To stop the Rubrix server, just stop the container:

```
docker stop rubrix
```

If you want to deploy your own Elasticsearch cluster via docker, we refer you to the excellent guide on the Elasticsearch homepage

### 5.4.4 Launching the web app via docker-compose

For this method you first need to install Docker Compose.

Then, create a folder:

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

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

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

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

**Warning:** Keep in mind, if you execute `docker-compose down`, you will loose all your datasets in Rubrix!

### 5.4.5 Configure Elasticsearch role/users

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

All you need to take into account is:

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

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

You can provide the credentials using the following scheme:

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

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

### 5.4.6 Deploy to aws instance using docker-machine

#### Setup an AWS profile

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

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

and follow command instructions. For more details, visit AWS official documentation

Once the profile is created (a new entry should 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:v0.9.0
    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

### 5.4.7 Install from master

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

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

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

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

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

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

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

### 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 transformers datasets -qqq
```

#### Text Classification

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

Let's see some examples.

#### Zero-shot classification pipelines

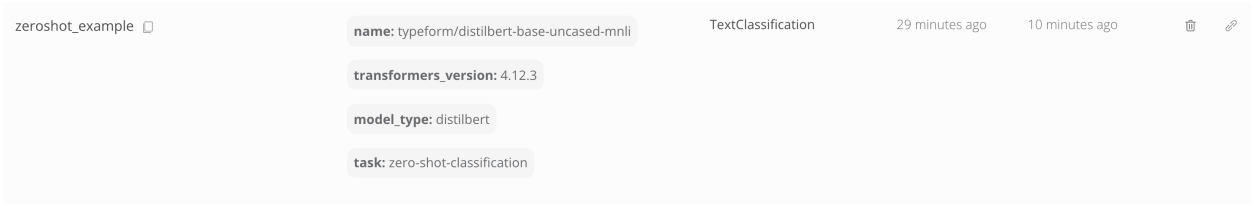
Let's load a `zero-shot-classification` pipeline:

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

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

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

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

It will create the following dataset:



zeroshot_example

name: typeform/distilbert-base-uncased-mnli

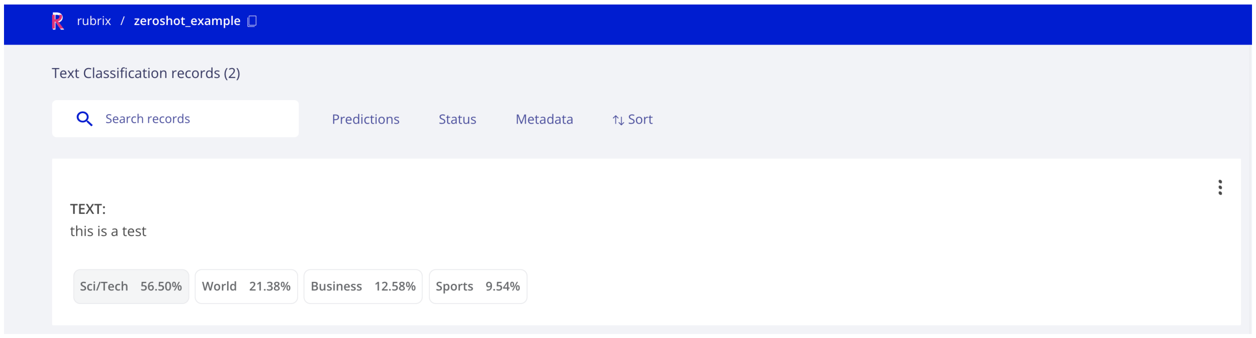
transformers_version: 4.12.3

model_type: distilbert

task: zero-shot-classification

TextClassification | 29 minutes ago | 10 minutes ago

which contains the following record:



Text Classification records (2)

TEXT:
this is a test

Sci/Tech 56.50% | World 21.38% | Business 12.58% | Sports 9.54%

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

```
[ ]: from datasets import load_dataset

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

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

### Text classification pipelines

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

Let's read a dataset with tweets:

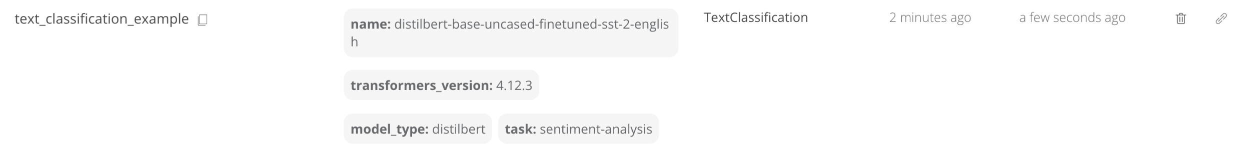
```
[17]: import pandas as pd

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

```
[17]:             Id                                             Category
      0  6.289494e+17  dear @Microsoft the newOoffice for Mac is grea...
      1  6.289766e+17  @Microsoft how about you make a system that do...
      2  6.290232e+17                                      Not Available
      3  6.291792e+17                                      Not Available
      4  6.291863e+17  If I make a game as a #windows10 Universal App...
```

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

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

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

which will create the following dataset:



text_classification_example

name: distilbert-base-uncased-finetuned-sst-2-english

transformers_version: 4.12.3

model_type: distilbert

task: sentiment-analysis

TextClassification

2 minutes ago

a few seconds ago

### Training

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

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

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

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

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

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

     # 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
     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")
```

(continues on next page)

(continued from previous page)

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

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

# Building TokenClassificationRecord
record = rb.TokenClassificationRecord(
    text=input_text,
    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
```

#### Token Classification (NER)

##### Inference

[ ]:
```
import rubrix as rb

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

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

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

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

### Training

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

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

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

import rubrix as rb


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

```
[ ]: # 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)
```

(continues on next page)

(continued from previous page)

```
    match = seqMatch.find_longest_match(0, len(string), 0, len(pattern))
    if (match.size == len(pattern)):
        start = match.a
        end = match.a + match.size
        match_tup = (start, end)
        string = string.replace(pattern, "X" * len(pattern), 1)
        match_list.append(match_tup)
    return match_list, string


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


def create_data(df, filepath):
    '''
    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)


# 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)
```

### 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`

[ ]:
```python
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')
```

[ ]:
```python
# 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)
```

(continues on next page)

(continued from previous page)

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

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


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

Let's make a prediction with flair `TextClassifier`

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

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

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

# predict class and print
classifier.predict(sentence)

print(sentence.labels)
```

### Zero-shot and Few-shot classifiers

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

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

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

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

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

(continues on next page)

(continued from previous page)

```
# 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 the German offensive language classifier

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

input_text = "Du erzählst immer Quatsch."

# Load our pre-trained classifier
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",
)
```

(continues on next page)

(continued from previous page)

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

### 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")
```

## 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`.

[ ]:
```python
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 suppose Stanza predicts the most likely label with 1.0 probability, and␣
↪the rest with 0.
entities = []

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

# Building a TextClassificationRecord
record = rb.TextClassificationRecord(
    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.

[ ]:
```python
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",
)

# 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.

[ ]:
```python
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",
        ],
    )

    # 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"},
```

(continues on next page)

(continued from previous page)

```
        )
    )

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

records = []

for record in dataset:

    # Making the prediction
    predictions = classifier(
```

(continues on next page)

(continued from previous page)

```
        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]")
```

(continues on next page)

(continued from previous page)

```python
# 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"},
        )
    )


# 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.

[ ]:
```python
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"]))

    # 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={
```

(continues on next page)

(continued from previous page)

```
        "task": "nli",
        "family": "text-classification",
        "dataset": "snli",
    },
)
```

### Stance Detection

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

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

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

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

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

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

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

records = []

for record in dataset:

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

(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["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
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 = []
```

(continues on next page)

(continued from previous page)

```python
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)

         # 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={
```

(continues on next page)

(continued from previous page)

```
        "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
    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,
```

(continues on next page)

(continued from previous page)

```
    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",
     )
```

(continues on next page)

(continued from previous page)

```
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={
        "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)
```

(continues on next page)

(continued from previous page)

```python
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.

[ ]:
```python
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(
        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."

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

### 5.7.1 Rubrix weak supervision in a nutshell

The recommended workflow for weak supervision is:

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

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

#### Weak labeling using the UI

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

#### Weak supervision from Python

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

Rubrix weak supervision support is built around two basic abstractions:

#### Rule

A rule encodes an heuristic for labeling a record.

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

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

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

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

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

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

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

#### Weak Labels

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

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

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

# 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 Detailed Workflow

A typical workflow to use weak supervision is:

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

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

### 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.

[2]:
```
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()
```

[2]:
```
   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(
```

(continues on next page)

(continued from previous page)

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

# build records from the test dataset with annotation
labels = ["HAM", "SPAM"]
records += [
    rb.TextClassificationRecord(
        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 that you can access via the *Rubrix web app*.

### 5.7.6 2. Defining rules

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

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

Let's see here some programmatic rules:

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

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

You can also define plain Python labeling functions:

```
[ ]: import re

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

(continues on next page)

(continued from previous page)

```
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
```

### 5.7.7 3. Building and analizing weak labels

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

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

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

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

[12]:
```
# show some stats about the rules, see the `summary()` docstring for details
weak_labels.summary()
```

[12]:
```
                    polarity  coverage  overlaps  conflicts  correct  \
check out             {SPAM}  0.242919  0.235839   0.029956       45
plz OR please         {SPAM}  0.090414  0.081155   0.019608       20
subscribe             {SPAM}  0.106754  0.083878   0.028867       30
my                    {SPAM}  0.190632  0.166667   0.049564       41
song                   {HAM}  0.132898  0.079521   0.033769       39
love                   {HAM}  0.092048  0.070261   0.031590       28
contains_http         {SPAM}  0.106209  0.073529   0.049564        6
short_comment          {HAM}  0.245098  0.110566   0.064270       84
regex_check_out       {SPAM}  0.226580  0.226035   0.027778       45
total            {SPAM, HAM}  0.754902  0.448802   0.120915      338

                 incorrect  precision
check out                0   1.000000
plz OR please            0   1.000000
subscribe                0   1.000000
my                       6   0.872340
song                     9   0.812500
love                     7   0.800000
contains_http            0   1.000000
short_comment            8   0.913043
regex_check_out          0   1.000000
total                   30   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:

#### A simple majority vote

As a first example we will show you, how to use the `WeakLabels` object together with a simple majority vote model. For this we will take the implementation by Snorkel, and evaluate it with the help of sklearn's metrics module.

[ ]:
```
%pip install snorkel scikit-learn -qqq
```

The majority vote model is arguably the most straightforward label model. On a per-record basis, it simply counts the votes for each label returned by the rules, and takes the majority vote. Snorkel provides a neat implementation of this logic in its `MajorityLabelVoter`.

[ ]:
```
from snorkel.labeling.model import MajorityLabelVoter

# instantiate the majority vote label model
majority_model = MajorityLabelVoter()
```

Let's go on and evaluate this baseline. To break ties when there is no majority vote, we choose the *"random"* policy that randomly selects one of the tied labels. In this way we avoid a bias towards label models that produce fewer but more certain weak labels, and makes the comparison between the different label models fairer.

[ ]:
```
# compute accuracy
majority_model.score(
    L=weak_labels.matrix(has_annotation=True),
    Y=weak_labels.annotation(),
    tie_break_policy="random",
)
# {'accuracy': 0.844}
```

As we will see further down, an accuracy of 0.844 is a very decent baseline. Choosing to simply ignore tiebreaks and abstentions (by setting the tiebreak policy to *"abstain"*), we would obtain an accuracy of nearly 0.96.

When predicting weak labels to train a down-stream model, you probably want to discard the abstentions and tiebreaks.

[ ]:
```
# get predictions for training a down-stream model
predictions = majority_model.predict(L=weak_labels.matrix(has_annotation=False))

# records for training
training_records = weak_labels.records(has_annotation=False)

# mask to ignore abstentions/tiebreaks
idx = predictions != -1

# combine records and predictions
```

(continues on next page)

(continued from previous page)

```
training_data = pd.DataFrame(
    [
        {"text": rec.inputs["text"], "label": weak_labels.int2label[label]}
        for rec, label in zip(training_records, predictions)
    ]
)[idx]
```

[240]:
```
# preview training data
training_data
```

[240]:
```
                                                  text label
0     Hey I&#39;m a British youtuber!!<br />I upload...  SPAM
1                                        NOKIA spotted   HAM
2                                             Dance :)   HAM
3     You guys should check out this EXTRAORDINARY w...  SPAM
4     Need money ? check my channel and subscribe,so...  SPAM
...                                                ...   ...
1579  Please check out my acoustic cover channel :) ...  SPAM
1580  PLEASE SUBSCRIBE ME!!!!!!!!!!!!!!!!!!!!!!!!!!!...  SPAM
1581  <a href="http://www.gofundme.com/Helpmypitbull...  SPAM
1582  I love this song so much!:-D I've heard it so ...   HAM
1585                   Check out this video on YouTube:  SPAM

[1055 rows x 2 columns]
```

### Label model with Snorkel

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

[ ]:
```
%pip install snorkel -qqq
```

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

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

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

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

[ ]:
```
# we check its performance
snorkel_model.score(tie_break_policy="abstain")
# {'accuracy': 0.848, ...}
```

When choosing to simply ignore tiebreaks and abstentions in the score (by setting the tiebreak policy to *"abstain"*), we would obtain an accuracy of about 0.95.

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 = snorkel_model.predict()

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

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

[225]:
```
# preview training data
training_data
```

[225]:
```
                                                   text label
0     Hey I&#39;m a British youtuber!!<br />I upload...  SPAM
1                                        NOKIA spotted   HAM
2                                             Dance :)   HAM
3     You guys should check out this EXTRAORDINARY w...  SPAM
4     Need money ? check my channel and subscribe,so...  SPAM
...                                                 ...   ...
1172  Please check out my acoustic cover channel :) ...  SPAM
1173  PLEASE SUBSCRIBE ME!!!!!!!!!!!!!!!!!!!!!!!!!!!...  SPAM
1174  <a href="http://www.gofundme.com/Helpmypitbull...  SPAM
1175  I love this song so much!:-D I've heard it so ...   HAM
1176                    Check out this video on YouTube:  SPAM

[1177 rows x 2 columns]
```

Note

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

### Label model with FlyingSquid

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

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

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

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

(continues on next page)

(continued from previous page)

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

[ ]:
```
# we check its performance
flyingsquid_model.score(tie_break_policy="random")
# {'accuracy': 0.832, ...}
```

When choosing to simply ignore tiebreaks and abstentions in the score (by setting the tiebreak policy to *"abstain"*), we would obtain an accuracy of about 0.93.

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 = flyingsquid_model.predict()

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

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

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

[231]:
```
                                                  text label
0     Hey I&#39;m a British youtuber!!<br />I upload...  SPAM
1                                         NOKIA spotted   HAM
2                                              Dance :)   HAM
3     You guys should check out this EXTRAORDINARY w...  SPAM
4     Need money ? check my channel and subscribe,so...  SPAM
...                                                 ...   ...
1172  Please check out my acoustic cover channel :) ...  SPAM
1173  PLEASE SUBSCRIBE ME!!!!!!!!!!!!!!!!!!!!!!!!!!!...  SPAM
1174  <a href="http://www.gofundme.com/Helpmypitbull...  SPAM
1175  I love this song so much!:-D I've heard it so ...   HAM
1176                    Check out this video on YouTube:  SPAM

[1177 rows x 2 columns]
```

### Joint Model with Weasel

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

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

Let's start with installing the Weasel python package:

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

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

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

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

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

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

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

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

     # instantiate our weasel end-to-end model
     weasel = Weasel(
         end_model=end_model,
         num_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 the `TransformersDataModule`, so that Weasel and PyTorch Lightning can work with it. In this step we also tokenize the data. Here we need to be careful to use the corresponding tokenizer to our end model.

```
[ ]: from transformers import AutoTokenizer
     from weasel.datamodules.transformers_datamodule import TransformersDataModule,␣
     ↪TransformersCollator
```

(continues on next page)

(continued from previous page)

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

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

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

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

[ ]:
```
import pytorch_lightning as pl

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

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

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

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

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

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

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

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

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

```
[ ]: from transformers import pipeline

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

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

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

## 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:

```
[ ]: from datasets import load_dataset

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

### Hugging Face Transformer Pipelines

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

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

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

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

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

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

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

```
[ ]: dataset
```

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

### spaCy

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

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

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

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

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

### Flair

Rubrix currently supports monitoring Flair NER pipelines component.

```
[ ]: import rubrix as rb

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

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

(continues on next page)

(continued from previous page)

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

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

The following logs the predictions over the IMDB dataset:

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

dataset.map(make_prediction)
```

### 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")
```

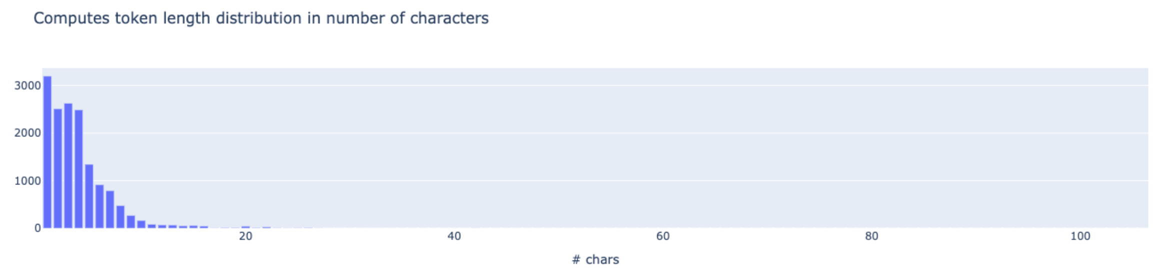
#### Log records into a Rubrix dataset

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

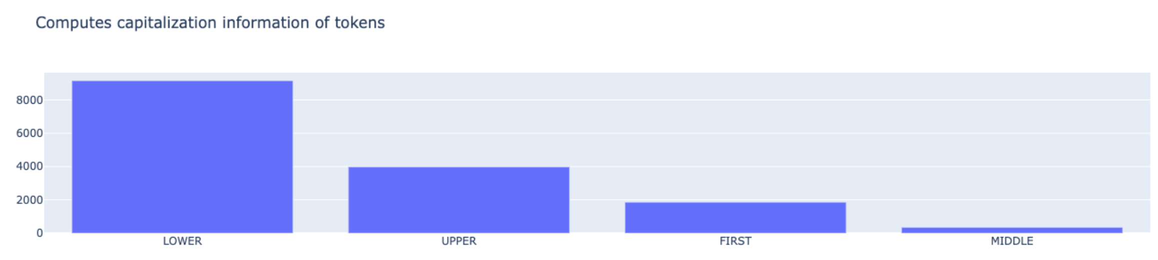
```
[ ]: nlp = rb.monitor(nlp, dataset="spacy_sm_wnut17")

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

     dataset.map(predict)
```

#### Explore some metrics for this pipeline

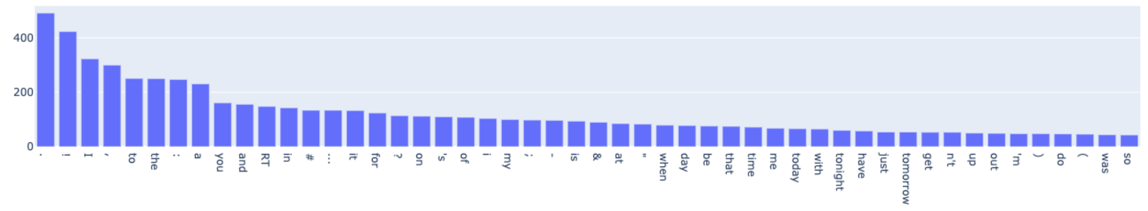
```
[17]: from rubrix.metrics.token_classification import token_length

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



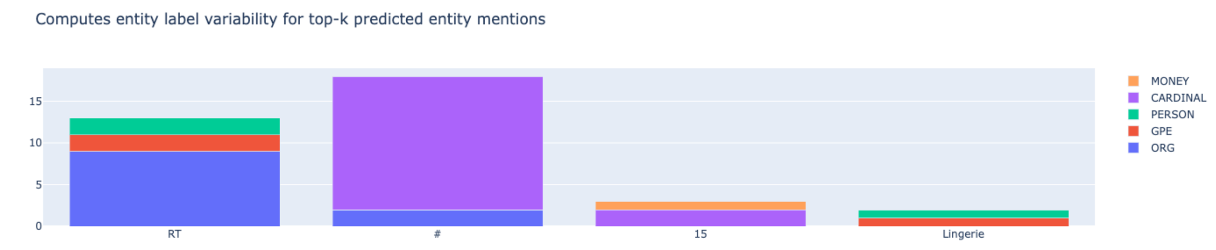
```
[18]: from rubrix.metrics.token_classification import token_capitalness

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



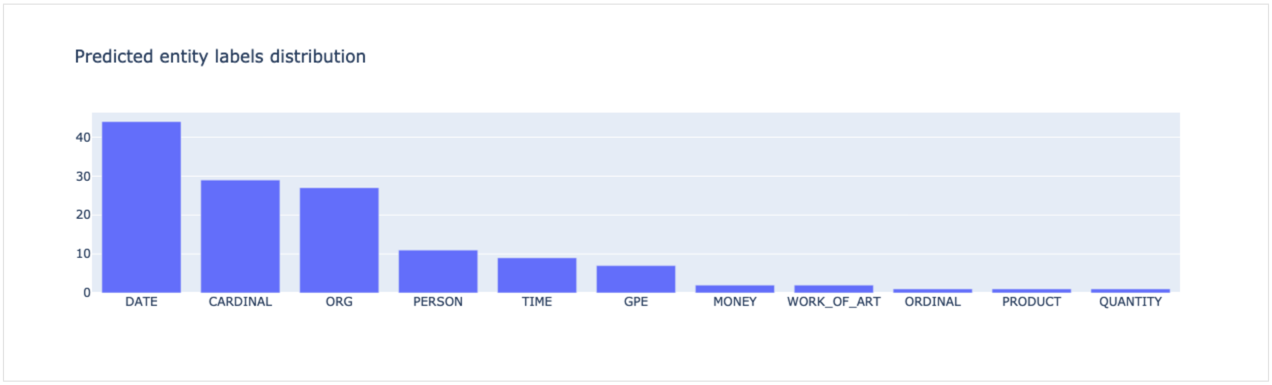
```
[20]: from rubrix.metrics.token_classification import token_frequency

token_frequency(name="spacy_sm_wnut17", tokens=50).visualize()
```



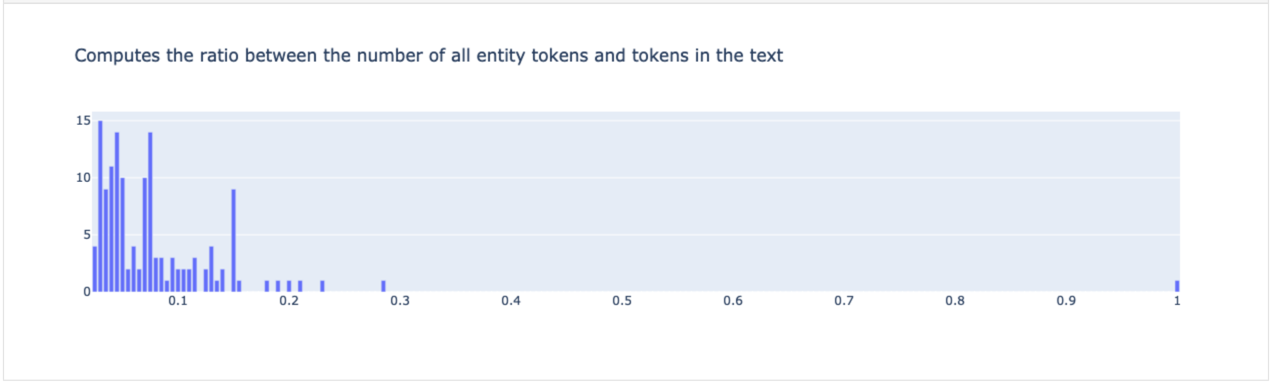
```
[21]: from rubrix.metrics.token_classification import entity_consistency

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



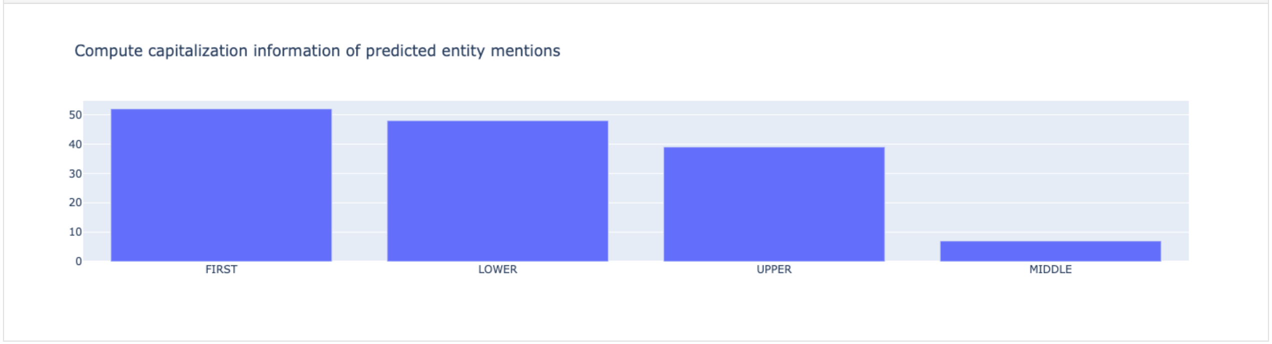
```
[5]: from rubrix.metrics.token_classification import entity_labels

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



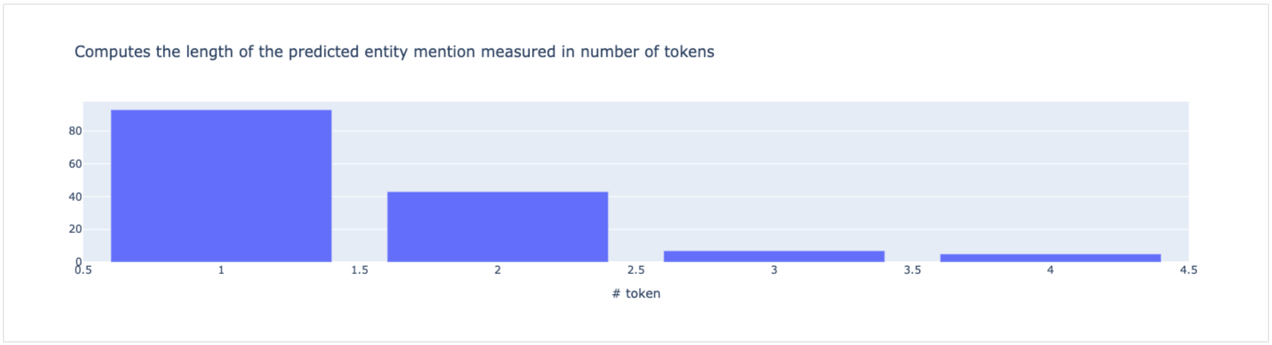
[6]:
```
from rubrix.metrics.token_classification import entity_density

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



[7]:
```
from rubrix.metrics.token_classification import entity_capitalness

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



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



### 5.9.3 2. Rubrix Metrics for NER training sets

#### Analyzing tags

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

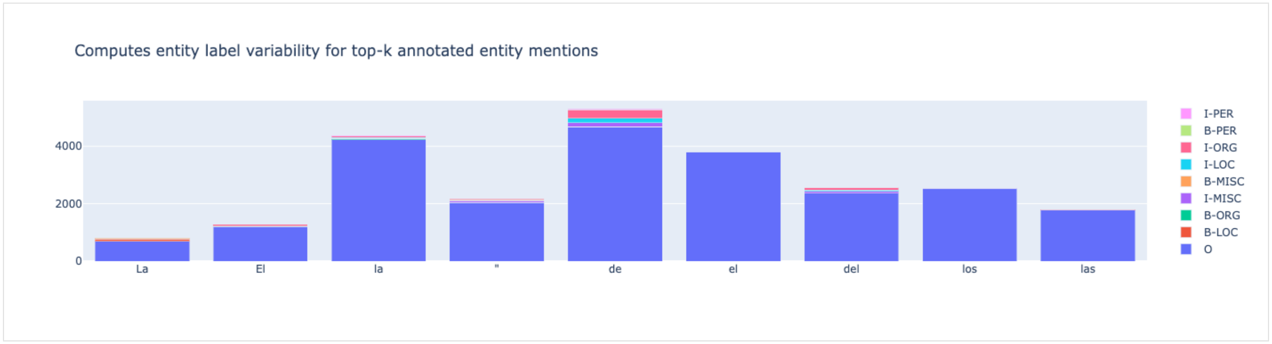
```
[ ]: dataset = load_dataset("conll2002", "es", split="train[0:5000]")
```

```
[24]: def parse_entities(record):
          entities = []
          counter = 0
          for i in range(len(record['ner_tags'])):
              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
      ]
```

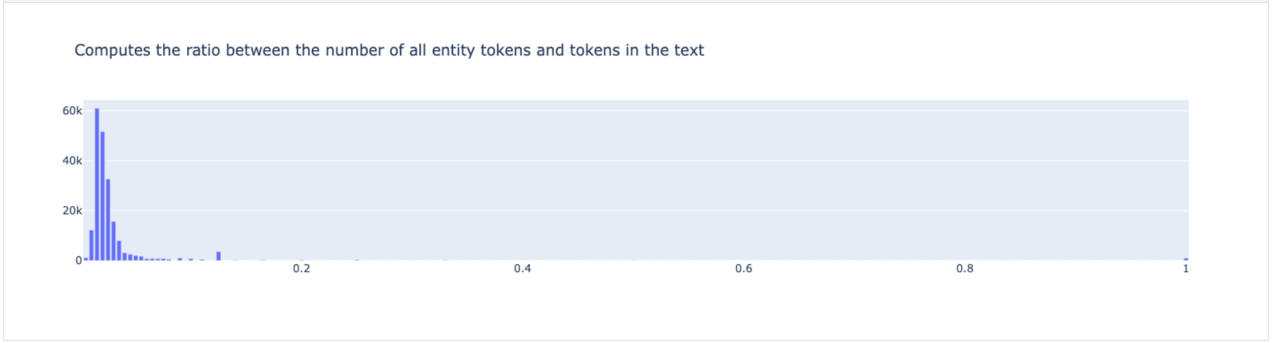
```
[ ]: rb.log(records, "conll2002_es")
```

```
[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()
```



From the above we see we can quickly detect an annotation issue: double quotes " are most of the time tagged as O (no entity) but in some cases (~60 examples) are tagged as beginning of entities like ORG or MISC, which is likely a hand-labelling error, including the quotes inside the entity span.

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

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



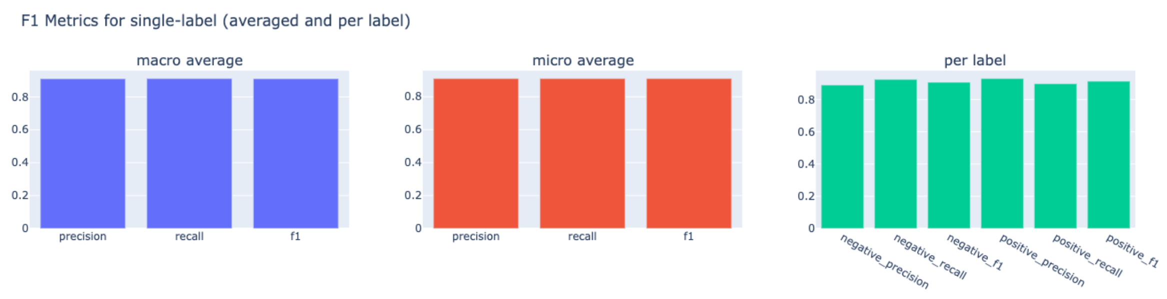
## 5.9.4 2. Rubrix Metrics for text classification

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

     import rubrix as rb

     sst2 = load_dataset("glue", "sst2", split="validation")
     labels = sst2.features["label"].names
     nlp = pipeline("sentiment-analysis")
```

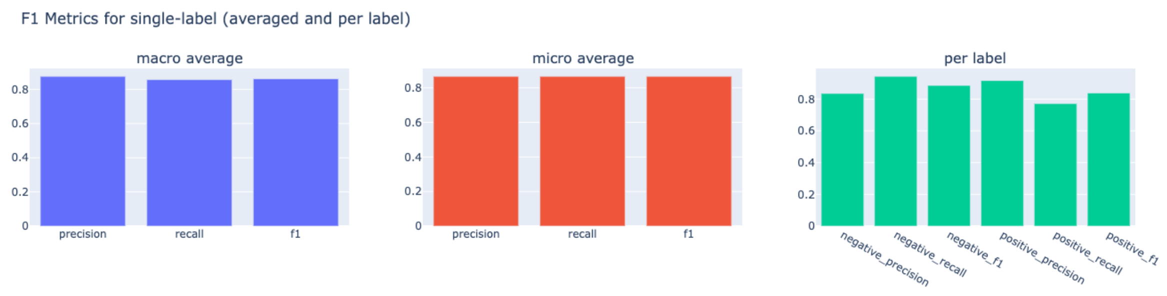
```
[11]: records = [
          rb.TextClassificationRecord(
              inputs=record["sentence"],
              annotation=labels[record["label"]],
              prediction=[(pred["label"].lower(), pred["score"]) for pred in nlp(record[
      ↪"sentence"])]
          )
          for record in sst2
      ]
```

```
[ ]: rb.log(records, name="sst2")
```

```
[13]: from rubrix.metrics.text_classification import f1

f1(name="sst2").visualize()
```



```
[20]: # now compute metrics for negation ( -> negative precision and positive recall go down)
f1(name="sst2", query="n't OR not").visualize()
```



## 5.10 Label your data to fine-tune a classifier with Hugging Face

In this tutorial, we'll build a sentiment classifier for user requests in the banking domain as follows:

- Start with the most popular sentiment classifier on the Hugging Face Hub (almost 4 million monthly downloads as of December 2021) which has been fine-tuned on the SST2 sentiment dataset.
- Label a training dataset with banking user requests starting with the pre-trained sentiment classifier predictions.
- Fine-tune the pre-trained classifier with your training dataset.
- Label more data by correcting the predictions of the fine-tuned model.
- Fine-tune the pre-trained classifier with the extended training dataset.

### 5.10.1 Introduction

This tutorial will show you how to fine-tune a sentiment classifier for your own domain, starting with no labeled data.

Most online tutorials about fine-tuning models assume you already have a training dataset. You'll find many tutorials for fine-tuning a pre-trained model with widely-used datasets, such as IMDB for sentiment analysis.

However, very often **what you want is to fine-tune a model for your use case**. It's well-known that NLP model performance usually degrades with "out-of-domain" data. For example, a sentiment classifier pre-trained on movie reviews (e.g., IMDB) will not perform very well with customer requests.

This is an overview of the workflow we'll be following:

Let's get started!

### 5.10.2 Setup

Rubrix, is a free and open-source tool to explore, annotate, and monitor data for NLP projects.

If you are new to Rubrix, check out the Github repository .

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

In this tutorial, we'll use the `transformers`, `datasets` and `sklearn` libraries. We'll also install `ipwidgets` for training progress bars.

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

### 5.10.3 Preliminaries

For building our fine-tuned classifier we'll be using two main resources, both available in the  Hub :

1. A **dataset** in the banking domain: banking77
2. A **pre-trained sentiment classifier**: distilbert-base-uncased-finetuned-sst-2-english

#### Dataset: `Banking 77`

This dataset contains online banking user queries annotated with their corresponding intents.

In our case, **we'll label the sentiment of these queries**. This might be useful for digital assistants and customer service analytics.

Let's load the dataset directly from the hub and split the dataset into two 50% subsets. We'll start with the `to_label1` split for data exploration and annotation, and keep `to_label2` for further iterations.

```
[ ]: from datasets import load_dataset

banking_ds = load_dataset("banking77")

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 December 2021, the `distilbert-base-uncased-finetuned-sst-2-english` is in the top five of the most popular text-classification models in the Hugging Face Hub.

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

As we will see later, this is a general-purpose sentiment classifier, which will need further fine-tuning for specific use cases and styles of text. In our case, **we'll explore its quality on banking user queries and build a training set for adapting it to this domain**.

Let's load the model and test it with an example from our dataset:

[2]:
```
from transformers import pipeline

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

to_label1[3]['text'], sentiment_classifier(to_label1[3]['text'])
```

[2]:
```
('Hi, Last week I have contacted the seller for a refund as directed by you, but i have␣
↪not received the money yet. Please look into this issue with seller and help me in␣
↪getting the refund.',
 [[{'label': 'NEGATIVE', 'score': 0.9934700727462769},
   {'label': 'POSITIVE', 'score': 0.006529912818223238}]])
```

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.

## 5.10.4 1. Run the pre-trained model over the dataset and log the predictions

As a first step, let's use the pre-trained model for predicting over our raw dataset. For this, we will use the handy `dataset.map` method from the `datasets` library.

The following steps could be simplified by using the auto-monitor support for Hugging Face pipelines. You can find more details in the *Monitoring guide*.

### Predict

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

     # add .select(range(10)) before map if you just want to test this quickly with 10␣
     ↪examples
     to_label1 = to_label1.map(predict, batched=True, batch_size=4)
```

```
  0%|          | 0/3 [00:00<?, ?ba/s]
```

### 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.

```
[ ]: import rubrix as rb

     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)
```

## 5.10.5 2. Explore and label data with the pretrained model

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

At first sight:

- The pre-trained sentiment classifier tends to label most of the examples as `NEGATIVE` (4.835 of 5.001 records). You can see this yourself using the `Predictions / Predicted as:` filter
- Using this filter and filtering by predicted as `POSITIVE`, we see that examples like "*I didn't withdraw the amount of cash that is showing up in the app.*" are not predicted as expected (according to our basic "annotation policy" described in the preliminaries).

Taking into account this analysis, we can start labeling our data.

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

1. **Start labeling examples sequentially**, without using search features. This way you will annotate a fraction of your data which will be aligned with the dataset distribution.
2. Once you have a sense of the data, you can **start using filters and search features to annotate examples with specific labels**. In our case, we'll label examples predicted as `POSITIVE` by our pre-trained model, and then a few examples predicted as `NEGATIVE`.

### Labeling random examples

### Labeling POSITIVE examples

After some minutes, we've labelled almost **5% of our raw dataset with more than 200 annotated examples**, which is a small dataset but should be enough for a first fine-tuning of our banking sentiment classifier:

## 5.10.6 3. Fine-tune the pre-trained model

In this step, we'll load our training set from Rubrix and fine-tune using the `Trainer` API from Hugging Face `transformers`. For this, we closely follow the guide Fine-tuning a pre-trained model from the `transformers` docs.

First, let's load the annotations from our dataset using the query parameter from the `load` method. The `Validated` status corresponds to annotated records.

[11]:
```
rb_df = rb.load(name='labeling_with_pretrained', query="status:Validated")
rb_df.head()
```

[11]:
```
                                              inputs  \
0  {'text': 'Why couldn't I make a withdrawal fro...
1  {'text': 'Hi, Last week I have contacted the s...
2         {'text': 'Why have I not received my PIN'}

                                          prediction annotation  \
0  [(NEGATIVE, 0.9984630346298218), (POSITIVE, 0...   NEGATIVE
1  [(NEGATIVE, 0.9934700727462769), (POSITIVE, 0...   NEGATIVE
2  [(NEGATIVE, 0.9959989786148071), (POSITIVE, 0...   NEGATIVE

                                 prediction_agent annotation_agent  \
0  distilbert-base-uncased-finetuned-sst-2-english           rubrix
1  distilbert-base-uncased-finetuned-sst-2-english           rubrix
2  distilbert-base-uncased-finetuned-sst-2-english           rubrix

   multi_label explanation                                    id  \
0        False        None  012e8ec1-2b3a-4efd-b593-4cbfc3fa4ec9
1        False        None  102f02fa-1474-42b6-8025-5d0ae67d1d8c
2        False        None  40db49db-92e7-46b0-b23d-6888e0149f14
```

(continues on next page)

(continued from previous page)

```
           metadata     status event_timestamp metrics
0  {'category': 26}  Validated            None      {}
1  {'category': 51}  Validated            None      {}
2  {'category': 38}  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, Features, Value, ClassLabel
from transformers import AutoTokenizer

# select text input and the annotated label
rb_df['text'] = rb_df.inputs.transform(lambda r: r['text'])
rb_df['labels'] = rb_df.annotation

ds = rb_df[['text', 'labels']].to_dict(orient='list')

# create  dataset from pandas with labels as numeric ids
train_ds = Dataset.from_dict(
    ds,
    features=Features({
        "text": Value("string"),
        "labels": ClassLabel(names=list(rb_df.labels.unique()))
    })
)
train_ds = train_ds.train_test_split(test_size=0.2) ; train_ds
```

```
[ ]: # tokenize our datasets
tokenizer = AutoTokenizer.from_pretrained("distilbert-base-uncased-finetuned-sst-2-
↪english")

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

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, but we leave this experiment to you.

Let's load the model:

```
[ ]: from transformers import AutoModelForSequenceClassification

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

Let's configure the **Trainer**:

[ ]:
```python
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, we can train our first model!

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

## 5.10.7 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:

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

Then, we can compare its predictions with the pre-trained model and an example:

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

As you can see, our fine-tuned model now classifies this general questions (not related to issues or problems) as POSITIVE, while the pre-trained model still classifies this as NEGATIVE.

Let's check now an example related to an issue where both models work as expected:

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

### 5.10.8 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.

[21]:
```
rb_df = rb.load(name='labeling_with_pretrained', query="status:Default")
rb_df['text'] = rb_df.inputs.transform(lambda r: r['text'])
ds = Dataset.from_pandas(rb_df[['text']])
```

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

[22]:
```
def predict(examples):
    return {"predictions": finetuned_sentiment_classifier(examples['text'])}

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

[ ]:
```
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)
```

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

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

At first sight, using the predicted as filter by POSITIVE and then by NEGATIVE, we can observe that the fine-tuned model predictions are more aligned with our "annotation policy".

Now that the model is performing better for our use case, we'll extend our training set with highly informative examples. A typical workflow for doing this is as follows:

1. Use the prediction score filter for labeling uncertain examples.
2. Label examples predicted by our fine-tuned model as POSITIVE and then predicted as NEGATIVE to correct the predictions.

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

## 5.10.10 7. Fine-tuning with the extended training dataset

In this step, we'll add the new examples to our training set and fine-tune a new version of our banking sentiment classifier.

### Adding labeled examples to our previous training set

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

```
[ ]: 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

     df = prepare_train_df('labeling_with_finetuned')
     train_dataset = train_dataset.remove_columns('__index_level_0__')
```

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

```
[ ]: 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'],
         })
```

### Training our sentiment classifier

As we want to measure the effect of adding examples to our training set we will:

- Fine-tune from the pre-trained sentiment weights (as we did before)
- Use the previous test set and the extended train set (obtaining a metric we use to compare this new version with our previous model)

```
[ ]: from transformers import AutoModelForSequenceClassification
     model = AutoModelForSequenceClassification.from_pretrained("distilbert-base-uncased-
     ↪finetuned-sst-2-english")
```

```
[ ]: train_ds = train_dataset.shuffle(seed=42)

     trainer = Trainer(
         args=training_args,
```

(continues on next page)

(continued from previous page)

```
    model=model,
    train_dataset=train_dataset,
    eval_dataset=eval_dataset,
    compute_metrics=compute_metrics,
)

trainer.train()
```

```
[ ]: model.save_pretrained("distilbert-base-uncased-sentiment-banking")
```

### 5.10.11 Summary

In this tutorial, you learned how to build a training set from scratch with the help of a pre-trained model, performing two iterations of `predict` > `log` > `label`.

Although this is somehow a toy example, you will be able to apply this workflow to your own projects to adapt existing models or building them from scratch.

In this tutorial, we've covered one way of building training sets: **hand labeling**. If you are interested in other methods, which could be combined witth hand labeling, checkout the following:

- *Building a news classifier with weak supervision*
- *Active learning with ModAL and scikit-learn*

### 5.10.12 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

In this tutorial, we will build a news classifier using rules and weak supervision:

- For this example, we use the AG News dataset but you can follow this process to programmatically label any dataset.
- The train split without labels is used to build a training set with rules, Rubrix and Snorkel's Label model.
- The test set is used for evaluating our weak labels, label model and downstream news classifier.
- We achieve 0.84 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:

### 5.11.1 Introduction

> *Weak supervision is a branch of machine learning where noisy, limited, or imprecise sources are used to provide supervision signal for labeling large amounts of training data in a supervised learning setting. This approach alleviates the burden of obtaining hand-labeled data sets, which can be costly or impractical. Instead, inexpensive weak labels are employed with the understanding that they are imperfect, but can nonetheless be used to create a strong predictive model.* [Wikipedia]

For a broader introduction to weak supervision, as well as further references, we recommend the excellent overview by Alex Ratner et al..

This tutorial aims to be a practical introduction to weak supervision and will walk you through its entire process. First we will generate weak labels with *Rubrix*, combine these labels with *Snorkel*, and finally train a classifier with *Scikit Learn*.

### 5.11.2 Setup

Rubrix, is a free and open-source tool to explore, annotate, and monitor data for NLP projects.

If you are new to Rubrix, check out the Github repository.

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

For this tutorial we also need some third party libraries that can be installed via pip:

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

### 5.11.3 1. Load test and unlabelled datasets into Rubrix

First, let's download the `ag_news` data set and have a quick look at it.

[ ]:
```
from datasets import load_dataset

# load our data
dataset = load_dataset("ag_news")

# get the index to label mapping
labels = dataset["test"].features["label"].names
```

[2]:
```
import pandas as pd

# quick look at our data
with pd.option_context('display.max_colwidth', None):
    display(dataset["test"].to_pandas().head())
```

```
                                                                                    ␣
↪                                                                                   ␣
↪                                                                                   ␣
↪                                                           text  \
0                                                                                   ␣
↪                                                                         Fears for T␣
↪N pension after talks Unions representing workers at Turner   Newall say they are
↪'disappointed' after talks with stricken parent firm Federal Mogul.
1  The Race is On: Second Private Team Sets Launch Date for Human Spaceflight (SPACE.
↪com) SPACE.com - TORONTO, Canada -- A second\team of rocketeers competing for the  #36;
↪10 million Ansari X Prize, a contest for\privately funded suborbital space flight, has␣
↪officially announced the first\launch date for its manned rocket.
```

(continues on next page)

(continued from previous page)

```
2                                                              Ky. Company Wins Grant␣
↪to Study Peptides (AP) AP - A company founded by a chemistry researcher at the␣
↪University of Louisville won a grant to develop a method of producing better peptides,␣
↪which are short chains of amino acids, the building blocks of proteins.
3    Prediction Unit Helps Forecast Wildfires (AP) AP - It's barely dawn when Mike␣
↪Fitzpatrick starts his shift with a blur of colorful maps, figures and endless charts,␣
↪but already he knows what the day will bring. Lightning will strike in places he␣
↪expects. Winds will pick up, moist places will dry and flames will roar.
4                                                                                    ␣
↪            Calif. Aims to Limit Farm-Related Smog (AP) AP - Southern California's␣
↪smog-fighting agency went after emissions of the bovine variety Friday, adopting the␣
↪nation's first rules to reduce air pollution from dairy cow manure.

   label
0      2
1      3
2      3
3      3
4      3
```

Now we will log the test split of our data set to *Rubrix*, which we will be using for testing our label and downstream models.

[ ]:
```python
import rubrix as rb

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

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

In a second step we log the train split without labels. Remember, our goal is to programmatically build a training set using rules and weak supervision.

[ ]:
```python
# build our training records without labels
records = [
    rb.TextClassificationRecord(
        inputs=record["text"],
        metadata={"split": "unlabelled"},
    )
    for record in dataset["train"].select(range(5000))
]

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

The result of the above is the following dataset in Rubrix, with **127,600 records** (120,000 unlabelled and 7,600 for testing).

You can use the web app to find good rules for programmatic labeling!

### 5.11.4 2. Interactive weak labeling: Finding and defining rules

After logging the dataset, you can find and save rules directly with the UI. Then, you can read the rules with Python to train a label or downstream model, as we'll see in the next step.

### 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, such as FlyingSquid for example (see the *Weak supervision guide* for more details). For convenience, Rubrix defines a simple wrapper over Snorkel's Label Model so it's easier to use with Rubrix weak labels and datasets

Let's first read the rules defined in our dataset and create our weak labels:

[27]:
```
from rubrix.labeling.text_classification import load_rules, WeakLabels

rules = load_rules(dataset="news")

weak_labels = WeakLabels(
    rules=rules,
    dataset="news"
)
weak_labels.summary()
```

```
Preparing rules:   0%|          | 0/18 [00:00<?, ?it/s]
```

```
Applying rules:   0%|          | 0/129100 [00:00<?, ?it/s]
```

[27]:
```
                                       polarity  coverage  overlaps  \
sci*                                 {Sci/Tech}  0.016600  0.003176
dollar*                              {Business}  0.016592  0.006723
*ball                                  {Sports}  0.030132  0.010015
conflict                                {World}  0.003052  0.000999
financ*                              {Business}  0.019620  0.007622
match                                  {Sports}  0.008629  0.002138
goal                                   {Sports}  0.005585  0.001774
election                                {World}  0.017235  0.011789
president*                              {World}  0.053346  0.018590
techn*                               {Sci/Tech}  0.030310  0.012277
software                             {Sci/Tech}  0.030132  0.010380
computer*                            {Sci/Tech}  0.027312  0.011782
game                                   {Sports}  0.038768  0.010333
team                                   {Sports}  0.031867  0.010875
minist*                                 {World}  0.033455  0.008923
stock*                               {Business}  0.041123  0.017800
oil                                  {Business}  0.035817  0.014694
internet                             {Sci/Tech}  0.028234  0.009032
total       {Sports, Sci/Tech, World, Business}  0.378056  0.079171
```

(continues on next page)

(continued from previous page)

```
             conflicts  correct  incorrect  precision
sci*          0.001588      138         33   0.807018
dollar*       0.002990      108         41   0.724832
*ball         0.001425      257         31   0.892361
conflict      0.000287       23          5   0.821429
financ*       0.005298       90         70   0.562500
match         0.000287       78          7   0.917647
goal          0.000395       41          9   0.820000
election      0.002192      128         27   0.825806
president*    0.007188      353        130   0.730849
techn*        0.005143      193         75   0.720149
software      0.003354      209         47   0.816406
computer*     0.003664      192         61   0.758893
game          0.002672      252         79   0.761329
team          0.002874      242         62   0.796053
minist*       0.004191      259         33   0.886986
stock*        0.006933      311         56   0.847411
oil           0.004376      247         60   0.804560
internet      0.002889      216         39   0.847059
total         0.025546     3337        865   0.794146
```

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

# create the label model
label_model = Snorkel(weak_labels)

# fit the model
label_model.fit()

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

### 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 web app, use them as is, or filter them by score, for example.

In this case, we assume the predictions are precise enough and use them without any revision. Our training set has ~38,000 records, which corresponds to all records where the label model has not abstained.

[30]:
```
import pandas as pd

# get records with the predictions from the label model
records = label_model.predict()

# build a simple dataframe with text and the prediction with the highest score
df_train = pd.DataFrame([
```

(continues on next page)

(continued from previous page)

```
    {"text": record.inputs["text"], "label": label_model.weak_labels.label2int[record.
↪prediction[0][0]]}
    for record in records
])

# quick look at our training data with the weak labels from our label model
with pd.option_context('display.max_colwidth', None):
    display(df_train)
```

```
                                                                                        ␣
↪                                                                                       ␣
↪                                                                                       ␣
↪                                                                                       ␣
↪                                                                text  \
0                                                                                       ␣
↪                                                                                       ␣
↪            Biotech Bug Busters Try to Save Venezuela Art Works (Reuters) Reuters -␣
↪Biotechnology is meeting art\in Venezuela as scientists try to save the country's art\
↪treasures from being ruined by its tropical insects, fungi and\humidity.
1                                                                                       ␣
↪                                                                                       ␣
↪            Wolves not entirely to blame for farm losses Paris - Wolves, lions,␣
↪cheetahs and other predators inflict relatively few losses on livestock and farmers␣
↪gain only a temporary boost if these marauders are culled, New Scientist says.
2                                                                                       ␣
↪                                                           EU, U.S. Talks on Aircraft␣
↪Aid Grounded  BRUSSELS (Reuters) - U.S. and EU negotiators disagreed on  Thursday␣
↪about state aid for aircraft rivals Airbus and Boeing,  winding up no closer on a␣
↪sensitive issue that has gathered  steam before the U.S. presidential election.
3                                                                                       ␣
↪                                                                                       ␣
↪                                             Gold Fields Appeal to Exchange on Harmony␣
↪Bid Fails (Update1) Gold Fields Ltd. #39;s appeal to South Africa #39;s stock market␣
↪regulator to block Harmony Gold Mining Co. #39;s 43.9 billion rand (\$7.
4                                                                                       ␣
↪                                                                   Video Games Go Live␣
↪for Annual Awards Show  LOS ANGELES (Reuters) - A felon will host, a Playboy model ␣
↪will work the red carpet, and "the most destructive band in  history" will play on the␣
↪first major live video game awards  show, airing on Spike TV on Tuesday.
...                                                                                     ␣
↪                                                                                       ␣
↪                                                                                       ␣
↪                                                                                       ␣
↪                                                                 ...
45393                                                                                   ␣
↪                                                                                       ␣
↪                    Singapore's Economy Grows in 2004 (AP) AP - Singapore's economy␣
↪expanded 5.4 percent in the fourth quarter from a year ago and grew by 8.1 percent for␣
↪the full year in 2004, the city-state's Ministry of Trade said Monday.
45394                                                                             Krispy␣
↪Kreme Posts Loss, Stock Off 16 Pct  LOS ANGELES (Reuters) - Krispy Kreme Doughnuts Inc.
↪ &lt;A HREF="http://www.investor.reuters.com/FullQuote.aspx?ticker=KKD.N target=/
↪stocks/quickinfo/fullquote"&gt;KKD.N&lt;/A&gt;  on Monday reported a quarterly loss␣
↪due to store closings and  sluggish sales, sending its stock down 16 percent.
```

(continues on next page)

(continued from previous page)

```
45395  Sun partners for high-speed Ethernet Sun Microsystems will integrate drivers for␣
↪S2io's Xframe 10 Gigabit Ethernet Adapter into the Solaris operating system for Sparc,␣
↪AMD Opteron, and Intel Xeon servers. In addition, S2io will partner with Sun to␣
↪develop a TCP/IP offload engine with remote direct memory access functionality to␣
↪enhance performance and scalability in intense computer and server environments.
45396                                                                                   ␣
↪                                                        Taking the Pulse of Planet␣
↪Earth Scientists are planning to take the pulse of the planet -- and more -- in an␣
↪effort to improve weather forecasts, predict energy needs months in advance,␣
↪anticipate disease outbreaks and even tell fishermen where the catch will be abundant.
45397                                                                                   ␣
↪                                                                                       ␣
↪                                                                        Woodward Can␣
↪Make Switch - Hogg Scotland back row forward Allister Hogg sees no reason why England␣
↪coach Sir Clive Woodward cannot make the switch from rugby to football.

       label
0          0
1          0
2          1
3          2
4          3
...      ...
45393      1
45394      2
45395      0
45396      0
45397      3

[45398 rows x 2 columns]
```

[31]:
```
# 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")

# transform data to match our training set format
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`:

[32]:
```
from sklearn.feature_extraction.text import TfidfTransformer, CountVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.pipeline import Pipeline

# define our final classifier
```

(continues on next page)

(continued from previous page)

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

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

```
[32]: Pipeline(steps=[('vect', CountVectorizer()), ('clf', MultinomialNB())])
```

```
[33]: # compute the test accuracy
accuracy = classifier.score(
    X=df_test.text.tolist(),
    y=label_model.weak_labels.annotation()
)

print(f"Test accuracy: {accuracy}")
```

```
Test accuracy: 0.8418681318681319
```

Not too bad!

We have achieved around **0.84 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 improved over the 0.81 accuracy of our Label Model.

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

```
[37]: from sklearn import metrics

# get predictions for the test set
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

       World       0.78      0.86      0.82      2285
      Sports       0.85      0.86      0.85      2278
    Business       0.88      0.67      0.76      2236
    Sci/Tech       0.87      0.98      0.92      2301

    accuracy                           0.84      9100
   macro avg       0.84      0.84      0.84      9100
weighted avg       0.84      0.84      0.84      9100
```

At this point, we could go back to the UI to define more rules for those labels with less performance. Looking at the above table, we might want to add some more rules for increasing the recall of the `Business` label.

### 5.11.8 Summary

In this tutorial, we saw how you can leverage weak supervision to quickly build up a large training data set, and use it for the training of a first lightweight model.

*Rubrix* is a very handy tool to start the weak supervision process by making it easy to find a good set of starting rules, and to reiterate on them dynamically. Since *Rubrix* also provides built-in support for the most common label models, you can get from rules to weak labels in a few straight forward steps. For more suggestions on how to leverage weak labels, you can checkout our *weak supervision guide* where we describe an *interesting approach* to jointly train the label and a transformers downstream model.

### 5.11.9 Next steps

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

#### Rubrix Github repo to stay updated.

#### Rubrix documentation for more guides and tutorials.

#### Join the Rubrix community on Slack

### 5.11.10 Appendix. Create rules and weak labels from Python

For some use cases, you might want to use Python for defining labeling rules and generating weak labels. Rubrix provides you with the ability to define and test rules and labeling functions directly using Python. This might be useful for combining it with rules defined in the UI, and for leveraging structured resources such as lexicons and gazeteers which are easier to use directly a programmatic environment.

In this section, we define the rules we've defined in the UI, this time directly using Python:

[14]:
```
from rubrix.labeling.text_classification import Rule

# define queries and patterns for each category (using ES DSL)
queries = [
  (["money", "financ*", "dollar*"], "Business"),
  (["war", "gov*", "minister*", "conflict"], "World"),
  (["footbal*", "sport*", "game", "play*"], "Sports"),
  (["sci*", "techno*", "computer*", "software", "web"], "Sci/Tech")
]

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

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

# generate the weak labels
weak_labels = WeakLabels(
    rules=rules,
```

(continues on next page)

(continued from previous page)

```
    dataset="news"
)
```

On our machine it took around 24 seconds to apply the rules and to generate weak labels for the 127,600 examples.

Typically, you want to iterate on the rules and check their statistics. For this, you can use `weak_labels.summary` method:

[16]:
```
weak_labels.summary()
```

[16]:
```
                                         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      {World, Sci/Tech, Business, Sports}  0.317022  0.053582   0.019561

           correct  incorrect  precision
money           30         37   0.447761
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.

## 5.12 Explore and analyze `spaCy` NER pipelines

In this tutorial, we will learn to log spaCy Name Entity Recognition (NER) predictions.

This is useful for:

- Evaluating pre-trained models.
- Spotting frequent errors both during development and production.
- Improving your pipelines over time using Rubrix annotation mode.
- Monitoring your model predictions using Rubrix integration with Kibana

Let's get started!

### 5.12.1 Introduction

In this tutorial we will learn how to explore and analyze spaCy NER pipelines in an easy way.

We will load the Gutenberg Time dataset from the Hugging Face Hub and use a transformer-based spaCy model for detecting entities in this dataset and log the detected entities into a Rubrix dataset. This dataset can be used for exploring the quality of predictions and for creating a new training set, by correcting, adding and validating entities.

Then, we will use a smaller spaCy model for detecting entities and log the detected entities into the same Rubrix dataset for comparing its predictions with the previous model. And, as a bonus, we will use Rubrix and spaCy on a more challenging dataset: IMDB.

### 5.12.2 Setup

Rubrix is a free and open-source tool to explore, annotate, and monitor data for NLP projects.

If you are new to Rubrix, visit and  star Rubrix for more materials like and detailed docs: Github repo

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

For this tutorial we also need the third party libraries datasets and of course spaCy together with pytorch, which can be installed via git:

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

### 5.12.3 Our dataset

For this tutorial, we're going to use the Gutenberg Time dataset from the Hugging Face Hub. It contains all explicit time references in a dataset of 52,183 novels whose full text is available via Project Gutenberg. From extracts of novels, we are surely going to find some NER entities.

```
[ ]: from datasets import load_dataset

     dataset = load_dataset("gutenberg_time", split="train", streaming=True)
```

Let's have a look at the first 5 examples of the train set.

[ ]:
```python
import pandas as pd

pd.DataFrame(dataset.take(5))
```

### 5.12.4 Logging spaCy NER entities into Rubrix

#### Using a Transformer-based pipeline

Let's download our Roberta-based pretrained pipeline and instantiate a spaCy `nlp` pipeline with it.

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

[ ]:
```python
import spacy

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

Now let's apply the nlp pipeline to the first 50 examples in our dataset, collecting the **tokens** and **NER entities**.

[ ]:
```python
import rubrix as rb
from tqdm.auto import tqdm

records = []

for record in tqdm(list(dataset.take(50))):
    # We only need the text of each instance
    text = record["tok_context"]

    # spaCy Doc creation
    doc = nlp(text)

    # Entity annotations
    entities = [
        (ent.label_, ent.start_char, ent.end_char)
        for ent in doc.ents
    ]

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

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

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

If you go to the `gutenberg_spacy_ner` dataset in Rubrix you can explore the predictions of this model.

You can:

- Filter records containing specific entity types,
- See the most frequent "mentions" or surface forms for each entity. Mentions are the string values of specific entity types, such as for example "1 month" can be the mention of a duration entity. This is useful for error analysis, to quickly see potential issues and problematic entity types,
- Use the free-text search to find records containing specific words,
- And validate, include or reject specific entity annotations to build a new training set.

### Using a smaller but more efficient pipeline

Now let's compare with a smaller, but more efficient pre-trained model.

Let's first download it:

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

[ ]:
```
import spacy

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

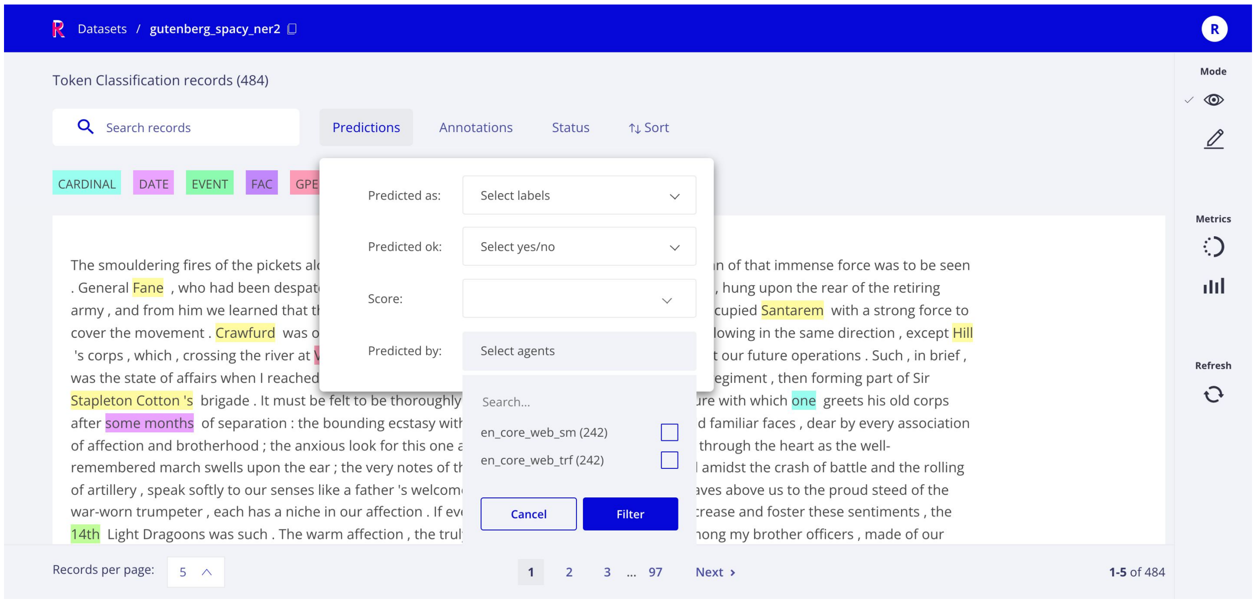
[ ]:
```
records = []    # Creating and empty record list to save all the records

for record in tqdm(list(dataset.take(50))):

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

    # 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="gutenberg_spacy_ner")
```

### 5.12.5 Exploring and comparing `en_core_web_sm` and `en_core_web_trf` models

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

To only see predictions of a specific model, you can use the `predicted by` filter, which comes from the `prediction_agent` parameter of your `TextClassificationRecord`.



### 5.12.6 Explore the IMDB dataset

So far, both **spaCy pretrained models** seem to work pretty well. Let's try with a more challenging dataset, which is more dissimilar to the original training data these models have been trained on.

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

```
[ ]: records = []
     for record in tqdm(imdb.select(range(50))):
         # We only need the text of each instance
         text = record["text"]

         # spaCy Doc creation
         doc = nlp(text)

         # Entity annotations
         entities = [
             (ent.label_, ent.start_char, ent.end_char)
             for ent in doc.ents
         ]

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

         # Rubrix TokenClassificationRecord list
         records.append(
```

(continues on next page)

(continued from previous page)

```
        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 easily check every example by using the filters and search-box.

### 5.12.7 Summary

In this tutorial, you learned how to log and explore differnt `spaCy` NER models with Rubrix. Now you can:

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

### 5.12.8 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 Active learning with ModAL and scikit-learn

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

- We train a spam filter using the YouTube Spam Collection data set.
- For this we embed a lightweight scikit-learn classifier in an active learner via ModAL.
- We design an active learning loop around *Rubrix*, to quickly build up a training data set from scratch.

### 5.13.1 Introduction

> *Active learning is a special case of machine learning in which a learning algorithm can interactively query a user (or some other information source) to label new data points with the desired outputs. In statistics literature, it is sometimes also called optimal experimental design. The information source is also called teacher or oracle.* [Wikipedia]

In this tutorial **our goal is to show you how to incorporate Rubrix into an active learning workflow involving a human in the loop**. We will build a simple text classifier by combining **scikit-learn**, the active learning framework **ModAL** and **Rubrix**. Scikit-learn will provide the model that we will embed in an active learner from ModAL, and you and *Rubrix* will serve as the information source that teach the model to become a sample efficient classifier.

This tutorial is only a proof of concept for educational purposes and to inspire you with some ideas involving interactive learning processes, and how they can help to quickly build a training data set from scratch.

### 5.13.2 Setup

Rubrix, is a free and open-source tool to explore, annotate, and monitor data for NLP projects.

If you are new to Rubrix, check out the Github repository.

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

For this tutorial we also need the third party libraries modAl, scikit-learn and matplotlib (optional), which can be installed via pip:

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

### 5.13.3 1. Loading and preparing data

Rubrix allows you to log and track data for different NLP tasks, such as `Token Classification` or `Text Classification`.

In this tutorial, we will use the YouTube Spam Collection dataset, which is a binary classification task for detecting spam comments in YouTube videos.

Let's load the data and have a look at it:

[1]:
```
import pandas as pd

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

[2]:
```
test_df
```

[2]:
```
                                    COMMENT_ID  \
0            z120djlhizeksdulo23mj5z52vjmxlhrk04
1              z133ibkihkmaj3bfq22rilaxmp2yt54nb
2          z12gxdortqzwhhqas04cfjrwituzghb5tvk0k
3    _2viQ_Qnc6_ZYkMn1fS805Z6oy8ImeO6pSjMLAlwYfM
4          z120s1agtmmetler404cifqbxzvdx15idtw0k
..                                          ...
387        z13pup2w2k3rz1lxl04cf1a5qzavgvv51vg0k
388          z13psdarpuzbjp1hh04cjfwgzonextlhf1w
389        z131xnwierifxxkj204cgvjxyo3oydb42r40k
```

(continues on next page)

(continued from previous page)

```
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 dataset to illustrate the effectiveness of the active learning approach with Rubrix. For the training dataset, we will simply ignore the column and assume that we are gathering training data from scratch.

### 5.13.4 2. Defining our classifier and Active Learner

In this tutorial, we will use a multinomial **Naive Bayes classifier** that is suitable for classification with discrete features (e.g., word counts for text classification):

[3]:
```python
from sklearn.naive_bayes import MultinomialNB

# Define our classification model
classifier = MultinomialNB()
```

Then, we will define our active learner, which uses the classifier as an estimator of the most uncertain predictions:

[4]:
```python
from modAL.models import ActiveLearner

# Define active learner
```

(continues on next page)

(continued from previous page)

```
learner = ActiveLearner(
    estimator=classifier,
)
```

The features for our classifier will be the counts of different word n-grams: that is, for each example we count the number of contiguous sequences of *n* words, where n goes from 1 to 5.

The output of this operation will be matrices of n-gram counts for our train and test data set, where each element in a row equals the counts of a specific word n-gram found in the example:

```
[5]: from sklearn.feature_extraction.text import CountVectorizer

# The resulting matrices will have the shape of (`nr of examples`, `nr of word n-grams`)
vectorizer = CountVectorizer(ngram_range=(1, 5))

X_train = vectorizer.fit_transform(train_df.CONTENT)
X_test = vectorizer.transform(test_df.CONTENT)
```

## 5.13.5 3. Active Learning loop

Now we can start our active learning loop that consists of iterating over following steps:

1. Annotate samples
2. Teach the active learner
3. Plot the improvement (optional)

Before starting the learning loop, let us define two variables:

- the number of instances we want to annotate per iteration,
- a variable to keep track of our improvements by recording the achieved accuracy after each iteration.

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

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

### Step 1: Annotate samples

The first step of the training loop is about annotating *n* examples having the most uncertain prediction. In the first iteration, these will be just random examples, since the classifier is still not trained and we do not have predictions yet.

```
[7]: from sklearn.exceptions import NotFittedError

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

# get predictions for the queried examples
try:
    probabilities = learner.predict_proba(X_train[query_idx])
```

(continues on next page)

(continued from previous page)

```
# For the very first query we do not have any predictions
except NotFittedError:
    probabilities = [[0.5, 0.5]]*n_instances
```

[8]:
```
import rubrix as rb

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

# Log the records
rb.log(records, name="active_learning_tutorial")
```

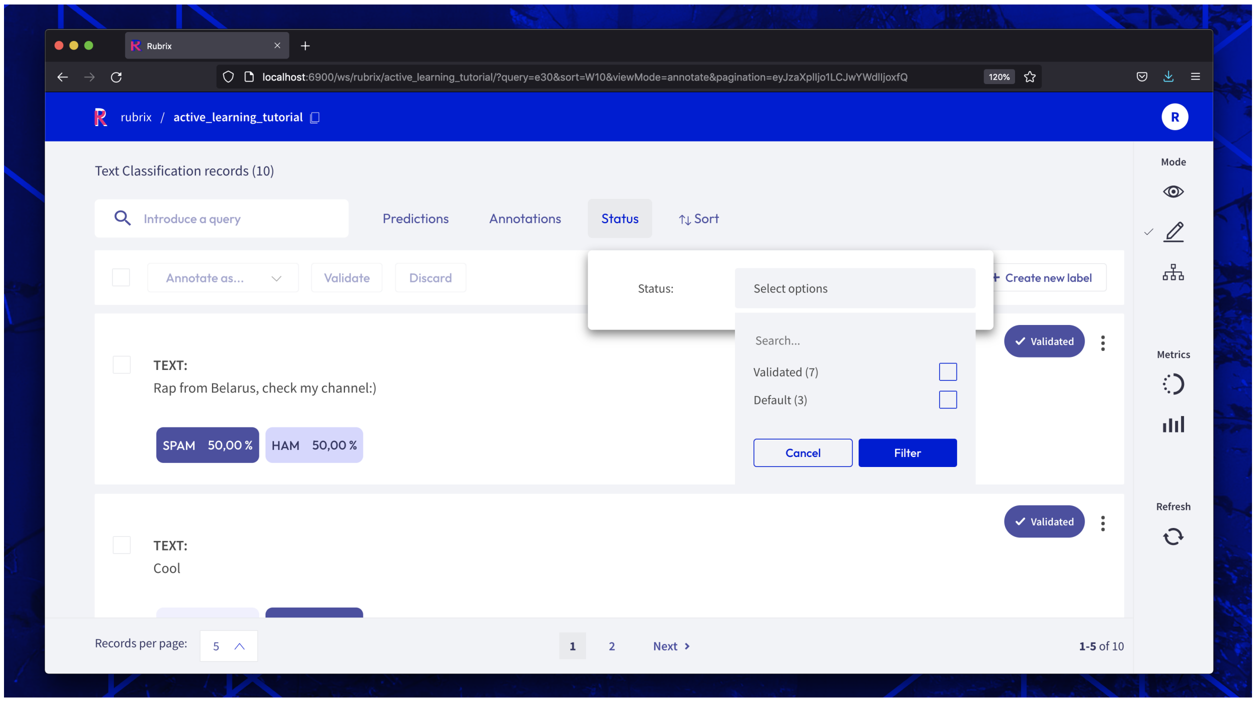
```
  0%|          | 0/10 [00:00<?, ?it/s]
```

```
10 records logged to http://localhost:6900/ws/rubrix/active_learning_tutorial
```

[8]:
```
BulkResponse(dataset='active_learning_tutorial', processed=10, failed=0)
```

After logging the records to Rubrix, we switch over to the UI, where we can find the newly logged examples in the `active_learning_tutorial` dataset.

To only show the examples that are still missing an annotation, you can select **"Default"** in the **Status** filter as shown in the screenshot below. After annotating a few examples you can press the **Refresh** button in the left side bar, to update the view with respect to the filters.



TEXT:
Rap from Belarus, check my channel:)

SPAM 50,00 % HAM 50,00 %

TEXT:
Cool

Once you are done annotating the examples, you can continue with the active learning loop. If your annotations only contained one class, consider increasing the `n_instances` parameter.

### Step 2: Teach the learner

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

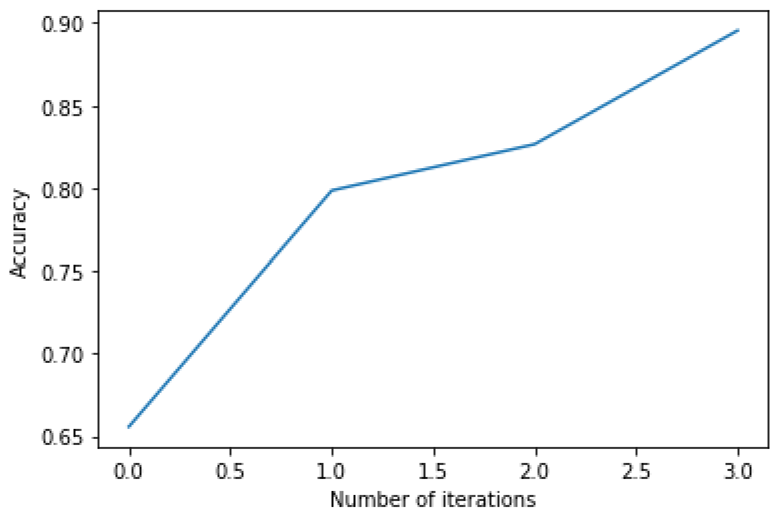
```
[ ]: # Load the annotated records into a pandas DataFrame
     records_df = rb.load("active_learning_tutorial", ids=query_idx.tolist())

     # check if all examples were annotated
     if any(records_df.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.annotation.map(lambda x: int(x == "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.

### Step 3. Plot the improvement (optional)

After a few iterations, we can check the current performance of our classifier by plotting the accuracies. If you think the performance can still be improved, you can **repeat step 1 and 2** and check the accuracy again.

```
[39]: import matplotlib.pyplot as plt

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



### 5.13.6 Summary

In this tutorial we saw how to embed *Rubrix* in an active learning loop, and also how it can help you to **gather a sample efficient data set by annotating only the most decisive examples**. We created a rather minimalist active learning loop, but *Rubrix* does not really care about the complexity of the loop. It will always help you to record and annotate data examples with their model predictions, allowing you to **quickly build up a data set from scratch**.

### 5.13.7 Next steps

#### Rubrix Github repo to stay updated.

#### Rubrix documentation for more guides and tutorials.

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

### 5.13.8 Bonus: Compare query strategies, random vs max uncertainty

In this bonus, we quickly demonstrate the effectiveness of annotating only the most uncertain predictions compared to random annotations. So the next time you want to build a data set from scratch, keep this strategy in mind and maybe use *Rubrix* for the annotation process!

```
[ ]: import numpy as np

n_iterations = 150
n_instances = 10
random_samples = 50


# max uncertainty strategy
accuracies_max = []
for i in range(random_samples):
    train_rnd_df = train_df#.sample(frac=1)
    test_rnd_df = test_df#.sample(frac=1)
    X_rnd_train = vectorizer.transform(train_rnd_df.CONTENT)
    X_rnd_test = vectorizer.transform(test_rnd_df.CONTENT)

    accuracies, learner = [], ActiveLearner(estimator=MultinomialNB())

    for i in range(n_iterations):
        query_idx, _ = learner.query(X_rnd_train, n_instances=n_instances)
        learner.teach(X=X_rnd_train[query_idx], y=train_rnd_df.CLASS.iloc[query_idx].to_
↪list())
        accuracies.append(learner.score(X=X_rnd_test, y=test_rnd_df.CLASS))
    accuracies_max.append(accuracies)

# random strategy
accuracies_rnd = []
for i in range(random_samples):
    accuracies, learner = [], ActiveLearner(estimator=MultinomialNB())

    for random_idx in np.random.choice(X_train.shape[0], size=(n_iterations, n_
↪instances), replace=False):
```

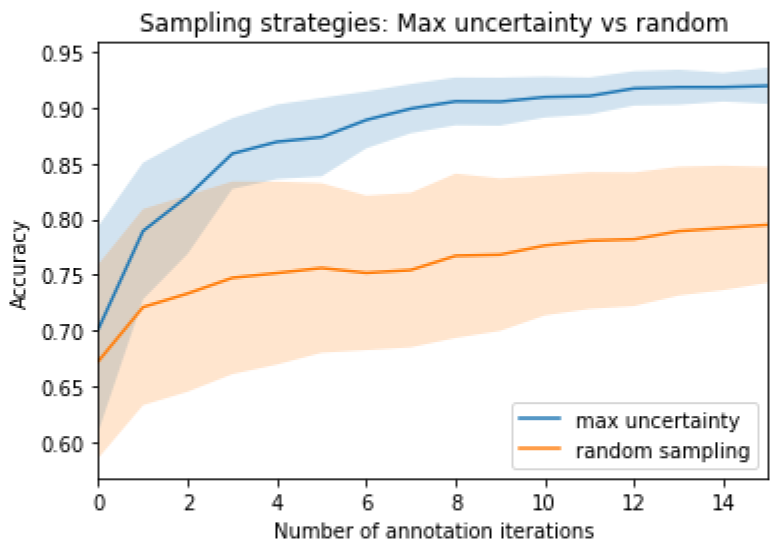
(continues on next page)

(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>
```



### 5.13.9 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 leverage *Rubrix* and cleanlab to find, uncover and correct potential label errors. You can do this following 4 basic steps:

- load a dataset with potential label errors, here we use the ag_news dataset;
- train a model to make predictions for a test set, here we use a lightweight sklearn model;
- use *cleanlab* via *Rubrix* and get potential label error candidates in the test set;
- uncover and correct label errors quickly and comfortably with the *Rubrix* web app;

### 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. They introduce a new principled framework to "identify label errors, characterize label noise, and learn with noisy labels" called **confident learning**. It is open-sourced as the cleanlab Python package that supports finding, quantifying, and learning with label errors in data sets.

*Rubrix* provides built-in support for *cleanlab* and makes it a breeze to find potential label errors in your dataset. In this tutorial we will try to uncover and correct label errors in the well-known ag_news dataset that is often used to benchmark classification models in NLP.

### 5.14.2 Setup

Rubrix, is a free and open-source tool to explore, annotate, and monitor data for NLP projects.

If you are new to Rubrix, check out the Github repository.

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

For this tutorial we also need the third party libraries datasets, sklearn, and cleanlab, which can be installed via pip:

```
[ ]: %pip install datasets scikit-learn cleanlab -qqq
```

### 5.14.3 1. Load datasest

We start by downloading the ag_news dataset via the very convenient datasets library.

```
[ ]: from datasets import load_dataset

     # download data
     dataset = load_dataset('ag_news')
```

We then extract the train and test set, as well as the labels of this classification task. We also shuffle the train set, since by default it is ordered by the classification label.

```
[ ]: # get train set and shuffle
     ds_train = dataset["train"].shuffle(seed=43)

     # get test set
     ds_test = dataset["test"]

     # get classification labels
     labels = ds_train.features["label"].names
```

### 5.14.4 2. Train model

For this tutorial we will use a multinomial **Naive Bayes classifier**, a lightweight and easy to train sklearn model. However, you can use any model of your choice as long as it includes the probabilities for all labels in its predictions.

The features for our classifier will be simply the token counts of our input text.

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

     # define our classifier as a pipeline of token counts + naive bayes model
     classifier = Pipeline([
         ('vect', CountVectorizer()),
         ('clf', MultinomialNB())
     ])
```

After defining our classifier, we can fit it with our train set. Since we are using a rather lightweight model, this should not take too long.

[ ]:
```
# fit the classifier
classifier.fit(
    X=ds_train["text"],
    y=ds_train["label"]
)
```

Let us check how our model performs on the test set.

[ ]:
```
# compute test accuracy
classifier.score(
    X=ds_test["text"],
    y=ds_test["label"],
)
```

We should obtain a decent accuracy of 0.90, especially considering the fact that we only used the token counts as input feature.

### 5.14.5 3. Get label error candidates

As a first step to get label error candidates in our test set, we have to predict the probabilities for all labels.

[ ]:
```
# get predicted probabilities for all labels
probabilities = classifier.predict_proba(ds_test["text"])
```

With the predictions at hand, we create Rubrix records that contain the text input, the prediction of the model, the potential erroneous annotation, and some metadata of your choice.

[ ]:
```
import rubrix as rb

# create records for the test set
records = [
    rb.TextClassificationRecord(
        inputs=data["text"],
        prediction=list(zip(labels, prediction)),
        annotation=labels[data["label"]],
        metadata={"split": "test"}
    )
    for data, prediction in zip(ds_test, probabilities)
]
```

We could log these records directly to Rubrix and conveniently inspect them by eye, checking the annotation of each text input. But here we will use a quicker way by leveraging Rubrix's built-in support for cleanlab. You simply import the find_label_errors function from *Rubrix* and pass in the list of records. That's it.

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

# get records with potential label errors
records_with_label_error = find_label_errors(records)
```

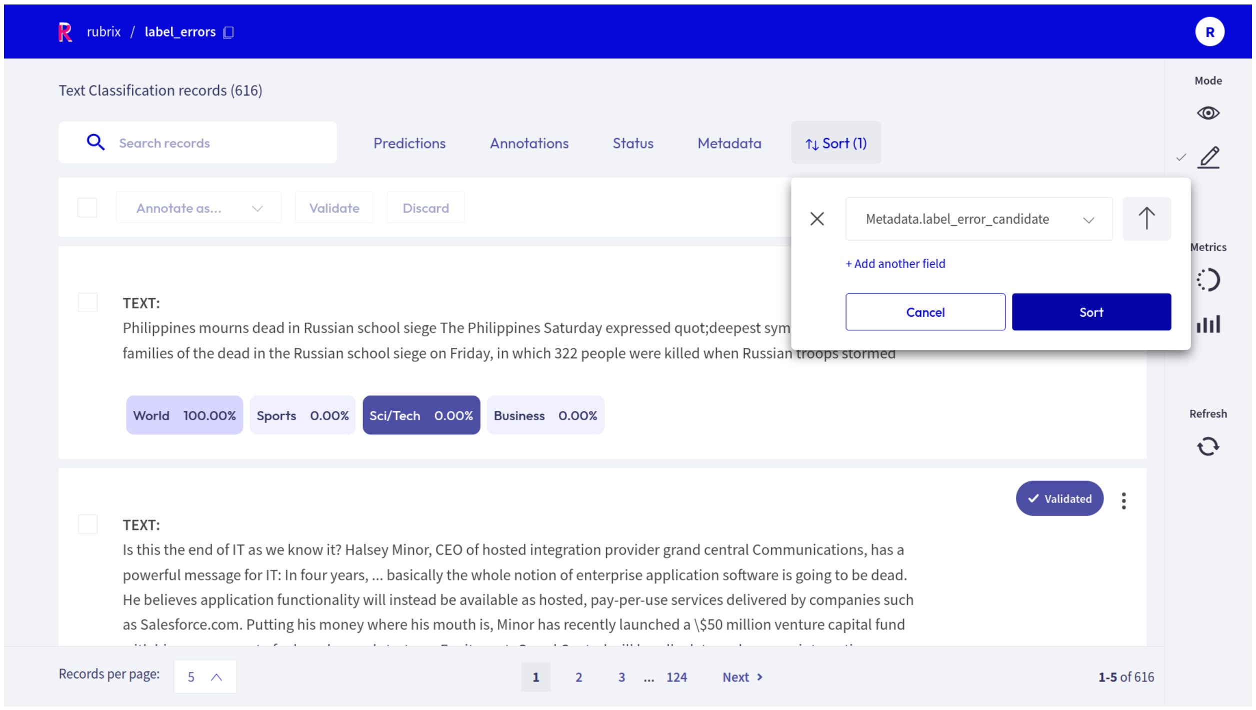
The `records_with_label_error` list contains around 600 candidates for potential label errors, which is more than 8% of our test data.

### 5.14.6 4. Uncover and correct label errors

Now let us log those records to the *Rubrix* web app to conveniently check them by eye, and to quickly correct potential label errors at the same time.

[ ]:
```python
# uncover label errors in the Rubrix web app
rb.log(records_with_label_error, "label_errors")
```

By default the records in the `records_with_label_error` list are ordered by their likelihood of containing a label error. They will also contain a metadata called *"label_error_candidate"* by default, which reflects the order in the list. You can use this field in the *Rubrix* web app to sort the records as shown in the screenshot below.



Text Classification records (616)

TEXT:
Philippines mourns dead in Russian school siege The Philippines Saturday expressed quot;deepest sym
families of the dead in the Russian school siege on Friday, in which 322 people were killed when Russian troops stormed

TEXT:
Is this the end of IT as we know it? Halsey Minor, CEO of hosted integration provider grand central Communications, has a powerful message for IT: In four years, ... basically the whole notion of enterprise application software is going to be dead. He believes application functionality will instead be available as hosted, pay-per-use services delivered by companies such as Salesforce.com. Putting his money where his mouth is, Minor has recently launched a \$50 million venture capital fund

1-5 of 616

We can confirm that the most likely candidates are indeed clear label errors. Towards the end of the candidate list, the examples get more ambiguous, and it is not immediately obvious if the gold annotations are in fact erroneous.

### 5.14.7 Summary

With *Rubrix* you can quickly and conveniently find label errors in your data. The built-in support for cleanlab, together with the optimized user experience of the Rubrix web app, makes the process a breeze, and allows you to efficiently correct label errors on the fly.

In just a few steps you can quickly check if your test data set is seriously affected by label errors and if your benchmarks are really meaningful in practice. Maybe your less complex models turns out to beat your resource hungry super model, and the deployment process just got a little bit easier .

Although we only used a sklearn model in this tutorial, Rubrix does not care about the model architecture or the framework you are working with. It just cares about the underlying data and allows you to put more humans in the loop of your AI Lifecycle.

### 5.14.8 Next steps

#### Rubrix Github repo to stay updated.

#### Rubrix documentation for more guides and tutorials.

#### Join the Rubrix community on Slack

### 5.14.9 Bonus: Find label errors in your train data using cross-validation

In order to check your training data for label errors, you can fall back to the cross-validation technique to get out-of-sample predictions. With a classifier from sklearn, cross-validation is really easy and you can do it conveniently in one line of code. Afterwards, the steps of creating *Rubrix* records, finding label error candidates, and uncovering them are the same as shown in the tutorial above.

```
[ ]: from sklearn.model_selection import cross_val_predict

# get predicted probabilities for the whole dataset via cross validation
cv_probs = cross_val_predict(
    classifier,
    X=ds_train["text"] + ds_test["text"],
    y=ds_train["label"] + ds_test["label"],
    cv=int(len(ds_train) / len(ds_test)),
    method="predict_proba",
    n_jobs=-1
)
```

```
[ ]: # create records for the training set
records = [
    rb.TextClassificationRecord(
        inputs=data["text"],
        prediction=list(zip(labels, prediction)),
        annotation=labels[data["label"]],
        metadata={"split": "train"}
    )
    for data, prediction in zip(ds_train, cv_probs)
]
```

```
[ ]: # uncover label errors for the train set in the Rubrix web app
rb.log(find_label_errors(records), "label_errors_in_train")
```

Here we find around 9400 records with potential label errors, which is also around 8% with respect to the train data.

## 5.15 Zero-shot Named Entity Recognition with Flair

In this tutorial you will learn how to analyze and validate NER predictions from the new zero-shot model provided by the Flair NLP library with Rubrix.

- Useful for quickly bootstrapping a training set (using Rubrix *Annotation Mode*) as well as integrating with weak-supervision workflows.
- We will use a challenging, exciting dataset: wnut_17 (more info below).
- You will be able to see and work with the obtained predictions.

### 5.15.1 Introduction

This tutorial will show you how to work with Named Entity Recognition (NER), Flair and Rubrix. But, what is NER?

According to Analytics Vidhya, "NER is a **natural language processing technique** that can automatically scan entire articles and pull out some fundamental entities in a text and classify them into predefined categories". These entities can be names, quantities, dates and times, amounts of money/currencies, and much more.

On the other hand, Flair is a library which facilitates the application of NLP models to NER and other NLP techniques in many different languages. It is not only a powerful library, but also intuitive.

Thanks to these resources and the *Annotation Mode* of *Rubrix*, we can quickly build up a data set to train a domain-specific model.

### 5.15.2 Setup

Rubrix, is a free and open-source tool to explore, annotate, and monitor data for NLP projects.

If you are new to Rubrix, check out the Github repository .

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

For this tutorial we also need the third party libraries datasets and flair, which can be installed via pip:

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

### 5.15.3 1. Load the `wnut_17` dataset

In this example, we'll use a challenging NER dataset, the **"WNUT 17: Emerging and Rare entity recognition"** , which focuses on unusual, previously-unseen entities in the context of emerging discussions. This dataset is useful for getting a sense of the quality of our zero-shot predictions.

Let's load the test set from the Hugging Face Hub:

[ ]:
```
from datasets import load_dataset

# download data set
dataset = load_dataset("wnut_17", split="test")
```

[2]:
```
# define labels
labels = ['corporation', 'creative-work', 'group', 'location', 'person', 'product']
```

### 5.15.4 2. Configure Flair TARSTagger

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

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

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

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

Let's test it with one example!

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

# wrap our tokens in a flair Sentence
sentence = Sentence(" ".join(dataset[0]['tokens']))
```

[6]:
```
# add predictions to our sentence
tars.predict(sentence)

# extract predicted entities into a list of tuples (entity, start_char, end_char)
[
    (entity.get_labels()[0].value, entity.start_pos, entity.end_pos)
    for entity in sentence.get_spans("ner")
]
```

[6]:
```
[('location', 100, 107)]
```

### 5.15.5 3. Predict over `wnut_17` and log into `rubrix`

Now, let's log the predictions in Rubrix:

[ ]:
```
import rubrix as rb

# build records for the first 100 examples
records = []
for record in dataset.select(range(100)):
    input_text = " ".join(record["tokens"])

    sentence = Sentence(input_text)
    tars.predict(sentence)
    prediction = [
        (entity.get_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],
```

(continues on next page)

(continued from previous page)

```
            prediction=prediction,
            prediction_agent="tars-ner",
        )
    )

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

Now you can see the results obtained! With the annotation mode, you can change, add, validate or discard your results. Statistics are also available, to better monitor your records!

### 5.15.6 Summary

Getting predictions with a zero-shot approach can be very helpful to guide humans in their annotation process. Especially for NER tasks, Rubrix makes it very easy to explore and correct those predictions thanks to its **Annotation Mode** .

### 5.15.7 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.

## 5.16 Clean labels using your model loss

In this tutorial, we will learn to introduce a simple technique for error analysis, **using model loss to find potential training data errors**.

- This technique is shown using a **fine-tuned text classifier from the Hugging Face Hub** on the **AG News dataset**.
- Using Rubrix, we will verify **more than 50 mislabelled examples on the training set** of this well-known NLP benchmark.
- This trick is useful for **model training with small and noisy datasets**.
- This trick is complementary with other "data-centric" ML methods such as `cleanlab` (see this *Rubrix tutorial*).

### 5.16.1 Introduction

This tutorial explains a simple trick you can leverage with Rubrix for finding potential errors in training data: *using your model loss to identify label errors or ambiguous examples*. This trick is not new (those who've worked with fastai know how useful the `plot_top_losses` method is). Even Andrej Karpathy tweeted about this some time ago:

When you sort your dataset descending by loss you are guaranteed to find something unexpected, strange and helpful.

— Andrej Karpathy (@karpathy) October 2, 2020

The technique is really simple: if you are training a model with a training set, train your model, and you apply your model to the training set to **compute the loss for each example in the training set**. If you sort your dataset examples by loss, examples with the highest loss are the most ambiguous and difficult to learn.

This technique can be used for **error analysis during model development** (e.g., identifying tokenization problems), but it turns out is also a really simple technique for **cleaning up your training data, during model development or after training data collection activities**.

In this tutorial, we'll use this technique with a well-known text classification benchmark, the AG News dataset. After computing the losses, we'll use Rubrix to analyse the highest loss examples. In less than 5 minutes, we manually check and relabel the first 50 examples. In fact, the first 50 examples with the highest loss, are all incorrect in the original training set. If we visually inspect further examples, we still find label errors in the top 500 examples.

#### Why it's important

1. **Machine learning models are only as good as the data they're trained on**. Almost all training data source can be considered *"noisy"* (e.g., crowd-workers, annotator errors, weak supervision sources, data augmentation, etc.)
2. With this simple technique **we're able to find more than 50 label errors on a widely-used benchmark in less than 5 minutes** (your dataset will probably be noisier!).
3. With advanced model architectures widely-available, **managing, cleaning, and curating data is becoming a key step for making robust ML applications**. A good summary of the current situation can be found in the website of the Data-centric AI NeurIPS Workshop.
4. This simple trick **can be used across the whole ML lifecycle** and not only for finding label errors. With this trick you can improve data preprocessing, tokenization, and even your model architecture.

### 5.16.2 Setup

Rubrix, is a free and open-source tool to explore, annotate, and monitor data for NLP projects.

If you are new to Rubrix, check out the Github repository.

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

For this tutorial we will also need the third party libraries transformers and datasets, as well as PyTorch, which can be installed via pip:

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

### 5.16.3 Preliminaries

1. A model fine-tuned with the AG News dataset (you could train your own model if you wish).
2. The AG News train split (the same trick could and should be applied to validation and test splits).
3. Rubrix for logging, exploring, and relabeling wrong examples (we provide a pre-computed datasets so feel free to skip to this step)

### 5.16.4 1. Load the fine-tuned model and the training dataset

Now, we will load the AG News dataset. But first, we need to define and set the device, the model and the tokenizer:

```
[ ]: import torch

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

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

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

```
[ ]: from datasets import load_dataset

     # load the training split
     ds = load_dataset('ag_news', split='train')
```

```
[ ]: # tokenize and encode the training set
     def tokenize_and_encode(batch):
         return tokenizer(batch['text'], truncation=True)

     ds_enc = ds.map(tokenize_and_encode, batched=True)
```

### 5.16.5 2. Computing the loss

The following code will compute the loss for each example using our trained model. This process is taken from the very well-explained blog post by Lewis Tunstall: "Using data collators for training and error analysis", where he explains this process for error analysis during model training.

In our case, we instantiate a data collator directly, while he uses the Data Collator from the Trainer directly:

```
[ ]: from transformers.data.data_collator import DataCollatorWithPadding

     # create the data collator for inference
     data_collator = DataCollatorWithPadding(tokenizer, padding=True)
```

```
[ ]: # function to compute the loss example-wise
     def loss_per_example(batch):
         batch = data_collator(batch)
         input_ids = torch.tensor(batch["input_ids"], device=device)
         attention_mask = torch.tensor(batch["attention_mask"], device=device)
         labels = torch.tensor(batch["labels"], device=device)

         with torch.no_grad():
             output = model(input_ids, attention_mask)
             batch["predicted_label"] = torch.argmax(output.logits, axis=1)
             # compute the probabilities for logging them into Rubrix
             batch["predicted_probas"] = torch.nn.functional.softmax(output.logits, dim=0)
```

(continues on next page)

(continued from previous page)

```
    # don't reduce the loss (return the loss for each example)
    loss = torch.nn.functional.cross_entropy(output.logits, labels, reduction="none")
    batch["loss"] = loss

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

    return batch
```

Now, it is time to turn the dataset into a Pandas dataframe and sort this dataset by descending loss:

```
[ ]: import pandas as pd

losses_ds = ds_enc.remove_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.
```

(continues on next page)

(continued from previous page)

```
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.
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]:
```python
# 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.

### 5.16.6 3. Log high loss examples into Rubrix

Using the amazing Hugging Face Hub we've shared the resulting dataset, which you can find here and load directly using the datasets library

Now, we log the first 500 examples into a Rubrix dataset:

[ ]:
```python
# if you have skipped the first two steps you can load the dataset here:
import pandas as pd
from datasets import load_dataset

dataset = load_dataset("dvilasuero/ag_news_training_set_losses", split='train')
losses_df = dataset.to_pandas()


ds = load_dataset('ag_news', split='test') # only for getting the label names
```

[7]:
```python
import rubrix as rb
# creates a Text classification record for logging into Rubrix
def make_record(row):

```

(continues on next page)

(continued from previous page)

```
    return rb.TextClassificationRecord(
        inputs={"text": row.text},
        # this is the "gold" label in the original dataset
        annotation=[(ds.features['label'].names[row.label])],
        # this is the prediction together with its probability
        prediction=[(ds.features['label'].names[row.predicted_label], row.predicted_
↪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-ner-agnews",
        # source of the gold label
        annotation_agent="ag_news_benchmark"
    )
```

[8]:
```
# if you want to log the full dataset remove the indexing
top_losses = losses_df.sort_values("loss", ascending=False)[0:499]

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

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

### 5.16.7 4. Using Rubrix Webapp for inspection and relabeling

In this step, we have a Rubrix Dataset available for exploration and annotation. A useful feature for this use case is **Sorting**. With Rubrix you can sort your examples by combining different fields, both from the standard fields (such as `score`) and custom fields (via the metadata fields). In this case, we've logged the loss so we can order our training examples by loss in descending order (showing higher loss examples first).

For preparing this tutorial, we have manually checked and relabelled the first 50 examples. Moreover, we've shared this re-annotated dataset in the Hugging Face Hub. In the next section, we show you how easy is to share Rubrix Datasets in the Hub.

### 5.16.8 5. Sharing the dataset in the Hugging Face Hub

Let's first load the re-annotated examples. Re-labelled examples are marked as `annotated_by` the user `rubrix`, which is the default user when launching Rubrix with Docker. We can retrieve only these records using the `query` param as follows:

[11]:
```
import rubrix as rb
dataset = rb.load("ag_news_error_analysis", query="annotated_by:rubrix")

# let's do some transformations before uploading the dataset
dataset['text'] = dataset.inputs.transform(lambda r: r['text'])
dataset['loss'] = dataset.metadata.transform(lambda r: r['loss'])
dataset = dataset.rename(columns={"annotation": "corrected_label"})

dataset.head()
```

[11]:
```
                                              inputs  \
0  {'text': 'Top nuclear official briefs Majlis c...
1  {'text': 'Fischer Delivers Strong Message in S...
2  {'text': 'The Politics of Time and Dispossessi...
3  {'text': 'Hadash Party joins prisoners #39; st...
4  {'text': 'China May Join \$10Bln Sakhalin-2 Ru...

                   prediction corrected_label  \
0     [(World, 0.1832696944)]           World
1     [(World, 0.0695228428)]           World
2   [(Sci/Tech, 0.100481838)]        Sci/Tech
3     [(World, 0.1749624908)]           World
4  [(Business, 0.1370282918)]        Business

                             prediction_agent annotation_agent  multi_label  \
0  andi611/distilbert-base-uncased-ner-agnews           rubrix        False
1  andi611/distilbert-base-uncased-ner-agnews           rubrix        False
2  andi611/distilbert-base-uncased-ner-agnews           rubrix        False
3  andi611/distilbert-base-uncased-ner-agnews           rubrix        False
4  andi611/distilbert-base-uncased-ner-agnews           rubrix        False

  explanation                                    id                metadata  \
0        None  071a1014-71e7-41f4-83e4-553ba47610cf  {'loss': 7.6656146049}
1        None  07c8c4f6-3288-46f4-a618-3da4a537e605  {'loss': 7.9892320633}
2        None  0965a0d1-4886-432a-826a-58e99dfd9972   {'loss': 7.133708477}
3        None  09fc7065-a2c8-4041-adf8-34e029a7fde0   {'loss': 7.339015007}
4        None  1ef97c49-2f0f-43be-9b28-80a291cb3b1d   {'loss': 7.321100235}

      status event_timestamp metrics  \
0  Validated            None      {}
1  Validated            None      {}
2  Validated            None      {}
3  Validated            None      {}
4  Validated            None      {}

                                                text      loss
0  Top nuclear official briefs Majlis committee T...  7.665615
1  Fischer Delivers Strong Message in Syria Germa...  7.989232
2  The Politics of Time and Dispossession Make a ...  7.133708
3  Hadash Party joins prisoners #39; strike for 2...  7.339015
4  China May Join \$10Bln Sakhalin-2 Russia said ...  7.321100
```

[12]:
```python
# let's add the original dataset labels to share them together with the corrected ones
# we sort by ascending loss our corrected dataset
dataset = dataset.sort_values("loss", ascending=False)

# we add original labels in string form
id2label = list(dataset.corrected_label.unique())
original_labels = [id2label[i] for i in top_losses[0:50].label.values]
dataset["original_label"] = original_labels
```

Now let's transform this into a `Dataset` and define the features schema:

[13]:
```python
from datasets import Dataset, Features, Value, ClassLabel

ds = dataset[['text', 'corrected_label', 'original_label']].to_dict(orient='list')

hf_ds = Dataset.from_dict(
    ds,
    features=Features({
        "text": Value("string"),
        "corrected_label": ClassLabel(names=list(dataset.corrected_label.unique())),
        "original_label": ClassLabel(names=list(dataset.corrected_label.unique()))
    })
)
```

[19]:
```python
hf_ds.features
```

[19]:
```
{'text': Value(dtype='string', id=None),
 'corrected_label': ClassLabel(num_classes=4, names=['World', 'Business', 'Sports', 'Sci/
↪Tech'], names_file=None, id=None),
 'original_label': ClassLabel(num_classes=4, names=['World', 'Business', 'Sports', 'Sci/
↪Tech'], names_file=None, id=None)}
```

Uploading the dataset with the `push_to_hub` method is as easy as:

[21]:
```python
hf_ds.push_to_hub("Recognai/ag_news_corrected_labels")
```

```
Pushing dataset shards to the dataset hub:   0%|          | 0/1 [00:00<?, ?it/s]
```

Now the dataset is publicly available at the Hub!

### 5.16.9 Summary

In this tutorial we say how you can leverage the **model loss** to find potential label errors in your training data set. The *Rubrix* web app makes it very convenient to sort your data by loss, inspect single records by eye, and allows you to easily correct label errors on the fly.

### 5.16.10 Next steps

If you are interested in the topic of training data curation and denoising datasets, check out the tutorial for using *Rubrix with cleanlab*.

#### Rubrix Github repo to stay updated.

#### Rubrix documentation for more guides and tutorials.

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

## 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. It will walk you through 4 basic steps:

- Load the model you want to use.
- Convert model output to Rubrix format.
- Create a FastAPI endpoint.
- Add middleware to automate logging to Rubrix

## 5.17.1 Introduction

Models are often deployed via an HTTP API endpoint that is called by a client to obtain the model's predictions. With FastAPI and *Rubrix* you can easily monitor those predictions and log them to a *Rubrix* dataset. Due to its human-centric UX, *Rubrix* datasets can be comfortably viewed and explored by any team member of your organization. But *Rubrix* also provides automatically computed metrics, both of which help you to keep track of your predictor and spot potential issues early on.

FastAPI and *Rubrix* allow you to deploy and monitor any model you like, but in this tutorial we will focus on the two most common frameworks in the NLP space: spaCy and transformers. Let's get started!

## 5.17.2 Setup

Rubrix is a free and open-source tool to explore, annotate, and monitor data for NLP projects.

If you are new to Rubrix, check out the Github repository.

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

Apart from Rubrix, we'll need a few third party libraries that can be installed via pip:

[ ]:
```
%pip install fastapi uvicorn[standard] spacy transformers[torch] -qqq
```

## 5.17.3 1. Loading models

As a first step, let's load our models. For spacy we need to first download the model before we can instantiate a spacy pipeline with it. Here we use the small English model `en_core_web_sm`, but you can choose any available model on their hub.

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

[ ]:
```
import spacy

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

The "text-classification" pipeline by transformers download's the model for you and by default it will use the `distilbert-base-uncased-finetuned-sst-2-english` model. But you can instantiate the pipeline with any compatible model on their hub.

[ ]:
```
from transformers import pipeline

transformers_pipeline = pipeline("text-classification", return_all_scores=True)
```

For more 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:

```
[5]: from pprint import pprint

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

```
[[{'label': 'NEGATIVE', 'score': 0.0003226407279726118},
  {'label': 'POSITIVE', 'score': 0.9996774196624756}]]
```

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

## 5.17.4 2. 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.

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

```
[{'labels': ['NEGATIVE', 'POSITIVE'],
  'scores': [0.0003226407279726118, 0.9996774196624756]}]
```

## 5.17.5 3. Create prediction endpoint

```
[ ]: from fastapi import FastAPI
     from typing import List

     app_transformers = FastAPI()

     # prediction endpoint using transformers pipeline
     @app_transformers.post("/")
     def predict_transformers(batch: List[str]):
         predictions = transformers_pipeline(batch)
         return [
             {
                 "labels": [p["label"] for p in prediction],
                 "scores": [p["score"] for p in prediction],
             }
             for prediction in predictions
         ]
```

### 5.17.6 4. Add Rubrix logging middleware to the application

[ ]:
```python
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 5. Do the same for spaCy

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

#### Mapper

[ ]:
```python
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

[ ]:
```python
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)

```python
# 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 6. Putting it all together

Now we can combine everything in order to see our results!

[ ]:
```python
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 learned 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 Github repo to stay updated.

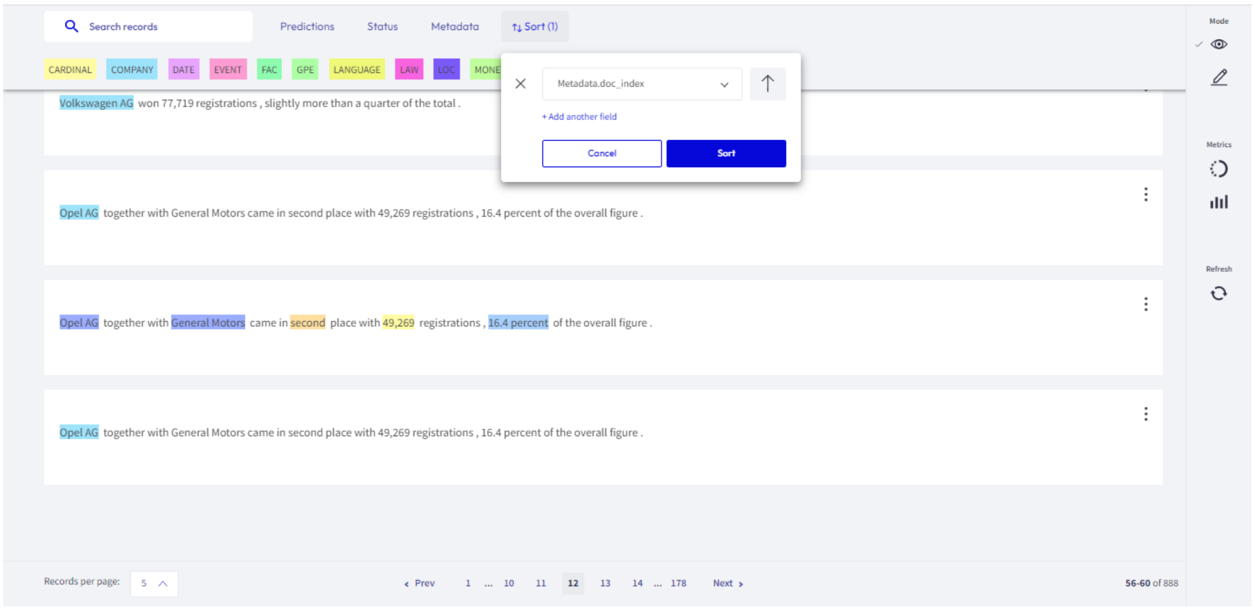
#### Rubrix documentation for more guides and tutorials.

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

## 5.18 Weakly supervised NER with `skweak`

This tutorial will walk you through the process of using Rubrix to improve weak supervision and data programming workflows with the skweak library.

- Using *skweak* and *spaCy*, we define heuristic labeling functions for the CoNLL 2003 dataset.
- We combine those with a pretrained NER model and use an aggregation model from skweak to obtain noisy NER annotations.
- We then log the documents to Rubrix and visualize the results via its web app.
- With the noisy labels, we fine-tune a spaCy NER model.
- Adding labeling functions from gazetteers to our aggregation model, we revise the updated noisy annotation with Rubrix, and retrain the spaCy model.
- Instead of a spaCy model, we fine-tune a transformers model with the help of the *simpletransformers* library.



Volkswagen AG won 77,719 registrations , slightly more than a quarter of the total .

Opel AG together with General Motors came in second place with 49,269 registrations , 16.4 percent of the overall figure .

Opel AG together with General Motors came in second place with 49,269 registrations , 16.4 percent of the overall figure .

Opel AG together with General Motors came in second place with 49,269 registrations , 16.4 percent of the overall figure .

## 5.18.1 Introduction

Our goal is to show you how you can incorporate Rubrix into data programming workflows to programatically build training data with a human-in-the-loop approach. We will use the skweak library.

### What is weak supervision? and skweak?

Weak supervision is a branch of machine learning based on getting lower quality labels more efficiently. We can achieve this by using skweak, a library for programmatically building and managing training datasets without manual labeling.

### This tutorial

In this tutorial, we bring content from the Quick Start Named-Entity Recognition and the Step-by-step NER tutorials from skweak's documentation and show you how to extend weak supervision workflows with Rubrix.

We will take records from the CoNLL 2003 dataset and build our own annotations with `skweak`. Then we are going to evaluate NER models trained on our annotations on the standard development set of CoNLL 2003.

## 5.18.2 Setup

Rubrix, is a free and open-source tool to explore, annotate, and monitor data for NLP projects.

If you are new to Rubrix, check out the Github repository.

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

For this tutorial we also need some third party libraries that can be installed via pip:

```
[5]: %pip install skweak spacy simpletransformers -qqq
     !python -m spacy download en_core_web_sm -qqq
     !python -m spacy download en_core_web_md -qqq
```

### 5.18.3 Named Entity Recognition with skweak and Rubrix

Rubrix allows you to log and track data for different NLP tasks (such as `Token Classification` or `Text Classification`).

In this tutorial, we will use the English portion of the CoNLL 2003 dataset, a standard Named Entity Recognition benchmark.

#### The dataset

In this tutorial we'll be using skweak's data programming methods for programatically building a training set with the help of Rubrix for analizing and reviewing data. We'll then train a model with this training set.

Although the gold labels for the training set of CoNLL 2003 are already known, we will purposefully ignore them, as our goal in this tutorial is to build our own annotations and see how well they perform on the development set.

We will load the CoNLL 2003 dataset with the help of the `datasets` library.

[ ]:
```python
from datasets import load_dataset

dataset = load_dataset("conll2003")
```

#### Preprocessing

Next, we simplify the tagset by replacing numbers with tags and removing the BIO encoding.

[ ]:
```python
tag_set = {'O': 0, 'B-PER': 1, 'I-PER': 2, 'B-ORG': 3, 'I-ORG': 4, 'B-LOC': 5, 'I-LOC':␣
↪6, 'B-MISC': 7, 'I-MISC': 8}
tag_set = { v:k for k,v in tag_set.items() }

def convert_ner_tags(record, tag_set=None):
    record['ner_tags'] = [ tag_set[x] for x in record['ner_tags'] ]
    return record

def strip_BI_tags(record):
    record['ner_tags'] = [ x.lstrip('B-').lstrip('I-') for x in record['ner_tags'] ]
    return record

dataset = dataset\
    .map(convert_ner_tags, fn_kwargs={"tag_set": tag_set})\
    .map(strip_BI_tags)
```

We will now convert the training and validation splits of our dataset into spaCy Doc objects.

spaCy demands strings to be given as inputs to a tokenizer. However, as our dataset is already tokenized, we bypass this restriction by using our own tokenizer and encapsulating our tokens in a class that inherits from `str`.

[8]:
```python
import spacy
from spacy.tokens import Doc
from dataclasses import dataclass

@dataclass
class Record(str):
    tokens: list
```

(continues on next page)

(continued from previous page)

```
def custom_tokenizer(text):
    return Doc(nlp.vocab, text.tokens)

nlp = spacy.load("en_core_web_sm", disable=["ner", "lemmatizer"])
nlp.tokenizer = custom_tokenizer

training_set = [ Record(x) for x in dataset["train"]["tokens"] ]
dev_set = [ Record(x) for x in dataset["validation"]["tokens"] ]

train_docs = list(nlp.pipe(training_set))
dev_docs = list(nlp.pipe(dev_set))
```

The gold labels must also be added to our validation Doc objects, so we can evaluate our model during training.

[9]:
```
from spacy.tokens import Span
from itertools import groupby
from dataclasses import dataclass
import copy

@dataclass
class IndexedLabel:
    index: int
    label: str

def annotate_labels_to_doc(doc, labels, null_label="O"):
    labels = [ IndexedLabel(idx, item) for idx, item in enumerate(labels) ]
    grouped_labels = [ list(group[1]) for group in groupby(labels) ]
    span_objects = [ Span(doc, item[0].index, item[-1].index + 1, item[0].label) for␣
↪item in grouped_labels ]
    span_objects = [ span for span in span_objects if span.label_ != null_label ]
    doc.set_ents(span_objects)
    return doc

dev_labels = [ x for x in dataset["validation"]["ner_tags"] ]

for idx, label_sequence in enumerate(dev_labels):
    dev_docs[idx] = annotate_labels_to_doc(dev_docs[idx], label_sequence)
```

### Labeling functions

Labelling functions (LFs) are at the core of skweak. They take a Doc as an input and return a list of spans with their associated labels.

In this tutorial, we will first define the LFs from the skweak tutorial and then show you how you can use Rubrix to enhance this type of weak-supervision workflow.

### Heuristics

One simple type of labelling functions are heuristics. For instance, we can write that commercial companies may be recognized by their legal suffix (such as Corp.):

[10]:
```python
import skweak

def company_detector_fun(doc):
    for chunk in doc.noun_chunks:
        if chunk[-1].lower_.rstrip(".") in {'corp', 'inc', 'ltd', 'llc', 'sa', 'ag'}:
            yield chunk.start, chunk.end, "COMPANY"

# We create the labelling function by giving it a name, and a function to apply
company_detector = skweak.heuristics.FunctionAnnotator("company_detector", company_
↪detector_fun)
```

We can write another example of heuristics for non-commercial organisations by looking for the occurrence of words that are quite typical of public organisations or NGOs:

[11]:
```python
OTHER_ORG_CUE_WORDS = {"University", "Institute", "College", "Committee", "Party",
↪"Agency",
                       "Union", "Association", "Organization", "Court", "Office",
↪"National"}
def other_org_detector_fun(doc):
    for chunk in doc.noun_chunks:
        if any([tok.text in OTHER_ORG_CUE_WORDS for tok in chunk]):
            yield chunk.start, chunk.end, "OTHER_ORG"

# We create the labelling function
other_org_detector = skweak.heuristics.FunctionAnnotator("other_org_detector", other_org_
↪detector_fun)
```

### NER models

We can also take advantage of machine learning models trained from data of related domains. Here, we will use a spacy model trained on OntoNotes 5.0 to get more named entities.

[12]:
```python
ner = skweak.spacy.ModelAnnotator("spacy", "en_core_web_sm")
```

Finally, we run our annotators over the documents.

[13]:
```python
train_docs = list(company_detector.pipe(train_docs))
train_docs = list(other_org_detector.pipe(train_docs))
train_docs = list(ner.pipe(train_docs))
```

### Aggregation

Once the labelling functions have been applied, we must then aggregate their results, so that we obtain a single annotation for each document.

This can be done in skweak through a Hidden Markov Model. Here we use a CustomHMM class as a workaround for tokens with impossible states, as suggested in this issue.

[ ]:
```
from typing import Dict
import numpy as np
import skweak

class CustomHMM(skweak.aggregation.HMM):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

    def _compute_log_likelihood(self, X: Dict[str, np.ndarray]) -> np.ndarray:
        """Computes the log likelihood for the observed sequence"""

        logsum = np.float32()
        for source in X:
            # Include the weights to the probabilities
            probs = (self.emit_probs[source]  # type:ignore
                     ** self.weights.get(source, 1))

            # We compute the likelihood of each state given the observations
            probs = np.dot(X[source], probs.T)

            # Impossible states have a logprob of -inf
            log_probs = np.ma.log(probs).filled(-np.inf)

            logsum += log_probs

        # We also add a constraint that the probability of a state is zero
        # if no labelling functions observes it
        X_all_obs = np.zeros(logsum.shape, dtype=bool)  # type: ignore
        for source in self.emit_counts:
            if source in X:
                if "O" in self.out_labels:
                    X_all_obs += X[source][:, :len(self.out_labels)]
                else:
                    X_all_obs += X[source][:, 1:len(self.out_labels)+1]
        logsum = np.where(X_all_obs, logsum, -100000.0)
        return logsum  # type: ignore

# We define the aggregation model
hmm_model = CustomHMM("hmm", ["COMPANY", "OTHER_ORG"])

# We indicate that "ORG" is an underspecified value, which may
# represent either COMPANY or OTHER_ORG
hmm_model.add_underspecified_label("ORG", ["COMPANY", "OTHER_ORG"])

# And run the estimation
train_docs = hmm_model.fit_and_aggregate(train_docs)
```

### Visualization with Rubrix

We can use Rubrix to visualize the outputs of our aggregation model.

First we define a doc_logger function that will log the predictions produced by all our annotators to Rubrix.

```
[15]: from tqdm import tqdm
import rubrix as rb

def doc_logger(texts, docs, rubrix_dataset="conll_2003"):
    records = []
    for idx, doc in enumerate(tqdm(docs, total=len(docs))):
        tokens = [token.text for token in doc]
        if not tokens:
            continue
        if doc.spans:
            for labelling_function, span_list in doc.spans.items():
                entities = [
                    (ent.label_, ent.start_char, ent.end_char)
                    for ent in span_list
                ]
                if entities:
                    records.append(
                        rb.TokenClassificationRecord(
                            text=texts[idx],
                            tokens=tokens,
                            prediction=entities,
                            prediction_agent=labelling_function,
                            metadata={
                                "doc_index": idx
                            }
                        )
                    )
    if records:
        rb.log(records=records, name=rubrix_dataset)
```

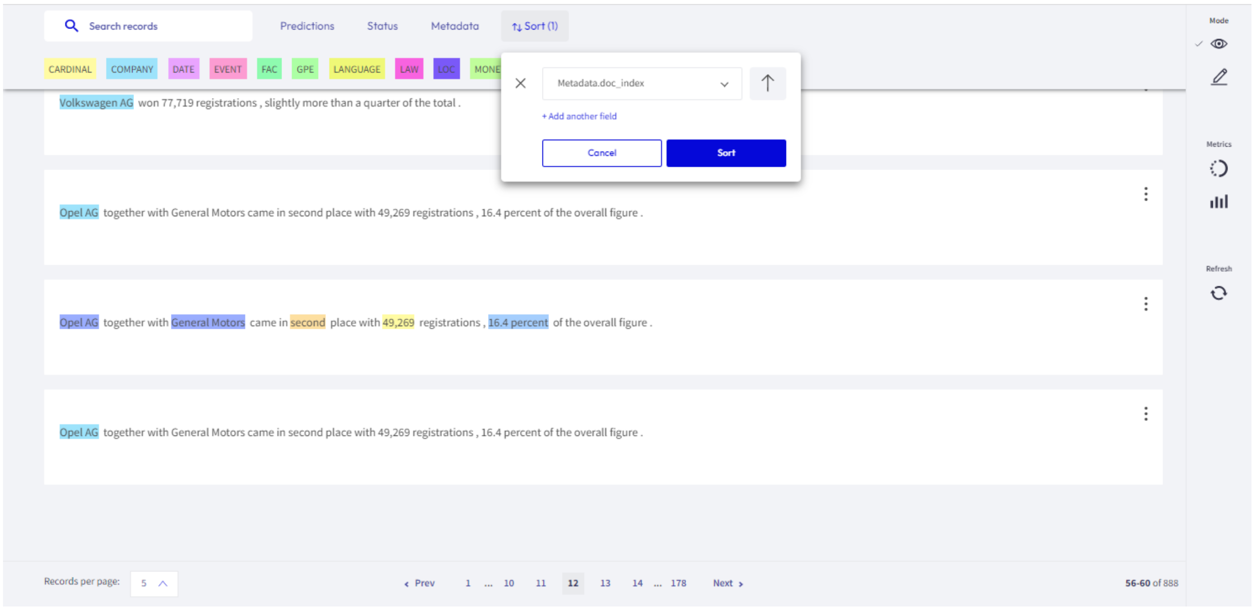
We choose to log the first 1000 documents of our dataset.

```
[ ]: def log_to_rubrix(tokens, docs, limit=None, rubrix_dataset="my_dataset_name"):
    texts = [ " ".join(x) for x in tokens ]
    text_sample = []
    doc_sample = []
    for idx, doc in enumerate(docs):
        text_sample.append(texts[idx])
        doc_sample.append(doc)
        if limit and idx >= limit:
            break
    doc_logger(
        text_sample,
        doc_sample,
        rubrix_dataset=rubrix_dataset
    )

log_to_rubrix(dataset["train"]["tokens"], train_docs, limit=1000, rubrix_dataset="conll_
↪2003_hmm")
```

(continues on next page)

(continued from previous page)

If we sort the Token Classification records on the `conll_2003_hmm` dataset according to the `Metadata.doc_index` field, we will be able to see the predictions produced for each record by our NER models, labelling functions and aggregation models.



Volkswagen AG won 77,719 registrations , slightly more than a quarter of the total .

Opel AG together with General Motors came in second place with 49,269 registrations , 16.4 percent of the overall figure .

Opel AG together with General Motors came in second place with 49,269 registrations , 16.4 percent of the overall figure .

Opel AG together with General Motors came in second place with 49,269 registrations , 16.4 percent of the overall figure .

## 5.18.4 Training a spaCy model

### Preprocessing

Before we train our own model, we need to make sure that our training and development sets are using exactly the same tags.

We map the tags on our training set to the standard tags in the CoNLL 2003 dataset through a replacement dictionary.

[ ]:
```
def get_doc_labels(docs):
    label_set = set()
    for doc in docs:
        for annotator, spans in doc.spans.items():
            for span in spans:
                label_set.update([span.label_])
        for entity in doc.ents:
            label_set.update([entity.label_])
    return label_set

get_doc_labels(train_docs)
```

[18]:
```
replacement_dict = {'CARDINAL': 'MISC',
 'COMPANY': 'ORG',
 'DATE': 'MISC',
 'EVENT': 'MISC',
 'FAC': 'MISC',
```

(continues on next page)

(continued from previous page)

```
 'GPE': 'LOC',
 'LANGUAGE': 'MISC',
 'LAW': 'MISC',
 'LOC': 'LOC',
 'MONEY': 'MISC',
 'NORP': 'MISC',
 'ORDINAL': 'MISC',
 'ORG': 'ORG',
 'OTHER_ORG': 'ORG',
 'PERCENT': 'MISC',
 'PERSON': 'PER',
 'PRODUCT': 'MISC',
 'QUANTITY': 'MISC',
 'TIME': 'MISC',
 'WORK_OF_ART': 'MISC'}
```

[19]:
```
from spacy.tokens import Span

def annotation_standardiser(doc, replacement_dict):
    for source in doc.spans:
        new_spans = []
        for span in doc.spans[source]:
            new_label = replacement_dict.get(span.label_, None)
            if "\n" in span.text:
                continue
            elif new_label:
                new_spans.append(
                    Span(doc, span.start, span.end, label=new_label)
                )
            else:
                new_spans.append(span)
        doc.spans[source] = new_spans
    return doc

train_docs_standard = []
for doc in train_docs:
    new_doc = annotation_standardiser(doc, replacement_dict=replacement_dict)
    train_docs_standard.append(new_doc)
```

[20]:
```
assert not get_doc_labels(train_docs).symmetric_difference(get_doc_labels(dev_docs))
```

After matching the tags, we can train our own NER model.

We choose to use the labels produced by the aggregation model as our first option, and take the labels produced by the spaCy model trained on OntoNotes 5.0 as a fallback for instances in which the HMM model failed to aggregate any tags.

[20]:
```
for doc in train_docs_standard:
    spacy_ents = doc.spans.get("spacy", ())
    hmm_ents = doc.spans.get("hmm", ())
    if hmm_ents:
        doc.ents = hmm_ents
```

(continues on next page)

(continued from previous page)

```
    else:
        doc.ents = spacy_ents
```

We use the `docbin_writer` method from the `skweak` library to save our documents for training and evaluation.

[32]:
```
from skweak.utils import docbin_writer

docbin_writer(train_docs_standard, "/tmp/train.spacy")
docbin_writer(dev_docs, "/tmp/dev.spacy")
```

```
Write to /tmp/train.spacy...done
Write to /tmp/dev.spacy...done
```

### Training

As it can be seen below, after 200 steps, or spaCy NER model was able to achieve a score of 21%.

[21]:
```
!spacy init config - --lang en --pipeline ner --optimize accuracy | \
spacy train - \
--training.max_steps 200 \
--paths.train /tmp/train.spacy \
--paths.dev /tmp/dev.spacy \
--initialize.vectors en_core_web_md \
--output /tmp/model
```

```
 Created output directory: /tmp/model
 Saving to output directory: /tmp/model
 Using CPU
 To switch to GPU 0, use the option: --gpu-id 0

=========================== Initializing pipeline ============================
[2022-01-03 10:31:34,283] [INFO] Set up nlp object from config
[2022-01-03 10:31:34,300] [INFO] Pipeline: ['tok2vec', 'ner']
[2022-01-03 10:31:34,305] [INFO] Created vocabulary
[2022-01-03 10:31:35,954] [INFO] Added vectors: en_core_web_md
[2022-01-03 10:31:36,121] [INFO] Finished initializing nlp object
[2022-01-03 10:32:33,750] [INFO] Initialized pipeline components: ['tok2vec', 'ner']
 Initialized pipeline

============================= Training pipeline ==============================
 Pipeline: ['tok2vec', 'ner']
 Initial learn rate: 0.001
E    #       LOSS TOK2VEC  LOSS NER  ENTS_F  ENTS_P  ENTS_R  SCORE
---  ------  ------------  --------  ------  ------  ------  ------
  0       0          0.00     43.00    0.00    0.00    0.00    0.00
  0     200         45.20   3765.66   21.21   23.23   19.52    0.21
 Saved pipeline to output directory
/tmp/model/model-last
```

### 5.18.5 Add Gazetteers

In addition to heuristics, we can also exploit labelling functions made from gazetteers. They search for the occurrences of entries, often extracted from a knowledge base.

#### Wikipedia

The database from Wikipedia is extracted from the NECKar dataset. This gazetteer is limited to wikidata objects containing a text description.

```
[ ]: tries = skweak.gazetteers.extract_json_data("./data/skweak/wikidata_small_tokenised.json.
     ↪gz")
     wikismall_gazetteer_cased = skweak.gazetteers.GazetteerAnnotator("wikismall_cased_
     ↪gazetteer", tries)
     wikismall_gazetter_uncased = skweak.gazetteers.GazetteerAnnotator("wikismall_uncased_
     ↪gazetteer", tries, case_sensitive=False)
```

#### Crunchbase

The Crunchbase gazetteer is extracted from the Open Data Map from Crunchbase, which contains lists of both organisations and (business) persons.

```
[ ]: tries = skweak.gazetteers.extract_json_data("./data/skweak/crunchbase_companies.json.gz")
     crunchbase_gazetteer = skweak.gazetteers.GazetteerAnnotator("crunchbase_gazetteer",␣
     ↪tries)
```

#### Geonames

The geonames database contains a large list of locations, including both geopolitical entities and “natural” locations.

```
[ ]: tries = skweak.gazetteers.extract_json_data("./data/skweak/geonames.json",  spacy_model=
     ↪"en_core_web_sm")
     geonames_gazetteer_cased = skweak.gazetteers.GazetteerAnnotator("geo_cased_gazetteer",␣
     ↪tries)
     geonames_gazetteer_uncased = skweak.gazetteers.GazetteerAnnotator("geo_uncased_gazetteer
     ↪", tries, case_sensitive=False)
```

#### DBPedia

This gazeetter utilizes DBPedia to extract a list of products and brands as products.

```
[ ]:
     tries = skweak.gazetteers.extract_json_data("./data/skweak/products.json",  spacy_model=
     ↪"en_core_web_sm")
     products_gazetteer_cased = skweak.gazetteers.GazetteerAnnotator("products_cased_gazetteer
     ↪", tries)
     products_gazetteer_uncased = skweak.gazetteers.GazetteerAnnotator("products_uncased_
     ↪gazetteer", tries)
```

We combine all gazetteers into a single annotator through the CombinedAnnotator class.

[25]:
```
from skweak.base import CombinedAnnotator

gazetteers = [wikismall_gazetteer_cased, crunchbase_gazetteer, geonames_gazetteer_cased,␣
↪products_gazetteer_cased]

combined_gazetteer = CombinedAnnotator()
for gazetteer in gazetteers:
    combined_gazetteer.add_annotator(gazetteer)

train_docs = list(combined_gazetteer.pipe(train_docs))
```

### Aggregation

Besides using a HMM model, we can also aggregate the annotations of our documents using majority voting.

We map our tags to the CoNLL format, and then apply the MajorityVoter aggregation model.

[26]:
```
train_docs_standard_v2 = []
for doc in train_docs:
    new_doc = annotation_standardiser(doc, replacement_dict=replacement_dict)
    train_docs_standard_v2.append(new_doc)
```

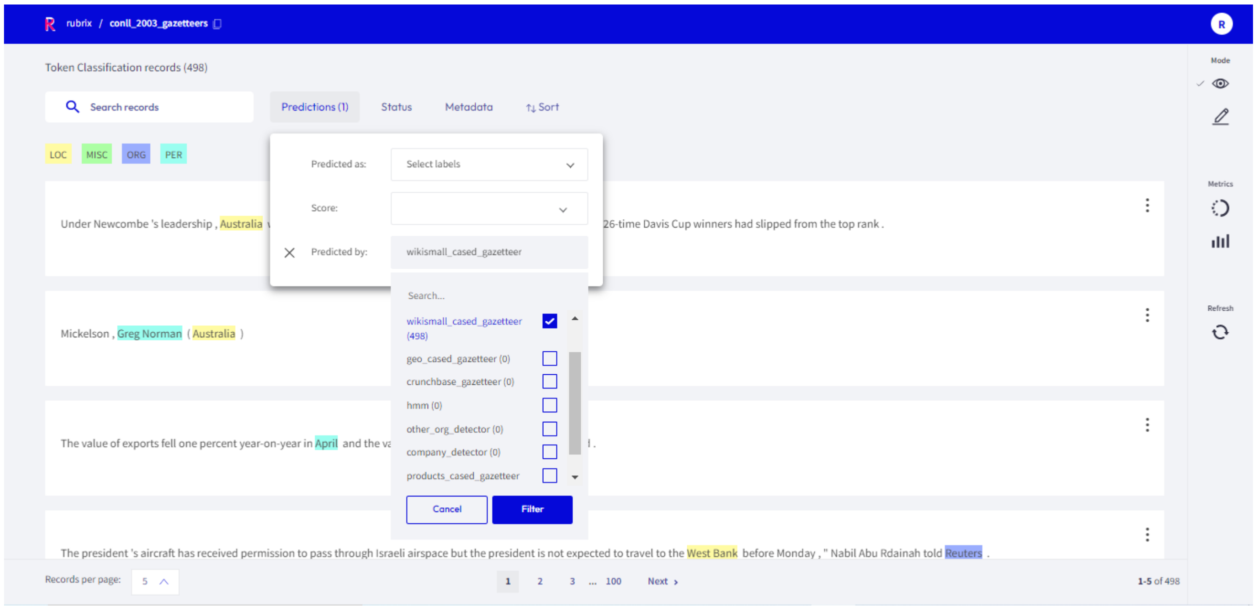
[27]:
```
mv = skweak.aggregation.MajorityVoter("mv", ["LOC", "MISC", "ORG", "PER"])
mv.add_underspecified_label("ENT", {"LOC", "MISC", "ORG", "PER"})
```

[28]:
```
train_docs_standard_v2 = list(mv.pipe(train_docs_standard_v2))
```

### Visualization with Rubrix

We can visualize the annotations produced by our gazetteers with Rubrix.

We are able to notice that, among all our gazetteers, only wikismall_gazetteer_cased was able to capture entities from the training data.



Token Classification records (498)

Predicted as: Select labels

Score:

Predicted by: wikismall_cased_gazetteer

wikismall_cased_gazetteer (498) ✓
geo_cased_gazetteer (0)
crunchbase_gazetteer (0)
hmm (0)
other_org_detector (0)
company_detector (0)
products_cased_gazetteer

Under Newcombe 's leadership , Australia 26-time Davis Cup winners had slipped from the top rank .

Mickelson , Greg Norman ( Australia )

The value of exports fell one percent year-on-year in April and the va

The president 's aircraft has received permission to pass through Israeli airspace but the president is not expected to travel to the West Bank before Monday , " Nabil Abu Rdainah told Reuters .

1-5 of 498

```
[ ]: log_to_rubrix(dataset["train"]["tokens"], train_docs_standard_v2, limit=1000, rubrix_
     ↪dataset="conll_2003_gazetteers")
```

### Training a spaCy model

We choose to use the labels produced by the aggregation model as our first option, and take the labels produced by the spaCy model trained on OntoNotes 5.0 as a fallback for instances in which the `MajorityVoter` failed to aggregate any tags.

```
[30]: for doc in train_docs_standard_v2:
          spacy_ents = doc.spans.get("spacy", ())
          mv_ents = doc.spans.get("mv", ())
          if mv_ents:
              doc.ents = mv_ents
          else:
              doc.ents = spacy_ents
```

We use the `docbin_writer` method from the `skweak` library to save our documents for training.

Our development set has already been saved as `dev.spacy` in our previous training iteration.

```
[ ]: docbin_writer(train_docs_standard_v2, "/tmp/train_v2.spacy")
```

As it can be seen below, after adding gazetteers and using a majority voter as our aggregation model, or trained NER model was able to achieve a F1-score of 22%, which is a 1% improvement over our previous result.

For the sake of brevity, we did not present all labelling functions in the `skweak` library in this tutorial. We should ideally stack several labelling functions and loop through annotation and training until we arrive at our desired results. Please refer to the Step-by-step NER tutorial and the official skweak documentation for a full overview of what is possible to achieve with the library.

```
[34]: !spacy init config - --lang en --pipeline ner --optimize accuracy | \
      spacy train - \
      --training.max_steps 200 \
```

(continues on next page)

(continued from previous page)

```
--paths.train /tmp/train_v2.spacy \
--paths.dev /tmp/dev.spacy \
--initialize.vectors en_core_web_md \
--output /tmp/model
```

```
 Saving to output directory: /tmp/model
 Using CPU

=========================== Initializing pipeline ===========================
[2022-01-03 16:06:22,664] [INFO] Set up nlp object from config
[2022-01-03 16:06:22,674] [INFO] Pipeline: ['tok2vec', 'ner']
[2022-01-03 16:06:22,678] [INFO] Created vocabulary
[2022-01-03 16:06:23,774] [INFO] Added vectors: en_core_web_md
[2022-01-03 16:06:24,127] [INFO] Finished initializing nlp object
[2022-01-03 16:07:29,953] [INFO] Initialized pipeline components: ['tok2vec', 'ner']
 Initialized pipeline

============================= Training pipeline =============================
 Pipeline: ['tok2vec', 'ner']
 Initial learn rate: 0.001
E    #       LOSS TOK2VEC  LOSS NER  ENTS_F  ENTS_P  ENTS_R  SCORE
---  ------  ------------  --------  ------  ------  ------  ------
  0       0          0.00     43.00    0.00    0.00    0.00    0.00
  0     200         46.47   4005.87   22.49   23.64   21.45    0.22
 Saved pipeline to output directory
/tmp/model/model-last
```

### 5.18.6 Simpletransformers

Rather than training our NER models in spaCy, we can also fine-tune pre-trained transformers to our annotations produced with `skweak`.

Here we use simpletransformers, a library built on top of the transformers library.

#### Preprocessing

First we have to convert our spaCy `Doc` objects into dataframes that can be utilized with the `simpletransformers` library.

[48]:
```python
import pandas as pd

def get_training_data_from_docs(docs):
    training_df = []
    for doc_idx, doc in enumerate(docs):
        tokens = [ x.text for x in doc ]
        label_array = ['O'] * len(tokens)
        for ent in doc.ents:
            for token_idx, token in enumerate(tokens):
                if token_idx >= ent.start and token_idx + 1 <= ent.end:
                    label_array[token_idx] = ent.label_
        training_df_rows = [ [doc_idx, tokens[idx], label_array[idx]] for idx in␣
↪range(len(tokens)) ]
```

(continues on next page)

(continued from previous page)

```
        training_df.extend(training_df_rows)
    training_df = pd.DataFrame(training_df, columns=["sentence_id", "words", "labels"])
    return training_df

def get_evaluation_data_from_dataset(dataset, tokens_field="tokens", tags_field="ner_tags
↪"):
    tokens = dataset[tokens_field]
    tags = dataset[tags_field]
    index_array = []
    for idx, item in enumerate(tokens):
        index_array.append([idx] * len(item))

    test_df_sents = [ list(zip(index_array[idx], tokens[idx], tags[idx]))
                         for idx, item in enumerate(tokens) ]
    eval_df = []
    for sentence in test_df_sents:
        eval_df.extend(sentence)

    eval_df = pd.DataFrame(eval_df, columns=["sentence_id", "words", "labels"])
    return eval_df


train_df = get_training_data_from_docs(train_docs_standard_v2)
eval_df = get_evaluation_data_from_dataset(dataset['validation'])
```

### Training

Here we fine-tune a distilbert model according to the instructions in the simpletransformers documentation.

[ ]:
```
# Configure the model
from simpletransformers.ner import NERModel, NERArgs

model_args = NERArgs()
model_args.train_batch_size = 16
model_args.evaluate_during_training = True

custom_labels = [ "O", "PER", "ORG", "LOC", "MISC" ]

model = NERModel(
    "distilbert", "distilbert-base-cased", args=model_args, use_cuda=True, labels=custom_
↪labels
)

# Train the model
model.train_model(train_df, eval_data=eval_df)

# Evaluate the model
result, model_outputs, preds_list = model.eval_model(eval_df)
```

After fine-tuning a `distilbert` model, we can see that we were able to raise our F1-score to 52%.

[53]:
```
# Print the results
result
```

[53]:
```
{'eval_loss': 0.677833871929666,
 'f1_score': 0.5242952373303349,
 'precision': 0.4152671755725191,
 'recall': 0.7109562186887388}
```

## 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: str*, *name_of_copy: str*, *workspace: Optional[str] = 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: str*) → None

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()` → str

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: Optional[str] = None*, *api_key: Optional[str] = None*, *workspace: Optional[str] = None*, *timeout: int = 60*) → None

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: str*, *query: Optional[str] = None*, *ids: Optional[List[Union[str, int]]] = None*, *limit: Optional[int] = None*, *as_pandas: bool = True*) → Union[pandas.core.frame.DataFrame, List[Union[*rubrix.client.models.TextClassificationRecord*, *rubrix.client.models.TokenClassificationRecord*, *rubrix.client.models.Text2TextRecord*]]]

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: 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*]]]*, *name: str*, *tags: Optional[Dict[str, str]] = None*, *metadata: Optional[Dict[str, Any]] = None*, *chunk_size: int = 500*, *verbose: bool = True*) → rubrix.client.models.BulkResponse

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: str*) → None

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.`**`Text2TextRecord`**(**, text: str, prediction: List[Union[str, Tuple[str, float]]] = None, prediction_agent: str = None, annotation: str = None, annotation_agent: str = None, id: Optional[Union[int, str]] = None, metadata: Dict[str, Any] = None, status: str = None, event_timestamp: datetime.datetime = None, metrics: Dict[str, Any] = None*)

Record for a text to text task

**Parameters**

- **`text`** (*`str`*) – The input of the record
- **`prediction`** (*`Optional[List[Union[str, Tuple[str, float]]]]`*) – A list of strings or tuples containing predictions for the input text. If tuples, the first entry is the predicted text, the second entry is its corresponding score.
- **`prediction_agent`** (*`Optional[str]`*) – Name of the prediction agent. By default, this is set to the hostname of your machine.
- **`annotation`** (*`Optional[str]`*) – A string representing the expected output text for the given input text.
- **`annotation_agent`** (*`Optional[str]`*) – Name of the prediction agent. By default, this is set to the hostname of your machine.
- **`id`** (*`Optional[Union[int, str]]`*) – The id of the record. By default (None), we will generate a unique ID for you.
- **`metadata`** (*`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: Optional[List[Union[str, Tuple[str, float]]]]*)

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`**(*, *inputs: Union[str, List[str], Dict[str, Union[str, List[str]]]]*, *prediction: List[Tuple[str, float]] = None*, *prediction_agent: str = None*, *annotation: Optional[Union[str, List[str]]] = None*, *annotation_agent: str = None*, *multi_label: bool = False*, *explanation: Dict[str, List[rubrix.client.models.TokenAttributions]] = None*, *id: Optional[Union[int, str]] = None*, *metadata: Dict[str, Any] = None*, *status: str = None*, *event_timestamp: datetime.datetime = None*, *metrics: Dict[str, Any] = 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.
- **prediction_agent** (*Optional[str]*) – Name of the prediction agent. By default, this is set to the hostname of your machine.
- **annotation** (*Optional[Union[str, List[str]]]*) – A string or a list of strings (multilabel) corresponding to the annotation (gold label) for the record.
- **annotation_agent** (*Optional[str]*) – Name of the prediction agent. By default, this is set to the hostname of your machine.
- **multi_label** (*bool*) – Is the prediction/annotation for a multi label classification task? Defaults to *False*.
- **explanation** (*Optional[Dict[str, List[rubrix.client.models.TokenAttributions]]]*) – A dictionary containing the attributions of each token to the prediction. The keys map the input of the record (see *inputs*) to the *TokenAttributions*.
- **id** (*Optional[Union[int, str]]*) – The id of the record. By default (*None*), we will generate a unique ID for you.
- **metadata** (*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: str, attributions: Dict[str, float] = None*)
Attribution of the token to the predicted label.

In the Rubrix app this is only supported for `TextClassificationRecord` and the `multi_label=False` case.

**Parameters**

- **token** (*str*) – The input token.
- **attributions** (*Dict[str, float]*) – A dictionary containing label-attribution pairs.

**Return type** None

**class** rubrix.client.models.**TokenClassificationRecord**(**, text: str, tokens: List[str], prediction: List[Union[Tuple[str, int, int], Tuple[str, int, int, float]]] = None, prediction_agent: str = None, annotation: List[Tuple[str, int, int]] = None, annotation_agent: str = None, id: Optional[Union[int, str]] = None, metadata: Dict[str, Any] = None, status: str = None, event_timestamp: datetime.datetime = None, metrics: Dict[str, Any] = 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.
- **prediction_agent** (*Optional[str]*) – Name of the prediction agent. By default, this is set to the hostname of your machine.
- **annotation** (*Optional[List[Tuple[str, int, int]]]*) – A list of tuples containing annotations (gold labels) for the record. The first entry of the tuple is the name of the entity, the second and third entry correspond to the start and stop char index of the entity.
- **annotation_agent** (*Optional[str]*) – Name of the prediction agent. By default, this is set to the hostname of your machine.
- **id** (*Optional[Union[int, str]]*) – The id of the record. By default (None), we will generate a unique ID for you.
- **metadata** (*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: str*, *query: Optional[str] = None*) → rubrix.metrics.models.MetricSummary

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/webapp/search_records.html)

**Returns** The f1 metric summary

**Return type** rubrix.metrics.models.MetricSummary

### Examples

```
>>> from rubrix.metrics.text_classification import f1
>>> summary = f1(name="example-dataset")
>>> summary.visualize() # will plot a bar chart with results
>>> summary.data # returns the raw result data
```

`rubrix.metrics.text_classification.metrics.`**f1_multilabel**(*name: str*, *query: Optional[str] = None*) → rubrix.metrics.models.MetricSummary

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/webapp/search_records.html)

**Returns** The f1 metric summary

**Return type** rubrix.metrics.models.MetricSummary

### Examples

```
>>> from rubrix.metrics.text_classification import f1_multilabel
>>> summary = f1_multilabel(name="example-dataset")
>>> summary.visualize() # will plot a bar chart with results
>>> summary.data # returns the raw result data
```

### Token classification

class rubrix.metrics.token_classification.metrics.**ComputeFor**(*value*)

An enumeration.

rubrix.metrics.token_classification.metrics.**entity_capitalness**(*name: str, query: Optional[str] = None, compute_for: Union[str, rubrix.metrics.token_classification.metrics.Compute = ComputeFor.PREDICTIONS*) → rubrix.metrics.models.MetricSummary

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: str*, *query: Optional[str] = None*, *compute_for: Union[str, rubrix.metrics.token_classification.metrics.Compute* *= ComputeFor.PREDICTIONS*, *mentions: int = 10*, *threshold: int = 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: str*, *query: Optional[str] = None*, *compute_for: Union[str, rubrix.metrics.token_classification.metrics.ComputeFor] = ComputeFor.PREDICTIONS*, *interval: float = 0.005*) → rubrix.metrics.models.MetricSummary

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: str*, *query: Optional[str] = None*, *compute_for: Union[str, rubrix.metrics.token_classification.metrics.ComputeFor] = ComputeFor.PREDICTIONS*, *labels: int = 50*) → rubrix.metrics.models.MetricSummary

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=20)
>>> summary.visualize() # will plot a bar chart with results
>>> summary.data # The top-20 entity tags
```

`rubrix.metrics.token_classification.metrics.`**`f1`**(*name: str*, *query: Optional[str] = None*) → rubrix.metrics.models.MetricSummary

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: str*, *query: Optional[str] = None*, *level: str = 'token'*, *compute_for: Union[str, rubrix.metrics.token_classification.metrics.ComputeFor] = ComputeFor.PREDICTIONS*, *interval: int = 1*) → rubrix.metrics.models.MetricSummary

Computes mentions length distribution (in number of tokens).

**Parameters**

- **name** (*str*) – The dataset name.
- **query** (*Optional[str]*) – An ElasticSearch query with the query string syntax
- **level** (*str*) – The mention length level. Accepted values are "token" and "char"
- **compute_for** (*Union[str, rubrix.metrics.token_classification.metrics.ComputeFor]*) – Metric can be computed for annotations or predictions. Accepted values are `Annotations` and `Predictions`. Defaults to `Predictions`.
- **interval** (*int*) – The bins or bucket for result histogram

**Returns** The summary for mention token distribution

**Return type** rubrix.metrics.models.MetricSummary

### Examples

```
>>> from rubrix.metrics.token_classification import mention_length
>>> summary = mention_length(name="example-dataset", interval=2)
>>> summary.visualize() # will plot a histogram chart with results
>>> summary.data # the raw histogram data with bins of size 2
```

`rubrix.metrics.token_classification.metrics.`**`token_capitalness`**(*name: str*, *query: Optional[str] = None*) → rubrix.metrics.models.MetricSummary

Computes the token capitalness distribution

**Parameters**

- **name** (*str*) – The dataset name.
- **query** (*Optional[str]*) – An ElasticSearch query with the query string syntax

**Returns** The summary for token length distribution

**Return type** rubrix.metrics.models.MetricSummary

### Examples

```
>>> from rubrix.metrics.token_classification import token_capitalness
>>> summary = token_capitalness(name="example-dataset")
>>> summary.visualize() # will plot a histogram with results
>>> summary.data # The token capitalness distribution
```

`rubrix.metrics.token_classification.metrics.`**`token_frequency`**(*name: str*, *query: Optional[str] = None*, *tokens: int = 1000*) → rubrix.metrics.models.MetricSummary

Computes the token frequency distribution for a numbe of tokens.

**Parameters**

- **`name`** (*`str`*) – The dataset name.
- **`query`** (*`Optional[str]`*) – An ElasticSearch query with the query string syntax
- **`tokens`** (*`int`*) – The top-k number of tokens to retrieve

**Returns** The summary for token frequency distribution

**Return type** rubrix.metrics.models.MetricSummary

### Examples

```
>>> from rubrix.metrics.token_classification import token_frequency
>>> summary = token_frequency(name="example-dataset", token=50)
>>> summary.visualize() # will plot a histogram with results
>>> summary.data # the top-50 tokens frequency
```

`rubrix.metrics.token_classification.metrics.`**`token_length`**(*name: str*, *query: Optional[str] = None*) → rubrix.metrics.models.MetricSummary

Computes the token size distribution in terms of number of characters

**Parameters**

- **`name`** (*`str`*) – The dataset name.
- **`query`** (*`Optional[str]`*) – An ElasticSearch query with the query string syntax

**Returns** The summary for token length distribution

**Return type** rubrix.metrics.models.MetricSummary

### Examples

```
>>> from rubrix.metrics.token_classification import token_length
>>> summary = token_length(name="example-dataset")
>>> summary.visualize() # will plot a histogram with results
>>> summary.data # The token length distribution
```

`rubrix.metrics.token_classification.metrics.`**`tokens_length`**(*name: str*, *query: Optional[str] = None*, *interval: int = 1*) → rubrix.metrics.models.MetricSummary

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: str*, *label: str*, *name: Optional[str] = None*, *author: Optional[str] = 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.
- **author** (*Optional[str]*) –

### Examples

```
>>> import rubrix as rb
>>> urgent_rule = Rule(query="inputs.text:(urgent AND immediately)", label="urgent",
↪ name="urgent_rule")
>>> not_urgent_rule = Rule(query="inputs.text:(NOT urgent) AND metadata.title_
↪length>20", label="not urgent")
>>> not_urgent_rule.apply("my_dataset")
>>> my_dataset_records = rb.load(name="my_dataset", as_pandas=False)
>>> not_urgent_rule(my_dataset_records[0])
"not urgent"
```

**`__call__`**(*record:* rubrix.client.models.TextClassificationRecord) → Optional[str]

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: str*)

Apply the rule to a dataset and save matching ids of the records.

**Parameters** **`dataset`** (*`str`*) – The name of the dataset.

**`property author`**

Who authored the rule.

**`property label: str`**

The rule label

**`metrics`**(*dataset: str*) → Dict[str, Union[int, float]]

Compute the rule metrics for a given dataset:

- **coverage**: Fraction of the records labeled by the rule.
- **annotated_coverage**: Fraction of annotated records labeled by the rule.
- **correct**: Number of records the rule labeled correctly (if annotations are available).
- **incorrect**: Number of records the rule labeled incorrectly (if annotations are available).
- **precision**: Fraction of correct labels given by the rule (if annotations are available). The precision does not penalize the rule for abstains.

**Parameters** **`dataset`** (*`str`*) – Name of the dataset for which to compute the rule metrics.

**Returns** The rule metrics.

**Return type** Dict[str, Union[int, float]]

**`property name`**

The name of the rule.

**`property query: str`**

The rule query

`rubrix.labeling.text_classification.rule.`**`load_rules`**`(`*dataset: str*`)` → List[*rubrix.labeling.text_classification.rule.Rule*]

load the rules defined in a given dataset.

**Parameters** **`dataset`** (*`str`*) – Name of the dataset.

**Returns** A list of rules defined in the given dataset.

**Return type** List[*rubrix.labeling.text_classification.rule.Rule*]

**`exception`** `rubrix.labeling.text_classification.weak_labels.`**`NoRulesFoundError`**

**`class`** `rubrix.labeling.text_classification.weak_labels.`**`WeakLabels`**`(`*dataset: str*, *rules: Optional[List[Callable]] = None*, *ids: Optional[List[Union[str, int]]] = None*, *query: Optional[str] = None*, *label2int: Optional[Dict[Optional[str], int]] = None*`)`

Computes the weak labels of a dataset by applying a given list of rules.

**Parameters**

- **`dataset`** (*`str`*) – Name of the dataset to which the rules will be applied.
- **`rules`** (*`Optional[List[Callable]]`*) – A list of rules (labeling functions). They must return a string, or `None` in case of abstention. If None, we will use the rules of the dataset (Default).
- **`ids`** (*`Optional[List[Union[int, str]]]`*) – An optional list of record ids to filter the dataset before applying the rules.
- **`query`** (*`Optional[str]`*) – An optional ElasticSearch query with the query string syntax to filter the dataset before applying the rules.
- **`label2int`** (*`Optional[Dict[Optional[str], int]]`*) – An optional dict, mapping the labels to integers. Remember that the return type `None` means abstention (e.g. `{None: -1}`). By default, we will build a mapping on the fly when applying the rules.

**Raises**

- ***`NoRulesFoundError`*** – When you do not provide rules, and the dataset has no rules either.
- **`DuplicatedRuleNameError`** – When you provided multiple rules with the same name.
- **`NoRecordsFoundError`** – When the filtered dataset is empty.
- **`MultiLabelError`** – When trying to get weak labels for a multi-label text classification task.
- **`MissingLabelError`** – When provided with a `label2int` dict, and a weak label or annotation label is not present in its keys.

### Examples

Get the weak label matrix from a dataset with rules:

```
>>> weak_labels = WeakLabels(dataset="my_dataset")
>>> weak_labels.matrix()
>>> weak_labels.summary()
```

Get the weak label matrix from rules defined in Python:

```
>>> def awesome_rule(record: TextClassificationRecord) -> str:
...     return "Positive" if "awesome" in record.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 the WeakLabels object with 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))
```

For a builtin integration with Snorkel, see *rubrix.labeling.text_classification.Snorkel*.

`annotation`(*include_missing: bool = False*, *exclude_missing_annotations: Optional[bool] = None*) → numpy.ndarray

Returns the annotation labels as an array of integers.

**Parameters**

- **`include_missing`** (`bool`) – If True, returns an array of the length of the record list (`self.records()`). For this we will fill the array with the `self.label2int[None]` integer for records without an annotation.
- **`exclude_missing_annotations`** (`Optional[bool]`) – DEPRECATED

**Returns** The annotation array of integers.

**Return type** numpy.ndarray

`change_mapping`(*label2int: Dict[str, int]*)

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: Optional[bool] = None*) → numpy.ndarray

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: Optional[bool] = None*) → List[*rubrix.client.models.TextClassificationRecord*]

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: Optional[List[str]] = None*, *rules: Optional[List[Union[str, int]]] = None*) → pandas.core.frame.DataFrame

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: bool = False*, *annotation: Optional[numpy.ndarray] = None*) → pandas.core.frame.DataFrame

Returns following summary statistics for each rule:

- **label**: Set of unique labels returned by the rule, excluding "None" (abstain).
- **coverage**: Fraction of the records labeled by the rule.
- **annotated_coverage**: Fraction of annotated records labeled by the rule (if annotations are available).
- **overlaps**: Fraction of the records labeled by the rule together with at least one other rule.
- **conflicts**: Fraction of the records where the rule disagrees with at least one other rule.
- **correct**: Number of records the rule labeled correctly (if annotations are available).
- **incorrect**: Number of records the rule labeled incorrectly (if annotations are available).
- **precision**: Fraction of correct labels given by the rule (if annotations are available). The precision does not penalize the rule for abstains.

**Parameters**

- **`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.**FlyingSquid**(*weak_labels: rubrix.labeling.text_classification.weak_labels.*
***kwargs*)

The label model by FlyingSquid.

**Parameters**

- **weak_labels** (`rubrix.labeling.text_classification.weak_labels.WeakLabels`) – A *WeakLabels* object containing the weak labels and records.
- ****kwargs** – Passed on to the init of the FlyingSquid's LabelModel.

## Examples

```
>>> from rubrix.labeling.text_classification import WeakLabels
>>> weak_labels = WeakLabels(dataset="my_dataset")
>>> label_model = FlyingSquid(weak_labels)
>>> label_model.fit()
>>> records = label_model.predict()
```

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

**Parameters**

- **include_annotated_records** (*bool*) – Whether or not to include annotated records in the training.
- ****kwargs** – Passed on to the FlyingSquid's LabelModel.fit() method.

**predict**(*include_annotated_records: bool = False*, *include_abstentions: bool = False*, *prediction_agent: str = 'FlyingSquid'*, *verbose: bool = True*, *tie_break_policy: str = 'abstain'*) → List[*rubrix.client.models.TextClassificationRecord*]
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.
- **prediction_agent** (*str*) – String used for the `prediction_agent` in the returned records.
- **verbose** (*bool*) – If True, print out messages of the progress to stderr.
- **tie_break_policy** (*str*) – Policy to break ties. You can choose among two policies:
  - *abstain*: Do not provide any prediction
  - *random*: randomly choose among tied option using deterministic hash

The last policy can introduce quite a bit of noise, especially when the tie is among many labels, as is the case when all of the labeling functions abstained.

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

**Raises** `NotFittedError` – If the label model was still not fitted.

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

`score`(*tie_break_policy: Union[rubrix.labeling.text_classification.label_models.TieBreakPolicy, str] = 'abstain'*, *verbose: bool = False*) → Dict[str, float]

Returns some scores/metrics of the label model with respect to the annotated records.

The metrics are:

- accuracy
- micro/macro averages for precision, recall and f1
- precision, recall, f1 and support for each label

For more details about the metrics, check out the sklearn docs.

**Parameters**

- **tie_break_policy** (*Union[rubrix.labeling.text_classification.label_models.TieBreakPolicy, str]*) – Policy to break ties. You can choose among two policies:
  - *abstain*: Do not provide any prediction
  - *random*: randomly choose among tied option using deterministic hash

  The last policy can introduce quite a bit of noise, especially when the tie is among many labels, as is the case when all of the labeling functions abstained.
- **verbose** (*bool*) – If True, print out messages of the progress to stderr.

**Returns** The scores/metrics as a dictionary.

**Return type** Dict[str, float]

**Note:** Metrics are only calculated over non-abstained predictions!

**Raises**

- **NotFittedError** – If the label model was still not fitted.
- **MissingAnnotationError** – If the `weak_labels` do not contain annotated records.

**Parameters**

- **tie_break_policy** (*Union[rubrix.labeling.text_classification.label_models.TieBreakPolicy, str]*) –
- **verbose** (*bool*) –

**Return type** Dict[str, float]

`class rubrix.labeling.text_classification.label_models.LabelModel`(*weak_labels: rubrix.labeling.text_classification.weak_labels.W*

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: bool = 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: bool = False*, *include_abstentions: bool = False*, *prediction_agent: str = 'LabelModel'*, ***kwargs*) → List[*rubrix.client.models.TextClassificationRecord*]
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.
- **prediction_agent** (*str*) – String used for the `prediction_agent` in the returned records.

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

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

**score**(**args*, ***kwargs*) → Dict
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: rubrix.labeling.text_classification.weak_labels.Weak*
*verbose: bool = True*, *device: str = '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 WeakLabels
>>> weak_labels = WeakLabels(dataset="my_dataset")
>>> label_model = Snorkel(weak_labels)
>>> label_model.fit()
>>> records = label_model.predict()
```

**fit**(*include_annotated_records: bool = 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: bool = False*, *include_abstentions: bool = False*, *prediction_agent: str = 'Snorkel'*, *tie_break_policy: Union[rubrix.labeling.text_classification.label_models.TieBreakPolicy, str] = 'abstain'*) → List[*rubrix.client.models.TextClassificationRecord*]

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.
- **prediction_agent** (*str*) – String used for the `prediction_agent` in the returned records.
- **tie_break_policy** (*Union[rubrix.labeling.text_classification.label_models.TieBreakPolicy, str]*) – Policy to break ties. You can choose among three policies:
  - *abstain*: Do not provide any prediction
  - *random*: randomly choose among tied option using deterministic hash
  - *true-random*: randomly choose among the tied options. NOTE: repeated runs may have slightly different results due to differences in broken ties

  The last two policies can introduce quite a bit of noise, especially when the tie is among many labels, as is the case when all 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: Union[rubrix.labeling.text_classification.label_models.TieBreakPolicy, str] = 'abstain'*) → Dict[str, float]

Returns some scores/metrics of the label model with respect to the annotated records.

The metrics are:

- accuracy
- micro/macro averages for precision, recall and f1
- precision, recall, f1 and support for each label

For more details about the metrics, check out the sklearn docs.

**Parameters** **tie_break_policy** (*Union[rubrix.labeling.text_classification.label_models.TieBreakPolicy, str]*) – Policy to break ties. You can choose among three policies:

- *abstain*: Do not provide any prediction
- *random*: randomly choose among tied option using deterministic hash
- *true-random*: randomly choose among the tied options. NOTE: repeated runs may have slightly different results due to differences in broken ties

The last two policies can introduce quite a bit of noise, especially when the tie is among many labels, as is the case when all of the labeling functions abstained.

**Returns** The scores/metrics as a dictionary.

**Return type** Dict[str, float]

**Note:** Metrics are only calculated over non-abstained predictions!

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

**Parameters** **`tie_break_policy`** (*`Union[rubrix.labeling.text_classification.label_models.TieBreakPolicy, str]`*) –

**Return type** Dict[str, float]

`rubrix.labeling.text_classification.label_errors.`**`find_label_errors`**(*records: List[*rubrix.client.models.TextClassificationRec*sort_by: Union[str, rubrix.labeling.text_classification.label_errors.*= 'likelihood', metadata_key: str = 'label_error_candidate', n_jobs: Optional[int] = 1, **kwargs*) → List[*rubrix.client.models.TextClassificationRec*

Finds potential annotation/label errors in your records using [cleanlab](https://github.com/cleanlab/cleanlab).

We will consider all records for which a prediction AND annotation is available. Make sure the predictions were made in a holdout manner, that is you should only include records that were not used in the training of the predictor.

**Parameters**

- **`records`** (*`List[rubrix.client.models.TextClassificationRecord]`*) – A list of text classification records
- **`sort_by`** (*`Union[str, rubrix.labeling.text_classification.label_errors.SortBy]`*) – One of the three options - "likelihood": sort the returned records by likelihood of containing a label error (most likely first) - "prediction": sort the returned records by the probability of the prediction (highest probability first) - "none": do not sort the returned records
- **`metadata_key`** (*`str`*) – The key added to the record's metadata that holds the order, if `sort_by` is not "none".
- **`n_jobs`** (*`Optional[int]`*) – Number of processing threads used by multiprocessing. If None, uses the number of threads on your CPU. Defaults to 1, which removes parallel processing.
- **`**kwargs`** – Passed on to *cleanlab.pruning.get_noise_indices*

**Returns** A list of records containing potential annotation/label errors

**Raises**

- **`NoRecordsError`** – If none of the records has a prediction AND annotation.
- **`MissingPredictionError`** – If a prediction is missing for one of the labels.

- **ValueError** – If not supported kwargs are passed on, e.g. 'sorted_index_method'.

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

### Examples

```
>>> import rubrix as rb
>>> records = rb.load("my_dataset", as_pandas=False)
>>> records_with_label_errors = find_label_errors(records)
```

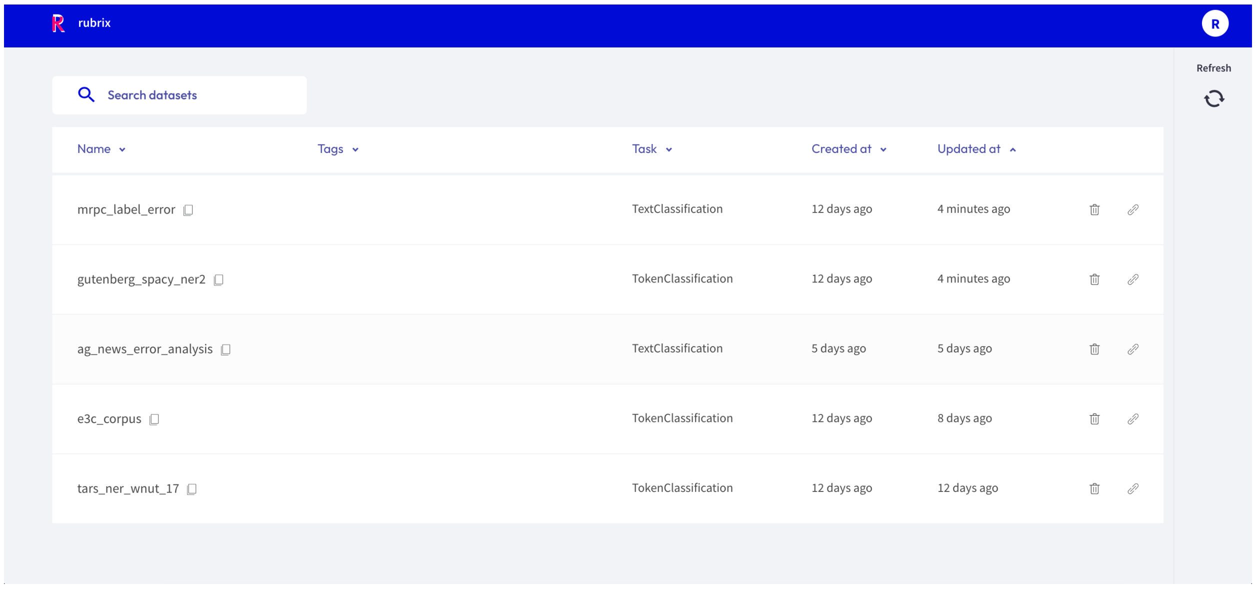
## 5.20 Web App UI

Here we provide a comprehensible overview of the Rubrix web app's User Interface (UI). To launch the web app, please have a look at our *setup and installation guide*.

### 5.20.1 Pages

- ***Workspace***: Search, access and share your datasets in a unified place
- ***Dataset***: Dive into your dataset to explore and annotate your records

#### Workspace



| Name | Tags | Task | Created at | Updated at |
|---|---|---|---|---|
| mrpc_label_error | | TextClassification | 12 days ago | 4 minutes ago |
| gutenberg_spacy_ner2 | | TokenClassification | 12 days ago | 4 minutes ago |
| ag_news_error_analysis | | TextClassification | 5 days ago | 5 days ago |
| e3c_corpus | | TokenClassification | 12 days ago | 8 days ago |
| tars_ner_wnut_17 | | TokenClassification | 12 days ago | 12 days ago |

The *Workspace* page is mainly a **searchable and sortable list** of **datasets**. It is the **entry point** to the Rubrix web app and is composed of the following 3 components.

### Search bar

The "*Search datasets*" bar on the top allows you to search for a specific dataset by its name.

### Dataset list

In the center of the page you see the list of datasets available in the current workspace. The list consists of following columns:

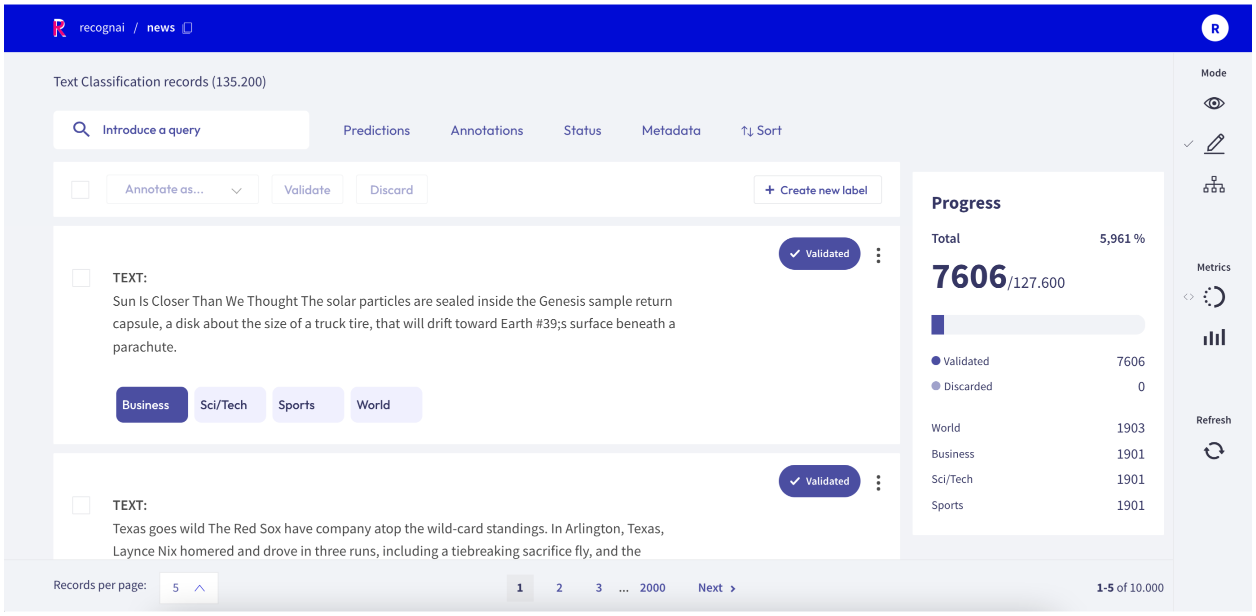
- **Name**: The name of the dataset, can be sorted alphabetically.
- **Tags**: User defined tags for the dataset.
- **Task**: The *task* of the dataset.
- **Created at**: When was the dataset first logged by the client.
- **Updated at**: When was the dataset last modified, either via the Rubrix web app or the client.

### Side bar

On the top right you can find a user icon and a refresh button:

- **User icon**: Showing the initials of your user name, this icon allows you to view and switch the workspace.
- **Refresh**: This button updates the list of datasets in case you just logged new data from the client.

### Dataset



TEXT:
Sun Is Closer Than We Thought The solar particles are sealed inside the Genesis sample return capsule, a disk about the size of a truck tire, that will drift toward Earth #39;s surface beneath a parachute.

TEXT:
Texas goes wild The Red Sox have company atop the wild-card standings. In Arlington, Texas, Laynce Nix homered and drove in three runs, including a tiebreaking sacrifice fly, and the

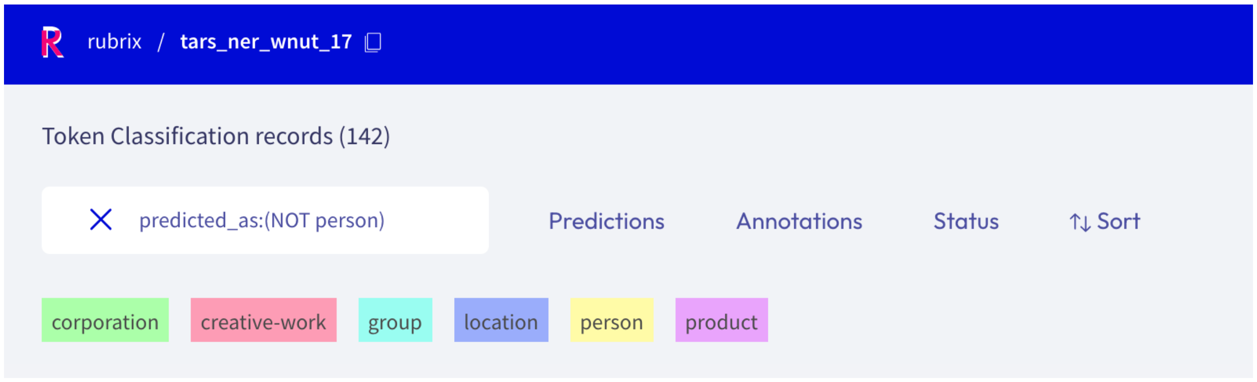
Progress

| | |
|---|---|
| Total | 5,961 % |
| 7606/127.600 | |
| ● Validated | 7606 |
| ● Discarded | 0 |
| World | 1903 |
| Business | 1901 |
| Sci/Tech | 1901 |
| Sports | 1901 |

The *Dataset* page is the main page of the Rubrix web app. From here you can access most of Rubrix's features, like **exploring and annotating** the records of your dataset.

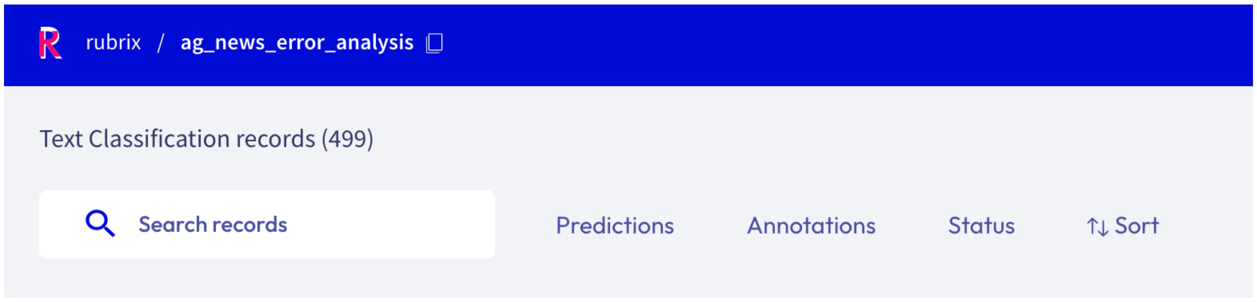
The page is composed of 4 major components:

### Search bar


rubrix / tars_ner_wnut_17
Token Classification records (142)
predicted_as:(NOT person)
Predictions
Annotations
Status
Sort
corporation
creative-work
group
location
person
product

Rubrix's *search bar* is a powerful tool that allows you to thoroughly explore your dataset, and quickly navigate through the records. You can either fuzzy search the contents of your records, or use the more advanced query string syntax of Elasticsearch to take full advantage of Rubrix's *data models*. You can find more information about how to use the search bar in our detailed *search guide*.

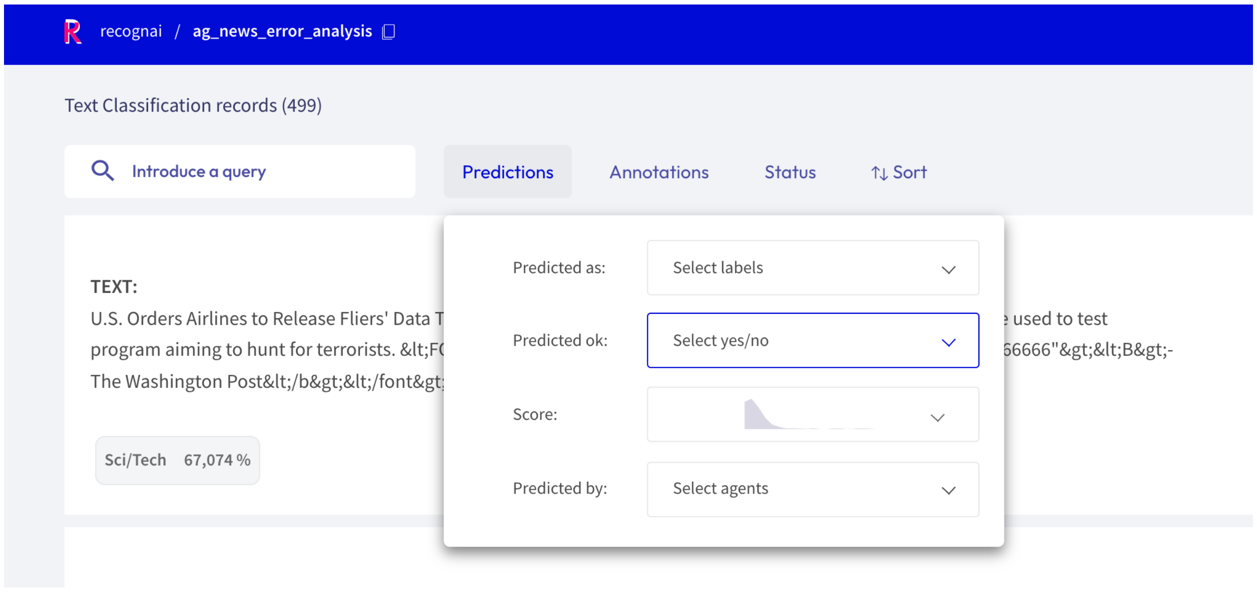
### Filters


rubrix / ag_news_error_analysis
Text Classification records (499)
Search records
Predictions
Annotations
Status
Sort

The *filters* provide you a quick and intuitive way to filter and sort your records with respect to various parameters. You can find more information about how to use the filters in our detailed *filter guide*.

**Note:** Not all filters are available for all *tasks*.

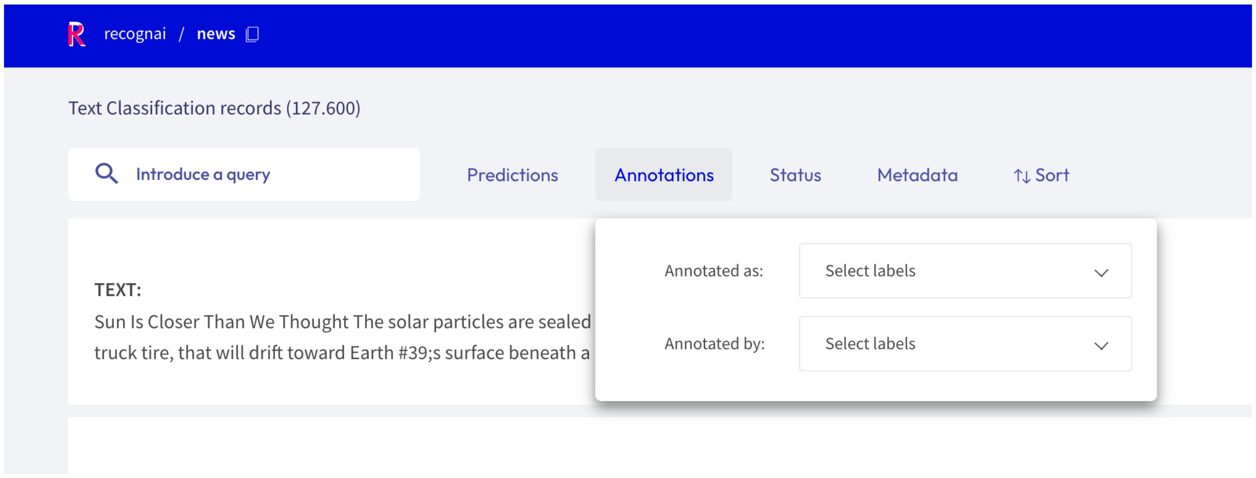
### Predictions filter


recognai / ag_news_error_analysis
Text Classification records (499)
Introduce a query
Predictions
Annotations
Status
Sort
TEXT:
U.S. Orders Airlines to Release Fliers' Data T
program aiming to hunt for terrorists. &lt;F
The Washington Post&lt;/b&gt;&lt;/font&gt;
used to test
66666"&gt;&lt;B&gt;-
Sci/Tech 67,074 %
Predicted as: Select labels
Predicted ok: Select yes/no
Score:
Predicted by: Select agents

This filter allows you to filter records with respect of their predictions:

- **Predicted as**: filter records by their predicted labels
- **Predicted ok**: filter records whose predictions do, or do not, match the annotations
- **Score**: filter records with respect to the score of their prediction
- **Predicted by**: filter records by the *prediction agent*

### Annotations filter

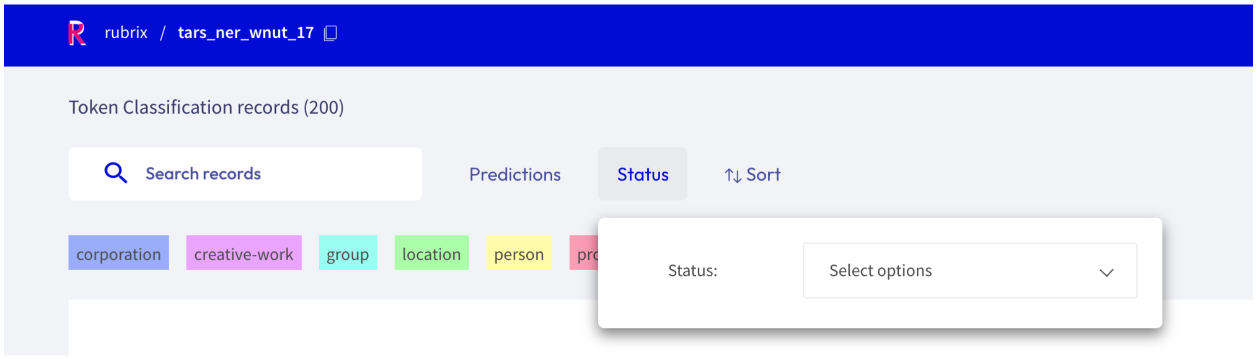


TEXT:
Sun Is Closer Than We Thought The solar particles are sealed
truck tire, that will drift toward Earth #39;s surface beneath a

This filter allows you to filter records with respect to their annotations:

- **Annotated as**: filter records with respect to their annotated labels
- **Annotated by**: filter records by the *annotation agent*

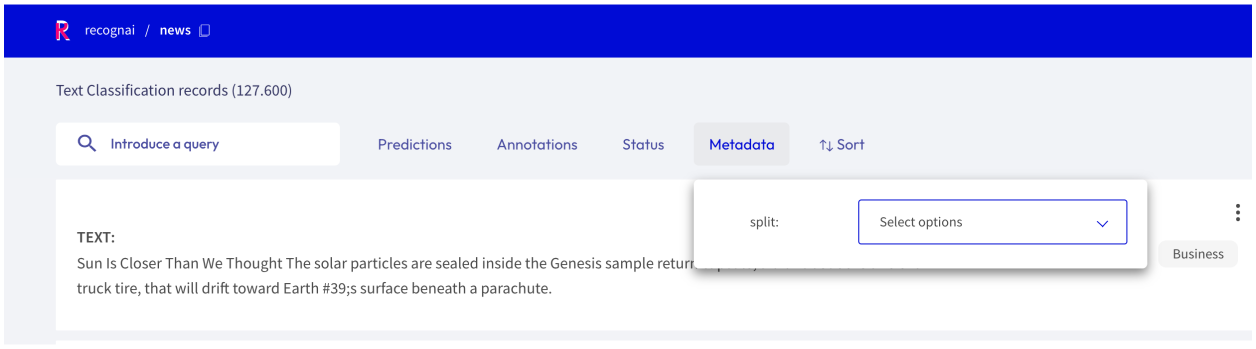
### Status filter



This filter allows you to filter records with respect to their status:

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

### Metadata filter

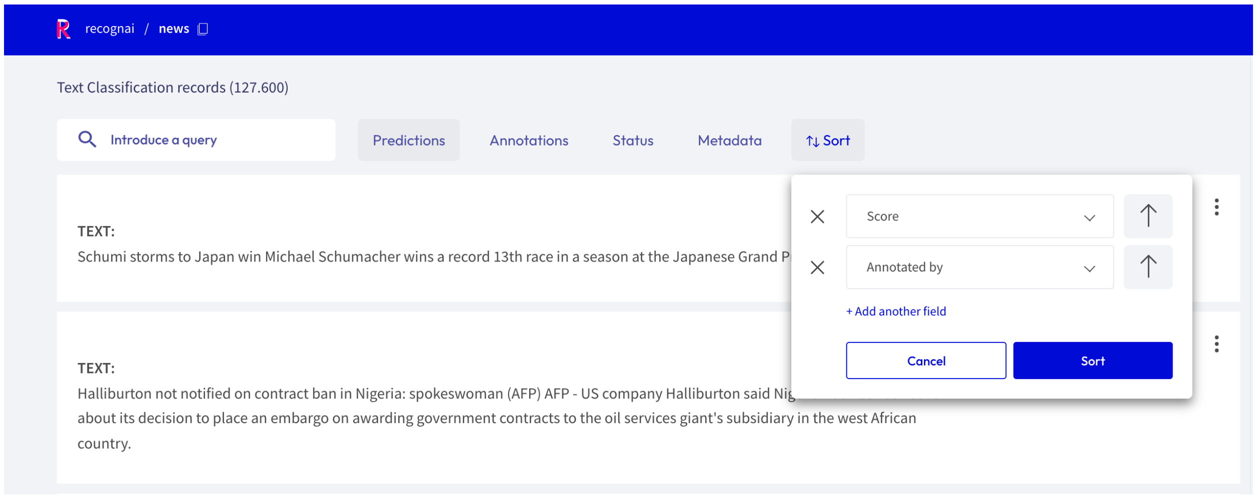


TEXT:
Sun Is Closer Than We Thought The solar particles are sealed inside the Genesis sample retur
truck tire, that will drift toward Earth #39;s surface beneath a parachute.

This filter allows you to filter records with respect to their metadata.

**Hint:** Nested metadata will be flattened and the keys will be joint by a dot.

### Sort records



TEXT:
Schumi storms to Japan win Michael Schumacher wins a record 13th race in a season at the Japanese Grand P

TEXT:
Halliburton not notified on contract ban in Nigeria: spokeswoman (AFP) AFP - US company Halliburton said Nig
about its decision to place an embargo on awarding government contracts to the oil services giant's subsidiary in the west African country.

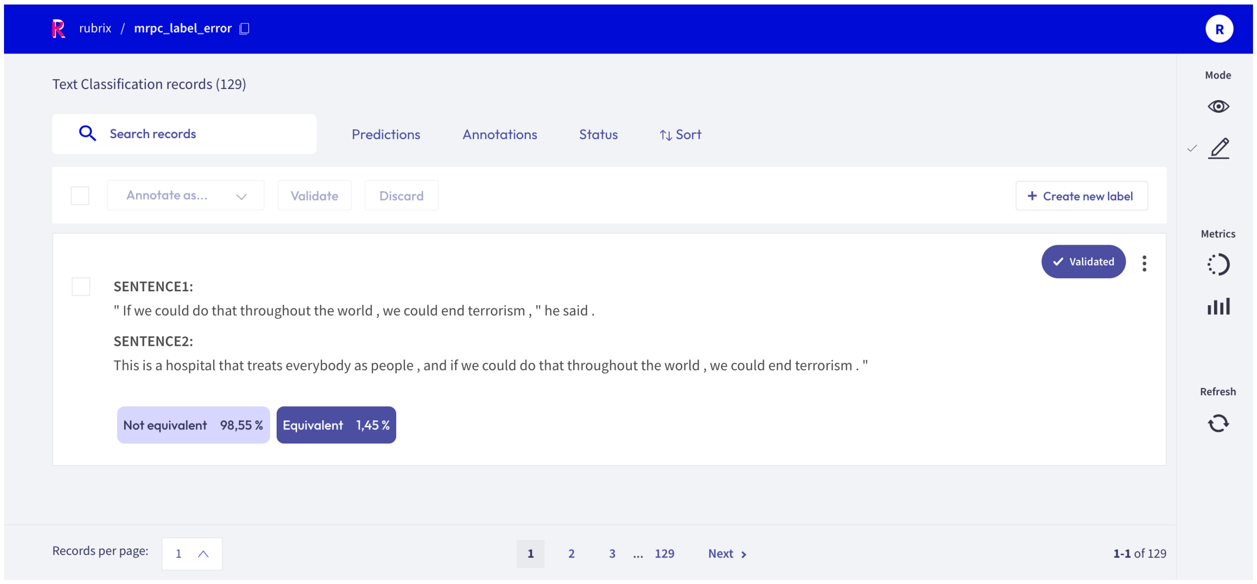
With this component you can sort the records by various parameters, such as the predictions, annotations or their metadata.

### Record cards

The record cards are at the heart of the *Dataset* page and contain your data. There are three different flavors of record cards depending on the *task* of your dataset. All of them share the same basic structure showing the input text and a vertical ellipsis (or "kebab menu") on the top right that lets you access the record's metadata. Predictions and annotations are shown depending on the current *mode* and *task* of the dataset.

Check out our *exploration* and *annotation* guides to see how the record cards work in the different *modes*.

### Text classification



Text Classification records (129)

SENTENCE1:
" If we could do that throughout the world , we could end terrorism , " he said .

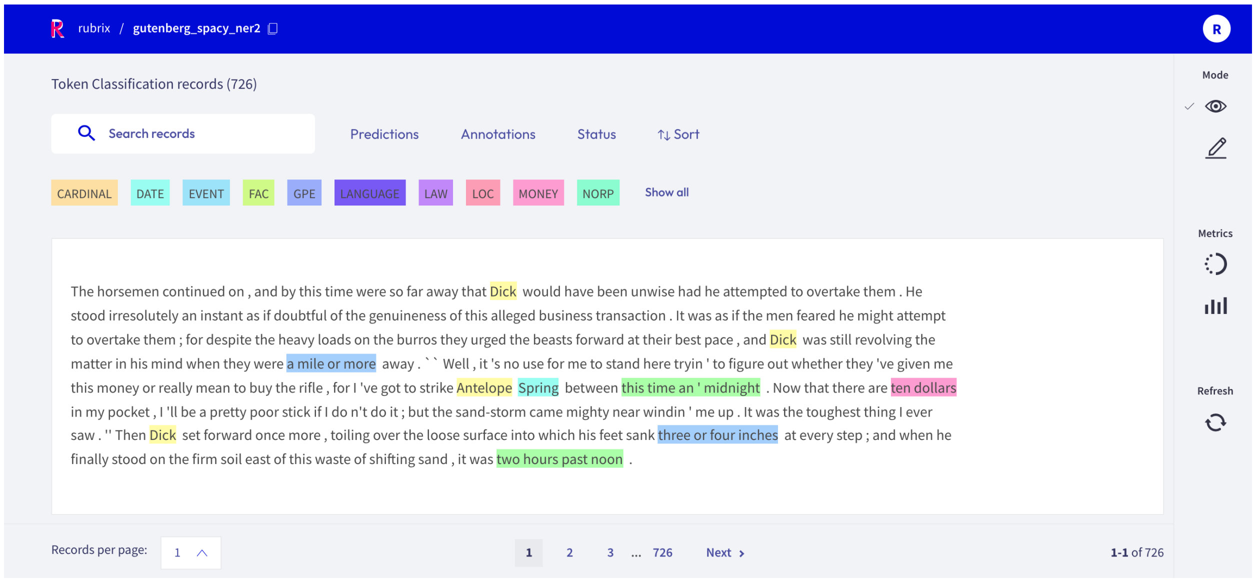
SENTENCE2:
This is a hospital that treats everybody as people , and if we could do that throughout the world , we could end terrorism . "

Not equivalent 98,55 %  Equivalent 1,45 %

In this task the predictions are given as tags below the input text. They contain the label as well as a percentage score. When in *Explore mode* annotations are shown as tags on the right together with a symbol indicating if the predictions match the annotations or not. When in *Annotate mode* predictions and annotations share the same labels (annotation labels are darker).

A text classification dataset can support either single-label or multi-label classification - in other words, records are either annotated with one single label or various.

### Token classification

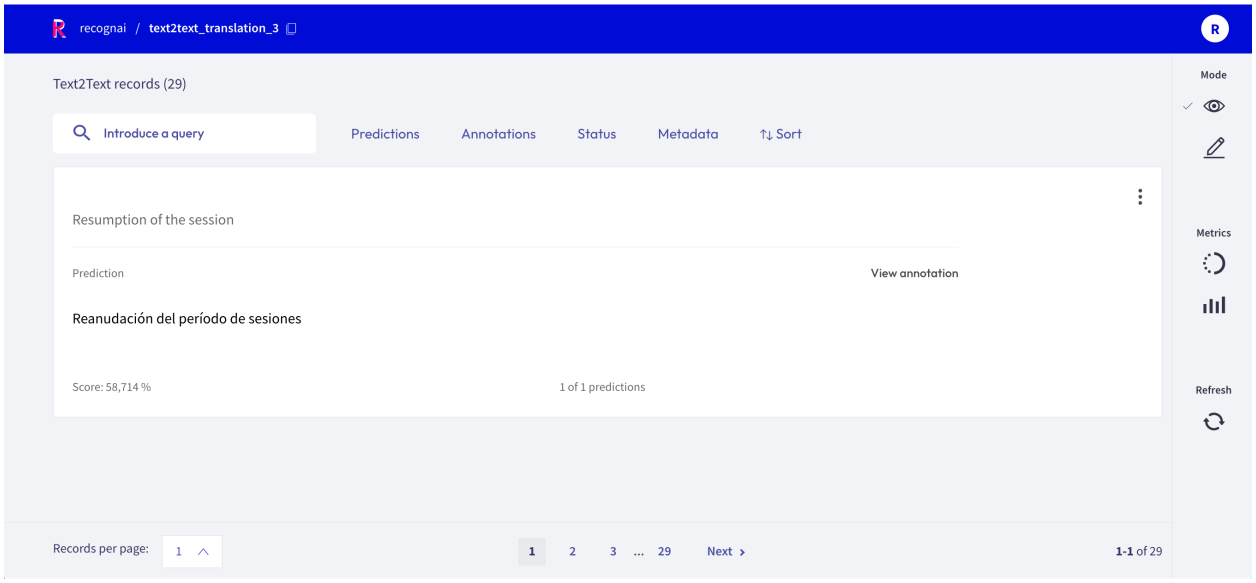


Token Classification records (726)

The horsemen continued on , and by this time were so far away that Dick would have been unwise had he attempted to overtake them . He stood irresolutely an instant as if doubtful of the genuineness of this alleged business transaction . It was as if the men feared he might attempt to overtake them ; for despite the heavy loads on the burros they urged the beasts forward at their best pace , and Dick was still revolving the matter in his mind when they were a mile or more away . `` Well , it 's no use for me to stand here tryin ' to figure out whether they 've given me this money or really mean to buy the rifle , for I 've got to strike Antelope Spring between this time an ' midnight . Now that there are ten dollars in my pocket , I 'll be a pretty poor stick if I do n't do it ; but the sand-storm came mighty near windin ' me up . It was the toughest thing I ever saw . '' Then Dick set forward once more , toiling over the loose surface into which his feet sank three or four inches at every step ; and when he finally stood on the firm soil east of this waste of shifting sand , it was two hours past noon .

In this task predictions and annotation are given as highlights in the input text. Work in progress …

### Text2Text



Resumption of the session

Prediction

View annotation

Reanudación del período de sesiones

Score: 58,714 %

1 of 1 predictions

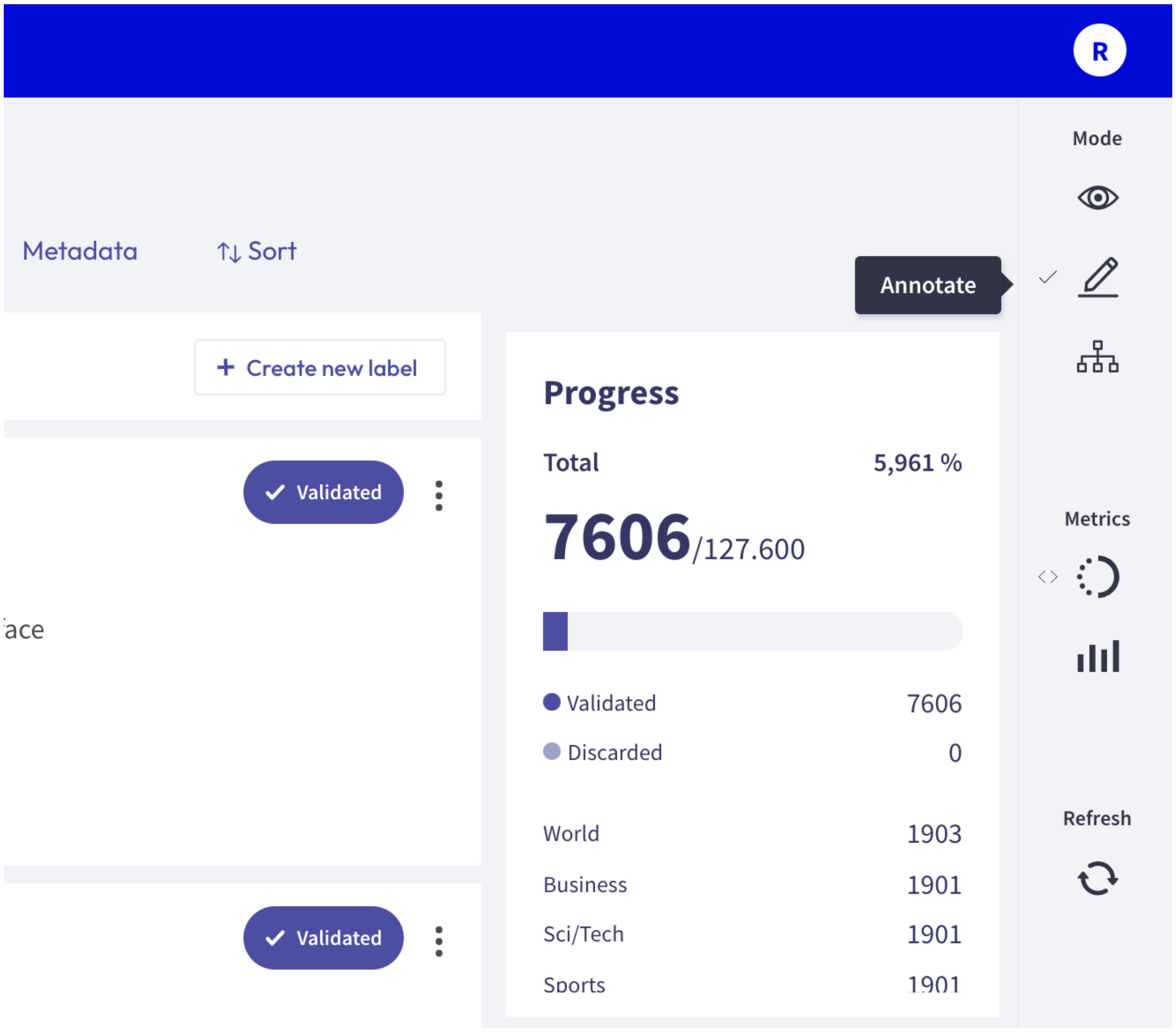
Records per page: 1

1 2 3 ... 29 Next

1-1 of 29

In this task predictions and the annotation are given in a text field below the input text. You can switch between prediction and annotation via the "*View annotation*"/"*View predictions*" buttons. For the predictions you can find an associated score in the lower left corner. If you have multiple predictions you can toggle between them using the arrows on the button of the record card.

### Sidebar



The sidebar is divided into three sections.

#### Modes

This section of the sidebar lets you switch between the different Rubrix modes that are covered extensively in their respective guides:

- **Explore**: this mode is for *exploring your dataset* and gain valuable insights
- **Annotate**: this mode lets you conveniently *annotate your data*
- **Define rules**: this mode helps you to *define rules* to automatically label your data

**Note:** Not all modes are available for all *tasks*.

### Metrics

In this section you find several "metrics" that can provide valuable insights to your dataset, or support you while annotating your records. They are grouped into two submenus:

- **Progress**: see metrics of your annotation process, like its progress and the label distribution
- **Stats**: check the keywords of your dataset and the error distribution of the predictions

You can find more information about each metric in our dedicated *metrics guide*.

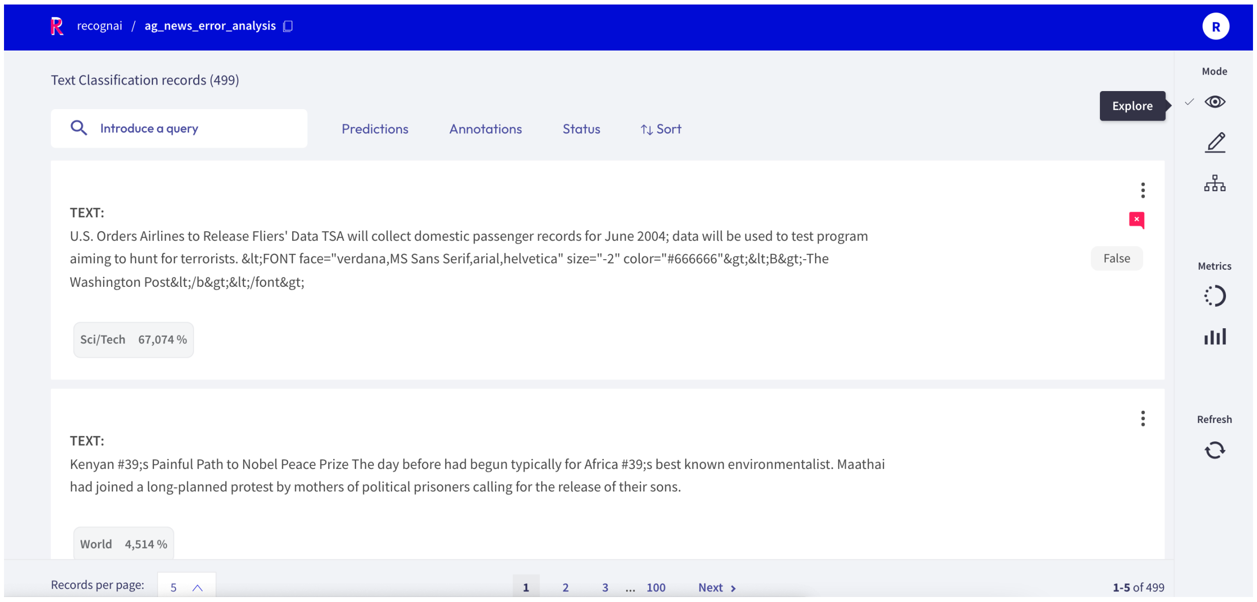
### Refresh

This button allows you to refresh the list of the record cards with respect to the activated filters. For example, if you are annotating and use the *Status filter* to filter out annotated records, you can press the *Refresh* button to hide the latest annotated records.

## 5.20.2 Features

- ***Explore records***: Explore your data and predictions, as well as your annotations
- ***Annotate records***: Annotate your records for different *tasks*
- ***Define rules***: Define rules to weakly supervise your data
- ***Search records***: Search your records with the powerful elasticsearch query syntax
- ***Filter records***: Quickly filter your records by predictions, annotations or their metadata
- ***View dataset metrics***: View insightful metrics of your dataset
- ***Switch workspaces***: Switch your workspace and access shared datasets

### Explore records



TEXT:
U.S. Orders Airlines to Release Fliers' Data TSA will collect domestic passenger records for June 2004; data will be used to test program aiming to hunt for terrorists. &lt;FONT face="verdana,MS Sans Serif,arial,helvetica" size="-2" color="#666666"&gt;&lt;B&gt;-The Washington Post&lt;/b&gt;&lt;/font&gt;

False

Sci/Tech 67,074 %

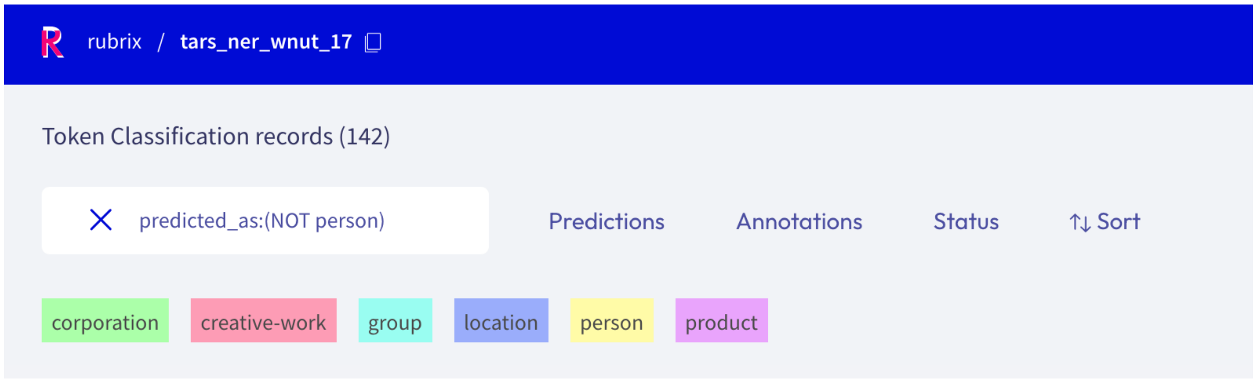
TEXT:
Kenyan #39;s Painful Path to Nobel Peace Prize The day before had begun typically for Africa #39;s best known environmentalist. Maathai had joined a long-planned protest by mothers of political prisoners calling for the release of their sons.

World 4,514 %

If you want to explore your dataset or analyze the predictions of a model, the Rubrix web app offers a dedicated Explore mode. The powerful search functionality and intuitive filters allow you to quickly navigate through your records and dive deep into your dataset. At the same time, you can view the predictions and compare them to gold annotations.

You can access the *Explore mode* via the sidebar of the *Dataset page*.

### Search and filter


rubrix / tars_ner_wnut_17
Token Classification records (142)
predicted_as:(NOT person)
Predictions
Annotations
Status
Sort
corporation
creative-work
group
location
person
product

The powerful search bar allows you to do simple, quick searches, as well as complex queries that take full advantage of Rubrix's *data models*. In addition, the *filters* provide you a quick and intuitive way to filter and sort your records with respect to various parameters, including predictions and annotations. Both of the components can be used together to dissect in-depth your dataset, validate hunches, and find specific records.

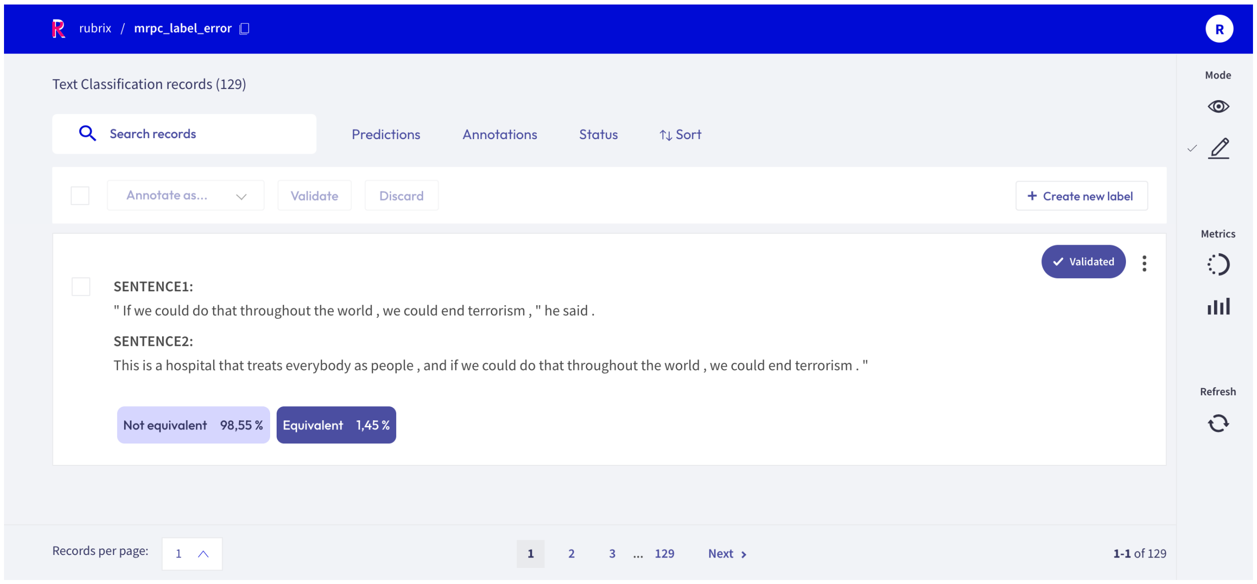
You can find more information about how to use the search bar and the filters in our detailed *search guide* and *filter guide*.

**Note:** Not all filters are available for all *tasks*.

### Predictions and annotations

Predictions and annotations are an integral part of Rubrix's *data models*. The way they are presented in the Rubrix web app depends on the *task* of the dataset.

### Text classification



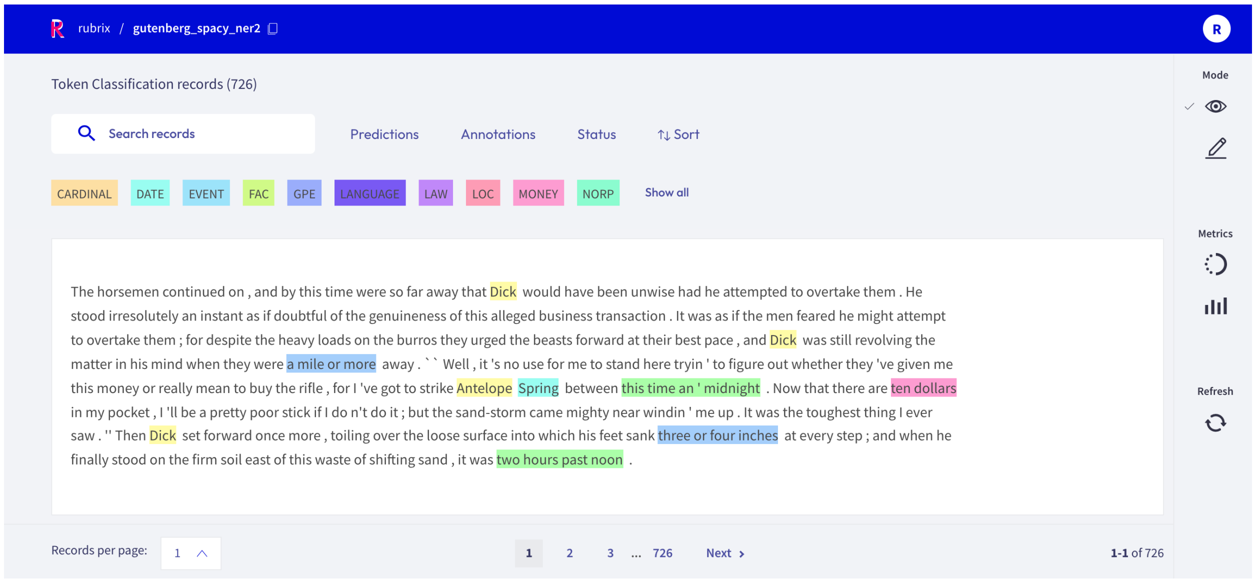
SENTENCE1:
" If we could do that throughout the world , we could end terrorism , " he said .

SENTENCE2:
This is a hospital that treats everybody as people , and if we could do that throughout the world , we could end terrorism . "

Not equivalent 98,55 % Equivalent 1,45 %

In this task the predictions are given as tags below the input text. They contain the label as well as a percentage score. Annotations are shown as tags on the right together with a symbol indicating if the predictions match the annotations or not.

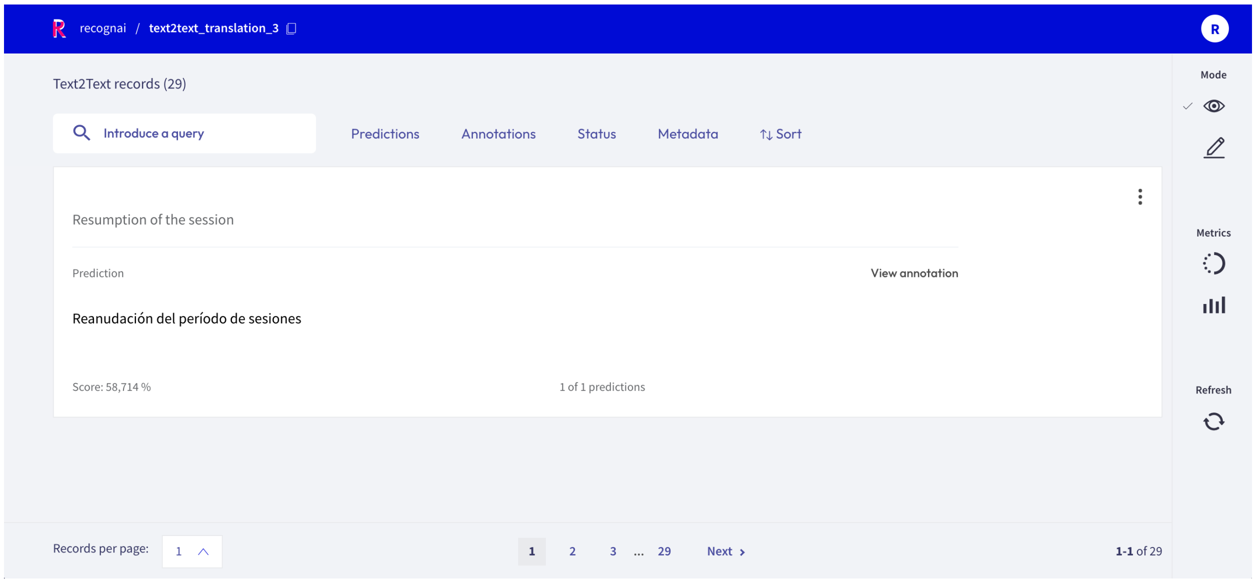
### Token classification



The horsemen continued on , and by this time were so far away that Dick would have been unwise had he attempted to overtake them . He stood irresolutely an instant as if doubtful of the genuineness of this alleged business transaction . It was as if the men feared he might attempt to overtake them ; for despite the heavy loads on the burros they urged the beasts forward at their best pace , and Dick was still revolving the matter in his mind when they were a mile or more away . ` ` Well , it 's no use for me to stand here tryin ' to figure out whether they 've given me this money or really mean to buy the rifle , for I 've got to strike Antelope Spring between this time an ' midnight . Now that there are ten dollars in my pocket , I 'll be a pretty poor stick if I do n't do it ; but the sand-storm came mighty near windin ' me up . It was the toughest thing I ever saw . '' Then Dick set forward once more , toiling over the loose surface into which his feet sank three or four inches at every step ; and when he finally stood on the firm soil east of this waste of shifting sand , it was two hours past noon .

In this task predictions and annotation are given as highlights in the input text. Work in progress . . .

## Text2Text



Resumption of the session

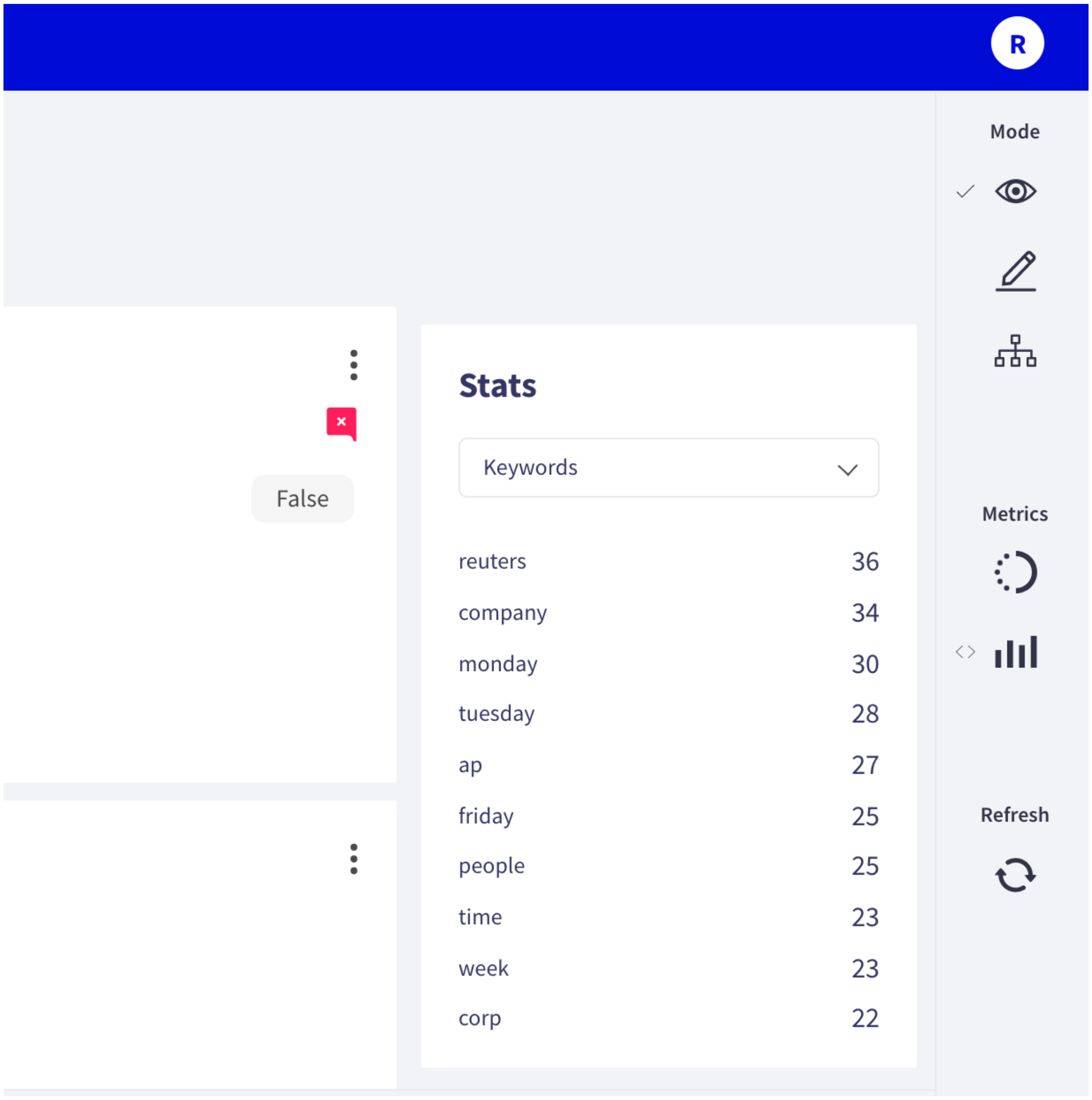
Reanudación del período de sesiones

Score: 58,714 %

1 of 1 predictions

In this task predictions and the annotation are given in a text field below the input text. You can switch between prediction and annotation via the "*View annotation*"/"*View predictions*" buttons. For the predictions you can find an associated score in the lower left corner. If you have multiple predictions you can toggle between them using the arrows on the button of the record card.

## Metrics



| | |
|---|---|
| reuters | 36 |
| company | 34 |
| monday | 30 |
| tuesday | 28 |
| ap | 27 |
| friday | 25 |
| people | 25 |
| time | 23 |
| week | 23 |
| corp | 22 |

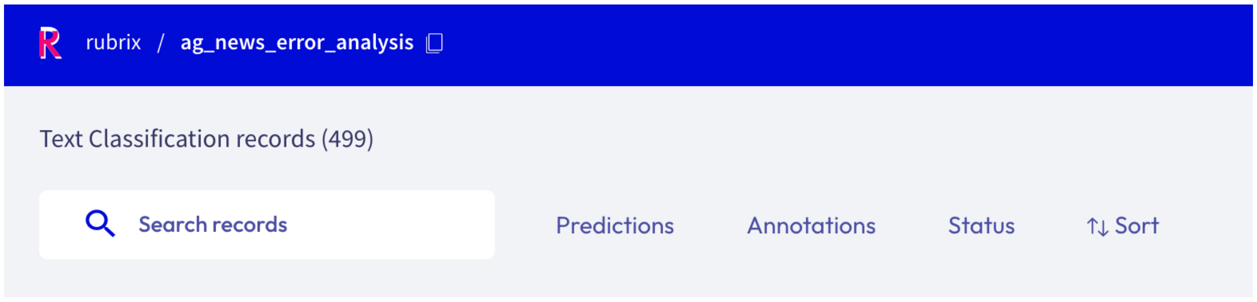
From the side bar you can access the *Stats metrics* that provide support for your analysis of the dataset.

### Annotate records

The Rubrix web app has a dedicated mode to quickly label your data in a very intuitive way, or revise previous gold labels and correct them. Rubrix's powerful search and filter functionalities, together with potential model predictions, can guide the annotation process and support the annotator.

You can access the *Annotate mode* via the sidebar of the *Dataset page*.

#### Search and filter


rubrix / ag_news_error_analysis
Text Classification records (499)
Search records
Predictions
Annotations
Status
Sort

The powerful search bar allows you to do simple, quick searches, as well as complex queries that take full advantage of Rubrix's *data models*. In addition, the *filters* provide you a quick and intuitive way to filter and sort your records with respect to various parameters, including the metadata of your records. For example, you can use the Status filter to hide already annotated records (*Status: Default*), or to only show annotated records when revising previous annotations (*Status: Validated*).

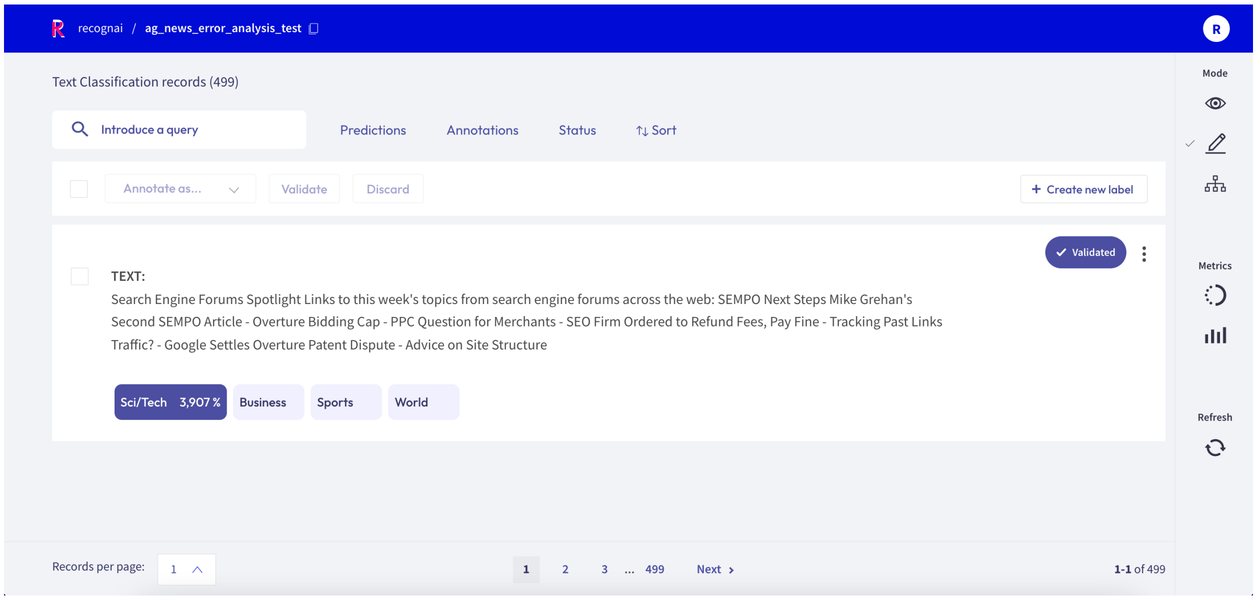
You can find more information about how to use the search bar and the filters in our detailed *search guide* and *filter guide*.

**Note:** Not all filters are available for all *tasks*.

#### Annotate

To annotate the records, the Rubrix web app provides a simple and intuitive interface that tries to follow the same interaction pattern as in the *Explore mode*. As the *Explore mode*, the record cards in the *Annotate mode* are also customized depending on the *task* of the dataset.

### Text Classification



TEXT:
Search Engine Forums Spotlight Links to this week's topics from search engine forums across the web: SEMPO Next Steps Mike Grehan's Second SEMPO Article - Overture Bidding Cap - PPC Question for Merchants - SEO Firm Ordered to Refund Fees, Pay Fine - Tracking Past Links Traffic? - Google Settles Overture Patent Dispute - Advice on Site Structure

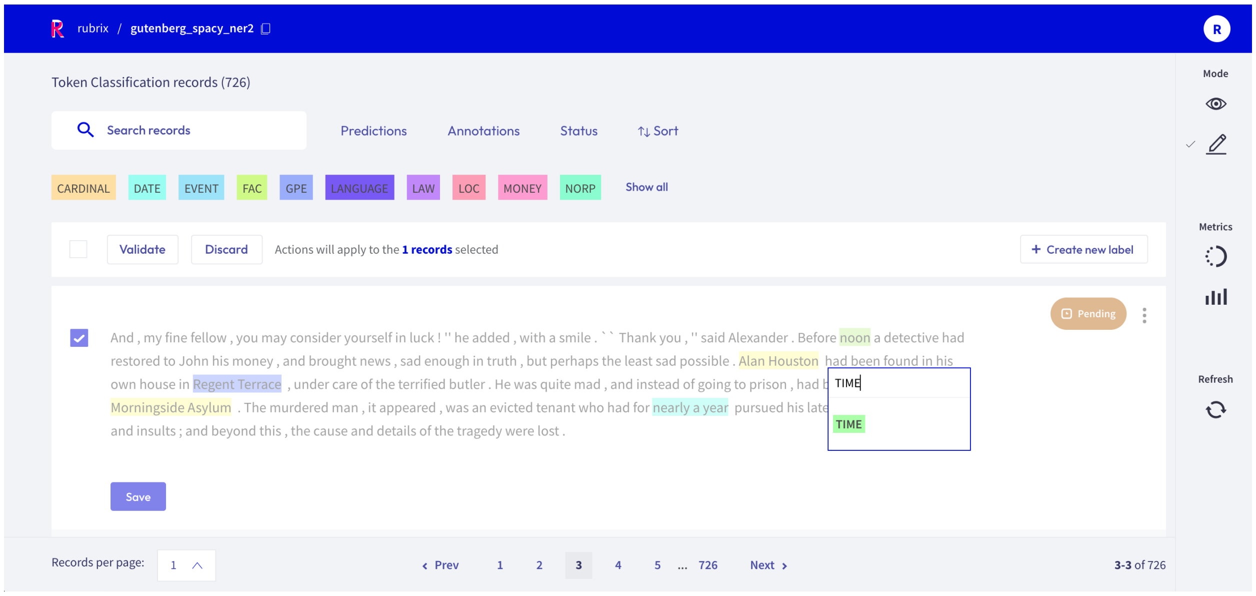
Sci/Tech 3,907 % | Business | Sports | World

When switching in the *Annotate mode* for a text classification dataset, the labels in the record cards become clickable and you can annotate the records by simply clicking on them. You can also validate the predictions shown in a slightly darker tone by pressing the *Validate* button:

- for a **single label** classification task, this will be the prediction with the highest percentage
- for a **multi label** classification task, this will be the predictions with a percentage above 50%

Once a record is annotated, it will be marked as *Validated* in the upper right corner of the record card.

### Token Classification

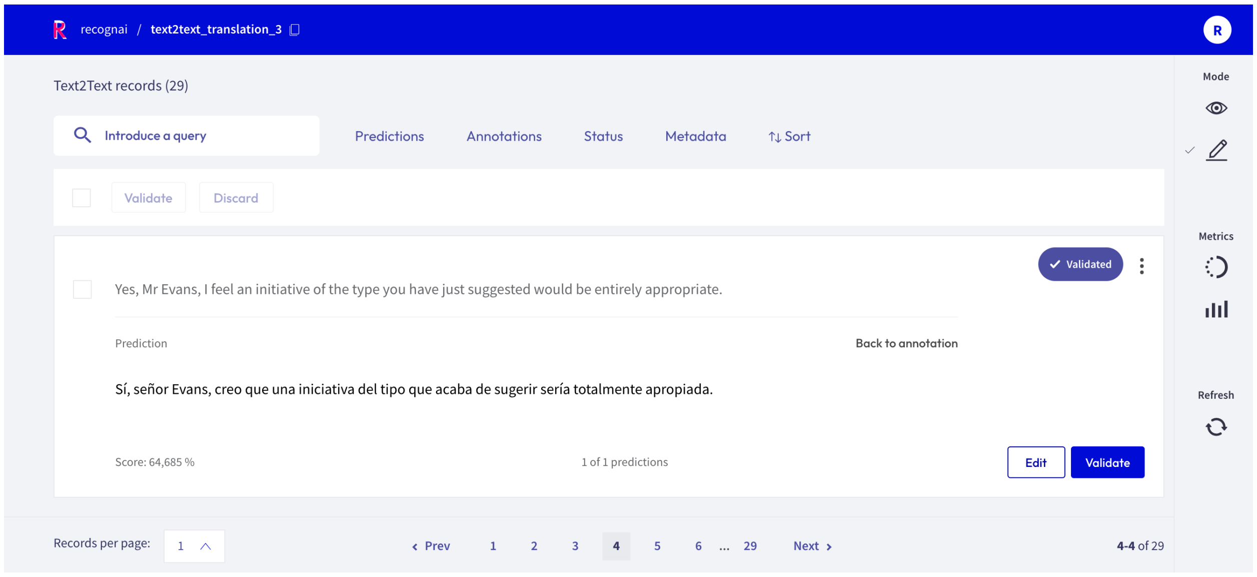


And , my fine fellow , you may consider yourself in luck ! '' he added , with a smile . `` Thank you , '' said Alexander . Before noon a detective had restored to John his money , and brought news , sad enough in truth , but perhaps the least sad possible . Alan Houston had been found in his own house in Regent Terrace , under care of the terrified butler . He was quite mad , and instead of going to prison , had b
Morningside Asylum . The murdered man , it appeared , was an evicted tenant who had for nearly a year pursued his late
and insults ; and beyond this , the cause and details of the tragedy were lost .

For token classification datasets, you can highlight words (tokens) in the text and annotate them with a label. Under the hood, the highlighting takes advantage of the `tokens` information in the *Token Classification data model*. You can also remove annotations by hovering over the highlights and pressing the *X* button.

After modifying a record, either by adding or removing annotations, its status will change to *Pending* and a *Save* button will appear. You can also validate the predictions (or the absent of them) by pressing the *Validate* button. Once the record is saved or validated, its status will change to *Validated*.

## Text2Text



Yes, Mr Evans, I feel an initiative of the type you have just suggested would be entirely appropriate.

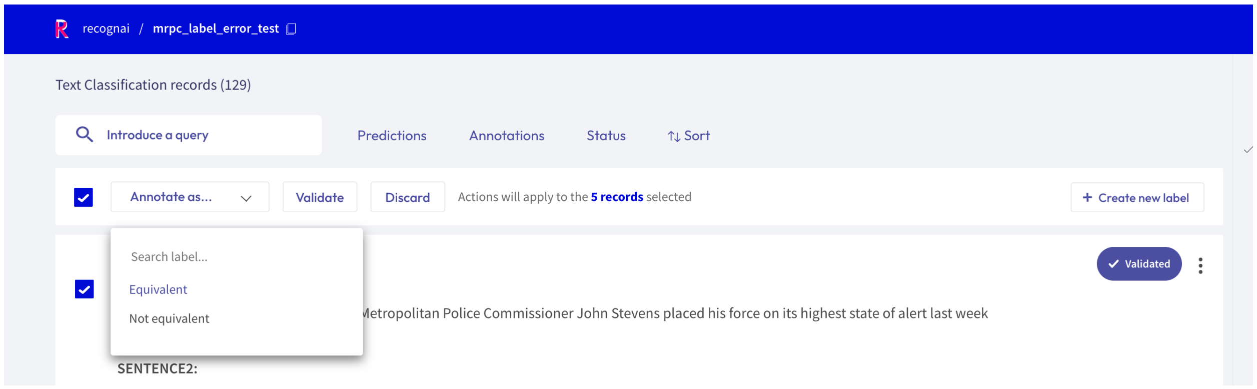
Sí, señor Evans, creo que una iniciativa del tipo que acaba de sugerir sería totalmente apropiada.

Score: 64,685 %

1 of 1 predictions

For text2text datasets, you have a text box available, in which you can draft or edit an annotation. You can also validate or edit a prediction, by first clicking on the *view predictions* button, and then the *Edit* or *Validate* button. After editing or drafting your annotation, don't forget to save your changes.

## Bulk annotate



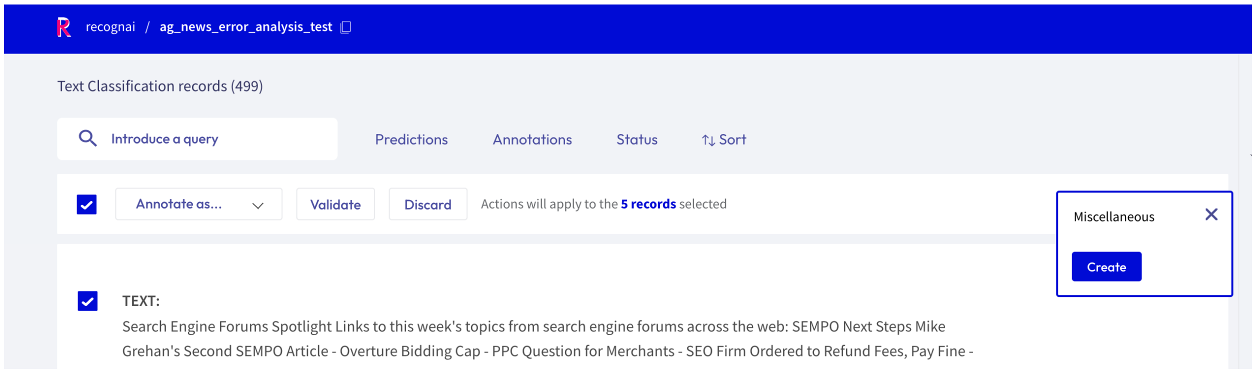
Actions will apply to the 5 records selected

Metropolitan Police Commissioner John Stevens placed his force on its highest state of alert last week

For all *tasks*, you can **bulk validate** the predictions of the records. You can either select the records one by one with the selection box on the upper left of each card, or you can use the global selection box below the search bar, which will select all records shown on the page. Then you can either *Validate* or *Discard* the selected records.

For the text classification task, you can additionally **bulk annotate** the selected records with a specific label, by simply selecting the label from the *"Annotate as . . . "* list.

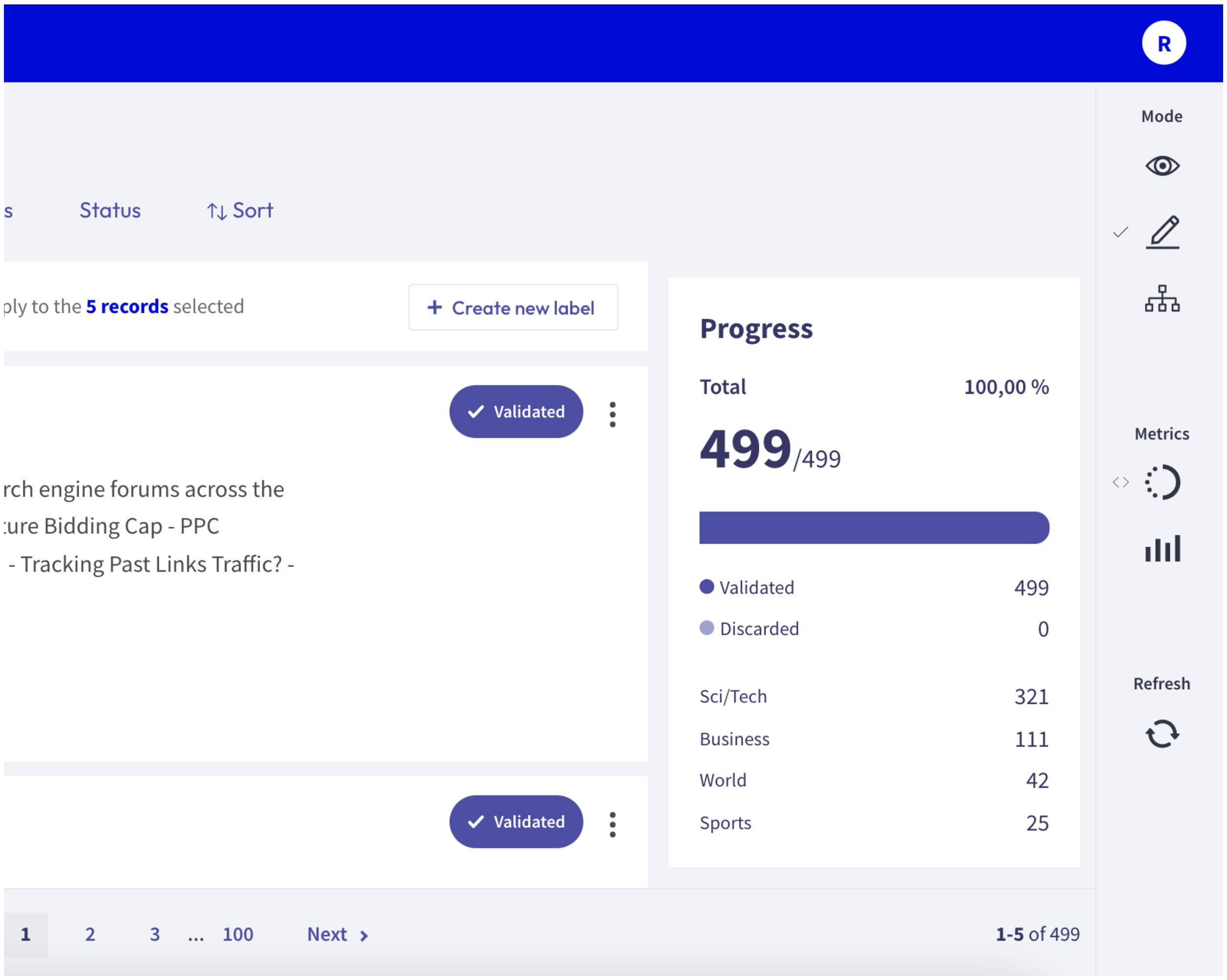
### Create labels


recognai / ag_news_error_analysis_test
Text Classification records (499)
Introduce a query
Predictions
Annotations
Status
Sort
Annotate as...
Validate
Discard
Actions will apply to the 5 records selected
Miscellaneous
Create
TEXT:

Search Engine Forums Spotlight Links to this week's topics from search engine forums across the web: SEMPO Next Steps Mike Grehan's Second SEMPO Article - Overture Bidding Cap - PPC Question for Merchants - SEO Firm Ordered to Refund Fees, Pay Fine -

For the text and token classification tasks, you can create new labels within the *Annotate mode*. On the right side of the bulk validation bar, you will find a *"+ Create new label"* button that lets you add new labels to your dataset.

### Progress metric



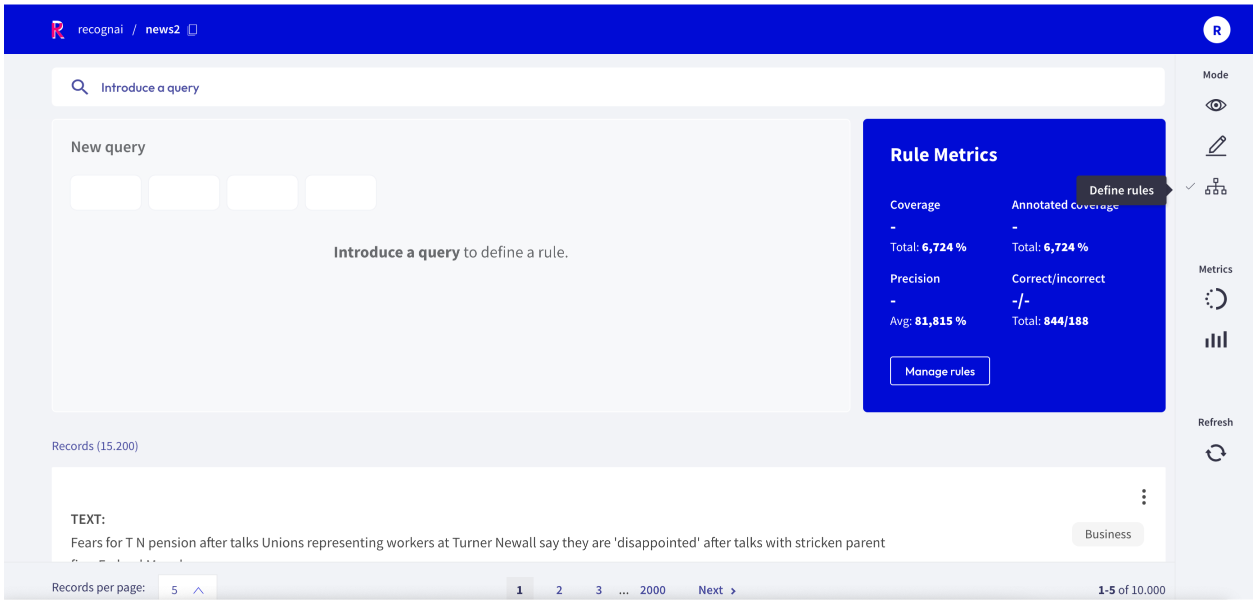
**Progress**

| Total | 100,00 % |
|---|---|
| 499/499 | |
| ● Validated | 499 |
| ● Discarded | 0 |
| Sci/Tech | 321 |
| Business | 111 |
| World | 42 |
| Sports | 25 |

From the sidebar you can access the *Progress metrics*. There you will find the progress of your annotation session, the distribution of validated and discarded records, and the label distribution of your annotations.

You can find more information about the metrics in our dedicated *metrics guide*.

### Define rules



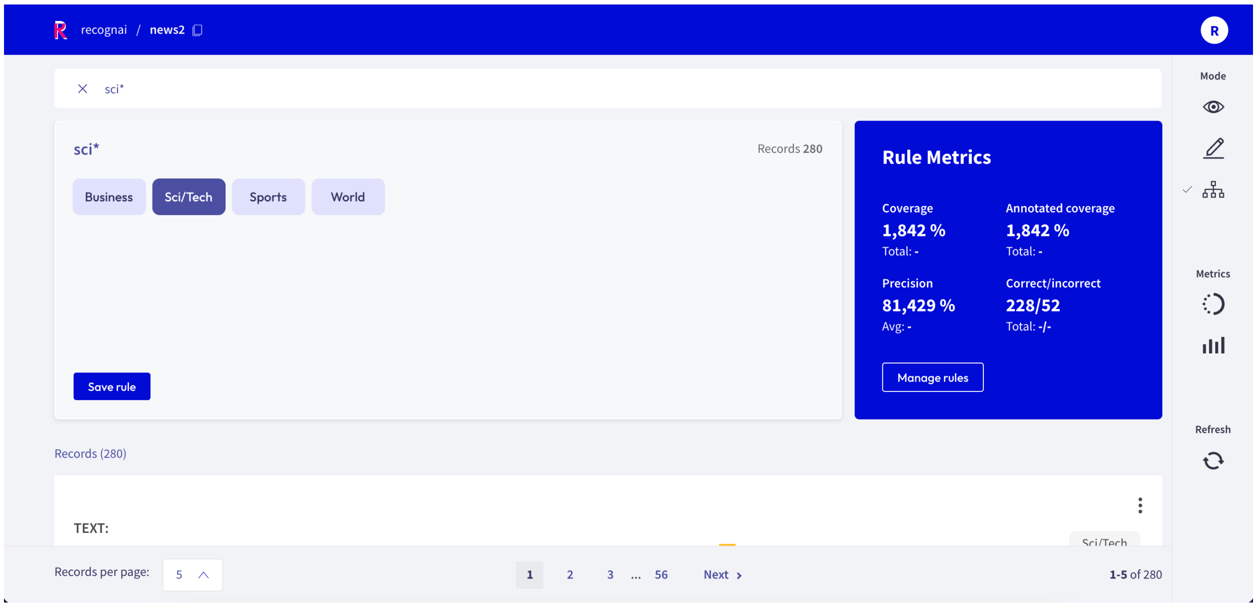
Fears for T N pension after talks Unions representing workers at Turner Newall say they are 'disappointed' after talks with stricken parent

The Rubrix web app has a dedicated mode to find good **heuristic rules**, also often referred to as *labeling functions*, for a weak supervision workflow. As shown in our *guide* and *tutorial*, these rules allow you to quickly annotate your data with noisy labels in a semiautomatic way.

You can access the *Define rules* mode via the sidebar of the *Dataset page*.

**Note:** The *Define rules* mode is only available for single-label text classification datasets.

### Query plus label

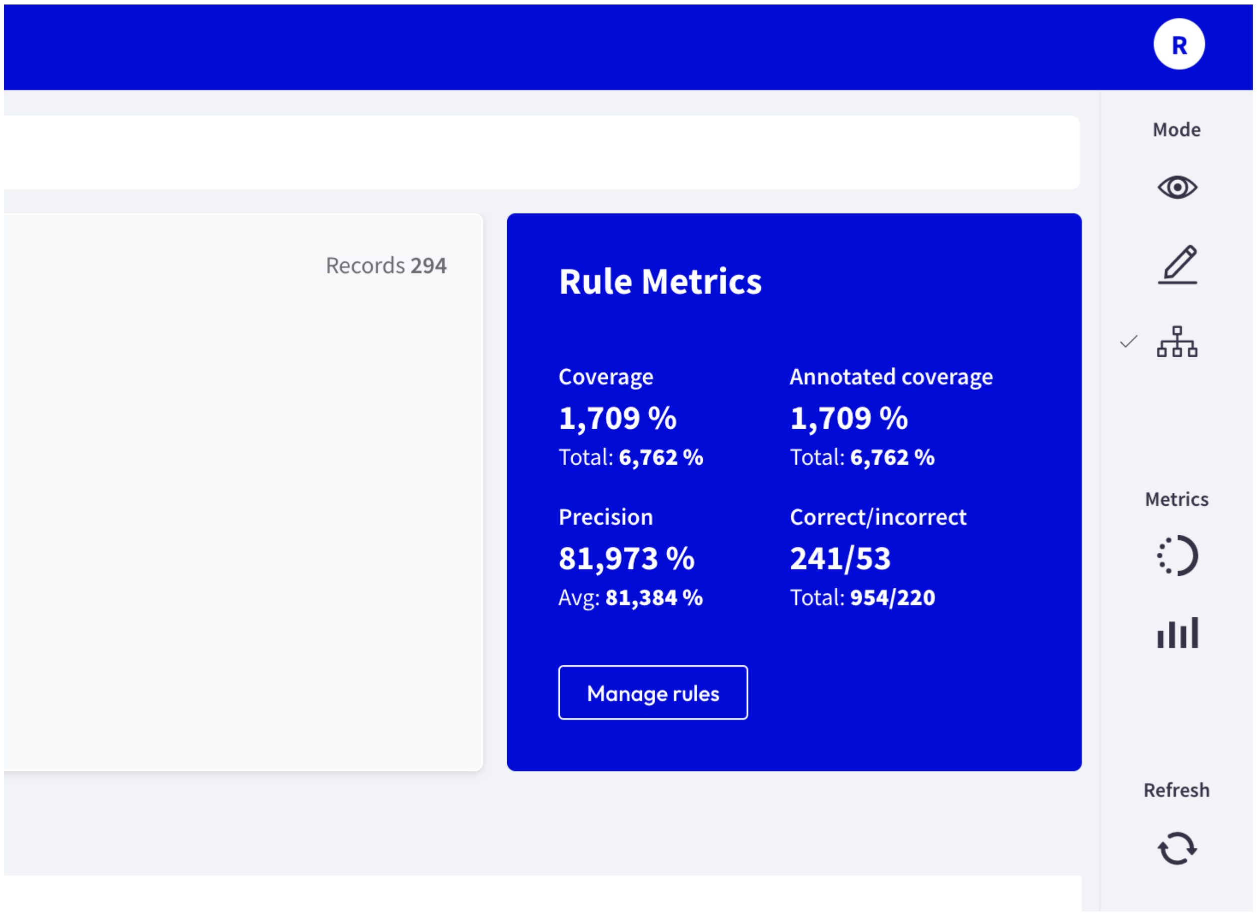


A rule in Rubrix basically applies a chosen label to a list of records that match a given *query*, so all you need is a query plus a label. After entering a query in the search bar and selecting a label, you will see some *metrics* for the rule on the right and the matches of your query in the record list below.

If you are happy with the metrics and/or the matching record list, you can save the rule by clicking on “Save rule”. In this way it will be stored as part of the current dataset and can be accessed via the *manage rules* button.

**Note:** If you want to add labels to the available list of labels, you can switch to the *Annotation mode* and create labels there.

### Rule Metrics



Rule Metrics

Coverage
1,709 %
Total: 6,762 %

Annotated coverage
1,709 %
Total: 6,762 %

Precision
81,973 %
Avg: 81,384 %

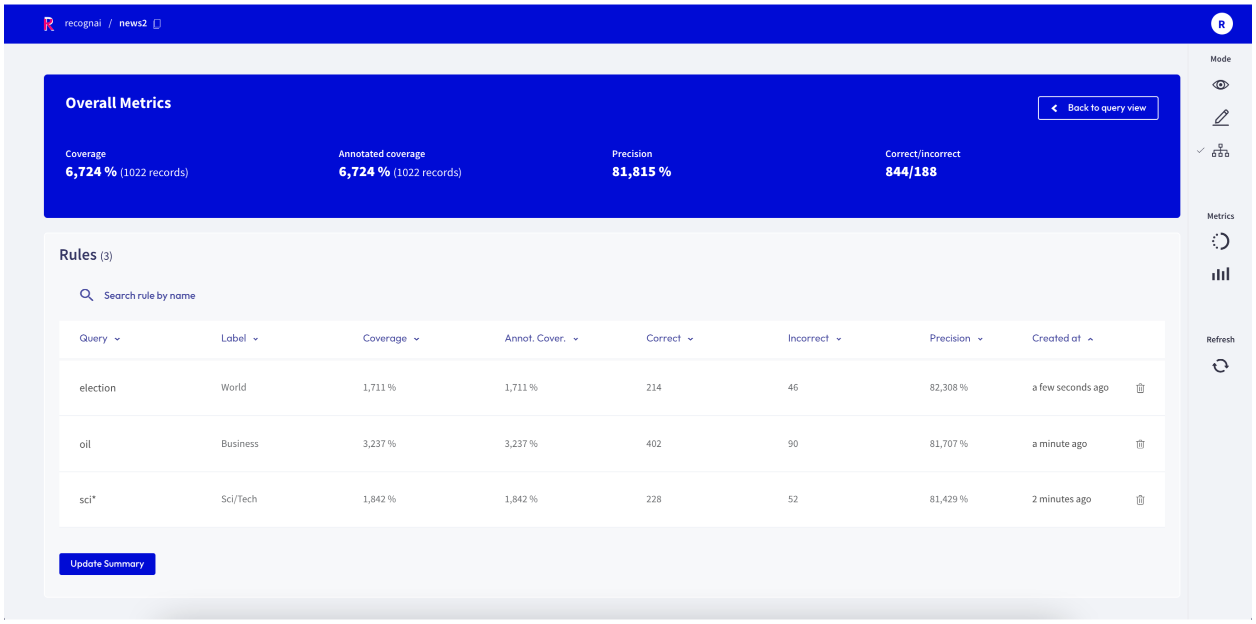
Correct/incorrect
241/53
Total: 954/220

After entering a query and selecting a label, Rubrix provides you with some key metrics about the rule. Some metrics are only available if your dataset has also annotated records.

- **Coverage**: Number of records (percentage) of records labeled by the rule
- **Annotated coverage**: Number of records (percentage) of annotated records labeled by the rule
- **Correct/incorrect**: Number of records the rule labelled correctly/incorrectly (if annotations are available)
- **Precision**: Percentage of correct labels given by the rule (if annotations are available)

### Manage rules

Here you will see a list of your saved rules as well as their overall metrics. You can edit a rule by clicking on its name, or delete it by clicking on the trash icon.



Overall Metrics

| Coverage | Annotated coverage | Precision | Correct/incorrect |
|---|---|---|---|
| 6,724 % (1022 records) | 6,724 % (1022 records) | 81,815 % | 844/188 |

Rules (3)

| Query | Label | Coverage | Annot. Cover. | Correct | Incorrect | Precision | Created at |
|---|---|---|---|---|---|---|---|
| election | World | 1,711 % | 1,711 % | 214 | 46 | 82,308 % | a few seconds ago |
| oil | Business | 3,237 % | 3,237 % | 402 | 90 | 81,707 % | a minute ago |
| sci* | Sci/Tech | 1,842 % | 1,842 % | 228 | 52 | 81,429 % | 2 minutes ago |

### Search Records

You can search records by using full-text queries (a normal search), or by Elasticsearch with its query string syntax.

This component enables:

1. **Full-text queries** over all record `inputs`.
2. Queries using **Elasticsearch's query DSL** with the query string syntax. Some examples are: `-inputs.text:(women AND feminists)` : records containing the words "women" AND "feminist" in the inputs.text field.

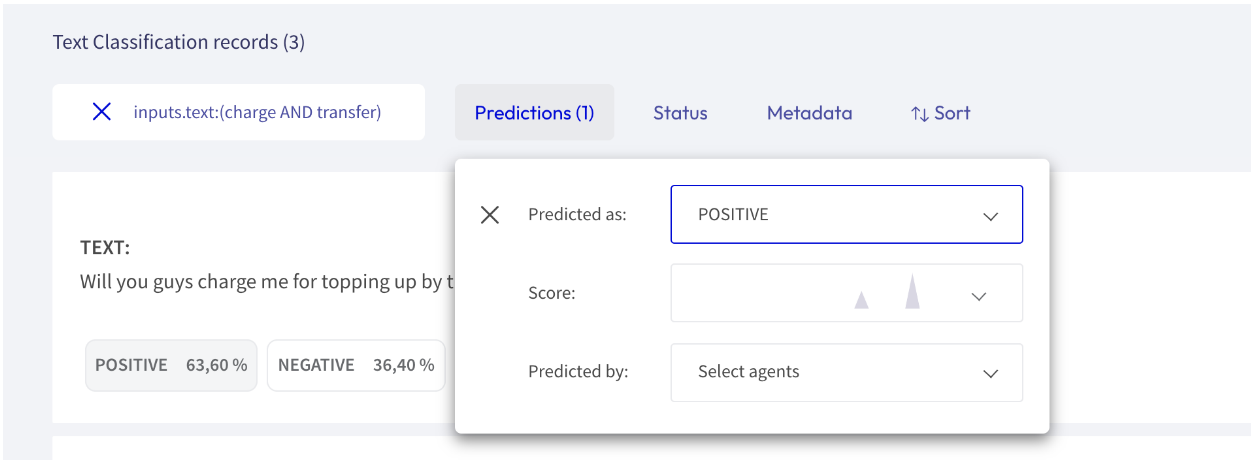
   `-inputs.text:(NOT women)` : records NOT containing women in the inputs.text field.

   `-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.



**NOTE**: 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`.

In both **Annotation** and **Exploration** modes, the search bar is placed in the upper left-hand corner. To search something, users must type one or several words (or a query) and click the **Intro** button.

Note that this feature also works as a kind of filter. If users search something, it is possible to explore and/or annotate the results obtained. *Filters* can be applied.

### Elasticsearch fields

Shown below is a summary of available fields that 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 | Text classification fields | Token classification fields |
|---|---|---|
| Annotated_as | inputs.* | tokens |
| Annotated_by | score | |
| event_timestamp | | |
| id | | |
| last_updated | | |
| metadata.* | | |
| multi_label | | |
| predicted | | |
| predicted_as | | |
| predicted_by | | |
| status | | |
| words | | |
| words.extended | | |

With this component, users are able to search specific information on the dataset, either by **full-text queries** or by queries using **Elasticsearch**.

### Filter Records

With this component, users are able to sort the information on a dataset by using different parameters. Filters are different for each task, and they can be used in both **Annotation** and **Exploration** modes.

#### How filters work

*To see a description of their components, click* here.

In both modes, filters work in a very similar way:

#### Filtering records in the Annotation Mode

Filtering records can be useful in big datasets, when users need to see and annotate a very specific part of the dataset.

For example, if users are annotating a dataset in a **token classification task** and they need to see how many records are annotated with one or more labels, they can use the **Annotation filter** and choose the desired combination.

#### Filtering records in the Explore Mode

In this case, the use is basically the same, but just for analysis purposes.

Another example would be the following: if in a **text classification task** with available ***score***, users want to sort records to see the highest or lowest loss, the **Sort filter** can be used choosing the **Score** field.

**NOTE**: There are no filters in the **Define label rules**, as this mode works with queries.

### View dataset metrics

The **Dataset Metrics** are part of the **Sidebar** placed on the right side of **Rubrix datasets**. To know more about this component, click *here*.

Rubrix metrics are very convenient in terms of assesing the status of the dataset, and to extract more valuable information.

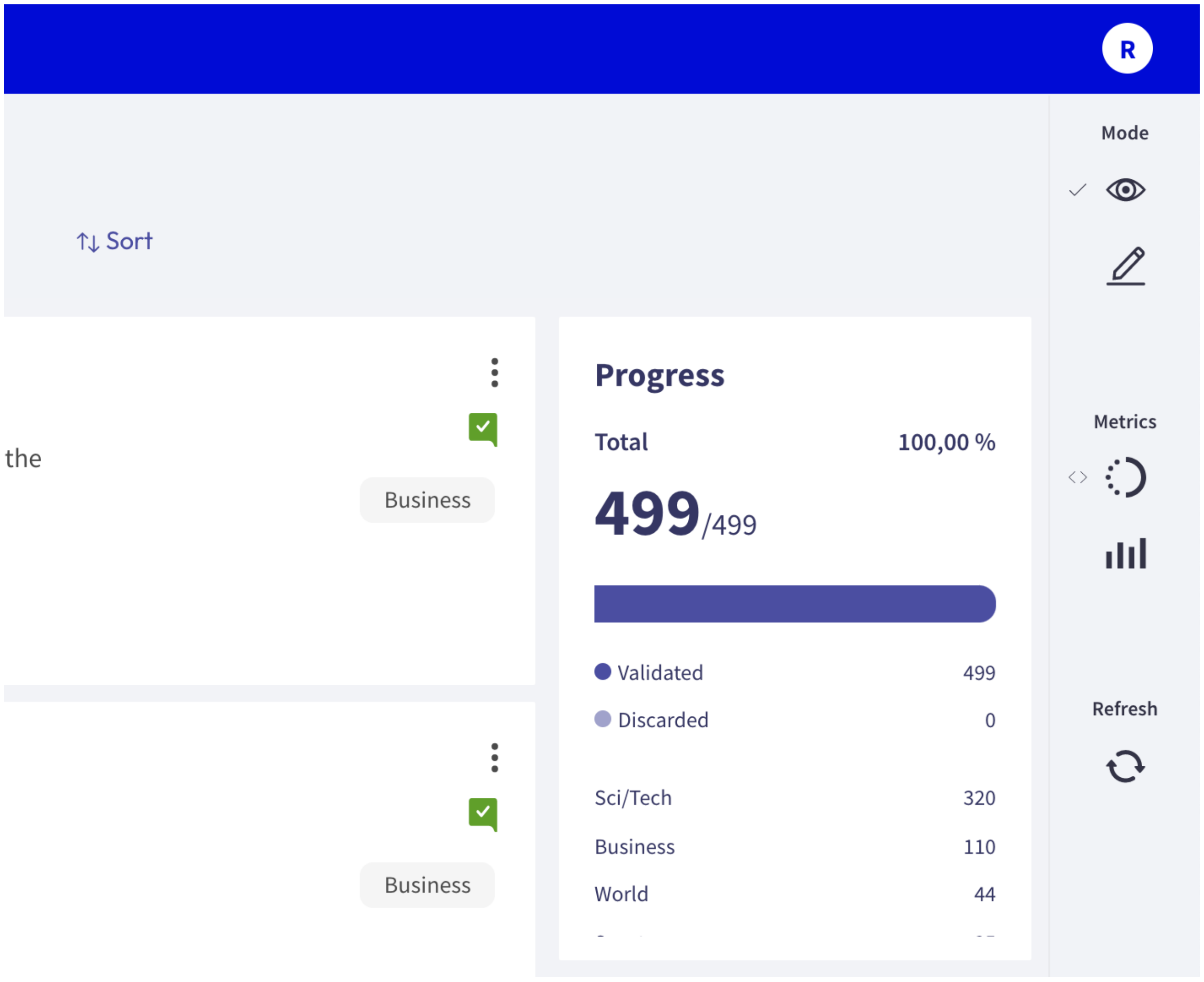
#### How to use Metrics

Metrics are composed of two submenus: **Progress** and **Stats**. They work similarly in **Annotation**, **Explore** and **Define rules** mode.

#### Progress

This submenu is useful when users need to know how many records are annotated, validated and/or discarded.

When clicking on this menu, not only the progress is shown, but also the number of labels and records.



Progress

| | |
|---|---|
| Total | 100,00 % |

499/499

| | |
|---|---|
| ● Validated | 499 |
| ● Discarded | 0 |
| Sci/Tech | 320 |
| Business | 110 |
| World | 44 |

### Stats

This submenu allows users to know more about the keywords and the error distribution of the dataset.

It is composed of a dropdown with two dropdowns:

- The **Keywords** dropdown displays a list of annotated words and the number of occurrences.
- The **Error Distribution** dropdown displays a pie chart with the number of records, correct and incorrect predictions.

Please, note that this section might vary, depending on the task carried out.

### Refresh button

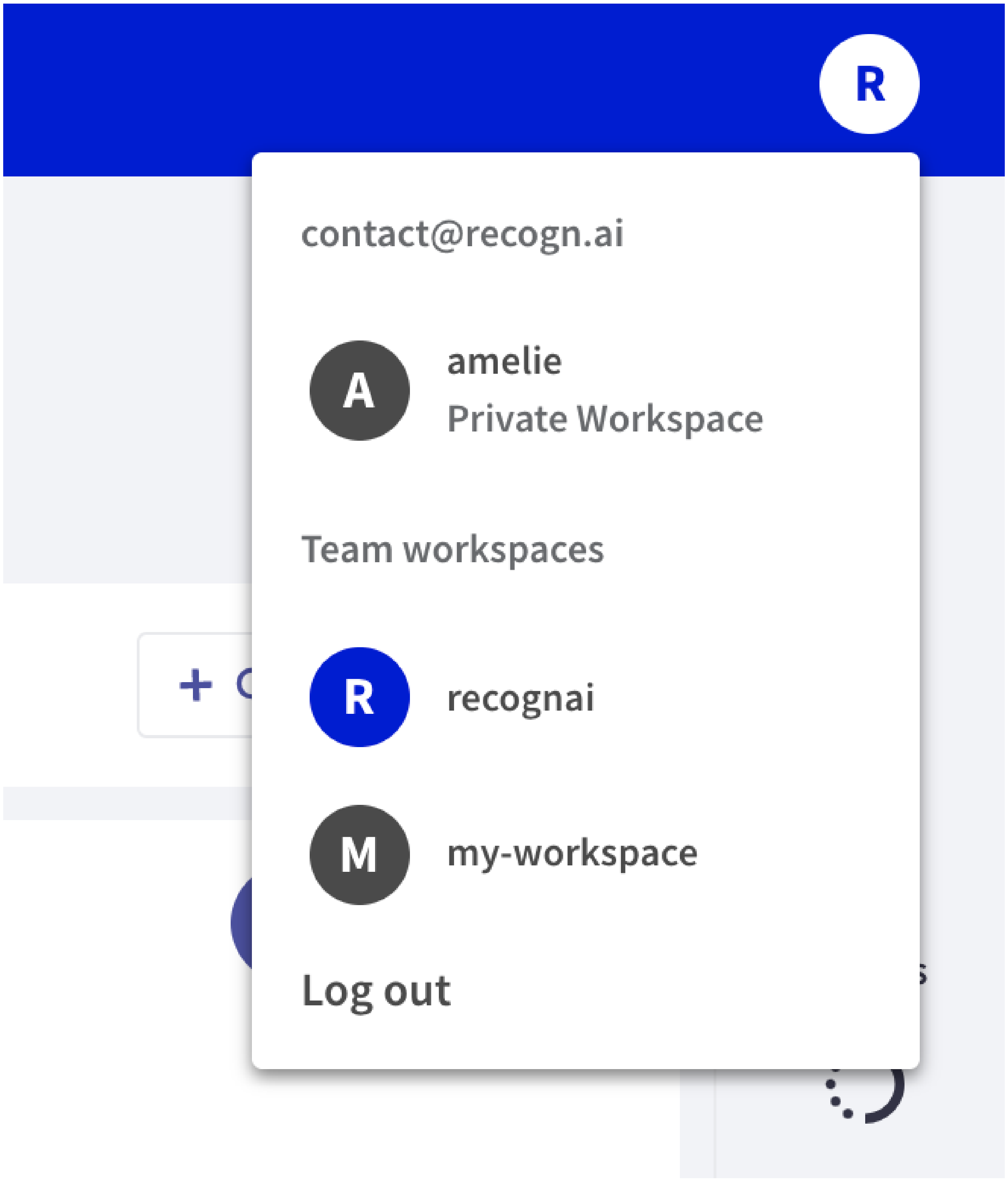
Users should click this button whenever they would like to see the page updated. If changes are made, this button displays the page updated.

### Switch workspaces

This feature turns out very convenient when users need to use different workspaces for different purposes.

The **switch workspace** menu is located in the upper right-hand corner, where the user icon is shown. It displays the different workspaces available for the user.

In order to switch workspaces, users must click on the user icon and select another one. This will drive the user to the main page of the **Workspace**.


R
contact@recogn.ai
A
amelie
Private Workspace
Team workspaces
R
recognai
M
my-workspace
Log out

## 5.21 Developer documentation

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

### 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
```

Install the *commit-msg* hook if you want to check your commit messages in your contributions:

```
pre-commit install --hook-type commit-msg
```

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

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

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

# INDEX