Rubrix

Release 0.4.2.dev0+g98b3e79.d20210921

Recognai

Sep 21, 2021

GETTING STARTED

CHAPTER

ONE

WHAT'S RUBRIX?

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

Key features:

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

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

CHAPTER

TWO

QUICKSTART

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

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

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

And then run:

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

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

```
pip install rubrix==0.4.0 transformers datasets torch
```

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

Why:

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

Ingredients:

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

What are we going to do:

1. Make predictions and log them into a Rubrix dataset.
2. Use the Rubrix web app to explore, filter, and annotate some examples.
3. Load the annotated examples and create a training set, which you can then use to train a supervised classifier.

Use your favourite editor or a Jupyter notebook to run the following:

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

(continues on next page)

(continued from previous page)

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

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

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

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

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

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

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

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

```
# load the Rubrix dataset as a pandas DataFrame
rb_df = rb.load(name='news_zeroshot')

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

# select text input and the annotated label
train_df = pd.DataFrame({
   "text": rb_df.inputs.transform(lambda r: r["text"]),
   "label": rb_df.annotation,
})
```

CHAPTER THREE

USE CASES

- **Model monitoring and observability:** log and observe predictions of live models.
- **Ground-truth data collection**: collect labels to start a project from scratch or from existing live models.
- **Evaluation**: easily compute "live" metrics from models in production, and slice evaluation datasets to test your system under specific conditions.
- **Model debugging**: log predictions during the development process to visually spot issues.
- **Explainability:** log things like token attributions to understand your model predictions.

CHAPTER

FOUR

NEXT STEPS

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

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

CHAPTER

FIVE

COMMUNITY

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

- Github page
- Github forum

5.1 Setup and installation

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

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

5.1.1 1. Install the Rubrix Python client

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

Then you can install Rubrix with `pip`:

```
pip install rubrix==0.4.0
```

5.1.2 2. Launch the web app

There are two ways to launch the webapp:

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

b. Executing the server code manually

a) Using `docker-compose` (recommended)

For this method you first need to install Docker Compose.

Then, create a folder:

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

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

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

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

b) Executing the server code manually

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

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

```
pip install rubrix[server]==0.4.0
```

3. Launch a local instance of the Rubrix web app

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

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

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

5.1.3 3. Start logging data

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

```
import rubrix as rb

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

If you now go to your Rubrix app at http://localhost:6900/ , you will find your first data set. **The default username and password are `rubrix` and `1234`** (see the *user management guide* to configure this). You can also check the REST API docs at http://localhost:6900/api/docs.

Congratulations! You are ready to start working with Rubrix.

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

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

5.1.4 Next steps

To continue learning we recommend you to:

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

5.2 Concepts

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

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

5.2.1 Rubrix data model

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

Dataset

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

Record

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

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

Text classification record

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

Let's see examples of a spam classifier.

```
 record = rb.TextClassificationRecord(
     inputs={
         "text": "Access this link to get free discounts!"
     },
     prediction = [('SPAM', 0.8), ('HAM', 0.2)]
     prediction_agent = "link or reference to agent",

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

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

)
```

Multi-label text classification record

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

```
record = rb.TextClassificationRecord(
     inputs={
         "text": "I can't wait to travel to Egypts and visit the pyramids"
     },
     multi_label = True,

     prediction = [('travel', 0.8), ('history', 0.6), ('economy', 0.3), ('sports', 0.2)],
     prediction_agent = "link or reference to agent",

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

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

)
```

Token classification record

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

```
record = rb.TokenClassificationRecord(
    text = "Michael is a professor at Harvard",
    tokens = token_list,

    # Predictions are a list of tuples with all your token labels and its starting and␣
↪ending positions
    prediction = [('NAME', 0, 7), ('LOC', 26, 33)],
    prediction_agent = "link or reference to agent",

    # Annotations are a list of tuples with all your token labels and its starting and␣
↪ending positions
    annotation = [('NAME', 0, 7), ('ORG', 26, 33)],
    annotation_agent = "link or reference to annotator",

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

Task

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

Annotation

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

Prediction

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

Metadata

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

5.2.2 Methods

To find more information about these methods, please check out the *Python 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 Tasks

This section gives you ideas about the kind of tasks you can use Rubrix for. It also describes some of the tasks on our roadmap, if there's some task you want and don't see here or you want to contribute a task, file an issue or use the Discussion forum at Rubrix's GitHub page.

5.3.1 Supported tasks

Text classification

According to the amazing NLP Progress resource by Seb Ruder:

> Text classification is the task of assigning a sentence or document an appropriate category. The categories depend on the chosen dataset and can range from topics.

Rubrix is flexible with input and output shapes, which means you can model many related tasks like for example:

- Sentiment analysis
- Natural Language Inference
- Semantic Textual Similarity
- Stance detection
- **Multi-label text classification**
- **Node classification in knowledge graphs**.

Token classification

The most well-known task in this category is probably Named Entity Recognition:

> 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.

Rubrix is flexible with input and output shapes, which means you can model related tasks like for example:

- Named entity recognition
- Part of speech tagging
- Slot filling

5.3.2 Tasks on the roadmap

Natural language processing

- Text2Text, covering summarization, machine translation, natural language generation, etc.
- Question answering
- Keyphrase extraction
- Relationship Extraction

Computer vision

- Image classification
- Image captioning

Speech

- Speech2Text

5.4 Advanced setup guides

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

5.4.1 Using docker

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

```
docker pull recognai/rubrix
```

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

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

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

```
docker ps
```

To stop the Rubrix server, just stop the container:

```
docker stop rubrix
```

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

5.4.2 Configure elasticsearch role/users

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

All you need to take into account is:

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

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

You can provide the credentials using the following scheme:

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

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

5.4.3 Deploy to aws instance using docker-machine

Setup an AWS profile

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

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

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

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

```
export AWS_PROFILE=rubrix
```

Create docker machine (aws)

```
docker-machine create --driver amazonec2 \
--amazonec2-root-size 60 \
--amazonec2-instance-type t2.large \
--amazonec2-open-port 80 \
--amazonec2-ami ami-0b541372 \
--amazonec2-region eu-west-1 \
rubrix-aws
```

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

Verify machine creation

```
$>docker-machine ls

NAME                   ACTIVE   DRIVER      STATE     URL                        SWARM  ␣
↪DOCKER     ERRORS
rubrix-aws             -        amazonec2   Running   tcp://52.213.178.33:2376          ␣
↪v20.10.7
```

Save asigned machine ip

In our case, the assigned ip is `52.213.178.33`

Connect to remote docker machine

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

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

Define a docker-compose.yaml

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

services:
  rubrix:
    image: recognai/rubrix:v0.4.0
    ports:
      - "80:80"
    environment:
      ELASTICSEARCH: <elasticsearch-host_and_port>
    restart: unless-stopped
```

Pull image

```
docker-compose pull
```

Launch docker container

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

Accessing Rubrix

In our case http://52.213.178.33

5.4.4 User management

The Rubrix server allows you to manage various users, which helps you to keep track of the annotation agents.

The default user

By default, Rubrix is only configured for the following user:

- username: `rubrix`
- password: `1234`
- api key: `rubrix.apikey`

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>"
```

How to add new users

To configure the Rubrix server for various users, you just need to create a yaml file like the following one:

```
#.users.yaml
# Users are provided as a list
- username: user1
  hashed_password: <generated-hashed-password> # See the previous section above
  api_key: "ThisIsTheUser1APIKEY"
- username: user2
  hashed_password: <generated-hashed-password> # See the previous section above
  api_key: "ThisIsTheUser2APIKEY"
- ...
```

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*.

Then open the provided `docker-compose.yaml` and configure the *rubrix* service in the following way:

```
# docker-compose.yaml
services:
  rubrix:
    image: recognai/rubrix:v0.4.0
    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 and their annotations will be tracked with their usernames.

5.4.5 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 Monitoring and collecting data from third-party apps

This guide will show you **how can Rubrix be integrated into third-party applications** to collect predictions and user feedback. To do this, we are going to use streamlit, an amazing tool to turn Python scripts into beautiful web-apps.

Let's make a quick tour of the app, how you can run it locally and how to integrate Rubrix into other apps.

5.5.1 What does our streamlit app do?

In our streamlit app we are working on a use case of *multilabel text classification*, including the inference process to make predictions and the annotations over those predictions. The NLP model is a zero-shot classifier based on SqueezeBERT, used to predict text categories. These predictions are **mutilabel**, which means that more than one category can be predicted for a given text, thus the sum of the probabilities of all the candidate labels can be greater than 1. For this reasons, we let the user pick a threshold, showing which labels will be included in the prediction when changing its value.

After the threshold is selected, the user can make its own annotation, whether or not she or he thinks the predictions are correct. This is where the *human-in-the-loop* comes into play, by responding to a model made prediction with a user made annotation, that could eventually be used to provide feedback to the model or to make retrainings.

Once the annotated labels are selected, the user can press the **log** button. A `TextClassificationRecord` will be created and logged into Rubrix with all the information about the process: the input text, the prediction and the annotation. This data is also displayed in the streamlit app, as the process ends. But you could always change the input text, the threshold or the annotated labels and log again!

5.5.2 How to run the app

We've created a standalone repository for this streamlit app, for you to clone and play around. To run the app, follow these steps:

1. Install the requirements into a fresh environment (or into your system, but take care with the dependency problems!) with Python 3, via `pip install -r requirements.txt`.
2. Run `streamlit run app.py`.
3. In the response prompt, streamlit will give you the localhost direction where your app will be running. You can now open it in your browser.

5.5.3 Rubrix integration

Rubrix can be used alongside any third-party apps via its REST API or its Python client. In our case, the logging of the record is made when the log button is pressed. In that moment, two lists will be populated:

- `labels`, with the predicted labels by the zero-shot classifier
- `selected_labels`, with the annotated labels, selected by the user.

Then, using the Python client we log instances of `rubrix.TextClassificationRecord` as follows:

```python
import rubrix as rb

item = rb.TextClassificationRecord(
                inputs={"text": text_input},
                prediction=labels,
                prediction_agent="typeform/squeezebert-mnli",
                annotation=selected_labels,
                annotation_agent="streamlit-user",
                multi_label=True,
                event_timestamp=datetime.datetime.now(),
                metadata={"model": "typeform/squeezebert-mnli"}
            )

dataset_name = "multilabel_text_classification"

rb.log(name=dataset_name, records=item)
```

5.6 Rubrix Cookbook

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

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

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

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

5.6.1 Hugging Face Transformers

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

[ ]:

```
%pip install torch
%pip install transformers
%pip install datasets
```

Text Classification

Inference

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

[ ]:

```
import rubrix as rb
from transformers import pipeline

input_text = "I love watching rock climbing competitions!"

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

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

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

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

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

Training

Let's read a Rubrix dataset, prepare a training set and use the Trainer API for fine-tuning a distilbert-base-uncased model. Take into account that a labelled_dataset is expected to be found in your Rubrix client.

[ ]:

```
from datasets import Dataset
import rubrix as rb

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

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

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


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

[ ]:

```
from transformers import AutoModelForSequenceClassification
from transformers import AutoTokenizer
from transformers import Trainer

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

(continues on next page)

(continued from previous page)

```
    "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")
```

5.6.2 spaCy

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

[ ]:

```
%pip install spacy
```

Token Classification

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

NER

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

[ ]:

```
!python -m spacy download fr_core_news_sm
```

[ ]:

```
import rubrix as rb
import spacy

input_text = "Paris a un enfant et la for^et a un oiseau ; l’oiseau s’appelle le moineau␣
↪; l’enfant s’appelle le gamin"
```

(continues on next page)

(continued from previous page)

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

# Creating spaCy doc
doc = nlp(input_text)

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

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

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

POS tagging

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

[ ]:

```
import rubrix as rb
import spacy

input_text = "Paris a un enfant et la for^et a un oiseau ; l’oiseau s’appelle le moineau␣
↪; l’enfant s’appelle le gamin"

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

# Creating spaCy doc
doc = nlp(input_text)

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

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

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

5.6.3 Flair

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

[ ]:

```
%pip install flair
```

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

[ ]:

```
import nltk
import ssl

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

nltk.download()
```

Text Classification

Zero-shot and Few-shot classifiers

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

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

[ ]:

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

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

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

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

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


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

(continues on next page)

(continued from previous page)

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

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

Custom and pre-trained classifiers

Let's see an example with Deutch offensive language model.

[ ]:

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

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

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

# Creating Sentence object
sentence = Sentence(input_text)

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

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

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

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

Token Classification

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

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

NER

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

[ ]:

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

input_text = "De Nachtwacht is in het Rijksmuseum"

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

# Creating Sentence object
sentence = Sentence(input_text)

# run NER over sentence
tagger.predict(sentence)

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

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

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

POS tagging

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

[ ]:

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

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

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

# Creating Sentence object
sentence = Sentence(input_text)

# run NER over sentence
tagger.predict(sentence)

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

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

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

5.6.4 Stanza

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

[ ]:

```
%pip install stanza
```

Text Classification

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

[ ]:

```
import rubrix as rb
import stanza

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

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

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

# Analizing the input text
doc = nlp(input_text)

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


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

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

# Building a TextClassificationRecord
record = rb.TextClassificationRecord(
    inputs=input_text,
    prediction=entities,
    prediction_agent="stanza/en",
)

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

Token Classification

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

POS tagging

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

[ ]:

```
import rubrix as rb
import stanza

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

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

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

# Analizing the input text
doc = nlp(input_text)

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

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

# 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")
```

5.7 Tasks Templates

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

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

5.7.1 Text Classification

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

Text Categorization

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

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

[ ]:

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

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

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

records = []

for record in dataset:

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

    # 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)

```
# 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.

[ ]:

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

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

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

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

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)

```
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.

5.7.2 Token Classification

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

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

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

NER

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

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

[ ]:

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

[ ]:

```
import rubrix as rb
import spacy
from datasets import load_dataset

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

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

records = []

for record in dataset:

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

    # spaCy Doc creation
    doc = nlp(text)

    # 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)

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

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

trainer.fit()
```

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

[ ]:

```
dataset = Dataset.from_json("test.json")

records = []

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

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

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

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

    # Rubrix TokenClassificationRecord list
    records.append(
        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",
    },
)
```

5.7.3 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")
```

5.8 Explore data and predictions with `datasets` and `transformers`

In this tutorial, we will walk through the process of using Rubrix to explore NLP datasets in combination with the amazing `datasets` and `transformer` libraries from Hugging Face.

5.8.1 Introduction

Our goal is to show you how to store and explore NLP datasets using Rubrix for use cases like training data management or model evaluation and debugging.

The tutorial is organized into three parts:

1. **Storing and exploring text classification data**: We will use the `datasets` library and Rubrix to store text classification datasets.
2. **Storing and exploring token classification data**: We will use the `datasets` library and Rubrix to store token classification data.
3. **Exploring predictions**: We will use a pretrained `transformers` model and store its predictions into Rubrix to explore and evaluate our pretrained model.

5.8.2 Install tutorial dependencies

In this tutorial we will be using `transformers` and `datasets` libraries. If you do not have them installed, run:

[ ]:
```
%pip install torch -qqq
%pip install transformers -qqq
%pip install datasets -qqq
%pip install tdqm -qqq # for progress bars
```

5.8.3 Setup Rubrix

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

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

5.8.4 1. Storing and exploring text classification training data

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

With Rubrix you can track both training data and predictions from models. In this part, we will focus only on training data. Typically, training data is data which has been curated or annotated by a human. Other terms for this same concept are: ground-truth data, “gold-standard” data, or even “annotated” data.

In this part of the tutorial, you will learn how to use datasets library for quick exploration of `Text Classification` and `Token Classification` training data. This is useful during model development, for getting a sense of the data, identifying potential issues, debugging, etc. Here we will use rather static “research”datasets but Rubrix really shines when you are collecting and using training data in the wild, or in other words in real data science projects.

Let’s get started!

Text classification with the `tweet_eval` dataset (Emoji 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 labeled examples and see how it starts predicting over the very same labels.

In this first case, we are going to play with `tweet_eval`, a dataset with a bunch of tweets from different authors and topics and the sentiment it transmits. This is, in fact, a very common NLP task called Sentiment Analysis, but with a cool tweak: we are representing these sentiments with emojis. Each tweet comes with a number between 0 and 19, which represents different emojis. You can see each one in a cell below or in the tweet_eval site at Hub.

First of all, we are going to load the dataset from Hub and visualize its content.

[ ]:

```
from datasets import load_dataset

dataset = load_dataset("tweet_eval", 'emoji', script_version="master")
```

[ ]:

```
labels = dataset['train'].features['label'].names; labels
```

Usually, datasets are divided into train, validation and test splits, and each one of them is used in a certain part of the training. For now, we can stick to the training split, which usually contains the majority of the instances of a dataset. Let’s see what’s inside!

[ ]:

```
with dataset['train'].formatted_as("pandas"):
    print(dataset['train'][:5])
```

Now, we are going to create our records from this dataset and log them into Rubrix. Rubrix comes with `TextClassificationRecord` and `TokenClassificationRecord` classes, which can be created from a dictionary. These objects pass information to Rubrix about the input of the model, the predictions obtained and the annotations made, as well as a metadata field for other important details.

In our case, we haven’t predicted anything, so we are only going to include the labels of each instance as annotations, as we know they are the ground truth. We will also include each tweet into inputs, and specify in the metadata section that we are into the training split. Once `records` is populated, we can log it with `rubric.log()`, specifying the name of our dataset.

[ ]:

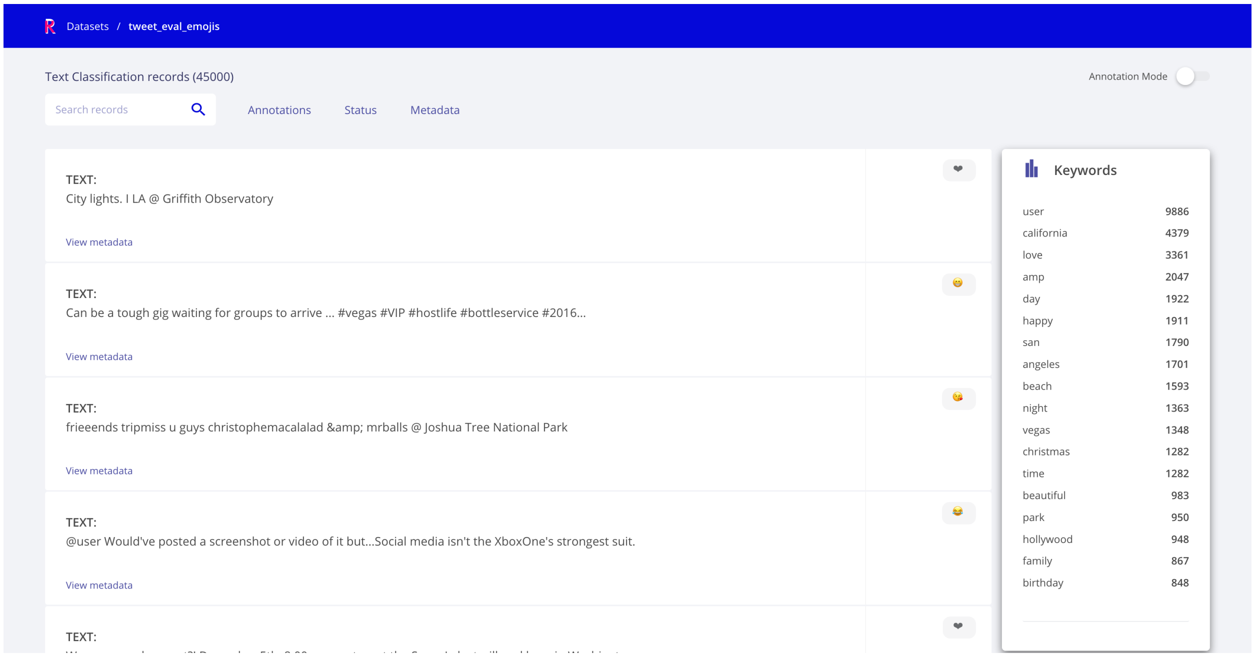
```
records = []

for record in dataset['train']:
    records.append(rb.TextClassificationRecord(
```

(continues on next page)

(continued from previous page)

```
        inputs=record["text"],
        annotation=labels[record["label"]],
        annotation_agent="https://huggingface.co/datasets/tweet_eval",
        metadata={"split": "train"},
        )
    )
```

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



TEXT:
City lights. I LA @ Griffith Observatory

View metadata

TEXT:
Can be a tough gig waiting for groups to arrive ... #vegas #VIP #hostlife #bottleservice #2016...

View metadata

TEXT:
frieeends tripmiss u guys christophemacalalad &amp; mrballs @ Joshua Tree National Park

View metadata

TEXT:
@user Would've posted a screenshot or video of it but...Social media isn't the XboxOne's strongest suit.

View metadata

TEXT:

user	9886
california	4379
love	3361
amp	2047
day	1922
happy	1911
san	1790
angeles	1701
beach	1593
night	1363
vegas	1348
christmas	1282
time	1282
beautiful	983
park	950
hollywood	948
family	867
birthday	848

Thanks to our metadata section in the Text Classification Record, we can log tweets from the validation and test splits in the same dataset to explore them using the Metadata filters.

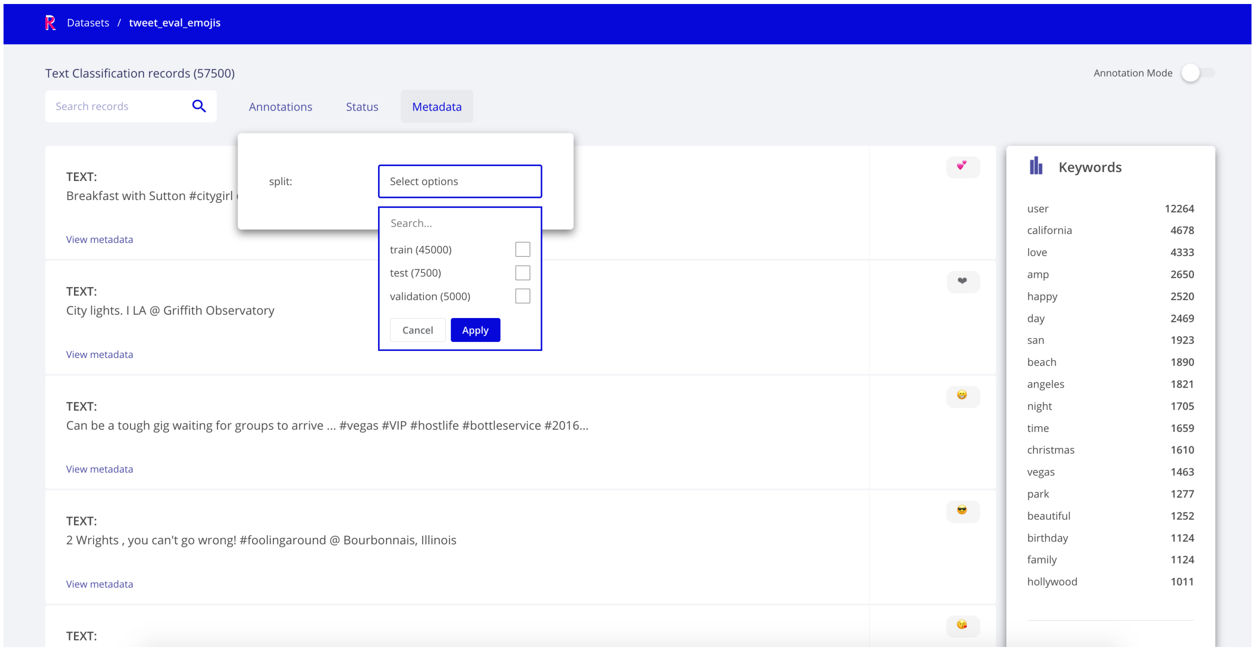
[ ]:
```
records_validation = []

for record in dataset['validation']:
    records_validation.append(rb.TextClassificationRecord(
        inputs=record["text"],
        annotation=labels[record["label"]],
        annotation_agent="https://huggingface.co/datasets/tweet_eval",
        metadata={"split": "validation"},
        )
    )

rb.log(records=records_validation, name="tweet_eval_emojis")
```

[ ]:
```
records_test = []

for record in dataset['test']:
    records_test.append(rb.TextClassificationRecord(
        inputs=record["text"],
        annotation=labels[record["label"]],
```

(continues on next page)

(continued from previous page)

```
        annotation_agent="https://huggingface.co/datasets/tweet_eval",
        metadata={"split": "test"},
        )
    )

rb.log(records=records_test, name="tweet_eval_emojis")
```



Text Classification records (57500)

train (45000) □
test (7500) □
validation (5000) □

TEXT:
Breakfast with Sutton #citygirl (

💕

TEXT:
City lights. I LA @ Griffith Observatory

❤

TEXT:
Can be a tough gig waiting for groups to arrive ... #vegas #VIP #hostlife #bottleservice #2016...

😁

TEXT:
2 Wrights , you can't go wrong! #foolingaround @ Bourbonnais, Illinois

😎

TEXT:

😘

Keywords

user	12264
california	4678
love	4333
amp	2650
happy	2520
day	2469
san	1923
beach	1890
angeles	1821
night	1705
time	1659
christmas	1610
vegas	1463
park	1277
beautiful	1252
birthday	1124
family	1124
hollywood	1011

Natural language inference with the `MRPC` dataset

Natural Language Inference (NLI) is also a very common NLP task, but a little different to regular Text Classification. In NLI, the model receives a premise and a hypothesis, and it must figure out if the premise hypothesis is true or not given the premise. We have three categories: entailment (true), contradiction (false) or neutral (undetermined or unrelated). With the premise *"We live in a flat planet called Earth"*, the hypothesis *"The Earth is flat"* must be classified as entailment, as it is stated in the premise. NLI works with a sort of close-world assumption, in that everything not defined in the premise cannot be inferred from the real world.

Another key difference from Text Classification is that the input come in pairs of two sentences or texts, not only one. Text Classification treats its input as a cohesive and correlated unit, while NLI treats its input as a pair and tries to find correlation.

To play around with NLI we are going to use Hub GLUE benchmark over the MRPC task. GLUE is a well-known benchmark resource for NLP, and allow us to use its data directly over the Microsoft Research Paraphrase Corpus, a corpus of online news.

[ ]:
```
from datasets import load_dataset
dataset = load_dataset('glue', 'mrpc', split='train')
```

[ ]:
```
dataset[0]
```

We can see the two input sentences instead of one. In order to simplify the workflow, let's just test if they are equivalent or not.

```
[ ]: labels = dataset.features['label'].names ; labels
```

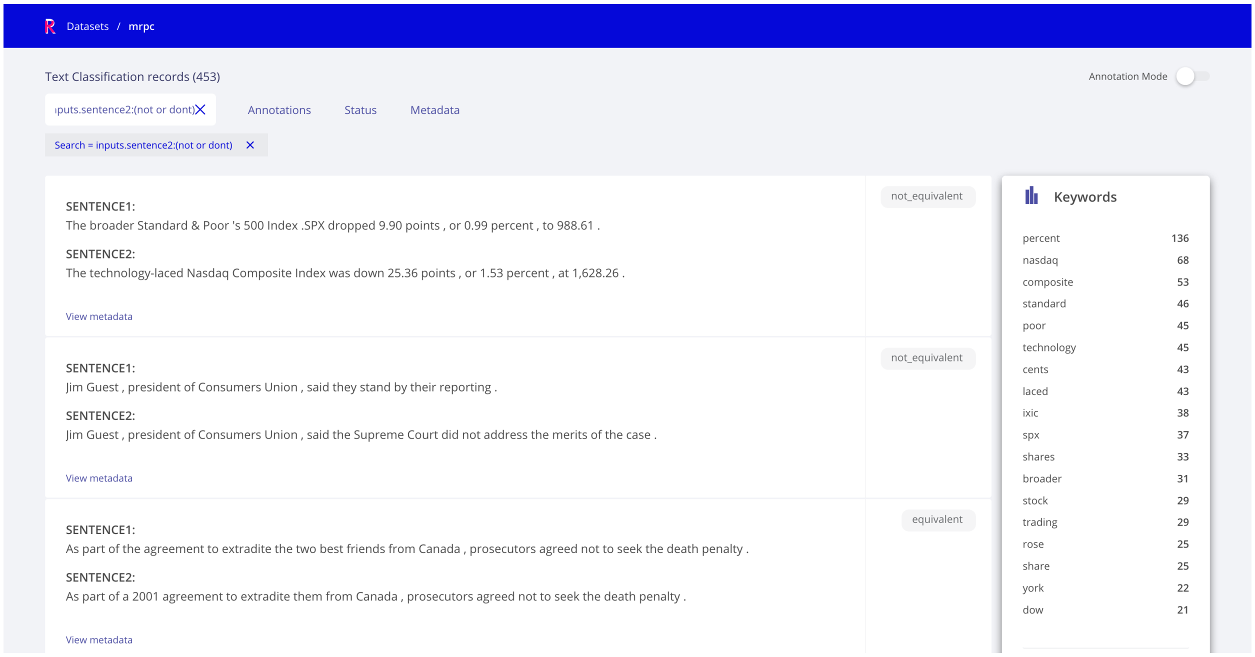
Populating our record list follows the same procedure as in Text Classification, adapting our input to the new scenario of pairs.

```
[ ]: records=[]

     for record in dataset:
         records.append(rb.TextClassificationRecord(
            inputs={
                "sentence1": record["sentence1"],
                "sentence2": record["sentence2"]
             },
             annotation=labels[record["label"]],
             annotation_agent="https://huggingface.co/datasets/glue#mrpc",
             metadata={"split": "train"},
             )
         )
```

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

Once your dataset is logged you can explore it using filters, keyword-based search and with Elasticsearch's query string DSL.

For example, the following query `inputs.sentence2:(not or dont)` lets you browse all examples containing `not` or `dont` inside the sentence2 field, which you can further filter by `Annotated as` to browse examples belonging to a specific category (e.g., `not_equivalent`)



Text Classification records (453)

ıputs.sentence2:(not or dont)

Search = inputs.sentence2:(not or dont)

SENTENCE1:
The broader Standard & Poor 's 500 Index .SPX dropped 9.90 points , or 0.99 percent , to 988.61 .

SENTENCE2:
The technology-laced Nasdaq Composite Index was down 25.36 points , or 1.53 percent , at 1,628.26 .

not_equivalent

SENTENCE1:
Jim Guest , president of Consumers Union , said they stand by their reporting .

SENTENCE2:
Jim Guest , president of Consumers Union , said the Supreme Court did not address the merits of the case .

not_equivalent

SENTENCE1:
As part of the agreement to extradite the two best friends from Canada , prosecutors agreed not to seek the death penalty .

SENTENCE2:
As part of a 2001 agreement to extradite them from Canada , prosecutors agreed not to seek the death penalty .

equivalent

Keywords

percent	136
nasdaq	68
composite	53
standard	46
poor	45
technology	45
cents	43
laced	43
ixic	38
spx	37
shares	33
broader	31
stock	29
trading	29
rose	25
share	25
york	22
dow	21

Multilabel text classification with go_emotions dataset

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

In Text Classification, the category with the highest score (which our model predicted) is going to be the category predicted, but in this task we need to establish a threshold, a value between 0 and 1, from which we will classify the labels as predictions or not. If we set it to 0.5, only categories with more than a 0.5 probability value will be considered predictions.

To get used to this task and see how we can log data to Rubrix, we are going to use Hub go_emotions dataset, with comments from different reddit forums and an associated sentiment (this experiment would also be considered Sentiment Analysis).

```
[ ]: from datasets import load_dataset

     dataset = load_dataset('go_emotions', split='train[0:10]')
```

Here's an example of an instance of the datasets, and the different labels, ordered. Each label will be represented in the dataset as a number, but we will translate to its name before logging to Rubrix, to see things more clearly.

```
[ ]: dataset[0]
```

```
[ ]: labels = dataset.features['labels'].feature.names; labels
```

Now, instead of a simple string we pass on a list of strings to the `annotation` argument of our record.

```
[ ]: records= []

     for record in dataset:
         records.append(rb.TextClassificationRecord(
             inputs=record["text"],
             annotation=[labels[cls] for cls in record['labels']],
             annotation_agent="https://huggingface.co/datasets/go_emotions",
             multi_label=True,
             metadata={
                 "split": "train"
                 },
             )
         )
```

And logging is just as easy as before!

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

5.8.5 2. Storing and exploring token classification training data

In this second part, we will cover Token Classification while still using the datasets library. This NLP task aims at dividing 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.

We are going to cover a few cases using datasets, and see how `TokenClassificationRecord` allows us to log data in Rubrix in a similar fashion.

Named-Entity Recognition with `wnut17` dataset

Named-Entity Recognition (NER) seeks to locate and classify named entities mentioned in unstructured text into predefined categories. And, what's powerful about NER is that this predefined categories can be whatever we want. Maybe grammatical categories, and be the best at syntax analysis in our English class, maybe person names, or organizations, or even medical codes.

For this case, we are going to use Hub WNUT 17 dataset, about rare entities on written text. Take for example the tweet "so.. kktny in 30 mins?" - even human experts find the entity kktny hard to detect and resolve. This task will evaluate the ability to detect and classify novel, emerging, singleton named entities in written text.

As always, let's first dive into the data and see how it looks like.

```
[ ]: from datasets import load_dataset

     dataset = load_dataset("wnut_17", split="train[0:10]")
```

```
[ ]: dataset[0]
```

We can see a list of tags and the tokens they are referring to. We have the following rare entities in this example.

```
[ ]: for entity, token in zip(dataset[0]["ner_tags"], dataset[0]["tokens"]):
         if entity != 0:
             print(f"""{token}: {dataset.features["ner_tags"].feature.names[entity]}""")
```

So, it makes a lot of sense to translate these tags into NER tags, which are much more self-explanatory than an integer.

```
[ ]: dataset = dataset.map(lambda instance: {"ner_tags_translated": [dataset.features["ner_
     ↪tags"].feature.names[tag] for tag in instance["ner_tags"]]})
```

What we did is a mapping function over dataset, which allow us to make changes in every instance of the dataset. The very same instance that we printed before is much more readable now.

```
[ ]: dataset[0]
```

Info about the meaning of the tags is available here, but to sum up, *Empire* and *ESB* has been classified as **B-LOC**, or beginning of a location name, *State* and *Building* has been classified as **I-LOC** or intermediate/final of a location name.

We need to transform a bit this information, providing an entity annotation. Entity annotations are simply tuples, with the following structure

```
(label, start_position, end_position)
```

Let's create a function that transform our dataset records into entities. It's a bit weird, but don't worry! What's doing inside is getting the entities' information as shown above.

```
[ ]: def parse_entities(record):

         entities, text, nr_tokens = [], " ".join(record["tokens"]), len(record["tokens"])
         token_start_indexes = [text.rfind(substr) for substr in [" ".join(record["tokens"][i:
     ↪]) for i in range(nr_tokens)]]

         entity = None
         for i, tag, start in zip(range(nr_tokens), record["ner_tags_translated"], token_
     ↪start_indexes):
             # end of entity
             if entity is not None and (not tag.startswith("I-") or i == nr_tokens -1):
                 entity += (start-1,)
                 entities.append(entity)
                 entity = None
             # start new entity
             if entity is None and tag.startswith("B-"):
                 entity = (tag[2:], start)

         return entities
```

Let's proceed and create a record list to log it

```
[ ]: records = []

     for record in dataset:
         entities = parse_entities(record)
         records.append(rb.TokenClassificationRecord(
             text=" ".join(record["tokens"]),
             tokens=record["tokens"],
             annotation=entities,
             annotation_agent="https://huggingface.co/datasets/wnut_17",
             metadata={
                 "split": "train"
                 },
             )
         )
```

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

```
[ ]: rb.log(records=records, name="ner_wnut_17")
```

Part of speech tagging with `conll2003` dataset

Another NLP task related to token-level classification is Part-of-Speech tagging (POS tagging). In this task we will identify names, verbs, adverbs, adjectives…based on the context and the meaning of the words. It is a little trickier than having a huge dictionary where we can look up that *drink* is a verb and *dog* is a name. Many words change its grammatical type according to the context of the sentence, and here is where AI comes to save the day.

With just our dictionary and a regular script, *dog* in `The sailor dogs the hatch.` would be classified as a name, because *dog* is a name, right? A trained NLP model would step up and say *No! That is a very common example to illustrate the ambiguity of words. It is a verb!*. Or maybe it would just say *verb*. That's up to you.

In this dataset from hub, we will see how different sentence has POS and NER tags, and how we can log this POS tag information into Rubrix.

[ ]:
```
from datasets import load_dataset

dataset = load_dataset("conll2003", split="train[0:10]")
```

[ ]:
```
dataset[0]
```

Each POS and NER tag are represented by a number. In `dataset.features` we can see to which tag they refer (this link may serve you to look up the meaning).

[ ]:
```
dataset.features
```

The following function will help us create the entities.

[ ]:
```
def parse_entities_POS(record):

    entities = []
    counter = 0

    for i in range(len(record['pos_tags'])):

        entity = (dataset.features["pos_tags"].feature.names[record["pos_tags"][i]],␣
↪counter, counter + len(record["tokens"][i]))
        entities.append(entity)

        counter += len(record["tokens"][i]) + 1

    return entities
```

[ ]:
```
records = []

for record in dataset:
    entities = parse_entities_POS(record)
    records.append(rb.TokenClassificationRecord(
        text=" ".join(record["tokens"]),
        tokens=record["tokens"],
        annotation=entities,
        annotation_agent="https://huggingface.co/datasets/conll2003",
        metadata={
            "split": "train"
            },
        )
    )
```

[ ]:
```
rb.log(records=records, name="conll2003")
```

And so it is done! We have logged data from 5 different type of experiments, which now can be visualized in Rubrix UI

5.8.6 3. Exploring predictions

In this third part of the tutorial we are going to focus on loading predictions and annotations into Rubrix and visualize them from the UI.

Rubrix let us play with the data in many different ways: visualizing by predicted class, by annotated class, by split, selecting which ones were wrongly classified, etc.

Agnews and zeroshot classification

To explore some logged data on Rubrix UI, we are going to predict the topic of some news with a zero-shot classifier (that we don't need to train), and compare the predicted category with the ground truth. The dataset we are going to use in this part is ag_news, with information of over 1 million articles written in English.

First of all, as always, we are going to load the dataset from Hub and visualize its content.

[ ]:
```
from datasets import load_dataset

dataset = load_dataset("ag_news", split='test[0:100]') # 20% is over 1500 records
```

[ ]:
```
dataset[0]
```

[ ]:
```
dataset.features
```

This dataset has articles from four different classes, so we can define a category list, which may come in handy.

[ ]:
```
categories = ['World', 'Sports', 'Business', 'Sci/Tech']
```

Now, it's time to load our zero-shot classification model. We present two options:

1. DistilBart-MNLI
2. squeezebert-mnli

With the first model, the obtained results are probably going to be better, but it is a larger model, which could take longer to use. We are going to stick with the first one, but feel free to change it, and even to compare them!

[ ]:
```
from transformers import pipeline

model = "valhalla/distilbart-mnli-12-1"

pl = pipeline('zero-shot-classification', model=model)
```

Let's try to make a quick prediction and take a look.

[ ]:
```
pl(dataset[0]['text'], ['World', 'Sports', 'Business', 'Sci/Tech'], hypothesis_template=
↪'This example is {}.',multi_label=False)
```

Knowing how to make a prediction, we can now apply this to the whole selected dataset. Here, we also present you with two options:

1. Traverse through all records in the dataset, predict each record and log it to Rubrix.
2. Apply a map function to make the predictions and add that field to each record, and then log it as a whole to Rubrix.

In the following categories, each approach is presented. You choose what you like the most, or even both (be careful with the time and the duplicated records, though!).

First approach

[ ]:

```
from tqdm import tqdm

for record in tqdm(dataset):

    # Make the prediction
    model_output = pl(record['text'], categories, hypothesis_template='This example is {}
↪.')

    item = rb.TextClassificationRecord(
        inputs=record["text"],
        prediction=list(zip(model_output['labels'], model_output['scores'])),
        prediction_agent="https://huggingface.co/valhalla/distilbart-mnli-12-1",
        annotation=categories[record["label"]],
        annotation_agent="https://huggingface.co/datasets/ag_news",
        multi_label=True,
        metadata={
            "split": "train"
            },
        )


    # Log to rubrix
    rb.log(records=item, name="ag_news")
```

Second approach

[ ]:

```
def add_predictions(records):

    predictions = pl([record for record in records['text']], categories, hypothesis_
↪template='This example is {}.')


    if isinstance(predictions, list):
        return {"labels_predicted": [pred["labels"] for pred in predictions],
↪"probabilities_predicted": [pred["scores"] for pred in predictions]}
    else:
        return {"labels_predicted": predictions["labels"], "probabilities_predicted":␣
↪predictions["scores"]}
```

[ ]:

```
dataset_predicted = dataset.map(add_predictions, batched=True, batch_size=4)
```

[ ]:

```
dataset_predicted[0]
```

[ ]:

```
from tqdm import tqdm

for record in tqdm(dataset_predicted):

    item = rb.TextClassificationRecord(
        inputs=record["text"],
```

(continues on next page)

(continued from previous page)

```
        prediction=list(zip(record['labels_predicted'], record['probabilities_predicted
↪'])),
        prediction_agent="https://huggingface.co/valhalla/distilbart-mnli-12-1",
        annotation=categories[record["label"]],
        annotation_agent="https://huggingface.co/datasets/ag_news",
        multi_label=True,
        metadata={
            "split": "train"
            },
        )

    # Log to rubrix
    rb.log(records=item, name="ag_news")
```

5.8.7 Summary

In this tutorial, we have learnt:

- To log and explore NLP training datasets with the `datasets` library.
- To explore NLP predictions using a `zeroshot classifier` from the `model hub`.

5.8.8 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.

5.9 Explore and analyze spaCy NER pipelines

In this tutorial, you'll learn to log spaCy Name Entity Recognition (NER) predictions.

This is useful for:

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

Let's get started!

5.9.1 Introduction

In this tutorial we will:

- Load the Gutenberg Time dataset from the Hugging Face Hub.
- Use a transformer-based spaCy model for detecting entities in this dataset and log the detected entities into a Rubrix dataset. This dataset can be used for exploring the quality of predictions and for creating a new training set, by correcting, adding and validating entities.
- Use a smaller spaCy model for detecting entities and log the detected entities into the same Rubrix dataset for comparing its predictions with the previous model.
- As a bonus, we will use Rubrix and spaCy on a more challenging dataset: IMDB.

5.9.2 Setup Rubrix

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

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

Once installed, you only need to import Rubrix:

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

5.9.3 Install tutorial dependencies

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

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

5.9.4 Our dataset

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

```
[ ]: from datasets import load_dataset

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

Let's take a look at our dataset!

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

5.9.5 Logging spaCy NER entities into Rubrix

Using a Transformer-based pipeline

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

[ ]:

```
!python -m spacy download en_core_web_trf
```

[ ]:

```
import spacy

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

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

[ ]:

```
records = []
for record in test:
    # We only need the text of each instance
    text = record["tok_context"]

    # spaCy Doc creation
    doc = nlp(text)

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

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

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

[ ]:

```
records[0]
```

[ ]:

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

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

- You can filter records containing specific entity types.
- You can see the most frequent "mentions" or surface forms for each entity. Mentions are the string values of specific entity types, such as for example "1 month" can be the mention of a duration entity. This is useful for error analysis, to quickly see potential issues and problematic entity types.

- You can use the free-text search to find records containing specific words.
- You could validate, include or reject specific entity annotations to build a new training set.

Using a smaller but more efficient pipeline

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

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

```
[ ]: import spacy

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

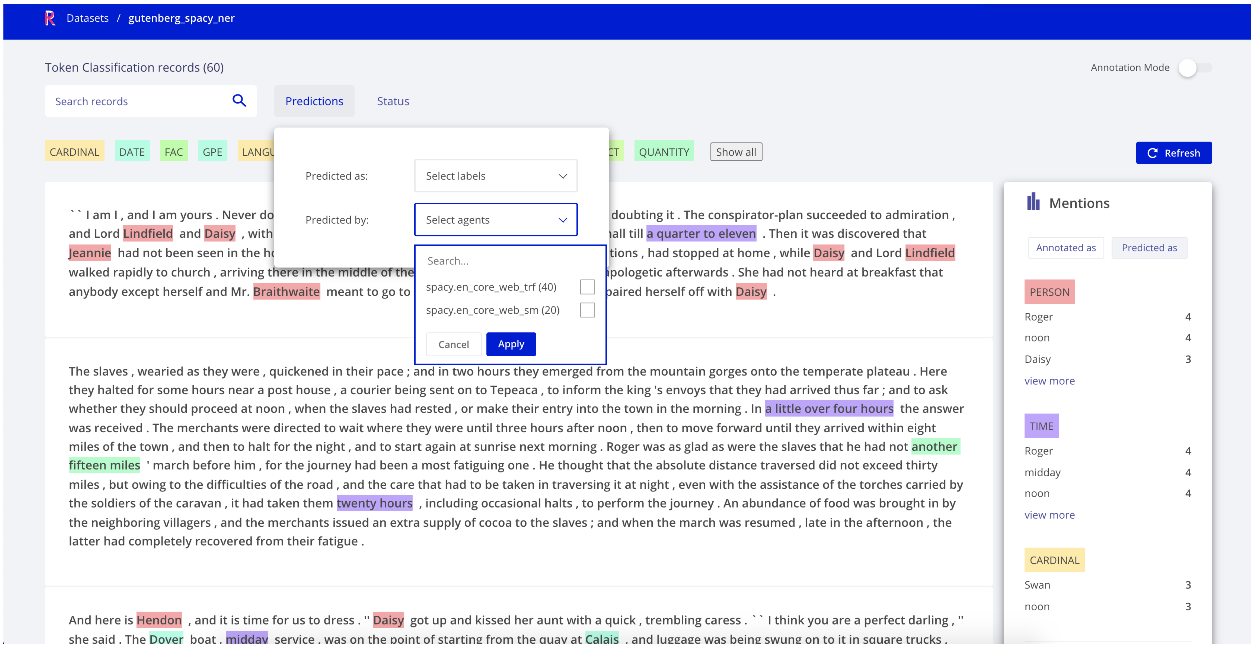
```
[ ]: records = []    # Creating and empty record list to save all the records

     for record in test:

         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")
```

5.9.6 Exploring and comparing `en_core_web_sm` and `en_core_web_trf` models

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

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



`` I am I , and I am yours . Never do … doubting it . The conspirator-plan succeeded to admiration ,
and Lord Lindfield and Daisy , with … all till a quarter to eleven . Then it was discovered that
Jeannie had not been seen in the ho … tions , had stopped at home , while Daisy and Lord Lindfield
walked rapidly to church , arriving there in the middle of the … apologetic afterwards . She had not heard at breakfast that
anybody except herself and Mr. Braithwaite meant to go to … paired herself off with Daisy .

The slaves , wearied as they were , quickened in their pace ; and in two hours they emerged from the mountain gorges onto the temperate plateau . Here they halted for some hours near a post house , a courier being sent on to Tepeaca , to inform the king 's envoys that they had arrived thus far ; and to ask whether they should proceed at noon , when the slaves had rested , or make their entry into the town in the morning . In a little over four hours the answer was received . The merchants were directed to wait where they were until three hours after noon , then to move forward until they arrived within eight miles of the town , and then to halt for the night , and to start again at sunrise next morning . Roger was as glad as were the slaves that he had not another fifteen miles ' march before him , for the journey had been a most fatiguing one . He thought that the absolute distance traversed did not exceed thirty miles , but owing to the difficulties of the road , and the care that had to be taken in traversing it at night , even with the assistance of the torches carried by the soldiers of the caravan , it had taken them twenty hours , including occasional halts , to perform the journey . An abundance of food was brought in by the neighboring villagers , and the merchants issued an extra supply of cocoa to the slaves ; and when the march was resumed , late in the afternoon , the latter had completely recovered from their fatigue .

And here is Hendon , and it is time for us to dress . '' Daisy got up and kissed her aunt with a quick , trembling caress . `` I think you are a perfect darling , ''
she said . The Dover boat , midday service , was on the point of starting from the quay at Calais , and luggage was being swung on to it in square trucks .

PERSON	
Roger	4
noon	4
Daisy	3
view more	

TIME	
Roger	4
midday	4
noon	4
view more	

CARDINAL	
Swan	3
noon	3

5.9.7 Extra: Explore the IMDB dataset

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

```
[ ]: imdb = load_dataset("imdb", split="test[0:5000]")
```

```
[ ]: records = []
     for record in imdb:
         # We only need the text of each instance
         text = record["text"]

         # spaCy Doc creation
         doc = nlp(text)

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

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

         # Rubrix TokenClassificationRecord list
```

(continues on next page)

(continued from previous page)

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

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

Exploring this dataset highlights the need of fine-tuning for specific domains.

For example, if we check the most frequent mentions for Person, we find two highly frequent missclassified entities: gore (the film genre) and Oscar (the prize). You can check yourself each an every example by using the filters and search-box.

5.9.8 Summary

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

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

5.9.9 Next steps

Rubrix documentation for more guides and tutorials.

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

Rubrix Github repo to stay updated.

5.10 Node classification with `kglab` and PyTorch Geometric

We introduce the application of neural networks on knowledge graphs using `kglab` and `pytorch_geometric`.

Graph Neural networks (GNNs) have gained popularity in a number of practical applications, including knowledge graphs, social networks and recommender systems. In the context of knowledge graphs, GNNs are being used for tasks such as link prediction, node classification or knowledge graph embeddings. Many use cases for these tasks are related to `Automatic Knowledge Base Construction` (AKBC) and completion.

In this tutorial, we will learn to:

- use `kglab` to represent a knowledge graph as a Pytorch Tensor, a suitable structure for working with neural nets
- use the widely known `pytorch_geometric` (PyG) GNN library together with `kglab`.
- train a GNN with `pytorch_geometric` and `PyTorch Lightning` for semi-supervised node classification of the recipes knowledge graph.

- build and iterate on training data using `rubrix` with a Human-in-the-loop (HITL) approach.

5.10.1 Our use case in a nutshell

Our goal in this notebook will be to build a semi-supervised node classifier of recipes and ingredients from scratch using kglab, PyG and Rubrix.

Our classifier will be able to classify the nodes in our 15K nodes knowledge graph according to a set of pre-defined flavour related categories: `sweet`, `salty`, `piquant`, `sour`, etc. To account for mixed flavours (e.g., sweet chili sauce), our model will be multi-class (we have several target labels), multi-label (a node can be labelled as with 0 or several categories).

5.10.2 Install `kglab` and `Pytorch Geometric`

```
[ ]: %pip install torch-scatter -f https://pytorch-geometric.com/whl/torch-1.8.0+cpu.html -qqq
     %pip install torch-sparse -f https://pytorch-geometric.com/whl/torch-1.8.0+cpu.html -qqq
     %pip install torch-cluster -f https://pytorch-geometric.com/whl/torch-1.8.0+cpu.html -qqq
     %pip install torch-spline-conv -f https://pytorch-geometric.com/whl/torch-1.8.0+cpu.html␣
     ↪-qqq
     %pip install torch-geometric -qqq
     %pip install torch==1.8.0 -qqq

     %pip install kglab -qqq

     %pip install pytorch_lightning -qqq
```

5.10.3 1. Loading and exploring the recipes knowledge graph

We'll be working with the "recipes" knowledge graph, which is used throughout the `kglab` tutorial (see the Syllabus).

This version of the recipes kg contains around ~15K recipes linked to their respective ingredients, as well as some other properties such as cooking time, labels and descriptions.

Let's load the knowledge graph into a `kg` object by reading from an RDF file (in Turtle):

```
[ ]: import kglab

     NAMESPACES = {
         "wtm":  "http://purl.org/heals/food/",
         "ind":  "http://purl.org/heals/ingredient/",
         "recipe":  "https://www.food.com/recipe/",
         }

     kg = kglab.KnowledgeGraph(namespaces = NAMESPACES)

     _ = kg.load_rdf("data/recipe_lg.ttl")
```

Let's take a look at our graph structure using the `Measure` class:

```
[ ]: measure = kglab.Measure()
     measure.measure_graph(kg)
```

(continues on next page)

(continued from previous page)

```
f"Nodes: {measure.get_node_count()} ; Edges: {measure.get_edge_count()}"
```

```
[ ]: measure.p_gen.get_tally() # tallies the counts of predicates
```

```
[ ]: measure.s_gen.get_tally() # tallies the counts of predicates
```

```
[ ]: measure.o_gen.get_tally() # tallies the counts of predicates
```

```
[ ]: measure.l_gen.get_tally() # tallies the counts of literals
```

From the above exploration, we can extract some conclusions to guide the next steps:

- We have a limited number of relationships, being `hasIngredient` the most frequent.
- We have rather unique literals for labels and descriptions, but a certain amount of repetition for `hasCookTime`.
- As we would have expected, most frequently referenced objects are ingredients such as `Salt`, `ChikenEgg` and so on.

Now, let's move into preparing our knowledge graph for PyTorch.

5.10.4 2. Representing our knowledge graph as a `PyTorch` Tensor

Let's now represent our `kg` as a `PyTorch` tensor using the `kglab.SubgraphTensor` class.

```
[ ]: sg = kglab.SubgraphTensor(kg)
```

```
[ ]: def to_edge_list(g, sg, excludes):
         def exclude(rel):
             return sg.n3fy(rel) in excludes

         relations = sorted(set(g.predicates()))
         subjects = set(g.subjects())
         objects = set(g.objects())
         nodes = list(subjects.union(objects))

         relations_dict = {rel: i for i, rel in enumerate(list(relations)) if not␣
     ↪exclude(rel)}

         # this offset enables consecutive indices in our final vector
         offset = len(relations_dict.keys())

         nodes_dict = {node: i+offset for i, node in enumerate(nodes)}


         edge_list = []

         for s, p, o in g.triples((None, None, None)):
             if p in relations_dict.keys(): # this means is not excluded
                 src, dst, rel = nodes_dict[s], nodes_dict[o], relations_dict[p]
                 edge_list.append([src, dst, 2 * rel])
```

(continues on next page)

(continued from previous page)

```
            edge_list.append([dst, src, 2 * rel + 1])

    # turn into str keys and concat
    node_vector = [sg.n3fy(node) for node in relations_dict.keys()] + [sg.n3fy(node) for␣
↪node in nodes_dict.keys()]
    return edge_list, node_vector
```

```
[ ]: edge_list, node_vector = to_edge_list(kg.rdf_graph(), sg, excludes=['skos:description',
↪'skos:prefLabel'])
```

```
[ ]: len(edge_list) , edge_list[0:5]
```

Let's create `kglab.Subgraph` to be used for encoding/decoding numerical ids and uris, which will be useful for preparing our training data, as well as making sense of the predictions of our neural net.

```
[ ]: sg = kglab.Subgraph(kg=kg, preload=node_vector)
```

```
[ ]: import torch
     from torch_geometric.data import Data

     tensor = torch.tensor(edge_list, dtype=torch.long).t().contiguous()
     edge_index, edge_type = tensor[:2], tensor[2]
     data = Data(edge_index=edge_index)
     data.edge_type = edge_type
```

```
[ ]: (data.edge_index.shape, data.edge_type.shape, data.edge_type.max())
```

5.10.5 3. Building a training set with Rubrix

Now that we have a tensor representation of our kg which we can feed into our neural network, let's now focus on the training data.

As we will be doing semi-supervised classification, we need to build a training set (i.e., some recipes and ingredients with ground-truth labels).

For this, we can use Rubrix, an open-source tool for exploring, labeling and iterating on data for AI. Rubrix allows data scientists and subject matter experts to rapidly iterate on training and evaluation data by enabling iterative, asynchronous and potentially distributed workflows.

In Rubrix, a very simple workflow during model development looks like this:

1. Log unlabelled data records with `rb.log()` into a Rubrix dataset. At this step you could use weak supervision methods (e.g., Snorkel) to pre-populate and then only refine the suggested labels, or use a pretrained model to guide your annotation process. In our case, we will just log recipe and ingredient "records" along with some metadata (RDF types, labels, etc.).
2. Rapidly explore and label records in your dataset using the webapp which follows a search-driven approach, which is especially useful with large, potentially noisy datasets and for quickly leveraging domain knowledge (e.g., recipes containing WhiteSugar are likely sweet). For the tutorial, we have spent around 30min for labelling around 600 records.
3. Retrieve your annotations any time using `rb.load()`, which return a convenient `pd.Dataframe` making it quite handy to process and use for model development. In our case, we will load a dataset, filter annotated entities, do a train_test_split with scikit_learn, and then use this for training our GNN.

4. After training a model, you can go back to step 1, this time using your model and its predictions, to spot improvements, quickly label other portions of the data, and so on. In our case, as we've started with a very limited training set (~600 examples), we will use our node classifier and `rb.log()` it's predictions over the rest of our data (unlabelled recipes and ingredients).

```
[ ]: LABELS = ['Bitter', 'Meaty', 'Piquant', 'Salty', 'Sour', 'Sweet']
```

Setup Rubrix

If you have not installed and launched Rubrix, check the installation guide.

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

Preparing our raw dataset of recipes and ingredients

```
[ ]: import pandas as pd
     sparql = """
         SELECT distinct *
         WHERE {
             ?uri a wtm:Recipe .
             ?uri a ?type .
             ?uri skos:definition ?definition .
             ?uri wtm:hasIngredient ?ingredient
         }
     """
     df = kg.query_as_df(sparql=sparql)

     # We group the ingredients into one column containing lists:
     recipes_df = df.groupby(['uri', 'definition', 'type'])['ingredient'].apply(list).reset_
     ↪index(name='ingredients') ; recipes_df

     sparql_ingredients = """
         SELECT distinct *
         WHERE {
             ?uri a wtm:Ingredient .
             ?uri a ?type .
             OPTIONAL { ?uri skos:prefLabel ?definition }
         }
     """

     df = kg.query_as_df(sparql=sparql_ingredients)
     df['ingredients'] = None

     ing_recipes_df = pd.concat([recipes_df, df]).reset_index(drop=True)

     ing_recipes_df.fillna('', inplace=True) ; ing_recipes_df
```

Logging into Rubrix

[ ]:

```
import rubrix as rb

records = []
for i, r in ing_recipes_df.iterrows():
    item = rb.TextClassificationRecord(
            inputs={
                "id":r.uri,
                "definition": r.definition,
                "ingredients": str(r.ingredients),
                "type": r.type
            }, # log node fields
            prediction=[(label, 0.0) for label in LABELS], # log "dummy" predictions for␣
↪aiding annotation
            metadata={'ingredients': [ing.replace('ind:','') for ing in r.ingredients],
↪"type": r.type}, # metadata filters for quick exploration and annotation
            prediction_agent="kglab_tutorial", # who's performing/logging the prediction
            multi_label=True
        )
    records.append(item)
```

[ ]:

```
len(records)
```

[ ]:

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

Annotation session with Rubrix (optional)

In this step you can go to your rubrix dataset and annotate some examples of each class.

If you have no time to do this, just skip this part as we have prepared a dataset for you with around ~600 examples.

Loading our labelled records and create a train_test split (optional)

If you have no time to do this, just skip this part as we have prepared a dataset for you.

[ ]:

```
rb.snapshots(name="kg_classification_tutorial")
```

Once you have annotated your dataset, you will find an snapshot id on the previous list. This id should be place in the next command. In our case, it was 1620136587.907149.

[ ]:

```
df = rb.load(name="kg_classification_tutorial", snapshot='1620136587.907149') ; df.head()
```

[ ]:

```
from sklearn.model_selection import train_test_split

train_df, test_df = train_test_split(df)
train_df.to_csv('data/train_recipes_new.csv')
test_df.to_csv('data/test_recipes_new.csv')
```

Creating PyTorch train and test sets

Here we take our train and test datasets and transform them into `torch.Tensor` objects with the help of our kglab `Subgraph` for turning `uris` into `torch.long` indices.

[ ]:
```
import pandas as pd

train_df = pd.read_csv('data/train_recipes.csv') # use your own labelled datasets if you
↪'ve created a snapshot
test_df = pd.read_csv('data/test_recipes.csv')

# we make sure lists are parsed correctly
train_df.labels = train_df.labels.apply(eval)
test_df.labels = test_df.labels.apply(eval)
```

[ ]:
```
train_df
```

Let's create label lookups for label to int and viceversa

[ ]:
```
label2id = {label:i for i,label in enumerate(LABELS)} ;
id2label = {i:l for l,i in label2id.items()} ; (id2label, label2id)
```

The following function turns our DataFrame into numerical arrays for node indices and labels

[ ]:
```
import numpy as np

def create_indices_labels(df):
    # turn our dense labels into a one-hot list
    def one_hot(label_ids):
        a = np.zeros(len(LABELS))
        a.put(label_ids, np.ones(len(label_ids)))
        return a

    indices, labels = [], []
    for uri, label in zip(df.uri.tolist(), df.labels.tolist()):
        indices.append(sg.transform(uri))
        labels.append(one_hot([label2id[label] for label in label]))
    return indices, labels
```

Finally, let's turn our dataset into PyTorch tensors

[ ]:
```
train_indices, train_labels = create_indices_labels(train_df)
test_indices, test_labels = create_indices_labels(test_df)

train_idx = torch.tensor(train_indices, dtype=torch.long)
train_y = torch.tensor(train_labels, dtype=torch.float)

test_idx = torch.tensor(test_indices, dtype=torch.long)
test_y = torch.tensor(test_labels, dtype=torch.float) ; train_idx[:10], train_y
```

Let's see if we can recover the correct URIs for our numerical ids using our `kglab.Subgraph`

[ ]:
```
(train_df.loc[0], sg.inverse_transform(15380))
```

5.10.6 4. Creating a Subgraph of recipe and ingredient nodes

Here we create a node list to be used as a seed for building our PyG subgraph (using k-hops as we will see in the next section). Our goal will be to start only with recipes and ingredients, as all nodes passed through the GNN will be classified and those are our main target.

```
[ ]: node_idx = torch.LongTensor([
         sg.transform(i) for i in ing_recipes_df.uri.values
     ])
```

```
[ ]: node_idx.max(), node_idx.shape
```

```
[ ]: ing_recipes_df.iloc[1]
```

```
[ ]: sg.inverse_transform(node_idx[1])
```

```
[ ]: node_idx[0:10]
```

5.10.7 5. Semi-supervised node classification with PyTorch Geometric

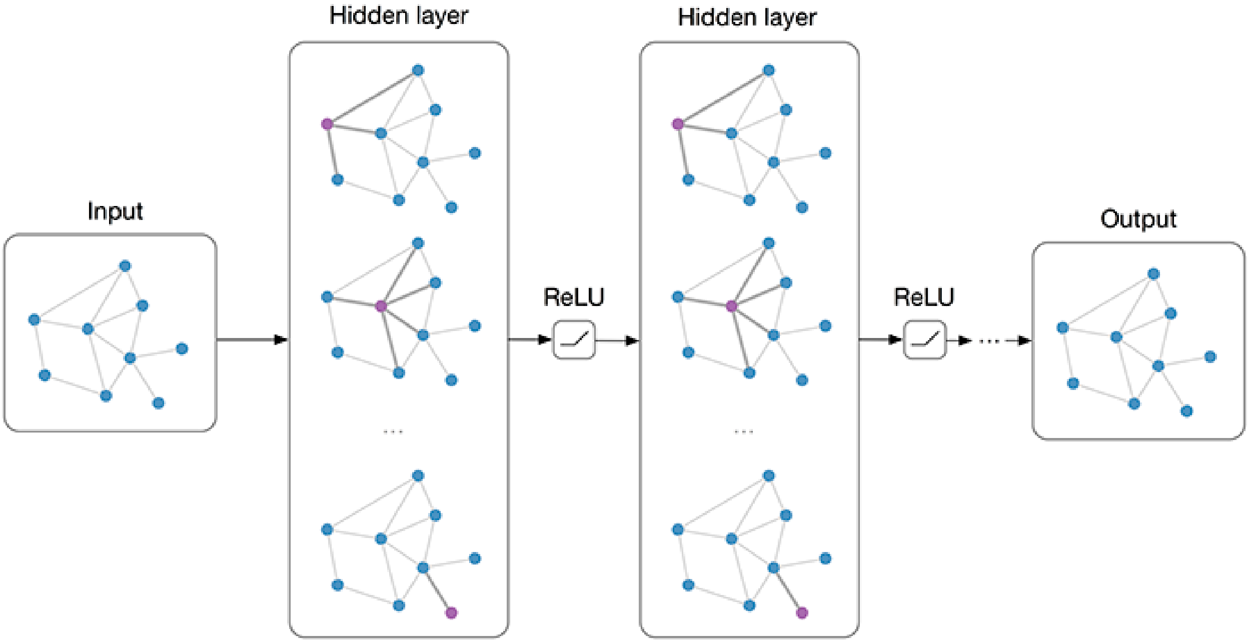
For the node classification task **we are given the ground-truth labels** (our recipes and ingredients training set) **for a small subset of nodes**, and **we want to predict the labels for all the remaining nodes** (our recipes and ingredients test set and unlabelled nodes).

Graph Convolutional Networks

To get a great intro to GCNs we recommend you to check Kipf's blog post on the topic.

In a nutshell, GCNs are multi-layer neural works which apply "convolutions" to nodes in graphs by sharing and applying the same filter parameters over all locations in the graph.

Additionally, modern GCNs such as those implemented in PyG use **message passing** mechanisms, where vertices exchange information with their neighbors, and send messages to each other.

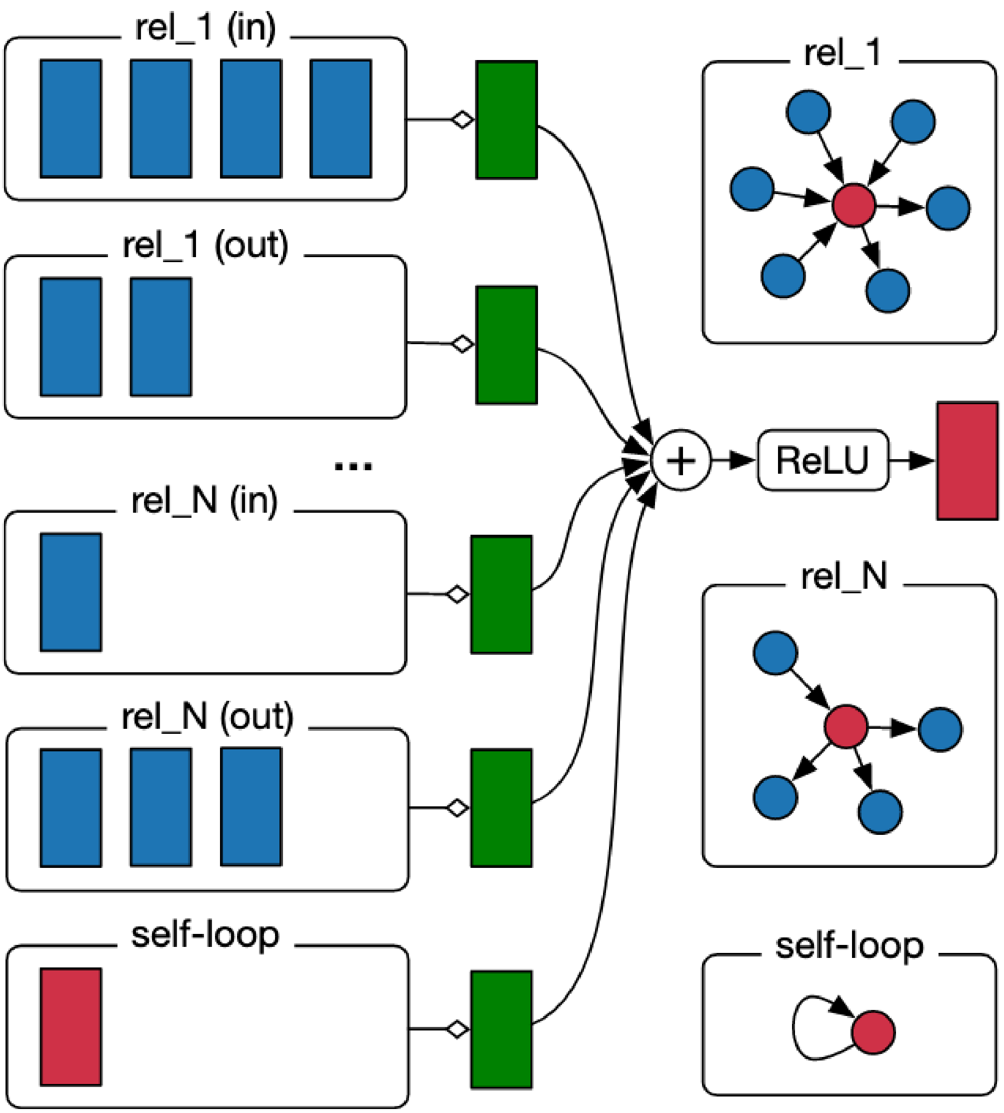


Multi-layer Graph Convolutional Network (GCN) with first-order filters. Source: https://tkipf.github.io/graph-convolutional-networks

Relational Graph Convolutional Networks

Relational Graph Convolutional Networks (R-GCNs) were introduced by Schlichtkrull et al. 2017, as an extension of GCNs to deal with **multi-relational knowledge graphs**.

You can see below the computation model for nodes:



Computation of the update of a single graph node(red) in the R-GCN model.. Source: https://arxiv.org/abs/1703.06103

Creating a PyG subgraph

Here we build a subgraph with k hops from target to source starting with all recipe and ingredient nodes:

```
[ ]: from torch_geometric.utils import k_hop_subgraph
# here we take all connected nodes with k hops
k = 1
node_idx, edge_index, mapping, edge_mask = k_hop_subgraph(
    node_idx,
    k,
    data.edge_index,
    relabel_nodes=False
```

(continues on next page)

(continued from previous page)

```
)
```

We have increased the size of our node set:

[ ]:
```
node_idx.shape
```

[ ]:
```
data.edge_index.shape
```

Here we compute some measures needed for defining the size of our layers

[ ]:
```
data.edge_index = edge_index

data.num_nodes = data.edge_index.max().item() + 1

data.num_relations = data.edge_type.max().item() + 1

data.edge_type = data.edge_type[edge_mask]

data.num_classes = len(LABELS)

data.num_nodes, data.num_relations, data.num_classes
```

Defining a basic Relational Graph Convolutional Network

[ ]:
```
from torch_geometric.nn import FastRGCNConv, RGCNConv
import torch.nn.functional as F
```

[ ]:
```
RGCNConv?
```

[ ]:
```
class RGCN(torch.nn.Module):
    def __init__(self, num_nodes, num_relations, num_classes, out_channels=16, num_
↪bases=30, dropout=0.0, layer_type=FastRGCNConv, ):

        super(RGCN, self).__init__()

        self.conv1 = layer_type(
            num_nodes,
            out_channels,
            num_relations,
            num_bases=num_bases
        )
        self.conv2 = layer_type(
            out_channels,
            num_classes,
            num_relations,
            num_bases=num_bases
        )
        self.dropout = torch.nn.Dropout(dropout)

    def forward(self, edge_index, edge_type):
```

(continues on next page)

(continued from previous page)

```
        x = F.relu(self.conv1(None, edge_index, edge_type))
        x = self.dropout(x)
        x = self.conv2(x, edge_index, edge_type)
        return torch.sigmoid(x)
```

Create and visualizing our model

```
[ ]: model = RGCN(
         num_nodes=data.num_nodes,
         num_relations=data.num_relations,
         num_classes=data.num_classes,
         #out_channels=64,
         dropout=0.2,
         layer_type=RGCNConv
     ) ; model
```

```
[ ]: # code adapted from https://colab.research.google.com/drive/
     ↪14OvFnAXggxB8vM4e8vSURUp1TaKnovzX
     %matplotlib inline
     import matplotlib.pyplot as plt
     from sklearn.manifold import TSNE
     from pytorch_lightning.metrics.utils import to_categorical

     def visualize(h, color, labels):
         z = TSNE(n_components=2).fit_transform(h.detach().cpu().numpy())

         plt.figure(figsize=(10,10))
         plt.xticks([])
         plt.yticks([])

         scatter = plt.scatter(z[:, 0], z[:, 1], s=70, c=color, cmap="Set2")
         legend = plt.legend(scatter.legend_elements()[0],labels, loc="upper right", title=
     ↪"Labels",) #*scatter.legend_elements()
         plt.show()
```

```
[ ]: pred = model(edge_index, edge_type)
```

```
[ ]: visualize(pred[train_idx], color=to_categorical(train_y), labels=LABELS)
```

```
[ ]: visualize(pred[test_idx], color=to_categorical(test_y), labels=LABELS)
```

Training our RGCN

[ ]:

```
device = torch.device('cpu') # ('cuda')
data = data.to(device)
model = model.to(device)
optimizer = torch.optim.AdamW(model.parameters())
loss_module = torch.nn.BCELoss()

def train():
    model.train()
    optimizer.zero_grad()
    out = model(data.edge_index, data.edge_type)
    loss = loss_module(out[train_idx], train_y)
    loss.backward()
    optimizer.step()
    return loss.item()

def accuracy(predictions, y):
    predictions = np.round(predictions)
    return predictions.eq(y).to(torch.float).mean()

@torch.no_grad()
def test():
    model.eval()
    pred = model(data.edge_index, data.edge_type)
    train_acc = accuracy(pred[train_idx], train_y)
    test_acc = accuracy(pred[test_idx], test_y)
    return train_acc.item(), test_acc.item()
```

[ ]:

```
for epoch in range(1, 50):
    loss = train()
    train_acc, test_acc = test()
    print(f'Epoch: {epoch:02d}, Loss: {loss:.4f}, Train: {train_acc:.4f} '
          f'Test: {test_acc:.4f}')
```

Model visualization

[ ]:

```
pred = model(edge_index, edge_type)
```

[ ]:

```
visualize(pred[train_idx], color=to_categorical(train_y), labels=LABELS)
```

[ ]:

```
visualize(pred[test_idx], color=to_categorical(test_y), labels=LABELS)
```

5.10.8 6. Using our model and analyzing its predictions with Rubrix

Let's see the shape of our model predictions

[ ]:

```
pred = model(edge_index, edge_type) ; pred
```

[ ]:

```
def find(tensor, values):
    return torch.nonzero(tensor[..., None] == values)
```

Analizing predictions over the test set

[ ]:

```
test_idx = find(node_idx,test_idx)[:,0] ; len(test_idx)
```

[ ]:

```
index = torch.zeros(node_idx.shape[0], dtype=bool)
index[test_idx] = True
idx = node_idx[index]
```

[ ]:

```
uris = [sg.inverse_transform(i) for i in idx]
predicted_labels = [l for l in pred[idx]]
```

[ ]:

```
predictions = list(zip(uris,predicted_labels)) ; predictions[0:2]
```

[ ]:

```
import rubrix as rb

records = []
for uri,predicted_labels in predictions:
    ids = ing_recipes_df.index[ing_recipes_df.uri == uri]
    if len(ids) > 0:
        r = ing_recipes_df.iloc[ids]
        # get the gold labels from our test set
        gold_labels = test_df.iloc[test_df.index[test_df.uri == uri]].labels.values[0]

        item = rb.TextClassificationRecord(
                inputs={"id":r.uri.values[0], "definition": r.definition.values[0],
↪"ingredients": str(r.ingredients.values[0]), "type": r.type.values[0]},
                prediction=[(id2label[i], score) for i,score in enumerate(predicted_
↪labels)],
                annotation=gold_labels,
                metadata={'ingredients': r.ingredients.values[0], "type": r.type.
↪values[0]},
                prediction_agent="node_classifier_v1",
                multi_label=True
        )
        records.append(item)
```

[ ]:

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

Analizing predictions over unseen nodes (and potentially relabeling them)

Let's find the ids for the nodes in our training and test sets

[ ]:
```
train_test_idx = find(node_idx,torch.cat((test_idx, train_idx)))[:,0] ; len(train_test_
↪idx)
```

Let's get the ids, uris and labels of the nodes which were not in our train/test datasets

[ ]:
```
index = torch.ones(node_idx.shape[0], dtype=bool)
index[train_test_idx] = False
idx = node_idx[index]
```

We use our `SubgraphTensor` for getting back our URIs and build `uri,predicted_labels` pairs:

[ ]:
```
uris = [sg.inverse_transform(i) for i in idx]
predicted_labels = [l for l in pred[idx]]
```

[ ]:
```
predictions = list(zip(uris,predicted_labels)) ; predictions[0:2]
```

[ ]:
```
import rubrix as rb

records = []
for uri,predicted_labels in predictions:
    ids = ing_recipes_df.index[ing_recipes_df.uri == uri]
    if len(ids) > 0:
        r = ing_recipes_df.iloc[ids]
        item = rb.TextClassificationRecord(
                inputs={"id":r.uri.values[0], "definition": r.definition.values[0],
↪"ingredients": str(r.ingredients.values[0]), "type": r.type.values[0]},
                prediction=[(id2label[i], score) for i,score in enumerate(predicted_
↪labels)],
                metadata={'ingredients': r.ingredients.values[0], "type": r.type.
↪values[0]},
                prediction_agent="node_classifier_v1",
                multi_label=True
        )
        records.append(item)
```

[ ]:
```
rb.log(records, name="kg_node_classification_unseen_nodes_v3")
```

5.10.9 Exercise 1: Training experiments with PyTorch Lightning

[ ]:
```
#!pip install wandb -qqq # optional
```

[ ]:
```
!wandb login #optional
```

[ ]:
```
from torch_geometric.data import Data, DataLoader

data.train_idx = train_idx
data.train_y = train_y
```

(continues on next page)

(continued from previous page)

```
data.test_idx = test_idx
data.test_y = test_y

dataloader = DataLoader([data], batch_size=1); dataloader
```

[ ]:

```
import torch
import pytorch_lightning as pl
from pytorch_lightning.callbacks import EarlyStopping, ModelCheckpoint
from pytorch_lightning.loggers import WandbLogger

class RGCNNodeClassification(pl.LightningModule):

    def __init__(self, **model_kwargs):
        super().__init__()

        self.model = RGCN(**model_kwargs)
        self.loss_module = torch.nn.BCELoss()

    def forward(self, edge_index, edge_type):
        return self.model(edge_index, edge_type)

    def configure_optimizers(self):
        optimizer = torch.optim.Adam(self.parameters(), lr=0.01, weight_decay=0.001)
        return optimizer

    def training_step(self, batch, batch_idx):
        idx, y = data.train_idx, data.train_y
        edge_index, edge_type = data.edge_index, data.edge_type
        x = self.forward(edge_index, edge_type)
        loss = self.loss_module(x[idx], y)
        x = x.detach()
        self.log('train_acc', accuracy(x[idx], y), prog_bar=True)
        self.log('train_loss', loss)
        return loss

    def validation_step(self, batch, batch_idx):
        idx, y = data.test_idx, data.test_y
        edge_index, edge_type = data.edge_index, data.edge_type
        x = self.forward(edge_index, edge_type)
        loss = self.loss_module(x[idx], y)
        x = x.detach()
        self.log('val_acc', accuracy(x[idx], y), prog_bar=True)
        self.log('val_loss', loss)
```

[ ]:

```
pl.seed_everything()
```

[ ]:

```
model_pl = RGCNNodeClassification(
    num_nodes=data.num_nodes,
    num_relations=data.num_relations,
    num_classes=data.num_classes,
    #out_channels=64,
```

(continues on next page)

(continued from previous page)

```
    dropout=0.2,
    #layer_type=RGCNConv
)
```

```
[ ]: early_stopping = EarlyStopping(monitor='val_acc', patience=10, mode='max')
```

```
[ ]: trainer = pl.Trainer(
         default_root_dir='pl_runs',
         checkpoint_callback=ModelCheckpoint(save_weights_only=True, mode="max", monitor="val_
     ↪acc"),
         max_epochs=200,
         #logger= WandbLogger(), # optional
         callbacks=[early_stopping]
     )
```

```
[ ]: trainer.fit(model_pl, dataloader, dataloader)
```

5.10.10 Exercise 2: Bootstrapping annotation with a zeroshot-classifier

```
[ ]: !pip install transformers -qqq
```

```
[ ]: from transformers import pipeline

     pretrained_model =  "valhalla/distilbart-mnli-12-1" # "typeform/squeezebert-mnli"

     pl = pipeline('zero-shot-classification', model=pretrained_model)
```

```
[ ]: pl("chocolate cake", LABELS, hypothesis_template='The flavour is {}.',multi_label=True)
```

```
[ ]: import rubrix as rb

     records = []
     for i, r in ing_recipes_df[50:150].iterrows():
         preds = pl(r.definition, LABELS, hypothesis_template='The flavour is {}.', multi_
     ↪label=True)
         item = rb.TextClassificationRecord(
                 inputs={
                     "id":r.uri,
                     "definition": r.definition,
                     "ingredients": str(r.ingredients),
                     "type": r.type
                 },
                 prediction=list(zip(preds['labels'], preds['scores'])), # TODO: here we log␣
     ↪he predictions of our zeroshot pipeline as a list of tuples (label, score)
                 metadata={'ingredients': r.ingredients, "type": r.type},
                 prediction_agent="valhalla/distilbart-mnli-12-1",
                 multi_label=True
             )
         records.append(item)
```

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

5.10.11 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.

5.11 Human-in-the-loop weak supervision with `snorkel`

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

5.11.1 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 widely-known Snorkel library, but a similar approach can be used with other data augmentation libraries such as Textattack or nlpaug.

What is weak supervision? and Snorkel?

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

This tutorial

In this tutorial, we'll follow the Spam classification tutorial from Snorkel's documentation and show you how to extend weak supervision workflows with Rubrix.

The tutorial is organized into:

1. **Spam classification with Snorkel**: we provide a brief overview of the tutorial
2. **Extending and finding labeling functions with Rubrix**: we analyze different strategies for extending the proposed labeling functions and for exploring new labeling functions

5.11.2 Install Snorkel, Textblob and spaCy

```
[ ]: !pip install snorkel textblob spacy -qqq
```

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

5.11.3 Setup Rubrix

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

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

Once installed, you only need to import Rubrix:

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

5.11.4 1. Spam classification with Snorkel

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 a binary classification task for detecting spam comments in youtube videos.

The dataset

We have a training set and and a test set. The first one does not include the label of the samples and it is set to -1. The test set contains ground-truth labels from the original dataset, where the label is set to 1 if it's considered SPAM and 0 for HAM.

In this tutorial we'll be using Snorkel'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 train set and evaluate it against the test set.

Let's load it in Pandas and take a look!

[3]:
```
import pandas as pd
df_train = pd.read_csv('data/yt_comments_train.csv')
df_test = pd.read_csv('data/yt_comments_test.csv')
display(df_train)
display(df_test)
```

```
      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
...          ...               ...                         ...
1581         443      Themayerlife                         NaN
1582         444       Fill Reseni  2015-05-27T17:10:53.724000
1583         445    Greg Fils Aimé                         NaN
1584         446             Lil M                         NaN
1585         447       AvidorFilms                         NaN

                                                   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
```

(continues on next page)

(continued from previous page)

```
4                         i am 2,126,492,636 viewer :D   -1.0      1
...                                                 ...    ...    ...
1581                         Check out my mummy chanel!   -1.0      4
1582               The rap: cool     Rihanna: STTUUPID   -1.0      4
1583  I hope everyone is in good spirits I&#39;m a h...   -1.0      4
1584  Lil m !!!!! Check hi out!!!!! Does live the wa...   -1.0      4
1585  Please check out my youtube channel! Just uplo...   -1.0      4

[1586 rows x 6 columns]
```

```
     Unnamed: 0                author                        date  \
0            27            2015-05-25T23:42:49.533000
1           194      MOHAMED THASLEEM  2015-05-24T07:03:59.488000
2           277            AlabaGames  2015-05-22T00:31:43.922000
3           132            Manish Ray  2015-05-23T08:55:07.512000
4           163         Sudheer Yadav  2015-05-28T10:28:25.133000
..          ...                   ...                         ...
245          32            GamezZ MTA  2015-05-09T00:08:26.185000
246         176          Viv Varghese  2015-05-25T08:59:50.837000
247         314  yakikukamo FIRELOVER  2013-07-18T17:07:06.152000
248          25            James Cook  2013-10-10T18:08:07.815000
249          11   Trulee IsNotAmazing  2013-09-07T14:18:22.601000

                                                  text  label  video
0                    Check out this video on YouTube:      1      5
1                                         super music      0      5
2    Subscribe my channel  I RECORDING FIFA 15 GOAL...      1      5
3                               This song is so beauty      0      5
4    SEE SOME MORE SONG OPEN GOOGLE AND TYPE Shakir...      1      5
..                                                 ...    ...    ...
245                        Pleas subscribe my channel      1      5
246            The best FIFA world cup song for sure.      0      5
247   hey you ! check out the channel of Alvar Lake !!      1      5
248  Hello Guys...I Found a Way to Make Money Onlin...      1      5
249             Beautiful song beautiful girl it works      0      5

[250 rows x 6 columns]
```

Labeling functions

Labeling functions (LFs) are Python function which encode heuristics (such as keywords or pattern matching), distant supervision methods (using external knowledge) or even “low-quality” crowd-worker label datasets. The goal is to create a probabilistic model which is able to combine the output of a set of noisy labels assigned by this LFs. Snorkel provides several strategies for defining and combining LFs, for more information check Snorkel LFs tutorial.

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

Let’s take a look at the original LFs:

[4]:
```
import re
```

(continues on next page)

(continued from previous page)

```
from snorkel.labeling import labeling_function, LabelingFunction
from snorkel.labeling.lf.nlp import nlp_labeling_function
from snorkel.preprocess import preprocessor
from snorkel.preprocess.nlp import SpacyPreprocessor

from textblob import TextBlob


ABSTAIN = -1
HAM = 0
SPAM = 1

# Keyword searches
@labeling_function()
def check(x):
    return SPAM if "check" in x.text.lower() else ABSTAIN

@labeling_function()
def check_out(x):
    return SPAM if "check out" in x.text.lower() else ABSTAIN

# Heuristics
@labeling_function()
def short_comment(x):
    """Ham comments are often short, such as 'cool video!'"""
    return HAM if len(x.text.split()) < 5 else ABSTAIN

# List of keywords
def keyword_lookup(x, keywords, label):
    if any(word in x.text.lower() for word in keywords):
        return label
    return ABSTAIN

def make_keyword_lf(keywords, label=SPAM):
    return LabelingFunction(
        name=f"keyword_{keywords[0]}",
        f=keyword_lookup,
        resources=dict(keywords=keywords, label=label),
    )

"""Spam comments talk about 'my channel', 'my video', etc."""
keyword_my = make_keyword_lf(keywords=["my"])

"""Spam comments ask users to subscribe to their channels."""
keyword_subscribe = make_keyword_lf(keywords=["subscribe"])

"""Spam comments post links to other channels."""
keyword_link = make_keyword_lf(keywords=["http"])

"""Spam comments make requests rather than commenting."""
keyword_please = make_keyword_lf(keywords=["please", "plz"])
```

(continues on next page)

(continued from previous page)

```
"""Ham comments actually talk about the video's content."""
keyword_song = make_keyword_lf(keywords=["song"], label=HAM)


# Pattern matching with regex
@labeling_function()
def regex_check_out(x):
    return SPAM if re.search(r"check.*out", x.text, flags=re.I) else ABSTAIN


# Third party models (TextBlob and spaCy)
# TextBlob
@preprocessor(memoize=True)
def textblob_sentiment(x):
    scores = TextBlob(x.text)
    x.polarity = scores.sentiment.polarity
    x.subjectivity = scores.sentiment.subjectivity
    return x

@labeling_function(pre=[textblob_sentiment])
def textblob_subjectivity(x):
    return HAM if x.subjectivity >= 0.5 else ABSTAIN

@labeling_function(pre=[textblob_sentiment])
def textblob_polarity(x):
    return HAM if x.polarity >= 0.9 else ABSTAIN

# spaCy

# There are two different methods to use spaCy:
# Method 1:
spacy = SpacyPreprocessor(text_field="text", doc_field="doc", memoize=True)

@labeling_function(pre=[spacy])
def has_person(x):
    """Ham comments mention specific people and are short."""
    if len(x.doc) < 20 and any([ent.label_ == "PERSON" for ent in x.doc.ents]):
        return HAM
    else:
        return ABSTAIN

# Method 2:
@nlp_labeling_function()
def has_person_nlp(x):
    """Ham comments mention specific people."""
    if any([ent.label_ == "PERSON" for ent in x.doc.ents]):
        return HAM
    else:
        return ABSTAIN
```

[5]:

```
# List of labeling functions proposed at
original_labelling_functions = [
```

(continues on next page)

(continued from previous page)

```
    keyword_my,
    keyword_subscribe,
    keyword_link,
    keyword_please,
    keyword_song,
    regex_check_out,
    short_comment,
    has_person_nlp,
    textblob_polarity,
    textblob_subjectivity,
]
```

We have mentioned multiple functions that could be used to label our data, but we never gave a solution on how to deal with the overlap and conflicts.

To handle this issue, Snorkel provide the `LabelModel`. You can read more about how it works in the Snorkel tutorial and the documentation.

Let's just use a `LabelModel` to test the proposed LFs and let's wrap it into a function so we can reuse it to evaluate new LFs along the way:

```
[6]: from snorkel.labeling import PandasLFApplier
     from snorkel.labeling.model import LabelModel

     def test_label_model(lfs):

         # Apply LFs to datasets
         applier = PandasLFApplier(lfs=lfs)
         L_train = applier.apply(df=df_train)
         L_test = applier.apply(df=df_test)
         Y_test = df_test.label.values # y_test labels

         label_model = LabelModel(cardinality=2, verbose=True) # cardinality = nº of classes
         label_model.fit(L_train=L_train, n_epochs=500, log_freq=100, seed=123)

         label_model_acc = label_model.score(L=L_test, Y=Y_test, tie_break_policy="random")[
             "accuracy"
         ]
         print(f"{'Label Model Accuracy:':<25} {label_model_acc * 100:.1f}%")
         return label_model

     label_model = test_label_model(original_labelling_functions)
```

```
100%|| 1586/1586 [00:14<00:00, 112.31it/s]
100%|| 250/250 [00:02<00:00, 98.86it/s]
```

```
Label Model Accuracy:     85.6%
```

5.11.5 2. Extending and finding labeling functions with Rubrix

In this section, we'll review some of the LFs from the original tutorial and see how to use Rubrix in combination with Snorkel.

Exploring the training set with Rubrix for initial inspiration

Rubrix lets you track data for different NLP tasks (such as *Token Classification* or *Text Classification*).

Let's log our unlabelled training set into Rubrix for initial inspiration:

[7]:

```
records= []

for index, record in df_train.iterrows():
    item = rb.TextClassificationRecord(
        id=index,
        inputs=record["text"],
        metadata = {
            "author": record.author,
            "video": str(record.video)
        }
    )
    records.append(item)
```

[8]:

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

[8]:

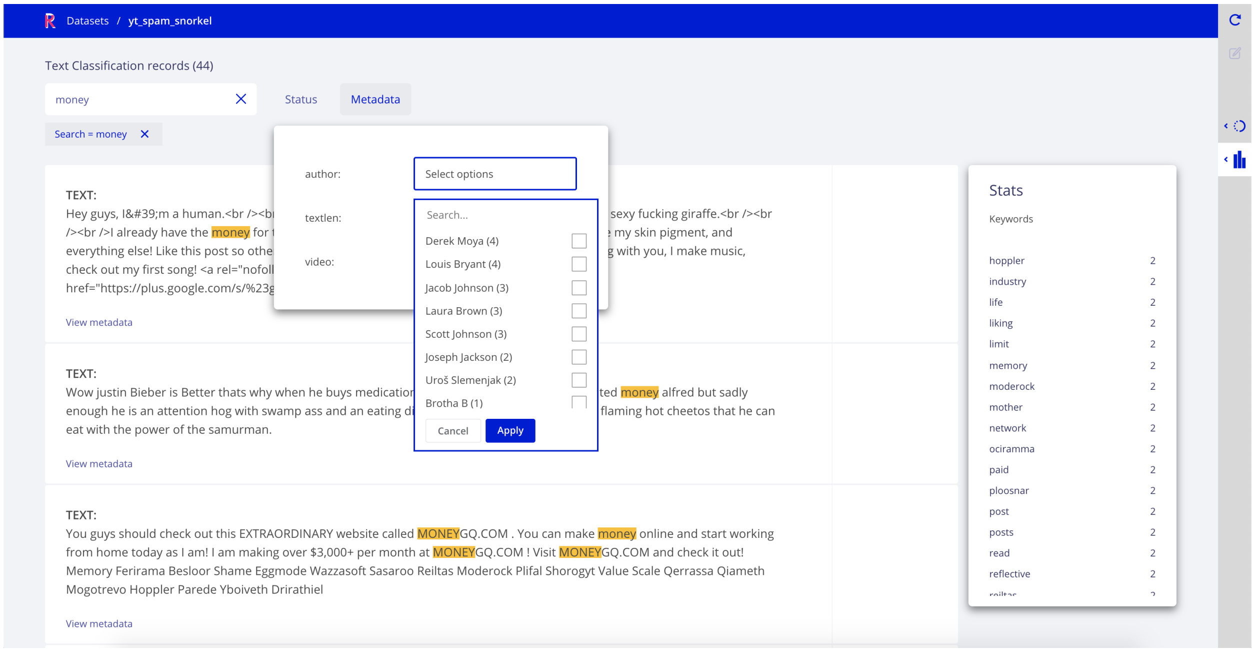
```
BulkResponse(dataset='yt_spam_snorkel', processed=1586, failed=0)
```

After a few seconds, we have a fully searchable version of our unlabelled training set, which can be used for quickly defining new LFs or improve existing ones. We can of course view our data on a text editor, using Pandas or printing rows on a Jupyter Notebook, but Rubrix focuses on making this easy and powerful with features like searching using the Elasticsearch's query string DSL, or the ability to log arbitrary inputs and metadata items.

First thing we can see on our Rubrix Dataset are the most frequent keywords on our text field. With just a quick look, we can see the coverage of two of the proposed keyword-based LFs (using the word "check" and "subscribe"):

Another thing we can do is to explore by metadata. Let's say we want to check the distribution by authors, as maybe some authors are posting SPAM several times with different wordings. Here we can see one of the top posting authors, who's also a top spammer, but seems to be using very similar messages:

Exploring some other top spammers, we see some of them use the word "money", let's check some examples using this keyword:



Datasets / yt_spam_snorkel

Text Classification records (44)

money ✕ | Status | Metadata

Search = money ✕

author: Select options

textlen:

video:

Search...

Derek Moya (4)	□
Louis Bryant (4)	□
Jacob Johnson (3)	□
Laura Brown (3)	□
Scott Johnson (3)	□
Joseph Jackson (2)	□
Uroš Slemenjak (2)	□
Brotha B (1)	□

Cancel | Apply

TEXT:
Hey guys, I&#39;m a human.<br /><br sexy fucking giraffe.<br /><br />I already have the money for t e my skin pigment, and everything else! Like this post so othe g with you, I make music, check out my first song! <a rel="nofoll href="https://plus.google.com/s/%23g

View metadata

TEXT:
Wow justin Bieber is Better thats why when he buys medicatior ted money alfred but sadly enough he is an attention hog with swamp ass and an eating di flaming hot cheetos that he can eat with the power of the samurman.

View metadata

TEXT:
You guys should check out this EXTRAORDINARY website called MONEYGQ.COM . You can make money online and start working from home today as I am! I am making over $3,000+ per month at MONEYGQ.COM ! Visit MONEYGQ.COM and check it out! Memory Ferirama Besloor Shame Eggmode Wazzasoft Sasaroo Reiltas Moderock Plifal Shorogyt Value Scale Qerrassa Qiameth Mogotrevo Hoppler Parede Yboiveth Drirathiel

View metadata

Stats

Keywords

hoppler	2
industry	2
life	2
liking	2
limit	2
memory	2
moderock	2
mother	2
network	2
ociramma	2
paid	2
ploosnar	2
post	2
posts	2
read	2
reflective	2
reiltas	2

Yes, it seems using "money" has some correlation with SPAM and a overlaps with "check" but still covers other data points (as we can see in the Keywords component).

Let's add this new LF to see its effect:

[22]:
```
@labeling_function()
def money(x):
    return SPAM if "money" in x.text.lower() else ABSTAIN
```

[23]:
```
label_model = test_label_model(original_labelling_functions + [money])
```

```
100%|| 1586/1586 [00:00<00:00, 3540.46it/s]
100%|| 250/250 [00:00<00:00, 4887.67it/s]
```

```
Label Model Accuracy:     86.8%
```

Yes! With just some quick exploration we've improved the accuracy of the Label Model by `1.2`%.

Exploring and improving heuristic LFs

We've already seen how to use keywords to label our data, the next step would be to use heuristics to do the labeling.

A simple approach proposed in the original Snorkel tutorial is checking the length of the comments' text, considering it SPAM if its length is lower than a threshold.

To find a suitable threshold we can use Rubrix to visually explore the messages, similar to what we did before with the author selection.

[24]:
```
records= []

for index, record in df_train.iterrows():
    item = rb.TextClassificationRecord(
        id=index,
        inputs=record["text"],
```

(continues on next page)

(continued from previous page)

```
        metadata = {
            "textlen": str(len(record.text.split())), # Nº of 'words' in the sample
        }
    )
    records.append(item)
```

[25]:
```
rb.log(records=records, name="yt_spam_snorkel_heuristic")
```

[25]:
```
BulkResponse(dataset='yt_spam_snorkel_heuristic', processed=1586, failed=0)
```

In the original tutorial, a threshold of 5 words is used, by exploring in Rubrix, we see we can go above that threshold. Let's try with 20 words:

[26]:
```
@labeling_function()
def short_comment_2(x):
    """Ham comments are often short, such as 'cool video!'"""
    return HAM if len(x.text.split()) < 20 else ABSTAIN
```

[27]:
```
# let's replace the original short comment function
original_labelling_functions[6]
```

[27]:
```
LabelingFunction short_comment, Preprocessors: []
```

[28]:
```
original_labelling_functions[6] = short_comment_2
```

[29]:
```
label_model = test_label_model(original_labelling_functions + [money])
```

```
100%|| 1586/1586 [00:00<00:00, 5388.84it/s]
100%|| 250/250 [00:00<00:00, 5542.86it/s]
```

```
Label Model Accuracy:     90.8%
```

Yes! With some additional exploration we've improved the accuracy of the Label Model by 5.2%.

[30]:
```
current_lfs = original_labelling_functions + [money]
```

Exploring third-party models LFs with Rubrix

Another class of Snorkel LFs are those third-party models, which can be combined with the Label Model.

Rubrix can be used for exploring how these models work with unlabelled data in order to define more precise LFs.

Let's see this with the original Textblob's based labelling functions.

Textblob

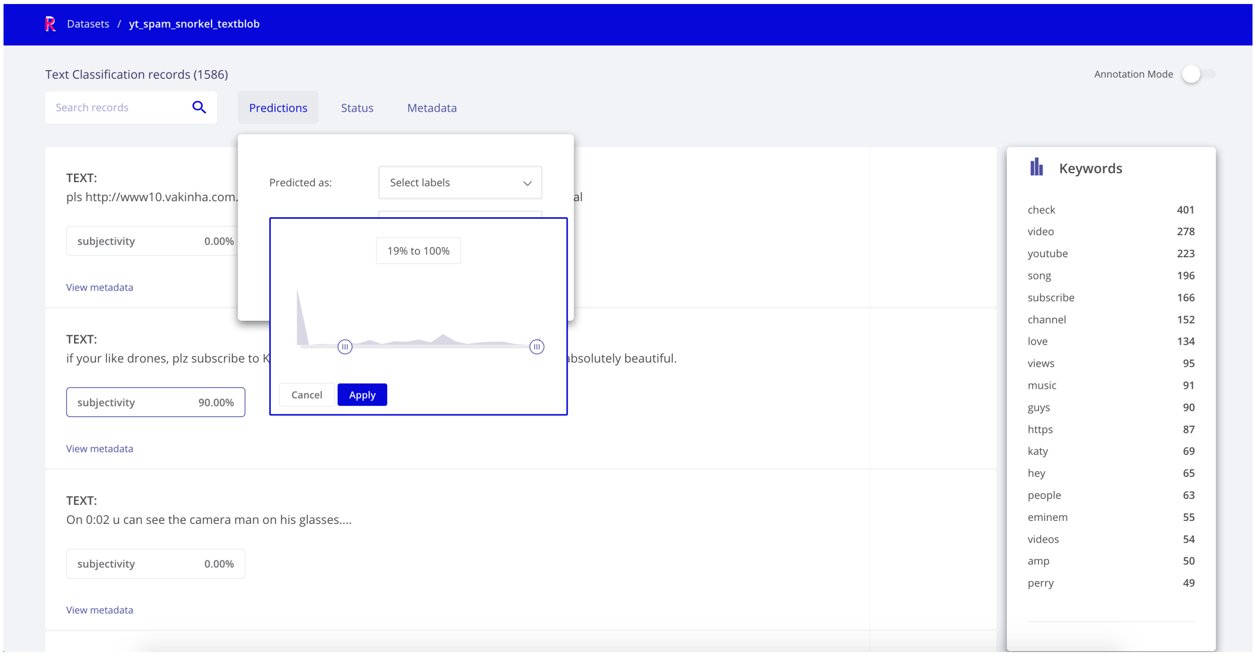
Let's explore Textblob predictions on the training set with Rubrix:

[31]:
```
from textblob import TextBlob

records= []
for index, record in df_train.iterrows():
    scores = TextBlob(record["text"])
    item = rb.TextClassificationRecord(
        id=str(index),
        inputs=record["text"],
        multi_label= False,
        prediction=[("subjectivity", max(0.0, scores.sentiment.subjectivity))],
        prediction_agent="TextBlob",
        metadata = {
            "author": record.author,
            "video": str(record.video)
        }
    )

    records.append(item)
```

[32]:
```
rb.log(records=records, name="yt_spam_snorkel_textblob")
```

[32]:
```
BulkResponse(dataset='yt_spam_snorkel_textblob', processed=1586, failed=0)
```

Checking the dataset, we can filter our data based on the prediction score of our classifier. This can help us since the predictions of our TextBlob tend to be SPAM the lower the subjectivity is. We can take advantage of this and filter the predictions by their score:



TEXT:
pls http://www10.vakinha.com. … al

subjectivity 0.00%

TEXT:
if your like drones, plz subscribe to K … absolutely beautiful.

subjectivity 90.00%

TEXT:
On 0:02 u can see the camera man on his glasses....

subjectivity 0.00%

Keywords

check	401
video	278
youtube	223
song	196
subscribe	166
channel	152
love	134
views	95
music	91
guys	90
https	87
katy	69
hey	65
people	63
eminem	55
videos	54
amp	50
perry	49

5.11.6 3. Checking and curating programatically created data

In this section, we’re going to analyse the training set we’re able to generate using our data programming model (the Label Model).

First thing, we need to do is to remove the unlabeled data. Remember we’re only labeling a subset using our model:

```
[ ]: from snorkel.labeling import filter_unlabeled_dataframe

     applier = PandasLFApplier(lfs=current_lfs)
     L_train = applier.apply(df=df_train)
     L_test = applier.apply(df=df_test)

     df_train_filtered, probs_train_filtered = filter_unlabeled_dataframe(
         X=df_train,
         y=label_model.predict_proba(L_train), # Probabilities of each data point for each␣
     ↪class
         L=L_train
     )
```

Now that we have our data, we can explore the results in Rubrix and manually relabel those cases that have been wrongly classified or keep exploring the performance of our LFs.

```
[38]: records = []
      for i, (index, record) in enumerate(df_train_filtered.iterrows()):
          item = rb.TextClassificationRecord(
              inputs=record["text"],
              # our scores come from probs_train_filtered
              # probs_train_filtered[i][j] is the probability the sample i belongs to class j
              prediction=[("HAM", probs_train_filtered[i][0]),   # 0 for HAM
                          ("SPAM", probs_train_filtered[i][1])], # 1 for SPAM
              prediction_agent="LabelModel",
```

(continues on next page)

(continued from previous page)

```
    )
    records.append(item)
```

[40]:
```
rb.log(records=records, name="yt_filtered_classified_sample")
```

[40]:
```
BulkResponse(dataset='yt_filtered_classified_sample_2', processed=1568, failed=0)
```

With this Rubrix Dataset, we can explore the predictions of our label model. We could add the label model output as `annotations` to create a training set and share it subject matter experts for review e.g., for relabelling problematic data points.

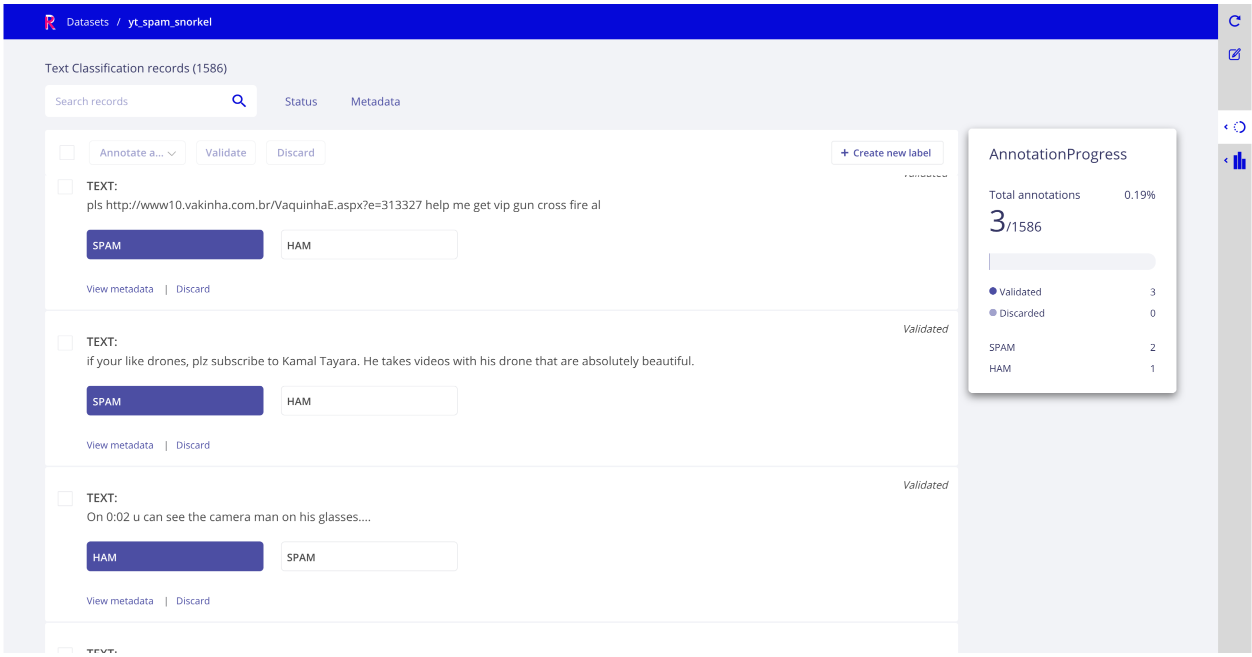
To do this, simply adding the max. probability class as `annotation`:

[36]:
```
records = []
for i, (index, record) in enumerate(df_train_filtered.iterrows()):
    gold_label = "SPAM" if probs_train_filtered[i][1] > probs_train_filtered[i][0] else "HAM"
    item = rb.TextClassificationRecord(
        inputs=record["text"],
        # our scores come from probs_train_filtered
        # probs_train_filtered[i][j] is the probability the sample i belongs to class j
        prediction=[("HAM", probs_train_filtered[i][0]),   # 0 for HAM
                    ("SPAM", probs_train_filtered[i][1])], # 1 for SPAM
        prediction_agent="LabelModel",
        annotation=[gold_label]
    )
    records.append(item)
```

[37]:
```
rb.log(records=records, name="yt_filtered_classified_sample_with_annotation")
```

[37]:
```
BulkResponse(dataset='yt_filtered_classified_sample_with_annotation', processed=1568, failed=0)
```

Using the Annotation mode, you and other users could review the labels proposed by the Snorkel model and refine the training set, with a similar exploration pattern as we used for defining LFs.



Datasets / yt_spam_snorkel

Text Classification records (1586)

Search records

Status Metadata

Annotate a... Validate Discard + Create new label

TEXT:
pls http://www10.vakinha.com.br/VaquinhaE.aspx?e=313327 help me get vip gun cross fire al

SPAM HAM

View metadata | Discard

Validated

TEXT:
if your like drones, plz subscribe to Kamal Tayara. He takes videos with his drone that are absolutely beautiful.

SPAM HAM

View metadata | Discard

Validated

TEXT:
On 0:02 u can see the camera man on his glasses....

HAM SPAM

View metadata | Discard

AnnotationProgress

Total annotations 0.19%

3/1586

● Validated 3
● Discarded 0

SPAM 2
HAM 1

5.11.7 4. Training and evaluating a classifier

The next thing we can do with our data is training a classifier using some of the most popular libraries such as Scikit-learn, Tensorflow or Pytorch. For simplicity, we will use scikit-learn, a widely-used library.

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

vectorizer = CountVectorizer(ngram_range=(1, 5)) # Bag Of Words (BoW) with n-grams
X_train = vectorizer.fit_transform(df_train_filtered.text.tolist())
X_test = vectorizer.transform(df_test.text.tolist())
```

Since we need to tell the model the class for each sample, and we have probabilities, we can assign to each sample the class with the highest probability.

```
[42]: from snorkel.utils import probs_to_preds

preds_train_filtered = probs_to_preds(probs=probs_train_filtered)
```

And then build the classifier

```
[ ]: from sklearn.linear_model import LogisticRegression

Y_test = df_test.label.values

sklearn_model = LogisticRegression(C=1e3, solver="liblinear")
sklearn_model.fit(X=X_train, y=preds_train_filtered)
```

```
[46]: print(f"Test Accuracy: {sklearn_model.score(X=X_test, y=Y_test) * 100:.1f}%")
```

```
Test Accuracy: 91.6%
```

Let's explore how our new model performs on the test data, in this case the annotation comes from the test set:

```
[47]: records = []
      for index, record in df_test.iterrows():
          preds = sklearn_model.predict_proba(vectorizer.transform([record["text"]]))
          preds = preds[0]
          item = rb.TextClassificationRecord(
              inputs=record["text"],
              prediction=[("HAM", preds[0]),   # 0 for HAM
                          ("SPAM", preds[1])], # 1 for SPAM
              prediction_agent="MyModel",
              annotation=["SPAM" if record.label == 1 else "HAM"]
          )
          records.append(item)
```

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

```
[48]: BulkResponse(dataset='yt_my_model_test', processed=250, failed=0)
```

This exploration is useful for error analysis and debugging, for example we can check all incorrectly classified examples using the Prediction filters.

5.11.8 Summary

In this tutorial, we have learnt to use Snorkel in combination with Rubrix for data programming workflows.

5.11.9 Next steps

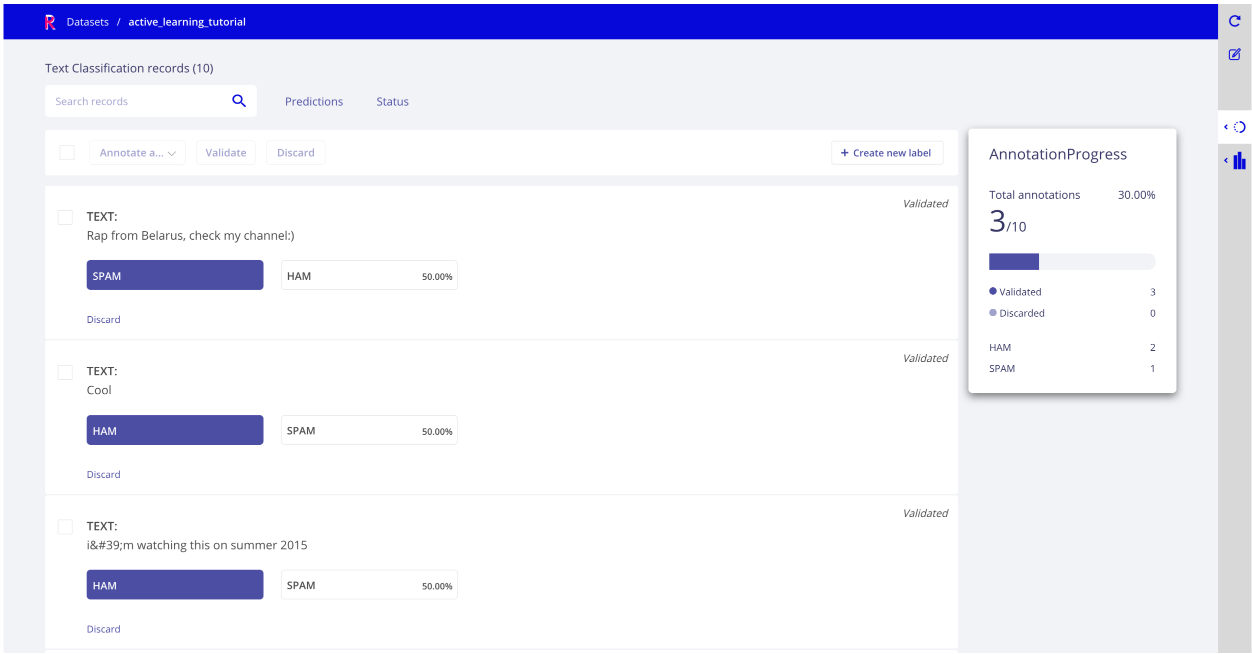
Rubrix documentation for more guides and tutorials.

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

Rubrix Github repo to stay updated.

5.12 Active learning with ModAL and scikit-learn

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



Text Classification records (10)

Validated

TEXT:
Rap from Belarus, check my channel:)

SPAM | HAM 50.00%

Validated

TEXT:
Cool

HAM | SPAM 50.00%

Validated

TEXT:
i&#39;m watching this on summer 2015

HAM | SPAM 50.00%

AnnotationProgress

Total annotations 30.00%

3/10

Validated	3
Discarded	0
HAM	2
SPAM	1

5.12.1 Introduction

Our goal is to show you how to incorporate Rubrix into interactive workflows involving a human in the loop. This is only a proof of concept for educational purposes and to inspire you with some ideas involving interactive learning processes, and how they can help to quickly build a training data set from scratch. There are several great tools which focus on active learning, being Prodi.gy the most prominent.

What is active learning?

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

This tutorial

In this tutorial, we will build a simple text classifier by combining scikit-learn, ModAL and *Rubrix*. Scitkit-learn will provide the model that we embed in an active learner from ModAL, and you and *Rubrix* will serve as the information source that teach the model to become a sample efficient classifier.

The tutorial is organized into:

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

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

5.12.2 Setup Rubrix

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

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

Once installed, you only need to import Rubrix:

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

5.12.3 Setup

Install scikit-learn and ModAL

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

```
[3]: !pip install modAL scikit-learn matplotlib -qqq
     exit(0)
```

Imports

Let us import all the necessary stuff in the beginning.

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

5.12.4 1. Loading and preparing data

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

In this tutorial, we will use the YouTube Spam Collection data set which is a binary classification task for detecting spam comments in YouTube videos. Let's load the data and have a look at it.

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

```
[4]: test_df
```

```
[4]:                                      COMMENT_ID  \
     0            z120djlhizeksdulo23mj5z52vjmxlhrk04
     1              z133ibkihkmaj3bfq22rilaxmp2yt54nb
     2          z12gxdortqzwhhqas04cfjrwituzghb5tvk0k
     3    _2viQ_Qnc6_ZYkMn1fS805Z6oy8ImeO6pSjMLAlwYfM
     4          z120s1agtmmetler404cifqbxzvdx15idtw0k
```

(continues on next page)

(continued from previous page)

```
..                                             ...
387        z13pup2w2k3rz1lxl04cf1a5qzavgvv51vg0k
388          z13psdarpuzbjp1hh04cjfwgzonextlhf1w
389        z131xnwierifxxkj204cgvjxyo3oydb42r40k
390          z12pwrxj0kfrwnxye04cjxtqntycd1yia44
391          z13oxvzqrzvyit00322jwtjo2tzqylhof04

                              AUTHOR                        DATE  \
0               Murlock Nightcrawler  2015-05-24T07:04:29.844000
1    Debora Favacho (Debora Sparkle)  2015-05-21T14:08:41.338000
2               Muhammad Asim Mansha                         NaN
3                        mile panika  2013-11-03T14:39:42.248000
4                     Sheila Cenabre         2014-08-19T12:33:11
..                               ...                         ...
387                  geraldine lopez  2015-05-20T23:44:25.920000
388                       bilal bilo  2015-05-22T20:36:36.926000
389                    YULIOR ZAMORA         2014-09-10T01:35:54
390                         2015-05-15T19:46:53.719000
391                        Octavia W  2015-05-22T02:33:26.041000

                                               CONTENT  CLASS  VIDEO
0                                   Charlie from LOST?      0      3
1                           BEST SONG EVER X3333333333      0      4
2                        Aslamu Lykum... From Pakistan      1      3
3    I absolutely adore watching football plus I’ve...      1      4
4    I really love this video.. http://www.bubblews...      1      1
..                                                 ...    ...    ...
387                          love the you lie the good      0      3
388                                      I liked<br />      0      4
389  I    loved        it            so       much  ...      0      1
390                                         good party      0      2
391                                         Waka waka       0      4

[392 rows x 6 columns]
```

As we can see the data contains the comment id, the author of the comment, the date, the content (the comment itself) and a class column that indicates if a comment is spam or ham. We will use the class column only in the test data set to illustrate the effectiveness of the active learning approach with *Rubrix*. For the training data set we simply ignore the column and assume that we are gathering training data from scratch.

5.12.5 2. Defining our classifier and Active Learner

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

[5]:
```
# Define our classification model
classifier = MultinomialNB()
```

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

[6]:
```
# Define active learner
learner = ActiveLearner(
```

(continues on next page)

(continued from previous page)

```
    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.

[7]:
```
# 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)
```

5.12.6 3. Active Learning loop

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

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

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

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

[8]:
```
# Number of instances we want to annotate per iteration
n_instances = 10

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

1. Annotate samples

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

[9]:
```
# query examples from our training pool with the most uncertain prediction
query_idx, query_inst = learner.query(X_train, n_instances=n_instances)

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

# Build the Rubrix records
```

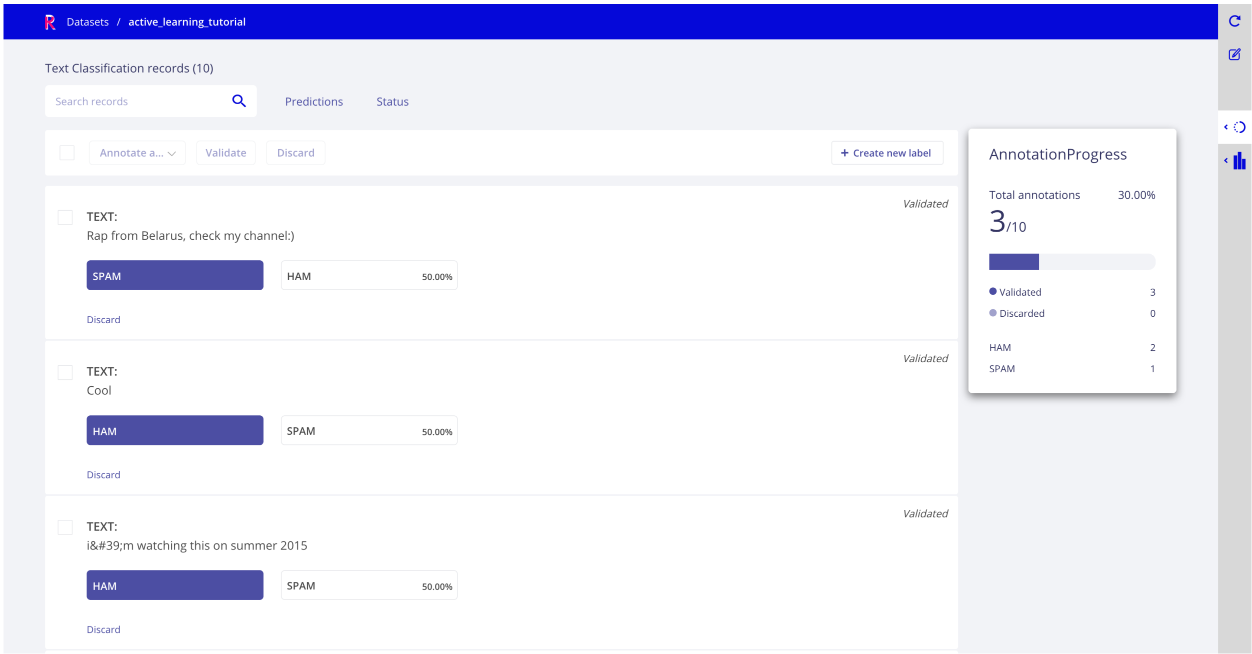
(continues on next page)

(continued from previous page)

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

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

```
[9]: 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 upper right corner to update the view with respect to the filters.



Text Classification records (10)

Validated
TEXT:
Rap from Belarus, check my channel:)
SPAM | HAM 50.00%

Validated
TEXT:
Cool
HAM | SPAM 50.00%

Validated
TEXT:
i&#39;m watching this on summer 2015
HAM | SPAM 50.00%

AnnotationProgress
Total annotations 30.00%
3/10
Validated 3
Discarded 0
HAM 2
SPAM 1

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

2. Teach the learner

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

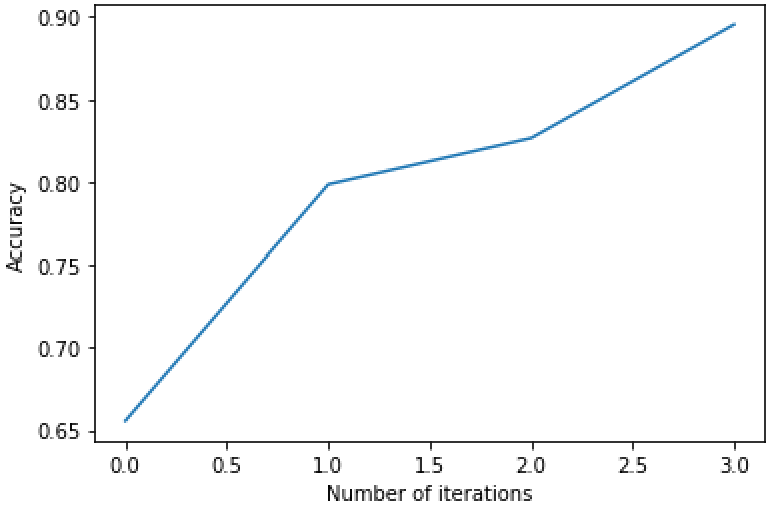
```
[ ]: # Load the annotated records into a pandas DataFrame
records_df = rb.load("active_learning_tutorial")

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

# check if all examples were annotated
if any(records_df[idx].annotation.isna()):
    raise UserWarning("Please annotate first all your samples before teaching the model")

# train the classifier with the newly annotated examples
y_train = records_df[idx].annotation.map(lambda x: int(x[0] == "SPAM"))
learner.teach(X=X_train[query_idx], y=y_train.to_list())

# Keep track of our improvement
accuracies.append(learner.score(X=X_test, y=test_df.CLASS))
```

Now go back to step 1 and repeat both steps a couple of times.

3. Plot the improvement (optional)

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

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



5.12.7 Summary

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

5.12.8 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.

5.12.9 Appendix: Compare query strategies, random vs max uncertainty

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

```
[ ]: import numpy as np

n_iterations = 150
n_instances = 10
random_samples = 50


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

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

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

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

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

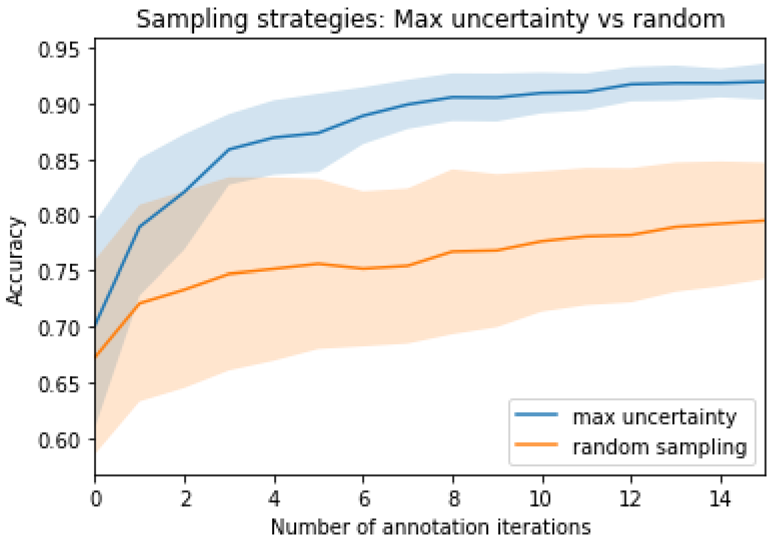
(continues on next page)

(continued from previous page)

```
        learner.teach(X=X_train[random_idx], y=train_df.CLASS.iloc[random_idx].to_list())
        accuracies.append(learner.score(X=X_test, y=test_df.CLASS))
    accuracies_rnd.append(accuracies)

arr_max, arr_rnd = np.array(accuracies_max), np.array(accuracies_rnd)
```

[ ]:
```
plt.plot(range(n_iterations), arr_max.mean(0))
plt.fill_between(range(n_iterations), arr_max.mean(0)-arr_max.std(0), arr_max.
↪mean(0)+arr_max.std(0), alpha=0.2)
plt.plot(range(n_iterations), arr_rnd.mean(0))
plt.fill_between(range(n_iterations), arr_rnd.mean(0)-arr_rnd.std(0), arr_rnd.
↪mean(0)+arr_rnd.std(0), alpha=0.2)

plt.xlim(0,15)
plt.title("Sampling strategies: Max uncertainty vs random")
plt.xlabel("Number of annotation iterations")
plt.ylabel("Accuracy")
plt.legend(["max uncertainty", "random sampling"], loc=4)
```

```
<matplotlib.legend.Legend at 0x7fa38aaaab20>
```



5.12.10 Appendix: How did we obtain the train/test data?

[ ]:
```
import pandas as pd
from urllib import request
from sklearn.model_selection import train_test_split
from pathlib import Path
from tempfile import TemporaryDirectory


def load_data() -> pd.DataFrame:
    """
    Downloads the [YouTube Spam Collection](http://www.dt.fee.unicamp.br/~tiago//
↪youtubespamcollection/)
```

(continues on next page)

(continued from previous page)

```
    and returns the data as a tuple with a train and test DataFrame.
    """
    links, data_df = [
        "http://lasid.sor.ufscar.br/labeling/datasets/9/download/",
        "http://lasid.sor.ufscar.br/labeling/datasets/10/download/",
        "http://lasid.sor.ufscar.br/labeling/datasets/11/download/",
        "http://lasid.sor.ufscar.br/labeling/datasets/12/download/",
        "http://lasid.sor.ufscar.br/labeling/datasets/13/download/",
    ], None

    with TemporaryDirectory() as tmpdirname:
        dfs = []
        for i, link in enumerate(links):
            file = Path(tmpdirname) / f"{i}.csv"
            request.urlretrieve(link, file)
            df = pd.read_csv(file)
            df["VIDEO"] = i
            dfs.append(df)
        data_df = pd.concat(dfs).reset_index(drop=True)

    train_df, test_df = train_test_split(data_df, test_size=0.2, random_state=42)

    return train_df, test_df

train_df, test_df = load_data()
train_df.to_csv("data/active_learning/train.csv", index=False)
test_df.to_csv("data/active_learning/test.csv", index=False)
```

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

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

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

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

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

- Start with the most popular sentiment classifier on the Hugging Face Hub (2.3 million monthly downloads as of July 2021) which has been fine-tuned on the SST2 sentiment dataset.
- Label a training dataset with banking user requests starting with the pre-trained sentiment classifier predictions.
- Fine-tune the pre-trained classifier with your training dataset.
- Label more data by correcting the predictions of the fine-tuned model.
- Fine-tune the pre-trained classifier with the extended training dataset.

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

Let's get started!

5.13.1 Setup Rubrix

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

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

Once installed, you only need to import Rubrix:

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

5.13.2 Install tutorial dependencies

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

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

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

Let's load the dataset directly from the hub:

```
[ ]: from datasets import load_dataset

     banking_ds = load_dataset("banking77")
```

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

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

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

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

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

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

[6]:
```
from transformers import pipeline

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

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

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

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

The model assigns more probability to the `NEGATIVE` class. Following our annotation policy (read more below), we'll label examples like this as `POSITIVE` as they are general questions, not related to issues or problems with the banking application. The ultimate goal will be to fine-tune the model to predict `POSITIVE` for these cases.

A note on sentiment analysis and data annotation

Sentiment analysis is one of the most subjective tasks in NLP. What we understand by sentiment will vary from one application to another and depend on the business objectives of the project. Also, sentiment can be modeled in different ways, leading to different **labeling schemes**. For example, sentiment can be modeled as real value (going from -1 to 1, from 0 to 1.0, etc.) or with 2 or more labels (including different degrees such as positive, negative, neutral, etc.)

For this tutorial, we'll use the **original labeling scheme** defined by the pre-trained model which is composed of two labels: `POSITIVE` and `NEGATIVE`. We could have added the `NEUTRAL` label, but let's keep it simple.

Another important issue when approaching a data annotaion project are the **annotation guidelines**, which explain how to assign the labels to specific examples. As we'll see later, the messages we'll be labeling are mostly questions with a neutral sentiment, which we'll label with the `POSITIVE` label, and some other are negative questions which we'll label with the `NEGATIVE` label. Later on, we'll show some examples of each label.

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

Predict

[16]:

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

[ ]:

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

Log

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

[18]:

```
records = []
for example in to_label1.shuffle():
    record = rb.TextClassificationRecord(
        inputs=example["text"],
        metadata={'category': example['label']}, # log the intents for exploration of␣
↪specific intents
        prediction=[(pred['label'], pred['score']) for pred in example['predictions']],
        prediction_agent="distilbert-base-uncased-finetuned-sst-2-english"
    )
    records.append(record)
```

[ ]:

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

5.13.5 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.

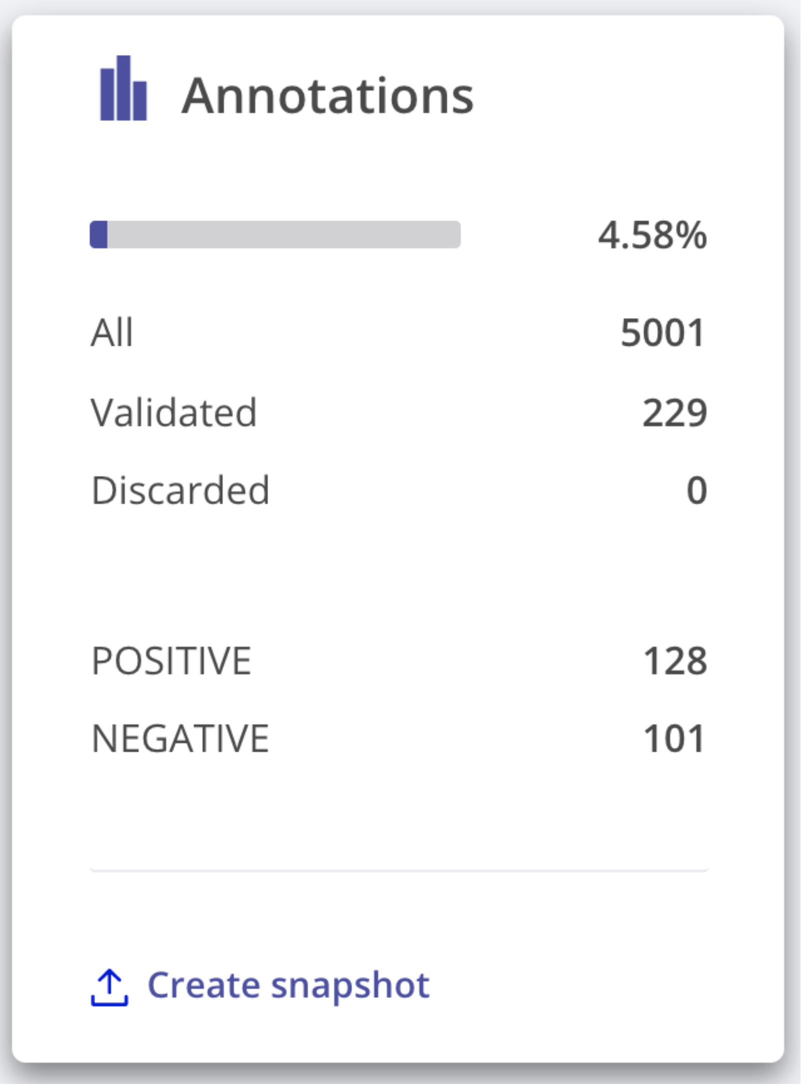
Rubrix provides you with a search-driven UI to annotated data, using free-text search, search filters and the Elasticsearch query DSL for advanced queries. This is most useful for sparse datasets, tasks with a high number of labels or unbalanced classes. In the standard case, we recommend you to follow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

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

Labeling random examples

Labeling POSITIVE examples

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



	4.58%
All	5001
Validated	229
Discarded	0
POSITIVE	128
NEGATIVE	101

5.13.6 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 our dataset:

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

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

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

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

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

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

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

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

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

Prepare training and test datasets

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

[ ]:

```
from datasets import Dataset

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


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

[6]:

```
train_ds = train_ds.train_test_split(test_size=0.2) ; train_ds
```

[6]:

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

[ ]:

```
from transformers import AutoTokenizer

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

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

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

Train our sentiment classifier

As we mentioned before, we’re going to fine-tune the `distilbert-base-uncased-finetuned-sst-2-english` model. Another option will be fine-tuning a distilbert masked language model from scratch, we leave this experiment to you.

Let’s load the model:

[1]:

```
from transformers import AutoModelForSequenceClassification

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

(continues on next page)

(continued from previous page)

Let's configure the Trainer:

[ ]:

```
import numpy as np
from transformers import Trainer
from datasets import load_metric
from transformers import TrainingArguments

training_args = TrainingArguments(
    "distilbert-base-uncased-sentiment-banking",
    evaluation_strategy="epoch",
    logging_steps=30
)

metric = load_metric("accuracy")

def compute_metrics(eval_pred):
    logits, labels = eval_pred
    predictions = np.argmax(logits, axis=-1)
    return metric.compute(predictions=predictions, references=labels)

trainer = Trainer(
    args=training_args,
    model=model,
    train_dataset=train_dataset,
    eval_dataset=eval_dataset,
    compute_metrics=compute_metrics,
)
```

And finally train our first model!

[ ]:

```
trainer.train()
```

5.13.7 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:

[33]:

```
finetuned_sentiment_classifier = pipeline(
    model=model,
    tokenizer=tokenizer,
    task="sentiment-analysis",
    return_all_scores=True
)
```

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

[34]:

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

(continues on next page)

(continued from previous page)

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

[34]:

```
([[{'label': 'NEGATIVE', 'score': 0.00024012489302549511},
   {'label': 'POSITIVE', 'score': 0.9997599124908447}]],
 [[{'label': 'NEGATIVE', 'score': 0.9992493987083435},
   {'label': 'POSITIVE', 'score': 0.0007506058318540454}]])
```

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

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

[35]:

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

[35]:

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

5.13.8 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.

[ ]:

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

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

[64]:

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

[65]:

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

[ ]:

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

[67]:

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

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

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

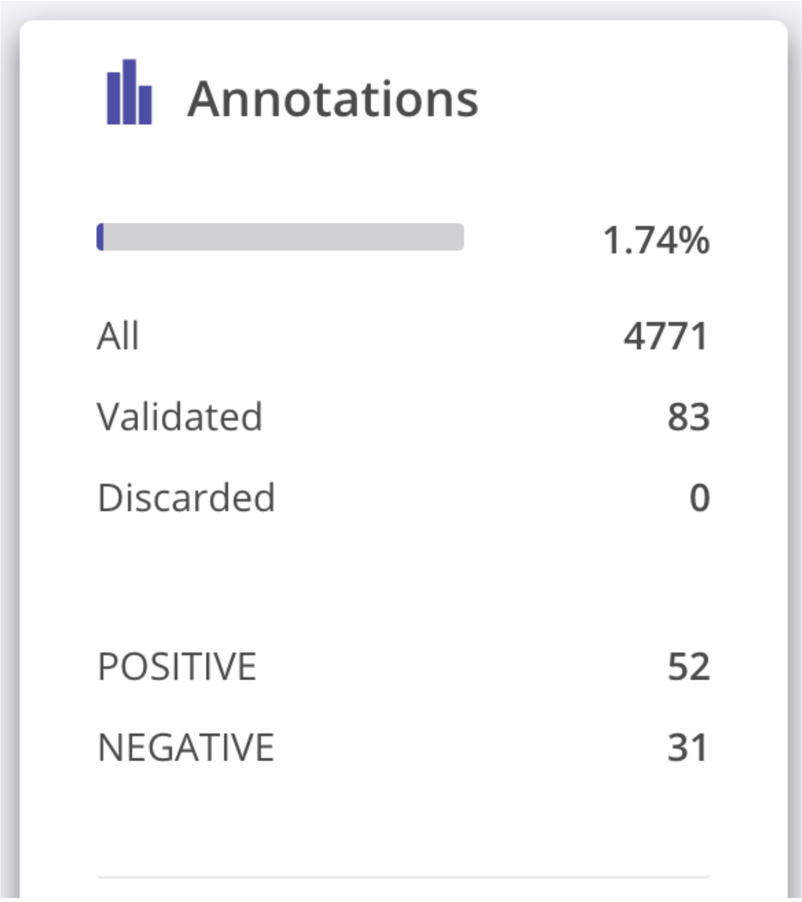
In this step, we'll start by exploring how the fine-tuned model is performing with our dataset.

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

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

1. **Use the prediction score filter** for labeling uncertain examples. Below you can see how to use this filter for labeling examples withing the range from 0 to 0.6.
2. Label examples predicted as `POSITIVE` by our fine-tuned model, and then predicted as `NEGATIVE` to correct the predictions.

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



Annotations

1.74%

All	4771
Validated	83
Discarded	0
POSITIVE	52
NEGATIVE	31

5.13.10 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.

Add labeled examples to our previous training set

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

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

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

```
[12]: 83
```

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

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

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

```
[15]: train_dataset
```

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

Train our sentiment classifier

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

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

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

[ ]:

```
train_ds = train_dataset.shuffle(seed=42)

trainer = Trainer(
    args=training_args,
    model=model,
    train_dataset=train_dataset,
    eval_dataset=eval_dataset,
    compute_metrics=compute_metrics,
)

trainer.train()
```

[ ]:

```
model.save_pretrained("distilbert-base-uncased-sentiment-banking", push_to_hub=True)
```

5.13.11 Wrap-up

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

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

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

- Active learning with modAL
- Weak supervision with Snorkel

5.13.12 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.

5.14 Find label errors with cleanlab

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

5.14.1 Introduction

As shown recently by Curtis G. Northcutt et al. label errors are pervasive even in the most-cited test sets used to benchmark the progress of the field of machine learning. In the worst-case scenario, these label errors can destabilize benchmarks and tend to favor more complex models with a higher capacity over lower capacity models.

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

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

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

5.14.2 Setup Rubrix

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

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

Once installed, you only need to import Rubrix:

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

Install tutorial dependencies

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

```
[2]: !pip install cleanlab torch transformers datasets
     exit(0)
```

Imports

Let us import all the necessary stuff in the beginning.

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

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

5.14.3 1. Load model and data set

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

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

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

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

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

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

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

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

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

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

5.14.4 2. Make predictions

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

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

         return {"probabilities": probabilities}
```

(continues on next page)

(continued from previous page)

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

5.14.5 3. Get label error candidates

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

[154]:

```
# Output the data as numpy arrays
dataset.set_format("numpy")

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

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

[164]:

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

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

5.14.6 4. Uncover label errors in Rubrix

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

[165]:

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

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

[166]:

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

    return rb.TextClassificationRecord(
        inputs={"sentence1": row.sentence1, "sentence2": row.sentence2},
```

(continues on next page)

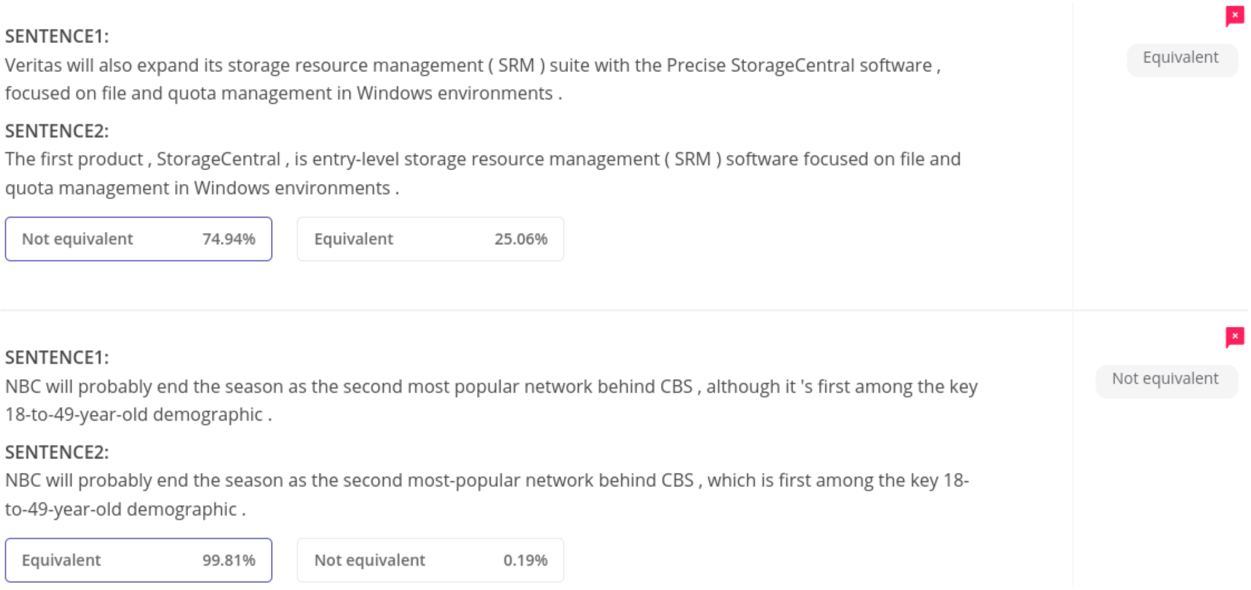
(continued from previous page)

```
        prediction=prediction,
        prediction_agent="textattack/roberta-base-MRPC",
        annotation=annotation,
        annotation_agent="MRPC"
    )

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

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

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

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



SENTENCE1:
Veritas will also expand its storage resource management (SRM) suite with the Precise StorageCentral software , focused on file and quota management in Windows environments .

SENTENCE2:
The first product , StorageCentral , is entry-level storage resource management (SRM) software focused on file and quota management in Windows environments .

Not equivalent 74.94% | Equivalent 25.06%

Equivalent

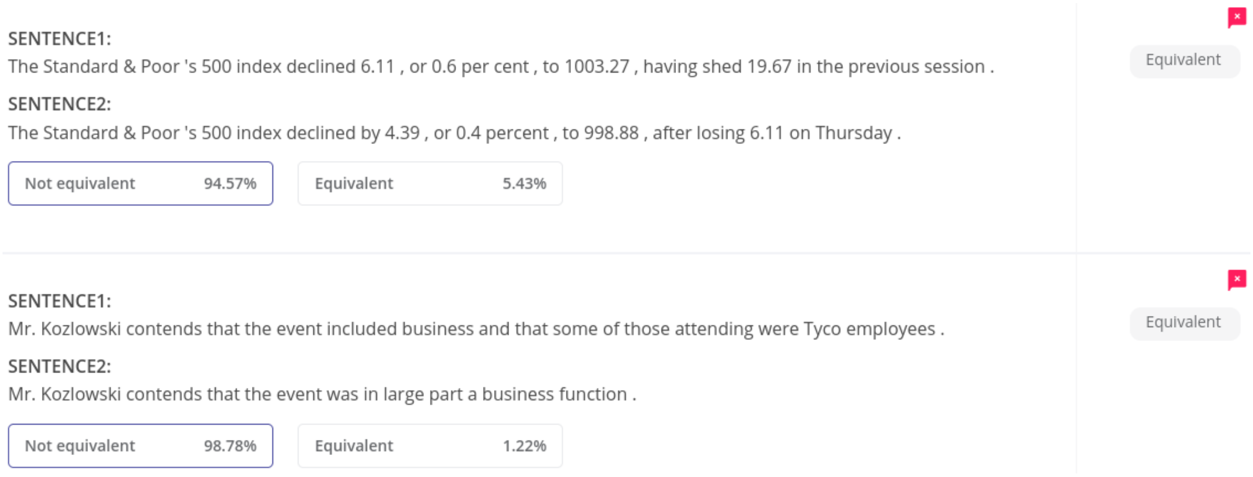
SENTENCE1:
NBC will probably end the season as the second most popular network behind CBS , although it 's first among the key 18-to-49-year-old demographic .

SENTENCE2:
NBC will probably end the season as the second most-popular network behind CBS , which is first among the key 18-to-49-year-old demographic .

Equivalent 99.81% | Not equivalent 0.19%

Not equivalent

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



SENTENCE1:
The Standard & Poor 's 500 index declined 6.11 , or 0.6 per cent , to 1003.27 , having shed 19.67 in the previous session .

SENTENCE2:
The Standard & Poor 's 500 index declined by 4.39 , or 0.4 percent , to 998.88 , after losing 6.11 on Thursday .

SENTENCE1:
Mr. Kozlowski contends that the event included business and that some of those attending were Tyco employees .

SENTENCE2:
Mr. Kozlowski contends that the event was in large part a business function .

5.14.7 5. Correct label errors

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

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

dataset_with_corrected_labels.head()
```

[181]:
```
                                              inputs  \
0  {'sentence1': 'Deaths in rollover crashes acco...
1  {'sentence1': 'Mr. Kozlowski contends that the...
2  {'sentence1': 'Larger rivals , including Tesco...
3  {'sentence1': 'The Standard & Poor 's 500 inde...
4  {'sentence1': 'Defense lawyers had said a chan...

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

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

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

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

5.14.8 Summary

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

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

5.14.9 Next steps

Rubrix documentation for more guides and tutorials.

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

Rubrix Github repo to stay updated.

5.15 Zero-shot Named Entity Recognition with Flair

5.15.1 TL;DR:

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

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

Install dependencies

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

Setup Rubrix

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

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

Once installed, you only need to import Rubrix:

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

Load the `wnut_17` dataset

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

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

[ ]:
```
from datasets import load_dataset

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

[ ]:
```
wnut_labels = [tag.split('-')[1] for tag in dataset.features['ner_tags'].feature.names if '-' in tag]
```

Configure Flair TARSTagger

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

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

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

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

Let's test it with one example!

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

[ ]:
```
tars.predict(sentence)

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

Predict over `wnut_17` and log into `rubrix`

Now, let's log the predictions in `rubrix`

[ ]:
```
records = []
for record in dataset.select(range(10)):
    input_text = " ".join(record["tokens"])

    sentence = Sentence(input_text)
```

(continues on next page)

(continued from previous page)

```
    tars.predict(sentence)
    prediction = [
        (entity.get_labels()[0].value, entity.start_pos, entity.end_pos)
        for entity in sentence.get_spans("ner")
    ]

    # Building TokenClassificationRecord
    records.append(
        rb.TokenClassificationRecord(
            text=input_text,
            tokens=[token.text for token in sentence],
            prediction=prediction,
            prediction_agent="tars-ner",
        )
    )

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

5.16 Clean labels using your model loss

5.16.1 TL;DR

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

5.16.2 Introduction

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

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

— Andrej Karpathy (@karpathy) October 2, 2020

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

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

In this tutorial, we'll use this technique with a well-known text classification benchmark, the AG News dataset. After computing the losses, we'll use Rubrix to analyse the highest loss examples. In less than 10 minutes, we manually check and relabel the first 100 examples. In fact, the first 100 examples with the highest loss, are all incorrect in the original training set. If we visually inspect further examples, we still find label errors in the top 500 examples.

5.16.3 Ingredients

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

5.16.4 Steps

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

5.16.5 Why it's important

1. **Machine learning models are only as good as the data they're trained on**. Almost all training data source can be considered "noisy" (e.g., crowd-workers, annotator errors, weak supervision sources, data augmentation, etc.)
2. With this simple technique **we're able to find more than 100 label errors on a widely-used benchmark in less than 10 minutes**. Your dataset will probably be noisier.
3. With advanced model architectures widely-available, **managing, cleaning, and curating data is becoming a key step for making robust ML applications**. A good summary of the current situation can be found in the website of the Data-centric AI NeurIPS Workshop.
4. This simple trick **can be used accross the whole ML life-cyle** and not only for finding label errors. With this trick you can improve data preprocessing, tokenization, and even your model architecture.

5.16.6 Setup Rubrix

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

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

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

Once installed, you only need to import Rubrix:

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

5.16.7 Tutorial dependencies

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

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

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

[ ]:
```
import torch

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

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

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

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

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

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

5.16.9 2. Computing the loss

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

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

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

[ ]:
```
# function to compute the loss example-wise
def loss_per_example(batch):
    batch = data_collator(batch)
    input_ids = 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():
```

(continues on next page)

(continued from previous page)

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

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

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

    return batch
```

[ ]:

```
import pandas as pd

losses_ds = ds_enc.remove_columns("text").map(loss_per_example, batched=True, batch_
↪size=32)

# turn the dataset into a Pandas dataframe, sort by descending loss and visualize the␣
↪top examples.
pd.set_option("display.max_colwidth", None)

losses_ds.set_format('pandas')
losses_df = losses_ds[:][['label', 'predicted_label', 'loss', 'predicted_probas']]

# add the text column removed by the trainer
losses_df['text'] = ds_enc['text']
losses_df.sort_values("loss", ascending=False).head(10)
```

```
        label  ...                                                                 ␣
↪                                                                                  ␣
↪                                                                                  ␣
↪              text
44984       1  ...                                                                 ␣
↪                   Baghdad blasts kills at least 16 Insurgents have detonated two␣
↪bombs near a convoy of US military vehicles in southern Baghdad, killing at least 16␣
↪people, Iraqi police say.
101562      1  ...                                          Immoral, unjust, oppressive␣
↪dictatorship. . . and then there #39;s &lt;b&gt;...&lt;/b&gt; ROBERT MUGABES␣
↪Government is pushing through legislation designed to prevent human rights␣
↪organisations from operating in Zimbabwe.
31564       1  ...  Ford to Cut 1,150 Jobs At British Jaguar Unit Ford Motor Co.␣
↪announced Friday that it would eliminate 1,150 jobs in England to streamline its␣
↪Jaguar Cars Ltd. unit, where weak sales have failed to offset spending on new products␣
↪and other parts of the business.
41247       1  ...                                            Palestinian gunmen kidnap␣
↪CNN producer GAZA CITY, Gaza Strip -- Palestinian gunmen abducted a CNN producer in␣
↪Gaza City on Monday, the network said. The network said Riyadh Ali was taken away at␣
↪gunpoint from a CNN van.
44961       1  ...             Bomb Blasts in Baghdad Kill at Least 35, Wound 120␣
↪Insurgents detonated three car bombs near a US military convoy in southern Baghdad on␣
↪Thursday, killing at least 35 people and wounding around 120, many of them children,␣
↪officials and doctors said.
```

(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]:
```
# save this to a file for further analysis
#losses_df.to_json("agnews_train_loss.json", orient="records", lines=True)
```

While using Pandas and Jupyter notebooks is useful for initial inspection, and programmatic analysis. If you want to quickly explore the examples, relabel them, and share them with other project members, Rubrix provides you with a straight-forward way for doing this. Let's see how.

5.16.10 3. Log high loss examples into Rubrix

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

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

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

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

(continues on next page)

(continued from previous page)

```
        # source of the gold label
        annotation_agent="ag_news_benchmark"
    )
```

[ ]:

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

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

[ ]:

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

5.16.11 4. Using Rubrix Webapp for inspection and relabeling

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

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

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

5.16.12 Next steps

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

Rubrix documentation for more guides and tutorials.

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

Rubrix Github repo to stay updated.

5.17 Python 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.

5.17.1 Methods

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

`rubrix.`**`copy`**(*dataset*, *name_of_copy*)

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

Examples

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

`rubrix.`**`delete`**(*name*)

Delete a dataset.

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

Return type None

Examples

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

`rubrix.`**`init`**(*api_url=None*, *api_key=None*, *timeout=60*)

Init the python client.

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

Parameters

- **`api_url`** (*`Optional[str]`*) – Address of the REST API. If *None* (default) and the env variable `RUBRIX_API_URL` is not set, it will default to *http://localhost:6900*.
- **`api_key`** (*`Optional[str]`*) – Authentification key for the REST API. If *None* (default) and the env variable `RUBRIX_API_KEY` is not set, it will default to *rubrix.apikey*.
- **`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*, *ids=None*, *limit=None*)

Load dataset data to a pandas DataFrame.

Parameters

- **name** (*str*) – The dataset name.
- **ids** (*Optional[List[Union[str, int]]]*) – If provided, load dataset records with given ids.
- **limit** (*Optional[int]*) – The number of records to retrieve.

Returns The dataset as a pandas Dataframe.

Return type pandas.core.frame.DataFrame

Examples

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

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

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.

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")
```

5.17.2 Models

This module contains the data models for the interface

`class rubrix.client.models.`**`BulkResponse`**`(`**, dataset, processed, failed=0*`)`

Summary response when logging records to the Rubrix server.

Parameters

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

Return type None

`class rubrix.client.models.`**`Text2TextRecord`**`(`**args, text, prediction=None, annotation=None, prediction_agent=None, annotation_agent=None, id=None, metadata=None, status=None, event_timestamp=None*`)`

Record for a text to text task

Parameters

- **`text`** (*`str`*) – The input of the record
- **`prediction`** (*`Optional[List[Union[str, Tuple[str, float]]]]`*) – A list of strings or tuples containing predictions for the input text. If tuples, the first entry is the predicted text, the second entry is its corresponding score.
- **`annotation`** (*`Optional[str]`*) – A string representing the expected output text for the given input text.
- **`prediction_agent`** (*`Optional[str]`*) – Name of the prediction agent.
- **`annotation_agent`** (*`Optional[str]`*) – Name of the annotation agent.
- **`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.

Return type None

`classmethod` **`prediction_as_tuples`**`(`*prediction*`)`

Preprocess the predictions and wraps them in a tuple if needed

Parameters **prediction** (*Optional[List[Union[str, Tuple[str, float]]]]*) –

class rubrix.client.models.**TextClassificationRecord**(**args*, *inputs*, *prediction=None*, *annotation=None*, *prediction_agent=None*, *annotation_agent=None*, *multi_label=False*, *explanation=None*, *id=None*, *metadata=None*, *status=None*, *event_timestamp=None*)

Record for text classification

Parameters

- **inputs** (*Union[str, List[str], Dict[str, Union[str, List[str]]]]*) – The inputs of the record
- **prediction** (*Optional[List[Tuple[str, float]]]*) – A list of tuples containing the predictions for the record. The first entry of the tuple is the predicted label, the second entry is its corresponding score.
- **annotation** (*Optional[Union[str, List[str]]]*) – A string or a list of strings (multilabel) corresponding to the annotation (gold label) for the record.
- **prediction_agent** (*Optional[str]*) – Name of the prediction agent.
- **annotation_agent** (*Optional[str]*) – Name of the annotation agent.
- **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.

Return type None

classmethod **input_as_dict**(*inputs*)

Preprocess record inputs and wraps as dictionary if needed

class rubrix.client.models.**TokenAttributions**(***, *token*, *attributions=None*)

Attribution of the token to the predicted label.

In the Rubrix app this is only supported for `TextClassificationRecord` and the `multi_label=False` case.

Parameters

- **token** (*str*) – The input token.
- **attributions** (*Dict[str, float]*) – A dictionary containing label-attribution pairs.

Return type None

class rubrix.client.models.**TokenClassificationRecord**(**args, text, tokens, prediction=None, annotation=None, prediction_agent=None, annotation_agent=None, id=None, metadata=None, status=None, event_timestamp=None*)

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[Tuple[str, int, int]]]*) – 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.
- **annotation** (*Optional[List[Tuple[str, int, int]]]*) – A list of tuples containing annotations (gold labels) for the record. The first entry of the tuple is the name of the entity, the second and third entry correspond to the start and stop char index of the entity.
- **prediction_agent** (*Optional[str]*) – Name of the prediction agent.
- **annotation_agent** (*Optional[str]*) – Name of the annotation agent.
- **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.

Return type None

5.18 Web App UI

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

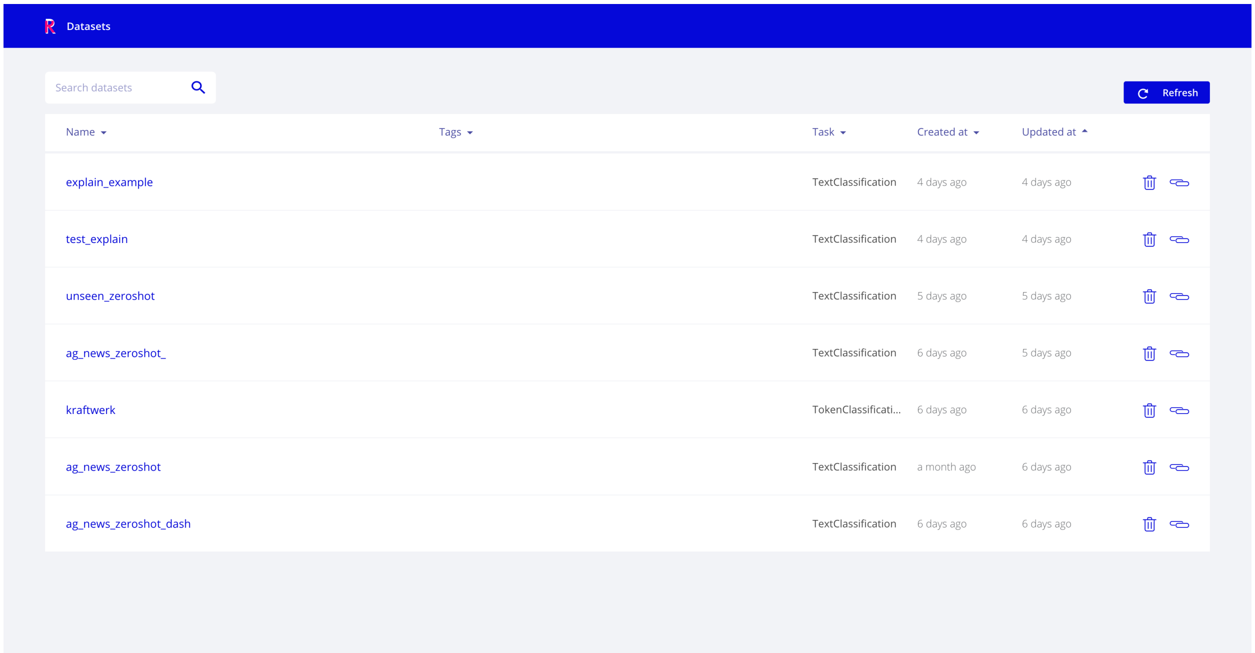
The web-app has two main pages: the **Home** page and the **Dataset** page.

5.18.1 Home page

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

- **Name**
- **Tags**, which displays the `tags` passed to the `rubrix.log` method. Tags are useful to organize your datasets by project, model, status and any other dataset attribute you can think of.
- **Task**, which is defined by the type of `Records` logged into the dataset.
- **Created at**, which corresponds to the timestamp of the Dataset creation. Datasets in Rubrix are created by directly using `rb.log` to log a collection of records.

- **Updated at**, which corresponds to the timestamp of the last update to this dataset, either by adding/changing/removing some annotations with the UI or via the Python client or the REST API.



Name	Tags	Task	Created at	Updated at
explain_example		TextClassification	4 days ago	4 days ago
test_explain		TextClassification	4 days ago	4 days ago
unseen_zeroshot		TextClassification	5 days ago	5 days ago
ag_news_zeroshot_		TextClassification	6 days ago	5 days ago
kraftwerk		TokenClassificati...	6 days ago	6 days ago
ag_news_zeroshot		TextClassification	a month ago	6 days ago
ag_news_zeroshot_dash		TextClassification	6 days ago	6 days ago

Fig. 1: Rubrix Home page view

5.18.2 Dataset page

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

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

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

Search input

This component enables:

Full-text queries over all record `inputs`.

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

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

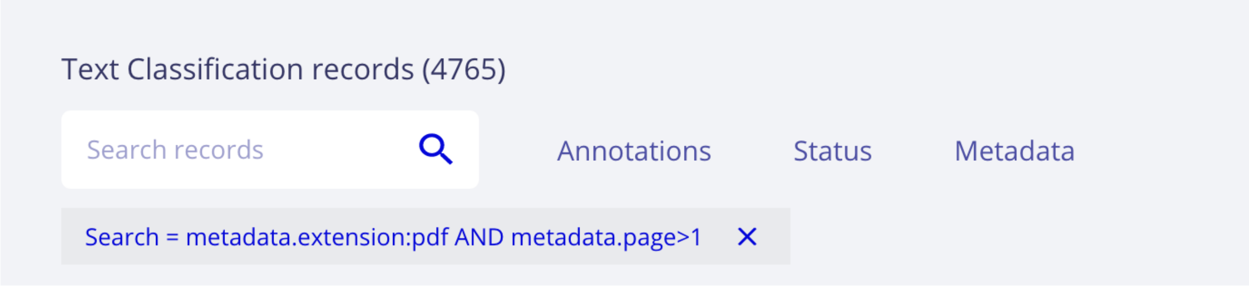
`inputs.text:(NOT women)` : records NOT containing women in the inputs.text field.

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

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

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

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



Search = metadata.extension:pdf AND metadata.page>1

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

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

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

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

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

Elasticsearch fields

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

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

Text classification fields
inputs.*
score

Tokens classification fields
tokens

Predictions filters

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

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

Annotations filters

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

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

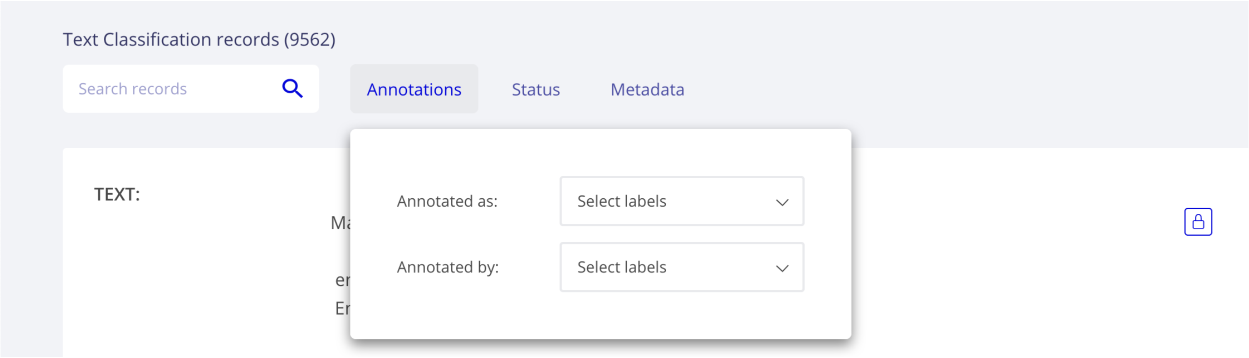


Fig. 3: Rubrix annotation filters

Status filter

This component allows filtering by record status:

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

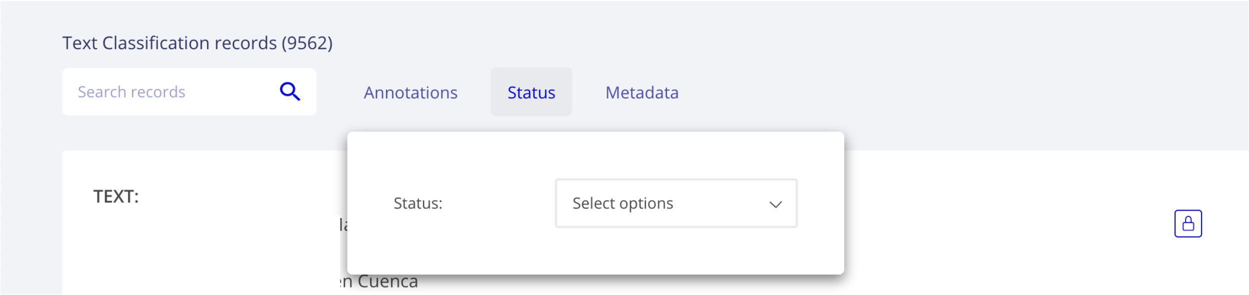


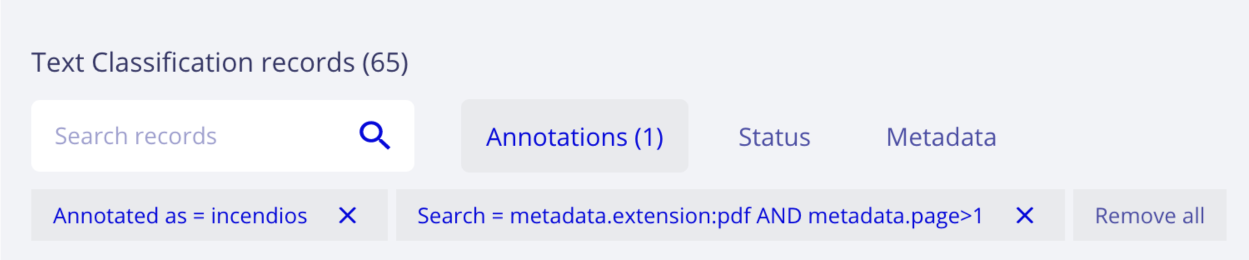
Fig. 4: Rubrix status filters

Metadata filters

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

Active query parameters

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



Annotated as = incendios ✕

Search = metadata.extension:pdf AND metadata.page>1 ✕

Fig. 5: Active query params module

Explore mode

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

Annotation mode

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

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

5.19 Developer documentation

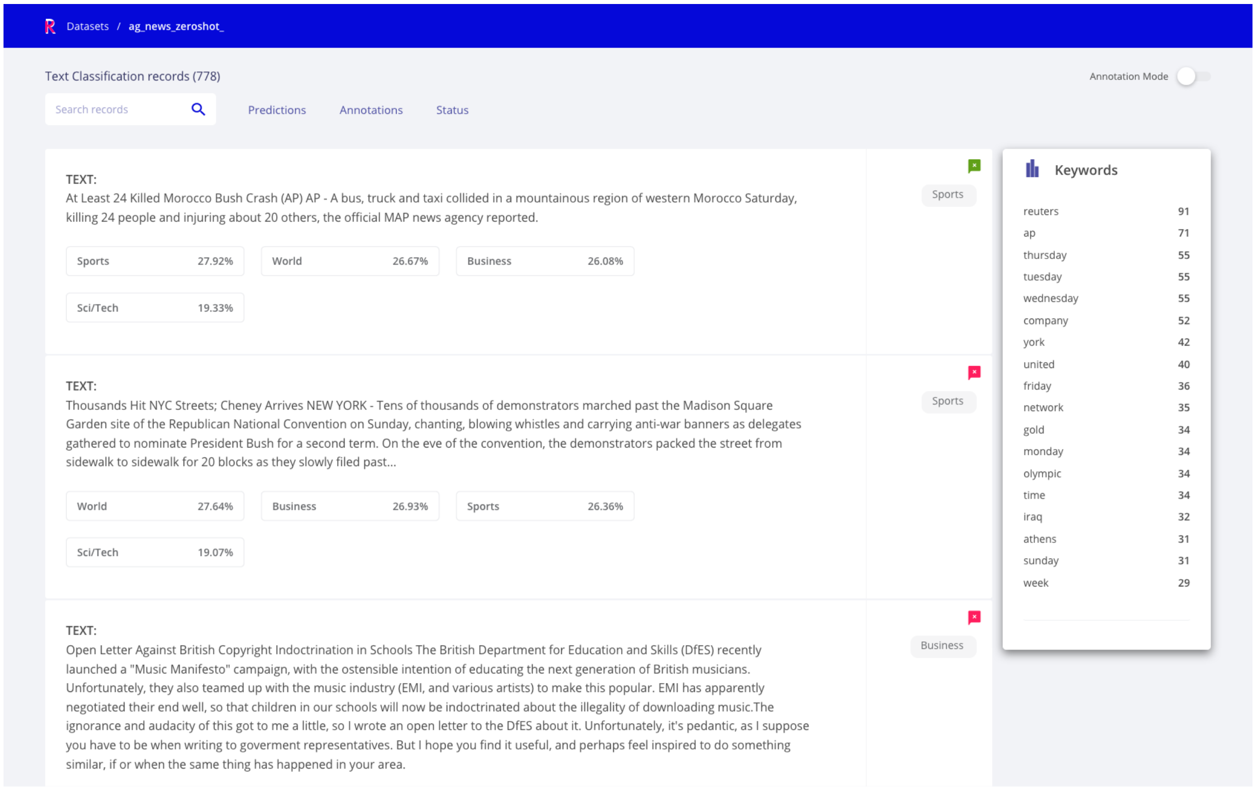
Here we provide some guides for the development of *Rubrix*.

5.19.1 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:



TEXT:
At Least 24 Killed Morocco Bush Crash (AP) AP - A bus, truck and taxi collided in a mountainous region of western Morocco Saturday, killing 24 people and injuring about 20 others, the official MAP news agency reported.

Sports	27.92%	World	26.67%	Business	26.08%
Sci/Tech	19.33%				

Sports

TEXT:
Thousands Hit NYC Streets; Cheney Arrives NEW YORK - Tens of thousands of demonstrators marched past the Madison Square Garden site of the Republican National Convention on Sunday, chanting, blowing whistles and carrying anti-war banners as delegates gathered to nominate President Bush for a second term. On the eve of the convention, the demonstrators packed the street from sidewalk to sidewalk for 20 blocks as they slowly filed past...

World	27.64%	Business	26.93%	Sports	26.36%
Sci/Tech	19.07%				

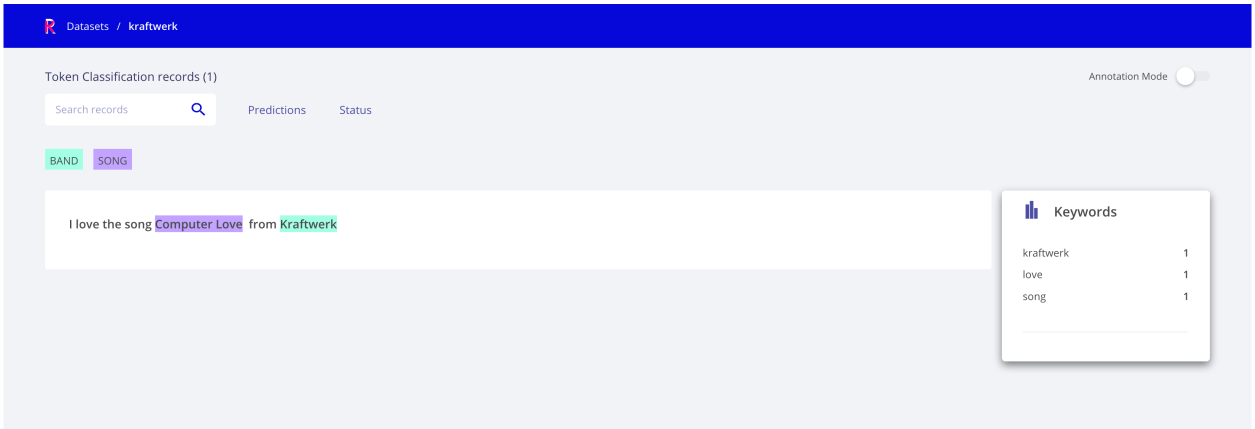
Sports

TEXT:
Open Letter Against British Copyright Indoctrination in Schools The British Department for Education and Skills (DfES) recently launched a "Music Manifesto" campaign, with the ostensible intention of educating the next generation of British musicians. Unfortunately, they also teamed up with the music industry (EMI, and various artists) to make this popular. EMI has apparently negotiated their end well, so that children in our schools will now be indoctrinated about the illegality of downloading music.The ignorance and audacity of this got to me a little, so I wrote an open letter to the DfES about it. Unfortunately, it's pedantic, as I suppose you have to be when writing to goverment representatives. But I hope you find it useful, and perhaps feel inspired to do something similar, if or when the same thing has happened in your area.

Business

Keywords	
reuters	91
ap	71
thursday	55
tuesday	55
wednesday	55
company	52
york	42
united	40
friday	36
network	35
gold	34
monday	34
olympic	34
time	34
iraq	32
athens	31
sunday	31
week	29

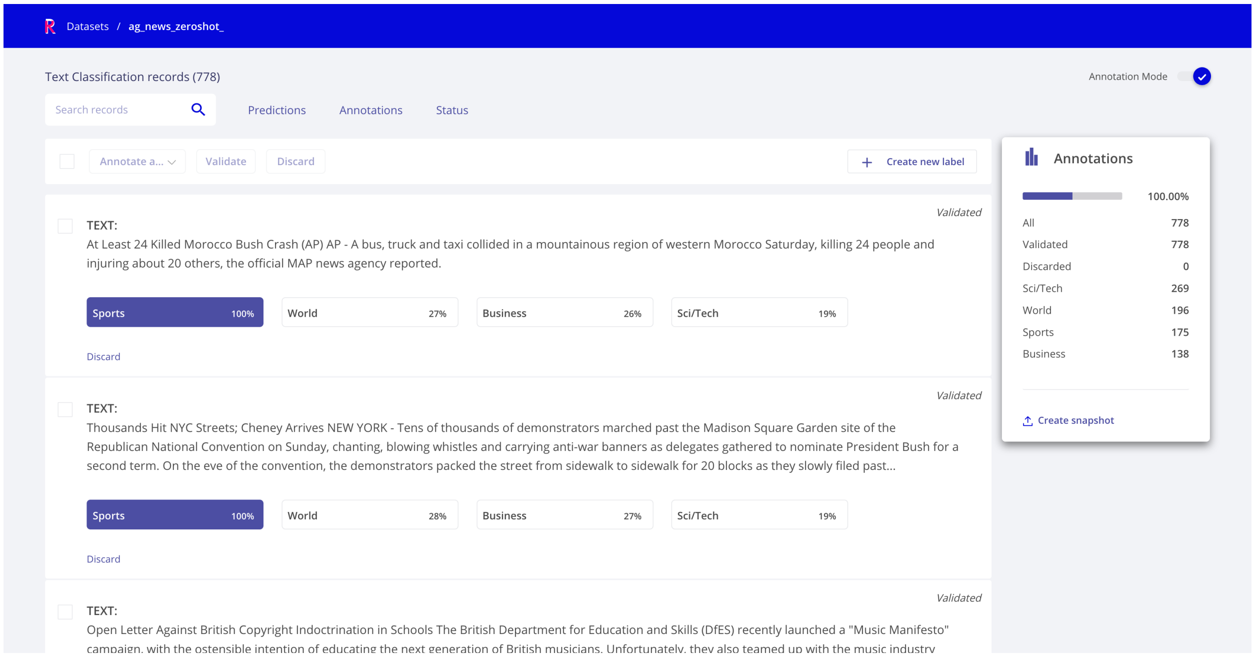
Fig. 6: Rubrix Text Classification Explore mode



I love the song Computer Love from Kraftwerk

Keywords	
kraftwerk	1
love	1
song	1

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


Datasets / ag_news_zeroshot_

Text Classification records (778)

Annotation Mode ✓

Search records

Predictions Annotations Status

□ Annotate a... Validate Discard

Create new label

Validated

□ TEXT:
At Least 24 Killed Morocco Bush Crash (AP) AP - A bus, truck and taxi collided in a mountainous region of western Morocco Saturday, killing 24 people and injuring about 20 others, the official MAP news agency reported.

Sports	World	Business	Sci/Tech
100%	27%	26%	19%

Discard

Validated

□ TEXT:
Thousands Hit NYC Streets; Cheney Arrives NEW YORK - Tens of thousands of demonstrators marched past the Madison Square Garden site of the Republican National Convention on Sunday, chanting, blowing whistles and carrying anti-war banners as delegates gathered to nominate President Bush for a second term. On the eve of the convention, the demonstrators packed the street from sidewalk to sidewalk for 20 blocks as they slowly filed past...

Sports	World	Business	Sci/Tech
100%	28%	27%	19%

Discard

Validated

□ TEXT:
Open Letter Against British Copyright Indoctrination in Schools The British Department for Education and Skills (DfES) recently launched a "Music Manifesto" campaign, with the ostensible intention of educating the next generation of British musicians. Unfortunately, they also teamed up with the music industry

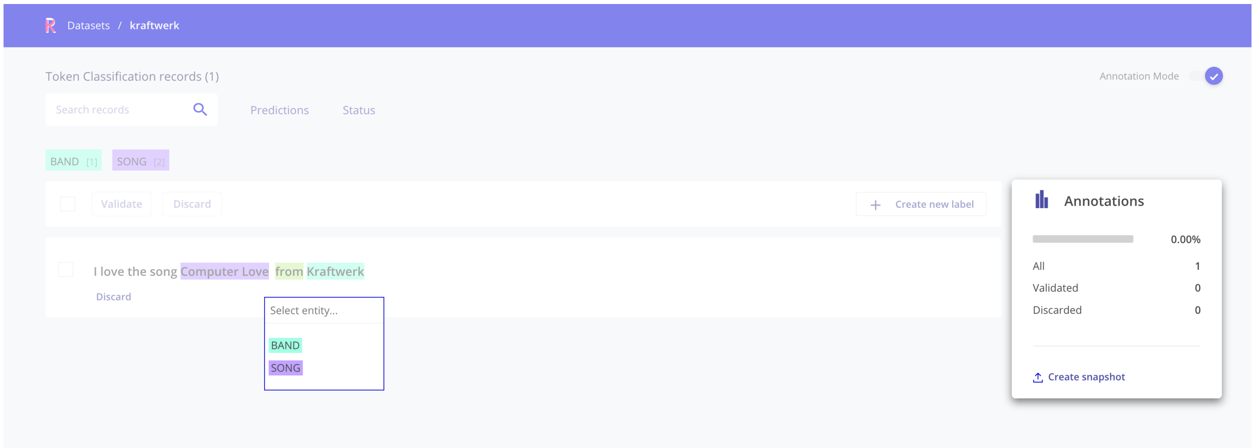
Annotations

100.00%

All	778
Validated	778
Discarded	0
Sci/Tech	269
World	196
Sports	175
Business	138

Create snapshot

Fig. 8: Rubrix Text Classification Annotation mode


Datasets / kraftwerk

Token Classification records (1)

Annotation Mode ✓

Search records

Predictions Status

BAND [1] SONG [2]

□ Validate Discard

Create new label

□ I love the song Computer Love from Kraftwerk

Discard

Select entity...

BAND
SONG

Annotations

0.00%

All	1
Validated	0
Discarded	0

Create snapshot

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

```
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
```

Once you activated the environment, it is time to install *Rubrix* in editable mode with its server dependencies:

```
pip install -e .[server]
```

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

5.19.2 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