
Rubrix

Release 0.1.1.dev0+gf4ed7bd.d20210614

Recognai

Jun 14, 2021

GETTING STARTED

1	What's Rubrix?	3
2	Quickstart	5
3	Use cases	7
4	Design Principles	9
5	Next steps	11
6	Community	13
6.1	Setup and installation	13
6.1.1	1. Install the Rubrix Python client	13
6.1.2	2. Setup and launch the webapp	13
6.1.3	3. Testing the installation by logging some data	14
6.1.4	Next steps	15
6.2	Concepts	15
6.2.1	Rubrix Data model	15
6.2.2	Methods	18
6.3	Tasks	18
6.3.1	Supported tasks	18
6.3.2	Tasks on the roadmap	19
6.4	Monitoring and collecting data from third-party apps	19
6.4.1	What does our streamlit app do?	20
6.4.2	How to run the app	20
6.4.3	Rubrix integration	20
6.5	Rubrix Cookbook	21
6.5.1	Hugging Face Transformers	21
6.5.2	spaCy	24
6.5.3	Flair	25
6.5.4	Stanza	28
6.6	Using Rubrix to explore NLP data with Hugging Face datasets and transformers	30
6.6.1	Introduction	31
6.6.2	Install tutorial dependencies	31
6.6.3	Setup Rubrix	31
6.6.4	1. Storing and exploring text classification training data	31
6.6.5	2. Storing and exploring token classification training data	36
6.6.6	3. Exploring predictions	39
6.6.7	Summary	42
6.6.8	Next steps	42
6.7	Using Rubrix with spaCy	42

6.7.1	Introduction	42
6.7.2	Install tutorial dependencies	42
6.7.3	Setup Rubrix	43
6.7.4	Our dataset	43
6.7.5	Logging spaCy NER entities into Rubrix	43
6.7.6	Exploring and comparing <code>en_core_web_sm</code> and <code>en_core_web_trf</code> models	45
6.7.7	Summary	46
6.7.8	Next steps	46
6.8	Node classification with <code>kglab</code> and PyTorch Geometric	46
6.8.1	Our use case in a nutshell	47
6.8.2	Install <code>kglab</code> and Pytorch Geometric	47
6.8.3	1. Loading and exploring the recipes knowledge graph	47
6.8.4	2. Representing our knowledge graph as a PyTorch Tensor	48
6.8.5	3. Building a training set with Rubrix	49
6.8.6	4. Creating a Subgraph of recipe and ingredient nodes	53
6.8.7	5. Semi-supervised node classification with PyTorch Geometric	53
6.8.8	6. Using our model and analyzing its predictions with Rubrix	59
6.8.9	Exercise 1: Training experiments with PyTorch Lightning	60
6.8.10	Exercise 2: Bootstrapping annotation with a zeroshot-classifier	62
6.9	Using Rubrix and Snorkel for human-in-the-loop weak supervision	63
6.9.1	Introduction	63
6.9.2	Install Snorkel, Textblob and spaCy	63
6.9.3	1. Spam classification with Snorkel	64
6.9.4	2. Extending and finding labeling functions with Rubrix	68
6.9.5	3. Checking and curating programatically created data	73
6.9.6	4. Training and evaluating a classifier	75
6.9.7	Summary	77
6.9.8	Next steps	77
6.10	Python client API	77
6.10.1	Methods	77
6.10.2	Models	79
6.11	Rubrix UI	81
6.11.1	Home page	81
6.11.2	Dataset page	81
6.12	Developer documentation	87
6.12.1	Development setup	87
6.12.2	Building the documentation	87
	Python Module Index	89
	Index	91

Welcome to Rubrix's documentation.

WHAT'S RUBRIX?

Rubrix is a free and open-source tool for tracking and iterating on data for AI projects.

With Rubrix, you can:

- **Monitor** the predictions of deployed models.
- **Collect** ground-truth data for starting up a project or evolving an existing one.
- **Iterate** on ground-truth data and predictions to debug, track and improve your models over time.
- **Build** custom applications and dashboards on top of your model predictions and ground-truth data.

Rubrix is designed to enable novel, human-in-the loop workflows involving data scientists, subject matter experts and data engineers for curating, understanding and evolving data for AI and data science projects.

We've tried to make Rubrix easy, fun and seamless to use with your favourite libraries while keeping it scalable and flexible. Rubrix's main components are:

- a **Python library** to enable data scientists, data engineers and DevOps roles to build bridges between data, models and users, which you can install with `pip`.
- a **web application** for exploring, curating and labelling data, which you can launch using `Docker` or with a local installation.
- a **REST API** for storing, retrieving and searching human annotations and model predictions, which is part of Rubrix's installation.

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

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 transformers datasets torch
```

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

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

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

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

# Our labels are: ['World', 'Sports', 'Business', 'Sci/Tech']
labels = dataset.features["label"].names

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

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

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


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.
- **App development:** get a powerful search-based API on top of your model predictions and ground truth data.

DESIGN PRINCIPLES

Rubrix's design is:

- **Agnostic:** you can use Rubrix with any library or framework, no need to implement any interface or modify your existing toolbox and workflows.
- **Flexible:** Rubrix does not make any strong assumption about your input data, so you can log and structure your data as it fits your use case.
- **Minimalistic:** Rubrix is built around a small set of concepts and methods.

NEXT STEPS

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

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

COMMUNITY

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

- [Github page](#)
- [Github forum](#)

6.1 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

6.1.1 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
```

6.1.2 2. Setup and launch the webapp

There are two ways to launch the webapp:

1. Using `docker-compose` (**recommended**).
2. Executing the server code manually

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.yml && docker-compose up
```

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

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]
```

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>. If you want to customize this, you can set the `ELASTICSEARCH` environment variable pointing to your endpoint.

Checking your webapp and REST API

Now you should be able to access Rubrix via <http://localhost:6900/>, and you can also check the API docs at <http://localhost:6900/api/docs>.

6.1.3 3. Testing the installation by logging some data

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

```
import rubrix as rb

rb.log(
    rb.TextClassificationRecord(inputs={"text": "my first rubrix example"},
    name='example-dataset'
)
```

You should receive this response in your terminal or Jupyter Notebook:

```
BulkResponse(dataset='example-dataset', processed=1, failed=0)
```

This means that the data has been logged correctly.

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

Congratulations! You are ready to start working with Rubrix.

6.1.4 Next steps

To continue learning we recommend you to:

- Check our **guides** and **tutorials**.
- Read about Rubrix's main **concepts**.

6.2 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.

6.2.1 Rubrix Data model

The Python library and the UI are built around a few simple but key concepts. This section aims to clarify what those concepts and 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.

Snapshot

A snapshot is a version of a `Dataset` containing `annotations` at a given time. Snapshots can be created through the Rubrix UI so they can be loaded and used using the Python library.

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 labeled examples and see how it start predicting over the very same labels.

Let's see examples over 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 Egypt and visit the pyramids" fits into the hashtag #Travel, which is accurate, in Multi-label Text Classification we can classify it as more than one hashtag, like #Travel #History #Africa #Sightseeing #Desert.

```
record = rb.TextClassificationRecord(  
    inputs={  
        "text": "I can't wait to travel to Egypt and visit the pyramids"  
    },  
    multi_label = True,  
  
    prediction = [('travel', 0.8), ('history', 0.6), ('economy', 0.3), ('sports', 0.2)],  
    prediction_agent = "link or reference to agent",  
  
    # When annotated, scores are supposed 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 syllables, 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!

6.2.2 Methods

To find more information about these methods, please check out the *Python client API*.

rb.init

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

rb.log

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

rb.load

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

rb.snapshots

Retrieve a list of dataset snapshots: `rubrix.snapshots()`

rb.delete

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

6.3 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](#).

6.3.1 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:

[Key phrase extraction](#)

- [Sentiment analysis](#)
- [Natural Language Inference](#)
- [Relationship Extraction](#)
- [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
- [Key phrase extraction](#)
- Slot filling

6.3.2 Tasks on the roadmap

Natural language processing

- Text2Text, covering summarization, machine translation, natural language generation, etc.
- Question answering

Computer vision

- Image classification
- Image captioning

Speech

- Speech2Text

6.4 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.

6.4.1 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 **multilabel**, 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-middle* 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 retrainsings.

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!

6.4.2 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.

6.4.3 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:

```
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"
```

(continues on next page)

(continued from previous page)

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

6.5 Rubrix Cookbook

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

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

With these examples you'll be able to start exploring and annotating 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](#).

6.5.1 Hugging Face Transformers

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

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

Text Classification

Inference

Let's try a zero-shot classifier using SqueezeBERT 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/squeezebert-mnli",
          framework="pt",
      )

      # Making the prediction
      prediction = classifier(
          input_text,
          candidate_labels=[
              "politics",
              "sports",
```

(continues on next page)

(continued from previous page)

```

        "technology",
    ],
    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="https://huggingface.co/typeform/squeezebert-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 flatten the annotations and create a dict for turning labels into numeric ids
    df['labels'] = df.annotation.transform(lambda r: r[0])
    label2id = {label:id for id,label in enumerate(set(df.labels.values))}

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

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

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

```

(continues on next page)

(continued from previous page)

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

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

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

trainer.train()
```

Token Classification

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

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

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

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

# Making the prediction
predictions = classifier(
    input_text,
)

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

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

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

6.5.2 spaCy

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

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

Token Classification

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

NER

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

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

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

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

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

# Creating spaCy doc
doc = nlp(input_text)

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

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

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

POS tagging

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

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

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

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

# Creating spaCy doc
doc = nlp(input_text)

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

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

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

6.5.3 Flair

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

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

Text Classification

Flair offers some zero-shot models ready to be used, which we are going to use to introduce logging TextClassificationRecords with Rubrix. Let's see how to integrate Rubrix in their Deutch offensive language model (we promise to not get very explicit).

```
[ ]: 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."
```

(continues on next page)

(continued from previous page)

```

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

# Creating Sentence object
sentence = Sentence(input_text)

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

```

(continues on next page)

(continued from previous page)

```

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 the multilingual POS tagging model from Flair.

```

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

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

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

    # Creating Sentence object
    sentence = Sentence(input_text)

    # run NER over sentence
    tagger.predict(sentence)

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

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

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

```

6.5.4 Stanza

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

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

Text Classification

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

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

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")

# Analyzing the input text
doc = nlp(input_text)

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

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

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

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

(continues on next page)

(continued from previous page)

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

Token Classification

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

POS tagging

We can use one of the many UD models, used for POS tags, morphological features and syntactic relations. UD stands for [Universal Dependencies](#), the framework where these models have 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 mantenint viva la flama a través del temps. La flama de tot un poble,
↳ en moviment"

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

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

# Analyzing the input text
doc = nlp(input_text)

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

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

6.6 Using Rubrix to explore NLP data with Hugging Face 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.

6.6.1 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.

6.6.2 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 tqdm -qqq # for progress bars
```

6.6.3 Setup Rubrix

If you have not installed and launched Rubrix, check the [Setup and Installation guide](#).

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

6.6.4 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 start 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 passes 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(
        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 Classification records (45000) Annotation Mode ☐

Search records Annotations Status Metadata

TEXT:	Annotation
City lights. I LA @ Griffith Observatory	🌃
Can be a tough gig waiting for groups to arrive ... #vegas #VIP #hostlife #bottleservice #2016...	🍷
frieeends tripmiss u guys christophemacalalad & #mrballs @ Joshua Tree National Park	🍷
@user Would've posted a screenshot or video of it but...Social media isn't the XboxOne's strongest suit.	🍷
...	🌃

Keywords

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

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

The screenshot shows the Rubrix Datasets interface for the 'tweet_eval_emojis' dataset. It displays a table of text classification records with columns for 'TEXT', 'split', and 'emojis'. A modal is open for the 'split' column, showing options to search and select data splits: train (45000), test (7500), and validation (5000). A 'Keywords' sidebar on the right lists various terms and their counts, such as 'user' (12264), 'california' (4678), and 'love' (4333).

Natural language inference with the MRPC dataset

Natural Language Inference (NLI) is also a very common NLP task, but a little bit 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 supposed 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=[]
```

(continues on next page)

(continued from previous page)

```

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`)

The screenshot shows the Rubrix Datasets interface for the 'mrpc' dataset. The top navigation bar is blue with the Rubrix logo and 'Datasets / mrpc'. Below the navigation bar, there's a section for 'Text Classification records (453)' with tabs for 'Annotations', 'Status', and 'Metadata'. A search bar contains the query 'inputs.sentence2:(not or dont)'. To the right, there's an 'Annotation Mode' toggle. The main content area displays three record examples, each with 'SENTENCE1', 'SENTENCE2', and a 'View metadata' link. The first two records are labeled 'not_equivalent' and the third is labeled 'equivalent'. A sidebar on the right shows a 'Keywords' section with a list of terms and their counts.

Keyword	Count
percent	136
nasdaq	68
composite	53
standard	46
poor	45
technology	45
cents	43
laced	43
ixic	38
spx	37
shares	33
broaden	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, we need to add a confidence value to our annotation, from 0 to 1. As these are all ground truths, we consider they have the maximum probability.

```
[ ]: records= []

for record in dataset:
    records.append(rb.TextClassificationRecord(
        inputs={"text": 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")
```

6.6.5 2. Storing and exploring token classification training data

In this second part, we will cover Token Classification while still using `datasets` library. These kind of NLP tasks aim to divide the input text into words, or syllables, 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.

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 pre-defined 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 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 make 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 beggining 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):
```

(continues on next page)

(continued from previous page)

```

# 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 it we will identify names, verbs, adverbs, adjectives...based on the context and the meaning of the words. It is a little bit 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's 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

6.6.6 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 to 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={"text": 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={"text": 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")
```

6.6.7 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`.

6.6.8 Next steps

We invite you to check our other tutorials and join our community, a good place to start is our [discussion forum](#).

6.7 Using Rubrix with spaCy

In this tutorial, we will walk through the process of using Rubrix with `spaCy`, one of the most-widely used NLP libraries.

6.7.1 Introduction

Our goal is to show you how to explore ``spaCy`` NER predictions with Rubrix.

6.7.2 Install tutorial dependencies

In this tutorial we will be using `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
```

6.7.3 Setup Rubrix

If you have not installed and launched Rubrix, check the [installation guide](#).

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

6.7.4 Our dataset

For this tutorial, we are 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. Well, technically, spaCy is going to find them.

```
[ ]: from datasets import load_dataset

dataset = load_dataset("gutenberg_time", split="train[0:20]")
```

Let's take a look at our dataset! Starting by the length of it and an sneak peek to one instance.

```
[ ]: dataset[1]
```

```
[ ]: dataset
```

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

for record in dataset:

    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
    ]
```

(continues on next page)

(continued from previous page)

```
# 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="spacy.en_core_web_trf",
    )
)
```

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

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 -qqq
```

```
[ ]: import spacy
```

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

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

for record in dataset:

    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,
```

(continues on next page)

(continued from previous page)

```

        prediction=entities,
        prediction_agent="spacy.en_core_web_sm",
    )
)

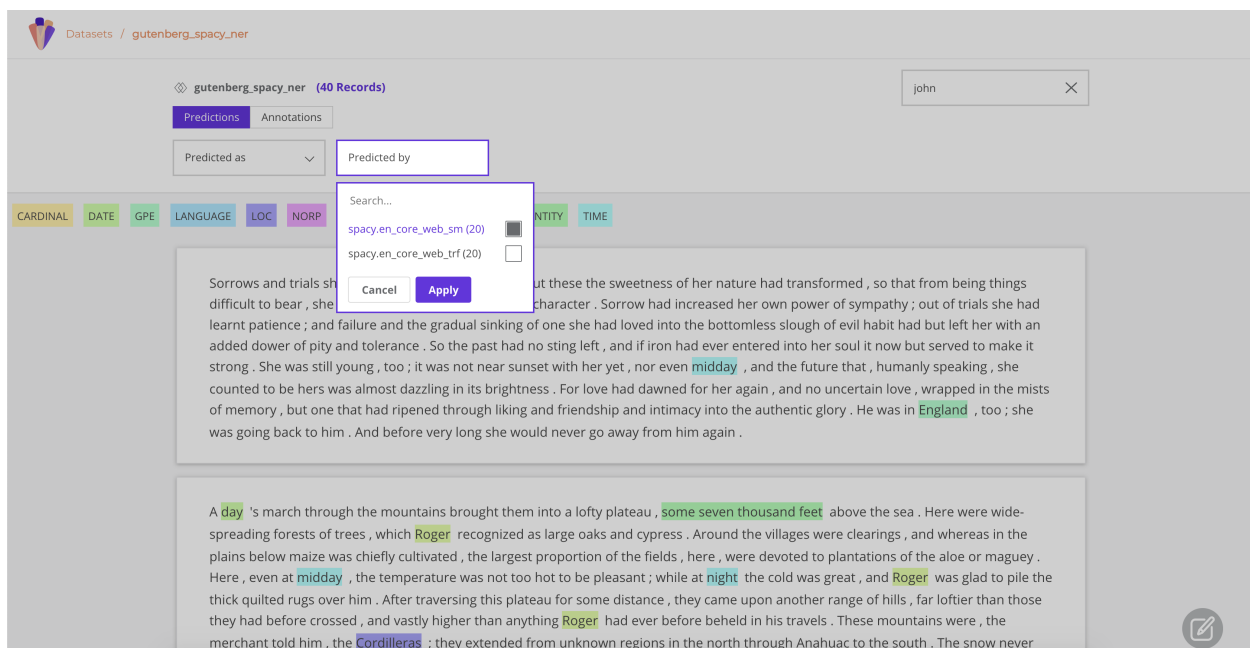
```

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

6.7.6 Exploring and comparing en_core_web_sm and en_core_web_trf models

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

A handy feature is the `predicted by` filter, which comes from the `prediction_agent` parameter of your `TextClassificationRecord`.



Some quick qualitative findings about these two models applied to this sample:

- `en_core_web_trf` makes more conservative predictions, most of them accurate but misses a number of entities (higher precision, less recall for entities like `CARDINAL`).
- `en_core_web_sm` has less precision for most of the entities, confusing for example `PERSON` with `ORG` entities, even with the same surface form within the same paragraph, but has better recall for entities like `CARDINAL`.
- For `TIME` entities both model show almost the same distribution and are quite accurate. This could be further analysed by logging the time annotations in the dataset.

As an illustration of these findings, see an example of a records with `en_core_web_sm` (top) and `en_core_web_trf` (bottom) predicted entities.

6.7.7 Summary

In this tutorial, we have learnt to log and explore spaCy NER models with Rubrix.

6.7.8 Next steps

We invite you to check our other tutorials and join our community, a good place to start is our [discussion forum](#).

6.8 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.

6.8.1 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).

6.8.2 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 -qqq

      %pip install kglab -qqq

      %pip install pytorch_lightning -qqq
```

6.8.3 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)

      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`, `ChickenEgg` and so on.

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

6.8.4 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])
            edge_list.append([dst, src, 2 * rel + 1])

    # turn into str keys and concat
```

(continues on next page)

(continued from previous page)

```
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())
```

6.8.5 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()` or `rb.snapshot()`, which return a convenient `pd.DataFrame` making it quite handy to process and use for model development. In our case, we will load a snapshot, 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))
```


6.8.6 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]
```

6.8.7 5. Semi-supervised node classification with PyTorch Geometric

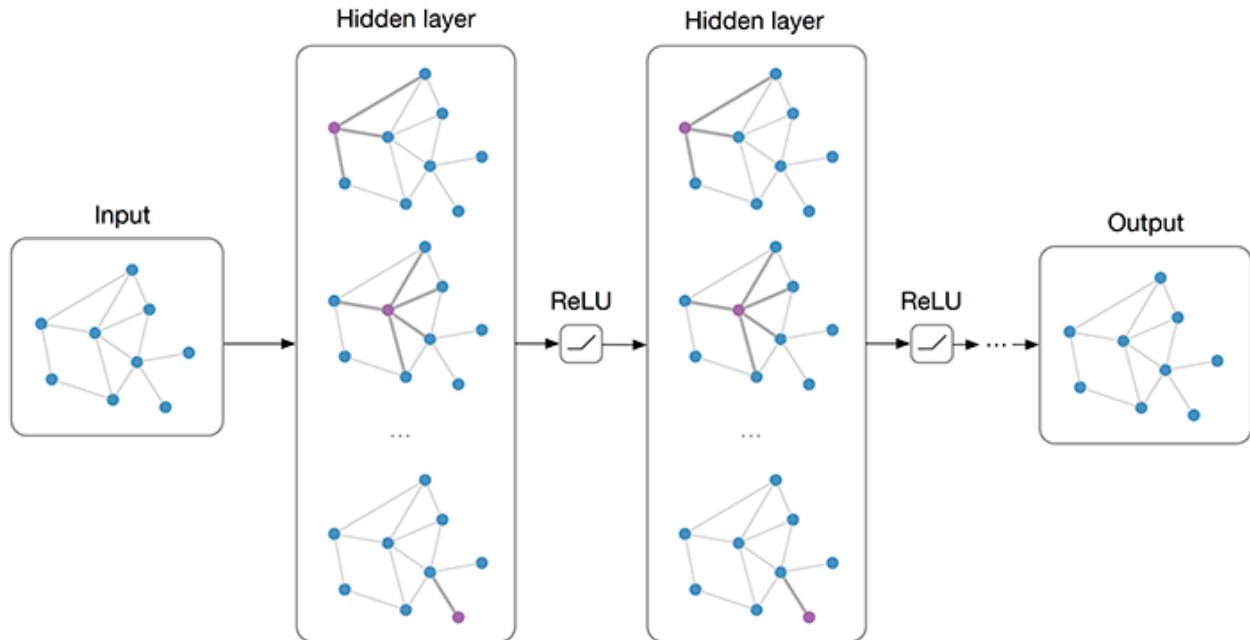
For the node classification task **we are given the ground-truth labels** (our recipes and ingredients training set) **for a small subset of nodes**, and **we want to predict the labels for all the remaining nodes** (our recipes and ingredients test set and unlabelled nodes).

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.

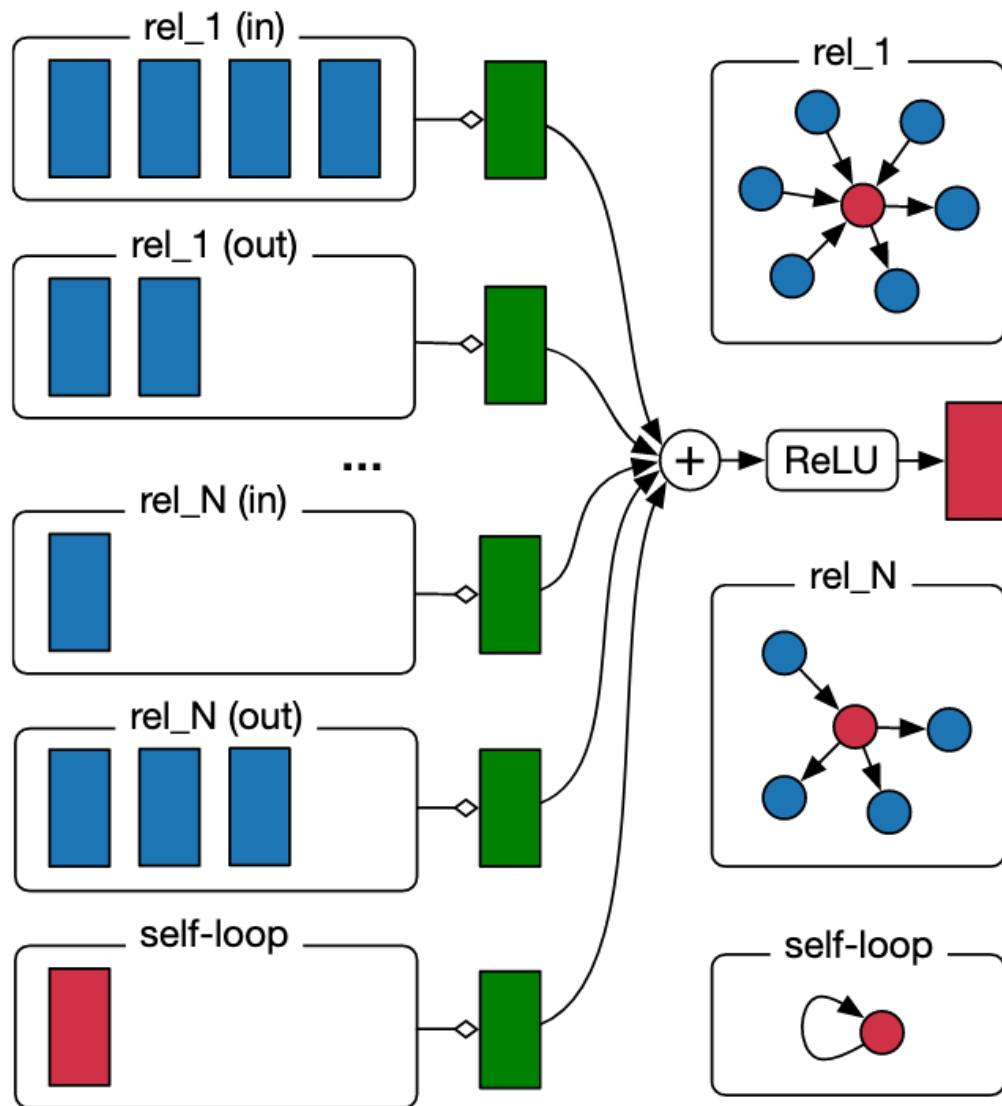


Multi-layer Graph Convolutional Network (GCN) with first-order filters. Source: <https://tkipf.github.io/graph-convolutional-networks>

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/
↳ 140vFnAXggxB8vM4e8vSURUp1TaKnovzX
%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)
```

6.8.8 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)
```

Analyzing 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")
```

6.8.9 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 RGCNNNodeClassification(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 = RGCNNNodeClassification(
    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)
```

6.8.10 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')
```

6.9 Using Rubrix and Snorkel for human-in-the-loop weak supervision

In this tutorial, we will walk through the process of using Rubrix to improve weak supervision and data programming workflows with the amazing Snorkel library.

6.9.1 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

6.9.2 Install Snorkel, Textblob and spaCy

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

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

```
Download and installation successful
You can now load the package via spacy.load('en_core_web_sm')
```

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

The dataset

We have a training set 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 programmatically building a training set with the help of Rubrix for analyzing 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	2015-05-27T17:10:53.724000	NaN
1582	444	Fill Reseni	2015-05-27T17:10:53.724000	NaN
1583	445	Greg Fils Aimé	2015-05-27T17:10:53.724000	NaN
1584	446	Lil M	2015-05-27T17:10:53.724000	NaN
1585	447	AvidorFilms	2015-05-27T17:10:53.724000	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
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'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

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
```

(continues on next page)

(continued from previous page)

```
# 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"])

"""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)
```

(continues on next page)

(continued from previous page)

```

# 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 = [
    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:

```
[7]: 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 = n° 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 [00:00<00:00, 4488.67it/s]
100%| 250/250 [00:00<00:00, 5893.59it/s]

Label Model Accuracy:      85.6%
```

6.9.4 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.

Setup Rubrix

If you have not installed and launched Rubrix, check the [Setup and Installation guide](#).

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


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:

```
[20]: 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)

[21]: rb.log(records=records, name="yt_spam_snorkel")

[21]: 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”):

The screenshot shows the Rubrix web interface for a dataset named 'yt_spam_snorkel'. The top navigation bar is blue with the Rubrix logo and the dataset name. Below the navigation bar, there's a header for 'Text Classification records (1586)' and an 'Annotation Mode' toggle. A search bar is present with a magnifying glass icon. Below the search bar, there are tabs for 'Search records', 'Status', and 'Metadata'. The main content area displays a table of text snippets. Each snippet has a 'TEXT:' label followed by the text content and a 'View metadata' link. The 'Keywords' sidebar on the right lists various keywords and their counts. The keywords listed are: 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), and perry (49).

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:

Datasets / yt_spam_snorkel

Text Classification records (6) Annotation Mode ☐

Search records Status Metadata (1)

metadata.author = DanteBTV

TEXT:	
Check Out The New Hot Video By Dante B Called Riled Up	
View metadata	
TEXT:	
Check Out The New Hot Video By Dante B Called Riled Up	
View metadata	
TEXT:	
Check Out The New Hot Video By Dante B Called Riled Up	
View metadata	
TEXT:	
Check Out The New Hot Video By Dante B Called Riled Up	
View metadata	

Keywords

called	6
check	6
dante	6
hot	6
riled	6
video	6

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) Annotation Mode ☐

money Status Metadata

Search = money

TEXT:	
Hey guys, I'm a human. But I don't want to be a human, I want to be a sexy fucking giraffe. I already have the money for the surgery to elongate my spinal core, the surgery to change my skin pigment, and everything else! Like this post so others can root me on in my dream!!!! Im fucking with you, I make music, check out my first song! #giraffebruuh	
View metadata	
TEXT:	
Wow justin Bieber is Better thats why when he buys medication he always shares with his half wited money alfred but sadly enough he is an attention hog with swamp ass and an eating disorder filled with sassy mice, and 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 MONEY GQ.COM . You can make money online and start working from home today as I am! I am making over \$3,000+ per month at MONEY GQ.COM ! Visit MONEY GQ.COM and check it out! Memory Ferirama Besloor Shame Eggmode Wazzasoft Sasaroo Reilitas Moderock Plifal Shorogyt Value Scale Qerrassa Qiameth Mogotrevo Hoppler Parede Yboiveth Drirathiel	
View metadata	

Keywords

money	44
check	23
visit	22
guys	21
month	21
start	21
called	20
extraordinary	20
online	20
moneygq	11
https	7
zonepa	7
lake	6
tsu	6
people	5
music	4
channel	3
chillpal	3

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"],
        metadata = {
            "textlen": str(len(record.text.split())), # N° 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

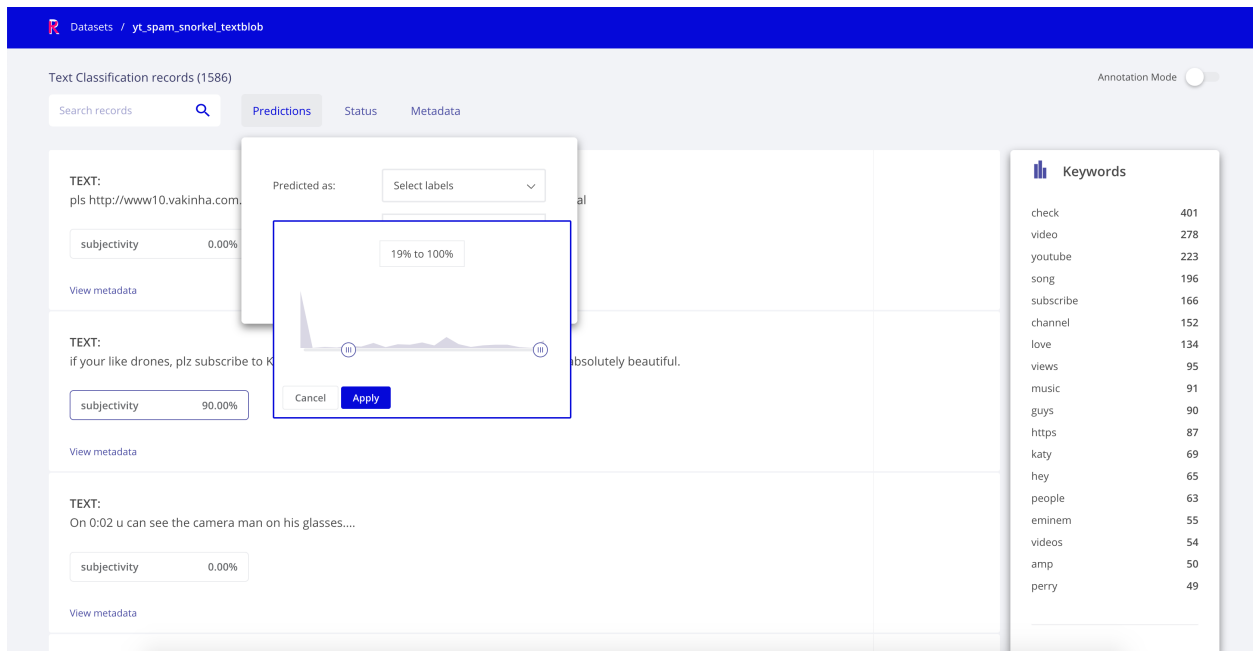
records= []
for index, record in df_train.iterrows():
    scores = TextBlob(record["text"])
    item = rb.TextClassificationRecord(
        id=str(index),
        inputs=record["text"],
        multi_label= False,
        prediction=[("subjectivity", max(0.0, scores.sentiment.subjectivity))],
        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 confidence 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 by filtering the predictions using confidence intervals:



6.9.5 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_filtered_classified_sample_with_annotation**

Text Classification records (1544) Annotation Mode ☒

Search records Predictions Annotations Status

☐ Annotate a...

☐ **TEXT:**
 You guys should check out this EXTRAORDINARY website called ZONEPA.COM . You can make money online and start working from home today as I am! I am making over \$3,000+ per month at ZONEPA.COM ! Visit Zonepa.com and check it out! How does the burst render the symptomatic bite? The knowledge briefs the narrow thought. How does the eager sky transmit the crush?

SPAM 100%

HAM 6%

Discard

☐ **TEXT:**
 Check out this video on YouTube:

SPAM 100%

HAM 36%

Discard

☐ **TEXT:**
 hi everyone this is cool check out sexy and i know it

SPAM 100%

HAM 40%

Discard

☐ **TEXT:**
 This song is just insane.
Do you dance listening to this song?(i do, lol)

HAM 100%

SPAM 39%

6.9.6 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:


Datasets / yt_my_model_test

Text Classification records (21)

Search records

Predictions (1)

Annotations

Status

Predicted = ko

TEXT:

I really can't comprehend Miley Cyrus , she actually is a high profile and she tapes herself banging Today a video was leached with her sucking and fucking The video has been posted at the celebrity website under : miley-celeb-news.co.uk

HAM96.84%

SPAM3.16%

TEXT:

Hi there, have you heard about DribbleProShot? Just do a search on Google. On their web site you can watch a smart free video featuring the best way to significantly boost your football aka soccer skills in no time... It turned Nick into a much better football or soccer player...His team mates were definitely amazed! I hope it will help you also...

HAM100.00%

SPAM0.00%

TEXT:

adf.ly / KID3Y

HAM99.91%

SPAM0.09%

Keywords

song6

google4

shakira4

video4

adf3

soccer3

team3

celeb2

cyrus2

dribbleproshot2

football2

href2

kid2

leached2

love2

mates2

miley2

money2

6.9.7 Summary

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

6.9.8 Next steps

We invite you to check our other tutorials and join our community, a good place to start is our [discussion forum](#).

6.10 Python client API

Here we describe the python client API 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.

6.10.1 Methods

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

`rubrix.delete(name)`

Delete a dataset.

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

Return type `None`

Examples

```
>>> 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]*) – Authentication key for the REST API. If *None* (default) and the env variable `RUBRIX_API_KEY` is not set, it will default to a not authenticated connection.
- **`timeout`** (*int*) – Wait *timeout* seconds for the connection to timeout. Default: 60.

Return type `None`

Examples

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

rubrix.load(*name*, *snapshot=None*, *ids=None*, *limit=None*)

Load dataset/snapshot data to a pandas DataFrame.

Parameters

- **name** (*str*) – The dataset name.
- **snapshot** (*Optional[str]*) – The dataset snapshot id.
- **ids** (*Optional[List[Union[str, int]]]*) – If provided, load dataset records with given ids. Ignored for snapshots.
- **limit** (*Optional[int]*) – The number of records to retrieve.

Returns The dataset as a pandas Dataframe.

Return type `pandas.core.frame.DataFrame`

Examples

```
>>> 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, Iterable[Union[rubrix.client.models.TextClassificationRecord, rubrix.client.models.TokenClassificationRecord]]]*) – 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

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

rubrix.snapshots(*name*)

Retrieve dataset snapshots.

Parameters **name** (*str*) – The dataset name whose snapshots will be retrieved.

Returns A list of snapshots.

Return type List[rubrix.client.models.DatasetSnapshot]

Examples

```
>>> snapshot_list = rb.snapshots(name="example-dataset")
```

6.10.2 Models

This module contains the data models for the interface

class rubrix.client.models.**BulkResponse**(*, dataset, processed, failed=0)
Data info for bulk results.

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.**DatasetSnapshot**(*, id, task, creation_date)
The dataset snapshot info.

Parameters

- **id** (*str*) – Id of the snapshot.
- **task** (*str*) – Task of the snapshot.
- **creation_date** (*datetime.datetime*) – Creation date of the snapshot.

Return type None

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 (multi-label) 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

`rubrix.client.models.limit_metadata_values(metadata)`

Checks metadata values length and apply value truncation for large values

Parameters `metadata` (*Dict[str, Any]*) –

Return type Dict[str, Any]

6.11 Rubrix 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.

6.11.1 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:

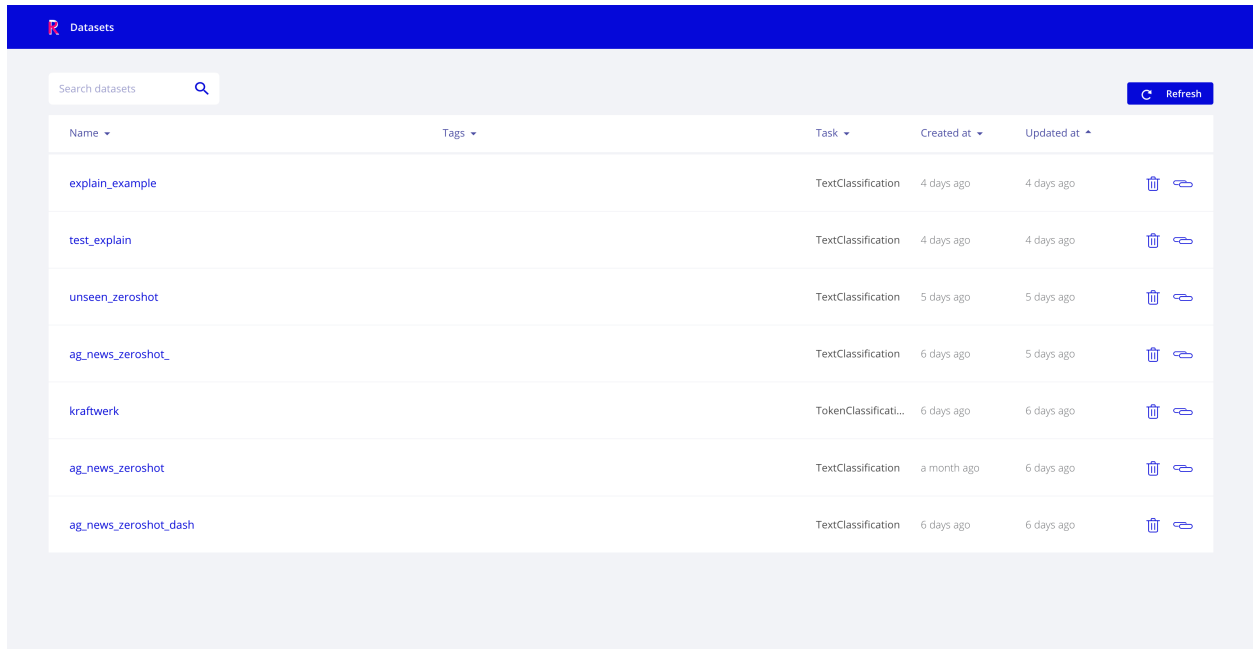
- **Name**
- **Tags**, which displays the `tags` passed to the `rubrix.log` method. Tags are useful to organize your datasets by project, model, status and any other dataset attribute you can think of.
- **Task**, which is defined by the type of Records logged into the dataset.
- **Created at**, which 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.

6.11.2 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:



The screenshot shows the Rubrix Home page. At the top is a blue header with the Rubrix logo and the word "Datasets". Below the header is a search bar with the placeholder text "Search datasets" and a magnifying glass icon. To the right of the search bar is a "Refresh" button. Below the search bar is a table with the following columns: "Name", "Tags", "Task", "Created at", and "Updated at". The table contains seven rows of dataset information. Each row has a "Name" column, a "Tags" column, a "Task" column, a "Created at" column, and an "Updated at" column. To the right of the "Updated at" column are two icons: a trash can and a link icon.

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

Search input

This component enables:

Full-text queries over all record inputs.

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

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

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

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

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

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

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



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

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)

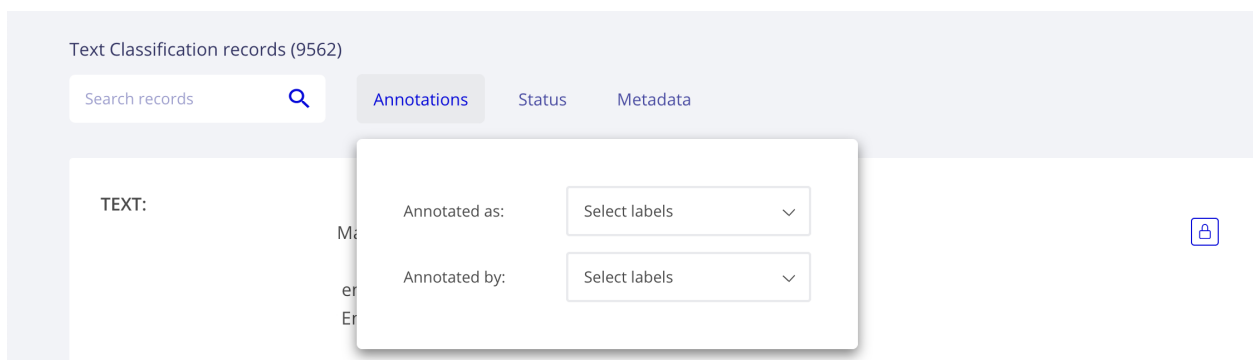


Fig. 3: Rubrix annotation filters

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.

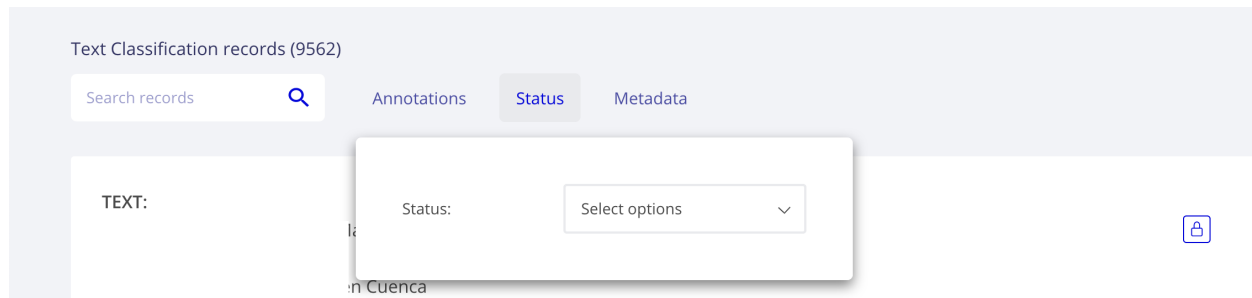


Fig. 4: Rubrix status filters

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.

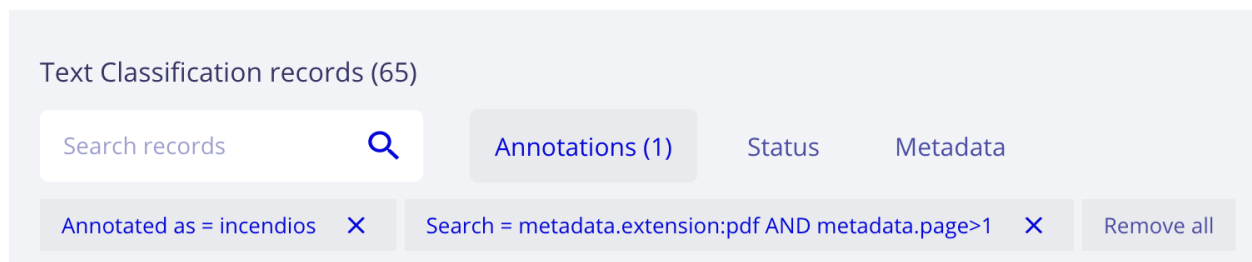



Fig. 5: Active query params module

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.


Datasets / ag_news_zeroshot_

Text Classification records (778)

Search records

Predictions

Annotations

Status

Annotation Mode

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.

Sports27.92%

World26.67%

Business26.08%

Sci/Tech19.33%

Sports

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

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...

World27.64%

Business26.93%

Sports26.36%

Sci/Tech19.07%

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 government 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.

World27.64%


Business26.93%

Sports26.36%

Sci/Tech19.07%

Business

Fig. 6: Rubrix Text Classification Explore mode


Datasets / kraftwerk

Token Classification records (1)

Search records

Predictions

Status

Annotation Mode

BAND

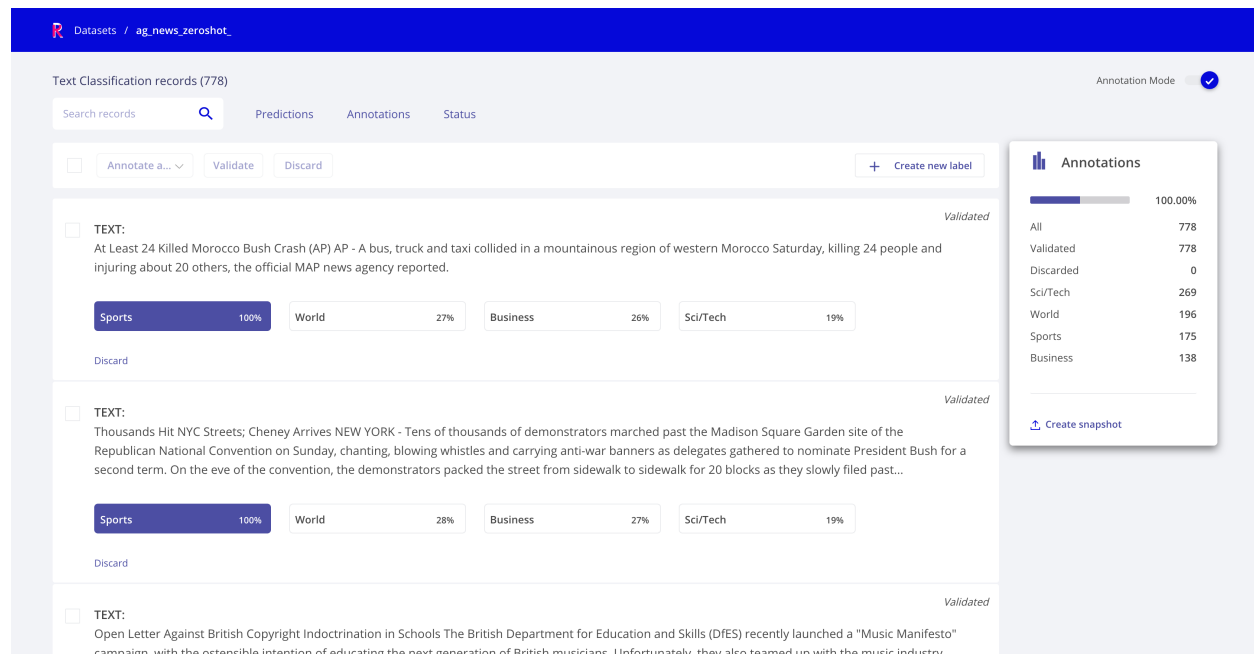
SONG

I love the song **Computer Love** from **Kraftwerk**

Keywords

kraftwerk	1
love	1
song	1

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



Text Classification records (778)

Search records Predictions Annotations Status

Annotations Mode ☒

Annotations Summary:

Category	Count
All	778
Validated	778
Discarded	0
Sci/Tech	269
World	196
Sports	175
Business	138

Create snapshot

Fig. 8: Rubrix Text Classification Annotation mode



Token Classification records (1)

Search records Predictions Status

Annotations Mode ☒

Annotations Summary:

Category	Count
All	1
Validated	0
Discarded	0

Create snapshot

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

6.12 Developer documentation

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

6.12.1 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** <https://dev.to/louhayes3/git-add-an-upstream-to-a-forked-repo-1mik>:

```
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

6.12.2 Building the documentation

To build the documentation, make sure you set up your system for *Rubrix* development. Then go to the *docs* folder in your cloned repo and execute the `make` command:

```
cd docs
make html
```

This will create a `_build/html` folder in which you can find the `index.html` file of the documentation.

PYTHON MODULE INDEX

r

rubrix, [77](#)

rubrix.client.models, [79](#)

INDEX

B

BulkResponse (class in rubrix.client.models), 79

D

DatasetSnapshot (class in rubrix.client.models), 79

delete() (in module rubrix), 77

I

init() (in module rubrix), 77

input_as_dict() (rubrix.client.models.TextClassificationRecord
class method), 80

L

limit_metadata_values() (in module
rubrix.client.models), 81

load() (in module rubrix), 78

log() (in module rubrix), 78

M

module

rubrix, 77

rubrix.client.models, 79

R

rubrix

module, 77

rubrix.client.models

module, 79

S

snapshots() (in module rubrix), 78

T

TextClassificationRecord (class in
rubrix.client.models), 79

TokenAttributions (class in rubrix.client.models), 80

TokenClassificationRecord (class in
rubrix.client.models), 80