# MetRuBert

# Documentation

Joost Grunwald
Radboud University

If you come across any problems, see section ?? for possible solutions or contact me at joost.grunwald@ru.nl

# Table of Contents


**1 Introduction to MetRuBert** **3**

1.1 Using MetRubert online . . . 3

1.2 Source Code . . . 3

1.3 Paper . . . 3

**2 Using MetRuBert online** **4**

2.1 Uploading files . . . 4

2.1.1 Non tokenised text files . . . 4

2.1.2 Tokenised text files . . . 4

2.1.3 Alpino xml files . . . 5

2.1.4 Dev tsv files . . . 5

2.2 Pos Tags . . . 5

2.3 File Cleaning . . . 5

2.4 Titles and metadata . . . 5

2.5 Pipeline . . . 6

2.6 Future work . . . 6

**3 the MetRuBert Model** **7**

3.1 data_loader.py . . . 7

3.2 main_config.cfg . . . 7

3.3 run_classifier_dataset_utils.py . . . 8

3.4 modeling.py . . . 8

3.5 main_dutch.py . . . 8

3.5.1 main . . . 8

3.5.2 run_train . . . 9

3.5.3 run_eval . . . 9

3.5.4 run_dev . . . 9

3.6 main_dutch_optima.py . . . 9

3.7 main_english.py . . . 9

3.8 scripts . . . 9

3.9 data_sample . . . 10

**4 CLAM webservice** **11**

4.1 metaphorclam.py . . . 12

4.2 metaphorclam_wrapper.py . . . 12

4.3 outputgenerator.py . . . 12

4.4 pasmaparserclam.py . . . 12

4.5 style_joost.css . . . 12

**5 Parsers and Data Creation** **13**
5.1 Pasmaparser_allpos . . . 13
5.2 Pasmaparser_verb . . . 13
5.3 Covidparser . . . 13
5.4 Train/Test Split . . . 14
5.5 pasma corpus . . . 14

**6 Covid dataset** **15**
6.1 Data choices . . . 15
6.2 Data scraping . . . 15
6.3 Dataset generation . . . 16
6.4 Corona filtering . . . 17
6.5 Data iteration and generation . . . 17

**7 Using the webserver** **19**

**About This File**

This file was created for the benefit of all users that want to use our model, use the source code of our model or who want to add support for additional languages.
The entirety of the contents within this file, and folder, are free for public use.

SECTION 1

# Introduction to MetRuBert

MetRuBert is short for Metaphor Detection with RuBert. RuBert is our Radboud University variant of the Linguistic Dutch MetRobert model. MetRubert is able to automaticly identify metaphors in dutch. It uses ucto to tokenize input texts and alpino as POS tagger and possible future supplier of additional input data. Our own model then does the actual predictions of metaphoricity.

## 1.1 Using MetRubert online

MetRubert is hosted as CLAM webservice on `https://webservices.cls.ru.nl` here you can upload either tokenised or untokenised sentences for processing and metaphor authentication. The tokenised input also allows direct uploading of dev.tsv files or alpino .xml files that are generated by earlier runs of the model. The output is a output.tsv file that gives you the corresponding sentence and word together with POS TAG and prediction.

## 1.2 Source Code

If interested in obtaining the source code for our model, see:
`https://github.com/joostgrunwald/MetRoBert`
Our github contains more then the actual model, such as lots of parsers and code for the CLAM webservice, we will further describe its content in ....

## 1.3 Paper

TODO

SECTION 2

# Using MetRuBert online

As briefly discussed already in the introduction, MetRubert is hosted as CLAM webservice on https://webservices.cls.ru.nl. When you want to use MetRubert, you might have to make a free account on here to be able to do so. You can do so by clicking "create a free account here" on the homepage of the webservices url.

## 2.1 Uploading files

The web-service is designed to allow different kind of inputs for the MetRubert model. Some of them are clearly described inside the web-service itself and some are a little bit more ambiguous and made for repetitive users. The service is capable of processing both single big files containing multiple texts as lots of smaller texts in seperate files.

### 2.1.1 Non tokenised text files

A user can upload non tokenised text files, files uploaded should be of .txt format and should be texts. Because MetRubert is trained on news data, it will provide the best results when given these news texts in .txt format. The model will then tokenise the sentence itself and will put the files trough the pipeline described later in this section.

### 2.1.2 Tokenised text files

If a user has text files that are already tokenised, the user can directly upload these .tok files into the model. The files will then be put into alpino in the pipeline similar as if they were just tokenised by our own software. Note that because of the fact that our models requires space

separated words this option might be a bit more buggy then the non tokenised version where we apply tokenization our self.

### 2.1.3 Alpino xml files

Using alpino on the input files is the most time consuming part of our pipeline. In the future it is hence important that if we do not use extra alpino we switch to a different method for POS tagging. In the meantime, however, you can use alpino .xml files that were generated as input for the model. This way alpino does not have to be ran again and the speedup is immense. Note that this is not made with the idea of using alpino parsed output, only with the idea of reusing alpino output we generated ourselves in this model.

### 2.1.4 Dev tsv files

The alpino files get used by my custom parsers to generate a dev.tsv file, this file contains sentences, words and pos tags from alpino. The model that is trained and tested already then runs on the dev data to generate new predictions that we output to the user. This dev.tsv file can be reused to quickly run the model again. It is the last form of input we accept.

## 2.2 Pos Tags

Our model is designed to predict metaphors in text for a huge variety of POS tags. Metaphors can be implicit or more on the edge where its possible to discuss whether something is a metaphor or not. Predicting metaphor is more implicit or less obvious for some POS tags hence we allow the user to select which POS tags to predict metaphoricity for. We can do this by using the checkboxes provided on the webservice.

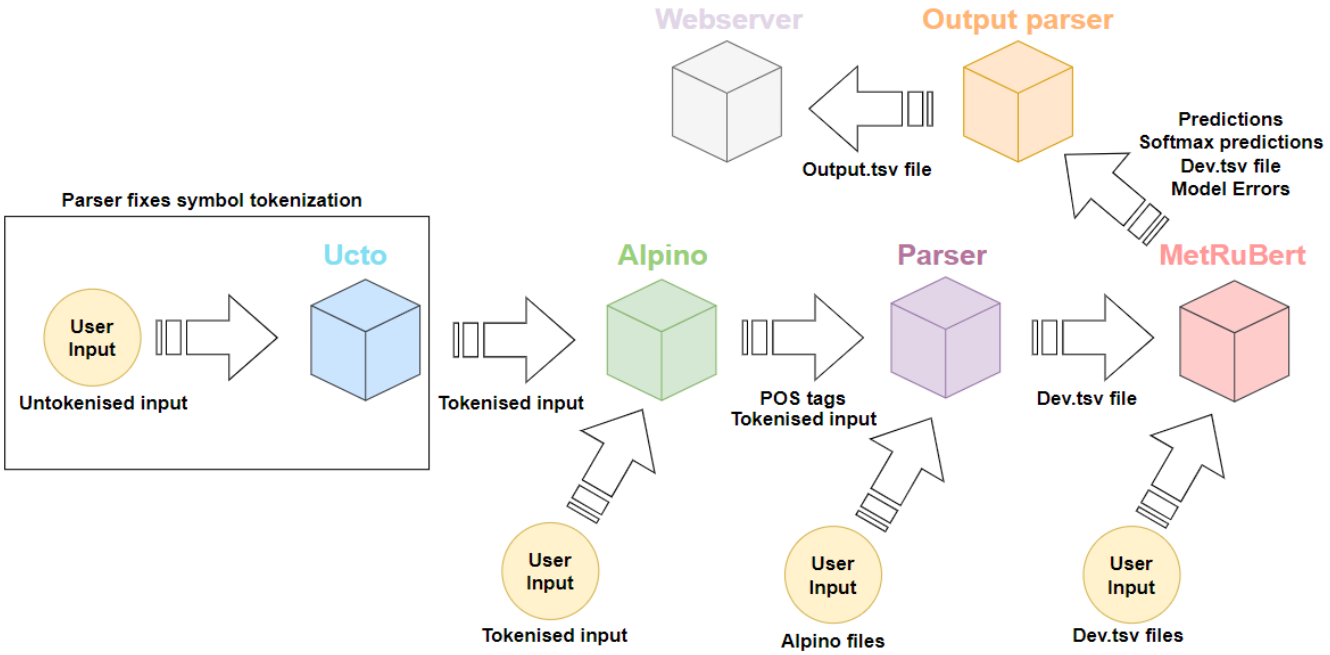
## 2.3 File Cleaning

The nature of the input for our model is in contrast with the nature of the input for alpino. Alpino needs tokenised input where our models expects words in a sentence to only be space separated. Hence symbols that are tokenised can mess up the model input because it expects only words. We use our own parser to adjust these tokenised symbols spacing. In the future we might want to apply a different parser for this. For now we handle this situation with our own parser. Any problems found in the parsing end up as ERROR inside of the output.

## 2.4 Titles and metadata

When putting text files trough our model, the sentences are separated by punctuation symbols and points. When using texts with metadata and titles this can mean that this metadata or the title ends up included in the sentences of our model. Try to clean and filter texts before parsing them trough our model.

## 2.5 Pipeline




Here we show our pipeline, where the user input spots stand for direct user input into the webserver. Parsers are used for the output, the input and for stopping errors with symbol tokenisation.

## 2.6 Future work

In the future we want to possible replace Alpino with a faster POS tagger or use information from alpino as extra features for our model. We also might want to write our own tokenizer to better separate texts into sentences usable for our model.

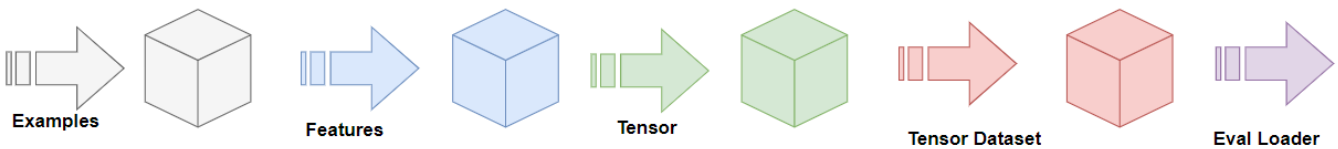
SECTION 3

# the MetRuBert Model

MetRubert is not only deployed as webservice. The code can be found at ... . It is also currently available on /vol/tensusers4/jgrunwald/MetRobert_run2 on Ponyland. We are going to describe the sub parts of the model and what they do.

## 3.1 data_loader.py

The data loader uses functions to select the right features and model and loads it based upon what model is used. First we get the training examples. We then use a function to convert the training eamples into features. The file makes tensors from features and then uses the tensor to create a TensorDataset. We use randomsampling on the tensordataset, then we create a dataloader.




## 3.2 main_config.cfg

The main config file is the settings file. It is used in the code multiple times to get the settings for the model. You can easily adjust the file to update the settings and the model wil rerun with different parameters. Note that most things in here are hyperparameters, they are used in hyperparameter optimalization.

## 3.3 run_classifier_dataset_utils.py

This file contains multiple classes. All classes exists of their own functions for the model associated with the class. The vua class here is used for the english vua model and for the dutch model. There are settings specified for using ponyland or using the model local. Some of the urls are still hardcoded, I plan to replace them with more flexible url parsing in the future. The important ones (used in ponyland) are already adjusted to be flexible. The functions to generate features from the training data are also used here, some debugging of data is done inside here. During testing for dutch we found some problems leading to crash, we fix this at the moment by skipping inputs that lead to errors. Our outputparser then catches those missing input variables and shows them as -1 predictions. The problems occurring here are very limited, during development it seems to happen 10 out of 38000 times. We write any problems to wrong_devs.txt. We also take the functions for model evaluation, such as accuracy, precision and recall here.

## 3.4 modeling.py

This file contains multiple classification classes. Each model type has its own classification class. So MIP has one, MIP SPV has one and MELBERT has once for example. All classes contain an __init__ constructor, an weight initatior and a forward method. Note that some model settings are taken from main_config here. We added some extra classes for the optima model, adding more is just copy past and adjusting.

## 3.5 main_dutch.py

This file is the main file you use when using the model. When you run the model on ponyland you run it via a command. You can run the file via 'nice -19 python3 main_dutch.py'. It is important that when using ponyland you add niceness, for more details check the documentation of ponyland. We added some global parameters to display more/less outputs from the model.

### 3.5.1 main

Executing the python fail executes the main() function, this function can either run an already trained model on new dev data or do training and testing from scratch. If keyword "saves" is found inside the model specifier in main config, we resume we are running dev data. Note that there should be an URL inside the model that uses the saves folder inside your model. Therefore we can check it. The model is designed to work best when using a GPU, if a GPU is not present a CPU can also be used. Note that the big batch size also needs quite a lot of RAM and a good GPU.

Another important part is the tokenizer, when applying this model to another language you should specify another roberta model here. After this we have different if else loops that run different code based upon the model used and whether or not to run bagging. The same happens for testing. The main then runs train and eval functions.

### 3.5.2 run_train

Run train does exactly what the name tells you it does, it handles training of the model. It specifies the optimizer, which is quite an interesting topics. In further research we could further examine which optimizer yields the best results for our setup. Run train then works with different epochs, as specified in main_config. For each epoch we train, adjusting the learning rate for optimal training. A training loss is calculated, it is often a very good indication of how good the model will score. Dev and eval functions are also called from this function.

### 3.5.3 run_eval

Run eval runs the trained model on the test data. When comparing the output of the model with the actual values we use this to calculate some metrics about the model. The function also is able to return the predictions for the test data. We go in more depth about these test predictions and provide an analysis of weak spots in a later chapter.

### 3.5.4 run_dev

Run dev runs the trained and tested model on new data. Data for which we want predictions. It outputs a combination of GUIDS and predictions. To be able to later attach the predictions to the right inputs. The predictions are floats, we then apply soft-max and then make them binary. The output of all these operations are printed in several files and can be seen as output.

## 3.6 main_dutch_optima.py

Main dutch optima is very similar to the original main dutch. The difference is that this variant is updated in multiple ways to enable users to do automatic hyper-parameter optimization. We have added options to automatically tweak multiple parameters and also to run cycles of custom hyper-parameter inputs. Using booleans, we can simply specify to either use this manual mode or the automatic mode. Using booleans we can also tell which hyper-parameters to optimize, if wanted simultaneously. The amount of models to try can be easily adjusted by adjusting the amount of optuna trials. The file is ran in the same way as main dutch is ran. The manual mode and automatic mode both speak for themselves and should not be hard to use.

## 3.7 main_english.py

Main English is still quite similar to the original MelBert model, for more documentation on that one should check that repository on GitHub.

## 3.8 scripts

The scripts included with the model can be used to run the model. (run.sh is an example of how to easily run the model.) The other one runs multiple bagging instances to see which bagging settings deliver us the best model.

## 3.9 data_sample

The data sample folder contains the data for our model. The dutch allpos model is under pasma_all, the dutch verb model is under pasma_verb. The other folders are all for the english model. We deliver train.tsv, test.tsv and possible dev.tsv data here.

4

SECTION

# CLAM webservice

MetRobert consists of different components.
You have the Metaphorclam folder, which is the main folder and a wrapper for the actual model with a LAMachine environment https://github.com/proycon/LaMachine.
In this folder, there are scripts and things for the server, such as startserver_development.sh, which allows you to start the LAMachine server. In this folder, there is also a userdata folder under projects/anonymous. This folder contains the runs that we have already done through the webserver.
There is also a new metaphorclam folder in this folder, which contains the base model and the wrappers for pre-training and conversion of the output.
In this folder, there is MetRobert_run2, which is the model folder with the PyTorch saves also present in it.
The file metaphorclam_wrapper.py has all the functionality to go from an upload and a set of options in the webserver to output of TSV files and Word files. This is useful for controlling pre-training/post-processing. MetRobert accepts many different inputs, such as pre-processed TSV files as they are made by our pre-training. (For re-running without re-running pre-training) but also Alpino parsed XML files. However, the most normal input is either .tok tokenized file files or just .txt text files with sentences. Tokenization and sentence segmentation are thus built into our pre-processing, as well as parsing with Alpino. This file also refers to the current Alpino installation in the LAMachine environment.
Metaphorclam.py contains some important elements for the webserver, including the output structure and parameters that a user can choose.
Outputgenerator generates better output based on output.tsv (for example, by showing the word instead of an index and by filtering out unnecessary columns). This is used for post-processing.

The most important folder is MetRobert_run2, where main_dutch.py launches the training model. main_Config contains parameters, saves contains models that you can also run on dev data only. Data is located in data_sample, and the data we use from Pasma is in pasma_all (all stands for all POS tags). Personally I think it might be easier to give access to ponyland to you.
python requirements (possibly outdated)
boto3==1.16.63 nltk==3.5 numpy==1.20.0 requests==2.25.1 scikit-learn==0.24.1 scipy==1.6.0 torch==1.6.0 torchvision==0.7.0 tqdm==4.56.0 transformers==4.2.2
if you inspect main_dutch.py you see we run dev only if save is in the arguments (which it is) argos come from main_config.
If you can generate the right data you should be able to easily use the dev version of the model. The data is .tsv (tab separated) which is very similar to csv It contains text name index 0 sentence pos tag word index
The text name doesn't really matter, so generating data is not that hard. You might be able to use our generator scripts
COV_fragment01 1 0 OP DE INTENSIVE CARE Zorgverleners op de intensive care staan in de frontlinie om het coronavirus te bestrijden. DET 6
pasmaparser_cov.....py is made for this, but normally we use quite a few different steps so reproducing this outside of our environment might be harder for now (you can bypass this by just using lamachine like we did)

## 4.1 metaphorclam.py

## 4.2 metaphorclam_wrapper.py

## 4.3 outputgenerator.py

## 4.4 pasmaparserclam.py

## 4.5 style_joost.css

5

SECTION

# Parsers and Data Creation

## 5.1 Pasmaparser_allpos

Pasmaparser_allpos is a tool that creates TSV files based on Trientje Pasma's output which can be used for our models. It parses all parts-of-speech (POS) and provides an output file containing the desired data.

## 5.2 Pasmaparser_verb

Similarly to Pasmaparser_allpos, Pasmaparser_verb provides TSV files based on Trientje Pasma's output. However, it only parses verb information and provides an output file containing verb data alone. The Model has a verb only part and a full POS part, this could eventually be changed in the future to introduce custom models for each POS. Early signs show that does this might improve the results. The idea would be that based the input data is split per POS tag and fed into different models. The verb model operates on the following used parameters: sentence, word index, POS tag. Note that all words have the same POS tag in the verb only model so the impact of using the POS tag might still be very limited/unneeded.

## 5.3 Covidparser

Covidparser is a tool that generates TSV files based on output from real texts instead of Trientje Pasma's output. This tool is useful for generating models using data from actual texts rather than a simulated corpus. Note that both the covid parser and the pasmaparsers use texts that have POS information already inside it. This is provided by Alpino for the covid parser and by Trientje Pasma for the pasmaparser. Note that Alpino is imperfect in generating POS tags and

makes erros sometimes. Because POS tags are an important input for our model, there might be potential for switching to a different approach. The RoBert variant used by our models seems to outscore Alpino in pos tag prediction. This could be a potential solution for the future. The verb model operates on the following used parameters: sentence, word index, POS tag. The possible POS tags that can be used are VERB, NOUN, ADJ, ADV, DET, PRON, ADV, DET, NUM. Note that this list is not complete and might be expanded in the future. Also note that the pasma corpus contains POS tags in dutch and hence these needed to be converted, we set up the following rule set by trial and error:

Pos tagging gebruikt in pasmapaser_melBERT.py

| Voorspelde woordsoorten | Pasma Tag | MelBERT Tag |
|---|---|---|
| Werkwoord | WW( | VERB |
| Zelfstandig naamwoord | N( | NOUN |
| Bijvoeglijk naamwoord | ADJ( | ADJ |
| Voornaamwoord | VNW( | ADV, DET, PRON** |
| Bijwoord | BW( | ADV |
| Lidwoord | LID( | DET |
| Nummer | * | NUM |

Figure 1: Pasma used POS-tags, and their equivalent in MetRoBert

## 5.4 Train/Test Split

The train/test split is a technique used when creating models in machine learning. This involves randomly dividing the data into two sets - the training set and the test set. The training set is used for training the model and the test set is used for validating the results. To ensure fairness, the data must be randomly shuffled prior to splitting so that the training and test sets contain the same proportions of each class. This is done to ensure that the model is not biased towards a specific class. The training set is used to train the model and the test set is used to validate the model's peformance. Our model also allows a dev set or can be ran on a dev set afterwards.

## 5.5 pasma corpus

We did basic analysis of the pasma corpus to analyse it's content. Note that the Pasma corpus contains both a news and a conversation side. We started adjusting our systems to also be able to use conversatoinal data but this is not finished and might be nice area for further enhancements/further research.

| Pasma | News | Conversational |
|---|---|---|
| Metaphorical vnw | 6.5% | 28.8% |
| Metaphorical vz | 70.7% | 53.4% |
| Metaphorical ww | 31.3% | 15.5% |

6

SECTION

# Covid dataset

## 6.1 Data choices

The choice has been made to gather dutch corona data in news articles. The data has been scraped from 2021 and 2022. The data has been identified using NexisUni. It was scraped based on the following requirements:

- The data should be from either of the Dutch newspapers Trouw, NRC, AD, Volkskrant, Telegraaf.
- The data should mention either of *covid* *corona* *ncov-2019* *wuhan-virus* *wuhan virus* (and some variations)
- The data should be in a specific timeframe, we scraped twice seperately for two years.

## 6.2 Data scraping

We scraped the entire datasets from nexisuni using our own nexisuni scraper, we later switched to a selenium based approach that was adjusted for this purpose. Nexisuni scraping is constrained, as data can only donwloaded per 100 articles, and only the first 1000 samples of a search query can be downloaded. To fix this we wrote a tool that automatically takes a query and divides it into sub queries (on date constraints) that are below 1000 results, this way the tool can enumerate the sub queries and download the data automatically. We are unfortunately not able to publish this solution as it goes against the guidelines of NexisUni to do so. We recommend new scrapers to simply use the latest version of existing selenium solutions, as they are probably more

up to date and less error prone then our attempts: https://github.com/chongshu/LexisNexis-Scraping.

## 6.3 Dataset generation

Nexisuni downloads data per 100 samples, leaving us with a lot of zip files and no usable output, we went trough multiple phases to ensure our data was in usable form. First we unzipped all output into one single folder. What is received is a gigantic folder containing a lot of word and pdf documents with each document given the name of the article. We have kept this dataset and upon interest might share parts of it. For the model however, this data is unusable and there are size constraints in place due to the image form of the pdf data. (The .docx files contain mostly plaintext where the pdf's are high res pictures of the actual newspaper.) We labeled this dataset dataset_full_nonrenamed.

The first step we took was trimming the dataset, the original dataset was 20GB in size per year containing 58.000 files. After copying the dataset while removing the pdf documents to a duplicate folder we are left with only docx files containing newspaper content, the size was shrinked to 729 MB due to this. This step can be reproduced with pdfremover.py.
The next step was converting the .docx files to .txt, this was done with antconverter. This leaves us with the same data, removing some unneeded pictures while maintaining the data. The new size is 157MB. The images removed where logo's and not of impact to our model or data. This step can be reproduced with antfileconverter.
The third step was renaming the .txt files. The txt files all still had the name of the original file as data. We did simple parsing of the txt files to gather source newspaper, month and index. We use this data and metaseperator.py to rename all the files to the following source-month-index.
The third step did not only rename the txt files, but also extracted metadate and seperated this, so each file got an additional file source-month-index-meta that contains the metadata. A metadata sample is included for showcasing:
2 Bron: TELEGRAAF
3 Datum: Onbekend
4 Section: WATUZEGT; Blz. 18
5 Length: 560 words
6 Auteur: Onbekend
7 Locatie: Onbekend
8 Highlight: Onbekend
9 Aantal foto's: 0
10 Load-Date: April 10, 2020

We were happy with the data in this form, as the normal file only contained the news article palintext content now, which was parsable for our model and neatly structured for analysing.

## 6.4 Corona filtering

Matching corona words does not automatically say a text is about corona a lot. For example a text could discuss something else and quickly mention corona once as it has become a big element of society in the years our research is focused on.
To combat this we introduce coronafilterer.py a program that scores all texts on how much corona related words are in it. The score is calculated by giving each corona word a score and dividing the cumulative score by the amount of words in the text. The corona score hence is a measure of the amount of corona per word.
This score is calculated by a set of different scores, some words get 2 points for being high corona words such as covid-19, pfizer, ncov, intensive care etc. Some points get 1 point like quarantaine, vaccin, wuhan, 1.5, rivm. Some get half points because they are often not really an indiciation like the word virus or corona, but also the word longs.
Some words then get negative scores, like WK, darten or soccer.
These scores have been determined and tweaked by looking at a lot of example outputs. The data is generated for the entire dataset and scores in covid_scores.txt, an example is shown below:

Filename: AD-April-0207.txt Covid score: 1.7302052785923754
Filename: AD-April-0245.txt Covid score: 0.9836065573770492
Filename: AD-April-0246.txt Covid score: 1.1466011466011463
Filename: AD-April-0261.txt Covid score: 4.503012048192771
Filename: AD-April-0297.txt Covid score: 0.6830031282586028
Filename: AD-April-0317.txt Covid score: 0.0915032679738562

After generating corona scores for the entire dataset. The same python files iterates over this and generates some medians. It then calculates subsets of the datasets:

bottom_covid_score = medium_covid_score * 0.25
half_covid_score = medium_covid_score * 0.5
low_covid_score = medium_covid_score * 0.75
high_covid_score = medium_covid_score * 1.5
two_covid_score = medium_covid_score * 2
top_covid_score = medium_covid_score * 4
We also print the procent of texts that get these scores, with top for example mostly being 2-4 percent of the data. This can then be used to select certain subsets that contain more or less corona related data.

## 6.5 Data iteration and generation

In order to iterate over our data and to make researchers use it, we have created TextSelector.py. This file uses our dataset and metadata to allow researchers to autoamtically create subsets of the dataset. Using this tool still requires a researcher to have python 3.8 installed. The file then

goes on to ask you a series of questions and will filter the data based on the response of this questions, you get a folder with the data that applies to your constraints to use. Example from the user guide:

Please enter a foldername so we can store the results of your query in this folder. test1
Please enter a minimum amounts of words per document to be considered for the analysis, type -1 if not applicable 1000
Please enter a maximum amounts of words per document to be considered for the analysis, type -1 if not applicable -1
Please enter a paper to filter on, choose AD, NRC, VOLKSKRANT, TROUW or TELEGRAAF or -1 if not applicable, you can give multiple -1
Please enter a covid score to filter on, everything higher or equal then your value will be kept, choose -1 if not applicable -1
Please enter a start date in DD-MM-YYYY format, type -1 if not applicable -1
Please enter an end date in DD-MM-YYYY format, type -1 if not applicable -1
If there is a certain word or sentence you want the text to include, please type it now, type -1 if not applicable, you can use |OR| or |AND|
as logical operators, do not combine them -1
If there is a certain word or sentence you do not want the text to include, please type it now, type -1 if not applicable , you can use |OR| or |AND| as logical operators, do not combine them -1

Hence we can easily filter on minimum amount of words, maximum amount of words, newspaper, start date, end date, covid scores, inclusion of target words and exclusion of target words.

SECTION 7

# Using the webserver

Using the webserver was tried to be made easy. If you go to the webserver page you get this picture:

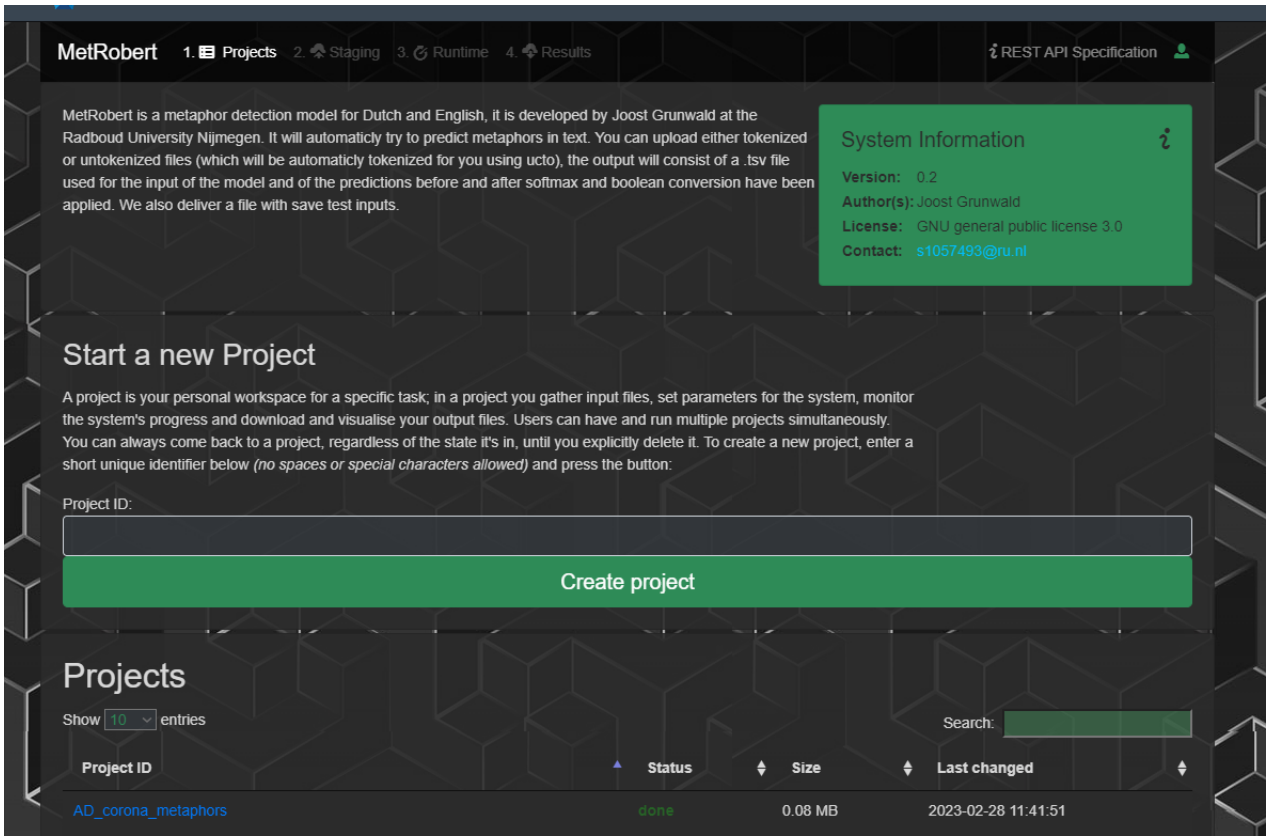



Figure 2: The main page

Now you can type a new project name and click create a new project. You get in the second stage:

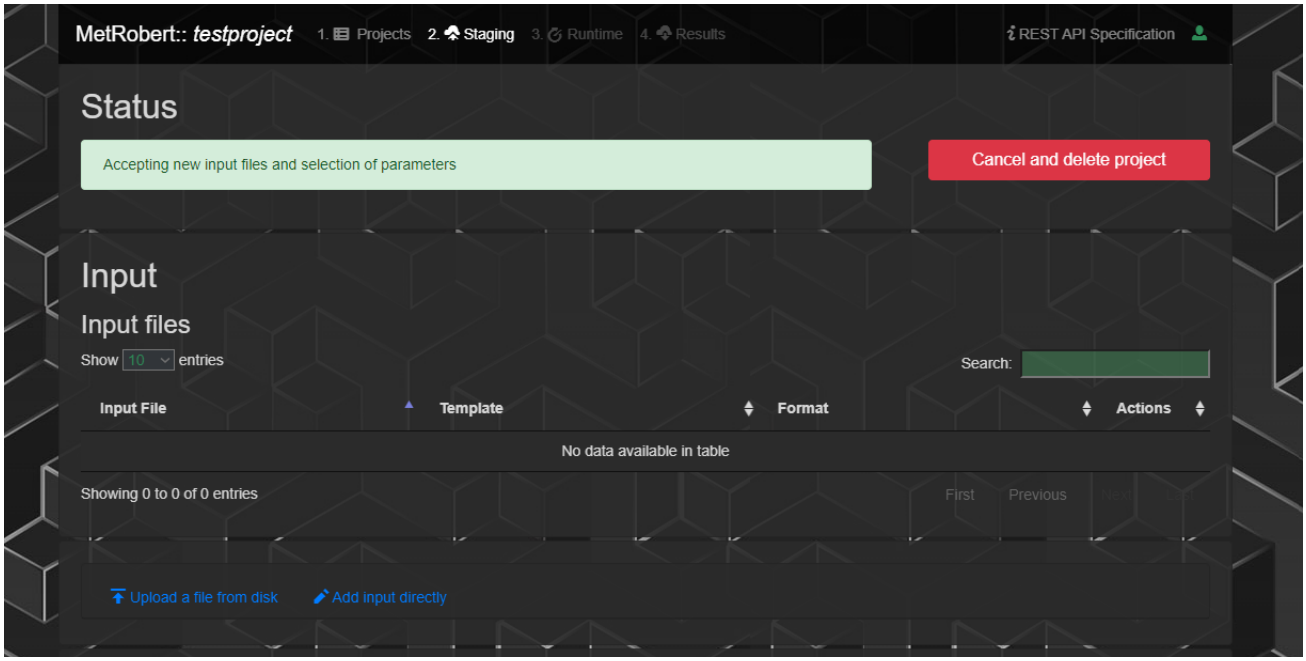



Figure 3: Creating a project

In this stage you can either choose add input directly, where you can just copy paste text in, or you can upload .txt or .tok files. (Tokenized files or just simple texts in txt). Choose untokenised if you have plain .txt files and otherwise choose tokenised if you have .tok files. After uploading scroll down and you see the following:

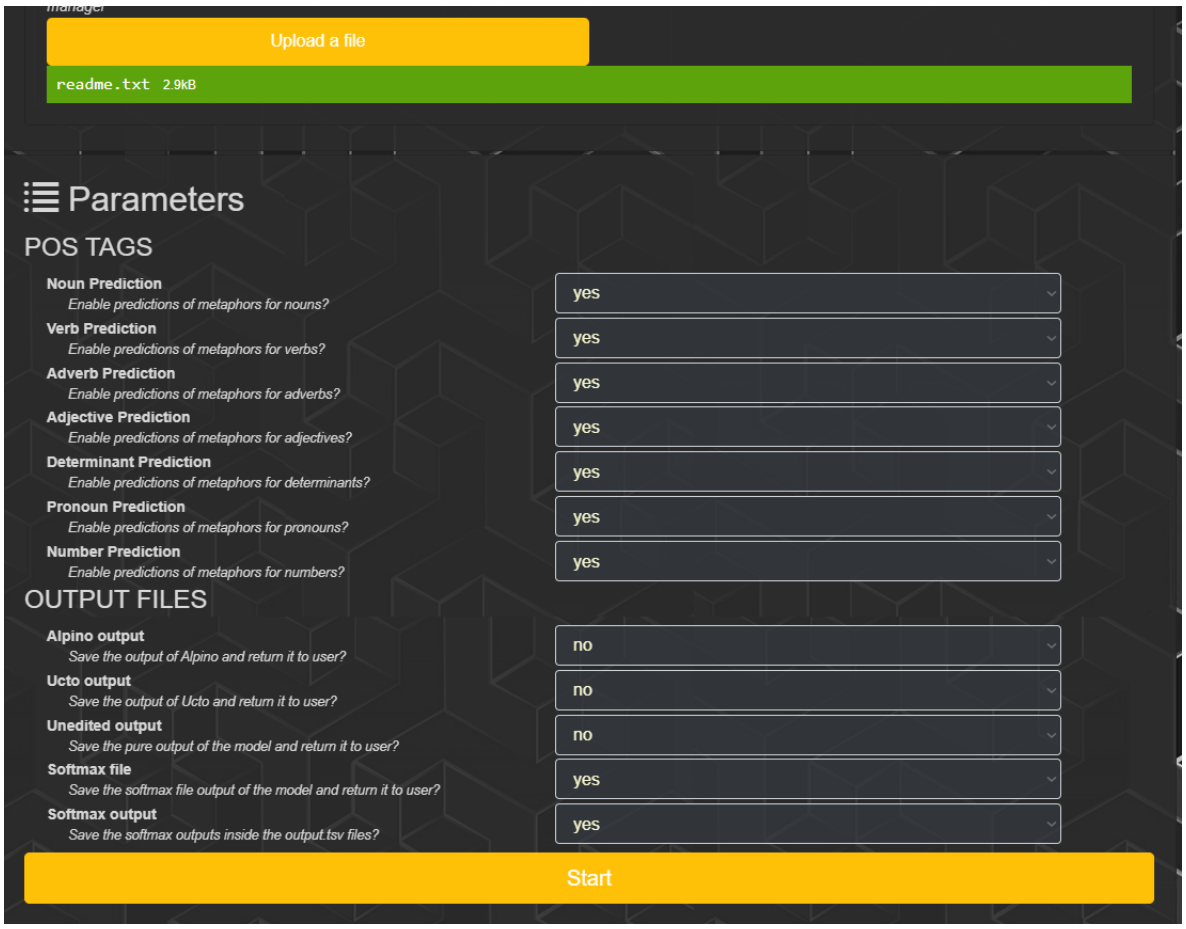



Figure 4: Setting the parameters

You now see the actual files that you inputed as green (succesfull) entries. Now you can select POS tags, the POS tags you chooce will be the POS tags that predictions will be made for, allowing the user to create specific tasks. The output files can also be choosen, alpino and ucto output is mostly not needed, as tokenized and pos predicted output is not really usefull output but might be to developers. Unedited model output is the same. The softmax files allow you to also see a certain measure of model confidence with your outputs. Press start after choosing your settings. The system now starts to process your input:

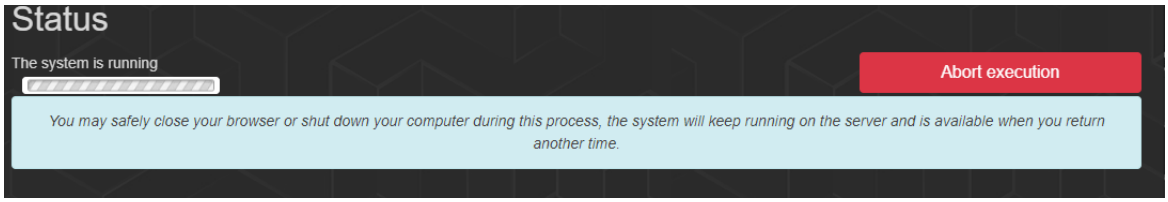



Figure 5: The model running

After the model is done running you get your output:

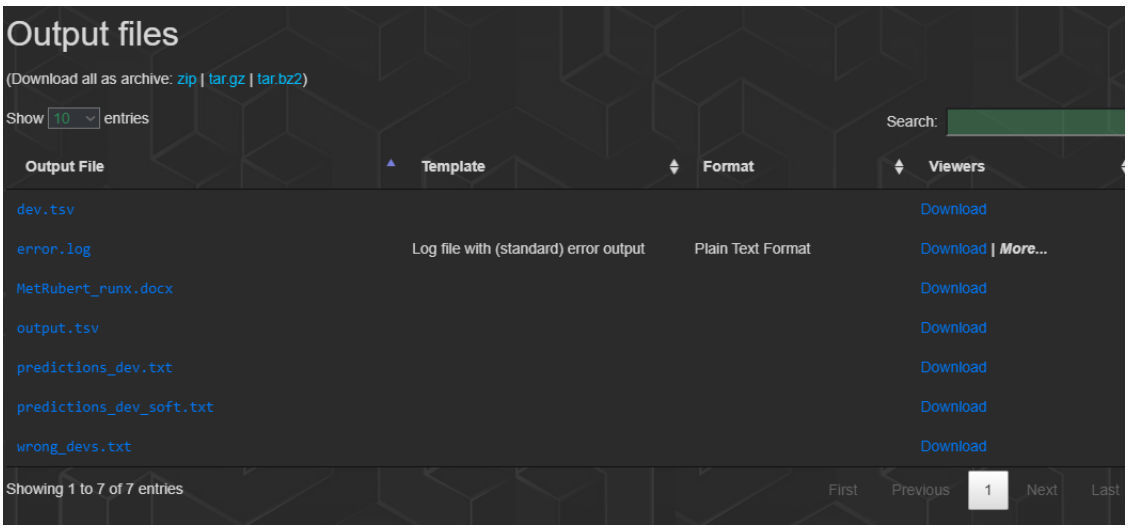



Figure 6: The model output

In this output, you get a word document about metaphors in the text, as well as prediction_dev data in txt, note here that the _soft version contains confidence intervals as well. The output tsv data can be imported into excel as sheet the error.log shows model errors. The word/tsv data is already adjusted to catch eventual errors.