Technical Report - STEREO

Steffen Epp, Marcel Hoffmann, Nicolas Lell, Michael Mohr

December 2020

Contents

1 Introduction

This technical report belongs to the paper "STEREO: Automatic Extraction of Statistic and Experimental Conditions from Scientific Papers" by Steffen Epp, Marcel Hoffmann, Nicolas Lell, Michael Mohr, in which methods for statistic extraction with aspects and experimental conditions have been developed. It describes the technical details of STEREO (STatistic ExtRaction of Experimental cOnditions), a Python tool for the extraction of statistical data, sentence topics, and experimental conditions based on those methods. The first step consists of an easy to use command line interface for the statistic extraction task and an active wrapper graphical interface to learn new rules for new data sets. The second step is divided into two parts. Both use notebooks for training and a graphical interface for evaluation. In Section 2 we describe the principal architecture and features of our application. In Section 3 we explain how we implemented it and in Section 4 we depict how to use it.

2 Application Design

2.1 Architecture

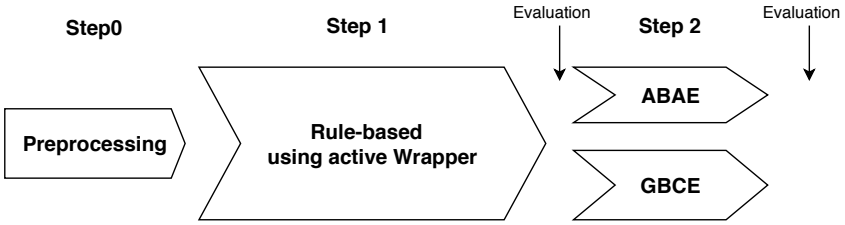

Figure 1: The Extraction Pipeline to extract statistics and experimental conditions with the evaluation points.

We structure our pipeline into three steps, which can be seen in Figure 1. In step 0 we prepare the data contained in the corpus for the following steps. In step 1, we detect significance test statistics and extract the sentence containing the found statistic and the according values of the statistic. In step 2 we find the sentence topics and experimental conditions of the statistics. For this step, two different approaches are used. The sentence topic approach is based on Attention-based Aspect Extraction (ABEA). The other approach is Grammar-based Condition Extraction (GBCE). It is a rule-based approach, that utilizes noun phrases based on the dependency tree of the library spaCy.

2.2 File structure

In the following, the different files of the program and their usage will be described.

2.2.1 Statistic extraction

The code for the statistic extraction part, can be found in the *Code* folder. The following files are artifacts or definitions which are necessary to use our tool.

- *skipCounterDocuments.txt*: Contains just a number that indicates how many documents have been completed by the active Wrapper. This documents will not be considered in further learning.
- *skipCounter.txt*: Only contains a number that indicates how many sentences of the current document shall be skipped. This regards the active Wrapper learning.
- *supStatistics.txt*: This file contains a keyword for each supported statistic type. To extend the program to new statistic types, add further keywords here.

- *supKeywords*: This directory contains a file for each statistic type. Each of those files stores keywords for the different statistic parameters that should be extracted.
- *rPlus.txt*: This file contains different rules to classify sentences as statistic.
- *rMinus.txt*: This file contains different rules to classify sentences as non statistic.
- *subRules*: This directory contains a file for each rPlus rule. The file *Rj.txt* belongs to line j in $rPlus.txt$. These rules extract the different values of a found statistic.
- *extracted.json*: This file contains the extracted sentences during the active wrapper process along with their statistical data.

2.2.2 ABAE

The code for ABAE is in the Code\ABAE folder. Most other used files are there or in a sub-folder as well. Exceptions are:

- *AEapaSen* and *AEnonApaSen* in Code\Evaluierung where the sentences for evaluation are stored,
- *extractedSamples.json* and *extractedOther.json* in Code containing the sentences with statistics extracted in step 1,
- the Cord-19 dataset extracted in the same folder as this git.

The code to use ABAE is in the Jupyter Notebooks in the *ABAE* folder and the file *argsParseDummy.py* which contains parameters like load and save paths. Plots are saved in the *plots* subfolder, processed data is stored in the *savedDataAndModels* subfolder. The trained models and their evaluation results are each stored in a subfolder of *savedDataAndModels* *outDirPath* with the name corresponding to the used embedding, training data and k.

2.2.3 GBCE

All components of GBCE are provided in the *Code\GBCE* folder. In the following, the different files of the program are described except the files already mentioned in Section 2.2.2.

- *preprocessing.py*: This file includes all essential methods for preprocessing the input-string provided by step 1 and extracting noun phrases.
- *helpFunctions.py*: This file provides frequently used methods which includes methods for loading files, output functions and other help functions.
- *rMinus.py, rConditions.py, rFull.py*: These files describe the different rulesets including the learned rules.
- *classRuleStat.py*: This file contains a class named *RuleStat*. For every input, a *RuleStat* object is instantiated where the noun phrases and conditions are stored as attributes.
- *main.ipynb*: This Jupyter Notebook executes GBCE. This is further described in 4.4.

The file *rMinus.py* contains sixteen rules to classify removable noun phrases, since they provide no statistical information. These rules have lower complexity, wherefore we will not explain them in more detail, except for: *def passiveAuxiliary(doc, output)*. This method contains three rules associated with passive auxiliaries, assigning an aspect. Since our approach is to extract experimental conditions, finding the aspect is not the main priority, but it can still be used to exclude noun phrases from nominees.
The first sub-rule set is *rFull.txt*. It contains rules where, when matched, all experimental conditions can be extracted and no further rules need to be applied.

- *handleBetween(doc, output)*: This method contains three rules, where the keyword *between* was identified. An example sentence is: "*We show here that NMDAr blockade increases cross-frequency coupling between delta and high-gamma activity (t (6) =4.947, p=0.0026)*". Here, the structure is *Aspect - Root verb - Condition 1 - auxiliary word - Condition 2*

- *def pearsonCorrelation(doc, output)*: This method combines 2 rules to identify basic patterns of pearson correlations. One exploited pattern is: *"correlated to" - condition 1 - conjunction - condition 2* . An example sentence for that is: "*The GAD-7 score was positively correlated to the level of concern about the professional future and the level of concern of contracting the COVID-19 shown by the dentists*"

- *def comparativeAdjective(doc, output)*: This method combines all rules concerning comparative adjectives. Since this is the pattern that was occurring the most, nine rules were created. An example sentence would be: "According to age, the mean willingness to receive was significantly higher among *those above 40 years of age* compared to *those under 40 years* (t(505) = 2.1, p = 0.037)

The file *rConditions.py* is the second sub-rule set of the *R+* Rules. It is applied for locating exactly one condition.

- def findwhWords(doc, output): This method presents a rule, that locates a wh-word. Subordinate clauses introduced by such words always indicate experimental conditions. This rule not only assigns noun phrases, it is also important since spaCy's dependency parser is not assigning relative clauses as part of a noun phrase. An example sentence is: "Respondents working in a medical facility have a higher perception for risk of getting COVID-19 than those *who do not work in a healthcare unit*"

- def bagOfWordsSetCondition(doc, output): This method combines four rules, each for specific words, which worked in every occasion in the tested sentences. One example sentence with the keyword "between" is: "Positive correlations were found between *the illness severity score* and *the reported 'number of days spent in bed'*, and *'number of days spent away from usual activities'* "

3 Implementation

3.1 Libraries

We implemented our pipeline in Python 3.8. Some important libraries were:

re Regular expressions are used to represent our rules in step 1. We used the python package *re*[1], which offers several methods and objects based on regular expressions. Different basic regular expression parts are used. For example to represent the statistic types the named grouping feature of the *re* library is applied e. g `(?P<ttest>...)`. Furthermore we used this feature to tag the parameter in the regular expression for each statistic type.

spaCy The python module spaCy[2] is used for natural language processing. The module provides several built-in components and helpers, including the model "en_core_web_sm" for tagging, parsing and entity recognition of English text.

3.2 Loading and preprocessing

The loading and preprocessing is done in *loadPaper.py*. We first split the full text of a document into single sentences using the regular expression `\.\s?[A-Z]` and the *finditer* method from the package

[1] `https://docs.python.org/3/library/re.html`
[2] `https://spacy.io/`

re. An iterable object containing every found match is returned. The sentences are split at each of those found positions to form a list. Afterwards, we apply a filter to throw out every sentence not containing at least a single digit, by using *re.search* with the regular expression \d. Sentences not containing any digit are ignored, because we assumed that they do not contain any statistic. *loadPaper.py* also contains methods to load and store all other files mentioned in Section 2.2.

3.3 Active Wrapper for statistic extraction

The active wrapper belongs to the first part of the pipeline and consists of mainly two parts. The *activeWrapperAllPaths()* method loads all paths of a given directory. For each path, the method *activeWrapperLoop()* loads the file and applies the available R^+ and R^- rules for each sentence. We mark every R- match from the sentence to check if there are any uncovered digits. The match from the individual rules are merged, if they are overlapping so that every match in the sentence will be marked. However, for every sentence within the corpus, we need to apply all of the R- rules, which is inevitably increasing the run-time. If at least one number of the sentence is not covered by any rule, the user is asked to define a new rule by a graphical user interface.

3.4 ABAE

Our implementation of ABAE, using TensorFlow 2.3, is based on Johannes Huber's implementation[3] of ABAE, which is a TensorFlow 1.9 adaption of the original authors implementation[4] using Theano 0.9. At it's base, each of our ABAE models is a sequential Keras model with TensorFlow backend with custom layers. This model is defined in the *model.py* with the custom layers defined in the *custom_layers.py* file.

3.5 GBCE

The input needs to be a sentence containing statistics, which provided by step 1. The GBCE can be split in three parts. The first part is preprocessing the input text. This is necessary, since step 1 modifies the text and conversion errors could have occurred. This improves the results of the next step, the extraction of noun phrases. Hereby, the noun phrases need to be localized in the text and checked for errors caused by e. g. quotations. Afterwards in part three, the rules of the different rulesets are applied to identify the experimental conditions included.

4 Manual

This chapter will explain how to use STEREO. The first subsection will explain what preparations need to be done. Then one subsection explaining the use of the individual parts: active wrapper (step 1), ABAE (step 2) and GBCE (step 2) follows.

4.1 Preparation

To fully use our tool you need to download the CORD-19 dataset, e. g. from here[5]. We used the version of 21st of September 2020, older or newer versions should work as well, but you should check if all the paths are still correct. Then you need to download our git from here[6]. Extract the CORD-19 dataset and our git into the same folder that we will call "FolderA" for the purpose of explanation. If done correctly, there should be a lot of *.json* files with hex-code names in the folder with path */FolderA/Cord-19/document_parses/pdf_json/*. There should be some *.py* and *.ipynb* files as well as an *ABAE* and a *GBCE* folder in the folder with path */FolderA/2020ss-project-ie/Code/*.

[3] https://github.com/harpaj/Unsupervised-Aspect-Extraction

[4] https://github.com/ruidan/Unsupervised-Aspect-Extraction

[5] https://www.kaggle.com/allen-institute-for-ai/CORD-19-research-challenge

[6] https://gitlab.informatik.uni-ulm.de/dbis/data-science-and-big-data-analytics/teaching/2020ss-project-ie

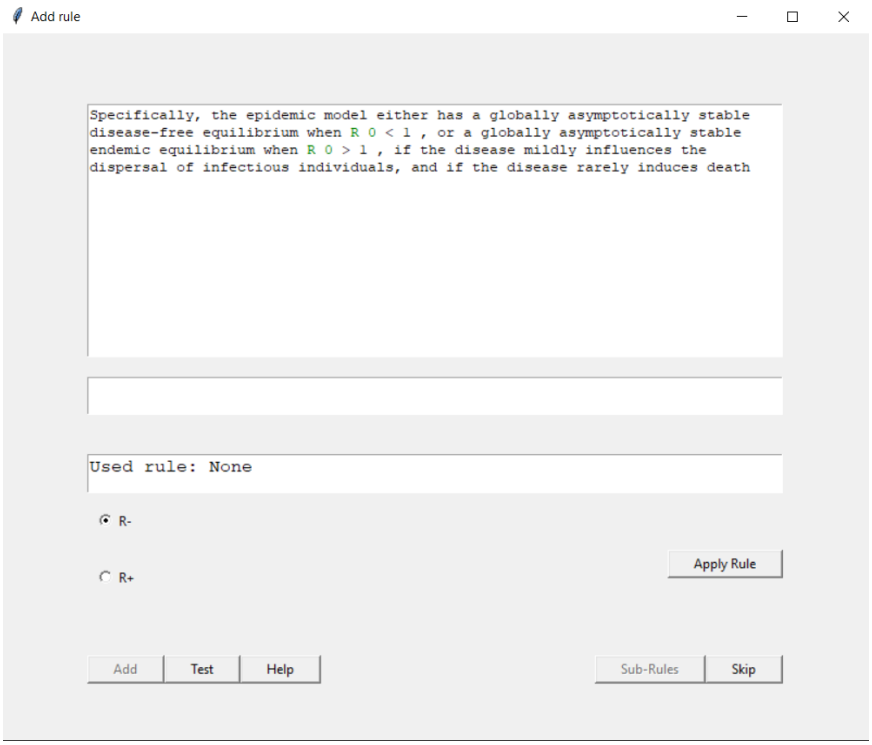

Figure 2: GUI dialog, for testing and defining new rules.

Make sure you have Python 3.8 installed and then either create a virtual environment or use your global one for the python libraries. The library versions are stored in our git in the requirements.txt file. To install them automatically, activate your virtual environment, navigate into FolderA and run "pip install -r 2020ss-project-ie\requirements.txt". For spaCy to recognize the English language, we used a pretrained model. You can download it by running "python -m spacy download en_core_web_sm". We ran all code on windows machines, but the paths are wrapped in pathlib Path objects to work with any other operating system, though this is not tested.

4.2 Statistic extraction

It is possible to extend or create completely new rule-sets for new data sets, using *ActiveWrapper.py*. Start the active wrapper with the command `python ActiveWrapper.py`. A GUI is offered for this process. There are two kind of GUI dialogues.

Learn rules with active Wrapper To start the active wrapper for statistic extraction, execute the python file *ActiveWrapper.py*. As soon as when the active wrapper found a sentence with a pattern that neither a R- nor a R+ rule can cover, the GUI dialogue of Figure 2 will be invoked. It is possible that other rules might still apply. These matches are marked green, so the user can see, which pattern has already been covered with a rule. Every rule has to be tested first, by clicking the *Test* button, before it is possible to add it. Before adding a rule, the right rule set, *R+* or *R-* has to be selected. To define a *R+* rule, a group name has to be set inside the regular expression, as introduced in Section 3. The general syntax of a *R+* will look like the following:

```
(?P<statistic type> regex )
```

For example statistic type could be `ttest` and regex could be `t\(\d+\)\s?=\s?\d+`. To actually extract values from a statistic, additional so called sub-rules has to be defined for every *R+*. Every statistic type has a own record of what kind of parameters and values are able to be extracted, e. g.

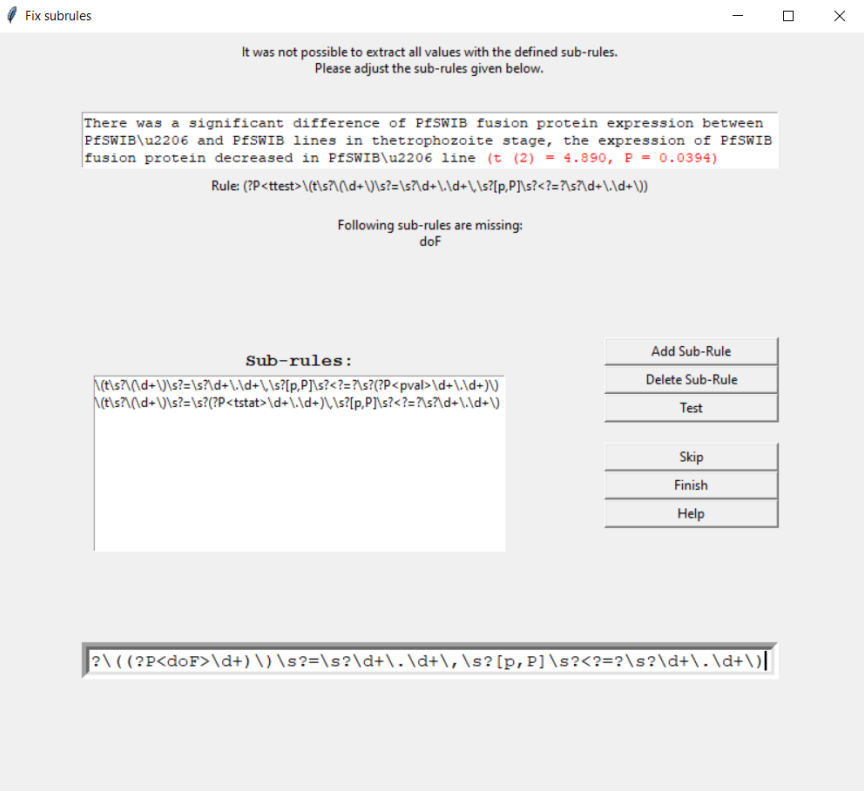

Figure 3: GUI dialog, for editing and creating new sub-rules.

the degree of freedom, p-value, etc. After clicking the *Sub-Rules* button, a new window will open. This window will take the match of the defined rule as input. Now, multiple sub-rules can be defined and added to a list, which will then be connected with the previously defined $R+$ rule. Like for $R+$ rules, for every sub-rule, a regular expression with a group name must be defined. These will generally look like the following:

```
regex (?P<statistic type> value ) regex
```

Here, *value* is a regex, which matches the numerical values of the parameter to extract. For example, the name of the parameter *groupname* could be `doF` for degree of freedom. With *Apply Rule*, 5000 randomly selected documents will be loaded from the corpus and tested for any matches of the entered rule. By pressing the *Skip* button, the dialog will be closed without adding a rule and the number in *skipCounter.txt* will be increased. Via the *Help* button, a help page will be opened, providing additional information.

Fix incomplete sub-rule If the active wrapper is able to apply a $R+$ rule on the given sentence and notices that in the respective sub-rule file a sub-rule is missing, the sub-rule dialog will be invoked, Figure 3. Like in the dialog described above, the complete sentence in which the match has been found will be printed and the match will be marked. Additionally, the actual matched rule will be printed along with the sub-rules defined for this rule. Furthermore the missing sub-rule tags will also be printed. The already defined sub-rules are included in a list box. It is possible to delete these rules by selecting them and clicking the *Delete* button. New rules can be added by entering a new rule, according to the syntax described above, and by clicking the *Add* button. To finish the dialog, the *Finish* button will update the respective sub-rule file with the new rules and close the dialog. By pressing the *Skip* button, the dialog will be closed without adapting any changes made. However this will not increase the skip value in *skipCounter.txt*.

Define new statistic type To define a new supported statistic type "yourType" open the file *supStatistics.txt* and write the name of your new statistic type below the others. Then create a new file in *supKeywords* and define the statistic parameters. It has to be named "yourType.txt". In this file you define line by line the name of the parameters belonging to your defined statistic type.

Reset (Skip/Document)counter The files *skipCounter.txt* and *skipCounterDocuments.txt* contain just a single number each. The first file specifies how many sentences have been skipped by the user. To see skipped sentences again set this counter to 0(or an other smaller number than the current count). The second file specifies how many documents have been completed by the active wrapper so far. If you restart it these number will be skipped in the dataset to avoid reevaluating them every time you start the tool. If you want to go back to earlier documents reduce this number.

Extract sentences from the dataset To extract all sentences in the dataset according to the rules in *rPlus.txt* you have to use the script *StatExtraction.py* with the parameters "-ex", "-source", "-target". The "-ex" tells that you want to extract statistic sentences along with their parameters, "-source" needs to be a document parse in json format and target is just a name in which file you want to find the results. Optionally an additional argument "-d" can be added to process a directory instead of just one json document. An example use could be "`StatExtraction.py -ex -d ../../Cord-19/document_parses/pdf_json extracted.json`"

Start the evaluation To evaluate the learned rulesets call *StatExtraction.py* with the parameters "-ev", "-samplesize","-type", "-source". The "-ev" tells the tool you want to evaluate the statistic extraction part, to do so a random sample of size "-samplesize" will be collected. With "-type" you need to define which statistic type you want to evaluate, it has to be a supported type or "other". In "-source" you have to specify a json file with extractions created by the tool. It is also possible to evaluate the R^- rules. Here it is necessary to provide the type "*rMinus*" and the "-source" has to be a directory with parsed json files from your dataset. An example use could be"`StatExtraction.py -ev 200 ttest extracted.json`". This process creates a sample of each type which is stored in a .json file, located in *Evaluierung/Samples*. The GUI for the evaluation can be seen in Figure 4. In the dialog you can see the sentence and the classification for this sentence. Additional information like extracted values and the amount of sentences left are also displayed.

You can choose if the sentence was classified correctly or not. After finishing the evaluation, the file *Evaluierung/results.txt* will be updated. In *results.txt*, each line says how many sentences have been evaluated, how many of these have been classified as correct and the type of the statistic.

4.3 ABAE

To run any ABAE related code navigate into the Code\ABAE folder. All used code is here and the relative paths assume this as the base folder. You can look up or change most of the used paths and parameters in the *argParseDummy.py* file. Now an explanation for the main steps follows, namely training a model, evaluating a model and using a model for inference

Training a model The first part of training a model is preparing the data and training the embedding. The code to do this is in *prepareDataAndEmbeddings.ipynb*. The first cell imports all common necessary packages, the second cell prepares data and embedding for cord, the third for supp-sen and the fourth for all-sen. If you apply our method on a different dataset and want to train an embedding on the whole dataset, adapt the second cell. If you want to train an embedding on the extracted sentences from step 1, do it like the third or fourth cells. The processed data and embeddings are saved in the *ABAE\savedDataAndModel folder*.

The second step for training a model is to train the actual model. This is combined with the third step which is inferring the aspects of the model. Code for both is in the *trainABAE.ipynb* file. The first cell imports the necessary packages and defines some methods. The second cell trains all models that train on supp-sen and the third cell trains all models that train on all-sen. The fourth

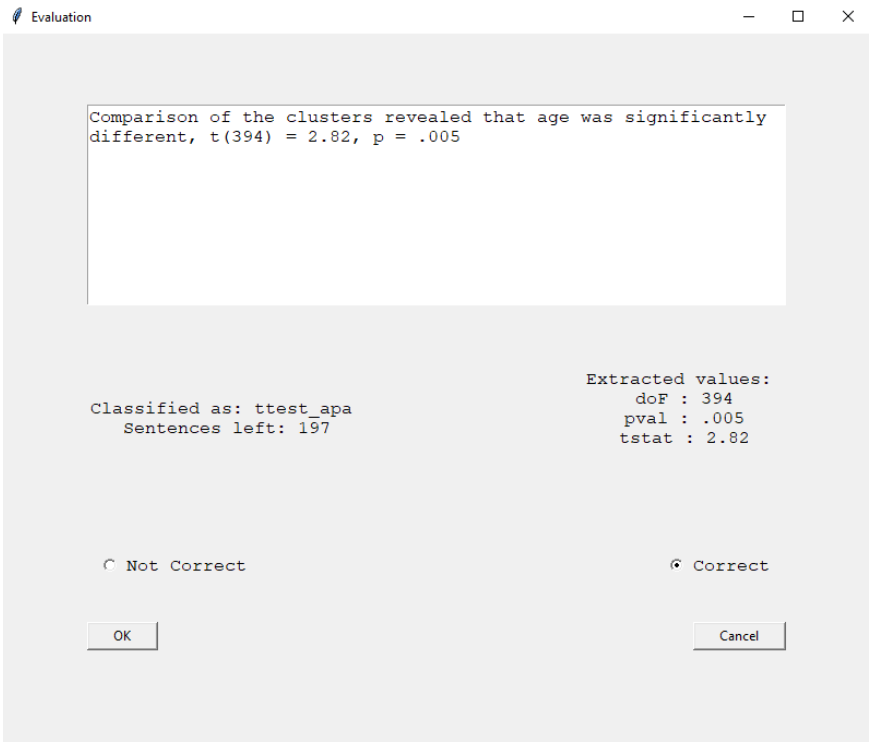

Figure 4: GUI dialog, for evaluating the learned rules.

```
inferring aspects for cordModelextr15

Aspect:  0
['correlated', 'positively', 'negatively', 'correlate', 'inversely', 'influenced', 'correlating', 'strongly', 'depended', 'link
ed', 'associated', 'affect', 'paralleled', 'significantly', 'impacted', 'contribute', 'measured', 'decreased', 'reflected', 'in
creased', 'enhances', 'insignificantly', 'promoted', 'coincided', 'contributes']

aspect:
```

Figure 5: Notebook console to input inferred aspects.

through sixth cells are for inputting the inferred aspects inferred from the most important words for that aspect. This process can be seen in Figure 5. If you want to train a own model, make sure to change the paths to the training data, embedding, and the save name. The models and their inferred aspects are saved in a folder each at *ABAE\savedDataAndModel\outDirPath*.

Evaluating a model The code for sampling $2 \cdot 100$ sentences from the results of step 1 for evaluation is in the *createEvalSample.ipynb* file. The first cell is for imports and the second through fourth for loading the paths where extracted sentences are stored. The fifth cell is an import hack to import a method from a parent folder and the sixth through eighth cells are for sampling and storing the sentences.

The code for evaluating a model is in the *evaluation.ipynb* file. The first cell imports the necessary packages, the second one defines the used methods. The third cell starts the evaluation process with the APA conform sentences, the fourth with non APA conform sentences. This opens a GUI that is shown in Figure 6. There you can see the current sentence at the top and a list of all loaded models' outputs as well as a pair of radio buttons for every model. Decide whether a given model output is correct by selecting the corresponding radio button in the same line. When all radio buttons are set for the current sentences, press "Ok" to repeat this process for the next sentence. After finishing the evaluation on the loaded sentences, a summary is saved to

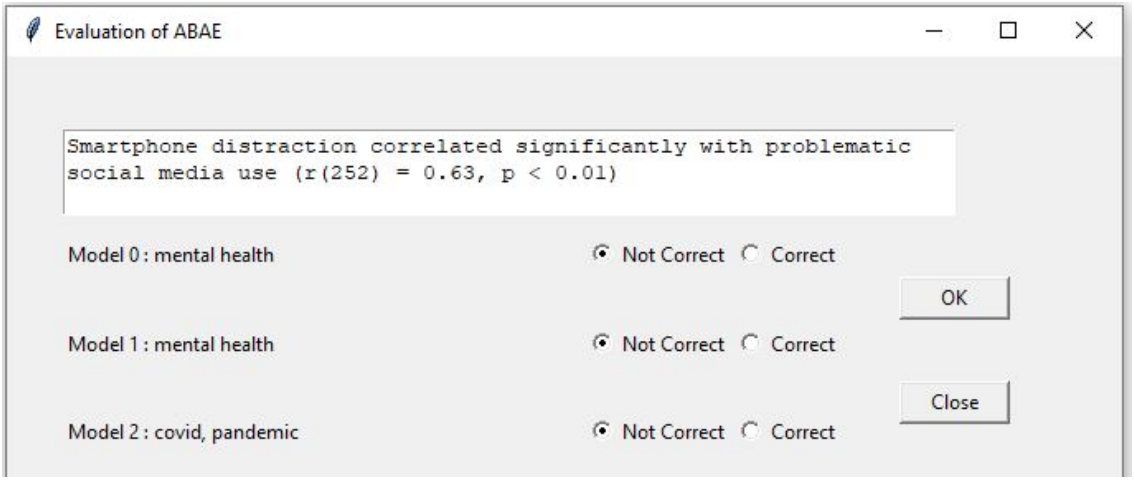

Figure 6: ABAE eval GUI, cut after 3 model outputs.

*ABAE\savedDataAndModels\outDirPath\evalResults** and the results of each individual model are saved in their respective folders to the *evalLog* files.

Using a model for inference If you want to apply a single model to the extractions of step one, use the *inference.ipynb* file. Choose the file to load, the file to save to and the model by providing paths in the second cell and then run the third cell. The results get saved to the provided path in the record syntax also used to store the results of step one.

Plotting stuff about the data You can use methods from the file *PlottingVocabStatsAndStuff.ipynb* to get information about your data like word- and sentence length-occurrences. There are also the methods we used for creating the total words over unique words and sentence coverage over sentence length loglog plots. If you use a different dataset, this may aid in the parameter choices like vocabulary size.

4.4 GBCE

More detailed information about the individual files got depicted in Section 2.2.3. This section shows how GBCE can be used. All code for GBCE are provided in the folder\GBCE. All paths expect relative paths with this as the base folder.

Extract experimental conditions from sentences To run GBCE, start the *main.ipynb* file. It includes two cells. The first cell runs all imports, the second cell starts GBCE. It is split content-related. The first part lets you adapt what information you want output:

- Chose a specific example sentence out of the file of extracted statistics
- Manually insert a sentence as string.
- Print out the actual content of the sentence
- Show the intern noun phrases of the sentence
- Visualize the ParseTree
- Get additional PoS-Data about every word in the sentence

The second part loads all essential imports and files. The third part starts the preprocessing and the fourth part applies the learned rules. The program then outputs the extracted conditions and also outputs noun phrases that were not assigned. These information can be used to create new rules.

Training of the active Wrapper We did not develop a GUI for GBCE, since the creation of rules is too complex and probably needs debugging. A manually created GUI for writing and saving new rules would be counter-productive. The active wrapper can be started by running the file *GBCE\main.ipynb*. The file is split in 5 parts.

- *Part 0* handles all the necessary imports.
- In *Part 1* there is the variable *pathToSentences* where you can add the path to the sentences you want to train. Part 1 also lets you add or remove additional outputs like PoS-tags or a visualization of the dependency tree.
- *Part 2* loads all the essentials like the model for the English language and the example sentences.
- *Part 3* preprocesses the input strings.
- *Part 4* applies all the *R+* Rules.

To create new rules, new functions can be added to one of the files in the *Code\GBCE* folder according to the context of the rule. An example is the creation of a rule for *graded adjectives*. A graded adjective makes it possible to assign a noun chunk as a condition. A rule for this pattern can be added as a method to *rConditions.py*. Further to invoke it in the active wrapper, add it in Part 4 of the code of *GBCE\main.ipynb*.

Evaluate the extraction of experimental conditions In contrast to the training of the active wrapper, the evaluation, file *evalGBCE.ipynb*, works with a GUI as depicted in Figure 7. The first cell imports the necessary packages, the second one is structured similar to the file *main.ipynb* with an additional option to set, namely the path, where the results of the evaluation should be stored.

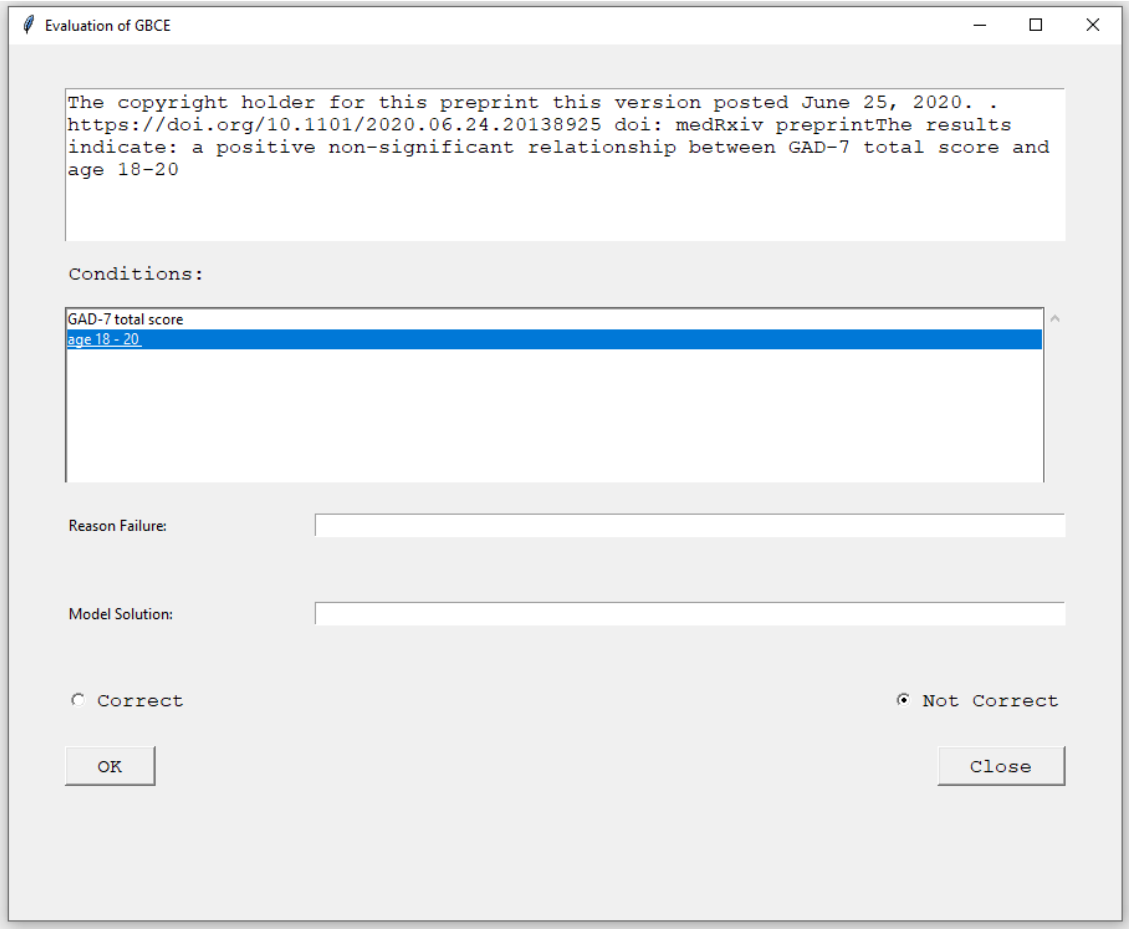

Figure 7: GUI for the eval of GBCE.