AAPVL Documentation

Release 0.1

Dorle Osterode

Mar 28, 2018

CONTENTS

CHAPTER

ONE

INSTALLATION AND REQUIREMENTS

This chapter gives a short overview how to install the backend of the AAPVL-Project and on which other projects this program relies on.

1.1 How to install the program:

First you should make sure, that all the dependencies are installed. If you install some dependencies manually, make sure they are in the proper path, so they can be found.

When all dependencies are installed you only need to clone the git-repository TODO: url here and everything should work.

1.2 List of dependencies:

1.2.1 Python packages:

These packages can be installed through the package manager of your distro or through the python package manager pip.

- numpy
- scipy
- sklearn (version 0.18)
- mysqldb (version 1.2.3. you need to install some extra packages (libmysqlclient-dev and python-dev) for that. instead of using pip you can install it directly with `aptitude install python-mysqldb`)
- opencv and python-opencv (you should use the official packages here: `aptitude install opencv python-opencv`)
- nltk
- sklearn-crfsuite
- bs4
- dateutil
- sphinx (to build the documentation only. If you use virtual environments, sphinx has to be installed in the same virtual enviroment to be able to build the api reference.)

1.2.2 non-standard libraries and programs:

For installation of the non-standard libraries and programs please follow the installation guides for that program.

- libpostal with python bindings (package name `postal`): https://github.com/openvenues/libpostal
- ParZu: https://github.com/rsennrich/ParZu
- Zmorge model for ParZu: http://kitt.ifi.uzh.ch/kitt/zmorge/

Some notes to both programs:

1.2.3 libpostal:

If you install libpostal into a user specific path and you want to install the python package postal afterwards, you have to specify the path to libpostall for the installation. With pip you can use:

```
pip install --global-option=build_ext --global-option="-L/home/foo/libs/libpostal_
↪build/lib" --global-option="-I/home/foo/libs/libpostal_build/include" postal
```

1.2.4 ParZu:

It is sufficient to follow the installation instructions for ParZu until step 3 first part. You don't have to use the script `install.sh`. All the components installed with this script are not used from this program.

CHAPTER

TWO

FIRST STEPS WITH THE PROGRAM

This chapter gives a short overview and some simple examples how to use the program. With the general flags `config` and `debug` can be used to provide another config file or to get more information about each module in the log. In the following examples they are omitted.

2.1 Testing the setup:

To test the proper setup of all libraries, models and external data files, you can use the simple test. You can invoke this test as follows:

```
python src/backend.py test simple
```

Some example files are loaded to the database and all modules are run on every file.

2.2 Invoking the backend:

You can invoke the backend, so that every job, that is added to the database, will be processed.

```
python src/backend.py run
```

In the configuration you can choose how often the backend should look for jobs in the database and when the backend should shut itself down.

2.3 Training a classifier:

You can retrain a single classifier, e.g. the shop classifier, like this:

```
python src/backend.py train shop path/to/data
```

The directory `path/to/data` has to contain two subdirectories `0` and `1`, which contain the single data files. `0` is interpreted as the positive class and `1` as the negative class. See chapter *How to train and test the single classifiers* for more information.

2.4 Training a crf:

Two modules use conditional random fields. You can train these two conditional random fields, e.g. the one used to extract addresses, like this:

```
python src/backend.py train imp addresses.txt labels.txt
```

The file `addresses.txt` contains one address per line and in `labels.txt` in each line a label for each word is contained. See chapter *How to train and test the single classifiers* for more information.

2.5 Testing a classifier:

To test one of the classifiers (using conditional random fields or support vector machines), you can use:

```
python src/backend.py test shop path/to/data
```

2.6 Updating a classifier:

When you obtain additional data and want to update one of the suport vector machine using classifiers, you can use this subcommand to update the classifier directly with data from a directory.

```
python src/backend.py update shop path/to/data
```

The directory has to conform with the assumptions. See chapter *How to train and test the single classifiers* for more information.

2.7 Loading data in db:

To test not only the setup but to test a broder spectrum you can load some files into the database with a specific selection of modules registered for these files. With

```
python src/backend.py load --modules "1,2,3" path/to/data
```

you load all files in `path/to/data` in the database and register the modules 1, 2 and 3 for them. With running

```
python src/backend.py run
```

you can process these files. Note that this is just for testing purposes and some functionality, like e.g. the information summary for all subpages, is not available here.

CHAPTER

THREE

HOW TO CONFIGURE THE PROGRAM

The program has some parameters (e.g. user name for the database connection etc.) that should be configured. Therefore a default configuration file (short: config file) exists and can be customized. The different parameters and their meaning is explained in this chapter and the default config file is shown as example. Note: the keywords are case sensitiv, so make sure you spell them correctly.

3.1 Configuration parameters:

host IP adress to which the database is reachable. You can leave the default value, when you have your database local on your machine. default: 127.0.0.1

user Username for the database user. default: foo

passwd Password for the database user. default: bar

db Name of the used database. default: aapvl

max_tries Maximum number of failed tries to obtain jobs from the database before ending the program. If this number is negative, the program tries four times per day to obtain jobs and then sleeps in the background for the rest of the time. default: 3

delay Time in seconds to sleep before trying to get jobs from the database. The delay time is only used, after a try to obtain jobs was not succesful and is not used when max_tries is negative (see also **max_tries**). default: 3600

delay_module Time in seconds after which each module has to finished. If a module needs more than delay_module seconds, it is terminated with all its subprocesses and None is returned as a result from this module. default: 3600

day Number of times per day the program should wakeup and try to obtain jobs from the database. default: 4

update_rate Number of days after which the database should be checked for altered classification results for online learning. default: 7

food_vocab Path to a file containing food relevant vocabulary for the food classifier (see also chapter *The food vocabulary (module 3):*). default: data/food_vocab.txt

map_file Path to a file containing an assignment between postal codes and german regions. For more details also information on the assumed format see chapter *Postal codes for address extraction (module 1):*. default: data/plz.csv

legal_numbers Path to a file containing all approved boards of control for ecological traders. For more information on the file and format see also chapter *Legal numbers for ecological control posts (module 6):*. default: data/legal_oeko_numbers.txt

health_claim_substances Path to a file with common substances used in health claims. For more information on the file, the format and how to update it see also chapter *List with substances (module 7):* and *List with substances:*. default: data/health_claim_substances.txt

health_claim_diseases Path to a file with common diseases used in health claims. For more information on the file, the format and how to update it see also chapter *List with disease (module 7):* and *List with disease:*. default: data/health_claim_diseases.txt

health_claim_rejected Path to a file with rejected health claims. For more information on the file, the format and how to update it see also chapter *List with rejected health claims (module 7):* and *List with rejected health claims:*. default: data/rejected_health_claim.txt

health_claim_declination Path to a file with declinations of verbs probably used in health claims. For more information on the file, the format and how to update it see also chapter *List with verb declinations (module 7):* and *List with verb declinations:*. default: data/health_claim_declination.txt

door_list Path to a file with the eu door list. For more information on the assumed format see chapter *EU door list (module 8):*. default: data/door_list.csv

ingredients_whitelist Path to a file with known ingredients. For more information on the format or how to update the list see chapter *White- and blacklist for ingredients (module 9):* and *Adding information to the white- and blacklist for ingredients (module 9):*. default: data/whitelist.txt

ingredients_blacklist Path to a file with prohibited ingredients. For more information on the format or how to update the list see chapter *White- and blacklist for ingredients (module 9):* and *Adding information to the white- and blacklist for ingredients (module 9):*. default: data/blacklist.txt

parzu Path to the parzu executable. See chapter *Installation and Requirements* for more information how to install parzu. default: /home/foo/ParZu/parzu

3.2 Example:

The default config file:

```
host = 127.0.0.1
user = foo
passwd = bar
db = aapvl
max_tries = 3
delay = 3600
delay_module = 3600
day = 4
update_rate = 7
food_vocab = data/food_vocab.txt
map_file = data/plz.csv
legal_numbers = data/legal_oeko_numbers.txt
health_claim_substances = data/health_claim_substances.txt
health_claim_diseases = data/health_claim_diseases.txt
health_claim_rejected = data/rejected_health_claim.txt
health_claim_declination = data/health_claim_declination.txt
door_list = data/door_list.csv
ingredients_whitelist = data/whitelist.txt
ingredients_blacklist = data/blacklist.txt
parzu = /home/foo/ParZu/parzu
```

CHAPTER

FOUR

EXTERNAL DATA

Several modules relie on data that is externally provided. This is data, that changes probably often and should therefore be extandable or exchangeable. There are two different formats in which the data is assumed, either plaintext with a single phrase per line or in a csv format. Each file is assumed to be utf-8 encoded.

For every module, that uses external data, the format is described in the following sections. How to change the default location of one of the files is described in chapter *How to configure the program*. Furthermore a description how to extend which files to obtain online learning for some modules is described in chapter *How to use online-learning with the program*.

4.1 Postal codes for address extraction (module 1):

The module for the extraction of addresses assumes a csv formatted file with an assignment from postal codes to regions (Bundesland and Kreis). An example of the assumed format is given below:

```
PLZ2,ZUST_BUNDESLAND_STAAT,KREIS
01067,Sachsen,"Dresden, Stadt"
```

Note that the first line contains header information and is ignored therefor. The corresponding key in the configuration file is “map_file” (see chapter *How to configure the program*).

4.2 The food vocabulary (module 3):

The food classifier (module 3) uses a fixed vocabulary to classify text. The file is in plaintext format and contains one word per line. An example is given below:

```
Apfel
Banane
Citrone
...
```

The corresponding key in the configuration file is “food_vocab” (see chapter *How to configure the program*). In section *Adding words to the food vocabulary (module 3):* is explained how you can add words to the vocabulary.

4.3 Legal numbers for ecological control posts (module 6):

The module for the verification of traders of ecological products uses a list of valid german “Ökonummern”. These are numbers from control posts, which control and certificate the traders. Because the control posts change from time to time, this list should be always up to date. The numbers are given in plaintext and an example is given below:

```
DE-ÖKO-001
DE-ÖKO-003
```

The corresponding key in the configuration file is “legal_numbers” (see chapter *How to configure the program*).

4.4 EU door list (module 8):

The module for the validation of the use of specific product names (restricted with PDO, PGI or TSG) uses a list with certified products (door list). This list contains every product that is registered in the EU for PDO, PGI or TSG and can be exported from here: http://ec.europa.eu/agriculture/quality/door/list.html The format is csv and an example is given below:

```
,,,,,,,,,,,,,,
,,,,,,,,,,,,,,
,,,,,,,,,,,,,,
Dossier Number  ,      Designation        , Country , ISO ,   Status   ,   Type    ,␣
↪Last relevant date ,   Product Categrory   ,      Latin Transcription     ,␣
↪Submission date , Publication date , Registration date , 1st Amendment date , 2nd␣
↪Amendment date , 3rd Amendment date
PL/PGI/0005/02154,Kiełbasa piaszczańska,Poland,PL,Registered,PGI,21/11/2017,"Class 1.
↪2. Meat products (cooked, salted, smoked, etc.)",,15/07/2016,29/06/2017,21/11/2017,,
↪,
```

Note that the first four lines contain header or no information and are ignored therefor. The corresponding key for the configuration file is “door_list” (see chapter *How to configure the program*).

4.5 White- and blacklist for ingredients (module 9):

To perform a validation of the found ingredients of a product a white- and a blacklist are used. It is checked, if an ingredient is already known and allowed (contained in the whitelist), already known and prohibited (contained in the blacklist) or unknown. Therefor a white- and a blacklist have to be provided. The assumed format is plaintext and an example is given below:

```
Milch
Zitronen
flüssig
...
```

The corresponding keywords in the configuration file are “ingredients_whitelist” and “ingredients_blacklist” (see chapter *How to configure the program*). These files can be extended with words to obtain some kind of online learning see section *Adding information to the white- and blacklist for ingredients (module 9):* for more information.

4.6 List with substances (module 7):

For the detection of possible health claims a list with commonly used substances in health claims should be provided. This list should contain single words and phrases. The file is assumed to be in plaintext and an example is given below:

```
Vitamin C
Eisen
...
```

The corresponding keyword in the configuration file is “health_claim_substances” (see chapter *How to configure the program*). This file can be extended with words to obtain some kind of online learning see section *List with substances:* for more information.

4.7 List with disease (module 7):

For the detection of possible health claims a list with commonly used diseases in health claims should be provided. This list should contain single words and phrases. The file is assumed to be in plaintext and an example is given below:

```
Herzinfarkt
rote Blutkörperchen
...
```

The corresponding keyword in the configuration file is “health_claim_diseases” (see chapter *How to configure the program*). This file can be extended with words to obtain some kind of online learning see section *List with disease:* for more information.

4.8 List with verb declinations (module 7):

To prefilter sentences after a semantic analysis of a given text, a list with relevant verbs for health claims should be provided. In this file all relevant declinations of the verb has to be listed. The declinations for one verb should be delimited by a newline and between two verbs there can be one additional newline. This file is assumed to be in plaintext and an example is given below:

```
beitragen
trägt bei
tragen bei
trug bei
trugen bei
hat beigetragen
haben beigetragen
wird beitragen
werden beitragen

haben
hat
haben
hatte
hatten
```

The corresponding keyword in the configuration file is “health_claim_declination” (see chapter *How to configure the program*). This file can be extended with more verb declinations (see chapter *List with verb declinations:*).

4.9 List with rejected health claims (module 7):

A list with rejected health claims should be provided to detect resellers, that use exactly these health claims. The file is assumed to be in plaintext and an example is given below:

```
Actimirell aktiviert Abwehrkräfte.
Milchschneideling macht starke Knochen.
...
```

The corresponding keyword in the configuration file is “health_claim_rejected” (see chapter *How to configure the program*). This file can be extended with words to obtain some kind of online learning see section *List with rejected health claims:* for more information.

CHAPTER

FIVE

ANALYSIS MODULES

In this chapter each available module is described in more detail. From this information a user should be able to decide, if a given module suites their purpose. Additionally the used technologies and the return values are described, so an interpretation of the results is easier.

5.1 Address extraction (module 1):

To extract addresses from text, snippets of the text are cut out around 5-digit words, that resemble german postal codes. The snippets are then labeled with an conditional random field to extract the single elements of a possible address, like company name, street name, house number, postal code, city and country name.

The results are returned in form of a list containing dictionaries. Each dictionary contains the extracted address elements, that can be accessed through the corresponding key. The possible keys are: Unternehmen, Strasse, PLZ, Ort, Land, Bundesland, Kreis. If for an address one or more elements haven't been found, the corresponding value is set to None.

5.2 Shop classifier (module 2):

The shop classifier consists of a pipeline of different steps (see chapter *Shop, Food and Product classifier (modules 2, 3, 4):* for more information). The training setup is the following: First, the given text is tokenized so that only words with more than two letters are kept and all special characters are discarded. These tokens are weighted with a common theme (Term frequency inverse document frequency), stopwords are removed and on top of this weighted bag-of-words representation a support vector machine is trained. For predicting the class the same scheme is used.

The result is a probability score which corresponds to the distance to the hyperplane. The default probability score where it is assumed that a website is a shop is 50. This threshold is only used for online learning, so that the user is free to interpret the probability score.

5.3 Foodshop classifier (module 3):

The food classifier consists of a pipeline of different steps (see chapter *Shop, Food and Product classifier (modules 2, 3, 4):* for more information). The training setup is the following: First, the given text is tokenized so that only words with more than two letters are kept and all special characters are discarded. These tokens are reduced to words in a fixed vocabulary and weighted with a common theme (Term frequency inverse document frequency) and on top of this weighted bag-of-words representation a support vector machine is trained. For predicting the class the same scheme is used.

The result is a probability score which corresponds to the distance to the hyperplane. The default probability score where it is assumed that a website contains food is 40. This threshold is only used for online learning, so that the user is free to interpret the probability score.

5.4 Product page classifier (module 4):

The product classifier consists of a pipeline of different steps (see chapter *Shop, Food and Product classifier (modules 2, 3, 4):* for more information). The training setup is the following: First, the given text is tokenized so that only words with more than two letters are kept and all special characters are discarded. These tokens are weighted with a common theme (Term frequency inverse document frequency) and on top of this weighted bag-of-words representation a support vector machine is trained. For predicting the class the same scheme is used.

The result is a probability score which corresponds to the distance to the hyperplane. The default probability score where it is assumed that a website offers a specific product is 50. This threshold is only used for online learning, so that the user is free to interpret the probability score.

5.5 Extracting product information (module 5):

This module extracts the product number and the product name. For the product number a regular expression is used. It is assumed, that before the product number a commonly used word indicates, that the following word is the product number. For the product name the title of the website is extracted and labeled with a conditional random field. Afterwards all words between the first positive labeled and the last positive labeled word are returned as product name. Both return values are strings.

5.6 Checking ecological traders (module 6):

To identify and validate ecological traders different kind of analysis are used. First the text is searched with a regular expression if any word contains “bio”, “öko”, “biologisch”, “ökologisch”. Afterwards a regular expression matching the specific german “Ökonummer” (DE-ÖKO-000) is used to extract all german “Ökonummern”. These are validated against a list with all valid “Ökonummern”. The result for this analysis is a dictionary with the keys “ads”, “fake” and “legal”, containing a list each with all ad-words, all not valid “Ökonummern” and all valid “Ökonummern” found in the text.

Furthermore if a screenshot of the website is available, it is searched for the official EU logo, that ecological traders must display. This analysis returns a dictionary with the key “logos” where the number of logos is stored. This dictionary can be combined with the dictionary from the text analysis to obtain a dictionary containing all results relevant for this module.

5.7 Checking health claims (module 7):

To detect possible and rejected health claims there are three strategies available. The first strategy searches only for suspicious substances and diseases, the second strategy searches for rejected health claims and the third strategy uses semantic parsing of language to detect relevant relationships with suspicious substances and diseases.

The first strategy searches the text for given substances and diseases. The found words can occur anywhere in the text and don’t stand always in a relation to each other. Here just the occurence of these words is enough to arouse suspicion. When at least one disease is found in the text, a list with the substances and a list with the diseases is returned. Else only two empty lists are returned.

The second strategy searches for already rejected health claims. These health claims have to be in the right language and only occurences with the exact wording and setting are found. If some rejected health claims are found a list with these health claims is returned, otherwise only an empty list is returned.

The third strategy uses a semantic parser to get more detailed relations between suspicious words. Each sentence of the text is parsed and the verb is identified. If the verb is relevant for this context, the parsed sentence is reported. To give an additional ranking of the reported sentences, the occurence of suspicious substances and diseases is counted and reported along the sentence. The return value is a list with lists containing the parsed sentence (a dictionary with the different phrases) and a ranking value. If no relevant verbs were found, an empty list is returned.

5.8 Checking PDO, PGI and TSG (module 8):

To identify products that are registered in the EU door list and therefor have a certificate (PDO, PGI or TSG) the product name has to be extracted from the website (compare module 5). In order to check a given product name against the entries in the door list a normalization scheme has to be applied to both sides. One complication in this matter is that the door list contains only product names in the original language. This can lead to worse results for products in other languages than german, because only german words are normalized. All other product names are just splitted on whitespace characters and converted to lower case. For the normalization step all words are stemmed and stop words are removed.

To search a product name in the preprocessed door list, the product name is normalized with the german scheme and the scheme for foreign languages. After that all possible n-grams of the product name are searched for in the preprocessed door list and the corresponding cerfiticate group (PDO, PGI or TSG) is returned when the product name was found. If the product name couldn't be found in the preprocessed door list None is returned.

5.9 Extracting ingredients (module 9):

This module extracts the list of ingredients from a website. To do so three different pretrained statistical models and a conditional random field are used. At first, a regular expression is used to extract at least 150 words after the word "Zutaten", which has to be in front of a list of ingredients after EU law. Each extracted word is assigned the probability, with which it is in a list of ingredients, the probability, with which it is in a normal text, and the probability, with which it is at the end of a list of ingredients. For the last probability the context is considered too and the maximum probability is taken. On the three assigned probabilities a conditional random field is used to determine the end of the list of ingredients. The return value consists of a list of results for every occurence of the word "Zutaten" in the text. For each word a dictionary with all words from the determined list of ingredients (key: "ingredients") and the count of occurences of the last word as indicator for the likelihood of this exact end (key: "count") is added to the list.

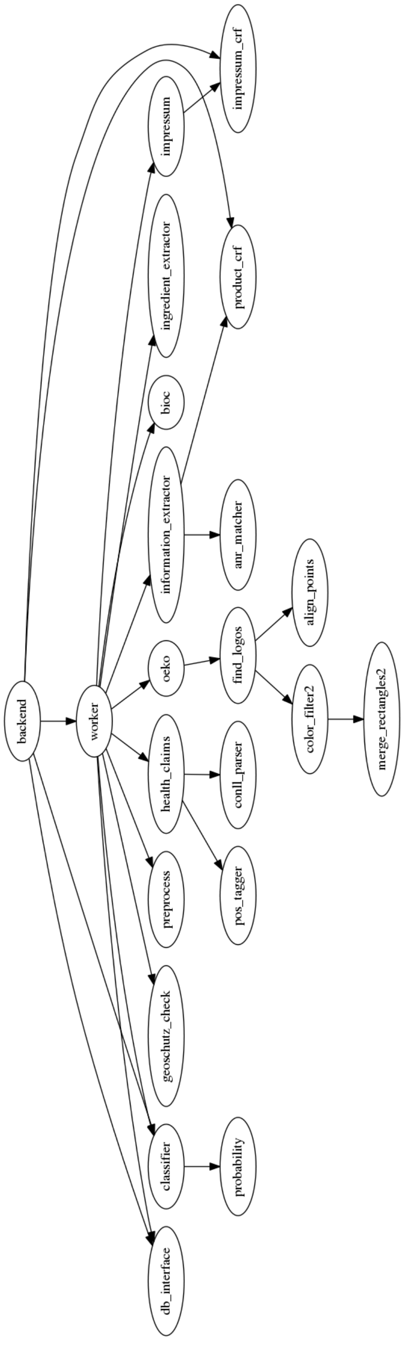
5.10 Checking BioC (module 10):

This module validates the EU certificate for ecological traders against one version of the BioC database. This version is from February 2018. To check if a trader has a certificate, this modules takes a normalized address and looks this address up in a pre-build dictionary. If there exists at least one certificate for this address, the information from this certificate is returned in a dictionary. The dictionary contains the following keys and values: "numbers": all "Ökonummern" stored within the BioC for the found certificate; "periods": the periods in which the certificate is valid (given by day, month and year in a dictionary); and "address": the address that was used in the certificate.

CHAPTER

SIX

OVERALL STRUCTURE



CHAPTER

SEVEN

HOW TO USE ONLINE-LEARNING WITH THE PROGRAM

Some classifiers and external data files can be updated on a regular basis to obtain some kind of oneline-learning for the backend. The modules 2, 3, 4, 7 and 9 can benefit from updating. In which way each module can be improved by updating the classifiers or external data is described in the following chapter. Other modules, that rely on external data, such as modules 1, 6 and 8 can't be updated to improve the results. Nevertheless the external data for these modules should be kept up to date to prevent false results.

7.1 The shop, food and product classifier (modules 2, 3, 4):

It is assumed, that in your workflow, you reevaluate at least some results from the classifiers. This manual classification results should be transfered back to the database table containing the results. In the database table "results" the column "manual_analysis" is for this purpose. When a manual classification result of one of the modules 2, 3 or 4 is transfered back, the flag "validation_result" should be set to 1. After using this information for online-learning the flag "updated_results" is set to 1.

For a given interval (see chapter *How to configure the program*) the program extracts all results, for which the validation flag is 1. For all these results the value of the manual analysis is compared to the value of the automatic analysis for each of these modules and only when they differ the corresponding module is refined with online-learning on this data point.

7.2 Adding words to the food vocabulary (module 3):

The food classifier (module 3) uses a fixed vocabulary to classify text. You can add more words to this vocabulary by extending the file "food_vocab" (see chapter *How to configure the program*). The file contains in every line one word, like:

```
Apfel
Banane
Citrone
...
```

To ensure, that your words can succesfully be used, follow this format. The words are transformed to lower characters and the file is assumed to be in utf-8 encoding.

7.3 Adding information to the white- and blacklist for ingredients (module 9):

Even though the white and blacklist aren't directly used to build the statistical word model for ingredints list, they are used to find possibly not allowed ingredients and prohibited ingredients. To add new words to this list, you can extend the files "ingredients_whitelist" and "ingredients_blacklist" (see chapter *How to configure the program*).

Both files have the following format:

```
Milch
Zitronen
flüssig
...
```

So each line contains one word. If there are some words, that are only meaningful together, you can add both words in one line. But note, that in this case only the two words seperated with the exact same character (e.g. space) are searched for. The words are transformed to lower characters and the file is assumed to be in utf-8 encoding.

7.4 Adding information to several lists to check health claims (module 7):

To detect possible health claims, several different strategies can be used (see section *Checking health claims (module 7):*). For each strategie several external information is used and can be extended. There exist four files containing the different informations: one file with possibly used substances in health claims, one file with possibly used diseases in health claims, one file with rejected health claims and one file with relevant verbs. The files are described in chapter *External data* in more detail.

7.4.1 List with substances:

This file should contain substances, that are often used in health claims. The file is in the following format:

```
Vitamin C
Eisen
...
```

The list contains also phrases with more than one word and you can extend this list with these too. Note that only occurences with the excat same delimiting character (e.g. space) are searched for. The words are transformed to lower characters and the file is assumed to be in utf-8 encoding.

7.4.2 List with disease:

This file should contain disease, that are often used in health claims. The file is in the following format:

```
Herzinfarkt
rote Blutkörperchen
...
```

The list contains also phrases with more than one word and you can extend this list with these too. Note that only occurences with the excat same delimiting character (e.g. space) are searched for. The words are transformed to lower characters and the file is assumed to be in utf-8 encoding.

7.4.3 List with verb declinations:

This file contains different verbs with relevant declinations. Based on this list, sentences with not relevant verbforms are filtered out and not considered a possible health claim. The file is in the following format:

```
beitragen
trug bei
...
```

The list contains also phrases with more than one word and you can extend this list with these too. Note that only occurences with the excat same delimiting character (e.g. space) are searched for. The words are transformed to lower characters and the file is assumed to be in utf-8 encoding.

7.4.4 List with rejected health claims:

This file should contain rejected health claims translated to german. The file is in the following format:

```
Actimirell aktiviert Abwehkräfte.
Milchschneideling macht starke Knochen.
...
```

The list contains only phrases with multiple words. Only a simple search is performed, where the exact wording (with delimiters and punctuation) is found. But the words are transformed to lower characters and the file is assumed to be in utf-8 encoding.

CHAPTER EIGHT

HOW TO TRAIN AND TEST THE SINGLE CLASSIFIERS

This chapter explains the necessary files and formats to train or test the single classifiers. The programs offers a commandline interface for this task, which is explained in chapter *Commandline Interface*. Some modules presented in chapter *Analysis modules* use classifiers to analyse and extract data. These modules are the shop (module 2), food (module 3) and product (module 4) classifier, the address extraction (module 1) and the product name extraction (module 5) modules.

If you want to measure the performance of more than one module you should add single jobs to the database and let the program run normally (see chapter XX and XX).

8.1 Shop, Food and Product classifier (modules 2, 3, 4):

The classifier for the shop, food and product domain consists of a support vector machine with a bag-of-words and a random forest as feature selection. For each domain some parameters were chosen by cross-validation on a specific training data set and are hard-coded in the program. So they can not easily be changed or evaluated. The parameters relate to the feature extraction, the feature selection and a hyperparameter from the support vector machine.

The chosen parameters for each domain are:

pipeline step	parameter	shop	food	product
tfidf	stop-word removal	yes	no	no
	fixed vocabulary	no	yes	no
random forest	nof estimators	100	0	100
	threshold	0.0004	0	0.0004
svm	alpha	0.00001	0.0001	0.0000001

To train a classifier for one of the three domains, a training set is needed. It should consist of either texts from webpages in utf-8 encoding or directly of stored webpages. For each webpage in the training set the corresponding class should be manually assigned (e.g. shop or no shop) for better performance of the trained classifier.

A special directory structure is assumed by the training and testing methods of the classifier. A directory with two subdirectories containing files for each class should be provided, like so:

```
train/
  0/
    file1.html
    file2.html
    file3.html
    ...
  1/
    file4.html
```

(continues on next page)

(continued from previous page)

```
    file5.html
    file6.html
    ...
```

To obtain the right measurements while testing, all positive examples (like shop, food or product) should be within the directory 0/ while all negative examples (like no shop, no food or no product) should be within the directory 1/. The calculation of precision and recall is based on the assumption, that the classes are assigned this way.

In general the performance of the classifier can be measured with a test set in the same directory structure as the training set. The test set should not be used for training.

8.2 Address extraction (module 1):

The extraction of addresses is learned with a conditional random field (CRF). To train a CRF a set of labeled sequences is needed. Each sequence contains text seperated by whitespace (except newline) characters and to each token a label has to be assigned. When a token is irrelevant the label OT (other) can be assigned. The module extracts the tokens with the following labels: FN (company name), ST (street), NR (house number), PLZ (postal code), CI (city) and CO (country).

To train a CRF one file with the text (x-file) and one file with the labels (y-file) has to be provided. The files can look somewhat like this:

```
x-file.txt
  Hier sitzt die Firma: Musterman AG A-Straße 123 98765 Krautheim komm vorbei
  Trifft dich Hansi Hinterseer B-Straße 456 12345 Alpenort Deutschland schönes Date!

y-file.txt
  OT OT OT OT FN FN ST NR PLZ CI OT OT
  OT OT OT OT ST NR PLZ CI CO OT OT
```

Note: The lines of text correspond with the lines of labels and not every label has to be present in every sequence (e.g. the first line contains no CO label and in the second example “Hansi Hinterseer” is a name of a person and not a company, so not labeled FN). But you should only add examples you want the CRF to recognize to your training data set. So leave out all the address fragments, when you don’t want to extract them too.

To test the performance of your CRF you can provide an unseen x-file and corresponding y-file to the classifier. There is functionality in the program to measure the performance of the trained CRF for every label.

8.3 Product name extraction (module 5):

Like the address extraction, the product name extraction uses a CRF to label a token sequence. The label set consists of two different labels: OT (other) and AN (article name). The input to the pretrained CRF were titles of webpages and their manually assigned labels. The titles were tokenized, so that only words of minimum length 2 and special characters are extracted. You should tokenize your input for training or testing in a similar way.

An example input looks like this:

```
x-file.txt
  Amore kaufen : Bratwurst Henning 180 gr - nur hier
  laden online - salzige gurken , lose
y-file.txt
  OT OT OT AN AN AN AN OT OT OT
  OT OT OT AN AN AN AN
```

To test the performance of your CRF you can provide an unseen x-file and corresponding y-file to the classifier. There is functionality in the program to measure the performance of the trained CRF for every label.

CHAPTER

NINE

COMMANDLINE INTERFACE

The commandline interface for the program allows the user to perform multiple tasks. The single tasks can be chosen by subcommands. In the following chapter, the description for each subcommand is shown. The same output is available via the –help flag.

9.1 General interface:

The general interface offers subcommands for training, testing, updating more testing and the usage of the program as tool. It contains the following subcommands and optional arguments:

```
usage: backend.py [-h] [--config CONFIG] [--debug]
                  {train,test,update,load,run} ...

Backend for the AAPVL-Project.

positional arguments:
 {train,test,update,load,run}
   train               train different classifier
   test                test different classifier directly and test different␣
↪functionalities
   update              update the different classifier with new data from a directory
   load                load data from a directory into database. this can be used for␣
↪testing
   run                 run the backend and process jobs

optional arguments:
 -h, --help            show this help message and exit
 --config CONFIG       line based configuration file
 --debug               enable debug-information in log-file
```

The help messages for the subcommands load and run are shown here in detail, while the help messages for the other subcommands are shown in the following sections.

Commandline interface for subcommand load:

```
usage: backend.py load [-h] [--modules MODULES] dir

positional arguments:
  dir                path to directory from which all files are added to the database

optional arguments:
  -h, --help         show this help message and exit
  --modules MODULES  comma separated list of modules that are added to the database␣
↪for every file. if omitted, all modules are registered
```

(continues on next page)

(continued from previous page)

Commandline interface for subcommand run:

```
usage: backend.py run [-h]

optional arguments:
  -h, --help  show this help message and exit
```

9.2 Interface for training:

Commandline interface for subcommand train:

```
usage: backend.py train [-h] {shop,food,product,imp,prod-name} ...

positional arguments:
  {shop,food,product,imp,prod-name}
    shop                train the shop classifier with the data from DIR
    food                train the shop classifier with the data from DIR
    product             train the product classifier with the data from DIR
    imp                 train the crf for address extraction
    prod-name           train the crf for product name extraction

optional arguments:
  -h, --help            show this help message and exit
```

Commandline interface for subcommand train shop:

```
usage: backend.py train shop [-h] dir

positional arguments:
  dir         directory with data

optional arguments:
  -h, --help  show this help message and exit
```

Commandline interface for subcommand train food:

```
usage: backend.py train food [-h] dir

positional arguments:
  dir         directory with data

optional arguments:
  -h, --help  show this help message and exit
```

Commandline interface for subcommand train product:

```
usage: backend.py train product [-h] dir

positional arguments:
  dir         directory with data

optional arguments:
  -h, --help  show this help message and exit
```

Commandline interface for subcommand train imp:

```
usage: backend.py train imp [-h] x y

positional arguments:
  x           file with addresses or titles of websites respectively in each line
  y           file with label sequences in each line corresponding to the tokens in X

optional arguments:
  -h, --help  show this help message and exit
```

Commandline interface for subcommand train prod-name:

```
usage: backend.py train prod-name [-h] x y

positional arguments:
  x           file with addresses or titles of websites respectively in each line
  y           file with label sequences in each line corresponding to the tokens in X

optional arguments:
  -h, --help  show this help message and exit
```

9.3 Interface for testing:

Commandline interface for subcommand test:

```
usage: backend.py test [-h] {shop,food,product,imp,prod-name} ...

positional arguments:
  {shop,food,product,imp,prod-name}
    shop                test the shop classifier with the data from DIR
    food                test the shop classifier with the data from DIR
    product             test the product classifier with the data from DIR
    imp                 test the crf for address extraction
    prod-name           test the crf for product name extraction

optional arguments:
  -h, --help            show this help message and exit
```

Commandline interface for subcommand test shop:

```
usage: backend.py test shop [-h] dir

positional arguments:
  dir         directory with data

optional arguments:
  -h, --help  show this help message and exit
```

Commandline interface for subcommand test food:

```
usage: backend.py test food [-h] dir

positional arguments:
  dir         directory with data
```

(continues on next page)

(continued from previous page)

```

optional arguments:
  -h, --help  show this help message and exit
```

Commandline interface for subcommand test product:

```
usage: backend.py test product [-h] dir

positional arguments:
  dir         directory with data

optional arguments:
  -h, --help  show this help message and exit
```

Commandline interface for subcommand test imp:

```
usage: backend.py test imp [-h] x y

positional arguments:
  x           file with addresses or titles of websites respectively in each line
  y           file with label sequences in each line corresponding to the tokens in X

optional arguments:
  -h, --help  show this help message and exit
```

Commandline interface for subcommand test prod-name:

```
usage: backend.py test prod-name [-h] x y

positional arguments:
  x           file with addresses or titles of websites respectively in each line
  y           file with label sequences in each line corresponding to the tokens in X

optional arguments:
  -h, --help  show this help message and exit
```

9.4 Interface for updating:

Commandline interface for subcommand update:

```
usage: backend.py update [-h] {shop,food,product,imp,prod-name} ...

positional arguments:
  {shop,food,product,imp,prod-name}
    shop                update the shop classifier with the data from DIR
    food                update the shop classifier with the data from DIR
    product             update the product classifier with the data from DIR

optional arguments:
  -h, --help            show this help message and exit
```

Commandline interface for subcommand update shop:

```
usage: backend.py update shop [-h] dir

positional arguments:
  dir         directory with data

optional arguments:
  -h, --help  show this help message and exit
```

Commandline interface for subcommand update food:

```
usage: backend.py update food [-h] dir

positional arguments:
  dir         directory with data

optional arguments:
  -h, --help  show this help message and exit
```

Commandline interface for subcommand update product:

```
usage: backend.py update product [-h] dir

positional arguments:
  dir         directory with data

optional arguments:
  -h, --help  show this help message and exit
```

CHAPTER

TEN

API REFERENCE FOR THE AAPVL BACKEND

10.1 The main entry point: Backend

`backend.`**`create_parser`**`()`

Create the argument parser.

`backend.`**`load_db`**`(`*config*, *directory*, *modules*`)`

Load data in database for broder test.

For every file in the directory a job is submitted to the database. The file is registered for every module in modules. Other options aren't available. Each file is treated, as if it was its own main page.

Parameters

- **config** – directory with important configuration information
- **directory** – path to directory with test data
- **modules** – comma separated string with module numbers

`backend.`**`main`**`()`

Main entry point.

`backend.`**`read_config`**`(`*filename*`)`

Parse the config file.

Parameters **filename** – filename of the config file

`backend.`**`test_clf`**`(`*config*, *directory*, *type_*`)`

Test the different classifier.

Parameters

- **directory** – path to directory with test data
- **type_** – type of the classifier. one of: 'shop', 'food' or 'product'

`backend.`**`test_impressum`**`(`*x_file*, *y_file*`)`

Test the impressum crf.

Parameters

- **x_file** – file with test sequences
- **y_file** – file with test labels for the sequences

`backend.`**`test_product_name`**`(`*x_file*, *y_file*`)`

Test the product name crf.

Parameters

- **x_file** – file with test sequences
- **y_file** – file with test labels for the sequences

`backend.`**`test_simple`**(*config*, *max_tries*, *delay*, *delay_module*, *day*, *update_rate*)

Test the general setup of the backend.

Loads some test jobs into the database and runs every module on them to check the general setup.

Parameters

- **config** – dictionary with important configuration information
- **max_tries** – maximal number of tries for worker
- **delay** – delay in seconds after no job was found
- **delay_module** – time in seconds a module is allowed to take
- **day** – number of times to check the database for jobs per day
- **update_rate** – number of days after which online learning should be performed

`backend.`**`train_clf`**(*config*, *directory*, *type_*)

Train the different classifiers.

Parameters

- **directory** – path to directory with training data
- **type_** – type of the classifier. one of: 'shop', 'food' or 'product'

`backend.`**`train_impressum`**(*x_file*, *y_file*)

Train the impressum crf.

Parameters

- **x_file** – file with training sequences
- **y_file** – file with training labels for the sequences

`backend.`**`train_product_name`**(*x_file*, *y_file*)

Train the product name crf.

Parameters

- **x_file** – file with training sequences
- **y_file** – file with training labels for the sequences

`backend.`**`update_clf`**(*config*, *directory*, *type_*)

Update the different classifiers with online learning.

Parameters

- **directory** – path to directory with new training data
- **type_** – type of the classifier. one of: 'shop', 'food' or 'product'

10.2 This is where the magic happens: Worker

`class` `worker.`**`Worker`**(*config*, *delay_module=180*)

Worker combines all the modules to get the jobs and process them.

`_delay_module`

time after that a module should be stopped

_interface
interface to the database

_lm_theta
probability threshold for the lm-classifier

_shop_theta
probability threshold for the shop-classifier

_product_theta
probability threshold for the product-classifier

_shop_clf
shop classifier

_food_clf
food classifier

_product_clf
product classifier

_p
preprocessor for websites

_imp
impressums handler

_extractor
information extractor

oeko
oeko module

health
health claim module

geo
geoschutz module

ingr
ingredients module

bioc
bioc module

__init__(*config*, *delay_module=180*)
Initialize all the needed modules.

Parameters **config** – dictionary with connection details for db_interface and modules

Keyword Arguments **delay_module** – time in seconds, after which each module is killed

get_job_and_process()
Get a job from database, calculate results, update database.

online_training_clfs()
Online train the classifiers.

From the database all data, that was manually evaluated and not yet used for training is selected. With a filter only changed values are used to train the three classifiers. The classifiers are then updated with online learning and all used data is marked as such.

process_job(*input_file*, *input_image*, *jobs_str*)
Process a job and return the results of the single modules.

Parameters

- **input_file** – path to file that should be processed. can be an url
- **input_image** – path to a screenshot of the file
- **jobs_str** – comma separated string with list of modules that should be used

schedule(*max_tries*, *delay*, *day*, *update_rate*)

Register a worker for processing jobs.

`get_job_and_process()` is called periodically to process all available jobs. If there is no job found, the worker sleeps for a given amount of time before trying to get a job again. Either after a specific number of tries the worker is shut down or the worker tries always to get a new job a specific number of times a day. At a given intervall the database is checked for manually evaluated data that is newly available.

Parameters

- **max_tries** – the maximal number of tries, the worker should check for new jobs. if -1, the worker won't stop checking
- **delay** – the time in seconds the worker should sleep between checking for jobs
- **day** – number of times the worker should check for new jobs per day. only used with max_tries = -1
- **update_rate** – number of days the worker should wait before checking for newly available manually evaluated data

shutdown()

Shut down the worker and close connection to database.

worker.**handler**(*signum*, *frame*)

Handle too long executions of modules.

Parameters

- **signum** – signal number, not used
- **frame** – not used

10.3 The Classifier (Shop, Food, Product):

class classifier.**Classifier**(*config*, *directory=None*, *new=False*, *type_='shop'*)

A classification pipeline for different settings.

Classifier implements an abstract interface to the underlying SVM for shop, food and product websites classification. It performs feature selection, feature extraction and classification. The different parameters for all three classifier instatiations are choosen with cross-validation and are hard coded.

pipeline

pipeline of a TfidfVectorizer, a RandomForrest for feature selection and a linear SVM.

type_

a string that describes the instantiation. Valid strings: 'shop', 'food' and 'product'

filename

filename of the file where the pickled classifier is stored.

prob

an instance of `Probability` that is trained on the same data.

`__init__`(*config*, *directory=None*, *new=False*, *type_='shop'*)

Initialize the pipeline.

Constructor to initialize the pipeline. This function tries to load the classifier from the file `self.filename`. If this file does not exist or new is True, a new classifier is trained and stored.

If the file `self.filename` does not exist or new is True, and directory is None the function ends with an error

Parameters config – dictionary with important configuration information

Keyword Arguments

- **directory** – path to the directory with the training samples (default: None)
- **new** – if True, a new classifier is trained with the set from directory (default: False)
- **type_** – specifies the type of the classifier. Valid strings:'shop', 'food', 'product'. (default: 'shop')

`load_files`(*container_path*)

Load the files in container_path.

A specific structure of directories is assumed within container_path. The names of the subdirectories are taken as target names and the data within the subdirectories is assumed to be within different classes. The assumed structure is:

container_path/

0/ file1.html file2.html …

1/ file3.html file4.html …

All the files within the subdirectories are preprocessed with BeautifulSoup. So HTML-files can directly be used as input for training and testing purposes. BeautifulSoup handles plain text files well, so that they can also be used for loading with this function. Even directories with mixed types of text files are possible.

Parameters container_path – path to the containing directory

`predict`(*data*)

Predict classes for data.

Predicts the class label for all data points in data.

Parameters data – list of data points to be classified.

`predict_prob`(*data*)

Predict the membership probability for the positive class.

Predicts the probability for all data points in data to be a member of the positive class.

Parameters data – a list of data points to be classified.

`test`(*test_data*, *prob=False*)

Test `self.pipeline`.

Tests the trained pipeline with the data from test_data. After predicting the label, the true positives, false positives, false negatives, true negatives, precision, recall and accuracy are calculated and printed on stdout.

Parameters test_data – path to the directory with the test data. The same directory structure as for `train()` is assumed.

Keyword Arguments prob – if True, `predict_prob()` is used to get the classification instead of `predict()`. (default: False)

train(*training_data*, *config*)

Create and train a pipeline.

For Training the data in training_data is used. After training the pipeline is stored in the file `self.filename`.

Parameters

- **training_data** – path to the directory with the training data. A special directory structure is assumed: the directory training_data should contain two sub-directories '0' and '1'. Sub-directory '0' contains the positive examples and sub-directory '1' contains the negative examples (e.g. when `self.type_` = 'shop', all shops are contained in '0' and all non-shops are contained in '1').
- **config** – dictionary with necessary configuration details.

train_batch(*data*, *labels*)

Refine `self.pipeline` with batch online learning.

This function is for online-learning the classifier with e.g. manually labeled data. It only trains the classifier and not the feature extraction, feature selection and probability distribution part of the pipeline. For training data and labels are used as a batch. After training the classifier, the whole pipeline is stored in the file `self.filename`.

Parameters

- **data** – a list with the text from the manually labeled websites.
- **labels** – a list with the correspondin labels, that were assigned by a human.

train_batch_dir(*directory*)

Wrap `train_batch()` to be used with directories.

Parameters **directory** – path to directory with data.

class probability.**Probability**(*type_*, *pipe=None*, *directory=None*, *new=False*)

A Probability-Calculator for the output of a SVM.

This implementation follows the procedure in [1] with additional changes from [2].

A, B

the calculated weights for the probability function: P(class | input) = 1 / (1 + exp(A * input + B))

Note:

References:

[1] John C. Platt, Probabilistic Outputs for Support Vector Machines and Comparison to Regularized Likelihood Methods, 2000

[2] Lin et al, A note on Platt's probabilistic outputs for support vector machines, 2007

__init__(*type_*, *pipe=None*, *directory=None*, *new=False*)

Initialize the Probability-Calculator.

Tries to read already trained parameters A and B from the file "models/type__sigmoid.txt" and expects them to be in the following format:

```
A <number>
B <number>
```

if new is True or the file doesn't exist, and pipe or directory is None this function ends with an error.

Parameters **type_** – string that indicates the type of the classifier. this string is used to identify the trained sigmoidal function in the future

Keyword Arguments

- **pipe** – pipeline to use to generate the training set (default: None)
- **directory** – path to the directory of training samples (default: None)
- **new** – if True, new parameters will be learned with the training samples in directory (default: False)

calculate_probability(*dec*)

Calculat the probability for dec to be in the positive class.

Parameters **dec** – distance to the hyperplane for this data point

generate_trainingset(*pipeline*, *directory*)

Generate training data fs and ys.

Generates a training set from the samples in directory consisting of the distance to the hyperplane for all points and their real labels (fs and ys). Therefore a 3fold cross-validation with a SVM is performed. The feature selection and parameters are not configurable.

Parameters

- **pipeline** – pipeline of classifier, which is used to generate the trainingset
- **directory** – path to the directory with the initial training samples

read_from_file(*_file_*)

Read parameters A and B from _file_.

Parameters **_file_** – filename of file to read from.

store_to_file(*_file_*)

Store parameters A and B to _file_.

Parameters **_file_** – filename of file to store to.

train_probs(*directory*)

Train the probability calculator.

Generates a training set from the samples in directory and performs a minimization to learn the parameters A and B. The new parameters are not stored in a file. Please use *store_to_file()* for this purpose. If the minimization is not successfull, it is reported on stderr.

Parameters **directory** – path to the directory with the initial training samples

train_probs_orig(*pipe*, *directory*)

Train the probability calculator.

Generates a training set from the samples in directory and performs the original algorithm from [1] to learn the parameters A and B. The new parameters are not stored in a file. Please use *store_to_file()* for this purpose. If the minimization is not successfull, it is reported on stderr.

Parameters **directory** – path to the directory with the initial training samples

10.4 The Interface to the database:

class db_interface.**DBInterface**(*config=None*)

Interface to the used database scheme. Connects with the database and performs all the needed queries.

db

database connection object

__init__(*config=None*)

Constructs a new Interface and connects to the database.

Keyword Arguments config – dictionary which contains important configuration information (default: None)

connect(*config=None*)

Connects to the database.

Keyword Arguments config – dictionary which contains important configuration information (default: None)

disconnect()

Close the open connection to the database.

get_job()

Select one job from the database.

A query on the database is performed to get a new job with all needed information. If no job is found, an empty list, else a list with the needed information is returned.

get_manual_data()

Select all rows where manual results have been altered.

Only rows, that weren't used for online learning before are selected here.

get_results_with_parent_resources(*parent_resources*)

Select the results from a already processed website.

The stored results for a main page can be retrieved with this function. The result contains the keyword "add" where additional information for all subpages is stored.

Parameters parent_resources – fk_resources id for the main page

set_error_state(*id_*)

Set status_jobs to an error-flag for a given pk_jobs.

For the given id status_jobs is set to 9.

Parameters id_ – pk_jobs id

update_results(*input_*, *results*)

Update the status and the result for one job.

Parameters

- **input_** – db information that was returned by `get_job()`.
- **results** – dictionary containing all results.

update_results_with_parent_resources(*pk_results*, *results*)

Update results for a given pk_results id.

Sets both, analysis_results and manual_results, to the given results dictionary.

Parameters

- **pk_results** – pk_results id for a main page
- **results** – dictionary with all information to be stored

update_update_flag(*rows*)

Update rows for being used in online-learning.

For all rows in rows the flag updated_results is set to 1.

Parameters **rows** – rows that have been used for online-learning

10.5 Extraction of text from html:

class preprocess.**Preprocessor**

Methods to preprocess html-files and text.

beautiful_soup(*url*, *links=True*)

Extract text and title of a webpage using beautiful soup.

With this function all the text and all links of a webpage are extracted.

Parameters **url** – url of the webpage or path to the file

Keyword Arguments **links** – if True, all links are extracted and added. (default: True)

preprocess_file(*file_*, *links=True*)

Extract all the text and title of a webpage.

Uses `beautiful_soup()` to extract all the text, all links and the title of a webpage.

Parameters **file_** – path to the webpage.

Keyword Arguments **links** – if True, all links are extracted as well. (default: True)

10.6 Extracting addresses:

class impressum.**Impressum_handler**(*config*)

Extracts addresses from text.

Holds a trained Conditional Random Field to identify addresses in text. The potential addresses are searched with a regular expression, that matches 5-digit numbers followed by text, and labeled with the CRF.

regex

a compiled regular expression to extract regions around a postal code

imp

a trained CRF which is accessible through ImpressumCRF

kr_mapping

a mapping from postal codes to regions

__init__(*config*)

Initialize regex and load the trained CRF.

Parameters **config** – dictionary, that contains configuration. The entry with the key “map_file” is interpreted as a csv file containing a mapping from postal code to regions.

process_text(*t*)

Search and return all addresses in t.

Parameters **t** – the text to process.

class impressum_crf.**ImpressumCRF**(*cities='/home/dorle/Dokumente/data/osm_germany/cities.txt'*, *new=False*, *x_file=None*, *y_file=None*)

A sequence labeling algorithm for german street names using CRFs.

cities

set with all german city names (from wikipedia)

filename

name for the file that is used to store the classifier

crf

conditional random field to label sequences

__init__(*cities='/home/dorle/Dokumente/data/osm_germany/cities.txt'*, *new=False*, *x_file=None*, *y_file=None*)

Initialize the conditional random field.

If new is False, it tries to read the classifier from `self.filename`. If this file does not exist or new is True, a new classifier is trained with the files x_file and y_file.

If new is True and x_file or y_file is None, the function ends with an error.

Keyword Arguments

- **cities** – filename of a city-file (default: city_file)
- **new** – if True, a new classifier is trained (default: False)
- **x_file** – file that contains the training sequences. Each sequence has to be in a single line. (default: None)
- **y_file** – file that contains the label sequences. The label sequences have to be in the same line as the corresponding training sequence in x_file. (default: None)

create_features(*word_list*, *pos*)

Create a feature vector for the word at pos in word_list.

Parameters

- **word_list** – list of words
- **pos** – position in word_list

predict(*samples*)

Predict the labels for every sample in samples.

Parameters samples – list of samples. The samples have to be lists of feature vectors.

seq2feat(*seq*)

Create a list with feature vectors for every word in seq.

Parameters seq – list of words

test(*x_test*, *y_test*)

Test the `self.crf`.

For testing the test_data from x_test and y_test is used. This function prints a metric to stdout.

Parameters

- **x_test** – list with the test sequences as list of words
- **y_test** – list with the label sequences as list of words. The label sequences have to correspond with the test sequences in x_test.

train (*x_list*, *y_list*)
Train a conditional random field.

For Training the CRF the samples from x_list and y_list are used. After training the CRF is stored to `self.filename`. `self.crf` is changed by this method.

Parameters

- **x_list** – list with the training sequences as list of words
- **y_list** – list with the label sequences as list of words. The label sequences have to correspond with the training sequences in x_list.

10.7 Extracting relevant product information:

class `information_extractor.`**InformationExtractor**
Functionality to extract information from product websites.

anr
extractor for "Artikelnummern"

pr_name
crf to extract product names from webpage titles

__init__ ()
Initialize all extractors.

extract_artikelnummer (*text*)
Extract "Artikelnummern" from text.

Parameters **text** – some text

extract_productname (*title*)
Extract productnames from webpage titles.

Parameters **title** – title from a webpage

class `anr_matcher.`**ANRMatcher**
A regular expression wrapper to match 'Artikelnummern'.

complete_re
regular expression that captures some common forms for 'Artikelnummern'

__init__ ()
Initialize the regular expression wrapper.

match (*t*)
Match t with the regular expression.

Parameters **t** – the text, against which `self.complete_re` shall be matched.

class `product_crf.`**ProductnameCRF** (*new=False*, *x_file=None*, *y_file=None*, *cros_val=False*)
A sequence labeling algorithm for product names using CRFs.

filename
name for the file that is used to store the classifier

crf
conditional random field to label sequences

`__init__`(*new=False*, *x_file=None*, *y_file=None*, *cros_val=False*)

Initialize the conditional random field.

If new is False, it tries to read the classifier from `self.filename`. If this file does not exist or new is True, a new classifier is trained with the files x_file and y_file.

If new is True and x_file or y_file is None, the function ends with an error.

Keyword Arguments

- **new** – if True, a new classifier is trained (default:false)
- **x_file** – file that contains the training sequences. Each sequence has to be in a single line. (default: None)
- **y_file** – file that contains the label sequences. The label sequences have to be in the same line as the corresponding training sequence in x_file. (default: None)
- **cros_val** – default: False

`create_features`(*word_list*, *pos*, *l*, *title=set([])*)

Create a feature dictionary for a given word.

The feature dictionary is build for the word at position pos in word_list.

Parameters

- **word_list** – list of words, for which a sequence of feature dictionaries should be created.
- **pos** – position of the current word in word_list
- **l** – length of word_list

Keyword Arguments **title** – a set with all common tokens in titles of websites (default: set())

`load_files`(*x_file*, *y_file*)

Load training data from files.

Parameters

- **x_file** – file that contains the training sequences. Each sequence has to be in a single line.
- **y_file** – file that contains the label sequences. The label sequences have to be in the same line as the corresponding training sequence in x_file.

`predict`(*samples*)

Predict the labels for samples.

Parameters **samples** – list of sequences, for which the labels shall be predicted

`seq2feat`(*seq*)

Create a list with feature dictionaries for seq.

A feature dictionary for every word in seq is created with the method *`create_features()`*.

Parameters **seq** – list with words, for which a sequence of feature dictionaries should be build

`test`(*x_test*, *y_test*)

Test the conditional random field.

The results are printed to stdout.

Parameters

- **x_test** – list with test sequences
- **y_test** – list with test labels. The labels have to be at the same index as the corresponding sequences.

train (*x_list*, *y_list*)

Train a conditional random field.

The trained conditional random field is stored in `self.filename`.

Parameters

- **x_list** – list with all training sequences
- **y_list** – list with all training labels. The labels have to be at the same index as the corresponding sequences.

10.8 Extracting text information about ecological traders:

class `oeko.`**Oeko** (*config*)

Functionality to check 'Biohändler' for correct labelling.

buzz_reg

regular expression matching special buzzwords

num_reg

regular expression matching german 'Ökonummern'

legal_numbers

list of legal 'Ökonummern'

__init__ (*config*)

Initialize attributes.

Parameters **config** – dictionary with important configuration information.

check_image (*image_name*)

Check a screenshot for the mandatory eu logo.

Parameters **image_name** – path to the screenshot

check_text (*text*)

Check text for buzzwords and 'Ökonummern'.

The found 'Ökonummern' are verified if they are legal. Illegal 'Ökonummern' are reported also.

Parameters **text** – text to check

10.9 EU Logo recognition module:

`find_logos.`**find_logos** (*image_name*)

Find logos in an image.

Parameters **image_name** – name of image

`find_logos.`**generate_train_snippets_and_store_them** (*image_list*, *file_name*)

Generate snippets for training.

The generated snippets are stored into countour_name.

Parameters

- **image_list** – list with image names
- **file_name** – not used

find_logos.**get_shapes**(*img*)

Calculate the contours of an image.

Parameters **img** – already loaded image

find_logos.**intersect**(*r*, *pos*)

Intersect two rectangles.

Parameters

- **r** – first rectangle
- **pos** – second rectangle

find_logos.**load_positions**(*file_name*)

Load example positions.

Parameters **file_name** – path to file

color_filter2.**area**(*r*)

Calculate area of rectangle.

Parameters **r** – rectangle

color_filter2.**color_filter**(*img*)

Apply color filter to img.

Returns all rectangles within which the color is according to the color filter.

Parameters **img** – image in opencv image data type

color_filter2.**is_rect_shape**(*c*)

Check if contour is approximately rectangular.

Parameters **c** – contour

color_filter2.**merge_rects**(*rects*)

Merge intersecting rectangles.

Parameters **rects** – list of rectangles

color_filter2.**ratio**(*r*)

Calculate width to height ratio.

Parameters **r** – rectangle

10.10 Detecting possible health claims:

class health_claims.**HealthClaims**(*config*)

Different strategies to check for not allowed health claims.

The first strategy assumes a list with simple substances and a list with diseases (default: './health_claim_substances.txt' and './health_claim_diseases.txt'). The text of all websites registered for this module is searched for every line in these lists. When one or more diseases are found in the text, the found substances and diseases are returned, while just a substance is not enough for a suspicion.

The second strategy searches for all health claims in a list. Therefore a list with prohibited health claims has to be provided (default: './rejected_claims.txt'). In this file every line is interpreted as a health claim and searched for.

The third strategy extracts all the relations from the text. Only relations with a verb phrase contained in a provided file (default: './vps.txt') are returned.

pattern_matcher_sub
a simple pattern matcher that can search for substances

pattern_matcher_dis
a simple pattern matcher that can search for disease

pattern_matcher_fix
simple pattern matcher for fix health claims

sub
list with relevant substances

dis
list with relevant diseases

verbs
set with relevant verbs

pos
Part-of-Speech-Tagger

punkt_name
filename of pretrained sentence tokenizer

punkt
pretrained sentence tokenizer

__init__(*config*)
Initialize all the matchers.

Parameters **config** – dictionary with important configuration information.

check_disease_substances(*text*)
Execute the first strategy.

Parameters **text** – the text that should be searched through

check_fix_patterns(*text*)
Execute the second strategy.

Parameters **text** – the text that should be searched through

check_semantic_relations(*text*)
Execute the third strategy.

Parameters **text** – the text that shoulb be searched

chunks(*l*, *n*)
Yield successive n-sized chunks from l.

class pos_tagger.**POSTagger**(*model_file='models/health_claim_model.crf.tagger'*)
Simple POS-Tagger using conditional random fields.

tagger
the pretrained tagger

__init__(*model_file='models/health_claim_model.crf.tagger'*)
Initialize tagger.

Keyword Arguments **model_file** – path to pretrained model (default: ./models/health_claim_model.crf.tagger')

pos_tag_io()
Read from stdin, tag and write to stdout.

pos_tag_lst (*lst*, *output*)

Tag every word of every sentence in lst and write to output.

Parameters

- **lst** – list of sentences
- **output** – filehandle

class conll_parser.**Entry** (*id_*, *form*, *pos*, *xpos*, *head*, *dep*)

Contains important information for a conll entry

id_

id of the entry

form

actual word

pos

POS-tag

xpos

STTS-POS-tag

head

id of parent node

dep

relationship to parent node

__init__ (*id_*, *form*, *pos*, *xpos*, *head*, *dep*)

Initialize entry.

Parameters

- **id_** – id of entry
- **form** – actual word
- **pos** – POS-tag
- **xpos** – STTS-POS-tag
- **head** – id of parent node
- **dep** – relationship to parent node

class conll_parser.**Node** (*data*)

Node for conll parser tree.

data

data of the represented row

lchilds

list of left childs, sorted by `self.data.id_`

rchilds

list of right childs, sorted by `self.data.id_`

__init__ (*data*)

add_child (*id_*)

Add a child to this node.

This function assumes, that the actual node is already initialized. Only the id is added to one of the list of children.

Parameters **id_** – id of the node to add

add_lchild(*id_*)

Add a left child to this node.

This function assumes, that the actual node is already initialized. Only the id is added to the list of left children.

Parameters **id_** – id of the node to add

add_rchild(*id_*)

Add a right child to this node.

This function assumes, that the actual node is already initialized. Only the id is added to the list of right children.

Parameters **id_** – id of the node to add

get_children()

Return a list with all children.

get_dep()

Return the dependency relation retrieved from `self.data`.

class conll_parser.**Tree**(*nodes*)

Dependency tree for one sentence.

This structure models the dependency relations of single phrases found with an dependency parser. It is constructed from the information in the conll format.

root

id of root node (always 0)

nodes

list of nodes

__init__(*nodes*)

Initialize tree.

This function only adds all nodes without dependencies to the tree. The structure has to be formed with *create_tree_structure()*.

Parameters **nodes** – list of entries

create_tree_structure(*head_indexed*)

Create the dependency structure.

This function creates the given dependency relationship between single nodes within the tree.

Parameters **head_indexed** – dictionary mapping id to a list of children

get_leftest_child(*id_*)

Return leftest child under a node.

Parameters **id_** – id of the node

get_rightest_child(*id_*)

Return rightest child under a node.

Parameters **id_** – id of the node

get_rightest_noun_part(*id_*)

Return the rightest part of a noun phrase.

Expands the noun phrase starting at a node heuristically to the right.

Parameters **id_** – id of the node

get_subtree(*id_*)

Return subtree under a node.

Parameters **id_** – id of the root for the subtree

string_from_to(*i*, *j*)

Return the string in the intervall from i to j.

Parameters

- **i** – id of left border
- **j** – id of right border

conll_parser.**get_relation**(*t*)

Splits tree at root and returns the relevant strings.

Parameters **t** – tree to split

conll_parser.**parse**(*text*)

Parse text and return a list of lists.

Parameters **text** – conll formated text for possibly more than one sentence

conll_parser.**parse_int_value**(*value*)

Parse int from string handling malformed ints.

Parameters **value** – value to parse

conll_parser.**parse_line**(*line*)

Parse one line of conll format and return Entry.

Parameters **line** – one line of conll format

conll_parser.**parse_tree**(*text*)

Parse text into dependency trees.

A list of trees is created containing one tree per sentence.

Parameters **text** – conll formated text for possibly more than one sentence

conll_parser.**split_tree**(*t*)

Split tree at root node into relevant phrases.

This function returns the intervalls of the phrases.

Parameters **t** – tree to split

10.11 Textanalysis for protected designation of geographical origin:

class geoschutz_check.**Geoschutz**(*config*)

Functionality to detect EU-certificated product names.

s

dictionary with the keys ‘PGI’, ‘PDO’ and ‘TSG’. For each key all the certificated productnames are stored in a set.

stemmer

a german stemmer

stop_words

a set with german stopwords

max_n
the length of the longest productname contained in s

compound_reg
regular expression with common delimiters for compound nouns

__init__(*config*)
Initialize all attributes.

Parameters **config** – dictionary with important configuration information

create_ngrams(*lst*, *min_n=1*, *max_n=1*)
Create n-grams from a list.

All n-grams for n in [min_n, max_n] are created.

Parameters **lst** – list of words

Keyword Arguments

- **min_n** – minimal length of an n-gram (default: 1)
- **max_n** – maximal length of an n-gram (default: 1)

remove_stop_words(*lst*)
Remove german stop words.

Parameters **lst** – list with words

search(*word*)
Search a productname in all the certificated productnames.

For the search word is normalized in the same ways, as all the productnames in `self.s`.

Parameters **word** – productname to search for in all certificated productnames.

stem_all_words(*lst*)
Stem all words.

A german stemmer is used, so words from other languages may be stemmed incorrectly. The result can be compared to other stemmed words, because the procedure is deterministic.

Parameters **lst** – list with words

10.12 Extracting ingredients list:

class ingredient_extractor.**IngredientExtractor**(*config*, *filename='models/crf_vocab_stuff_europarl_complete_count.pkl'*)
Statistical Word Model to extract ingredients lists correctly.

A conditional random field is used to determine the correct borders depending on the probabilities given by the statistical word model.

vocabulary
dictionary, mapping all known ingredients to their frequence

trigram_model
dictionary, mapping word-triples to their number of occurence at the end of seen ingredients lists

base_model
dictionary, mapping words from european parliament speeches to their frequence

`crf`
conditional random field to determine the correct borders

`whitelist`
a set with known and allowed ingredients

`blacklist`
a set with known and prohibited ingredients

`zutaten_pat`
regular expression to match the beginning of an ingredients list

`token_pattern_just_words`
regular expression to tokenize all words with more than 2 characters and discard the rest

`__init__`(*config*, *filename='models/crf_vocab_stuff_europarl_complete_count.pkl'*)
Initialize the models.

Parameters **config** – dictionary with important configuration information

Keyword Arguments **filename** – path to a pickled file containing the trained statistical word model and the conditional random field (default: models/crf_vocab_stuff_europarl_complete_count.pkl)

`blacklist_check`(*lst*)
Check if an ingredient from lst is prohibited.

Parameters **lst** – ingredient list

`create_feat_lists`(*tokens*)
Create lists with probabilities for each token.

For each token the probability to be in an ingredient list, not to be in an ingredient list and to be at the end of an ingredient list is calculated. For convenience, three lists are returned containing all the probabilities to be in an ingredient list, not to be in an ingredient list and to be at the end of an ingredient list.

Parameters **tokens** – a list of tokens to evaluate

`create_vec_list`(*tokens*)
Create a list of vectors for the crf for tokens.

Parameters **tokens** – a list of tokens to evaluate

`extract`(*text*)
Extract the ingredient list and calculate an occurence value.

The occurence value is an estimate for how often an ingredient list with the same ending has been seen previously.

Parameters **text** – text to extract the ingredient list from

`extract_zutaten`(*text*)
Extract tokens from all possible ingredients list.

Parameters **text** – text to extract from

`prob_end`(*w2*, *w1*, *w*)
Calculate probability for words to be at the end.

To obtain a better estimate of the probability for a word, this function should be used three times for each word. The following setups should be used to evaluate the last word, given a list of tokens l: prob_end(l[-3], l[-2], l[-1]), prob_end(l[-2], l[-1], None) and prob_end(l[-1], None, None).

Parameters

- **w2** – word preceding the word to evaluate by 2
- **w1** – word preceding the word to evaluate by 1
- **w** – word to evaluate

`prob_lm`(*w*)
Calculate probability for a word to be in an ingredients list.

Parameters **w** – word to evaluate

`prob_nlm`(*w*)
Calculate probability for a word not to be in an ingredients list.

Parameters **w** – word to evaluate

`tokenize_words`(*t*)
Tokenize a text.

Parameters **t** – text to tokenize

`whitelist_check`(*lst*)
Check if an ingredient from lst is not known.

Parameters **lst** – ingredient list

10.13 Reconciliation with BioC information:

`class bioc.BiocStore`
Functionality to check certificates from an offline storage.

The information about certificates is taken from the BioC database. Therefore all the certificates where downloaded and preprocessed in january 2018.

`oeko_re`
regular expression to extract the german 'Ökonummer'

`filename`
path to a pickled bioc_store

`info_dict`
dictionary mapping unique ids to certificate information

`mapping`
dictionary mapping all normalised addresses to ids used in info_dict

`__init__`()
Initialize the store.

`build_dict`(*directory*)
Build dictionaries for certificates and store them.

The certificates should be stored in a directory with subdirectories. Each subdirectory contains websites with certificate information, while directory only contains subdirectories.

Parameters **directory** – path to a directory with a given structure.

`check_validity`(*valids*)
Check the validity of a certificate.

TODO: this is outdated and needs to be refactored

Parameters **valids** – list with valid periods

extract_all(*filename*)

Extract all certificate information from a website.

For a certificate the following information is extracted and normalized: all addresses, all periods where the certificate is valid, the responsible 'Ökokontrollstellen'.

Parameters **filename** – path to a website, could also be an url.

get_certificate_for_addresses(*addresses*)

Get certificate information from the store for some addresses.

Parameters **addresses** – list of normalised addresses

10.14 Utils:

exception my_exceptions.**TooLongException**(*value*)

Exception, when a module takes to long.

value

message of the exception

__init__(*value*)

Initialize the exception.

Parameters **value** – message of the exception

class simple_check.**SimpleCheck**

Wrapper for approximative string matching.

matchers_lst

list of lists of different matchers that are used in every check method

words_lst

list of lists of the corresponding search terms

__init__()

Initialize the list of matchers and words as empty.

add_list(*lst*)

Add a matcher for every word in lst.

Adds a list of words that are searched for in the texts. The list should contain a word with multiple synomymous words.

Parameters **lst** – list of words that are added

lst_distance(*lst*, *distance*)

Filter the matches within a given distance.

Parameters

- **lst** – list with matching results
- **distance** – allowed distance in characters

range_check_text(*text*, *distance*, *k=0*)

Check for search terms within a given distance.

Parameters

- **text** – text to be searched
- **distance** – allowed distance of matches in characters

Keyword Arguments **k** – number of acceptable errors (default: 0)

`simple_check_text`(*text*, *k=0*)

Check the text for occurences of terms.

Parameters **text** – text that should be searched

Keyword Arguments **k** – number of acceptable errors (default: 0)

class `approx_str_matching.`**`Matcher`**(*pattern*)

This class implements a fast text searching algorithm allowing for errors. This Algorithm is implemented after [0]. For every Pattern that should be searched in a text, there has to be a little preprocessing. So the Matcher Objects are for one pattern each. The Matcher needs the pattern upon its initialization, so the pattern has to be passed to the constructor.

`s`

default dictionary, that contains a bitvector for each character in the pattern. The bitvecor has the length of the pattern and contains a 1 at every position the corresponding character is in the pattern.

`m`

length of the pattern.

Note:

[0]: Sun Wu and Udi Manber. 1992. Fast text searching: allowing errors. Commun. ACM 35, 10 (October 1992), 83-91.

`__init__`(*pattern*)

Initialize the Matcher Object for a given pattern.

Constructs the default dictionary `self.s` and inserts the bitvectors for every character in pattern.

Parameters **pattern** – pattern, that should be searched in the text.

`match_approx`(*text*, *k*)

Search the corresponding pattern in text with max k errors.

Only the match with the lowest error rate for one position of the text is reported. Returns a list with the ending positions in text of the matches.

Parameters

- **text** – the text to be searched.
- **k** – number of allowed errors

`match_exact`(*text*)

Search the corresponding pattern in text.

Returns a list with the ending positions in text of the matches.

Parameters **text** – the text to be searched.

PYTHON MODULE INDEX

INDEX