Named-Entity Recognition Project Documentation

Natural Language Processing and the Web

Florian Schneider, Julian Betz

December 5, 2017

1 Newly implemented classes

The following classes were newly implemented.

1.1 `NERWriter`

An Analysis Engine that processes a CAS to generate the evaluation file and to calculate statistics for the `NEIOBAnnotations` generated earlier in the pipeline. It finds all `NEIOBAnnotations` that have an attached prediction value and searches for the `NEIOBAnnotation` that has the corresponding gold standard value. Each gold standard / prediction pair, along with the corresponding token, is printed to a text file that can be used as input for the evaluation scripts.

For each gold standard value, the number of predictions of each named-entity type is shown in the output. As an aggregate result, the absolute and relative amounts of correct classification are given. Furthermore, a table is generated that shows the number of classifications of a token as a named entity or non-named entity. Finally, the absolute and relative amounts of correct classification of tokens that are named entities according to the gold standard are given. (This number may be of interest as there are far more non-named entities than named entities in the data, according to the gold standard.)

1.1.1 Configuration parameters

The configuration parameter `PARAM_FILENAME` is the filename of the evaluation file to be generated.

The configuration parameter `PARAM_NULL_TYPE` determines which string is used for marking non-named entities in the input. (Set to “O” in our case.)

If the configuration parameter `PARAM_VERBOSE` is set to `true`, all incorrect predictions are printed out to the log before printing the statistics.

The configuration parameter `PARAM_EXPECTED_ENTITY_TYPE_NUM` is used for the initialization of data structures and only affects efficiency, but not functionality.

1.2 `NEListExtractor<Token>`

This class provides a functionallity to create a feature if the covered text of a token appears in a gazetteer. As shown in figure 2, the `NEListExtractor` class implements the `FeatureFunction` interface. This is because this feature extractor works on the covered text of a token as mentioned before. Therefore, to use the `NEListExtractor` one has to do use it with the `CoveredTextExtractor` from the ClearTK Framework. An example can be seen in figure 1.

```
private static FeatureFunctionExtractor createCityListExtractors() throws IOException {
    return new FeatureFunctionExtractor<>(
                    new CoveredTextExtractor<Token>(),
                    FeatureFunctionExtractor.BaseFeatures.EXCLUDE,
                    new NEListExtractor( neListName: "src/main/resources/ner/germanCityNames.txt", featureName: "gerCity_LOC"),
                    new NEListExtractor( neListName: "src/main/resources/ner/englishCityNames.txt", featureName: "engCity_LOC"));
}
```

Figure 1: Example on how to use NEListExtractors

Please note, that in the example two `NEListExtractor`s are instantiated with two different gazetteers and feature names. The constructor of the `NEListAnnotator` requires two Strings, which represent the name (or path) to the list of Named Entities and the value of the feature that'll be created, respectivley (see 2). The provided list of Named Entities that will be used by the `NEListExtractor` has to contain one single column. Each row contains one word, that represent the Named Entity. Please note, that it is not possible to use e.g. 'New York' as a single Named Entity since it contains a space. This restriction is due to the `CoveredTextExtractor` that only 'looks' at one Token at a time and the Segmenter which is used in the pipeline creates two Tokens in this case - namlely 'New' and 'York'. Theoretically this can be changed by using a better Segmenter or using a different `TokenFeatureExtractor` than the `CoveredTextExtractor`, that takes multiple Tokens at a time in account.

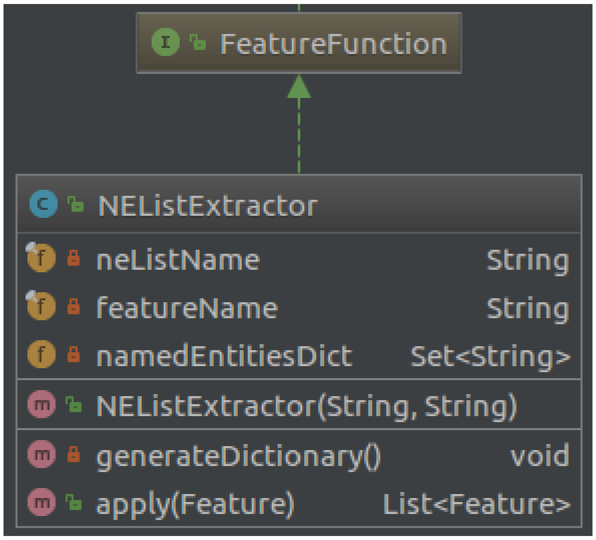

Figure 2: Class diagram of the NEListExtractor

The functionallity of the `NEListExtractor` is implemented in the overridden `apply()` method. A graphical representation of this method can be seen in 3. Whenn the `apply()` method gets called (indirectly from the `NERAnnotator`) at first the list of Named Entities gets generated in the `generateDicionary()` method. This is simply done by reading the list file line by line and storing the Named Enity in a Hash Set for fast look up. Since this has to be done only once and not everytime the apply() method gets called, there is a check if the dict has already been initialized. Then a simple look up of the Tokens covered text in the dictionary is done. If it appears in the dictionary, a Feature for the Token that holds the neListName as Features name and featureValue as the Features value member variables gets created. Since the interface of the `apply()` method requires a list of Features as return type, the Feature gets added to a singleton list (i.e. an immuatable list that only contains one item). If the list of named entities does not contain the Tokens covered text an empty list will be returned.

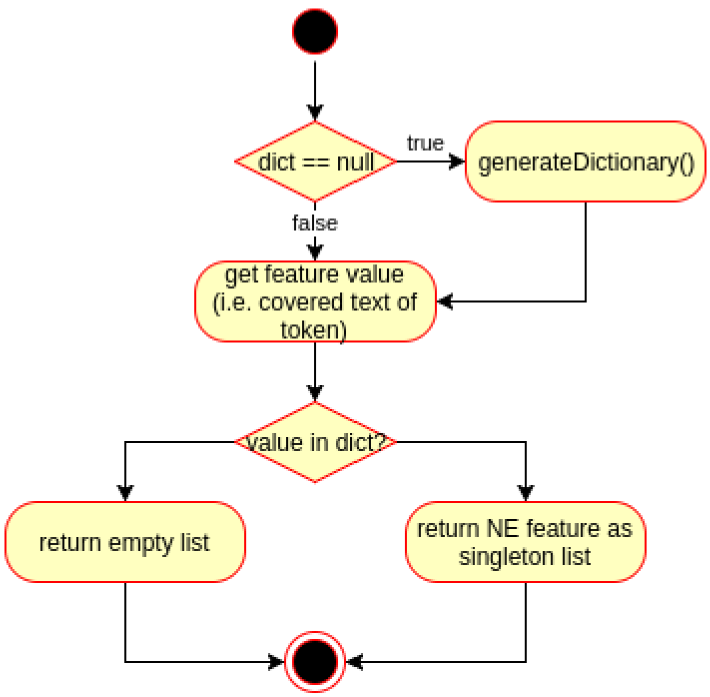

Figure 3: Activity diagram of the apply() method of NEListExtractor

1.3 `FeatureExtractorFactory`

This class only serves as untility class to instantiate the different Feature Extractors that will be used during NER and helps to reduce code redundancy. The class diagram is shown in figure 4. The names of the methods represent exactly what the method does - no magic at all. A more detailed description of the different Features Extractors have a look at section **??**

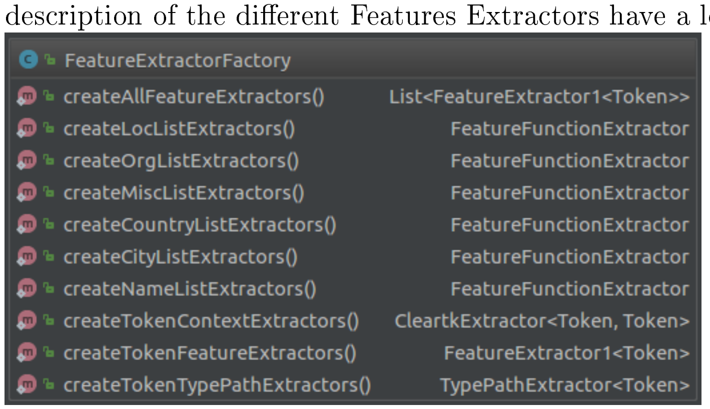

Figure 4: Class diagram of the FeatureExtractorFactory

1.4 `AblationTestRunner`

This class implements the `Runnable` interface and holds the algorithm of one single 'ablation test run'. The methods and algorithm in general were already provided by the boilerplate code available in the moodle course and contains only three steps shown in figure 5.

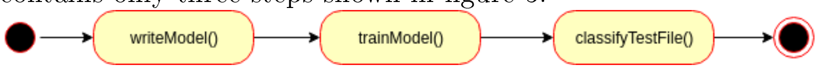

Figure 5: Activity diagram of the high level algorithm of the `AblationTestRunner`

The only adaption, that was made to the methods is, that each instance of an `AblationTestRunner` gets initialized with the configuration file, training file and test file, that will be used within the methods, whereas the boilerplate code used hardcoded file names. The constructor of the class also requires an Integer that represents the ID of one instance. This ID is used to write the generated models to different locations. Since the `AblationTestRunner`s are running in parallel, this step is neccessary because if we would do otherwise, the runners would always read/write to/from the same model. This, of course, leads to fatal errors since the Features used in the models are different in different models.

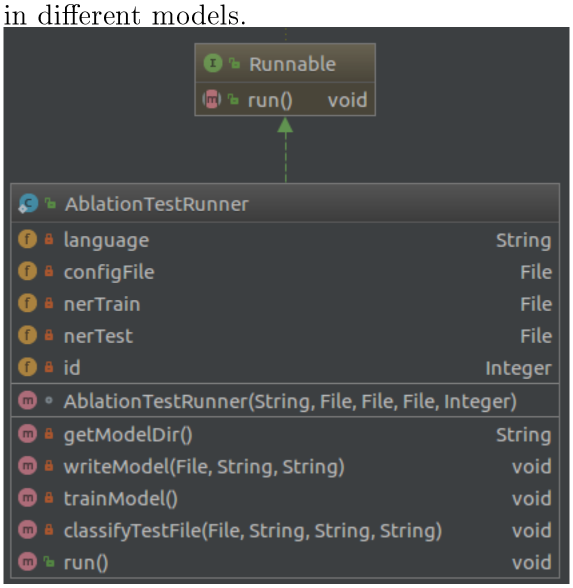

Figure 6: Class diagram of the AblationTestRunner

1.5 `ExecuteFeatureAblationTest`

This class holds the algorithm to do the Feature Ablation. It basically just initializes the variables and configuration parameters and then instantiates the `AblationTestRunner`s and hands them over to the managed thread pool. For a more detailed description have a look at section 4.1

2 Adapted classes

The following classes were adapted to suit the project.

2.1 `ner.ExecuteNER`

The Analysis Engine `NERWriter` was added to the end of the pipeline in `classifyTestFile`.

2.2 Features2Xml

This class was only slightly modified by refactoring some methods to reduces code redundancy. The methods `generateFeatureAblationTestFiles()` holds the functionallity to generate the XML configuration files for all combinations of Feature Extractors that will be tested during the Feature Ablation. It expects an Integer representing the number of minimum Feature Extractors that will be used and a String holding the output directory for the generated files. Those files are named by the Feature Extractors that get instantiated when using the file. For examlple the filename will include 'contextFeature' when the `ContextFeatureExtractor` is used in this configuration. See section 4 for a more detailed describtion.

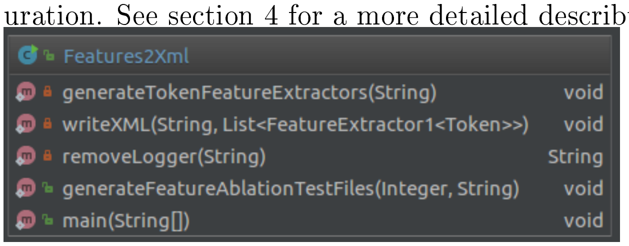

Figure 7: Class diagram of the Features2Xml class

3 Used Feature Extractors

In this section the Feature Extractors that are used during the NER are described. Most of the time we used FeatureExtractors of the UIMA or

ClearTK Framework and will therefor not describe them in this document. For a detailed description please visit the API Documentaion of the ClearTK Framework which can be found via this URL: `https://cleartk.github.io/cleartk/apidocs/2.0.0/`

3.1 Description

StemExtractor - Type: `TypePathExtractor<Token>`

This `TypePathExtractor` will create a `Feature` from the stem value of the Token

TokenFeatureExtractor - Type: `FeatureFunctionExtractor<Token>`

This `FeatureFunctionExtractor` uses a `CoveredTextExtractor` as base `FeatureExtractor` and creates `Features` with the following `FeatureFunctions`: `LowerCaseFeatureFunction`, `CapitalTypeFeatureFunction`, `NumericTypeFeatureFunction`, `CharacterNgramFeatureFunction` to create a bigram suffix, `CharacterNgramFeatureFunction` to create a trigram suffix, `CharacterCategoryPatternFunction`

TokenContextExtractor - Type: `CleartkExtractor<Token, Token>`

This FeatureExtractor creates a Feature from the context of the Token it analyses. As base `FeatureExtractor` a `CoveredTextExtractor` is used and the context is set to the two preceeding and following Tokens.

NameListExtractor - Type: `FeatureFunctionExtractor<Token>`

This `FeatureFunctionExtractor` uses a `CoveredTextExtractor` as base `FeatureExtractor` and two `NEListExtractors` (see section 1.2) to create a `Feature`. The gazetteers that are used are descriped in section 3.2.

CityListExtractor - Type: `FeatureFunctionExtractor<Token>`

This `FeatureFunctionExtractor` uses a `CoveredTextExtractor` as base `FeatureExtractor` and two `NEListExtractors` (see section 1.2) to create a `Feature`. The gazetteers that are used are descriped in section 3.2.

CountryListExtractor - Type: `FeatureFunctionExtractor<Token>`

This `FeatureFunctionExtractor` uses a `CoveredTextExtractor` as base `FeatureExtractor` and two `NEListExtractors` (see section 1.2) to create a `Feature`. The gazetteers that are used are descriped in section 3.2.

MiscListExtractor - Type: `FeatureFunctionExtractor<Token>`

This `FeatureFunctionExtractor` uses a `CoveredTextExtractor` as base `FeatureExtractor` and two `NEListExtractors` (see section 1.2) to create a `Feature`. The gazetteers that are used are descriped in section 3.2.

LocListExtractor - Type: `FeatureFunctionExtractor<Token>`

This `FeatureFunctionExtractor` uses a `CoveredTextExtractor` as base `FeatureExtractor` and two `NEListExtractors` (see section 1.2) to create a `Feature`. The gazetteers that are used are descriped in section 3.2.

OrgListExtractor - Type: `FeatureFunctionExtractor<Token>`

This `FeatureFunctionExtractor` uses a `CoveredTextExtractor` as base `FeatureExtractor` and two `NEListExtractors` (see section 1.2) to create a `Feature`. The gazetteers that are used are descriped in section 3.2.

3.2 Gazeteers for the different `NEListExtractors`

Country and City Lists

Source: `https://dev.maxmind.com/geoip/geoip2/geolite2/`
This dataset contains a list of countries and cities in multiple languages a long some information which is not needed in out application. We filtered the neccessary information, which contains a list of countries in german and english language and stored it in a simple text file.

Name Lists

Source: `http://www.quietaffiliate.com/free-first-name-and-last-name-databases-csv-and-sql/`
This dataset is made up from 5494 first names and 88799 last names. We also formatted the data to meet the interface contraints from the `NEListExtractors`.

Loc(ation), Misc(ellanous) and Org(anisation) List

Source: provided via Moodle Course
Here we just splitted the list, that is available in the Moodle Course into three lists containing location, miscellaneous and organisation names. Since the interface of the `NEListExtractor` requires the lists to be in only one column, we removed the first column of the original list. Note that we dropped the entries in the original list that hold personal names since we use the dataset described above.

4 Feature Ablation

In this section the process of Feature Ablation that is done in this project gets described. The classes that are used for this are described in the section 1. The goal of this process is to find the combination of Feature Extractors that yields the best NER results.

4.1 General Approach

To see the impact of the different Feature Extractors that are used during the NER, we evaluate the results of the NER when using different combinations of Feature extractors. Since only commenting out the extractors we want not to use in one single Ablation test and run the programm manually again and again, is very boring and takes a lot of time, we thought of an highly automated process which tests a lot of combinations of Feature Extractors. Theoretically we could test every single possible combination, which in our case would lead to $\sum_{k=1}^{n} \binom{n}{k} = 511$ different combinations, where $n = |M| = 9$ and $M = \{FeatureExtractor_1, .., FeatureExtractor_n\}$, denoting the set of Feature Extractors we use in the NER. We don't do this since firstly, it's not very useful to test all combinations since it is obvious that the results will be worse if only one or two Extractors are used and secondly, it would require too much computing power (i.e. time). One argument why one could do this anyways is, that one can get more detailed information about the impact of a single Feature Extractor on the result of the NER.
We think a good tradeoff would be if we test all possible combinations of Feature Extractors when using at least seven out of the nine Extractors we use in the NER. When doing this, the number of possible combinations gets reduced to $\sum_{k=7}^{9} \binom{9}{k} = 46$. Because this is still requires a lot of time to compute, we designed the algorithm to run the tests for the different combinations cuncurrently (see section 1.4 and 1.5 for more detailed information

about the algorithm).
The algorithm is explained graphically in figure 8

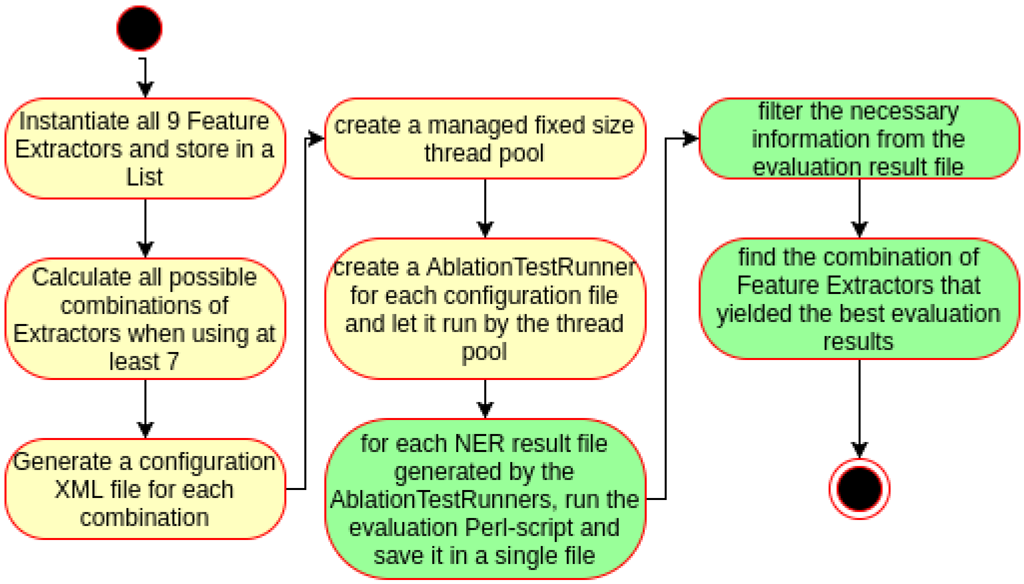

Figure 8: (High level) Algorithm to find the best combination of Feature Extractors for NER

4.2 Results

The results of the tests for different combinations on the development dataset can be seen in Figure 9, sorted by accuracy. As can be seen, the best results can be obtained when omitting the OrgListExtractor and MiscListExtractor, as opposed to using all Feature extractors. This setting also has a high precision on the data.
The results strongly suggest that the TokenFeatureExtractor is most vital for correctly predicting Named Entities; the ContextFeatureExtractor is the second most important Feature extractor.

Accuracy	Precision	Stem	TokenFeature	ContextFeature	NameList	CityList	CountryList	MiscList	OrgList	LocList
95.78%	80.60%	O	O	O	O	O	O			O
95.75%	80.38%	O	O	O	O			O	O	O
95.75%	80.38%	O	O	O	O		O	O	O	O
95.75%	79.92%	O	O	O	O		O		O	O
95.74%	80.29%	O	O	O	O	O		O	O	O
95.74%	80.29%	O	O	O	O	O	O	O	O	O
95.72%	79.88%	O	O	O	O	O			O	O
95.72%	79.88%	O	O	O	O	O	O		O	O
95.69%	80.45%	O	O	O	O	O	O		O	
95.69%	79.68%	O	O	O	O	O		O		O
95.69%	79.68%	O	O	O	O	O	O	O		O
95.67%	80.91%	O	O	O	O	O		O	O	
95.67%	80.91%	O	O	O	O	O	O	O	O	
95.67%	79.65%	O	O	O	O		O	O		O
95.64%	79.49%	O	O	O		O	O		O	O
95.63%	79.66%		O	O	O	O		O	O	O
95.63%	79.66%		O	O	O	O	O	O	O	O
95.63%	79.58%		O	O	O		O	O	O	O
95.62%	80.41%	O	O	O	O	O	O	O		
95.60%	79.55%	O	O	O		O		O	O	O
95.60%	79.55%	O	O	O		O	O	O	O	O
95.60%	79.05%	O	O	O		O	O	O		O
95.58%	79.45%	O	O	O			O	O	O	O
95.57%	78.91%		O	O	O	O	O		O	O
95.54%	80.51%	O	O	O		O	O	O	O	
95.53%	79.00%		O	O	O	O	O	O		O
95.51%	78.95%		O	O		O	O	O	O	O
95.49%	79.98%	O	O	O	O		O	O	O	
95.44%	79.48%		O	O	O	O	O	O	O	
95.32%	77.61%	O	O		O	O		O	O	O
95.32%	77.61%	O	O		O	O	O	O	O	O
95.30%	77.46%	O	O		O		O	O	O	O
95.26%	76.83%	O	O		O	O	O		O	O
95.19%	76.76%	O	O			O	O	O	O	O
95.06%	76.56%	O	O		O	O	O	O	O	
95.00%	75.94%		O		O	O	O	O	O	O
94.82%	76.01%	O	O		O	O	O	O		O
90.17%	64.59%			O	O	O	O	O	O	O
84.56%	39.77%	O		O	O		O	O	O	O
84.28%	38.60%	O		O		O	O	O	O	O
84.16%	38.19%	O		O	O	O		O	O	O
84.16%	38.19%	O		O	O	O	O	O	O	O
83.75%	38.39%	O		O	O	O	O	O	O	
81.64%	33.72%	O		O	O	O	O		O	O
78.13%	24.74%	O		O	O	O	O	O		O
74.53%	17.27%	O			O	O	O	O	O	O

Figure 9: Feature ablation results.

5 Final Evaluation and Results

To get the results, we instantiated the best combination of FeatureExtractors determined by the Feature Ablation test described in section 4. We then concatenated the 'old' training and test files provided in the Moodel Course (because it is allowed to do so) and used the concatenated file as new training file. The new test file is the file provided for the final evaluation. When running the Perl script for the evaluation on the generated output file the results are as shown in figure 10.

```
└─➤ ./conlleval_ner.pl < finalConfig.xml_evalOutput.txt
processed 51578 tokens with 5917 phrases; found: 5349 phrases; correct: 4271.
accuracy:  95.10%; precision:  79.85%; recall:  72.18%; FB1:  75.82
              LOC: precision:  85.82%; recall:  85.63%; FB1:  85.72  1826
             MISC: precision:  85.27%; recall:  54.49%; FB1:  66.49  584
              ORG: precision:  62.26%; recall:  53.39%; FB1:  57.49  1150
              PER: precision:  83.29%; recall:  81.33%; FB1:  82.30  1789
```

Figure 10: Output of the evaluation script on the final test file.