Named-Entity Recognition Project Documentation
Natural Language Processing and the Web

Florian Schneider, Julian Betz   December 4, 2017

1   Newly implemented classes

The following classes were newly implemented in `de.unihamburg.informatik.nlp4web.tutorial.tut5`.

1.1   `writer.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

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.
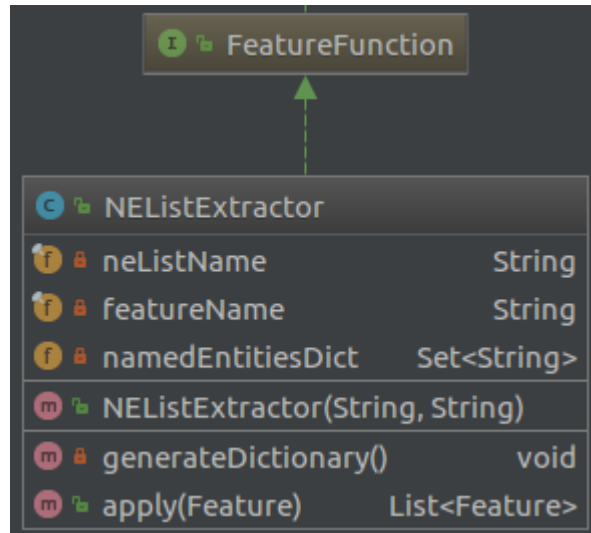
Figure 2: Class diagram of the NEListExtractor

The functionallity of the `NEListExtractor` is implemented in the overrid-
den `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.

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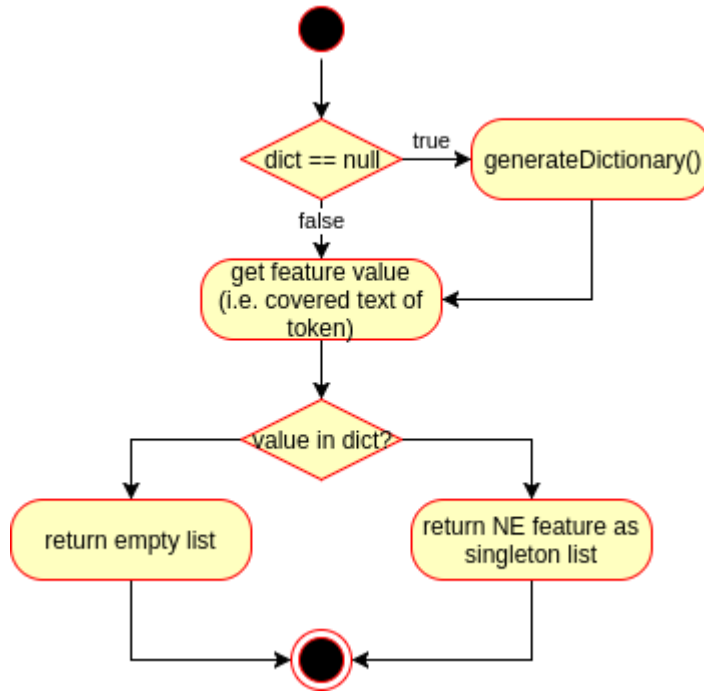
Figure 3: Activity diagram of the apply() method of NEListExtractor

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 4The names of the methods represent exactly what the method does - no magic at all.
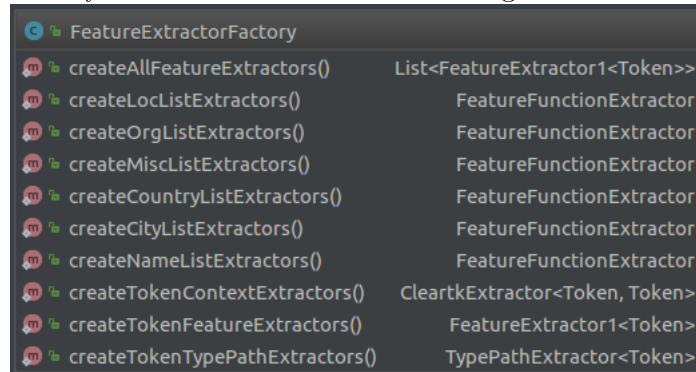
Figure 4: Class diagram of the FeatureExtractorFactory

1.4 `AblationTestRunner`

This class implements the Runnable interface and holds the algorithm of one single run 'ablation test run'.
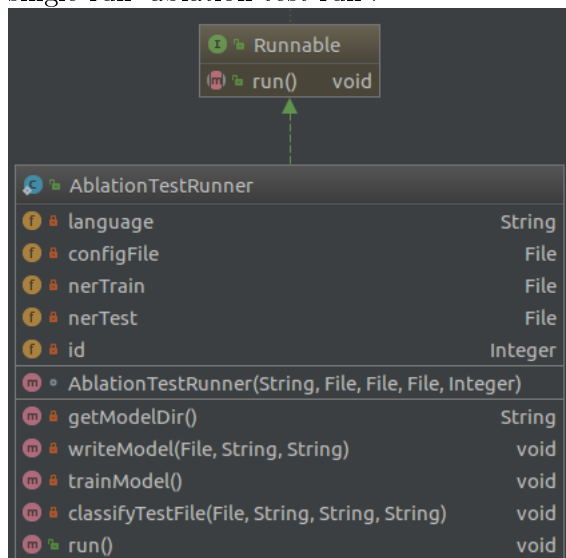
Figure 5: Class diagram of the AblationTestRunner

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 3.1

2 Adapted classes

The following classes were adapted in `de.unihamburg.informatik.nlp4web.tutorial.tut5`.

2.1 `ner.ExecuteNER`

The Analysis Engine `writer.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 3 for a more detailed describtion.

Figure 6: Class diagram of the Features2Xml class

3 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 **??**. The goal of this process is to find the combination of Feature Extractors that yields the best NER results.

3.1 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 7
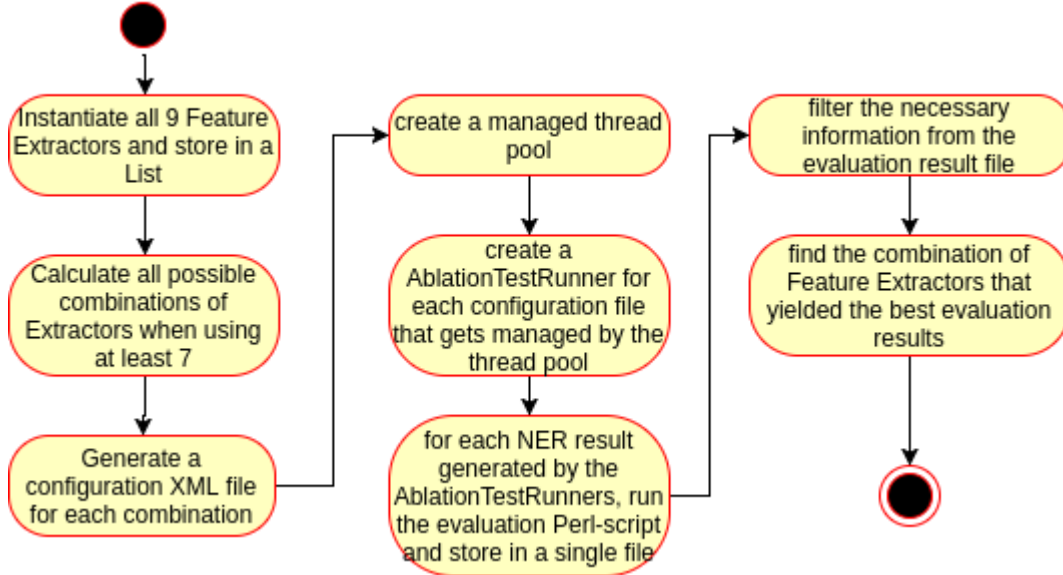
Figure 7: (High level) Algorithm to find the best combination of Feature Extractors for NER

3.2   Results

For the implemented Feature extractors we

Figure 8: Feature ablation results.

Accuracy	Precision
95.78%	80.60%
95.75%	80.38%
95.75%	80.38%
95.75%	79.92%
95.74%	80.29%
95.72%	79.88%
95.72%	79.88%
95.69%	80.45%
95.69%	79.68%
95.69%	79.68%
95.67%	80.91%
95.67%	80.91%
95.65%	79.65%
95.64%	79.49%
95.63%	79.66%
95.63%	79.66%
95.63%	79.58%
95.62%	80.41%
95.62%	79.55%
95.60%	79.05%
95.60%	79.55%
95.58%	79.45%
95.57%	78.91%
95.54%	80.51%
95.53%	79.00%
95.51%	78.95%
95.49%	79.98%
95.44%	79.48%
95.32%	77.61%
95.32%	77.61%
95.30%	77.46%
95.26%	76.83%
95.19%	76.76%
95.06%	76.56%
95.00%	75.94%
94.82%	76.01%
90.17%	64.59%
84.56%	39.77%
84.28%	38.60%
84.16%	38.19%
84.16%	38.19%
83.75%	38.39%
81.64%	33.72%
78.13%	24.74%
74.53%	17.27%