Advanced Human Languages Technologies
NERC Report

Denaldo Lapi and Francesco Aristei

November 5, 2022

1 Introduction

This report contains some possible solutions of the task 9.1 of the *SemEval-2013 challenge* (link to the paper), concerning the named entity recognition and classification of drugs (NERC).

We were provided with a dataset consisting of XML files containing sentences and entities representing various types of classes. There are four general classes: drug, brand, group and drug_n. The data is already splitted in three subsets (folders): Train, Devel and Test. The goal of the task is to develop a model to recognize and classify between the four classes described in the XML documents.

We were also provided with some external resources containing knowledge extracted from other databases (*DrugBank*, *HSDB*) and with evaluation scripts to compute metrics like precision, recall and F1 score.

In order to solve the task, we focused on 2 main approaches:

- Rule-based (that was used as a baseline)
- Machine Learning based

The aim of this report is to explain how we used these two approaches to solve the NERC task, describing the main aspects, rules and decisions taken in order to otimize the performance of each method.

2 Rule-based baseline

At the beginning, we developed a simple rule-based system in order to have a baseline for our NERC task, i.e. a lower bound for the ML system that we built in the second part of our work.

The idea of this approach is to build a simple list of rules, chaining them as a cascade of *if-then-else* statements, and to decide the class for each token (or multiple tokens as we will see later on in the report) based on the results of the evaluated conditions.

2.1 Ruleset construction

In order to extract some meaningful rules able to distinguish among the different classes, we performed some kind of preliminary data exploration, to extract with little effort some general

and simple patterns able to describe the different classes.
We performed this activity on the *train* folder of the provided dataset, while we checked the performance of the various rules on the *devel* folder; this last step is needed, in order to choose the most meaningful rules for this approach and their right order inside the *if-then-else* cascade of rules.
We were provided with an initial version of the code which chieves 36% macro average F1 on the devel set with 3 simple rules.

We will now make a brief digression on the main characteristics we observed for each class in our *train* folder, that we exploited to extend and modify this initial rules.

- We first saw that the already present rule which classifies text in upper case in the "brand" class generated a lot of false positives in the "brand" category, so we tried to remove it and we obtained a quite good improvement in the F1 score that reached a value of 43.9% on the devel set (all the following results were evaluated on the devel dataset).
- We then noticed, by visually inspecting the XML files, that a lot of drugs in the "group" class were characterized by having a quite high number of characters and most of them ends with the character 's'. We therefore added a rule that matches tokens ending with 's' and with a length greater than a certain value that, after some trials, we decided to be 12. This rule allowed us to increase the F1 score up to 45.2%
- We noticed that a lot of elements of the class "drug" contain the 'x' character and have a length greater than 8: we tried this rule in every position inside the cascade of *if-then-else* statements, but we obtained no improvement.
- In order to improve the performance of the algorithm on the "drug_n" class, which has a very low F1 score (very low precision w.r.t. the recall) we tried several rules. We noticed that most of the elements of this class appear with a quite short length and with upper-case characters. We therefore tried this rule, which didn't work.
- By a simple visual inspection of the elements of the class "drug_n" we noticed that some of them contain digits, have a length greater than 5 (this value was finetuned) and also contain a dash character inside. By adding a rule derived from this observation we improved the F1 score that reached the value of 47.3%
- By removing the rule (already present) for the drug suffixes, we obtained an improvement of 0.1% in the F1 score, indeed the number of false positives was slightly reduced.
- We then visually analyzed the suffixes of the class "group" and, after some trial and error, we built the following python tuple with the most common suffixes for the strings belonging to that class: ('tics', 'lant', 'ants', 'ones', 'ists', 'alis', 'iral', 'ioid'). We therefore added a rule for the class "group", checking whether the text of a token (in lower case) has one of this suffixes. This rule allowed us to improve the F1 score of 0.5%
- We did the same analysis as before for the class "drug_n" and we built another python tuple containing the most common suffixes we found for that class: ('hydro', 'methyl'). This addition increased the F1 score to 49.1%

Until now, the rules we built were just able to classify each token in a sentence separately, without taking into account multi-token entities, i.e. drug names composed my multiple consecutive tokens.

During the phases of data exploration we observed that the type "group" contained several drug names composed by two words, that's why we modified the function *extract_entities* in order to check whether two consequent tokens are classified as drug by the *classify_token* function and, in case of two consecutive "group" tokens, we put them together as a unique drug of the type "group".

This allowed us to improve the F1 up to 50.1%.

At the very end, in order to further improve the performance of the "drug_n" class, we added to the list of its suffixes the word 'Nep', and this addition allowe us to further boost the performance up to 54.6% of F1 score.

The final results on the test and devel datasets will be reported in the following sections.

2.2 Code

We include below the code of the function *extract_entities* and the function *classify_token*. The explanation of the functions is provided in the comments reported inside the code snippets.

Below the code of the *extract_entities function*:

```
## --------- Entity extractor -----------
## -- Extract drug entities from given text and return them as
## -- a list of dictionaries with keys "offset", "text", and "type"

def extract_entities(stext) :

    # tokenize text
    tokens = tokenize(stext)
    result = []

    # previously classified drug: we need it to manage multi-token drug names
    previous_drug = ""

    # classify each token and decide whether it is an entity.
    for (token_txt, token_start, token_end)  in tokens:
        # classify current token
        drug_type = classify_token(token_txt)

        # create and add the entity to the result list
        if drug_type in ['drug', 'drug_n', 'brand']:

            # save current recognized drug to check it at the next iteration
            previous_drug = str(token_txt)+" "+str(token_start)+" "+str(drug_type)
            # create entity dictionary
            e = { "offset" : str(token_start)+"-"+str(token_end),
                  "text" : stext[token_start:token_end+1],
                  "type" : drug_type
                }
            result.append(e) # append entity to the result list

        # case of 'group' current drug type: check previously recognized drug
        elif drug_type == 'group':
            if previous_drug != "": # check if the previous token is a drug of type
            ↪  'group'
```

```
                prev_drug_info = previous_drug.split(" ")
                if len(result) > 0 and prev_drug_info[2] == "group":
                    result.pop() # multi-token drug of type 'group' recognized: pop the previously inserted entity to add the new multi-token one
                    e = { "offset" : str(prev_drug_info[1])+"-"+str(token_end), # initial offset is the one of the previous recognized drug
                          "text" : stext[int(prev_drug_info[1]):token_end+1], # text composed of multiple tokens
                          "type" : 'group'
                        }

            # previous token not a recognized drug: just add the current entity as type 'group'
            else:
                e = { "offset" : str(token_start)+"-"+str(token_end),
                  "text" : stext[token_start:token_end+1],
                  "type" : 'group'
                 }
                result.append(e)
            previous_drug = str(token_txt)+" "+str(token_start)+" "+str(drug_type)

        # current token is not a drug
        else:
            previous_drug = ""

    return result
```

The function *extgract_entities* takes as input a whole sentence, tokenizes it (i.e. transforms it into a sequence of tokens) and iterates through the tokens classifying each of them into one of the four classes of drugs. It includes also the management of the multi-token drugs, as explained before.

Below the code of the *classify_token* function:

```
## -- check if a token is a drug part, and of which type

# suffixes for group
group_suffixes = ('tics', 'lant', 'ants', 'ones', 'ists', 'alis', 'iral','ioid')

# suffixes for drug_n
drug_n_suff = ('hydro', 'methyl', 'Nep')

def classify_token(txt):

    # check presence in external knowledge resources
    if txt.lower() in external : return external[txt.lower()]

    # group rules
    elif txt.lower().endswith(group_suffixes): return "group"
    elif len(txt) > 12 and txt[-1:] == 's': return "group"

    # drug_n rules
```

```python
    elif any(char.isdigit() for char in txt) and len(txt) > 5 and '-' in txt:
    ↪  return "drug_n"
    elif any(suff in txt for suff in drug_n_suff): return "drug_n"
    elif '[' in txt and len(txt) > 2: return 'drug_n'

    else : return "NONE"
```

The function is simple the cascade of *if-then-else* rules we described before.

2.3 Experiments and results

We report in figure 1 the statistics regarding the performance obtained on the devel and on the test datasets for what regards our best performing rule-based baseline obtained at the end of the rules building process.

	tp	fp	fn	#pred	#exp	P	R	F1
brand	318	251	56	569	374	55.9%	85.0%	67.4%
drug	1588	305	318	1893	1906	83.9%	83.3%	83.6%
drug_n	20	58	25	78	45	25.6%	44.4%	32.5%
group	232	415	455	647	687	35.9%	33.8%	34.8%
M.avg	-	-	-	-	-	50.3%	61.6%	54.6%
m.avg	2158	1029	854	3187	3012	67.7%	71.6%	69.6%
m.avg(no class)	2310	877	702	3187	3012	72.5%	76.7%	74.5%

	tp	fp	fn	#pred	#exp	P	R	F1
brand	251	284	23	535	274	46.9%	91.6%	62.1%
drug	1630	295	497	1925	2127	84.7%	76.6%	80.5%
drug_n	3	70	69	73	72	4.1%	4.2%	4.1%
group	266	499	427	765	693	34.8%	38.4%	36.5%
M.avg	-	-	-	-	-	42.6%	52.7%	45.8%
m.avg	2150	1148	1016	3298	3166	65.2%	67.9%	66.5%
m.avg(no class)	2327	971	839	3298	3166	70.6%	73.5%	72.0%

Figure 1: Best rule-based model performance on the Devel set (on the left) and on the Test set (on the right

We would like to briefly comment the difference between the performance of the "drug_n" class between devel and test tests. What happens is that most probably our rules for what regards that class overfit too much the devel dataset: we basically capture too much the characteristics and the noise of that class on dhe devel XML files. This results into a worsening of the performance on the test set.

3 Machine Learning NERC

After trying the rule-based system, in the second part of the task we tried to solve the same problem (NERC) using machine learning techniques. Our main objective on this part was to further improve the performance of the rule-based system by overcoming the limitations characterizing it.
The machine learning approach to solve the NERC task consists on the following main steps:

- Extract and define different features for each token to be used to classify the tokens into the 4 different drug types
- Train a ML model with the obtained feature vectors
- Perform feature engineering and hyperparameter selection by using the Devel dataset
- Evaluate the final obtained model on the Test dataset

3.1 Selected algorithm

At the beginning we focused our attention on the feature engineering part by testing the "goodness" of the various feature combinations on the Devel dataset and training two types of ML models: a CRF model and a MEM model.
After doing several trials we realized that the CRF model always obtained better performances on the Devel dataset, so we decided to mainly focus our attention on tuning its hyperparameters. In order to perform hyperparameter tuning on the CRF algorithm we decided to use a *Grid Search* approach. To do so, we analyzed the parameters of the CRF algorithm, provided by the *CRFsuite* library used in the codebase. Specifically, the parameters that the model allows to modify and tune are the following ones:

- *feature.minfreq*: The minimum frequency of features.
- *feature.possible_transitions*: Force to generate possible transition features.
- *feature.possible_states*: Force to generate possible state features.
- *c2*: Coefficient for L2 regularization.
- *max_iterations*: The maximum number of iterations (epochs) for SGD optimization.
- *period*: The duration of iterations to test the stopping criterion.
- *delta*: The threshold for the stopping criterion; an optimization process stops when the improvement of the log likelihood over the last period iterations is no greater than this threshold.
- *calibration.eta*: The initial value of learning rate (eta) used for calibration.
- *calibration.rate*: The rate of increase/decrease of learning rate for calibration.
- *calibration.samples*: The number of instances used for calibration.
- *calibration.candidates*: The number of candidates of learning rate.
- *calibration.max_trials*: The maximum number of trials of learning rates for calibration.

Considering the huge number of possible models we could generate combining the different parameters with different values, we decided to train the model tuning one parameter at a time, to understand its impact on the overall performance. For example, we trained the model tuning the *calibration.rate*, leaving the other parameters with the default values. What we observed is that the performance of the model (on the Devel set) never exceeded significantly the one trained with all the default values (here we used an intermediate version of the features, which is the one that allowed to have a F1 of 70.5% on the Devel set and that will be shown later). The results are shown in the table 1:

Models Comparison	
calibration.rate	**Model Performance (F1)**
0.1	**70.4%**
0.5	**70.5%**
1.0	**70.4%**
2.0 (D)	**70.5%**
10.0	**70.4%**
100	**70.4%**

Table 1: Performance of models with different *calibration.rate* parameter

Therefore, we decided to not include this parameter in the grid search. In some other cases, we saw an important decrease in the performance of the model when the parameter started to diverge from a certain value. For example in the case of *feature.minfreq*, as it is shown in the table 2:

Models comparison	
feature.minfreq	**Model performance (F1)**
1.0 (D)	**70.5%**
2.0	**71.1%**
6.0	**71.1%**
7.0	**70.5%**
8.0	**68.8%**
10	**67.1%**
20	**64.7%**
50	**61.8%**

Table 2: Performance of models with different *feature.minfreq* parameter

So, in this situation, we used only few values in the grid search.

Notice that the obtained performances exceeded significantly the ones of the default model (around 55% of F1 on devel) because we had already done some feature extraction (our default CRF model with these features had a 70.5% F1 score on the Devel set).

After this preliminary analysis, we concluded that the main parameters to focus on when performing the hyperparameter tuning were: *feature.minfreq*, *max_iterations*, *delta*, *c2*, *calibration.eta*.

After having completed the feature extraction process (which will allow us to achieve a F1 score of 75.7% on the Devel, as we will see in the next section), we implemented a list for each parameter, with the values that gave us the best performances, i.e. we implemented the grid search. Then, we combined the parameters together simply looping through each list in a nested way. After this, we trained the model, each time with a different combination of the parameters, and we saved it as a *.crf* file. In the end we generated 540 different models and finally we used the provided codebase to evalute their performances.

What we obtained is that the performances never exceeded the one of the CRF model without hyperparameter tuning. The best performing model obtained with the grid search achieve a performance of 74.6% F1 with the following parameters:

- *feature.minfreq*: 2
- *max_iterations*: 100

- *delta*: 1e-06
- *c2*: 0.1
- *calibration.eta*: 0.5

The complete statistics are the ones showed in figure 2.

devel-CRFmodelfreq:2c2:0.1delta:1e-06maxitr:100_eta:0.5.crf.out :

	tp	fp	fn	#pred	#exp	P	R	F1
brand	319	12	55	331	374	96.4%	85.3%	90.5%
drug	1737	101	169	1838	1906	94.5%	91.1%	92.8%
drug_n	9	5	36	14	45	64.3%	20.0%	30.5%
group	563	83	124	646	687	87.2%	82.0%	84.5%
M.avg	-	-	-	-	-	85.6%	69.6%	74.6%
m.avg	2628	201	384	2829	3012	92.9%	87.3%	90.0%
m.avg(no class)	2680	149	332	2829	3012	94.7%	89.0%	91.8%

Figure 2: Best performing CRF model obtained with grid search

The fact that we were not able to improve the performance of the model with default parameters (75.7% F1 on Devel) is due to the fact that we did feature engineering by optimizing the performance on the Devel set by training the CRF model having the default hyperparameters: in this way we tried to build the best features that comply with that default choice of the parameters. In order to have a better hyperparameter tuning we should perform another step of feature selection taking into account the selected parameters of the CRF algorithm.

3.2 Feature extraction

As briefly explained before, we first focused our attention on the feature engineering task: we tried to build several features able to encode the information contained in each token and its surroundigs and that could allow to boost the performance of the classifier. In this first phase we evaluated the goodness of the various features on both the algorithms, i.e CRF and MEM, mainly focusing on the F1 score obtained in the Devel dataset.

3.2.1 Tried features

We list here the various features we tried and the performances of both classifiers (CRF and MEM) trained over these features (on the Train dataset) evaluated on the Devel dataset (in terms of F1 score).
We tried the following feature, in order:

- Prefixes and suffixes (from 2 characters to 5) of the current token. We added them incrementally and at each addition we checked whether there was an improvement of the classifier score on the devel dataset. We obtained the best performance with prefixes and suffixes up to the fifth character. From these experiments we understood that only prefixes and suffixes up to 5 characters are relevant features for our task (adding the sixth characters reduces the performance).
  CRF: 67.4%; MEM: 58.9%
- Prefixes and suffixes for the previous token. Again, we used up to five characters.
  CRF: 67.5%; MEM: 64.9%

- Prefixes and suffixes for the following token. Same approach as before.
  CRF: 66.9%; MEM: 65.2%
  This feature improved only the MEM performance, that's why we discarded it.

- Uppercase word for the current token: no improvement

- Current token contains the dash character: no improvement

- Length of the current token, previous tokens and next tokens. We added these features incrementally to the current token, previous , following, 2nd previous and so on and we considered only the best option.
  CRF: 70.0%; MEM: 67.0%

- Current token contains any digit.
  CRF: 70.2%; MEM: 66.5%

- Then, we tried to add again the prefixes and suffixes for the subsequent token, which gave an improvement with the new features combination.
  CRF: 71.1%; MEM: 67.4%

- Current token composed of all digits: no improvement

- Prefixes and suffixes for the token 2 positions before of the current one:
  CRF: 71.7%; MEM: 68.3%

- Prefixes and suffixes for the token 2 position after the current one: no improvement

- Length of the 2nd previous and 2nd subsequent token (even if it didn't improve, we kept it because with the next features we added it gave some boost to the performance):
  CRF: 71.7%; MEM: 66.6%)

- Lower form of the text for the current token, the previous, the next, the 2nd previous and the 2nd subsequent.
  CRF: 71.8%; MEM: 68.2%

- Title form of the text for the current token, the previous, the next, the 2nd previous and the 2nd subsequent.
  CRF: 72.0%; MEM: 67.0%

- Camel-case form of the text for the current token, the previous, the next, the 2nd previous and the 2nd subsequent: no improvement

- Capitalized form of the text for the current token (it doesn't improve the performance if applied to the previous and next tokens):
  CRF: 72.1%; MEM: 68.3%

- Feature encoding the presence of the current, previous, next, 2nd precedent and 2nd subsequent tokens in the external dictionary containing the drugs of *HSBD* and *DrugBank*:
  CRF: 74.9%; MEM: 70.6%

- Feature checking whether the current token is a stopword (no improvements if applied also to the previous and subsequent tokens):
  CRF: 75.1%; MEM: 72.3%

- Current token contains only aplphabets(*is_alpha()*) for the current, previous and next tokens:
  CRF: 75.3%; MEM: 72.2%
- Token starts and ends with digit for the current, previous, next tokens: no improvement
- Current token contains both upper and lower case characters:
  CRF: 75.4%; MEM: 72.3%

  At the very end we tried to add the features that didn't work and we tried also several combinations of features. This process showed that by adding the features encoding the camel-case form, the presence of the dash and the presence of only digits for what regards only the current token, we further improved the F1 score of the CRF model up to 75.7%, while the F1 of the MEM model reached the value of 72.6%. The final results on devel and test of the best performing models will be shown in the following sections

3.3 Code

We include here the code of the *extract_features()* function which generates a list of features for each token, according to the features we described above.

```python
## --------- Feature extractor -----------
## -- Extract features for each token in given sentence

# generate camel-case version of a string
def camel_case(s):
  s = re.sub(r"(_|-)+", " ", s).title().replace(" ", "")
  if len(s) == 0: return s
  return ''.join([s[0].lower(), s[1:]])

# dictionary containing information from external knowledge resources
external = {}

# build the dictionary
path = "../TaskData/resources/HSDB.txt" # read the HSDB file
with open(path, encoding="utf8") as h :
    for x in h.readlines() :
        external[x.strip().lower()] = "drug" # add drug into the dictionary

path = "../TaskData/resources/DrugBank.txt" # read the DrugBank file
with open(path, encoding="utf8") as h :
    for x in h.readlines() :
        (n,t) = x.strip().lower().split("|")
        external[n] = t # add drug into the dicitonary with the corresponding type

# obtain the list of english stopword
stop_words = stopwords.words('english')

# build regular expression patterns for lower and upper chars
lower = re.compile(r'.*[a-z]+')
upper = re.compile(r'.*[A-Z]+')

# feature extractor function
```

```
def extract_features(tokens) :

   # for each token, generate list of features and add it to the result
   result = []
   for k in range(0,len(tokens)):
      tokenFeatures = []; # features of the current token
      t = tokens[k][0]

      # external-knowledge based feature
      if t.lower() in external:
         tokenFeatures.append("external="+external[t.lower()])

      tokenFeatures.append("form="+t)

      # lower_case form
      tokenFeatures.append("lower_form="+ t.lower())

      # title_form
      tokenFeatures.append("title_form="+t.title())

      # camel-case
      tokenFeatures.append("camelCase_form="+camel_case(t))

      # capitalize
      tokenFeatures.append("capitalized_form="+t.capitalize())

      #stopword
      if t.lower() in stop_words:
         tokenFeatures.append("is_stopword")

      # is_alpha
      if t.isalpha():
         tokenFeatures.append("is_alpha="+t)

      # suffixes and prefixes of the current token
      tokenFeatures.append("suf3="+t[-3:])
      tokenFeatures.append("pref3="+t[:3])
      tokenFeatures.append("suf2="+t[-2:])
      tokenFeatures.append("pref2="+t[:2])
      tokenFeatures.append("suf4="+t[-4:])
      tokenFeatures.append("pref4="+t[:4])
      tokenFeatures.append("suf5="+t[:-5])
      tokenFeatures.append("pref5="+t[:5])

      # upper
      if t.isupper():
         #tokenFeatures.append("isUpper="+t)
         tokenFeatures.append("isUpper=UP")

      # dash
      if '-' in t:
         tokenFeatures.append("contains_dash")
```

```
        # contains any digit (taken from the baseline rules)
        if any(char.isdigit() for char in t):
            tokenFeatures.append("text=withDigit")

        # is upper and lower current word
        if lower.match(t) and upper.match(t):
            tokenFeatures.append("lowerAndUpper")

        # contains only digits
        if all(char.isdigit() for char in t):
            tokenFeatures.append("text=onlyDigits")

        # length
        tokenFeatures.append("length=%s" %len(t) )


        # features for the previous token
        if k>0 :
            tPrev = tokens[k-1][0]

            # external knowledge based feature
            if tPrev.lower() in external:
                tokenFeatures.append("externalPrev="+external[tPrev.lower()])

            # form
            tokenFeatures.append("formPrev="+tPrev)

            #lower form
            tokenFeatures.append("lower_formPrev="+tPrev.lower())

            # title_form
            tokenFeatures.append("title_formPrev="+tPrev.title())

            # is_alpha
            if tPrev.isalpha():
                tokenFeatures.append("is_alphaPrev="+tPrev)

            #stopword
            if tPrev.lower() in stop_words:
                tokenFeatures.append("is_stopwordPrev")

            # pref and suf
            tokenFeatures.append("suf3Prev="+tPrev[-3:])
            tokenFeatures.append("pref3Prev="+tPrev[:3])
            tokenFeatures.append("suf2Prev="+tPrev[-2:])
            tokenFeatures.append("pref2Prev="+tPrev[:2])
            tokenFeatures.append("suf4Prev="+tPrev[-4:])
            tokenFeatures.append("pref4Prev="+tPrev[:4])
            tokenFeatures.append("suf5Prev="+tPrev[-5:])
            tokenFeatures.append("pref5Prev="+tPrev[:5])
```

```
            # length
            tokenFeatures.append("lengthPrev=%s" %len(tPrev) )

        else :
            tokenFeatures.append("BoS")


        # features for the token 2 positions before from the current one
        if k > 1:
            tPrev2 = tokens[k-2][0]

            # external knowledge based features
            if tPrev2.lower() in external:
                tokenFeatures.append("externalPrev2="+external[tPrev2.lower()])

            tokenFeatures.append("formPrev2="+tPrev2)

            #lower form
            tokenFeatures.append("lower_formPrev2="+tPrev2.lower())

            # title_form
            tokenFeatures.append("title_formPrev2="+tPrev2.title())

            # pref and suf
            tokenFeatures.append("suf3Prev2="+tPrev2[-3:])
            tokenFeatures.append("pref3Prev2="+tPrev2[:3])
            tokenFeatures.append("suf2Prev2="+tPrev2[-2:])
            tokenFeatures.append("pref2Prev2="+tPrev2[:2])
            tokenFeatures.append("suf4Prev2="+tPrev2[-4:])
            tokenFeatures.append("pref4Prev2="+tPrev2[:4])
            tokenFeatures.append("suf5Prev2="+tPrev2[-5:])
            tokenFeatures.append("pref5Prev2="+tPrev2[:5])

            #length
            tokenFeatures.append("lengthPrev2=%s" %len(tPrev2) )


        # # features for the token 1 position ahead from the current one
        if k<len(tokens)-1 :
            tNext = tokens[k+1][0]

            # external knowledge based feature
            if tNext.lower() in external:
                tokenFeatures.append("externalNext="+external[tNext.lower()])

            tokenFeatures.append("formNext="+tNext)

            #lower form
            tokenFeatures.append("lower_formNext="+tNext.lower())

            # title_form
            tokenFeatures.append("title_formNext="+tNext.title())
```

```

            #stopword
            if tNext.lower() in stop_words:
                tokenFeatures.append("is_stopwordNext")

            # is_alpha
            if tNext.isalpha():
                tokenFeatures.append("is_alphaNext="+tNext)

            # pref and suf
            tokenFeatures.append("suf3Next="+tNext[-3:])
            tokenFeatures.append("pref3Next="+tNext[:3])
            tokenFeatures.append("suf2Next="+tNext[-2:])
            tokenFeatures.append("pref2Next="+tNext[:2])
            tokenFeatures.append("suf4Next="+tNext[-4:])
            tokenFeatures.append("pref4Next="+tNext[:4])
            tokenFeatures.append("suf5Next="+tNext[-5:])
            tokenFeatures.append("pref5Next="+tNext[:5])

            # length of the following token
            tokenFeatures.append("lengthNext=%s" %len(tNext) )
        else:
            tokenFeatures.append("EoS")


        # features for the token 2 positions ahead from the current one
        if k<len(tokens) - 2:
            tNext2 = tokens[k+2][0]

            # external knowledge based features
            if tNext2.lower() in external:
                tokenFeatures.append("externalNext2="+external[tNext2.lower()])

            tokenFeatures.append("formNext2="+tNext2)

            #lower form
            tokenFeatures.append("lower_formNext2="+tNext2.lower())

            # title_form
            tokenFeatures.append("title_formNext2="+tNext2.title())

            #length
            tokenFeatures.append("lengthNext2=%s" %len(tNext2) )


        result.append(tokenFeatures)

    return result
```

3.4 Experiments and results

We report below the statistics regarding the performance obtained on the devel and on test datasets for what regards our best performing CRF model (figure 3) and also the MEM model(figure 4), obtained at the end of the feature extraction process.

	tp	fp	fn	#pred	#exp	P	R	F1
brand	318	11	56	329	374	96.7%	85.0%	90.5%
drug	1744	99	162	1843	1906	94.6%	91.5%	93.0%
drug_n	10	4	35	14	45	71.4%	22.2%	33.9%
group	567	77	120	644	687	88.0%	82.5%	85.2%
M.avg	-	-	-	-	-	87.7%	70.3%	75.7%
m.avg	2639	191	373	2830	3012	93.3%	87.6%	90.3%
m.avg(no class)	2688	142	324	2830	3012	95.0%	89.2%	92.0%

	tp	fp	fn	#pred	#exp	P	R	F1
brand	250	25	24	275	274	90.9%	91.2%	91.1%
drug	1849	91	278	1940	2127	95.3%	86.9%	90.9%
drug_n	2	12	70	14	72	14.3%	2.8%	4.7%
group	559	110	134	669	693	83.6%	80.7%	82.1%
M.avg	-	-	-	-	-	71.0%	65.4%	67.2%
m.avg	2660	238	506	2898	3166	91.8%	84.0%	87.7%
m.avg(no class)	2749	149	417	2898	3166	94.9%	86.8%	90.7%

Figure 3: Best CRF model performance on the Devel set (on the left) and on the Test set (on the right

	tp	fp	fn	#pred	#exp	P	R	F1
brand	315	12	59	327	374	96.3%	84.2%	89.9%
drug	1745	105	161	1850	1906	94.3%	91.6%	92.9%
drug_n	7	6	38	13	45	53.8%	15.6%	24.1%
group	555	87	132	642	687	86.4%	80.8%	83.5%
M.avg	-	-	-	-	-	82.7%	68.0%	72.6%
m.avg	2622	210	390	2832	3012	92.6%	87.1%	89.7%
m.avg(no class)	2669	163	343	2832	3012	94.2%	88.6%	91.3%

	tp	fp	fn	#pred	#exp	P	R	F1
brand	250	27	24	277	274	90.3%	91.2%	90.7%
drug	1849	95	278	1944	2127	95.1%	86.9%	90.8%
drug_n	1	10	71	11	72	9.1%	1.4%	2.4%
group	546	109	147	655	693	83.4%	78.8%	81.0%
M.avg	-	-	-	-	-	69.5%	64.6%	66.3%
m.avg	2646	241	520	2887	3166	91.7%	83.6%	87.4%
m.avg(no class)	2738	149	428	2887	3166	94.8%	86.5%	90.5%

Figure 4: Best MEM model performance on the Devel set (on the left) and on the Test set (on the right

4 Conclusions

During this assignment we faced the NERC task by using 2 different types of approach, a rule-based one and a ML one. The rule-based approach, based on simple heuristic rules, allowed us to build a sort of baseline to be used to assess the performance of the ML algorithm we built in the 2nd phase of the task. By using a list of simple rules, we reached performances both on the devel set and on the test set above 50%, which represent a robust baseline performsnce, from which starting the development of ML models.

We then moved to a ML learning approach for doing classification: this approach required us quite more effort in feature engineering, which is fundamental in order to build useful features for the classification task at hand. We dedicated most of our effort to this phase, since we know that feature engineering is a fundamental and difficult step in a ML pipeline. We know that this problematic phase is nowaday overcomed by a data-driven feature engineering, done by means of DL which allows an end-to-end learning, i.e. the DL algorithm learns not only the classifier function built upon the features, but it also learns the right features in order to classify.