Report 1 : NERC
AHLT - Advanced Human Languages Technologies

Lackmann Simon
Piotrowski Maciej

1 Introduction

This document is a report for the NERC project of the AHLT - Advanced Human Languages Technologies course. It consists of the implementation of a simple python program that parses all XML files in the folder given as argument and recognizes and classifies drug names. It is splitted in two parts: A baseline and a machine learning version of NERC. It's inspired by SemEval 2013 Task 9: DDIExtraction. We received some documents extracted from DrugBank (Drug description leaflets database) and MedLine (abstracts of medical papers) as the initial documentation. We also received three directories containing data files that we had to use as training, validation (devel) and test sets for our experiments. For both parts we received an initial structure with some initial rules that we had to improve. Basic F1 scores were 36% for the first part and 55%(CRF)/47%(MEM) for the second part. The goal was to reach respectively 45% and 74%(CRF)/66%(MEM).

2 Rule-based baseline

In this section, we describe the experiments we did with a rule-based baseline, what we observed during the data exploration and which rules we wrote according to those observations. Our baseline consists of an if-then-else cascade of simple rules. We didn't implement any statistical approaches for this section. The purpose was to find a lower bound for the performance of ML systems that we build in the second part of this assignement. We include our code and the results we obtained on the devel and test datasets with our best combination. We also describe all the other rule combinations we used that we think are relevant and the according results.

2.1 Ruleset construction

We obtained the best results with the following ruleset construction (in order):

1. If the string is already in our external database, we apply the category from the database.
2. If the last substring (five characters long) of the string are included in our "suffixes" list, we categorize it as a drug.
3. If the last substring (five characters long) of the string are included in our "suffixesGroup" list, we categorize it as a group.
4. If the string contains the substring "methyl", we categorize it as a "drug_n.
5. If the string contains the substring "acids", we categorize it as a "group".
6. If the string contains the substring "blocker", we categorize it as a "group".
7. If the string contains the substring "toxin", we categorize it as a "drug_n.
8. If the string contains the substring "gaine", we categorize it as a "drug_n.
9. Else, we consider it as an unknown ("NONE") category.

2.2 Code

```
def extract_entities(stext) :
# WARNING: This function must be extended to deal with multi-token entities.
# tokenize text
```

```
tokens = tokenize(stext)
result = []

# classify each token and decide whether it is an entity.
for (token_txt, token_start, token_end)  in tokens:
    drug_type = classify_token(token_txt)
    if drug_type != "NONE" :
        e = { "offset" : str(token_start)+"-"+str(token_end),
              "text" : stext[token_start:token_end+1],
              "type" : drug_type
             }
        result.append(e)
return result

## --------- tokenize sentence -----------
## -- Tokenize sentence, returning tokens and span offsets
def tokenize(txt):
    offset = 0
    tks = []
    for t in word_tokenize(txt):
        offset = txt.find(t, offset)
        tks.append((t, offset, offset+len(t)-1))
        offset += len(t)
    return tks

## ---------------------------------------------
## -- Lists used in the classify_token() function
suffixes = ['azole','mycin']
suffixesGroup = ['otics','tives','sants','ators']

## ---------------------------------------------
## -- check if a token is a drug part, and of which type
def classify_token(txt):
   import re

   #check if the string contains a digit
   def hasNumbers(text):
       return any(elem.isdigit() for elem in text)
   #check if the string contains any special character
   def hasSpecialCharacter(text):
       return any(not elem.isalnum() for elem in text)

   if txt.lower() in external : return external[txt.lower()]
   elif txt[-5:] in suffixes : return "drug"
   elif txt[-5:] in suffixesGroup : return "group"
   elif "methyl" in txt : return "drug_n"
   elif "acids" in txt : return "group"
   elif "blocker" in txt : return "group"
   elif "toxin" in txt : return "drug_n"
   elif "gaine" in txt : return "drug_n"
```

```
    else : return "NONE"
```

2.3 Experiments and results

With our best ruleset construction described in the section 2.1, we are able to reach a precision of 55.7%, a recall of 49.8% and a F1 score of 47.1% on the devel data. On the test data, we obtain a precision of 45.6%, a recall of 50.1% and a F1 score of 44.6%

	tp	fp	fn	#pred	#exp	P	R	F1
brand	319	252	55	571	374	55.9%	85.3%	67.5%
drug	1589	313	317	1902	1906	83.5%	83.4%	83.5%
drug_n	2	2	43	4	45	50.0%	4.4%	8.2%
group	179	354	508	533	687	33.6%	26.1%	29.3%
M.avg	-	-	-	-	-	55.7%	49.8%	47.1%
m.avg	2089	921	923	3010	3012	69.4%	69.4%	69.4%
m.avg(no class)	2241	769	771	3010	3012	74.5%	74.4%	74.4%

Figure 1: Best ruleset construction scores on the devel data

	tp	fp	fn	#pred	#exp	P	R	F1
brand	251	284	23	535	274	46.9%	91.6%	62.1%
drug	1631	298	496	1929	2127	84.6%	76.7%	80.4%
drug_n	1	6	71	7	72	14.3%	1.4%	2.5%
group	213	371	480	584	693	36.5%	30.7%	33.4%
M.avg	-	-	-	-	-	45.6%	50.1%	44.6%
m.avg	2096	959	1070	3055	3166	68.6%	66.2%	67.4%
m.avg(no class)	2264	791	902	3055	3166	74.1%	71.5%	72.8%

Figure 2: Best ruleset construction scores on the test data

We tried many other combinations as well but unfortunately we couldn't reach a better score with them. It only decreased it. I would say that the most important and the most tricky step was to remove the "elif txt.isupper() : return "brand"" condition which was already included in the baseline file we received. It wasn't obvious because as it was included in the original file, we were thinking that it is actually a good condition but as a matter of fact, when we take our best ruleset and add this condition, our F1 score decreases from 47.1% to 40.0% on the devel data and from 44.6% to 38.3% on the test data. One other big difficulty that we encountered during the experiment phase was the fact that the data on which we were supposed to find the different patterns, differs a lot from the devel data on which we were supposed to test our program. So some patterns that we found, even if they were relevant on our training data, weren't working well when tested on our devel data. Here are the most interesting conditions that we expected to perform well (but didn't) and the corresponding **F1 scores (devel/test)** if we add it to our best ruleset:

- If the substring "chlor" is included in the string, categorize it as "drug". F1 score : 40.0% / 44.5%
- If the last substring (five characters long) of the string are included in our "suffixesBrand" list, we categorize it as "brand". F1 score : 47.1% / 44.6%
- If the substring "depress" is included in the string, categorize it as "group". F1 score : 47.0% / 44.5%
- If the string contains some numbers, categorize it as a "drug_n". F1 score : 45.4% / 44.4%
- If the string contains some special characters (detecting them with the built-in isalnum() puthon function), categorize it as "drug_n". F1 score : 44.8% / 43.7%

Here are some of the bash commands we used to find some patterns in the training data:

- python ner2gold.py ../data/train/ | cut -d'|' -f3- | tr '[:upper:]' '[:lower:]' | sort | uniq -c | sort | grep "group"
- python ner2gold.py ../data/train/ | cut -d'|' -f3- | grep "toxin" | cut -d'|' -f2 | sort | uniq -c

The first command converts all the uppercase characters to lower ones, sort them alphabetically, remove duplicates, sort them again and shows the items belonging to the group category. The second command looks for all the items containing the substring "toxin", sort them alphabetically, remove duplicates and show the number of occurences per category.

3 Machine learning NERC

3.1 Selected algorithm

Selected algorithms for this task are CRF and MEM as well as different common machine learning algorithms included in the sklearn package. The following classifiers are used: Support Vector Machine, Random Forest, K Nearest Neighbors, AdaBoost and Naive-Bayes. Using this classifiers, there is some crucial disadvantage. Those implementations do not work on Categorical Features, which we have with our binary string features. There are two ways to make it work. On the one hand, using OneHot-Encoding, which translates each string feature into a high-dimensional sparse vector. This was an infeasible task for us, as it required us to change the structure of the framework and furthermore needed an incredible long time to preprocess, because we would have to store a csr-matrix for each computed feature, so we needed to discard this approach. On the other hand we tried to use an Ordinal Encoder, which just maps every appearance of a feature value into numbers. With the latter one, we were able to use the framework as intended. But it has one limitation, which could lead to worse results: The classifiers can assume some ordering on these values, which is not true. Moreover, we needed to change the feature extraction, as classifiers need to have a constant amount of features for every data point. At the end, we get some results worse than with CRF or MEM, but maybe they would be better with OneHotEncoding than with OrdinalEncoding. We discuss the results in the "Experiments and results" section.

3.2 Feature extraction

At first we tried to include different suffixes with length from one to six. Surprisingly, only a suffix length of three and five did improve the results, so we discarded the rest. We also added a feature for capital letters, which yields better results. Furthermore, we added the text as lower letters as a features, but discarded it as the results get worse. We implemented helper functions to search for numbers and special characters which are also helpful. Moreover, we took the databases from session1 into account, which obviously improves the result. For each of this features, we implemented it also for the next and previous word.

3.3 Code

Here we present the code for our feature extractor. At first we present the usual one, for models CRF and MEM. Afterwards we present the code with adaptions for other algorithms.

```
## --------- Feature extractor -----------
## -- Extract features for each token in given sentence
def extract_features(tokens):

    # load external databases HSDB and DrugBank
    external = {}
    with open("../session1/HSDB.txt") as h :
     for x in h.readlines() :
```

```
        external[x.strip().lower()] = "drug"
with open("../session1/DrugBank.txt",encoding="utf-8") as h :
 for x in h.readlines() :
        (n,t) = x.strip().lower().split("|")
        external[n] = t

# add some helper functions
# check and return True if text has any numbers
def hasNumbers(tokenForm):
    return any(elem.isdigit() for elem in tokenForm)

# check and return True if text has any special characters
def hasSpecialCharacter(tokenForm):
    return any(not elem.isalnum() for elem in tokenForm)

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

   # add the text as feature
   tokenFeatures.append("form="+t)
   # add a suffix of last three and last five characters as features
   tokenFeatures.append("suf3="+t[-3:])
   tokenFeatures.append("suf5="+t[-5:])

   # add feature if text is written in capitalized letters
   if t.isupper() : tokenFeatures.append("isupper")

   # if text has numbers/special characters, add a new feature
   if hasNumbers(t) : tokenFeatures.append("hasNumbers")
   if hasSpecialCharacter(t) : tokenFeatures.append("hasSpecialCharacter")

   # check same databases as in session1 and add a feature if we have a match
   if t.lower() in external : tokenFeatures.append(str(external[t.lower()]))

   # do the same things for next and previous words
   if k>0 :
      tPrev = tokens[k-1][0]
      tokenFeatures.append("formPrev="+tPrev)
      tokenFeatures.append("suf3Prev="+tPrev[-3:])
      tokenFeatures.append("suf5Prev="+tPrev[-5:])
      if tPrev.isupper() : tokenFeatures.append("isupperPrev")
      if hasNumbers(tPrev) : tokenFeatures.append("hasNumbersPrev")
      if hasSpecialCharacter(tPrev) : tokenFeatures.append("hasSpecialCharacterPrev")
      if tPrev.lower() in external : tokenFeatures.append(str(external[tPrev.lower()]))
   else :
      tokenFeatures.append("BoS")
```

```
        if k<len(tokens)-1 :
            tNext = tokens[k+1][0]
            tokenFeatures.append("formNext="+tNext)
            tokenFeatures.append("suf3Next="+tNext[-3:])
        else:
            tokenFeatures.append("EoS")

        result.append(tokenFeatures)

    return result
```

In the following we have the code for feature extraction for algorithms from sklearn, as for example Random Forest needs constant feature length. So for each feature we needed to add an "else" case in order to have the same numbers of features every time. Furthermore, we added the code for ordinal encoding into the pipeline, but this is in the main part of the extract-features.py script, so we did not add it here. In addition to that we had to made small changes to train.py and predict.py in order to read and handle the ordinal encoded data.

```
    ## --------- Feature extractor -----------
## -- Extract features for each token in given sentence
def extract_features(tokens):

    # load external databases HSDB and DrugBank
    external = {}
    with open("../session1/HSDB.txt") as h :
        for x in h.readlines() :
            external[x.strip().lower()] = "drug"
    with open("../session1/DrugBank.txt",encoding="utf-8") as h :
        for x in h.readlines() :
            (n,t) = x.strip().lower().split("|")
            external[n] = t

    # helper functions
    # check and return True if text has numbers
    def hasNumbers(tokenForm):
        return any(elem.isdigit() for elem in tokenForm)

    # check and return True if text has any special characters
    def hasSpecialCharacter(tokenForm):
        return any(not elem.isalnum() for elem in tokenForm)

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

        # add the text as feature
        tokenFeatures.append("form="+t)
        # add a suffix of last three and last five characters as features
        tokenFeatures.append("suf3="+t[-3:])
        tokenFeatures.append("suf5="+t[-5:])
```

```
# add feature if text is written in capitalized letters
if t.isupper() : tokenFeatures.append("isupper")
# for all of the features, add an opposite variable to have a constant amount of features
# for every data point at the end. Sklearn Classifiers needed the input to be in this form
else:
    tokenFeatures.append("Notisupper")

# if text has numbers/special characters, add a new feature
if hasNumbers(t) : tokenFeatures.append("hasNumbers")
else:
    tokenFeatures.append("NothasNumbers")
if hasSpecialCharacter(t) : tokenFeatures.append("hasSpecialCharacter")
else:
    tokenFeatures.append("NothasSpecialCharacter")
# check same databases as in session1 and add a feature if we have a match
if t.lower() in external : tokenFeatures.append(str(external[t.lower()]))
else:
    tokenFeatures.append("NotInExternal")

# do the same things for next and previous words
if k>0 :
    tPrev = tokens[k-1][0]
    tokenFeatures.append("formPrev="+tPrev)
    tokenFeatures.append("suf3Prev="+tPrev[-3:])
    tokenFeatures.append("suf5Prev="+tPrev[-5:])
    if tPrev.isupper() : tokenFeatures.append("isupperPrev")
    else:
        tokenFeatures.append("NotisupperPrev")
    if hasNumbers(tPrev) : tokenFeatures.append("hasNumbersPrev")
    else:
        tokenFeatures.append("NothasNumbersPrev")
    if hasSpecialCharacter(tPrev) : tokenFeatures.append("hasSpecialCharacterPrev")
    else:
        tokenFeatures.append("NothasSpecialCharacterPrev")
    if tPrev.lower() in external : tokenFeatures.append(str(external[tPrev.lower()]))
    else:
        tokenFeatures.append("NotInExternalPrev")
    tokenFeatures.append("notBoS")
else :
    tokenFeatures.append("formPrev=None")
    tokenFeatures.append("suf3Prev=None")
    tokenFeatures.append("suf5Prev=None")
    tokenFeatures.append("isupperPrev=None")
    tokenFeatures.append("hasNumbersPrev=None")
    tokenFeatures.append("hasSpecialCharacterPrev=None")
    tokenFeatures.append("InExternalPrev=None")
    tokenFeatures.append("BoS")
```

```
            if k<len(tokens)-1 :
                tNext = tokens[k+1][0]
                tokenFeatures.append("formNext="+tNext)
                tokenFeatures.append("suf3Next="+tNext[-3:])
                tokenFeatures.append("suf5Next="+tNext[-5:])
                if tNext.isupper() : tokenFeatures.append("isupperNext")
                else:
                    tokenFeatures.append("NotisupperNext")
                if hasNumbers(tNext) : tokenFeatures.append("hasNumbersNext")
                else:
                    tokenFeatures.append("NothasNumbersNext")
                if hasSpecialCharacter(tNext) : tokenFeatures.append("hasSpecialCharacterNext")
                else:
                    tokenFeatures.append("NothasSpecialCharacterNext")
                if tNext.lower() in external : tokenFeatures.append(str(external[tNext.lower()]))
                else:
                    tokenFeatures.append("NotInExternalNext")
                tokenFeatures.append("notEoS")
            else:
                tokenFeatures.append("formNext=None")
                tokenFeatures.append("suf3Next=None")
                tokenFeatures.append("suf5Next=None")
                tokenFeatures.append("isupperNext=None")
                tokenFeatures.append("hasNumbersNext=None")
                tokenFeatures.append("hasSpecialCharacterNext=None")
                tokenFeatures.append("InExternalNext=None")
                tokenFeatures.append("EoS")

            result.append(tokenFeatures)

        return result
```

3.4 Experiments and results

Here we present results we get for the devel and test datasets. As there was a lot of testing and a lot of classifiers, we will focus on the best results and feature combinations for every algorithm. For features, the combinations are explained in section "Feature Extraction" above. They yield the best results for our algorithms.

CRF:
For CRF tested different parameters like L2-regularizers and its coefficients. The best was the default regularizer and a "c2" coefficient of 0.05 instead of 0.1. Allowing more iterations did not help. We achieved results of 74.1% for the F1-Score on the devel dataset and 69% on the test dataset.

MEM:
For MEM we worked with the default parameters and achieved 68.5% on the test dataset and 63.3% on the devel dataset.

RandomForest:
For Random Forest we tested different sizes for the amounts of trees and criterions "gini" and "entropy". But it is hard to determine the impact as the algorithm is not deterministic. We have results in the range of 15 to 35%

F1-Score on both sets.

K Nearest Neighbors:
KNN was surpringsly good, we tried different values for the number of neighbors parameter and reached up to 43% on the devel set and 41% on the test set with 25 as neighbor parameter.

AdaBoost:
With AdaBoost, we reached 22% on devel set and 25% on test set.

Naive-Bayes:
Naives Bayes performed very bad, under 1% was achieved on both sets.

Support Vector Machine:
For SVM, we tested Kernels "linear" and "rbf", where we achieved 47% on devel and 46% on test dataset with kernel "rbf" and 32% and 35% with kernel "linear".

To summarize our results, we provide the following table where best results for each algorithm and dataset is presented.

Algorithm	Devel-Set	Test-Set
CRF	74.9%	69%
MEM	68.5%	63.3%
RF	43.1%	39.2%
KNN	43.0%	41.7%
AdaBoost	22.7%	25.6%
NB	0.3%	0.0%
SVM	47%	46.3%

4 Conclusions

We reached the recommended percentage of F1-Score for each proposed algorithm and also tried to improve it even more. For the first task, we reached a F1-Score of 47.1%. We tried a lot of rules to reach the best possible score and different combinations of them. Some of them improved the score and some decreased. It was quite unexpected because all the rules that we tried seemed to be relevant in our training data but the problem is that the devel data differs in some ways from the training data. That's why it was quite tricky to find good rules to improve the score.
Moreover, for the second part, we wanted to test some different algorithms, which are popular machine learning classifiers, which at least produced some decent results. Original algorithms CRF and MEM yield the best results with about 70%. Alternative algorithms of the sklearn package did perform very differently, from 0% up to 43% F1-Score. Maybe a further finetuning of the parameters would lead to better results, but we would expect that even with the best parameter choice they would not beat the performance of CRF and MEM. This shows how important it is to develop new algorithms for specific tasks and that there is no unique algorithm which solves every problem. Furthermore, the use of OneHot-Encoding instead of Ordinal-Encoding could possibly increase the results, but was unfortunatley not feasible in this pipeline.

The whole project was very interesting, it allowed us to have a first grip on one among many different tasks related to NLP. It allowed us to observe all the advantages and disadvantages of the different techniques, different models and even to discover some new ones. Of course, there's still a possibility to improve even more the models to obtain a better F1-Score (by finding some better rules or tuning even more the parameters) but we are happy

with the scores we reached and we considered that it's more interesting to experience different ML models and their advantages/disadvantages instead of focusing to reach the best possible score with only one model. Also, the project allowed us to put into practice the material we saw during the lectures and to get a better understanding of it.