Named-Entity Recognition and Classification of Drugs

Zofia Tarant, Marcel Pons

March 24, 2021

This report encompasses the first subtask of the SemEval 2013 DDI Extraction competition: name entity recognition and classification (NERC). The objective of this subtask is to recognize and classify pharmacological substances in biomedical texts from the DrugBank database and MedLine abstracts on the subject of drug-drug interactions (DDI corpus).

Two approaches are followed in order to perform the task: a rule-based and a Machine Learning approach. The first approach will serve as a baseline to compare with the other more complex models, such as using Machine Learning techniques.

Both implementations consist in writing one or more python programs that parse all the XML files of the DDI corpus, perform an entity extraction of the mentions of pharmacological substances and classify each one in either *drug*, *drug_n*, *brand* or *group*. The performance of the models is evaluated by means of an evaluate (`evaluator.pyc`) script provided by the instructors. The evaluator compares the results with a ground truth and returns classification metrics (precision, recall, F1, etc.) for each type, a microaverage and a macroaverage.

1 Rule-based NERC

In this model, simple rules are implemented for the recognition and classification, that is, a baseline model is built that will be compared with future models.

The final program consists on three functions: `tokenize()`, `token_type_classifier()` and `extract_entities()`.

Tokenization of sentences

In order to extract the tokens from sentences, `word_tokenize` and `stopwords` functions from the `nltk` library are used. The `tokenize()` function splits the given input sentence into tokens, and adds each token its start and end offset in the original sentence. The `.find()` method is used to determine the position of the first letter (start). By adding the length of the word to the start position, the location where the token ends is obtained (one unit is be subtracted since the sentence starts at 0).

Since the output tokens are going to be used for recognition and classification of pharmacological terms, the stopwords and punctuation are removed in this step, therefore reducing somehow the final program's workload.

Classifier

The `token_type_classifier()` is the function responsible for identifying which tokens are *drugs*, *drug_n*[1], *groups* of drugs, or pharmacological *brands*. The recognition and classification are based on simple *if, elif, else* rules. With the purpose of finding rules that are right in most of the cases a visual inspection and simple statistics are performed on the train dataset, from which by means of parsing all its XML files a dataframe is created with the names of the entities and their entity class (*drug*, *drug_n*, *brand* or *group*).

By `pandas` manipulation, we obtain a bar plot counting the number of entities in each type that are fully capitalized (i.e., uppercase words). It can be appreciated that the majority of capitalized words are of type *brand* (57% of the *group* words), followed by type *drug_n* (27.9% of *drug_n* words). On the other hand, the words that contain digits are also analyzed, where it can be seen that the type with more words with digits is *group*, with a 10% of its total words.

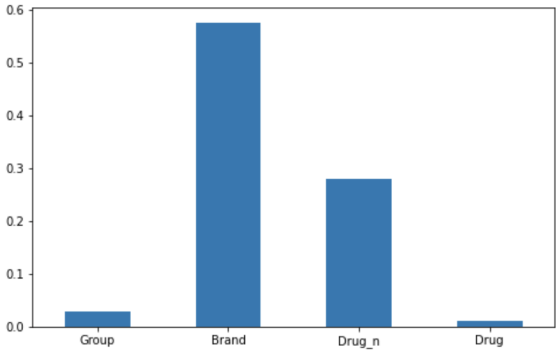

Figure 1: Nº of fully capitalized words

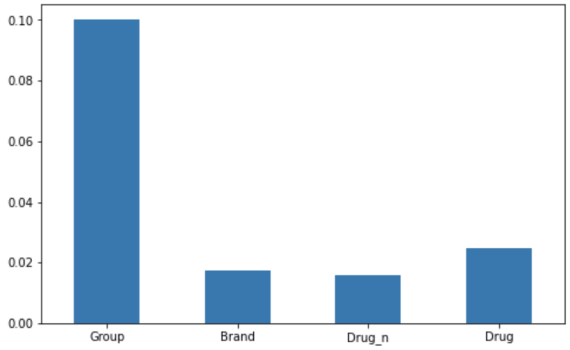

Figure 2: Nº of words containing number of signs

Besides the inspection of upper-cased words and the words including digits, the same procedure is carried out to analyze the pharmacological entities that contain spaces, that is, that are composed by more than 1 word (multitoken). In Figure 3 it is visualized that 42.26% of the *group* entities contain spaces.

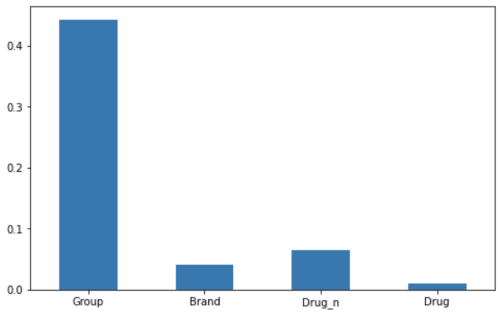

Figure 3: Nº entities with spaces

[1]Drug_n includes all chemical agents that affect living organisms but are not approved to be used in humans with a medical purpose, which differentiates this type from the drug type.

On the other hand, four dataframes are obtained, each one with the names of the entities of each type and their corresponding number of occurrences. From each dataframe manual and visual explorations are done to detect characteristics that determine whether a word is of certain type. Visual explorations consist on performing barplots depicting the number of occurrences of the top frequent terms on each type, whereas manual exploration consists on manual inspection of the dataframes, like inspecting the top n elements in each group (example of top ten in Table 2) and extracting the most common suffixes and prefixes (example of top ten suffixes in Table 3).

With these analysis, we found several characteristics. For example, the uppercase words of *brand* type tend to have a length greater than 4. On the other side the uppercase terms of *drug_n* tend to be lower than 4. We also noticed that several terms of type *group* tend to finish with capital letter + s. There are also characteristic suffixes, prefixes and substrings of each type. Furthermore there are drugs that appear significantly in biomedical text than others.

From the insights that are extracted from the inspection, the different rules of the classifier are established. It is worthy to mention that the order in which the rules are implemented is important, since they are executed sequentially.

1. If a word is uppercase and its length is greater or equal than 4, it is a *brand*.
2. If the word ends with some of the suffixes in a list previously created, it is a *drug*.
3. If a word contains `[''depressants'', ''steroid'', ''ceptives'', ''urates'', ''amines'', ''azines'', ''phenones'', ''inhib'', ''coagul'', ''block'', ''acids'', ''agent'', ''+'', ''-'', ''NSAI'',''TCA'', ''SSRI'', ''MAO'' ]`, and has a length greater than 8 and end with -s, it is a *group*.
4. If a word contains one of the following: `[''PCP'' ''18-MC'', ''methyl'', ''phenyl'',''tokin'', ''fluo'', ''ethyl'']`, is upper-cased with length between 2 and 4, it is a *drug_n*.

Bigrams are new "tokens" created by concatenating each pair of consecutive tokens, taking `offset_from` from the first one and the `offset_to` from the latter. By considering them, the final program will try to classify multitoken terms that appear in the biomedical literature referencing pharmacological entities.

Apart from these rules, the possibility of looking up resources is also available. If this option is activated, the first rules assessed consist in evaluating if a word can be found in the resource files `HSDB.txt` and `DrugBank.txt`[2]. For this purpose, we read those files and store the words in the variables `SimpleDrugDb` (list) and `DrugBank` (dictionary) respectively. We treat all words in `SimpleDrugDb` as drugs and check it first. `DrugBank` is more detailed, because the words are classified as *drug*, *brand* or *group*.

[2]The function `read_drug_files()` in charge of extracting the resources can be found in the Annex.

```
def token_type_classifier(word, should_look_up=False):

    threes = ["nol", "lol", "hol", "lam", "pam"]
    fours = ["arin", "oxin", "toin", "pine", "tine", "bital", "inol", "pram"]
    fives = ["azole", "idine", "orine", "mycin", "hrine", "exate", "amine", "emide"]

    drug_n = ["PCP", "18-MC", "methyl", "phenyl", "tokin", "fluo", "ethyl"]

    groups = ["depressants", "steroid", "ceptives", "urates", "amines", "azines",
              "phenones", "inhib", "coagul", "block", "acids", "agent", "+", "-",
              "NSAID", "TCA", "SSRI", "MAO"]

    if should_look_up:
        if (word.lower() in SimpleDrugDb):
            return True, "drug"
        if (word.lower() in DrugBank["drug"]):
            return True, "drug"
        if (word.lower() in DrugBank["brand"]):
            return True, "brand"
        if (word.lower() in DrugBank["group"]):
            return True, "group"
    if word.isupper() & (len(word) >= 4):
        return True, "brand"
    elif (word[-3:] in threes) | (word[-4:] in fours) | (word[-5:] in fives):
        return True, "drug"
    elif (True in [t in word for t in groups]) | ((word[-1:] == "s") & len(word) >= 8):
        return True, "group"
    elif (True in [t in word for t in drug_n]) | (word.isupper() & (len(word) < 4 & len(word) >=␣
↪2)):
        return True, "drug_n"
    else:
        return False, ""
```

Extract entities

The `extract_entities()` function calls the classifier function previously implemented for each token of a given tokenized sentence (with start and end offsets), takes the tokens that are either *drug*, *drug_n*, *brand* or *group* and returns the desired output of the main program: `[start-end] | name | type`. The parameter `should_look_up` is a boolean and indicates if the entities should be looked up in external files.

```
def extract_entities(sentence, should_look_up):

    output = []
    for t in sentence:
        token_text = t[0]
        (is_brand_drug_group, type_text) = token_type_classifier(token_text, should_look_up)

        if is_brand_drug_group:
```

```
            offset_from = t[1]
            offset_to = t[2]
            entity = {"name": token_text,
                      "offset": str(offset_from) + "-" + str(offset_to),
                      "type": type_text}
            output.append(entity)

    return (output)
```

Results of the Ruled Based Model

From these functions and the rules established, the program `baseline-NER.py` is developed, which takes as arguments the path where the XML are located, the name of the output file, and the optional flag `-l` to control whether the resources are used or not. The output of the program recreates the hidden gold annotation `IdSentence|startOffset-endOffset|text|type` which will be used to evaluate the performance.

The development dataset has been used in order to assess which rules are useful for the classification and which are not. In general, the macroaverage F-measure is 40% when drugs are looked up.

	tp	fp	fn	#pred	#exp	P	R	F1
brand	176	240	184	416	360	42.3%	48.9%	45.4%
drug	547	140	1366	687	1913	79.6%	28.6%	42.1%
drug_n	1	11	44	12	45	8.3%	2.2%	3.5%
group	176	588	505	764	681	23.0%	25.8%	24.4%
M.avg	-	-	-	-	-	38.3%	26.4%	28.8%
m.avg	900	979	2099	1879	2999	47.9%	30.0%	36.9%
m.avg(no class)	963	916	2036	1879	2999	51.3%	32.1%	39.5%

Figure 4: Results of Rule-based model without resources (devel)

	tp	fp	fn	#pred	#exp	P	R	F1
brand	280	434	80	714	360	39.2%	77.8%	52.1%
drug	1445	375	468	1820	1913	79.4%	75.5%	77.4%
drug_n	1	3	44	4	45	25.0%	2.2%	4.1%
group	263	801	418	1064	681	24.7%	38.6%	30.1%
M.avg	-	-	-	-	-	42.1%	48.5%	40.9%
m.avg	1989	1613	1010	3602	2999	55.2%	66.3%	60.3%
m.avg(no class)	2106	1496	893	3602	2999	58.5%	70.2%	63.8%

Figure 5: Results of Rule-based model with resources (devel)

With the test dataset, the macroaverage F-measure is 39.9% with drugs being looked up and 27.6% otherwise.

	tp	fp	fn	#pred	#exp	P	R	F1
brand	132	172	156	304	288	43.4%	45.8%	44.6%
drug	585	166	1535	751	2120	77.9%	27.6%	40.8%
drug_n	0	24	72	24	72	0.0%	0.0%	0.0%
group	186	596	513	782	699	23.8%	26.6%	25.1%
M.avg	–	–	–	–	–	36.3%	25.0%	27.6%
m.avg	903	958	2276	1861	3179	48.5%	28.4%	35.8%
m.avg(no class)	982	879	2197	1861	3179	52.8%	30.9%	39.0%

Figure 6: Results of Rule-based model without resources (test)

	tp	fp	fn	#pred	#exp	P	R	F1
brand	245	418	43	663	288	37.0%	85.1%	51.5%
drug	1534	397	586	1931	2120	79.4%	72.4%	75.7%
drug_n	0	7	72	7	72	0.0%	0.0%	0.0%
group	295	838	404	1133	699	26.0%	42.2%	32.2%
M.avg	–	–	–	–	–	35.6%	49.9%	39.9%
m.avg	2074	1660	1105	3734	3179	55.5%	65.2%	60.0%
m.avg(no class)	2235	1499	944	3734	3179	59.9%	70.3%	64.7%

Figure 7: Results of Rule-based model with resources (test)

Rules that were tried and discarded

We discovered that if we use the presence of digits to classify *groups*, the F-measure drops significantly. We also experimented with suffixes longer than 5 but found out that they were too specific and not particularly useful. After creating bigrams, we tried using trigrams as well, but we discovered that they negatively affected the performance.

We tried different orders of rules. We decided that the presence of the item in an external resource should have the highest priority. The next one is brand, because the rule *if a word is uppercase and it is longer than 3, it is a brand* has a very high precision. There are so few elements of class *drug_n* and they are so difficult to characterize that we decided that it would be the last rule.

2 Machine Learning NERC

In this model, a Conditional Random Fields model (CRF) is implemented in order to recognize and classify drugs in the biomedical texts.

Since in NLP applications like this one, features are related to appearing words, suffixes, prefixes, lemmas, etc., a feature extraction process has to be previously done.

Feature Extractor

The feature extractor takes as arguments the tokenized sentence and the boolean variable `should_look_up` indicating if the entities should be looked up in external resources.

Extracted features are related to the composition of the token, such as its suffix, if it is capitalized, if it is a number, if it is a stopword etc. Some features of the previous and next token are added as features of the current token as well. If the current token is the first word of the sentence, it is indicated by specifying that the previous token is `_BoS_` - the beginning of the sentence. Likewise for the last word of the sentence.

If the entities should be looked up in external resources[3], an additional feature is added: *Ruled*, where the value is indicated by the type found in the external resources, or **O** if the value is not found.

The extracted features are represented and described in Table 4 provided in Annex.

```
def extract_features(tokenized_sentence, should_look_up=False):

    features = []
    for i in range(0, len(tokenized_sentence)):
        t = tokenized_sentence[i][0]
        punct = [".",",",";",":","?","!"]

        tokenFeatures =  [
            "form=" + t,
            "formlower=" + t.lower(),
            "suf3=" + t[-3:],
            "suf4=" + t[-4:],
            "suf5=" + t[-5:],
            "prfx3=" + t[:3],
            "prfx4=" + t[:4],
            "prfx5=" + t[:5],
            "capitalized=%s " % t.istitle(),
            "uppercase=%s" % t.isupper(),
            "digit=%s" % t.isdigit(),
            "stopword=%s" % (t in stopwords),
            "punctuation=%s" % (t in punct),
            "length=%s" % len(t),
```

[3] In this scenario, `extract_features()` calls `use_db_resources()`, the code of which can be found in the Annex.

```
            "lemma=%s" % wordnet_lemmatizer.lemmatize(t),
            "numDigits=%s" % num_digits(t)]

        features.append(tokenFeatures)

        if should_look_up:
            read_drug_list_files()
            (is_drug, isType) = use_db_resources(t)
            if is_drug:
                tokenFeatures.append("Ruled = %s" %isType)
            else:
                tokenFeatures.append("Ruled = O")

    for i, current_token in enumerate(features):
        if i > 0:
            prev_token = features[i-1][0][5:]
            current_token.append("prev=%s" % prev_token)
            current_token.append("suf3Prev = %s" % prev_token[-3:])
            current_token.append("suf4Prev = %s" % prev_token[-4:])
            current_token.append("prevIsTitle = %s" % prev_token.istitle())
            current_token.append("prevIsUpper = %s" % prev_token.isupper())
            current_token.append("PrevIsDigit = %s" % prev_token.isdigit())
        else:
            current_token.append("prev=_BoS_")

        if i < len(features)-1:
            next_token = features[i+1][0][5:]
            current_token.append("next=%s" % next_token)
            current_token.append("suf3Next = %s" % next_token[-3:])
            current_token.append("suf4Next = %s" % next_token[-3:])
            current_token.append("NextIsTitle = %s" % next_token.istitle())
            current_token.append("NextIsUpper = %s" % next_token.isupper())
            current_token.append("NextIsDigit = %s" % next_token.isdigit())
        else:
            current_token.append("next=_EoS_")

    return features
```

CRF Modeling

Conditional Random Fields modeling is a statistical method which is useful when applying to terms that depend on each other. It uses graphical models, which represent the dependencies between the predictions. Particularly, when it comes to text classification, sequential dependencies are considered.

For this purpose, two programs are developed: `crf-learner.py`, which creates the train model, and `crf-classifier.py`, which tags the tokens using that model. The programs use the `Tagger` and `Trainer` from the `pycrfsuite` module to carry out the method.

The **learner** takes as arguments the name of the file where the trained model will be saved and the path to the extracted train features. It reads the data and feeds it to the CRF suite.

The CRF suite uses the default algorithm: `lbfgs`, which stands for Gradient Descent using the L-BFGS method, which is a Quasi-Newton optimization algorithm. It approximates the classic BFGS using limited memory. It is based on the approximation of the Hessian matrix of the function.

The values of the hyperparameters `c1`, `c2` and `epsilon` hyperparameters were optimized using 3-fold cross validation, with the help of `RandomizedSearchCV` from scikit-learn.

	Without	With
`c1`	0.34637301473721926	0.18367022602498456
`c2`	0.11659431627458852	0.09098395409834513
`epsilon`	0.12791638597501329	0.008871890649707953

Table 1: Best CRF parameters for the models trained without and with resources.

The trainer is trained for 100 iterations, and subsequently the model is saved to an external file.

The **classifier** takes as arguments, similarly, the name of the file where the trained model is saved and the path to the extracted features. The trained model is fed to the tagger and the dataset is classified sentence by sentence. Once all the sentences have been tagged, they are printed out in an evaluator-friendly format.

The `output_entities` function iterates over the list of tokens and their respective tags. The tokens with the tag **O** are ignored. If the tag begins with a **B**, it means the current token is the beginning of an entity. If one or more of the (next) neighbouring tokens are **I**nside tokens, they are concatenated and the full entity is printed.

Using the classifier to predict the devel dataset yields the results that can be found in the Annex Figures 8 and 9, *without* and *with* looking up resources. In general, the macroaverage F-measure is around 68% when drugs are looked up.

	tp	fp	fn	#pred	#exp	P	R	F1
brand	253	20	107	273	360	92.7%	70.3%	79.9%
drug	1509	118	404	1627	1913	92.7%	78.9%	85.3%
drug_n	6	6	39	12	45	50.0%	13.3%	21.1%
group	497	115	184	612	681	81.2%	73.0%	76.9%
M.avg	–	–	–	–	–	79.2%	58.9%	65.8%
m.avg	2265	259	734	2524	2999	89.7%	75.5%	82.0%
m.avg(no class)	2316	207	683	2523	2999	91.8%	77.2%	83.9%

Figure 8: Results of ML model without resources (devel)

	tp	fp	fn	#pred	#exp	P	R	F1
brand	280	13	80	293	360	95.6%	77.8%	85.8%
drug	1567	106	346	1673	1913	93.7%	81.9%	87.4%
drug_n	7	5	38	12	45	58.3%	15.6%	24.6%
group	496	112	185	608	681	81.6%	72.8%	77.0%
M.avg	–	–	–	–	–	82.3%	62.0%	68.7%
m.avg	2350	236	649	2586	2999	90.9%	78.4%	84.2%
m.avg(no class)	2387	198	612	2585	2999	92.3%	79.6%	85.5%

Figure 9: Results of ML model with resources (devel)

With the test dataset, the classifier performs worse with a F-measure of 63.1% with drugs being looked up and 61.1% otherwise.

	tp	fp	fn	#pred	#exp	P	R	F1
brand	220	44	68	264	288	83.3%	76.4%	79.7%
drug	1614	136	506	1750	2120	92.2%	76.1%	83.4%
drug_n	2	7	70	9	72	22.2%	2.8%	4.9%
group	531	163	168	694	699	76.5%	76.0%	76.2%
M.avg	–	–	–	–	–	68.6%	57.8%	61.1%
m.avg	2367	350	812	2717	3179	87.1%	74.5%	80.3%
m.avg(no class)	2465	251	714	2716	3179	90.8%	77.5%	83.6%

Figure 10: Results of ML model without resources (test)

	tp	fp	fn	#pred	#exp	P	R	F1
brand	244	24	44	268	288	91.0%	84.7%	87.8%
drug	1652	109	468	1761	2120	93.8%	77.9%	85.1%
drug_n	2	10	70	12	72	16.7%	2.8%	4.8%
group	514	162	185	676	699	76.0%	73.5%	74.8%
M.avg	–	–	–	–	–	69.4%	59.7%	63.1%
m.avg	2412	305	767	2717	3179	88.8%	75.9%	81.8%
m.avg(no class)	2491	224	688	2715	3179	91.7%	78.4%	84.5%

Figure 11: Results of ML model with resources (test)

What was tried and discarded

There were several features that were tried but we decided to discard. For example, the part-of-speech tag did not meaningfully affect the results. We also found that it is better to use the number of digits in a token rather than checking if it is a digit (`numDigits` vs `isDigit`). The integer value indicating the number of digits carries more information than a boolean, so that makes sense. We also discarded the feature indicating if the token contains the dash (-) character.

We tried using different algorithms for the CRF learner, such as `l2sgd` (Stochastic Gradient Descent with L2 regularization) or `ap` (Averaged Perceptron) with optimized hyperparameters, but we found that the default algorithm, Gradient Descent using L-BFGS, showed the best performance.

3 Conclusions

The initial analysis of the train dataset allowed us to distinguish several different rules that allowed us to build a baseline solution. The number of datapoints is rather small, so the options were limited. We struggled to find rules that would result in high precision. We discovered that bigrams slightly improved the score, especially for groups, of which over 40 percent are two words or longer. We managed to obtain a relatively high macroaverage F-score of 27.6% and close to 40% with external resources.

For the CRF method, we extracted features related to the structure of each word, such as the suffix, if it is capitalized, how many digits it has etc., but also qualities such as if it is a stopword or not and the features of the previous and following tokens. We tried different features to see if they had any impact on the performance of the learner. We also tried several algorithms and optimized the hyperparameters using cross validation. We struggled to surpass the 70% mark no matter what we used and ended up with the macroaverage F-score of 61.1% and 63.1% with external resources.

While working on this task, we experienced the most difficulties classifying *drug_n*. Its low F-score always significantly worsened the overall performance of both models. The representation of *drug_n* in all data sets is much smaller than other types, which makes it difficult to find meaningful rules that would be exclusive to *drug_n* and be effective enough to differentiate them from other entities. *Drug_n* and *drug* are very similar, they share common suffixes, and both *drug_n* and *brand* are often uppercase. Moreover, the small number of *drug_n* makes any misclassification more "costly" when it comes to the performance measures. What is more, the external resource `DrugBank.txt` does not specify *drug_n*, which also contributes to the low performance of this category.

To sum up, this task allowed us to gain some insight into the challenges involved in entity recognition and classification. The data sets were small and very unbalanced, which made it harder to find the features of the categories that were underrepresented. Moreover, the difficulty is increased by the fact that the differences between the categories are not very clear, entities tend to be similar in a lot of ways, which makes it challenging to distinguish between the classes.

4 Annex

Rule Based Model

Brand	Count	Group	Count	Drug	Count	Drug_n	Count
aspirin	51	tricyclic antidepressants	61	warfarin	167	PCP	22
INDOCIN	28	corticosteroids	51	digoxin	145	18-MC	16
TAXOL	25	NSAIDs	50	phenytoin	141	ibogaine	14
PEGASYS	23	contraceptives	47	theophylline	99	tetrahydropyridine	13
VIOXX	20	anticoagulants	41	lithium	94	MHD	12
Mexitil	17	barbiturates	40	ketoconazole	91	endotoxin	11
NIMBEX	15	antihistamines	39	alcohol	90	dmPGE2	10
AMEVIVE	15	diuretics	38	cimetidine	70	toxin A	9
Aspirin	14	antacids	35	cyclosporine	70	beta-endorphin	9
SPRYCEL	14	phenothiazines	35	carbamazepine	69	As(V)	8

Table 2: Top 10 words of each type

Drug	Count	Group	Count	Brand	Count
dine	307	tics	262	irin	65
pine	285	ants	250	OCIN	28
zole	283	tors	193	AXOL	25
mine	265	ines	186	ASYS	23
arin	241	ents	183	IOXX	20
line	218	rugs	117	itil	18
ycin	198	oids	109	VIVE	15
mide	190	ives	106	ROTM	15
rine	184	ates	88	MBEX	15
toin	177	kers	73	IVIL	14

Table 3: Most frequent 4 character suffixes found in *drug*, *group* and *brand*. *Drug_n* type is not included due to the fact there was not any suffix which considerably outnumbered the others.

Machine Learning Model

feature	description
form	the token
formlower	the token in all lowercase letters
suffn	n term suffix of the token, where n=3,4,5 are considered
prefxn	n term prefix of the token, where n=3,4,5 are considered
capitalized	boolean representing whether the token starts with capital letter
uppercase	boolean representing whether the token is uppercase
digit	boolean representing whether the token is a digit
stopword	boolean representing whether the token is a stopword
punctuation	boolean representing whether the token is a punctuation
length	the token length
lemma	the token root or lemma
numDigits	the number of digits a token contains
containsDash	boolean representing whether the token contains a dash
prev	previous token
sufnPrev	n term suffix of the previous token, where n=3,4
prevIsTitle	boolean representing whether the previous token starts with capital letter
prevIsUpper	boolean representing whether the previous token is uppercase
prevIsDigit	boolean representing whether the previous token is a digit
next	next token
sufnNext	n term suffix of the next token, where n=3,4
nextIsTitle	boolean representing whether the next token starts with capital letter
nextIsUpper	boolean representing whether the next token is uppercase
nextIsDigit	boolean representing whether the next token is a digit
Ruled	string *drug*, *brand* or *group* if found in external resources, *O* otherwise

Table 4: Extracted features

Functions implemented to use external resources

```
def read_drug_list_files():

    resource_file = open(SIMPLE_DB_PATH, 'r')
    lines = resource_file.readlines()
    global SimpleDrugDb
    SimpleDrugDb = set([d[:-1].lower() for d in lines])

    resource_file = open(DRUG_BANK_PATH, 'r')
    lines = resource_file.readlines()
    global DrugBank
    split_lines = [(line.split('|')[0].lower(), line.split('|')[1][:-1]) for line in lines]

    for name, n_type in split_lines:
        DrugBank[n_type].add(name)
```

```
def use_db_resources(word):

    if (word.lower() in SimpleDrugDb):
        return True, "drug"
    elif (word.lower() in DrugBank["drug"]):
        return True, "drug"
    elif (word.lower() in DrugBank["brand"]):
        return True, "brand"
    elif (word.lower() in DrugBank["group"]):
        return True, "group"
    else:
        return False, ""
```