Report 2: ML-based NERC and DDI

Àlex Martorell i Locascio
Pim Schoolkate

April 2022

1 Introduction and Report structure

The goal of this report is to present the results obtained from the Machine Learning techniques applied to 2 well-known NLP tasks: Name Entity Recognition and Relation Extraction.

This report has two main sections - which correspond to the two different tasks - detailing the steps taken during all the process. The goal is not to show only the best results, but try to and list all the options that were tried and why did not produce good results. Observe that the structure of both sections is identical: First, the choice of algorithm is justified. The choice maken is to give special importance to this part because it is only with a concrete understanding of the algorithm that the results of an experiment can be interpreted with maximum sense and efficacy. This explanation is followed by a detailed description of the feature vector (which ones were tried and kept, and which ones were discarded). A section with the code of the function is also included. Finally, the results are detailed as well as problems arisen.

2 Task 2: Machine Learning NERC

2.1 Selection Algorithm

Theory lectures have covered the *Name Entity Recognition* (NER) problem, where solutions given varied from Hand Crafted Rules to Discriminative models and Conditional Random Fields (CRF from now on). In this lab task, the options to use as a model were CRF, Maximum Entropy Markov Model (MEMM from now on) or one of choice. First observe that the Problem statement already suggests that CRFs are a better option. (Baseline case: 55%(CRF)/30%(MEM)) It is the purpose of this report to try to explain why CRF is a better choice.

In [1], CRFs were presented has an evolution of Hidden Markov models (HMM from now on) and MEMs for the Name Entity Recognition task. HMM although the powerful tool they are, historically have fallen short for Name Entity Recognition tasks. They are generative model are based on the Bayes Theorem and join, i.e:

$$P(X|O) = P(X_1, \dots, X_T|O_1, \dots O_T) \sim P(X_1, \dots, X_T)P(O_1, \dots, O_T|X_1, \dots, X_T) \tag{1}$$

where $X_1, \dots, X_T$ are the random variables that represent the tagging problem (B-I-O approach: drug, group, brand in our case) and $O_1, \dots, O_T$ are the observed words. The issue with generative models is that they are heavily dependant on the hefty calculation of $P(Y)$ and $P(X_1, \dots, X_T)$. Note that this can become very impractical if the number of features keeps growing or the dependency between words is of a high range. Conditional models appear as an alternative. They are discriminative models, in

the sense that they assume what $P(X|O)$ is like and then estimate the parameters. In a discriminative model instead of computing the joint probability distribution, $P(X|O = O_i)$ is calculated.

CRFs are seen as an evolution from MEMMs because they overcome called the **Label bias** problem. (Refer to [1] for a detailed explanation with examples). The **Label bias** problem is when transitions leaving a state S_i (brand, drug, group) only compete against each other, instead of all transitions in the model. Further down this is detailed with an examples. In MEMMs the probability is computed as following (making 2 important assumptions):

$$P(X_1, \ldots, X_T|O_1, \ldots O_T) = \prod_{i=1}^{n} P(X_i|X_{i-1}, O_i)$$

Note that the equality is possible if X_i is conditionally independent from the other variables given X_{i-1} and O_i. The general formula for computing this probability in MEMMs is:

$$P(X_i \mid X_{i-1}, O_i) = \frac{1}{Z(O_i)} \exp\left(\sum_a \lambda_a f_a(O_i, X_{i-1}, X_i)\right)$$

where $Z(O_i)$ is a normalization factor, λ_a are the parameters to be learned and f_a are the different features. The **label bias** problem is strictly related to the $Z(O_i)$ normalization factor.

Consider a graph that shows transition probabilities. If it were possible to look at an un-normalized graph i.e. the graph that shows the number of outgoing transitions from one node to an other (instead of the probability) we would observe differences. Consider the following example: [parenteral, antibiotics, are, generally, used]. The final tagging should be: [O, B-drug, O , O, O].

Recall that in our task the objective is to detect **sub-sequences** that represent **drug names** using the BIO approach. O indicates outside a sub-sequence, B beginning and I inside. Let us consider a fictitious case with the transitions S $\rightarrow$ (B-drug) $\rightarrow$ (O) and S $\rightarrow$ (O) $\rightarrow$ (B-drug). (S indicates the starting node of the subsentence.).

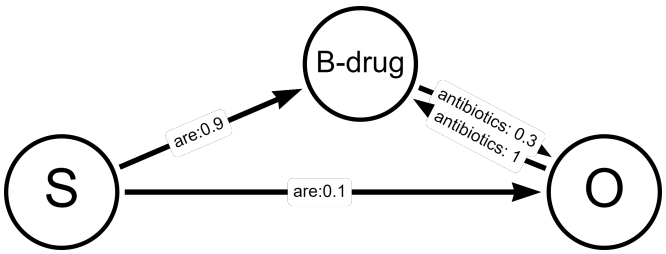

Figure 2.1: Simple NERC example

Observe that [B-drug, O] has a probability of $0.9 \times 0.3 = 0.27$ and [O, B-drug] has a probability 0.1×1.0. (This is because "Are" is not expected to be in the beginning of a sentence). Note how the least likely option has a higher probability. However, let us consider the un-normalized graph in 2.2 which just has the scores: [B-drug, O] has now $5 + 10 = 15$ where as [O, B-drug] has score $20 + 100 = 120$. This is before the scores have been exponenciated and normalized. In other words, the scores s come from the factor $\sum_a \lambda_a f_a(O_i, X_{i-1}, X_i)$. So "local normalization" is the problem here.

For a more deeper understanding, one might ask where does the label-bias come from. This is the concept of Conservation of Score Mass (Boutou, 1991). Summarizing and simplifying, this has to do with outgoing transitions, meaning that actual observations can be ignored if there is just one outgoing transition of a state. Low-entropy impacts negatively on the model. Entropy bias exceeds the purpose of this report, but it is crucial to understanding CRFs appeared.

In summary, CRFs are the solution to the label-bias problem. The normalization factor is tuned so it is a constant over the space defined by the tags.

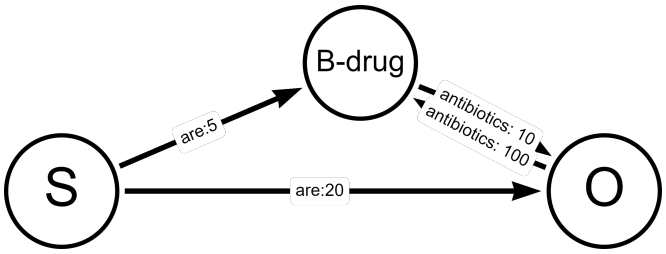

Figure 2.2: Simple NERC example un-normalized

Finally, if one refers to the `train-crf.py` file, there is the option to change some of the CRF model features when training the model. They seem to be optimal: With regards to the training algorithm, **L2SDG** (Stochastic Gradient Descent with L2 regularization term) gives better results than the other Gradient descent using the L-BFGS method, Averaged Perceptron, Passive Aggressive (PA) and Adaptive Regularization Of Weight Vector (AROW). L2SDG has a coefficient for penalty which is based on $\|\cdot\|_2$ norm (more complex models have larger penalties). The coefficent asssociated to the L2-norm is set to 0.1, and reducing it or enlarging it negatively impacts the performance of the CRF algorithm.

2.2 Feature extraction

Before trying different features, it is important to make some observation of the text that the model analyzes:

- **Suffixes.**A lot of drug names share the same suffix, or in other words, the possibilities of suffixes in drug naming belong to a limited set. For example, endings in "-in", "-ole", "-ine", "-ol" are very common. In the code provided, the only features considered were the suffixes of length 3. Suffixes of length 4 were considered, as well as the suffixes for previous and posterior words. Check 1 for details.

- **Word length.** The number of characters in a word is deemed important because drugs tend to have long names, but groups may have even longer names and more than word. On the contrary, brands tend to have shorter names.

- **Number of syllables.** A python package that estimates the number of syllables could be seen as an addition to the word length feature.

- **Punctuation / Structure of sentence.** It is reasonable to try and see if removing any of the data will increase the model's performance . However, quickly inspecting the structure of the sentence shows that symbols might be key in identifying drug names, as they can appear between parenthesis. The results in table 8 show that this is terrible choice.

- **Checking for special characters in particular**. Some entities present hyphens (-) or slashes (/) in their spelling. Not many other words in the English language have that characteristic, so a feature indicating this might help the . It is found that detecting hyphens does help the overall performance, but detecting slashes impacts it negatively.

- **All uppercase / lowercase.** The data trained seems to show some correlation between how drug names are labelled: Most drug names are all lowercase, whereas the brands tend to begin with an uppercase. Some entities are also found to be all in uppercase.

- **Word shape.** [3] gives an interesting option to include as a feature, which is to retrieve the word shape for each entity. The characteristics considered are 4: Uppercase and lowercase characters,

numbers and special characters (i.e. symbols). The word is encoded as follows (there are many possibilities):

$$\text{Adenosine} \rightarrow \text{Xxxxxxxx} \quad \text{or} \quad \text{Xx}$$
$$\text{C57BL/6N} \rightarrow \text{XOOXXoOX} \quad \text{or} \quad \text{XOo}$$

[3] present two different encoding possibilities: The whole word or just one instance if a one of the 4 possibilities listed appears. Observe that X and x denote Uppercase and lowercase, O is for numbers and o is for special characters.

- **Dictionary of Drugs.** A text file called DrugBank is used as a dictionary during model training. This database provided by the University of Alberta explicitly states if a given name is a brand, group, name. If a corresponding entity is found in this dictionary, two additional features are added to the vector indicating a match, as well as the kind of entity the DrugBank is labeled as. Note the amount of time needed to run this feature is considerable (around 2 hours) because it has to search through the whole text file, although this could be optimized by indexing. In the end, this did not help the performance.
- **Plurals.** Many group names happen to be plural (opioids, CNS depressants, tranquilizers). A feature is added to detect -s termiations. This slightly improves the overall score.

2.3 Experiments and results

Note. The baseline case includes the following features: Beginning/End of Sentence, Previous and Next Word, Previous and next 3-letter suffix. This is considered to be a 4 or 5 dimension feature vector.
Note 2. The scores of table reflect the performance of the prediction at a given state. Note that the letters in the left column indicate the version, meaning that for instance if the current version is D, that includes all the features of C.

Version	**Case**	**Max No. of Features**	**Macro F1**
A	Baseline	5	54.5%
B	Add length 3 suffix of 2 words before & after	7	58.0%
C	Length of word	8	61.6%
D	Length of previous and next word	10	61.9%
E	Checking if the whole word is uppercase	12	66.7%
F	Checking if the whole word is lowercase	13	71.8%
G	Checking for hyphen (-) symbol	14	72.0%
H	Checking for syllables	15	72.2%
I	Checking for "plural" *s*	16	72.6%

Table 1: Macro F1 score for kept features on the **devel** dataset

Let us go over the results summarized: One can clearly see which features work well, as we note the substantial increase in F1 score over 4 points in some cases. The length of the word, checking for uppercase and lowercase, and the suffixes of length 3 show to increase performance the most.

Until now the focus has been put especially put on improving the Macro F1 score. But looking at the F1 score per tag (3, it easy to see that `drug_n` entity was being predicted very poorly. The entity itself is already strange, since it does not show any significant . In the XML files, entities tagged with `drug_n` are also drug names, just like the ones that have the `drug`. It easy to see that this must impact

Tested with version	Feature	Macro F1
A	Suffixes of length 4	55.4%
F	Suffixes of length 3 and 4	69.5%
D	Removing special characters	16.3%
I	Prefix of next word	72.3%
I	Word shape	72.2%
I	Drugbank	72.0 %

Table 2: Macro F1 for discarded features

the model negatively, because it confuses. That is why that if we reassign all entities recognized as `drug_n` to `drug`) the Macro F1 score improves drastically (See 4).

Tag	F1
`brand`	86.4%
`drug`	91.3%
`drug_n`	30.3%
`group`	82.4%

Table 3: Detailed results for version I of experiment

Tags	Macro F1
`brand, drug, drug_n, group`	72.6%
`brand, drug, group`	86.4%

Table 4: Comparison of results once tag is removed

2.4 Conclusions

- Regarding the feature extraction, it was important to understand the structure of the sentence: Since this is a very specific problem related to Drug Name recongition, one must perform a basic analysis of the sentence in order to comprehend which features work and which do not. For instance, we observed that punctuation was key to identifying drug, brands, names. Uppercase and lowercase characters also showed relevance in how the words are identified. Suffixes are also a main characteristic of drug names (as they tend to be complex and/or stemming from Latin). It was important to note that the length of suffixes is also relevant (as suffixes of length 4 did not perform well neither on their own or as an addition to suffixes of length 3).

- CRF are seen as an evolution of the previous existing models and that is why. The label-bias problem happens in MEMMs and that is why it negatively impacts the performance execution of our problem. The goal was to explain why CRFs appear and what is the main difference with previous models.

- Finally, observe that a little change in the tagging improved the F1 score substantially. This is because a conflicting tag was lowering the performance. Although this is a preprocessing job more than feature extraction, it also relevant to take into consideration.

2.5 Code

```
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 = []
      t = tokens[k][0]

      tokenFeatures.append("form="+t)
      tokenFeatures.append("suf3="+t[-3:])


      # Length of word
      tokenFeatures.append(f"length={len(t)}")
      # Checking if the whole word is uppercase
      tokenFeatures.append(f"isUpper={t.isupper()}")
      # Checking if the whole word is lowercase
      tokenFeatures.append(f"isLower={t.islower()}")

      # Checking if the first letter is uppercase
      tokenFeatures.append(f"isfirstLetterUppercase={t[0].isupper()}")

      # Checking for special characters
      dash = any(c in "-" for c in t)
      tokenFeatures.append(f"hasDash={dash}")


      # Estimate syllables
      tokenFeatures.append(f"syllables={syllables.estimate(t)}")

      #Ending with s
      isplural = t.endswith('s')
      tokenFeatures.append(f"isPlural={isplural}")

      if k>0 :
         tPrev = tokens[k-1][0]
         # adding length of previous word
         tokenFeatures.append(f"lengthPrev={len(tPrev)}")


         #adding suffix 2 words prior
         if k>1:
            tPrev2 = tokens[k-2][0]
            tokenFeatures.append("formPrev2="+tPrev2)
            tokenFeatures.append("suf3Prev2="+tPrev2[-3:])
         tokenFeatures.append("formPrev="+tPrev)
         tokenFeatures.append("suf3Prev="+tPrev[-3:])

      else :
         tokenFeatures.append("BoS")

      if k<len(tokens)-1 :
         tNext = tokens[k+1][0]
         # Adding length of next word
         tokenFeatures.append(f"lengthNext={len(tNext)}")

         # Adding suffix 2 words posterior
         if k<len(tokens)-2:
            tNext2 = tokens[k+2][0]
            tokenFeatures.append("formPrev2="+tNext2)
            tokenFeatures.append("suf3Prev2="+tNext2[-3:])

         tokenFeatures.append("formNext="+tNext)
         tokenFeatures.append("suf3Next="+tNext[-3:])

```

```
66
67          else:
68              tokenFeatures.append("EoS")
69
70          result.append(tokenFeatures)
71
72      return result
```

3 Task 3: Machine Learning DDI

In order to complete this exercise, large portions of the code structure needed to be rewritten to a Windows compatible format, as we were unable to run the provided structure in Linux. Most notably, the `megam-64.opt` file, which provides the a maximum entropy GA model, does not run on windows and no clear alternative was provided. Thus a change was needed, which will be explained next in order of execution. The `extract_features.py` file has largely remained the same, with the exception of the additional features. After the features have been extracted, a script called `feature_collector.py` collects all features and transforms them in a dictionary, so that they are compatible with SKlearns `DictVectorizer`. The `model_builder.py` provides a `model` class which trains and stores both a classifier and a vectorizer. The former can be any kind of SKlearn classification model, which can be easily added to the structure of the code. The latter is important since the `devel.feat` have to be vectorized too, which can only be done properly with the same trained vectorizer. The `model` class also includes functionality to load the model again and perform a prediction, and thus in the `predict.py` file we reinitialize the classifier and vectorizer to predict the values. Lastly, the bash files have also been altered slightly to improve workflow. First, a separate bash for feature extraction (`extract_features.sh`) and for the training, predicting, and evaluation of the model (`train_predict_evaluate.sh`) were created. Another two bash files help starting up and killing the `CoreNLP` server, for testing purposes. With this structure, the application is compatible with all operating systems, instead of only Linux.

3.1 Selection algorithm

As mentioned above, it is possible to test different SKlearn classifiers. For this project, Decision Tree, Support Vector Machine, and Multinomial Naive Bayes (DT, SVM, NB from now on) classifiers were tested for their performance.

[4] and [2] present Information Extraction (IE) tasks using SVM, presenting it as one of the better models for this task. SVM are used in many NLP context (recall its application in Task 1), but the goal here is to explain how does it work for IE tasks. SVM defines a hyperplane or a set of hyperplanes in a n-dimensional space:

$$
\begin{aligned}
&H_1 : a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1 \\
&\vdots \\
&H_n : a_{1n}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n = b_n
\end{aligned}
$$

in order to separate n-dimensional points. Note that in a n- dimensional space the hyperplane might not be enough, and they are defined in an $n + m$ $(m > 0)$ dimensional space. This is why a kernel function is used to be able to compute the dot product between hyperplanes.

In this task, the vectors are of features. Note that lengths of vectors describing the relation between two entities are varying depending on the amount of features. In Task 2, the feature vector was per each token, whereas now it is based on the two entities. Also observe that the vector is designed (depicted in 5) in a one-hot encoding manner. Recall that a feature k may have many possible values, and more than one might occur in a single instance. Note that this vector is **sparse**. All in all, this

means `feature-collector.py` file has been modified to construct this vector that allows for SVM training.

Feature k					Feature k'			
v1	v2	v3	v4	v5	v1	v2	v3	v4
0	0	0	1	0	1	1	0	0

Table 5: Projection of a feature vector ready for SVM training

In general, the problem is to maximize the distance between the two hyperplanes (note that in our case the features are labelled with 1 and 0, which changes the calculations):

$$\begin{aligned} H &: a^T x - b = 1 \\ H' &: a^T x - b = -1 \end{aligned} \tag{2}$$

It is assumed without proof that the distance is between these two hyperplanes is $\frac{2}{\|w\|}$, so we must minimize $\|w\|$ with the constraints above.

One of the first things noted was that SVM perfoms way better than other classification algorithms (check section 3.4 for more details). Decision Trees are a good option for many classification problems, but in this the nature of the problem is not suited for this algorithm. The feature vectors are sparse objects and of high dimension, which results in a very complex tree. A tree of depth 70 was built which not only resulted in very long training time but also in a low improvement regarding accuracy.

Generally, NB outperforms other models if there is only a little bit of data. Simply, the determined probability is often stronger than a converged model on a few examples. In this case, the data is big enough that models such as SVM, DT or Neural Networks (which we unfortunately weren't allowed to use) converge well. This explains one reason of why NB is not well performing here. Another one is, that given that all probabilities are calculated by addition, it is impossible to model the interaction between features. Especially given the large amount of possible features, it is likely that some combination of two features is indicative of an interaction, which is not captured by NB.

3.2 Feature extraction

In this section we will be discussing the different features that were tested with the SVM classifier, as well as the features that we would have added given more time. We will discuss the main idea of each feature, as well as why it did or didn't improve the model.

- **Lowest Common Subsumer (LCS)**: Also called Lowest Common Ancestor, this is a feature based on the structure of the Constituency Tree. This is a syntactic feature as it is defined as the word that joins the two branches where each of the entities is contained. This could be key to identifying interactions as they rely heavily on the structure of a sentence. A verb usually might be a sign of an interaction between a Noun Phrase and the verbal objects (i.e. prepositional phrase), so the Lower Common Ancestor would be the sentence, in this case.

- **PoS of one word between entities**: A proposed feature that could help identify interactions by [4], involves checking if, when there is only one word between the entities, the word is a preposition. Namely, interactions are often described as for example *paracetamol **with** alcohol has an affect*. To be more inclusive, we include all cases of entities with one word in between, setting the value to the PoS tag. With this inclusion, a small improvement in terms of true positives and false negatives was observed.

- **PoS tags of all words in between**: The given code structure already included features, one of which denoted the lemma and tag in one value. This yields many features, which might be

too specific for the classifier, and thus it was opted to add a feature, which only denotes the tag of the words in the between. This is also an extension of the previous point. Adding this feature yielded a small improvement in true positives and false positives.

- **Syntactic paths**: A syntactic path can describe the relation between the entities through the different PoS tags. By only focusing on the PoS tags, we can uncover patterns such as the one described in the point of PoS of one word between entities, but also more complex ones that lead to interactions. For example, `NP<PP>NP` could indicate an interaction. The addition of the syntactic paths resulted in a small but notable increase in performance of the model, which mostly affected the false positives.

- **Distance between entities**: If two entities interact, the distance in words between them cannot be too big. As a matter of fact, most drug to drug interactions should occur between 1 and 20 words. It is rare to find interactions if the entities are spaced too far apart.

- **General clue verbs**: Verbs in a sentence are important because they are indicative of the action, which is can be key to predicting if an interaction between drugs exists, and what kind of interaction this is. In order to find clue verbs that help identify an interaction, we compared the occurrence of verbs in sentences with an interaction with those without an interaction and selected only those that occurred 90% more in sentences with an interaction. This heuristic was arbitrarily chosen, and not much explored due to the limited amount of time available. As can be seen in the results in table 7, this made barely any improvement. Two possible reasons for this were theorized. First, the heuristic might be flawed which would imply that it could be useful, but it was not executed correctly. Secondly, clue words might actually confuse the model more. It could be that only interaction that were already correctly predicted include the clue words, which would not yield any improvement. Instead, 10% of the pairs of entities without interaction will be labeled with an indicator for an interaction, resulting in more false positives. This is also reflected in the results.

- **Specific clue verbs**: Another option is to include a feature that looks for clue words specific to the type of interaction {effect, interaction, mechanism, advise}. For this, we used the same heuristic as above, except that now, the sentences with interaction were only selected for the specific interaction. The results, just like the general clue verbs did not yield any improvement, but rather, showed a slight worse performance. The main indicator for this was the increase of false positives, and the lack of increase in true positives. Our theorized reasons for this are the same as those described above.

- **Position of clue verbs**: Regarding the two entities interacting, the position of the clue verb is considered. This is because syntactically it has some meaning: A verb between two entities could be indicating an interaction. For this feature, there are three different possibilities: before, in between or after. This feature, however, seems to negatively impact the F1 score, most possibly because other features (such as syntactic paths) are more precise in their syntactic explanation of the sentence.

- **PoS Patterns**: During this project, we have been actively comparing results with other groups in order to validate and understand the working of the added features. Due to our time constraints, we were not able to implement a predefined PoS pattern feature ourselves. We were allowed to borrow code from Louis van Langendonck and Enric Reverter (highlighted in code) in order to see if this would improve our results. Remarkably, the addition of predefined PoS patterns increased the performance of the model the most. A good explanation for this, is that we are explicitly declaring how an interaction in terms of PoS tags should look like, instead of, like with the paths, considering all possible paths, and hoping that the model will learn which ones are indicative of interactions and which not (this might work with Neural Networks and

large amount of examples and iterations though). Unfortunately, we did not have enough time implementing this ourselves.

3.3 Experiments and results

Table 6 shows the baseline case F1 score for the three different algorithms that are tested for the DDI task.

Algorithm	F1 score
Decision Tree	27.9%
Decision Tree with depth 70	43.5%
Support Vector Machine	45.7%
Naïve Bayes	34.3%

Table 6: Baseline case F1 score with different algorithms

Version	**Case**	**Macro F1**
A	Baseline	45.9%
B	Distance between target entities	47.0%
C	Lowest Common Subsumer	49.8%
D	Preposition between entities	50.0%
E	All the words between entities	50.5 %
F	syntactic path	51.7 %
G	PoS patterns	55.0 %

Table 7: Macro F1 score for kept features on the **devel** data set using SVM

Tested with version	**Feature**	**Macro F1**
B	Specific words	46.6%
C	Clue words	49.0%

Table 8: Macro F1 for discarded features

3.4 Conclusions

This assignment posed several hard challenges. First of all, it was impossible to start the assignment until the last day, due to the incompatibility with Windows and the inability to run Virtual Machines on our computers. This severely impacted the time and effort that was put into assignment 3, and thus resulted in only a limited exploration to what is possible. We both regret not being able to dive in further as we really like this topic.

A success however, was the achievement of getting everything to run smoothly on Windows and providing the lectures with a multi-platform solution that can be adapted and used for future classes in order to reduce the time spend on software engineering and increase the time spend on mining unstructured data.

Now regarding the assignment itself, it was clear that the SVM classifiers outperformed both the DT and the NB. However, when comparing to our peers, who were able to run the `megam-64.opt` file, we noticed that they were able to achieve a better performance. We would have liked to explore this, but due to the above mentioned complications, and the poor documentation provided by the maker of `megam-64.opt`, the maximum entropy model remains to be somewhat of a black box to us.

Lastly, when selecting features for tasks such as text extraction, focus should be on the syntactic information that can be gained from the sentences. Especially when predicting if any interaction is present, this seems to be worthwhile, whereas for the type of interaction, indicative words might be more suited. These can be found manually, or trained by sophisticated models, which were out of the scope of this project.

3.5 Code

```
def extract_features(tree, entities, e1, e2):
    feats = set()

    # get head token for each gold entity
    tkE1 = tree.get_fragment_head(entities[e1]['start'], entities[e1]['end'])
    tkE2 = tree.get_fragment_head(entities[e2]['start'], entities[e2]['end'])

    if tkE1 is not None and tkE2 is not None:

        # features for tokens in between E1 and E2
        for tk in range(tkE1 + 1, tkE2):
            if not tree.is_stopword(tk):
                word = tree.get_word(tk)
                lemma = tree.get_lemma(tk).lower()
                tag = tree.get_tag(tk)
                feats.add(f"lib={lemma}")
                feats.add(f"wib={word}")
                feats.add(f"lpib={lemma}_{tag}")
                feats.add(f"pib={tag}")

                # feature indicating the presence of an entity in between E1 and E2
                if tree.is_entity(tk, entities):
                    feats.add("eib=1")

        if tkE2 - tkE1 == 2:
            feats.add(f"oneib={tree.get_tag(tkE1+1)}")

        # features about paths in the tree
        lcs = tree.get_LCS(tkE1, tkE2)

        ## Start Added features
        feats.add(f"lcs_stop={tree.is_stopword(lcs)}")
        feats.add(f"lcs_syntax={tree.get_rel(lcs)}")
        feats.add(f"lcs_tag={tree.get_tag(lcs)}")
        feats.add(f"lcs_lemma={tree.get_lemma(lcs)}")
        ## End Added features

        path1 = tree.get_up_path(tkE1, lcs)
        path1 = "<".join([tree.get_lemma(x) + "_" + tree.get_rel(x) for x in path1])
        feats.add("path1=" + path1)

        path2 = tree.get_down_path(lcs, tkE2)
        path2 = ">".join([tree.get_lemma(x) + "_" + tree.get_rel(x) for x in path2])
        feats.add("path2=" + path2)

        path = path1 + "<" + tree.get_lemma(lcs) + "_" + tree.get_rel(lcs) + ">" +
    path2
        feats.add("path=" + path)

        ## Start Added features
        path1 = tree.get_up_path(tkE1, lcs)
        # path1_words = "<".join([tree.get_lemma(x) for x in path1])
        path1_syntactic = "<".join([tree.get_tag(x) for x in path1])
        # feats.add(f"path1_words={path1_words}")
        feats.add(f"path1_syntactic={path1_syntactic}")
```

```
        ## End Added features

        ## Start Added features
        path2 = tree.get_down_path(lcs, tkE2)
        # path2_words = ">".join([tree.get_lemma(x) for x in path2])
        path2_syntactic = ">".join([tree.get_tag(x) for x in path2])
        # feats.add(f"path2_words={path2_words}")
        feats.add(f"path2_syntactic={path2_syntactic}")
        ## End Added features

        ## Start Added features
        # path_words = path1_words + "<" + tree.get_lemma(lcs) + ">" + path2_words
        path_syntactic = path1_syntactic + "<" + tree.get_lemma(lcs) + ">" +
    path2_syntactic
        # feats.add(f"path_words={path_words}")
        feats.add(f"path_syntactic={path_syntactic}")
        ## End Added features

        ## Start Added Features
        feats.add(f"distance={tkE2 - tkE1}")

        general_clue_lemmas = ['ingest','correspond','react','present','regulate','
    represent','counteract','antagonize',
                               'stimulate','describe','protect','augment','
    anaesthetise','deplete','blunt','switch',
                               'share','dictate','select','bear','exaggerate','accord'
    ,'shorten','wait','threaten',
                               'necessitate','warn','anticipate','expose','exert','
    continue','withdraw','term',
                               'alkalinize','fold','halogenate','sensitize','
    vasoconstrict','lengthen','recognize',
                               'progress','depolarise','precipitate','propose','watch'
    ,'isrecommend','postmarket',
                               'last','stand']

        advise_clue_lemmas = ['present','advise','undertake','switch','seem','dictate'
    ,'select','bear','hear','accord',
                              'wait','threaten','warn','deplete','exert','continue','
    exceed','conflict','outweigh',
                              'anaesthetise','precipitate','watch','function','
    beadminister','isrecommend','interrupt']
        effect_clue_lemmas = ['present','regulate','counteract','antagonize','
    stimulate','protect','augment',
                              'cecectomize','blunt','exaggerate','necessitate','expose
    ','term','halogenate','sensitize',
                              'vasoconstrict','lengthen','shorten','describe','
    recognize','progress','depolarise',
                              'weaken','propose','postmarket','stand']
        interaction_clue_lemmas = ['anaesthetise','exist','pose','depend','threaten','
    nondepolarize','kill']
        mechanism_clue_lemmas = ['ingest','phosphorylate','correspond','react', 'match
    ', 'present', 'share', 'promote',
                                 'empty','evidence','anticipate','alkalinize','market'
    ,'fold','hydroxylate','fall',
                                 'propose','threaten','average','represent','desire','
    leave','denote','last']

        for node in tree.get_nodes():
            if tree.get_lemma(node) in general_clue_lemmas:
                feats.add("gen_clue=1")
                feats.add(f"gen_loc={find_clue_position(tkE1, tkE1, node)}")
            if tree.get_lemma(node) in advise_clue_lemmas:
                feats.add("adv_clue=1")
                feats.add(f"adv_loc={find_clue_position(tkE1, tkE1, node)}")
            if tree.get_lemma(node) in effect_clue_lemmas:
```

```
                feats.add("eff_clue=1")
                feats.add(f"eff_loc={find_clue_position(tkE1, tkE1, node)}")
            if tree.get_lemma(node) in interaction_clue_lemmas:
                feats.add("int_clue=1")
                feats.add(f"int_loc={find_clue_position(tkE1, tkE1, node)}")
            if tree.get_lemma(node) in mechanism_clue_lemmas:
                feats.add("mec_clue=1")
                feats.add(f"mec_loc={find_clue_position(tkE1, tkE1, node)}")


        ## Features taken from Louis van Langendonck and Enric Reverter
        verb_list = ['reduce', 'induce', 'recommend', 'administer', 'produce', 'use',
    'enhance', 'give', 'result',
                     'receive', 'potentiate', 'coadminister'
            , 'decrease', 'cause', 'inhibit', 'report', 'take', 'contain']
        pattern_list = [('JJ', 'NN', 'IN'), ('DT', 'NN', 'IN'), ('DT', 'JJ', 'NN'), ('
    IN', 'DT', 'NN'),
                        ('NN', 'CC', 'NN'), ('IN', 'JJ', 'NN'),
                        ('JJ', 'NNS', ','), ('NN', 'NN', 'NNS'), ('-LRB-', 'NN', 'NN')
    , ('NN', 'IN', 'DT'),
                        ('IN', 'NN', 'IN')]
        pos_pattern = []
        for tk in (tree.get_nodes()):
            pos_pattern.append(tree.get_tag(tk))
            if 'VB' in tree.get_tag(tk):
                feats.add("verb=" + tree.get_lemma(tk))
                if tree.get_lemma(tk) in verb_list:
                    feats.add("typical_verb=True")
        threegrams = ngrams(pos_pattern, 3)
        for grams in threegrams:
            if grams in pattern_list:
                feats.add("typical_threegram=True")

    return feats
```

References

[1] John Lafferty, Andrew McCallum, and Fernando Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. 2001.

[2] Yaoyong Li, Kalina Bontcheva, and Hamish Cunningham. Svm based learning system for information extraction.

[3] Shengyu Liu, Buzhou Tang, Qingcai Chen, Xiaolong Wang, and Xiaoming Fan. Feature engineering for drug name recognition in biomedical texts: Feature conjunction and feature selection. 2015.

[4] Anne-Lyse Minard, Lamia Makour, Anne-Laure Ligozat, and Brigitte Grau. Feature selection for drug-drug interaction detection using machine-learning based approaches. Sep 2011.