

Deliverable 2 – NLP

Named Entity Recognition

Structured Perceptron and RNN

Authors:

Aleix Casellas Comas

Rubén Barco Terrones

Andreu Masdeu Ninot

Pablo Lázaro Terrones

Marco Gani Remane

Master on Fundamental Principles of Data Science

Natural Language Processing

16th of June 2019

Document structure

For the sake of clearness, we are going to explain in a few lines how we have divided the document. First part of the document is a description of the problem and how data looks like. Then we comment the Structured Perceptron for NER, the explanation of the method, how we worked to achieve the results, comparison between different outputs and a justification of these results. In the third part of the document, we talk about using RNN for NER and the structure followed in this section is very similar to the previous one. Finally, we make a comparison between both methods and explain strengths and weakness of each method.

1. Description of the problem

In this work we deal with the problem of Name Entity Recognition using a dataset that contains 47.959 different sentences. In order to train our models, we have divided the dataset into two subsets: training and test. We are going to use this division of the data for all the experiments using RNN and Structured Perceptron to compare the results.

We have prepared the data splitting it into two datasets and saving them as txt: 'X_train.txt', 'X_test.txt', 'Y_train.txt' and 'Y_test.txt'. The main data (X_train and X_test) consists of two lists of lists in which each list is a sentence splitted into words. The tags or labels follow the same structure: Y_train and Y_test are lists of lists in which each list contains the tags of the words of each sentence. So, for example an element (a list) of X_train and its correspondant element (another list) in Y_train have the same length:

```
X = ['The', 'arrests', 'occurred', 'Friday', 'in', 'Alexandria', '.']
Y = [ 'O',          'O',          'O', 'B-tim', 'O',          'B-geo', 'O']
Sentence = 'The/O arrests/O occurred/O Friday/B-tim in/O Alexandria
/B-geo ./O'
```

The tags used in this work are these entities:

- geo = Geographical Entity
- org = Organization
- per = Person
- gpe = Geopolitical Entity
- tim = Time indicator
- art = Artifact
- eve = Event
- nat = Natural Phenomenon

For each of these entities we are going to have two variations. For example, for each entity, we have B-per and I-per. B-per corresponds to the first word in the name and I-per corresponds to the following word in the name of the person (i.e. Rubén/B-per Barco/I-per Terrones/I-per). Besides, we have the 'O' entity, which is used for the rest of the words.

With all this information, we have created four python dictionaries: *word_to_pos*, *pos_to_word*, *tag_to_pos* and *pos_to_tag* with the functions *build_word_to_pos* and *build_tag_to_pos*. With this information we can create the *sequence_list_train* file, which is a list in which each element is a sequence of word/tag in numbers instead of words. We have created it with the positions corresponding to the words and tags using the

dictionaries that we have created. This SequenceList is the one that we are going to pass as argument to the ExtendedFeature function to create the ExtendedFeatureMapper in the Structured Perceptron case. All this information will be explained in the **Pre-processing and structure** subsection of the **Structured Perceptron** section.

The problem consists of training the RNN or the Structured Perceptron to predict the tags of each word in a sequence of words (i.e. a sentence). For this purpose, we are going to show the accuracy, the confusion matrix and the percentage of sentences without any label error to study the performance of each model and to be able to compare them.

2. Structured Perceptron

Model Descriptions:

The first model we have to implement to solve the problem of Named Identity Recognition (NER) is called Structured Perceptron (SP). The idea of this algorithm is based on Hidden Markov Models (HMM), but slightly different. The main difference with the classical HMM is that the Structured Perceptron updates w in order to correctly classify the training set. SP is a discriminative sequence model and it basically tries to solve the posterior probability of a sequence ($P(Y|X)$) instead of the joint probability ($P(X,Y)$) as generative models do. The main idea of this algorithm is shown in this table:

score	Hidden Markov Models	Discriminative Models
score _{emiss}	$\log P(x_i y_i)$	$w \cdot f_{\text{emiss}}(i, x, y_i)$
score _{init}	$\log P(y_1 \text{start})$	$w \cdot f_{\text{init}}(x, y_1)$
score _{trans}	$\log P(y_{i+1} y_i)$	$w \cdot f_{\text{trans}}(i, x, y_i, y_{i+1})$
score _{final}	$\log P(\text{stop} y_N)$	$w \cdot f_{\text{final}}(x, y_N)$

In Hidden Markov Models we use these conditional probabilities to compute the joint distribution of the sequence, but with a Discriminative Model we have a learnable weight vector w and a feature vector f . The score will not be anymore the log of a conditional probability, it will be the product of the weights times the feature vector. As we can see in the table the main features are the same (Emission, Initial, Transition and Final). If we look at the feature vector, we can see that it depends on x but as well on the output y . This is because the Structured Perceptron will compute different weights depending on y and the probability and the path that a word follows will determine the final weight vector. At this point, what we have to do is create new features apart from the HMM one that can improve the results of our model.

This features are local, because the HMM structure is in charge of compute relations between words, so now we have to add some local features to help the model. Some features could be if a word is uppercase, if it contains digits, if it starts by some suffix, etc. Once we have all features representing some conditional probabilities, we have to do decoding to obtain the most likely label y for the observation x . Here, we can use for example Viterbi decoding or Forward-Backward algorithm. Instead of computing all

cases for each path in our model (multiplying and summing all probabilities until we arrive at the final word) we can use Viterbi or Forward-Backward algorithm to compute this likelihood in a naive way (Dynamic programming). In the case of Viterbi, we just compute the multiplication of each conditional probability until that word between tags and words and we keep the argmax to obtain the best path. In Forward-Backward is similar but what we do is a similar forward step (Viterbi) and then a backward step that will compute all posterior probabilities needed to find the best tags for my sequence. In our case, our algorithm uses **Viterbi decoding** in order to compute the outputs.

To illustrate how we can add new features to the model, here we have an example of the first feature we added:

```
#Hyphen
if str.find(word, "-") != -1:
    # Generate feature name.
    feat_name = "hyphen:%s" % y_name
    # Get feature ID from name.
    feat_id = self.add_feature(feat_name)
    # Append feature.
    if feat_id != -1:
        features.append(feat_id)
```

As we can see, we just check if the present word (string) has a hyphen with the method `str.find` and then we give a name to this feature (`feat_name`). The SP will create as much numbers as it finds in the training corpus, so we will add a variable length features to our feature vector.

But there is a question that we always must answer in this kind of problems: What happens if a word never seen in the train is seen in the test set? The answer is simple: for the Structured Perceptron is not a problem. You train the model to obtain the best weights and with these weights you obtain a set of probabilities for the word even if it has not been seen in the training process, so the model will always predict a tag even if the model has never seen that word. The training set is used to obtain the best weights, it is not used to have a vocabulary that limits the words that can be decoded for the model.

Pre-processing and structure:

First, we have used only one notebook and you have links at the beginning to access the cells for each experiment easily.

As it is explained in the first point of the document, we have 47959 sentences of length variable and each word in the sentence has a different label. Is mandatory to divide the dataset in train (from 1 to 35970 sentence) and test (from 35971 to 47959 sentence). To do so, we have modified a little bit the code provided by the professor and take this numbers to divide correctly our sets. It takes a few hours to execute all the operations, so we decided to store in disk our sets needed for training and test. Once we have obtained our list of lists of words and sentences in train and test, we save it in disk and start creating the sequences.

The Structured Perceptron we use need a special object type to work and it is called *sequences* and *sequence_list*. To obtain these objects we just use an existing class

provided by the professor where we have to pass a list and it will become a sequence. If we pass more than one list, we will obtain a *sequence_list*. This will be the input of our model and it looks like this:

```
49/0 50/0 9/0 51/0 1/0 52/0 53/0 54/0 55/0 56/0 57/0 58/0 59/0 60/0 21/0
```

Here the first number we find before '/' is the word (the class creates automatically a tokenized version) and the number after '/' is the label. In the next case, we can see another sentence different from the previous one but translated in strings:

```
'Officials/O said/O Sunday/B-tim Jean/B-per Rene/I-per Anthony/I-per was/O killed/O in/O a/O shootout/O with/O police/O near/O  
the/O capital/O city/O of/O Port-au-Prince/B-geo ./O'
```

Once we have the *sequence_list* for training, we store it in disk to reduce time execution. The next step is to make experiments changing features to correct some error we can see in our test data. We have only one notebook for the Structured Perceptron and we have divided the notebook in experiments after all the pre-processing. While we go down, we can see the training and results of different experiments, always adding a new feature plus the others that were already added.

To initialize the SP, first we have to create a *feature_mapper* that is just a class (with *sequence_list_train* as argument) with dictionaries that contains all the possible features created with the features inside *structured_perceptron.py* and the training data. Once we have the *feature_mapper* we initialize our SP as follows:

```
corpus = skseq.readers.pos_corpus.PostagCorpus()  
sp = spc.StructuredPerceptron(word_to_pos, tag_to_pos, ex_feature_mapper)
```

As we can see, this class receive as argument the dictionaries previously created by us and the *feature_mapper*. To train the model we have just to call '*fit*' method, saving after X epochs the model on disk. Then we apply all the testing functions and implementations, creating our own functions to get faster all the results and more visually. At the end of each experiment, you can see some sentences with the real label it should have and the predicted label we achieve by testing our trained model.

IMPORTANT: Just as advice, we don't recommend running any cell of the Notebook. All the results that we think are important are showed in it. This is because for each of the experiments we change the *extended_feature.py* so, if you want to rerun, for example, all the computations of the experiment 2, you should first delete or comment the features of the experiments 3 to 20 in the *extended_feature.py* code to have only the exact ones that we have used in the second experiment.

Experiments and results:

In this section, we are going to show the results and give a justification of all of them. As well, we are going to discuss about the different numbers that are inside the confusion matrix and about the mistakes we had and how we tried to correct them, or we think it can be corrected.

Then, we are going to comment all the experiments that we have performed. After all these comments, we show a table with all the results, so if we refer to a number or a result in the following explanations, you can see better this result in the table. Also, if we refer to a test sentence or a correction of a prediction of a word, we show this example in the correspondent part of the experiment in the notebook.

Experiment 1

For the first experiment we have just used the `extended_feature.py` without adding anything. We wanted to have this model as a baseline and from here start to introduce new features in order to correct errors. In the table we can see all the results computed for this first experiment.

Experiment 2

Here we have added the 'hyphen' feature (if there is a '-' in the word), because we have seen that in the first sentence in *X_test* the word 'Port-au-Prince' is predicted as O and it would be B-geo. As we can see in the notebook, adding this new feature does not solve the problem, so we are going to introduce new features related to the upper letters in the word to try to solve this and to try to solve other problems with capital letters.

Experiment 3

We add here a feature if the first letter of the word is upper. We are not solving neither the problems mentioned before, but the model starts to recognise some words like Paris or Robin that were predicted before as O, even though the new predictions are still not good. With this feature, as we can see in the 10 test sentences, the model starts to predict something for the mistaken cities such as 'Barchelona'. We don't know if it is expected to predict these wrong names as their true tag or as O, but we suppose that they have to be predicted well, for example as B-geo for 'Barchelona'.

Experiment 4

Here we have introduced a feature for the verbs because they are one of the more important and meaningful parts of the sentences. We include a feature if the verb ends with '-ed'.

Experiment 5

We add another termination feature: '-ly'. This termination is really used in adverbs so it could help the system. Here we can see some improvements. Now Barcelona is predicted as B-geo and 'Barchelona' as B-org (bad prediction). In the tenth sentence of the test sentences, Arabia Saudi is well predicted as a geographic place, while in the previous experiments it was predicted as a person. But there are still a lot of errors. Some names are still being predicted as organizations as we can see in the 900th sentence of the test set.

Experiment 6

The sixth experiment consists of adding a feature if there are a point in any position of the word. We are trying this because we can see that the 'U.S.A' word in the seventh sentence of the test sentences is always predicted as an organization. In the results we see that this is not sufficient to detect 'U.S.A' as a geographical place, but it is useful to detect for example that 'Paris.' and 'Parris.' are B-geo (these two words have been previously detected as events).

We think that this feature provides small information to the system because in all the sentences in the train the final point indicating the end of the sentence is separated from the last word (the point is treated as another word), and in the 10 test sentences the points are taken as part of the last word, so we think that it is difficult to correct the predictions of these words.

Experiment 7

Here we have introduced a new feature for the word 'to' because it is common to find this word near words of geographical places. Here we have a clear example. If we look at the tenth sentence of the test sentences, in the previous experiment 'Saudi Arabia' was wrong predicted as a person, and in this experiment, it is predicted as B-geo and I-geo. We think that the presence of the 'to' feature can improve this examples because the whole sentence is: 'Robin does not want to go to Saudi Arabia.'

Experiment 8

We introduce here a feature that indicates that the word has an upper letter but without taking into consideration the first letter of the word.

For these small changes (note that we are only adding one feature per experiment) sometimes we are not going to tell a lot because we are not seeing or not finding important changes or variations in the results of the predictions. But we are introducing them because they could be helpful in some examples.

Experiment 9

Again, we introduce a word that should be near important words: 'of'. For example, we have two 'of' words in the eighth sentence of the test sentences, one between neutral words and the second in the set of words 'United States of America', and both predictions are correct.

Another example is that, now, 'city of Port-au-Prince' is well predicted.

Experiment 10

Here we introduce another suffix: '-ia'.

Experiment 11

We add another suffix: '-ing'.

Experiment 12

Again, we introduce a word that it is usually near important words: 'from'. But we can see in the two firsts sentences from the test sentences that this feature is not enough to make a correct prediction for 'Barcelona' and 'Barchelona'.

Experiment 13

Here we introduce two new features. One of them is a feature that indicates that all the word is a digit and the second one indicates that the word is not a digit but contains digits. We have introduced them because we had not added any information about numbers in the previous features. It is really difficult to find an example of the influence of this feature in the test set.

Experiment 14

The next experiment consists of adding a feature if the word is 'the'. It is such a common word that it is also difficult to find an example that is corrected using this feature, but we believe that it is an important feature for the model.

Experiment 15

Then, we have added another suffix: '-ent'. It is a common suffix for nouns and the majority of the words that we have to predict are nouns so we think that it is going to help.

Experiment 16

This feature is another important word: 'in'.

Experiment 17-20

At this point, we thought that including features one by one could be useful for seeing the impact of some of them in the predictions, but we also wanted to introduce a big set of features at one time to see if the performance of the model improves or not. One way to do this is introducing some prefixes and suffixes.

The last four experiments consist of adding some sets of suffixes and prefixes because there are good features for the Structured Perceptron model. Here are the suffixes and prefixes that we have introduced:

- Exp. 17: 15 suffixes → or, er, y, ful, al, ble, ize, en, fy, ate, ship, ist, ism, nce, dom
- Exp. 18: 18 prefixes → co, auto, dis, ex, anti, in, ir, im, il, un, up, trans, en, de, com, con, mis, re
- Exp. 19: 2 prefixes → homo, fore; and the apostrophe.
- Exp. 20: 13 suffixes → acy, ity, ty, ness, sion, tion, ise, ic, ical, ous, ish, ive, less

We started the first experiment with 38.298 features and, after adding features in the remaining 19 experiments, we have a total of **38.818** features. This means that we have added a total of 520 features to the model. If we compare the results obtained for both models:

- **Experiment 1:** 95.17% of test accuracy and 49.61% of correct test sentences
- **Experiment 2:** 95.25% of test accuracy and 49.95% of correct test sentences

we can see that we have improved approximately in 0.1% of accuracy and 0.34% of sentences corrected without any mistake in the test set. We will talk about these results better in the following lines in which we can see a table with all the results.

			Accuracy		Correct sentences	
			<i>Train</i>	<i>Test</i>	<i>Train</i>	<i>Test</i>
1	Default features	38298	96.29%	94.97%	54.59%	47.80%
2	Hyphen	38313 (+15)	96.25%	95.04%	55.52%	48.37%
3	Capital letter at the beginning	38330 (+17)	96.33%	94.75%	50.62%	45.97%
4	'-ed' suffix	38242 (+12)	96.34%	94.89%	51.29%	46.83%
5	'-ly' suffix	38353 (+11)	96.33%	94.94%	52.33%	47.41%
6	Points (.) anywhere	38366 (+13)	96.30%	94.82%	50.86%	46.29%
7	Preposition 'to'	38370 (+4)	96.33%	94.85%	50.85%	46.37%
8	Any letter is upper	38386 (+16)	96.34%	94.98%	53.21%	47.93%
9	Preposition 'of'	38397 (+11)	96.34%	94.91%	51.74%	47.38%
10	'-ia' suffix	38411 (+14)	96.34%	94.96%	51.75%	47.33%
11	'-ing' suffix	38427 (+16)	96.33%	95.20%	53.98%	49.14%
12	Preposition from	38430 (+3)	96.35%	95.13%	53.95%	49.00%
13	Digits + containing digits	38453 (+23)	96.33%	95.12%	54.95%	49.20%
14	Article 'the'	38460 (+7)	96.37%	95.03%	53.03%	47.91%
15	'-ent' suffix	38472 (+12)	96.34%	95.04%	53.09%	48.28%
16	Preposition 'in'	38476 (+4)	96.38%	95.17%	54.65%	49.61%
17	15 suffix groups (**1)	38622 (+46)	96.31%	95.04%	53.30%	48.04%
18	14 prefix groups (**2)	38687 (+55)	96.34%	95.20%	54.90%	49.60%
19	2 prefix (-homo,-fore) and apostrophe	38702 (+15)	96.36%	95.19%	54.34%	49.74%
20	13 suffix groups (**3)	38818 (+116)	96.31%	95.25%	55.09%	49.94%

¹ Suffix: or,er,y,ful,al,ble,ize,en,fy,ate,shit,ist,ism,nce,dom

² Prefix: co,auto,dis,ex, anti,[in,im,ir,il],un,up,trans,en,de,[com,con],mis,re

³ Suffix: acy,ity,ty,ness,sion,tion,ise,ic,ical,ous,ish,ive,

To obtain these results we have used Structured Perceptron with the number of epochs fixed to 15 to compare each experiment. Maybe if we let the model go longer (30 epochs) it could improve, but it will take a lot of time to train. For this reason, we chose 15 epochs. In grey, we have the baseline model and in yellow the best model in terms of test accuracy.

First, we can see in the table 20 different experiments. In each experiment, we add new features plus the other features in the previous experiments. We start with the baseline model (without adding any feature). If we have a look at the results, we do not too much improvement. Comparing the last model with the first one, we have improved 0.28% in test and at sentence level more than 2%. This tell us that is difficult to improve a Structured Perceptron just adding a few features and we need more features to improve the results.

If we look at the global features, we can see that we have added comparing to the baseline model more or less 520 features in total. With 520 we have just improve 0.28% in test, so maybe to improve 2% we need thousands of features. We did not create more features because the lack of time and the computational power it requires. Each model last approximately one hour to train and a few minutes to test, so we are not able to include many features because of this.

In the next section, we will talk about how we improve individually each mistake with the presence of a new feature, but we can sense from the table that including suffixes and prefixes always improve the accuracy. Therefore, in future work we can add more of these features in order to have a better model.

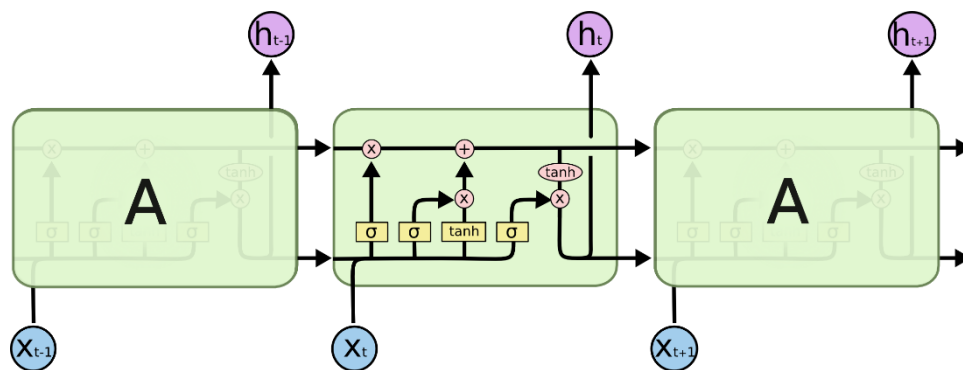
If we look at the results when we include a preposition it does not improve too much and sometimes it decreases the accuracy, but it can fixes local errors as it is shown in the notebook. Maybe it could be a good idea to erase some of them, because it for instance the preposition 'from' only includes three new features and the accuracy for test decreases, as well for sentence level.

3. Recurrent Neural Networks

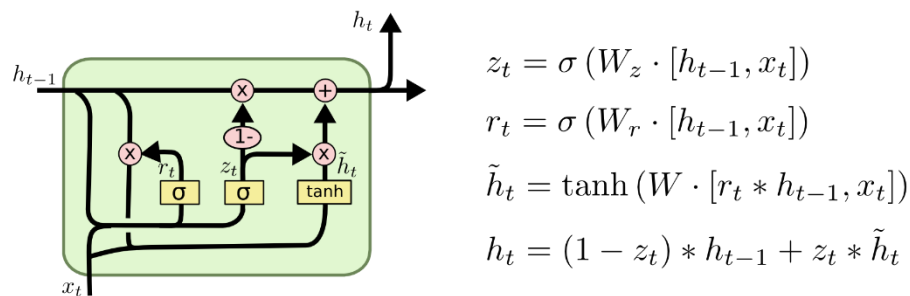
LSTM

We have tried different methods using LSTM to face the problem of NER. First of all, a simple bidirectional LSTM that just takes as an input the words of the sentences and predicts the tags, the named entity. The second one takes also the words of the sequences, but also each character of each word as an input. The third one, it uses the words and also their POS (part-of-speech). We are going to explain the results obtained with each method, as well as their strengths and weaknesses. As we don't want to put all the information and images in the memory, we will explain better are in more detail the last model, which is the one that has obtained the best results. The examples of sentences (both from dataset and the ones provided in the pdf) are just going to be displayed for the last method.

LSTM's are a type of recurrent neural network designed for solving the vanishing gradient problem of classical RNN's and also the fact that vanillas RNN could not learn long-term dependencies in text. For that, LSTM contains different states where different information is stored. The following diagram shows the operations carried inside a LSTM cell.



The key idea is the cell state, the line in top of the diagram. This cell state runs along the network with minor modifications, allowing information to persist over time. The three gates of the LSTM let it add or delete new information from the cell state. Another kind of recurrent unit are GRU's. GRU's are similar to LSTM but less complex. They merge two gates into a single one and the hidden state and the cell state are merged as well. In this task we have explored the use of both units. The following graph shows the structure of a GRU.



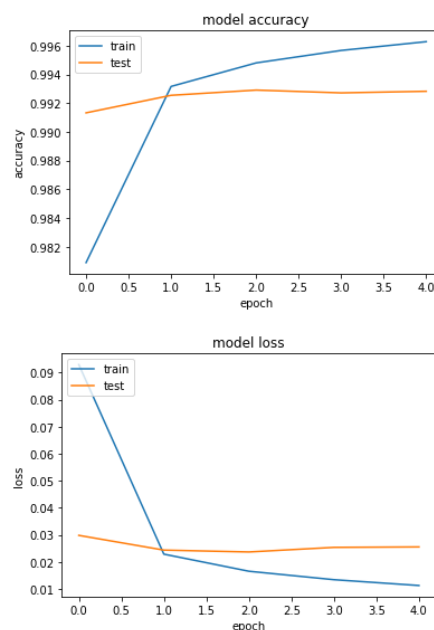
There is a preprocess part that is common for all models, explained as follows. After reading the dataset, we keep all the different words, as well as the different tags. We then create three dictionaries, for word, pos and tag, which associate to each element

a value. Note that for the words and pos, the values that we give start in 1, since we later set the element 0 to be 'PAD'. After getting the sentences from the data, separately taking word, pos and tags, we will create the variables X and y, the data and labels. We then need to pad the sequences to make them all equal. We would like to avoid this, but since we are going to use Keras and it is based on tensorflow, we have to know the lengths of our sequences at the beginning. A solution of this would be using pytorch, but since we don't have any previous knowledge about it we decided to stay with Keras. Once we have all the sentences of the same length, we transform the labels y to be one hot encoding. We finally create three dictionaries that given the value of word, pos or tag, it returns the element that it refers.

Model 1:

The first model is a simple RNN consisting of just an embedding layer followed by a bidirectional LSTM and ended with a Dense layer using TimeDistributed. We use 'adam' optimizer and 'categorical_crossentropy' as a loss function since we are doing a multiclass problem. We train the model for 10 epochs using a batch size of 32. We use as a validation set directly the test set that was proposed. We use the variable 'patience' to do early stopping and avoid overfitting and training without making any progress.

The results that we obtain in terms of accuracy and loss are the followings.



As we can see, the accuracy, both in train and test is really high. We see that after the first epoch, the validation accuracy and loss doesn't increase (decrease for loss) a lot. The program stops because of the 'patience' at the epoch 5.

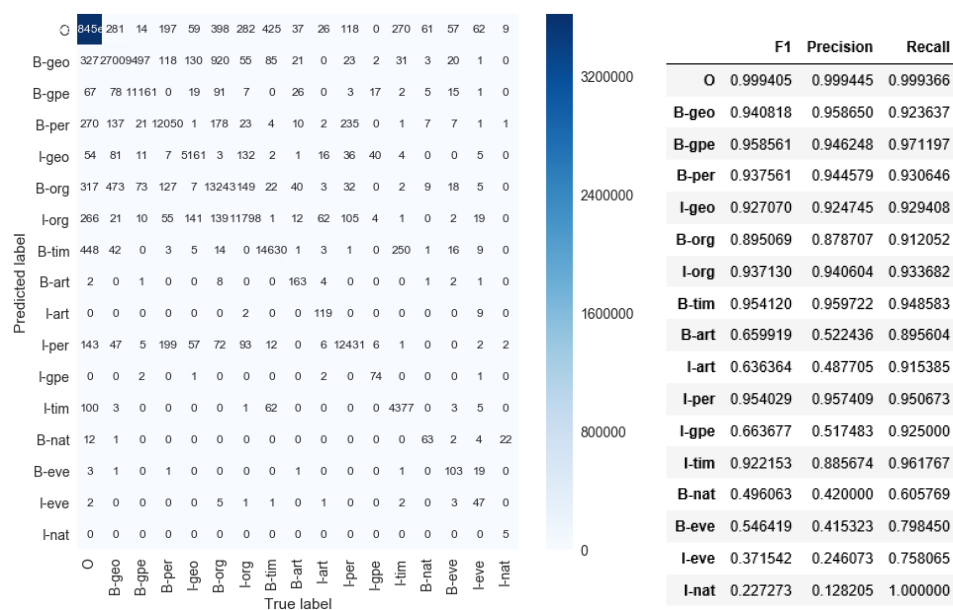
We create a function to compare a desired sentence of the dataset and its prediction, 'show predictions'. If you run it a few times, you can see that qualitatively the results obtained are quite good.

Finally, we check the number of correct sentences (all the predicted labels are correct) over the total sentences. We obtained as a result a value of **61.33%** for the test dataset

and **81.91%** for training. This result is more accurate than just the train and test accuracy, which are of **99.63%** and **99.28%**. We are adding a lot of 'PAD' words when we do 'pad_sequences', and we predict these labels always correctly. That is why the accuracy is always very high, and we have to focus on other results such as the fully correct sentences and the f1 score to check if a model is good or not.

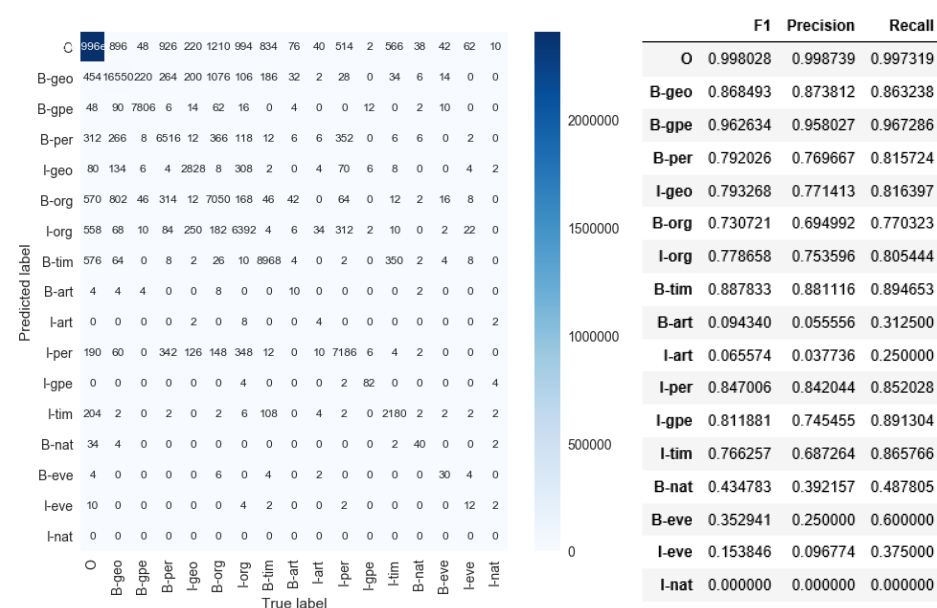
So we have also checked the F1 score and Confusion matrix for train and test. We can see the results obtained in the next images.

Train:



Looking at the results, we can see that the tags that most appear and more accurate are 'O'. We also get really good results for other tags like 'B-geo' or 'B-time'. The tags with a lower f1 score are usually the ones that appear just a little bit in the dataset, like 'I-nat' or 'I-eve'. We get as a mean f1-score of **76.63%**, which is quite good.

Test:



The results obtained in test are worse than in train, we get a mean f1-score of **60.81%**. But the patterns on the confusion matrix and f1 score are more or less the same. In that ones that we had a lot of data we have a high score, and in the ones we a few examples sometimes the value is pretty low.

With this method we obtain a great accuracy for the train and test sets, and acceptable f1-score for training and good enough for test. But the problems are with the unknown words or the misspelled ones. This method will predict all the new words with the tag 'O', even if they start with a capital letter and could be the name of a person or country. Also, if there is a misspelled word pretty close to a real one, for example 'Barchelona' or 'Microsof', it will take this word as a new one and predict it with a 'O' tag. We can see that in the results of the notebook 'RNN' in the bottom part, using the function 'make_prediction_text'.

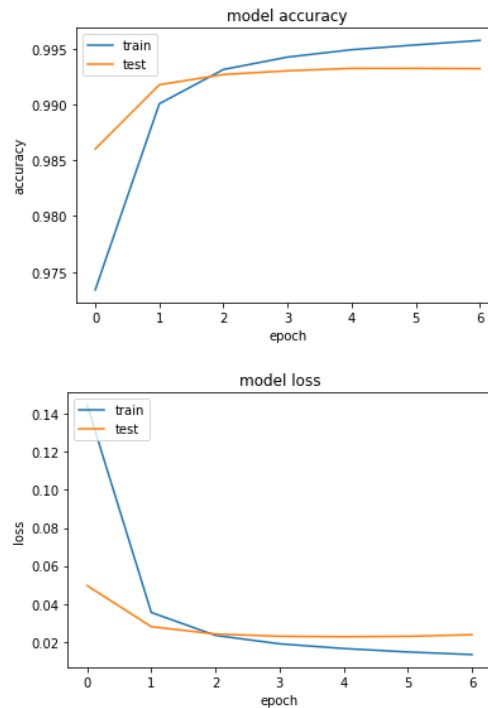
Model 2:

As we have mentioned before, this method concatenates two values as input. We take also the words as input, and additionally all the characters that make those words. The first entry, as before, is of size (47959,104) and the second one of size (47959,104,15). So for each of the 47959 sentences, we have a matrix of 104x15, i.e 104 words (max length of a sequence) and each of this word is described at most with 15 characters. If the word is longer than 15 characters, we just take the first 15.

The idea behind using this method is for solving the problem of unknown or misspelled words. It seems that it should work better for these cases, since you are now also focusing of the characters that form each word, and you could get patterns like when a word starts with a capital letter, it is never tagged with 'O'.

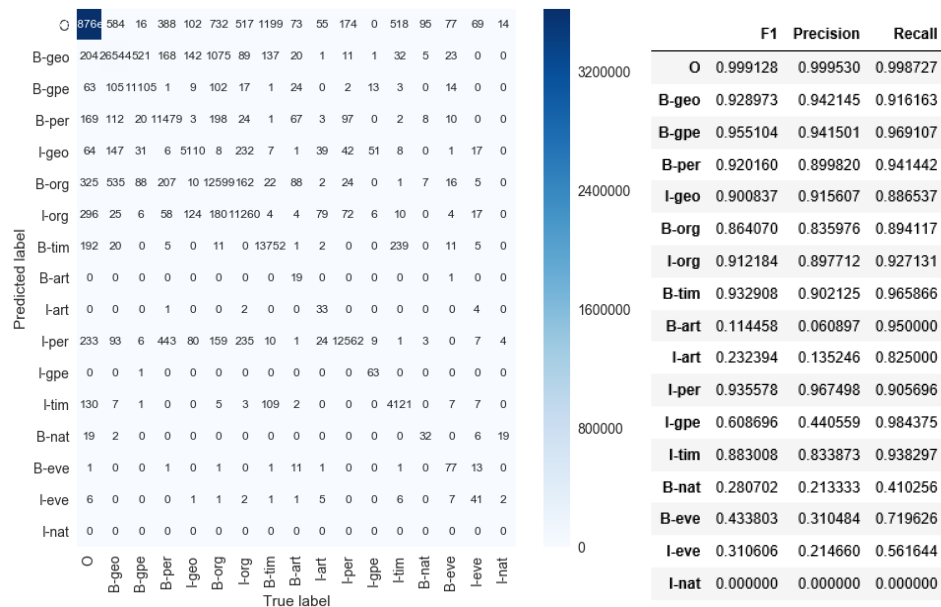
We need to create a similar dictionary that we did before, in this case for all the existing characters of the sentences. The model also uses a bidirectional LSTM, but we have to modify a little bit the previous one, since we have 2 different inputs now. First of all, we create the embeddings for the two different inputs, the words and the characters. We then concatenate them, and we apply a Bidirectional LSTM to it. As before, we finally put a Dense layer with a softmax to retrieve one of the 17 tags.

We also train for 10 epochs, but due to early stopping we finish the process earlier, in 7. The accuracy and loss results are the following graphics

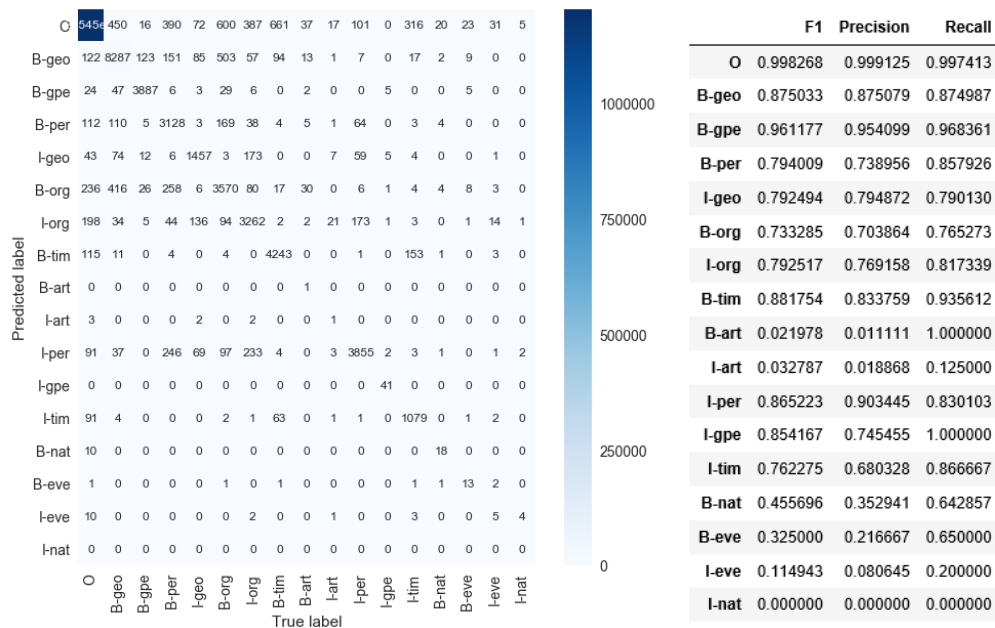


As before, the results obtain both in train and test are almost perfect, **99.57%** and **99.32%**. But as I said, this are not the results that we have to look at. In terms of f1 score and confusion matrix, we were expecting to get better results than before, but the numbers indicate the opposite.

Results of train:



Now we get as mean f1-score of train less than before, around **65.96%**. And if we look into more detail to the results in both graphics, we see that they are also worse. In the test set, the results are more similar to the first model.



Now the mean f1-score obtained is **60.35%**.

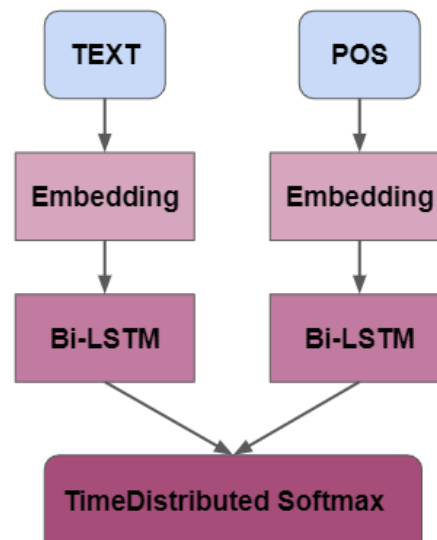
If we look at the fully correct sentences predicted, the results obtained are **61.80%** for test and **76.95%** for train, also worse than before. So this model seems not to improve the previous one, and also if we have a look to the sentences that you give us to predict (the ones in the pdf), it does exactly the same as the model 1. It is also not able to predict a tag different to 'O' to 'Barchelona' or 'Microsof'. We thought that this was a method that could work, but looking at the results it seems that this is not the way to proceed. That is why we created the third and last method.

Model 3

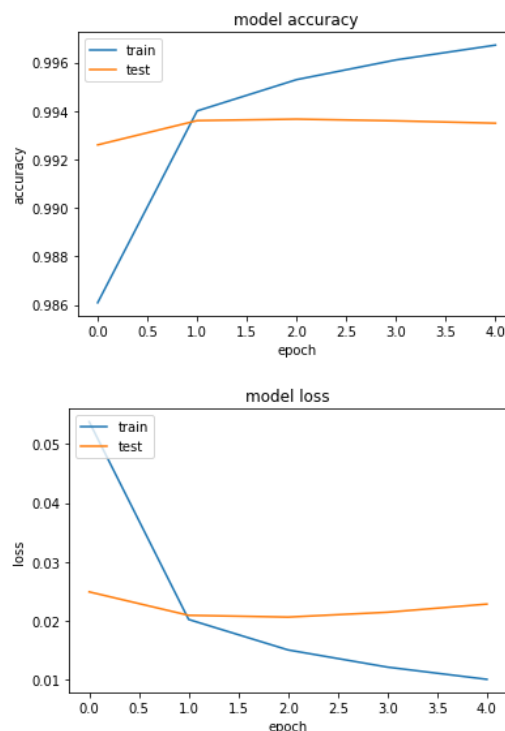
In order to treat unknown words better, we can also use the POS column of the dataframe. For unknown words we will at least have known POS so maybe this can help our LSTM predict accurately tags for unknown words. The idea is then, that instead of a single sequence, we have two sequences of the same length, one containing tokens representing words, and the other one containing tokens representing POS tags. This new feature is also sequential, so the same baseline model, the one that only takes words into account.

In fact, what we do is to create two separate LSTM networks, each one preceded with a embedding layer. One processes text information and the other one processes POS tagging information. If a word has not been seen so it does not have a token, the token for the unknown words is used. However, since this token is not used during training the embedding will be random for the unknown word. But the information from the POS will not be random. Therefore, we fuse the features learned by the LSTM's in a single feature vector, of shape (104, 300). As before, we have one feature vector for each word, where 200 features come from text information and 100 features come from POS tagging information. We are also using a Bidirectional LSTM, and a Dense layer for each word with a softmax activation for TAG classification. We use the TimeDistributed layer of

Keras to usely apply a Dense layer to each word or “timestep” of the sequence. The following figure is a graphic representation of the model.



The network is trained with the ADAM optimizer and categorical cross entropy as loss function. If we let the network train for 8 epochs with batch size of 32 we obtain the following plot:



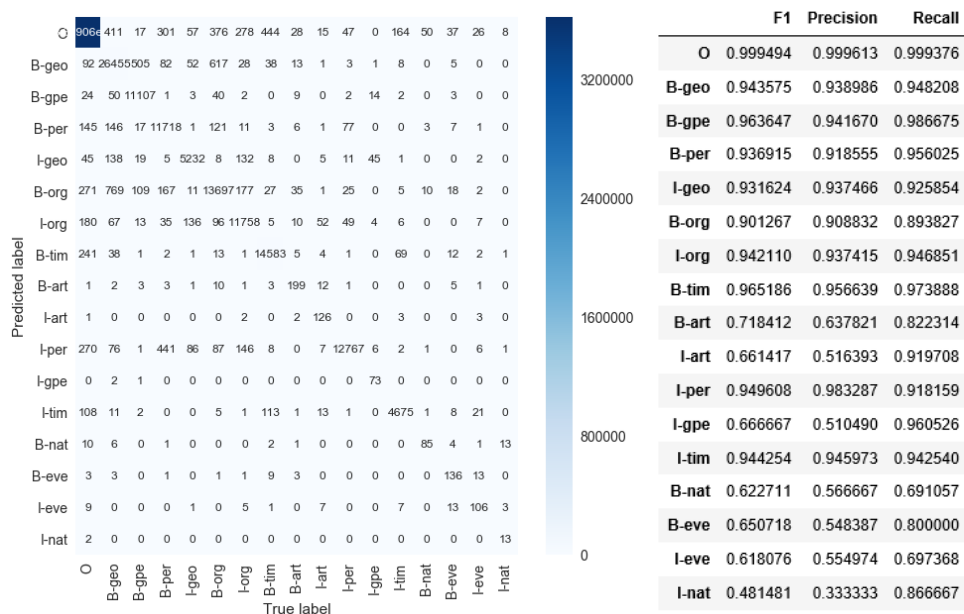
After 2 epochs we see that the model starts to overfit, the learning curves diverge. Because of early stopping, we just do 5 of the 8 epoch. Note that this model contains two LSTM's, training it is computationally expensive. However, we could try different hyperparameters, specially the most important ones, the dimension of the embedding spaces and the number of units of the two LSTM's. The best results were achieved using

128 dimensional embedding spaces, and **100 and 50 units** for the two LSTM's. Here are some examples of predictions for the model, done running the function 'show_predictions3'

Word	True	Pred	Word	True	Pred
Mr.	: B-per	B-per	British	: B-org	B-org
Sharon	: I-per	I-per	Airways	: I-org	I-org
's	: 0	0	canceled	: 0	0
remarks	: 0	0	just	: 0	0
come	: 0	0	19	: 0	0
amid	: 0	0	flights	: 0	0
renewed	: 0	0	from	: 0	0
international	: 0	0	London	: B-geo	B-geo
efforts	: 0	0	's	: 0	0
to	: 0	0	Heathrow	: B-geo	B-geo
restart	: 0	0	Airport	: I-geo	I-geo
the	: 0	0	Thursday	: B-tim	B-tim
Middle	: B-geo	B-geo	.	: 0	0
East	: I-geo	I-geo			
peace	: 0	0			
process	: 0	0			
.	: 0	0			

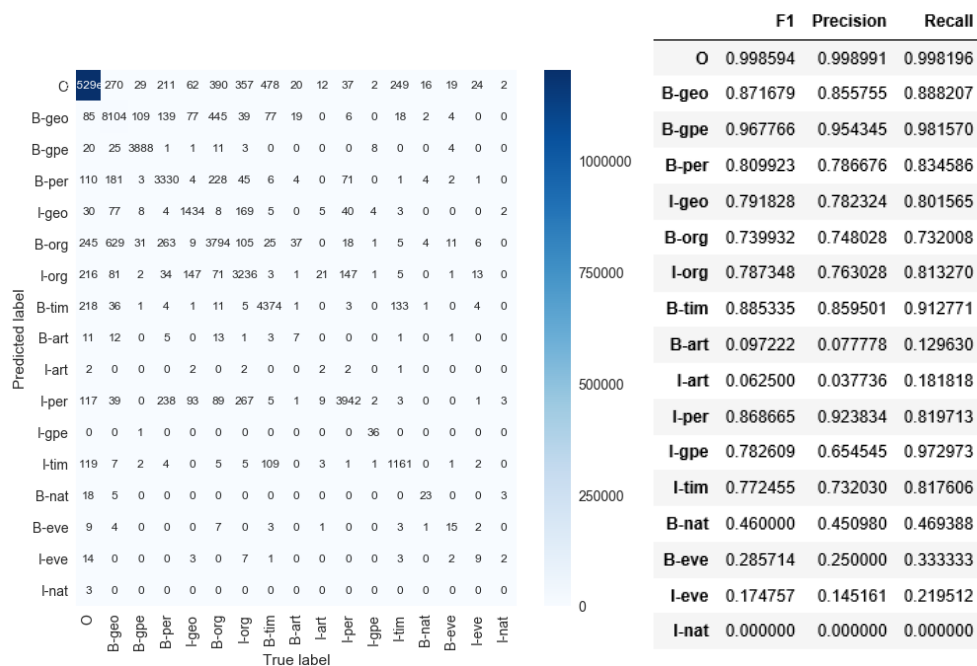
In terms of the single word accuracy it achieves **99.67% accuracy** on the test set and **99.35%** on the train set. But this is unrealistic, since the majority of the words are "PAD" and the model predicts 100% accurate the "PAD" token, as we said before. It is better to check how many sentences are entirely correctly tagged. This model achieves **62.6% accuracy** in this sense in the test set. In the train set, the accuracy is quite higher, a **88%**. So the results are better than first and second models. It is also interesting, to take a look in the F1 Scores for each classes.

The following table shows the results for this model in the train set.



We get an average f1 score of **81.75%**.

For the test set:



This yields an average f1-score of **60.92%**. We see that some tags have really low score such as **art** or **eve**. The confusion matrix may help us understand why this happens and where the classes are confused. We see for example that events and art and confused with organizations constantly. However, the classes are very imbalanced so it is not surprising. We also can see that for almost all tags, there are a lot of null predictions (predicting tag “O”).

This results in terms of f1 score are also the best ones between the three models, both in training and test sets.

Now let's see if including this new information helps when it comes to predict new words. For the 10 test sentences we have the following POS taggings:

```

1 st1 = "The programmers from Barcelona might write a sentence without a spell checker ."
2 pos1 = ["DT", "NNS", "IN", "NNP", "MD", "VB", "DT", "NN", "IN", "DT", "NN", "NN", "."]
3
4 st2 = "The programmers from Barchelona cannot write a sentence without a spell checker ."
5 pos2 = ["DT", "NNS", "IN", "NNP", "MD", "VB", "DT", "NN", "IN", "DT", "NN", "NN", "."]
6
7 st3 = "Jack London went to Parris ."
8 pos3 = ["NNP", "NNP", "VBD", "IN", "NNP", "."]
9
10 st4 = "Jack London went to Paris ."
11 pos4 = ["NNP", "NNP", "VBD", "IN", "NNP", "."]
12
13 st5 = "We never though Microsoft would become such a big company ."
14 pos5 = ["PRP", "RB", "VBD", "NNP", "MD", "VB", "JJ", "DT", "JJ", "NN", "."]
15
16 st6 = "We never though Microsof would become such a big company ."
17 pos6 = ["PRP", "RB", "VBD", "NNP", "MD", "VB", "JJ", "DT", "JJ", "NN", "."]
18
19 st7 = "The president of U.S.A though they could win the war"
20 pos7 = ["DT", "NN", "IN", "NNP", "IN", "PRP", "MD", "VB", "DT", "NN"]
21
22 st8 = "The president of the United States of America though they could win the war"
23 pos8 = ["DT", "NN", "IN", "DT", "NNP", "NNPS", "IN", "NNP", "VBD", "PRP", "MD", "VB", "DT", "NN"]
24
25 st9 = "The king of Saudi Arabia wanted total control ."
26 pos9 = ["DT", "NN", "IN", "NNP", "NNP", "VBD", "JJ", "NN", "."]
27
28 st10 = "Robin does not want to go to Saudi Arabia ."
29 pos10 = ["NNP", "VBZ", "RB", "VB", "IN", "VB", "IN", "NNP", "NNP", "."]

```

From the predictions we can see that including the POS information is helpful for treating unknown words or misspellings. The predictions are as follows.

Word	Prediction	Word	Prediction
The	: 0	The	: 0
programmers	: 0	programmers	: 0
from	: 0	from	: 0
Barcelona	: B-geo	Barchelona	: B-geo
might	: 0	cannot	: 0
write	: 0	write	: 0
a	: 0	a	: 0
sentence	: 0	sentence	: 0
without	: 0	without	: 0
a	: 0	a	: 0
spell	: 0	spell	: 0
checker	: 0	checker	: 0
.	: 0	.	: 0

Word	Prediction	Word	Prediction
Jack	: B-per	Jack	: B-per
London	: I-per	London	: I-per
went	: 0	went	: 0
to	: 0	to	: 0
Parris	: B-geo	Paris	: B-geo
.	: 0	.	: 0

Word	Prediction	Word	Prediction
We	: 0	We	: 0
never	: 0	never	: 0
though	: 0	though	: 0
Microsoft	: B-org	Microsof	: B-per
would	: 0	would	: 0
become	: 0	become	: 0
such	: 0	such	: 0
a	: 0	a	: 0
big	: 0	big	: 0
company	: 0	company	: 0
.	: 0	.	: 0

Word	Prediction	Word	Prediction
The	: 0	The	: 0
president	: 0	president	: 0
of	: 0	of	: 0
U.S.A	: B-org	the	: 0
though	: 0	United	: B-geo
they	: 0	States	: I-geo
could	: 0	of	: 0
win	: 0	America	: I-geo
the	: 0	though	: 0
war	: 0	they	: 0
		could	: 0
		win	: 0
		the	: 0
		war	: 0

Word	Prediction	Word	Prediction
The	: 0	Robin	: B-per
king	: 0	does	: 0
of	: 0	not	: 0
Saudi	: B-per	want	: 0
Arabia	: I-per	to	: 0
wanted	: 0	go	: 0
total	: 0	to	: 0
control	: 0	Saudi	: B-geo
.	: 0	Arabia	: I-geo
		.	: 0

We see that the model still has mistakes. However, at least is able to tag all the words taggable except for Saudi in the sentence *“The king of Saudi Arabia wanted total control.”*. For some reason, it recognizes U.S.A as organization, but predicts correctly United States. With ‘Microsof’ it recognizes it as person instead of organization. But at least it is able to recognize Microsof as an entity. The sentence where Paris is misspelled with Parris is an example of the success of this approach. However, as for the other models it cannot recognize London as a person instead of a location.

GRU Model

Model Description

Also a GRU architecture was tested. GRUs are similar to LSTMs (they are both RNNs) however they do not have a memory unit. Instead, GRUs use two gates - reset and update. These gates are responsible for regulating the flow of information so for example, they can decide whether to pass information about a particular word in a sentence being singular or plural along the chain. The update gate decides what information to delete and add. The reset gate decides how much information for past states is kept.

GRUs also generates less parameters than LSTMs so they tend to be a bit faster to train.

For our architecture two GRU layers of 50 neurons were used each with two dropout layers to prevent overfitting followed by a final Dense layer of 17 neurons. We trained our own embeddings.

The initial training corpus was created by vectorizing all sentences and transforming words into padded vectors of word indexes with fixed length of 140. Our dataset contained 17 distinct labels (tags) so *categorical_crossentropy* was used to take into account all classes.

loss: 0.1064 - acc: 0.9823 - f1_score: 0.9687 - val_loss: 0.0292 - val_acc: 0.9925 - val_f1_score: 0.9925

Our GRU model was trained in 1 epoch with a training loss of 0.1064 against a validation loss of 0.0292. This indicates no overfitting. We achieved an evaluation score of 99.25%.

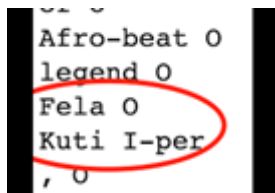
Test-set predictions

Los B-geo	[('Los', 'B-geo'),	Police O
Angeles B-geo	('Angeles', 'I-geo'),	in O
is O	('is', 'O'),	Zimbabwe B-geo
one O	('one', 'O'),	have O
of O	('of', 'O'),	arrested O
the O	('the', 'O'),	a O
world O	('world', 'O'),	nephew O
's O	('s', 'O'),	of O
most O	('most', 'O'),	President B-per
diverse O	('diverse', 'O'),	Robert I-per
cities O	('cities', 'O'),	Mugabe I-per
, O	(' ', 'O'),	on O
and O	('and', 'O'),	suspicion O
a O	('a', 'O'),	of O
summer B-tim	('summer', 'B-tim'),	smuggling O
music O	('music', 'O'),	30 O
series O	('series', 'O'),	tons O
there O	('there', 'O'),	of O
brings O	('brings', 'O'),	scarce O
together O	('together', 'O'),	flour O
the O	('the', 'O'),	to O
city O	('city', 'O'),	neighboring O
's O	('s', 'O'),	Mozambique B-geo
many O	('many', 'O'),	
ethnic O	('ethnic', 'O'),	
communities O	('communities', 'O'),	
. O	('.', 'O')]	

```
[('Police', 'O'),
 ('in', 'O'),
 ('Zimbabwe', 'B-geo'),
 ('have', 'O'),
 ('arrested', 'O'),
 ('a', 'O'),
 ('nephew', 'O'),
 ('of', 'O'),
 ('President', 'B-per'),
 ('Robert', 'I-per'),
 ('Mugabe', 'I-per'),
 ('on', 'O'),
 ('suspicion', 'O'),
 ('of', 'O'),
 ('smuggling', 'O'),
 ('30', 'O'),
 ('tons', 'O'),
 ('of', 'O'),
 ('scarce', 'O'),
 ('flour', 'O'),
 ('to', 'O'),
 ('neighboring', 'O'),
 ('Mozambique', 'B-geo'),
```

Before predicting tags with custom sentences, decoding of a few sentences from our testing dataset were made with mixed results as seen below for the two sentences (note: image sequence is predicted tags vs correct tags).

As we can see our GRU model sometimes fails with detection of IOB tags. It can predict the correct type of tag entity but not whether it was found at the beginning or inside a chunk.



Another example mismatch with a person names occurred on sentence with sentence # is on the image to the right.

Our model detected “Kuti” as belonging to the inside of a chunk but could not predict “Fela” to be the beginning of this same tag (*B-per*)

Phrase predictions

Our GRU model had limitations when a word was not found in the corpus since our training set was based on a corpus of indexes of existent words of the provided dataset. Words like [spell, Barchelona, Parris, Microsof, U.S.A] could not be converted to padded vectors for prediction.

The sentences below were tested successful:

- ☐ "Jack London went to Paris."
- ☐ "We never though Microsoft would become such a big company."

- "The president of the United States of America though they could win the war"
- "The king of Saudi Arabia wanted total control."
- "Robin does not want to go to Saudi Arabia."

The O		
president O		
of O		
the O	We O	
United B-geo	never O	The O
States I-geo	though O	president O
of O	Microsoft O	of O
America B-geo	would O	U.S.A B-geo
though O	become O	though O
they O	such O	they O
could O	a O	could O
win O	big O	win O
the O	company O	the O
war O	. O	war O
Robin B-per	The O	
does O	king O	
not O	of O	
want O	Saudi B-org	Jack B-per
to O	Arabia I-org	London B-gpe
go O	wanted O	went O
to O	total O	to O
Saudi O	control O	Paris B-geo
Arabia I-geo	. O	. O
. O		

Observations

London was detected as a geo-political entity instead of a name (I-per).

Microsoft is an organization (B-org),

Saudi Arabia is a country not an organization.

Saudi should have had the beginning tag (B-geo).

Comparison with Structured Perceptron

From the results, we can see that Recurrent Neural Networks beat Structured Perceptron models with a large margin. LSTM's have a huge complexity compared to those other models that seems to struggle more when facing this kind of data. LSTM's are ideally designed for treating this types of sequences and yield state of the art results in NLP currently.