Assignment1 – Writeup

Dorin Keshales - 313298424

- Describe how you handled unknown words in the HMM.

  First of all, handling unknown words is very important as there is a 99% chance of encountering such words when trying to predict on the dev set and/or on the test set or when trying to face with real-world data which is pretty much infinite. So, here in order to handle unknown words I gathered all the 'rare words' in the training set, when rare words according to my definition are words that appeared only once in the training set (there were a lot of such words), and tried to think and create different kinds of patterns also called word signatures. The purpose standing behind using the word signatures was to avoid using some general 'UNK' symbol for all unknown words found in the dev set and test set by considering some information about the word like its prefix, its suffix, is it a number? Is it a name? and etc. The only case an unknown word found in the dev or test set is replaced with the 'UNK' symbol is when none of the (expressive) word signatures fitted to the word's form.

  It's worth mentioning that in order to not hurt the model scores, I didn't replace rare words found in the training set with their suitable words signatures but added the words signatures as an addition, so even if there is a slight chance for the tagger to encounter the same rare word in the dev set and/or test set it will have a better chance to tag it correctly. Although, I heard a saying that a word appearing only once in the training data with some specific tag might not be a representative case, but I chose to have some faith here and seems it turned out well from current perspective.

  In order for the model to train on the word signatures and the 'UNK' symbol as well, I calculated their e counts. That is, counting for each word signature how many times it appeared on the training set according to the tags associated with the original words answering this pattern.

  I mostly used word signatures with patterns of common suffixes for nouns, verbs and adjectives. And also, more general word signatures like identifying if a word represents an hour, a number, a Name, is all-capitalized, is alpha-numeric and etc.

- Describe your pruning strategy in the Viterbi HMM.

  I use the dictionary that for each word and the tag associated with it that are read from the file, saves this tag as a possible tag for this word. At the end of reading the file I get a dictionary that for each word appeared in the training set and for each word signature, holds all the possible tags that the word can be tagged with according to the different contexts the word appeared in

on the training data. Around this dictionary I was able to build my pruning strategy I used in the greedy Tagger (GreedyTag) and the Viterbi tagger (HMMTag) for getting a reasonable running time. In this pruning strategy, for each word I encounter in the dev set and/or test set, the tagger predicts the most fit tag for the word from the list of possible tags under this word in the dictionary. This way I was able narrow the number of tags I had to go through when calculating which tag is the most probable tag for the word. It's like giving emission probability of 0 to tags that weren't seen with that word in the training set. If we encounter with unknown word then it is replaced with its fit word signature or the 'UNK' symbol if the word form doesn't fit to any of the word signature patterns and as mentioned above each word signature including the 'UNK' symbol have too a list of possible tags it can be tagged with. Therefore, my pruning strategy is valid for all possible cases and was indeed able to shorten the running time to 15 seconds for the Viterbi tagger and less than 5 seconds for the Greedy tagger in the POS dataset. I got even shorter running times for the NER dataset.

- On each dataset, report your test scores when running each tagger (hmm- greedy, hmm-viterbi, feature-based-classifier).
- For the NER dataset, report token accuracy as well as span precision, recall and F1.

Tagger	*POS*	*NER*
HMM-Greedy Tagger	*94.130*	Token accuracy (without 'O'): **73.622** Token accuracy (with 'O'): **94.836** F-Score: **75.518** All-types Prec:76.243 Rec:74.806 MISC Prec:76.717 Rec:71.475 LOC Prec:82.300 Rec:81.001 ORG Prec:57.021 Rec:69.947 PER Prec:89.591 Rec:73.832
HMM-Viterbi Tagger	*96.013*	Token accuracy (without 'O'): **80.141** Token accuracy (with 'O'): **96.267** F-Score: **81.193** All-types Prec:81.782 Rec:80.612

		PER Prec:90.515 Rec:82.899 ORG Prec:65.222 Rec:76.360 MISC Prec:82.311 Rec:75.704 LOC Prec:87.956 Rec:83.886
Feature-based Tagger	96.534	Token accuracy (without 'O'): **85.529** Token accuracy (with 'O'): **97.407** F-Score: **87.357** All-types Prec:88.212 Rec:86.519 MISC Prec:87.781 Rec:80.260 ORG Prec:83.243 Rec:80.760 LOC Prec:91.532 Rec:90.038 PER Prec:88.652 Rec:90.336

- Is there a difference in behavior between the greedy and viterbi taggers? Discuss.

  The greedy tagger works in the form of local decisions that is, for the current location it is in (the last tag it predicted) it calculates the multiplication of the emission and transition probabilities for each tag from the set of possible tags it can predict as the next tag and based on those calculations the greedy tagger makes a local decision in which it predict the tag with the highest probability to be the next tag on the tags sequence. This working method of the greedy tagger cannot guarantee an optimal solution since making local decisions means that subsequent words to have no effect on each decision, so the result is likely to be suboptimal. For example, if in a particular step the greedy tagger predicts the wrong tag as the next tag it's actually causes to further errors as it continues to predict. In other words, this prediction error shifts the greedy tagger from reaching an optimal solution - the best sequence of tags for the sentence. On the other hand, we have the Viterbi tagger which is a dynamic programming algorithm that performs the exhaustive search we would have want in order to find the most likely sequence of tags for a sentence. Unlike the greedy algorithm, the Viterbi algorithm gives an optimal global solution since it postpones decision about any tag until It can be sure it's optimal, but a downside of the Viterbi algorithm is that it requires some bookkeeping (= more computation).

  Since the Greedy tagger doesn't go through all the possible paths in order to find the best one but relies on local decisions it works faster (less than a second for both datasets using pruning) with a time complexity of $O(|t|^2 \times n)$ for trigrams, where t is the number of tags we have in the

dataset. If we use pruning in the greedy algorithm, as I did in the GreedyTag script, then t is an upper bound on the amount of possible tags a word can have. The greedy algorithm's space complexity is $O(1)$, since it only requires the list of already predicted tags (For the model with trigrams only the last 2 tags are enough), a variable to hold the best probability for the next tag and another variable as a reference to the tag with the highest probability. Although the greedy algorithm is fast, it is more likely for it to make errors along the way because of its working methodology which is based on local decisions. However, the Viterbi algorithm makes an exhaustive search and therefore is 3 times slower than the greedy algorithm and has a time complexity of $O(|t|^3 \times n)$ for the trigrams model, where again when using pruning t is an upper bound on the amount of possible tags a word can have. The Viterbi space complexity is $O(|t|^2 \times n)$. But hard work pays off and indeed the Viterbi algorithm has better performance than the greedy algorithm on both POS and NER datasets. For the NER dataset the improvement in performance is quite more significant than in the POS dataset and by using pruning the running time of Viterbi takes up to 15 seconds.

- Is there a difference in behavior between the datasets? Discuss.

(1) The NER dataset we've got has only 8 tags (which are actually 4 and the IOB format makes them 8) where the POS dataset has 45 tags. Since NER dataset has much less tags, therefore the per-token accuracy is very high when considering the 'O' tag. This is because most of the words in the NER data (like 83%) are assigned with the 'O' tag, where in POS data there is no dominant tag, meaning that the data is more balanced when considering how much words in the dataset are associated with every tag.

(2) The NER data also has the need for external knowledge resources and the need for non-local features to leverage the multiple occurrences of named entities in the text.

(3) In English, it is commonplace for what appears to be the same word to belong to two or more parts of speech. It's one of the main challenges in POS tagging and it's called ambiguity. Many words in English can take several possible parts of speech — a similar observation is true for many other languages. Where in named entities data it hardly happens – most of the the words in the data is associated with one tag only, although there is some ambiguity sometimes but minor when comparing to the POS dataset.

(4) In recovering POS tags, it is useful to think of two different sources of information. First, individual words have statistical preferences for their part of speech. Second, the context has an important effect on the part of speech for a word.

(5) As written 2 questions down, different word signatures might be used for named-entity recognition versus POS tagging. Also, I think that the dataset itself can affect the pruning strategy to apply in the tagger if at all since the NER dataset has much less tags as stated in section (1).

- Why are span scores lower than accuracy scores?

  From all of my taggers I got the opposite pattern of results, that is my span scores were higher than the accuracy score. I think it probably happened since maybe when the taggers were wrong they were wrong on all span tokens or on most of them, which means that the model gets penalized multiple times when computing its per-token accuracy, but only once when computing its span scores. In other words, I can assume that when the taggers were wrong it was mostly on tokens that are on the same span. That is, when calculating the span scores only few of the total spans were penalized for this phenomenon while the tokens of all other spans were predicted perfectly by the taggers. To sum up, according to this phenomenon the per-token accuracy will be lower than the span scores. This answer is a hypothesis that attempts to give explanation for why I got the opposite pattern of results and I find this explanation making sense for this phenomenon.

- What would you change in the HMM tagger to improve accuracy on the named entities data?

  For the HMM tagger, I used specific word-signatures which I've defined in MLETrain script in order to deal with unknown words I might encounter in the dev set and test set, but choosing those word-signatures at the time was particularly based on the fact that then my current task was POS tagging. Therefore, most of the word signatures I've defined were particularly designed to find patterns that will help taggers like the greedy tagger and the Viterbi tagger to predict the correct POS tag for rare or unknown words. Meaning that in order to improve HMM tagger's accuracy on the named entities data I would have define a different set of word-signatures based on patterns that can contribute to the task of named entity recognition. As mentioned in Mike Collins' HMM notes: "different mappings might be used for named-entity recognition versus POS tagging".

  Also, I think that using different pruning strategy would give a chance for every word to not be forcibly tagged with the one tag it was associated with in the training set. This could solve some of the ambiguity problem, since we can't know if a word that is ambiguate appeared in the training set with all of the tags that can be fit for it. As a matter of fact, since the NER dataset has much less tags then the POS dataset, pruning might not need here at all. Then, as I said before, more tags will be taken into account when searching for the tag that matches the word. So, in my opinion it may help.

- How did you improve the accuracy of the NER tagger? what did you try? what worked? what didn't work? What does your best NER model include?

  For the basic feature-based tagger, I've used the following features and they were my kick-start features when trying to improve the feature-based tagger scores (per-token accuracy - **85.529**, F1 score - **87.357**) for the NER task: (1) previous two predictions t_{i-1} and t_{i-2} (2) current word x_i (3) prefixes and suffixes of x_i (4) next two words x_{i+1} and x_{i+2}.

  First thing I've tried was some of the recommended features for the named entity recognition task such as Lexical Features like capitalization (all-capitalized, is-capitalized), numbers (all-digits,

alphanu-meric, fractions, decimal numbers), punctuation (is-punctuation-mark). I've tried to use the base tagger's features with all the new features and permutations of some of them, but I've found that all those features just made my per-token accuracy and F1-score worse than the score I've began with in the base feature-based tagger. Later, I've found it's making sense, since all of those features were helpful in tagging the 'O' data, but our per-token accuracy and span-score measure metrics doesn't consider the 'O' data when computing the per-token accuracy and F1 score. Therefore, the new features could only overshadow the other features and harm the scores.

Then, with that understanding I've tried a different approach of using gazetteers focusing helping the tagger to better its scores on the PER, LOC and ORG tags and therefore for an overall improvement in both per-token accuracy and F1 score. I used some of these lexicons as follows:

(1) For Person – 'firstname.5k', 'firstname.1000', 'lastname.5000', 'people.family_name' and 'people.person.lastnames' lexicons.
(2) For Location – 'location', 'location.country' and 'venues' lexicons.
(3) For Organization – 'venture_capital.venture_funded_company', 'automotive.make', 'business.brand' and 'business.sponsor' lexicons.

Those lexicons indeed improved both per-token accuracy (85.259) and F1 score (87.27) , but I decided to also try and adapt the viterbi code to search over the classifier's score for each tag, rather than over the product of q and e estimates and basically to implement an MHMM model. Using the MHMM model I got per-token accuracy (87.165) and F1 score (87.759) when used the all lexicons mentioned above, but even better F1 score when I used only the Person and Location lexicons - per-token accuracy (87.165) and F1 score (87.776). I was about to finish trying to improve, but then I suddenly had the idea to use a word POS tag as a feature. In order to do it, I used my Viterbi tagger to tag NER's training set, dev set and test set with POS tags. Adding this feature to my tagger brought a slight improvement of 0.1 in the F1 score when using the Person lexicons only, so I let it be and got final F1 score of **87.853** and per-token accuracy of **87.185** on the dev set. Although the improvement was relatively small, it was worth trying just for the experience ☺.

In conclusion, my best NER model include the following features: (1) previous two predictions t_{i-1} and t_{i-2} (2) current word x_i (3) prefixes and suffixes of x_i (4) next two words x_{i+1} and x_{i+2}, (5) POS tag of current word (6) Person lexicons.

- For the 5 bonus points:

I managed to write the FeaturesTagger.py script efficiently such that when predicting on the dev set and wring those predictions to a file takes at most 15 seconds for POS dataset and only a few seconds for NER dataset.
I will explain how I shortened the runtime from 24 minutes to few seconds by using reference to some values in the POS dev set (for convenience):

The whole idea behind this implementation technique was to reduce the number of calls to the predict function of sklearn Logistic Regression classifier. The obvious way to predict tag for each word in the dataset while consulting the previous tag predictions is by going over the data - sentence by sentence - and predict tag for one word at a time. I started with this kind of implementation of the Feature tagger, which made me wait around 24 minutes (on the dev set of POS) for the program to finish its run. After a while I've noticed that the predict operation which is done thousands of times during running causes the runtime to lengthen.
The Solution – I've restricted the number of times the predict function is being called to be at most as the length of the longest sentence on the data set (for example in POS the longest sentence has 118 words). In order to do it, I set up a loop of i from 0 to 118 (not including 118) and in each iteration I simply extract features for all the words in position i in every sentence in the dataset (or to be exact - every sentence participating in the current iteration). Then I use the DictVectorizer to convert the features I've extracted from all of those words to features vectors, one for every word, and then at the end of the current iteration I call the predict function *once* for predicting the tags for all words in position i. In order to keep track on the sentences in which I should keep predicting tags for (weren't over yet), before every iteration I make a list with all the sentences indexes such that their length >= i+1, meaning there is still words left in those sentences which their tag wasn't predicted yet.

This way with every iteration (from 0 to 117) there are less and less sentences left to predict on. Therefore, we get a massive reduction in the number of predict calls we need to perform and also in every iteration we predict for smaller number of words.

As I mentioned before, by doing that instead of calling the predict function for every single word when going over the data sentence by sentence which sums up to tens of thousands of calls (as the number of words in the data), I managed to limit the number of calls to the predict function to be the length of the longest sentence in the data set.