Abhijit Kulkarni
CS 190I HW 4

# Named Entity Recognition Report

I worked with no one else on this.

I programmed this Named Entity Recognition (NER) project in Python. I decided to use a Hidden Markov Model implementation, all from scratch. This means that I didn't use any CRF libraries or special libraries other than the standard ones such as NumPy, random, etc.

Here, let me describe how the HMM implementation works for NER. We are given a bunch of states or tags that are associated with the individual words, from either the Spanish file or Twitter file. The amazing thing about HMM is that we aren't limited by the type of terms; the same code can be used to predict for either one, it is only necessary to change the states. I did some pre-processing for the Spanish file since I started it earlier; for example, I truncated similar rooted words that only differed by a tense into one term. I also noticed anything that had a number in it in training/validation was always a "O" term, so I paid attention to that as well. However, I was not able to pre-process the Twitter file because of time constraints, so hopefully my model was able to generate the correct tags.

HMM is a Markov model that is generative. The model assigns a joint probability to a pair of observation and labels. The three parameters of HMM are then trained to maximize the joint probability over the testing set. However, this shows the reason HMM isn't perfect: in testing we want to sequence label, meaning that we want to have maximum performance of predicting label given the observation. But HMM learns the joint probability of observations and labels instead, which is why perhaps CRF may be better for the NER task. However, I wanted to implement HMM myself in order to understand it better and also just for fun.

HMM's three parameters are the start probabilities, the transition probability matrix, and the emission probability matrix. The start probability matrix is only a 1x|S|, where S is the total number of states. This is predicting the probability that each sentence is starting with a particular tag. Thus in order to train it, all that is needed is to look at the tag for the first word in each sentence. Next, we have the transition probability matrix. For every pair of tags $T_x$ and $T_y$, there is a probability that $T_y$ occurs right after $T_j$. Its dimensions are |S| x |S|. Training this is similarly easy. Our last parameter is the emission probability, which represents, for every word, the probability matrix of that word occurring as every tag. This means that its dimensions are |V| x |S|.

Now that we have created all the parameters, I implemented the Viterbi algorithm in order to test. The first problem we have to deal with is words that are unknown, or out of the vocabulary of the training set. At first I was not sure what to do; after some thought, I went with this option. During training, every new word had a 1% probability of being labeled as UNKNOWN. This way, I was able to get a distribution of unknown words and what their parameters would look like that I was able to use on the testing set. This allowed me to get pretty good accuracy for OOV words.

For me, training was very short while testing took a significant amount of time since the Viterbi algorithm takes quite a while. For Spanish data, it took 1.5 seconds to train and about 5.5 minutes to test over all the testing set. Whereas for Twitter, it took 0.33 seconds to train and 33 seconds to test. This is because the Twitter testing dataset was very small. Interestingly, my HMM performed better in comparison to others for Twitter rather than Spanish. This points out to me that the HMM performs decently at labelling with less data than a CRF does, since I assume that most people implemented CRF's.

I know that this is not a full implementation of HMM – a true implementation would include tuning hyper-parameters using the validation dataset, with the Baum-Welch algorithm. This algorithm computes the forwards and backwards probabilities and uses them to update the three parameters. I never actually used the validation set in my implementation, so I definitely could have done this. However I started working on it and was met with some impedance, and due to time restraints i.e. studying for the final, I was not able to fully implement the Baum-Welch EM algorithm. However, I am sure that my accuracies would have increased had I had time to implement this. Even so, I still achieved about 70% F1 for the Spanish set, and placed about halfway through my peers for the Twitter set. Had I had time to implement the B-W algorithm, training would have gone on for much longer than the testing time. This is because it needs to compute the forward matrix, the backward matrix, and then use them to update all the parameters. Computing the forward/backwards matrices takes about as much computing power as the Viterbi algorithm so it is safe to say that training would have taken anywhere from 2-3x as much time as testing did if B-W was implemented.

All in all, I learned a lot through implementing my own HMM, and just wish I had a little extra time so that I could see how much implementing the B-W algorithm would have increased my F1/ overall score.