NLP Project 2: Kaggle Team “lite lite”

Nithish Raja Chidambaram (nr293), Akash Peri (ap659), Vinayaka Suryanarayana Bommathanahalli (vb247)

Implementation Details:

For this task, we chose to use HMMs. HMMs are good at modelling how known information relates to unknown information. If animal walks like a duck, quacks like a duck, and swims like a duck, then it is likely a duck. For us, HMMs are good at modelling sequences as well. This is because once we train an HMM, it contains the various hidden states, transition probabilities between those states, and an initial probability matrix. We can easily determine the probability of a sequence happening, and also track partial sequence probabilities as well (in case we want to classify "University of British Columbia" by the probability for "University of"). The disadvantage to HMMs however, and especially the one we are implementing (as opposed to MEMM) is that we are severely limited in the number of additional features we can consider (for instance, we can't directly use the POS tags in our HMM). In addition, we can't capture higher dimensional relationships, like we might be able to with kernelized SVM, to find trends based on augmenting the data with functions. This is related to the previous point, where an SVM can easily be extended to contain a new feature for Proper noun POS tags where the word starts with a capital for instance.

When implementing HMM, we started by reading in the training data. We then split the data into sentences, at which point we were able to tokenize the data. With this, we then build counts of the words we’d seen. An important note is that we decided not to use unknown word recognition, but we did implement +k smoothing. We then build our generation and transition probabilities from the counts.

Originally, we hadn’t bothered with a validation data set, but as seen in our analysis, we ended up allocating 20% of our data for that.

We decided on contrasting CRF with our HMM as the extension. With our HMM, we were able to make use of the conditional independence assumption, the first order Markov assumption. Conditional independence allows us to assume that two states will be independent given the conditional probability distribution. That is, we assumed that the state transition probability was only dependent on the previous state, and not a function of all previous and future states. This significantly reduces the probability matrices we need to maintain, as we can just build state transition probabilities for the 9 states we have.

However, by assuming conditional independence, we drop a lot of context. For instance, if the current paragraph talk about food, it is likely that the next few sentences will reference food as well, which can shift our probabilities. Practically, every NE assignment we do should incorporate the NE assignments of nearby states.

We chose to use a linear chain CRF, by implementing CRFSuite as per the tutorial listed below, to evaluate it's performance, and determine how having slightly more context than HMM would benefit CRF. That is, for every word state we have, we consider the previous and the next words for context to classify it's position. Another important part of setting up CRF is choosing how to convert features to probabilities. The features you include and the probabilities assigned to them have a significant impact on the final outcome. Note that we also chose to evaluate the test input on a sentence by sentence basis. This was because our original HMM implementation also evaluated on a sentence basis, and we wanted to compare our hopeful improvement to the baseline.

Here were the features we chose to add:

For all words (current and previous):

whether the word is all upper case

whether the word is a title

whether the word is a digit

POS tag

Truncated POS tag

Beginning of sentence

End of sentence

For current word:

word in lower case

First two characters of the word (prefix)

Last two characters of the word (suffix)

For all words, we considered some basic features, such as whether the word was all upper case, or whether it was capitalized. Both of these are indictive of Named Entities, and the all upper case one in particular could be for organizations. We also considered the POS tag, as there is a strong correlation with certain POS tags such as NNP, and NEs.

We used the sklearn suite to implement CRF. There were several parameters we tested locally before choosing the best combination. The first was the algorithm. The sklearn suite performs tuning internally, and we felt the Limited Memory BFGS (lbfgs) algorithm would perform better over others like Stochastic Gradient Descent with L2 regularization (l2sgd) or perceptron (ap). Using l2sgd would have speeded up our code (it makes orders of magnitude fewer iterations when compared to lbfgs), but would have produced a sub-optimal solution when compared to lbfgs. Since the dataset we were operating on was quite small, we decided to opt for the more accurate algorithm. If instead of just Stochastic Gradient Descent, we had the option of Adagrad (covered in 4780), we would have chosen that, as it has been shown to greatly improve the robustness of SGD. There were also two other optimizations techniques, Averaged Perceptron and Passive Aggresive that we decided to try.

We also set the coefficient for both L1 and L2 regularization to 0.1, a number we found performed reasonably well through trial and error. We set the maximum number of iterations to 100, as we noticed that the rate of improvement in the result drastically shrunk at around this point. Finally, we also set all\_possible\_transitions to true. This flag ensured that if the training data did not contain transition probabilities for some states (due to no data), CRF would automatically generate some (strongly negative) weights for these transitions.

After that, we also tried out the PyStruct library. Instead of using a linear chained CRF, the library used an SVM with CRF. Hoefel (2008) suggests this is to take advantage of the max-margin classifier property of SVM in the hopes of finding the best assignment, as opposed to using just likelihood maximization approaches. With the combination, we can find the most likely labels for a word, and have that feed into the CRF to have it assign labels based on the surrounding words. We expect this to perform better than just the linear chain CRF used above.

Pre-Processing

We decided to try the algorithm as is, without any significant pre-processing, with the intent to add them if necessary. This was also because we noticed the data was actually in quite good shape. After obtaining our results, we did see that they were adequate, so we decided to not add any preprocessing.

The basic preprocessing we did was to break up the input into words, POS tags and NER tags if training, or token count if test.

#make sure you mention we want to keep punctuation in

#not heavily optimizing k for smoothing

#not handling unknown words; if it only occurs once, treat as UNK

Experiments:

As part of this, we swiftly built our HMM. This was a standard implementation, summing up all the counts for the various states to build generation probabilities, and their previous neighbour to build transition probabilities. We had a few parameters we needed to tune, such as our k value for +k smoothing. To determine this, we ran through several values manually, from 0.0001 to 1 before settling on a value close to 0.2 as the optimal k for the dataset. We didn’t have code to do this dynamically because we were primarily working off our validation data set.

We expected to get around 70% precision with this implementation, as we figured we'd capture the majority of the tokens. Originally, we had not split our training data set, and we were shocked to see that our Kaggle score was only 0.11.

After this, we split our training data set into an 80/20 split so we would have some to evaluate on. We also had 14% locally. It took us a few days of debugging, but we were able to isolate the problem to the way we'd implemented the back pointers for the Viterbi algorithms. The back pointers were one less than the total number of states, and we had implemented it with an off-by-one error, which had shifted all our results over. After fixing that, we had a precision of 73% locally and 67% for HMM on Kaggle.

At this point, we decided to try with CRF. We knew that CRFs were found to be better for NER tasks than HMMs because they could take in more context, so we expected a moderate increase over our previous run. We first setup CRFSuite as described above, and found that the precision was now 85%. This is slightly low for a Named Entity recognition system that should be above 90% precision, seeing as the majority of words in normal discourse are not NEs. However, the training data set had a higher than normal ratio for NEs, explaining why our initial precision may have been lower than expected.

Results:

Table 1: Summary of the Precision, Recall and F1 calculated

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Local | | | Kaggle (if available) |
| Approach: | Precision | Recall: | F1 |  |
| HMM before fix: | 14% | - |  | 0.11 |
| HMM after fix: | 43% | 67% | 0.52 | 0.67 |
| CRFSuite l2sgd: | 84% | 83% | 0.83 | 0.85 |
| CRFSuite lbgfs: | 84% | 83% | 0.83 | 0.86 |
| CRFSuite ap: | 84% | 83% | 0.83 | - |
| CRFSuite pa: | 84% | 83% | 0.83 | 0.84 |
| CRFSuite arow: | 81% | 80% | 0.8 | - |

Table 2: Broken down results for HMM, and 2 of the CRF Suite runs off validation data set:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | HMM | | | CRFSuite - lbfgs | | | CRFSuite - pa | | |
|  | Correct | Incorrect | Precision | Correct | Incorrect | Precision | Correct | Incorrect | Precision |
| B |  |  |  |  |  |  |  |  |  |
| PER | 698 | 33 | 95.49 | 1401 | 219 | 86.48 | 1426 | 229 | 86.16 |
| LOC | 995 | 61 | 94.22 | 1339 | 195 | 87.29 | 1336 | 193 | 87.38 |
| ORG | 366 | 111 | 76.73 | 719 | 250 | 74.2 | 718 | 245 | 74.56 |
| MISC | 424 | 885 | 32.39 | 499 | 91 | 84.58 | 513 | 109 | 82.48 |
| I |  |  |  |  |  |  |  |  |  |
| PER | 784 | 110 | 87.7 | 1184 | 121 | 90.73 | 1170 | 111 | 91.33 |
| LOC | 133 | 235 | 36.14 | 137 | 40 | 77.4 | 137 | 41 | 76.97 |
| ORG | 420 | 403 | 51.03 | 435 | 129 | 77.13 | 430 | 125 | 77.48 |
| MISC | 231 | 3594 | 6.04 | 166 | 36 | 82.18 | 169 | 49 | 77.52 |
| O | 26270 | 251 | 99.05 | 31460 | 383 | 98.8 | 31442 | 361 | 98.86 |

Above, we have the results when we run HMM on our validation data set. We noticed that for B, while PER and LOC were labelled quite well, ORG was at around 80%, and MISC was at only 33%. For I, PER was the only one we were able to label well, with I-MISC having a significant number of incorrectly classified words. This was quite a bit different from what we were expecting, which was an even distribution off correct and incorrect NER tags amongst the different states.

When we analyzed why this was the case, we realized that a lot of the incorrectly classified counts for MISC were actually other B tags. We suspected this was because PER had the narrowest range of possible values, and situations where it may appear in. Thus, our unoptimized HMM was able to identify the majority of them without difficulty. MISC, on the other hand, had the greatest possible range amongst the NE tags. This appeared to be because it was misclassifying many other B/I tags as B/I-MISC.

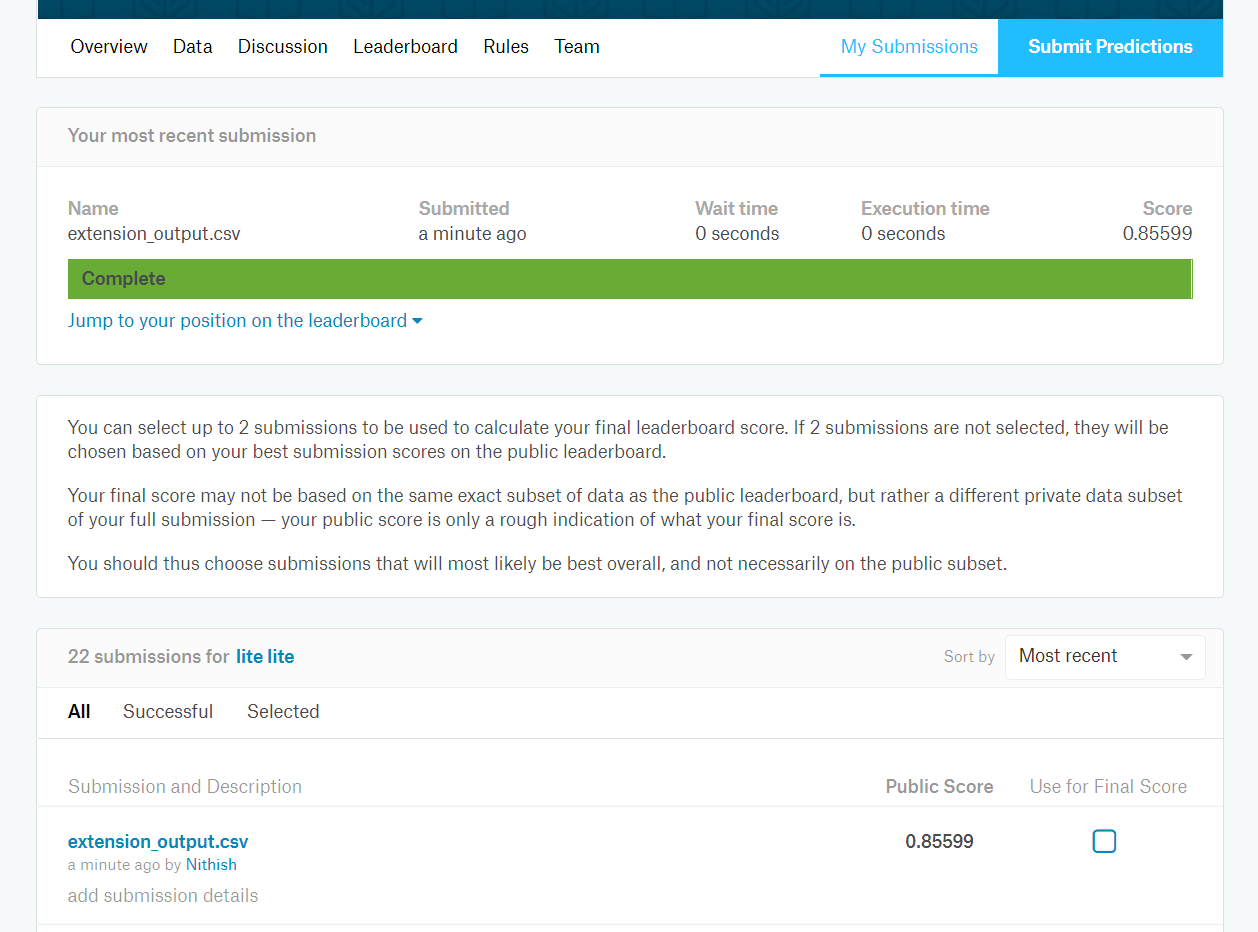
As B/I-MISC had the greatest range, this made sense, as it was the most likely to have one-count words. Indeed, a quick scan of the incorrectly classified words were one-count words such as 'Kofi'. One way to improve this would be to implement some sort of UNK handling like we had in Project 1. That is, we would substitute all 1-count words with UNK (or UNK-PER, UNK-LOC, etc), allowing us to then include that in our probability distribution.

We did find it incredibly strange that we had so many mis-classified I's. If anything, we were expecting it to go the other way, as I's should only follow B's or other I's. We suspect that the latter part was not correctly terminating, and we were having long chains of I's following other I's where they shouldn't have.

Comparing this to the HMM output, it looks much better. We still classify B-PER and I-PER very effectively, but now we no longer have any significant outliers unlike for the HMM. This is likely because CRF is a sequence classifier. It assigns NE tags based on the surrounding assignments, and is much more effective for this kind of task than the HMM, which only took in one feature, or did not have any of the processing built into CRFSuite. This is reflected in the F1 score. The F1 score for HMM is quite poor, and gets much better once we switch to CRFSuite.

Looking at the individual CRFSuite precision, we did not see any significant difference. We also tested the other four algorithms available from CRFSuite, but did not break the results down for them, because they all had similar F1 values.

Kaggle score:



Acknowledgements/References:

We used the tutorial below to implement CRFSuite.

https://sklearn-crfsuite.readthedocs.io/en/latest/tutorial.html

G. Hoefel and C. Elkan. Learning a two-stage SVM/CRF sequence classifier. In Proceedings of the ACM Conference on Information and Knowledge

Management, pages 271–278, 2008.