Project 2

Team Members: Benjamin Steeper (bds238); John Trujillo (jpt86); Elaine Hwang (yh467)

Kaggle Team Name: JEB

1. Baseline

Implementation:

For this part of our assignment we decided to follow a similar path to that of what was suggested in Section 3.4 of the handout. That is, a baseline system where we build a lexicon of all named entities that appear in the training corpus. And then, identify during testing, only those named entities that we saw as part of our lexicon. More intrinsically, however, our lexicon followed a 2-dimensional dictionary data structure, where the key is the word, and the value is another dictionary, of tag to frequency key-value relation. Ultimately giving us a data structure as seen below

$$\{w_1 : \{t_1 : freq_1 \ldots t_k : freq_k\} \ldots w_n : \{t_n : freq\ ;\ tn\}\}$$

If there exists a named entity that was seen on the test lexicon in our 2-dimensional dictionary then we return the named entity's tag for which frequency is maximized.

Experiments:

Hypothesis: For a model that is based on the maximum likelihood of a tag based on frequency, given a word, we would expect to only get roughly half of the NE tags correct, given that our baseline model should also do at least better than even.

Methodology: We decided to use 85% of our training data as part of our lexicon, and tested it with the remaining 15%, which we made as our validation test. From this, after we had satisfactory results, we tested this with our test set.

Expectation: Given that our hypothesis makes it difficult to discretely assign an estimate, we predict that our model can provide roughly 55% precision.

Results & Error Analysis:

Our model returned rather positively unexpected results, and competed very well with our HMM but more on this in Section 2. Our baseline model returned a precision-score of .9402, a recall-score of .6409, and an F1-score of .76 . This percentage of accuracy was most definitely unexpected as some Piazza posts had their baseline model averaging 50%, but for us, our Kaggle submission had us at roughly 64%. A tabularized form of this PRF can be seen in section 4 *Comparing the Three Models.*

2. HMM

Implementation:

In this HMM we defined our hidden-states as our *tags*, and our observations as our *words,* and our transition-emission probabilities as the probability of current tag, given that we know what the previous tag is. More formally, $P(t_i \mid t_{i-1})$. Given our framework, we implemented the Viterbi Algorithm, a dynamic programming approach to calculating the most optimal path sequence given a transmission for each possible path. To reduce it to our implementation, we have that our path will be the output sequence of tags, and observations are the words.

Our transition probability is stored in a 2D dictionary, in the format **`transProb[prevTag][currTag]`**. Our emission probability data is structured the same way, with **`emProb[tag][word]`** as our emission probability. Before calculating our emission probabilities, we first replaced words that only appeared in the training corpus once as "**`<unk>`**". Then, we used add *k*-smoothing for the emission probabilities. After smoothing, we used backoff, as was suggested in the Jurafsky & Martin textbook, to produce our final emission probability dictionary.

Experiment:

Hypothesis: HMMs have shown a to be great predictor, but are limited in their window, averaging out in the 60-70 percentile.

Methodology: We worked on calculating the estimated best *k* for our *k*-smoothing, through trial-and-error. We trained on this for our HMM using 85% of our text, and then used the other 15%, that is, the remaining percentage, of our text as validation text, just like for our baseline model. We were also interested in seeing how well our HMM implementation paired up with that of SKLearns's. More information on this can be seen in the Results section below.

Expectations: If our hypothesis is true, we believe that our model can provide a ~65% accuracy in prediction.

Results:

Our results for HMM were encouraging, but they were definitely not uplifting. We marginally did better than our baseline, that is by 1.4%, giving us 65.4% on Kaggle. This very well

Due to Academic Integrity we did not want to submit the SKLearn HMM result to Kaggle to compare accuracy, so we decided to do it locally with the test set, and compare with ours. Simply, the SKLearn HMM well outperformed our HMM implementation by well over an average of 13% through precision-score, recall-score, and F1-score. A tabularized PRF error analysis can be seen in Section 4 *Comparing the Three Models.*

3. MEMM

Implementation:

For the implementation of Maximum Entropy Markov Models, we decided on roughly 179 features, although they fall into 6 general categories. More on this in Feature Justification. Given that each feature must have a weight, we used the SKLearn library to help us in determining the weight quantity for each feature. More specifically, we use the Linear Model library, and used the `fit` function, which trains on the features and word to tags, and the `.coef_` function, which returns a weight matrix of size TokenSize × FeatureSize. After this, the rest is rather straightforward, and we followed a similar approach to that of our HMM Viterbi algorithm to output the optimized sequence.

Feature Justification:

POS Tag : `isProperNoun()`

If the word was a noun, we wanted to add it as a distinctive feature to further increase the likelihood of it being an NE. That is we identified if the word was NNP(S).

Caps: `allCapsLTE3()`

We noticed, that for tokens that had all caps and had less than or equal to 4 words, there was very high probability that it was an organization. The reason we restrict the length to less than or equal to 4 is to rule out sentences in all caps, which should not return a sequence of non-O tags.

Capitalization: `isCapital()`

By definition, an NE must be capitalized per the rules of English. However, not all capitalized word is an NE. *An* example is the first word of this sentence, which is clearly not an NE, unless the POS tag said it was an NNP(S). So we built a feature function that identified it as follows, where *w* is the string.

```
isCapital = True if w[0].isupper() and tag[w] is NNP
```

PreviousTag: `prevTag()`

We aimed at giving our classifier a windows of at least the previous word given that its part of what defines our HMM, and to give this notion of a 'window' classification.

NextTag: `nextTag()`

Similar to our `prevTag()`, we wanted to have phrase clustering to increase the information the the SKLearn fitting algorithm has to improve weight class.

TagTransition: `tagTrans()`

The second part of the HMM's, our goal was to build on top of the HMM, that is, get the best of HMMs while including an abundance of other features.

Experiments:

It is worth mentioning that for all our experiments dealing with our validation set, we used an 85:15 ratio, where 85% of the text is to train, and the remaining is the validation.

Experiment 1:

- **Hypothesis:** Adding our baseline, given its success, should improve our estimation, as well as the integration of other minor features, like capitalization
- **Methodology:** We added features such as Capitalization, All Caps, current POS tag, probability of current word being tagged as each BIO tag, etc.
- **Expectation:** We think adding basic features that only concerns the current word and current POS tag would give us a slightly better result compared to our HMM model, which is around 65%

Experiment 2:

- **Hypothesis:** Adding Baseline features into our MEMM models should help increase our accuracy
- **Methodology**: We implemented additional features for each BIO tag, and if the current word's highest occurring tag is the BIO tag, we indicate it with 1 and 0 otherwise.
- **Expectation**: We expected our F score to go up by 0.1

Experiment 3:

- **Hypothesis**: Since we're only outputting the categories of the BIO tags, we wanted to see if our performance would be better if we only classified it with 'PER', 'ORG', 'LOC', 'MISC', and 'O'
- **Methodology**: During preprocessing, we categorized the POS tags as above.
- **Expectation**: We expected our F score to reach 0.7

Experiment 4:

- **Hypothesis**: We think that adding features that concerns previous POS tags and words, as well as next POS tags and words would improve our model
- **Methodology**: We added features that indicated the previous and next POS tags, as well as running the baseline features on our previous and next words.
- **Expectation**: We expected our F score to reach 0.8

Experiment 5:

Test hyper-parameter of *k* for smoothing. Throughout multiple attempts with our validation, we tested *k* from 0.1 to 1. And after multiple trials, we ended up having an optimal *k* value of .87.

Results:

Experiment (implementing very basic features) did not allow our MEMM model to reach a higher F score than our HMM model. Experiment 2's features was the one that really boosted our MEMM accuracy. For Experiment 3, our PRF scores were much lower compared to what we had before, so we decided not to implement it. Experiment 4's features were also really helpful, and allowed our MEMM model to reach a F score of 0.73 on our development set.

On Kaggle, we only got a 0.64 score for our MEMM, which is not very encouraging. However, we were able to achieve a .73 F score on our development set after updating some of our features, but missed the deadline for the kaggle competition.

4. Comparison of the Three Models:

Baseline v HMM:

From the data, it is patently clear that there was no significant advantage, in implementing HMM over the baseline, rather the baseline outperformed our expectations and was only slightly surpassed by the HMM.

It was very interesting to see how accurate we could get our baseline model with such a trivial approach. We evaluated this model, and its comparison by comparing it to the development set, and its PRF values. It's also surprising to see that including these emission probabilities.

We evaluated this model, and its comparison by comparing it to the development set, and its PRF values, of which can be seen in the Error Analysis of this section.

HMM v MEMM:

The compared results can be more closely seen below, but to summarize our MEMM outperformed our HMM by roughly 6.5%.

When we printed out the predicted tags, we noticed that HMM were better at getting the correct I- tags compared to MEMM. We suspect that this is because HMM does not consider as many other factors, resulting in more weight on the transition probabilities.

However, as we added more and more features (that were relevant) to our MEMM model, the performance was much better compared to HMM. We learned from this experience that it is crucial for MEMM models to take in enough features for it to outperform HMM and baseline systems.

Error Analysis:

To better visualize and compare between the models, we attach a table below.

	Baseline	**HMM**	**MEMM**
Precision	**.918**	**.921**	**.946**
Recall	**.539**	**.546**	**.596**
F1-Score	**.679**	**.685**	**.727**

The errors that occurred:

Initially, we collected all token tags, BIO tags and POS tags into one long list for processing. However, we soon found that this technique resulted in far too many O's. The longer the list of tokens, the higher the probability of O's.

All tags were either I-PER or O at one point, with about a 50/50 ratio (definitely not correct)

We think this happened because:

The number kept increasing due to the compounding probabilities of O -> O tags occurring over time. We did not have enough features at the time. We only had 13 features at this point.

How we solved the problem:

To solve this problem, we parsed the tokens by sentence and created a nested list of sentences. Then we called our viterbi function on each sentence one at a time in sequence. We added many features to include 179 total features (from our initial 13).

Improving this system:

After analyzing the SKLearn Linear Model, we noticed that it's `predict()` function didn't use the Viterbi algorithm, and rather than running in O(sn_2), it runs in O(*nlogn)* which would definitely be more efficient. But most interesting of all is that it produces much more accurate predictions than that of ours. This in turn, would provide a much more accurate model for us.

5. Individual Member Contribution:

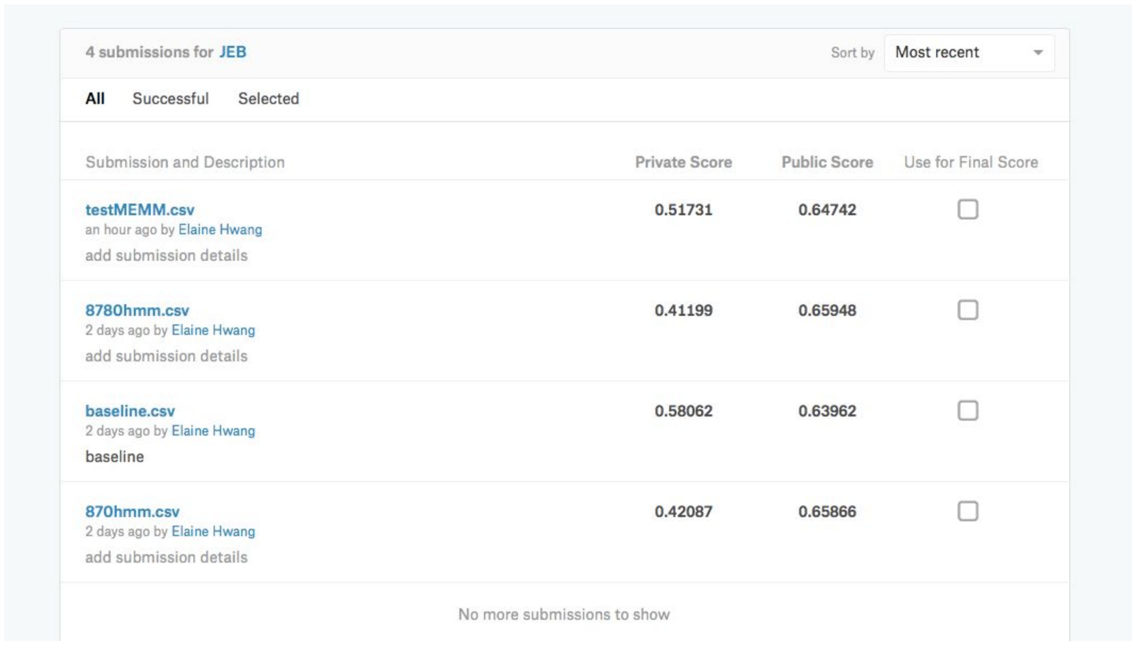
Ben: Worked on preprocessing, implementing feature testing and extraction functions, calculating P.R.F, as well as the csv output function.

John: Developed the baseline model, helped work out the details for viterbi, coded part of the training probability function for MEMM, worked on the SKLearn part of the MEMM implementation, conceptual feature extraction implementation, as well as working with SKlearn's HMM function. Worked heavily on the write up.

Elaine: Wrote the Viterbi function for HMM and MEMM, helped sort out the details of the MEMM formula and tried different experiments on the features. Also worked on SKLearn and implementing feature extraction.

6. Kaggle Submission:

Screenshot:



4 submissions for JEB

Sort by: Most recent

All | Successful | Selected

Submission and Description	Private Score	Public Score	Use for Final Score
testMEMM.csv an hour ago by Elaine Hwang add submission details	0.51731	0.64742	☐
8780hmm.csv 2 days ago by Elaine Hwang add submission details	0.41199	0.65948	☐
baseline.csv 2 days ago by Elaine Hwang baseline	0.58062	0.63962	☐
870hmm.csv 2 days ago by Elaine Hwang add submission details	0.42087	0.65866	☐

No more submissions to show

(We didn't check the *Use for Final Score* box before the 8PM deadline, but all these submissions were before the deadline)

Team Name: JEB

Score:

Baseline :	63.962%
HMM :	65.948%
MEMM :	64.742%