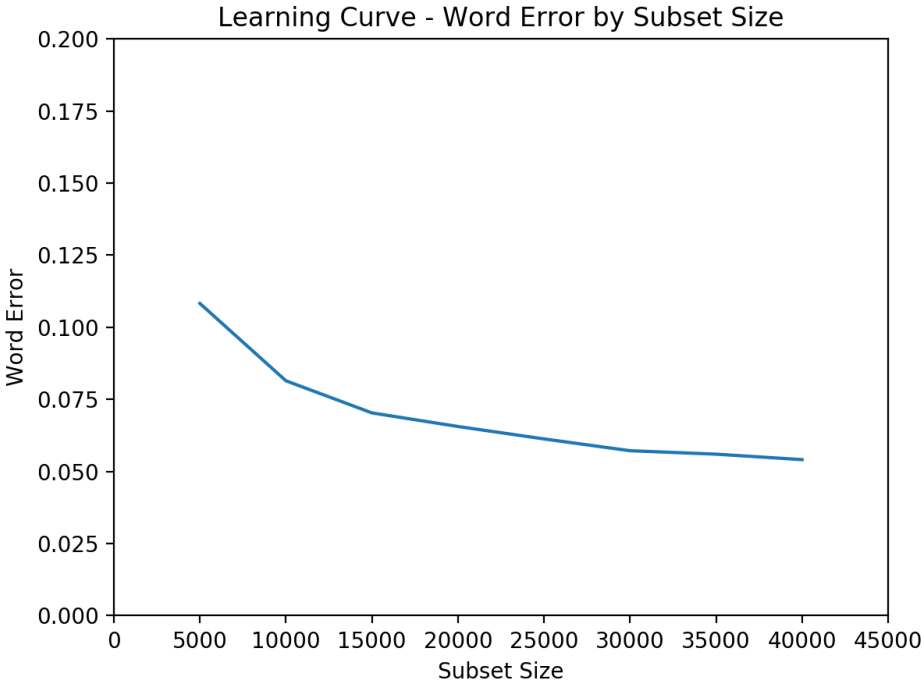
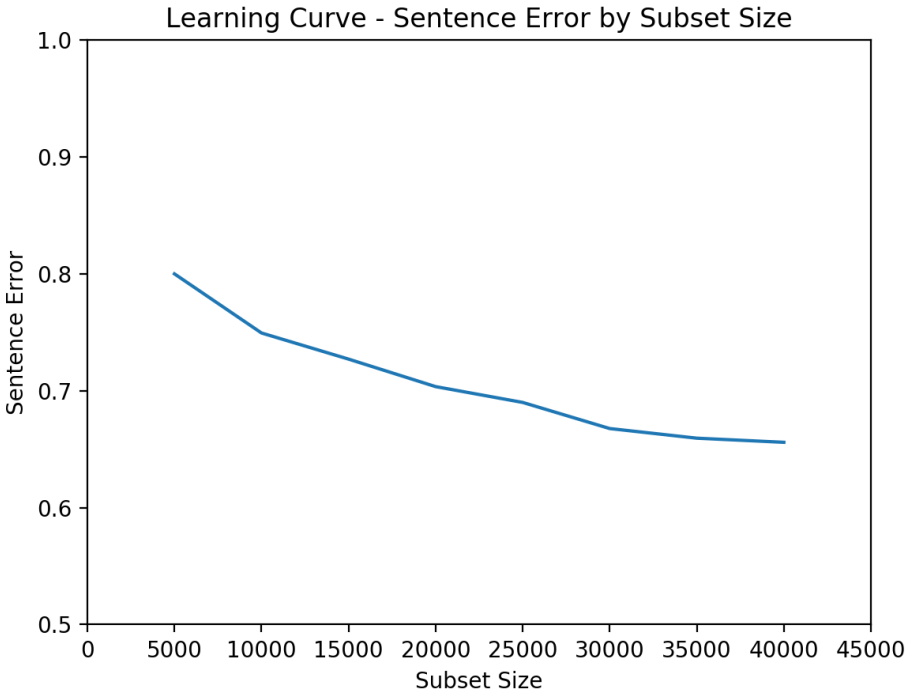
ECS189G POS Tagging, Student: Hariharan Sezhiyan, ID: 913644281

Task 1:

Larger datasets clearly had a positive impact on accuracy, reducing error rate in words and sentences across the board. For each increase in subset size, a strict reduction in error rate could be seen. The reason is obvious: with more data, the model is more easily able to determine correlations in the language. In particular, the state transition probabilities and emission probabilities are more fine-tuned.





Task 2:

Performance:

Performance on ptb.23.txt → word error: **0.0508762**, sentence error: **0.6223529**

Method:

I implemented a trigram HMM and wrote my own viterbi.py to account for trigram HMM. Within my train_hmm.py file, I calculated transition probabilities for unigram, bigram, and trigram. For trigram, I determine number of times every unique trigram appeared (in other words, a count: C (tag1, tag2, tag3)). I then divided this by the number of unique counts corresponding to the first two tags (C (tag1, tag2)) to get the transition probability for this particular trigram. Bigram transition was calculated as: $\frac{C(tag_1,tag_2)}{C(tag_1}$, and unigram calculated as $\frac{C(tag_1)}{N}$, where N is the total number of tokens in the training corpus. To further improve performance, I used deleted interpolation to deal with data scarcity. In the case of trigrams, data scarcity becomes an important issue because the probability of three tags occurring together and in order is rare, considering there are 47 total tags. For reference, there are $47^3 = 103823$ possible tag orderings. As a result, some trigrams may not occur in the training set or occur in the training set with a very lower probability, but should still be included in the model. For this reason, every time an n-gram has a relatively high probability, its corresponding lambda is incremented. This lambda can be thought of a weightage, providing more important to n-grams better representing a particular tag sequence.

To produce the emissions, I simply counted the number of times a particular token occurred alongside a tag and divided it by the number of times a tag appeared altogether. I performed no special smoothing here.

The crux of my code was my implementation of viterbi.py. At the top of the viterbi.py file, I pull emission and transmission data from the .hmm file, and store it in two separate dictionaries (emissions and transmissions). I then used the tags in my emissions dictionary to fill in K, a python set containing all possible tags. I simply iterated through emissions and added every tag into K not previously in it. K_{-1} and K_0 are initialized to Set[“init”] because the represents the set of all tags sentence start can transition from (only “init”). I also use the emissions dictionary to fill a vocabulary set. As before, I iterated through and added words not already in the vocabulary.

The main for loop of my code iterates through all lines in the testing corpus, and performs POS tagging on individual words within each sentence. I first initialize the policy table to 0 (log(1) = 0). Then, for every word in the sentence, we “construct” a trigram tagging sequence, with all possible of tags v, u, and w from K_{k-2}, K_{k-1}, and K_k, respectively. The score of each new tagging sequence is based on the sum of its transmission probability, emission probability, and the policy value based on the previous iteration. Because viterbi invokes a policy value based on the previous iteration, this is a dynamic programming approach.

Task 3:

Performance

Performance: jv.test.txt → word error: **0.0462266**, sentence error: **0.210155**
Performance: btb.test.txt → word error: **0.113414**, sentence error: **0.763819**

Performance Difference:

The performance metrics clearly show the data works best on the Japanese language and worst on the Bulgarian language. I noticed in Japanese that a few words and phrases have high repetition rate. For example, the Japanese word hai, which means "yes" in English, is repeated a total of 191 in the test text file and 3000 times in the train text file, and almost every single time is the only word in the sentence. Therefore, when the trigram sees this word in the test set, it certainly tags it correct as "ITJ", and since the word occurs often, it is able to correctly tag a lot of times.

The reason for the Bulgarian's poor error rate is less clear. One possibility is that the Bulgarian language has 3 genders (male, female, and neuter), and verbs and adjectives need to follow the gender of the noun. This requires the model to learn gender-based features instead of using simple transitions, which might be hard with this training corpus size. Furthermore, verbs in the language are extremely complex, and can take as many as 3000 different forms depending on mood, gender, subject, etc. The training set, though quite large compared to English and having 12,000 lines, is still not able to predict all transitions. Verbs are an important feature of this language and occur often, and my model (and bigram) rarely correctly tag them.

It's important to note that my model performs better than the baseline model for word error in both Japanese and Bulgarian (0.0628 and 0.1159, respectively), but worse on sentence error for both Japanese and Bulgarian (0.1368 and 0.7512, respectively). For Japanese, this makes sense because many sentences have length in the range 1 to 4. For these sentences, a trigram contains a lot of "noise" which might weaken error rates. For Bulgarian, the trigram might do better with words because a few sequences of 3 words repeat often. For example, the tag sequences "A NC Punct" and "R Nc R" occur 34 and 118 times in the test set, respectively. This is a lot considering the test size. This indicates the trigram works well with groups of three and tags words in those sequences well, but sentence error is worse because no such pattern exists across a full sentence. Furthermore, almost every single sentence in the btb.test.txt file was composed of 4 or more words, which by pure probability means lower chance of tagging it all correctly.

Bonus Task:

Script:

Run script as → python score_script.py test_file.out base_file.tgs

Results:

Input: ptb.22.out, ptb.22.tgs. Output: The trigram score is: 0.8611
Input: jv.out, jv.test.tgs. Output: The trigram score is: 0.8549
Input: btb.out, btb.tgs. Output: The trigram score is: 0.7035

Method:

The script file iterates through the output files and test.tgs files for each language, and counts the total number of trigram tags, which in this case is every set of three consecutive words. It then counts the number of trigrams that agreed exactly between the two output files. I simply divided these two values.

As expected, the Bulgarian files performed worst, considering their poor performance in the other metrics. Curiously enough, English did better on this metric than did Japanese. This indicates the model tags individual words in Japanese better, but perform worse when trying to tag groups of words.

This metric makes sense because my approach was trigram hmm, and so it's intuitive to judge the model by how well it correctly tags every trigram. Its validity is proven by the fact that it agrees with other metrics.

Consultation:

1) Songwen Su – 913888348 (another student in ECS189G). We helped each other understand the concepts and formulas, but in no way wrote each other's code.