CS5012 – Language and Computation
Practical 1 – POS tagging with HMM
Student number: 180019199

1 - Deliverables achieved and instructions:

Viterbi and Beam algorithms:

- A working beam search algorithm that generalises for any value of K (and therefore works for Viterbi where K = total number of states). Viterbi accuracy is c. 94-95%.

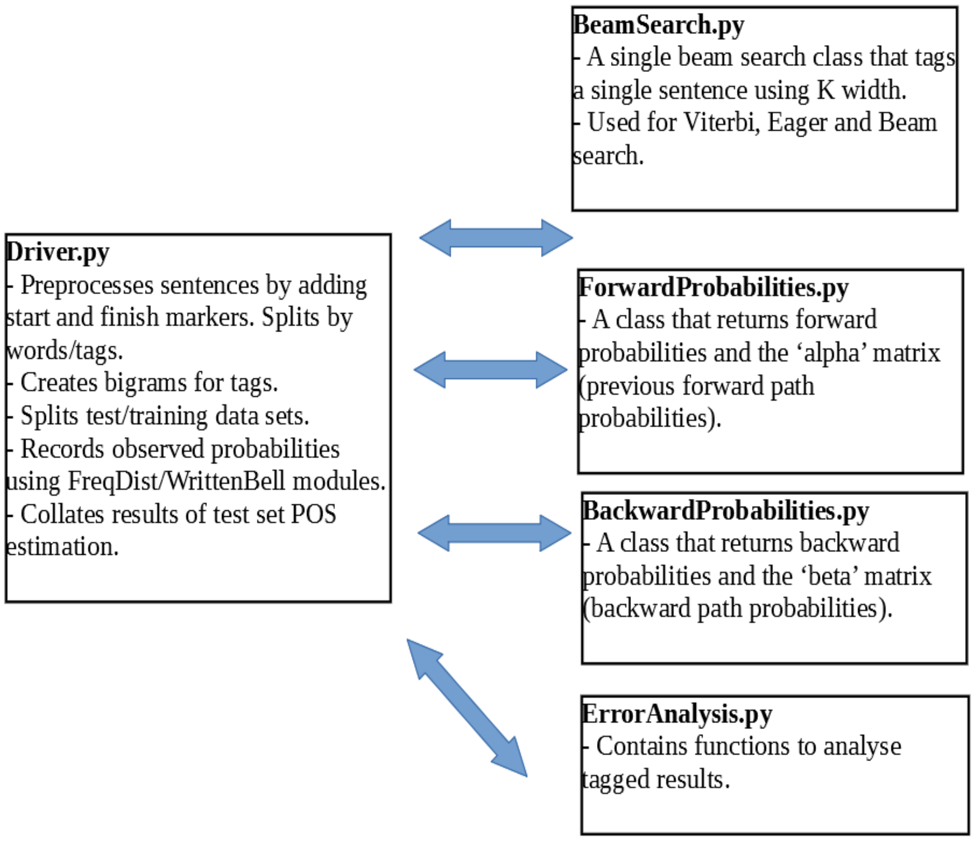
Forward and Backward algorithm (‘FWBW’):

- A working FWBW algorithm with accuracy of 94-95%.

Empirical analysis discussing the impact of various beam widths, and how accuracy is affected by sentence properties (such as length, diversity of tags).

Running instructions: Copy all files to root directory. Running command ‘python Driver.py’, will then provide the output for every algorithm used for the Section 3 results.

2 – Overall Design:
While the code is extensively commented, the program flow is described below.



BeamSearch.py
- A single beam search class that tags a single sentence using K width.
- Used for Viterbi, Eager and Beam search.

Driver.py
- Preprocesses sentences by adding start and finish markers. Splits by words/tags.
- Creates bigrams for tags.
- Splits test/training data sets.
- Records observed probabilities using FreqDist/WrittenBell modules.
- Collates results of test set POS estimation.

ForwardProbabilities.py
- A class that returns forward probabilities and the ‘alpha’ matrix (previous forward path probabilities).

BackwardProbabilities.py
- A class that returns backward probabilities and the ‘beta’ matrix (backward path probabilities).

ErrorAnalysis.py
- Contains functions to analyse tagged results.

Reader notes:

- *When reviewing the code, I recommend reading the BeamSearch class first, then the Forward class, and finally the Backward class. This is because I wrote less detailed comments as I implemented the subsequent classes that had overlapping concepts (in order to avoid repetition).*
- *In the results output, ‘beam_search_all’ corresponds to a beam search with all states (i.e. the Viterbi algorithm).*

Key Design Considerations:

- **Use of log probabilities:** To avoid underflow from multiplying many probability values together (resulting in very small values), the sum of the natural log of probabilities was used. Because in this excercise we are only interested in the ranking of possible tags (and not the probabilities themselves), I did not convert them back to normal probabilities.
- **Case of tokens**: I initially considered converting all the tokens into lower case, so that strings like 'Church' and 'church' would not be treated separately, and improve the P(W|S) values. Eventually I reconsidered this approach because it would result in information loss, and it could make it difficult for the tagger to distinguish common nouns and proper nouns like 'apple', which, if case is not used, could be ambiguous (i.e. between the fruit and the company).
- **Viterbi and Beam Search:** To create Viterbi, I simply followed the steps described in Jurafsky & Martin (Section 5.3.3). For Beam Search, I created an extra method *'find_top_k_tags'* which would order the states according to prior state Viterbi matrix values, and I would then limit the subsequent state matrix column updates to this ordered list.
- **Forward-Backward:** First, we observe that the formula provided for algorithm 3 is comprised of the highest combined forward probability and backward probability up to token 'i'. Therefore, my approach was to (1) generate the alpha and beta matrices from the Forward and Backward probabilities respectively for all observed tokens; and then (2) use these matrices in a separate function to find the highest probability tag for a given token.
  1. As per the P1 specification, the Forward and Backward classes are very similar to the BeamSearch class, but instead of finding a single, most-likely path, we are computing the sum of all possible paths to a particular state for a particular token observation. My general approach was implementing the formulae described within Jurafsky & Martin (Section 6.3 and 6.5). Particular care needed to be taken for the start and end components of each algorithm, which differ slightly. Each observed sequence of tokens will output it's own alpha (forward) and beta (backward) matrix that represent the sum of all sequential POS paths to a token position.
  2. In *Driver.py,* the function *'forward_and_backward_combined'* then cycles through each token, and finds the maximum log probability sum of a alpha and beta matrix for that state (as described in equation 6.42 of Jurafsky & Martin). This most likely tag, is then allocated as the best POS guess for that token.

3 – Evaluation:

Test conditions:

- Per the specification guidance the training set was 10,000 sentences from Brown Corpus using the universal tag set. The test set was a further 500 (different) sentences. Sentence size was capped at 100 tokens per guidance (resultant from data-sparsity caused by long sequences with low-probability).
- Again, per guidance, WittenBell smoothing was used to handle unseen 'current tag'/word and 'current tag'/'prior tag' combinations.

Results:
Under the above conditions the following results were achieved.

Algorithm	**Accuracy on test set**
Viterbi (i.e. Beam with K = 12)	94.99%
Beam Search, K = 3	94.87%
Eager Search (i.e. Beam with K = 1)	93.26%
Forward-Backward	94.33%

Comment:

- These results are mostly what we expect. Viterbi chooses the optimal path from all 12 possible tags, so its underlying probability path matrix should better represent all the possible tag paths a sentence can take.
- Eager and beam perform relatively worse because they check fewer paths from state-to-state. That said, I was quite surprised how well the Beam Search with K=3 performed given that it only checks the top three paths per token. To further understand the trade-offs between choosing beam width, I performed several experiments in section 4.
- Lastly, the Forward-Backward did not perform as well as I expected. I tested a variety of training and test samples, and while sometimes it would be the best-performer, it did not do this consistently. Because it sometimes had lower accuracy, it leads me to believe I may could have made an error in my implementation.

4 – Further Empirical analysis:

Per the previous page, I performed some further testing with various beam widths. Below, 'N' refers to the number of unique tags (which is 12 because the universal tagset is used). Given that the Viterbi approach is the same as K=12, I no longer distinguish between Beam and Viterbi going forward.

Hypothesis 1: Gains in accuracy uplift from higher K values declines as the training set size increases:
In machine learning, research suggests[1] the most important factor for accuracy is training size rather than model type. As such, I believe increasing the K value becomes less important as training size increases.

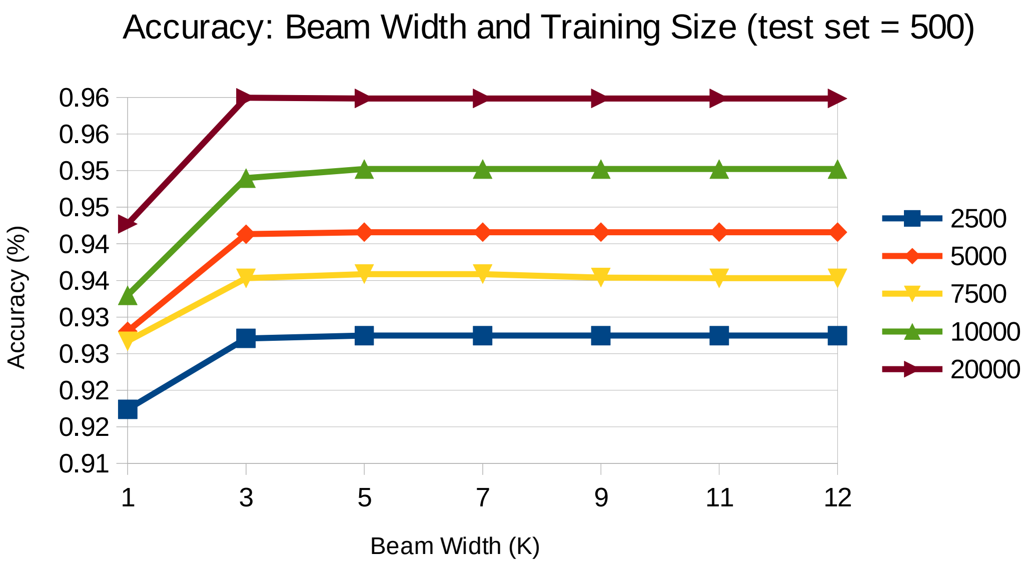
Hypothesis 2: Following Hypothesis 1, as K increases from 1, there will be a value of K smaller than N which provides best time-to-accuracy trade-off.
I expect that as we increase K, that the increase in accuracy will grow at a much lower rate than the increase in processing time. Therefore, there will be a 'best-beam-width' that provides a 'middle-ground' compromise between extra time computing path probabilities and accuracy uplift.

Hypothesis 3: Larger K values are more accurate for longer sentences
I expect that longer sentences are 'harder' for the smaller K models to predict. This is because as we move token-by-token through a sentence, the path probability matrix is updated with the previous values. Smaller K values are more likely to make errors, and so for longer sentences, these errors will propagate further along through the matrix, and as such, I expect them to perform materially worse.

Hypothesis 4: Larger K values are more accurate for sentences that contain more 'diverse' tags
Here 'tag diversity' is defined as the number of unique tag elements in a test sentence (as measured by the gold standard). I expect that lower K value models are worse at correctly tagging a sentence that jumps between many POS states, relative to higher K models.

[1]Refer "Scaling to Very Very Large Corpora for Natural Language Disambiguation", Banko, M. and Brill, E. (2001) and "The Unreasonable Effectiveness of Data", Halevy, A., Norvig, P. and Pereira, F (2010)

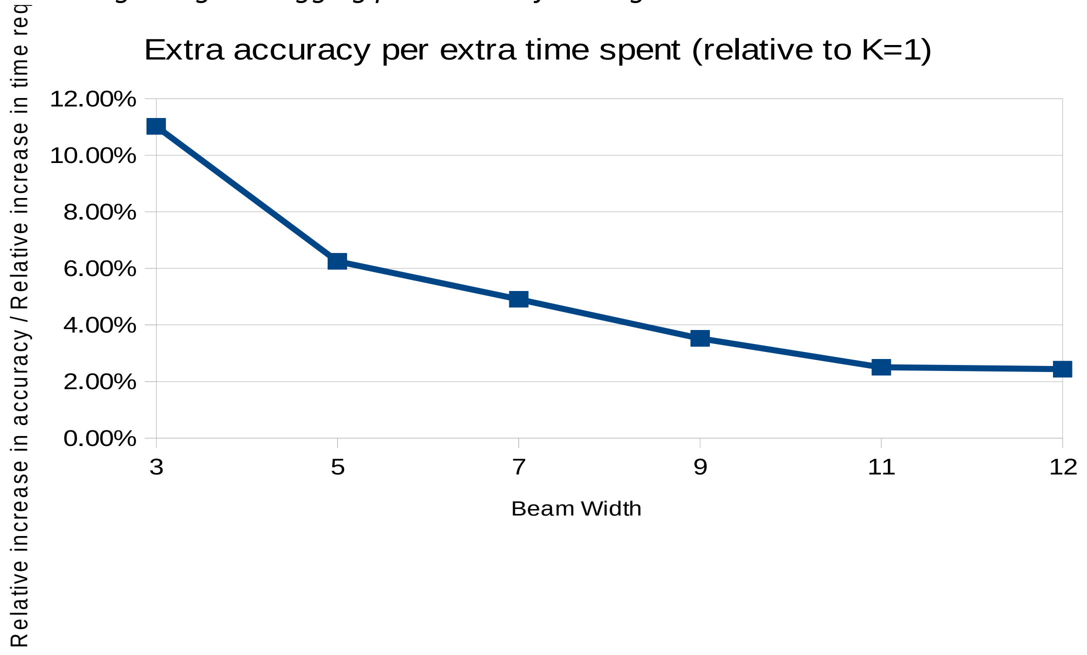
Results:
The graph below plots beam width accuracy against various training set sizes. We see that after some material accuracy uplift from K=1 to K=3, that beam width does not matter as much, and that training data size is far more important. This supports Hypothesis 1.



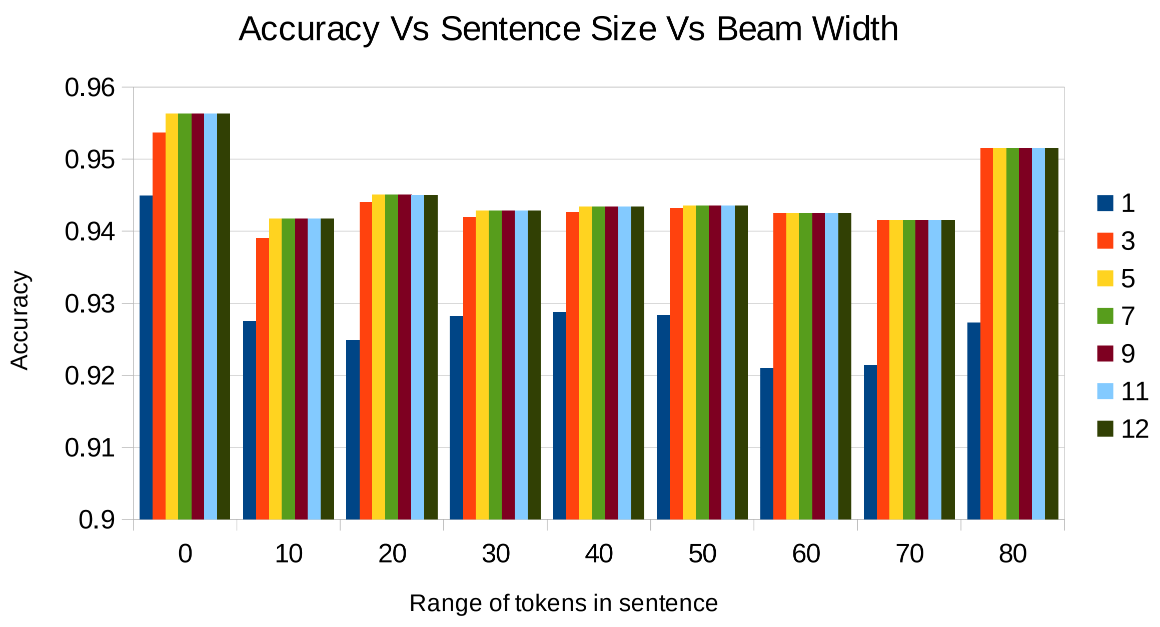
The graph below shows the diminishing returns from increasing width from K=1 as an approach to improve tagging accuracy.

The width of K=3 shows the best trade-off in terms of accuracy-to-cost, which can be interpreted as ‘a 100% increase in the time tagging the unlabelled data results in an 11% increase in accuracy’. As we increase K, we spend a lot more time computing path probabilities in the tagging process, but gain much less in terms of accuracy uplift. This supports hypothesis 2.

Reader Notes: The vertical axis label is difficult to read, but it says “(% increase in accuracy) / (% increase in time required)”. Also, the ‘time’ measurement refers to the time (in secs) required in between beginning the tagging process and finishing it.



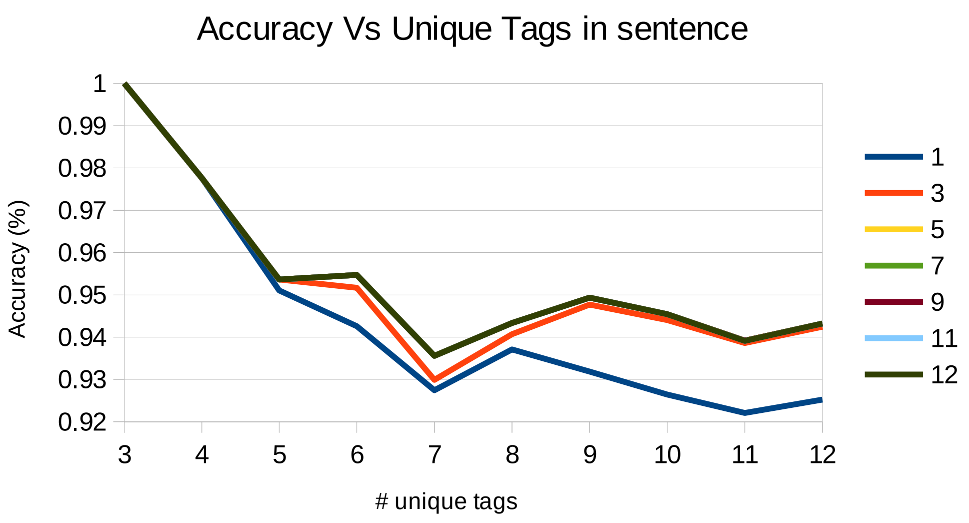
The graph below shows the accuracy performance for different sentence sizes (incremental buckets of size 10), across increasing beam widths. For K=1, the accuracy drops quickly as sentence size increases, and does not recover. Contrary to my hypothesis however, there appears no real relationship between beam width performance and sentence size for K >= 3. Unexpectedly, the longest sentences (80-90 tokens), actually perform better than all other buckets except the 0-10 bucket. I was quite surprised by this, and cannot explain it. If I was to test this effect further, I would repeat the experiment on different test sets and corpora.



Lastly, the graph below shows that:

1. Sentences with more diverse tags have lower accuracy; and
2. Lower beam widths (different lines) are impacted more by this effect, but from K=5 the results are indistinguishable.

Perhaps the first observation is a result of diverse sentences naturally having lower observed frequencies, and therefore less accurate probability estimates. These result support hypothesis 4.



Overall Conclusion / Thoughts:

I was quite surprised how well the models with K >= 3 performed relative to the Viterbi approach (or K=12). If I was going to build and implement my own HMM tagging program that required fast tagging, I would first accumulate as much training data as possible, and then only use a lower width model (such as K=3) to tag the unlabelled data because the extra cost that Vitberi requires gives marginal benefit in terms of accuracy.