1. **Deriving the Viterbi Algorithm**

Solution

1.1 *Part - I*

$$P(t|w) = \frac{P(t,w)}{P(w)} \tag{1}$$

$$P(t|w)\ \alpha\ P(t,w) \tag{2}$$

Taking log on both sides,

$$log(P(t|w))\ \alpha\ log(P(t,w)) \tag{3}$$

Since log is a monotonic function the best *'t'* which maximizes $P(t|w)$ will always be the best *'t'* which maximizes $log(P(t|w))$.

Hence,

$$argmax\ P(t|w) = argmax\ log(P(t|w)) \tag{4}$$

1.1 *Part - II*

We have

$$\pi_j(t_j) = argmax_{t_1 \ldots t_{n+1}} \sum_{i=1}^{j} score(w, i, t_i, t_{i-1}) \tag{5}$$

RHS of (5) can be written as:

$$argmax_{t_1 \ldots t_{n+1}}[(\sum_{i=1}^{j-1} score(w, i-1, t_{i-1}, t_{i-2}) + score(w, i, t_i, t_{i-1})] \tag{6}$$

$$argmax_{t_1 \ldots t_{n+1}}[(\sum_{i=1}^{j-1} score(w, i-1, t_{i-1}, t_{i-2}) + score(w, i, t_i, t_{i-1})] \tag{7}$$

Simplifying (6) further we have,

$$argmax_{t_1 \ldots t_{n+1}}[(\sum_{i=1}^{j-1} score(w, i-1, t_{i-1}, t_{i-2}) + score(w, i, t_i, t_{i-1})] \tag{8}$$

Hence $\pi_j(t_j)$ can be written as a compound of π_{j-1} state.
Q.E.D

1.1 *Part - III*

If there are K tags and M position in the sequence, then there are MxK viterbi variables to compute. Computing each variable requires finding a maximum over K possible predecessor tags. The total time complexity of populating the trellis is therefore $O(MK^2)$ with an additional factor for the number of active features at each position. After completing the trellis, we simply trace the backwards pointers to the beginning of the sequence, which takes O(M) operations.

1.1 *Part - IV*

In this problem, we will be exploring the relation between $\pi_j(t_j)$ and π_{j-1} states like we did in problem 1.2. But this time, we'll be working on semiring verison of Viterbi algorithm. According to which, we have:

$$\hat{t} = \bigoplus_{t_1 \ldots t_{n+1}} \bigotimes_{i=1}^{j} score(w, i, t_i, t_{i-1}) \tag{9}$$

Now, for every value of t_j, we have:

$$\pi_j(t_j) = \bigoplus_{t_1 \ldots t_{n+1}} \bigotimes_{i=1}^{j} score(w, i, t_i, t_{i-1}) \tag{10}$$

We know that the score function has dependencies on the given word, its tag, the previous word's tag. Hence using this information, the above equation can now be written as:

$$\pi_j(t_j) = \bigoplus_{t_1 \ldots t_{n+1}} \left(\left(\bigotimes_{i=1}^{j-1} score(w, i-1, t_{i-1}, t_{i-2}) \right) \otimes score(w, i, t_{j-1}, t_{j-2}) \right) \tag{11}$$

Applying the distributive properties of semiring ($(x \oplus y) \otimes z = (x \otimes z) \oplus (y \otimes z)$), we can write the above equation as:

$$\pi_j(t_j) = \bigoplus_{t_1 \ldots t_{n+1}} \left(\left(\bigoplus_{t_1 \ldots t_{n+1}} \bigotimes_{i=1}^{j-1} score(w, i-1, t_{i-1}, t_{i-2}) \right) \otimes score(w, i, t_{j-1}, t_{j-2}) \right) \tag{12}$$

Simplifying (12) further,

$$\pi_j(t_j) = \bigoplus_{t_1 \ldots t_{n+1}} \left(\pi_{j-1}(t_{j-1}) \otimes score(w, i, t_{j-1}, t_{j-2}) \right) \tag{13}$$

Hence $\pi_j(t_j)$ can be written as a compound of π_{j-1}.

Unlike problem 1.2, where we were working with argmax scores, and expanding through basic addition, this time we used the properties of semirings to prove the relation between $\pi_j(t_j)$ and its previous state π_{j-1}.
Q.E.D

2. Programming Hidden Markov Model

Solution

2.1 *Introduction*

A statistical model called a Hidden Markov Model (HMM) is employed in applications of natural language processing. It can be used to explain how external, indirectly observable causes influence the evolution of observable occurrences. We can forecast a series of unknown variables using this class of probabilistic graphical models by using a collection of observed variables. In this problen, we implement a hidden markov model tagger (HMMTagger) by estimating parameters from a training set using maximum a posteriori (MAP) estimation. Along with this we add α-smoothing in the model class to tweak the tagger.

2.2 *Implementation*

We were provided with the Penn Treebank dataset along with a starter code to commence with the HMM based POS tagger. After rendering the dataset and getting pandas based dataframe, we create a model class for the tagger and create functions such as *'get_transitions'* for getting transition paths, *'get_emissions'* for calculating emission probability, *'create_transition_table'*, *'create_emission_table'* for getting the table of transition and emission probabilities. The complete implementation of the model can be found in the python notebook attached with this report in the submission.

In the next question, we will try to run the training by and get accuracy metrics for baseline tagger and HMM tagger using viterbi decoding.

3. **Programming Viterbi Decoding**

Solution

In this step, we first start by implementing a baseline tagger, assigning highest tag frequencies for each word. After training the model and running predictions for dev and test set, we came up with the classification reports which can be found in the code.

Classification report for the baseline models on dev and test are given in figure 1 and 2 respectively.

accuracy			0.91	148157
macro avg	0.65	0.62	0.63	148157
weighted avg	0.92	0.91	0.91	148157

Figure 1: Classification report for baseline on dev set

accuracy			0.92	171138
macro avg	0.72	0.69	0.69	171138
weighted avg	0.92	0.92	0.92	171138

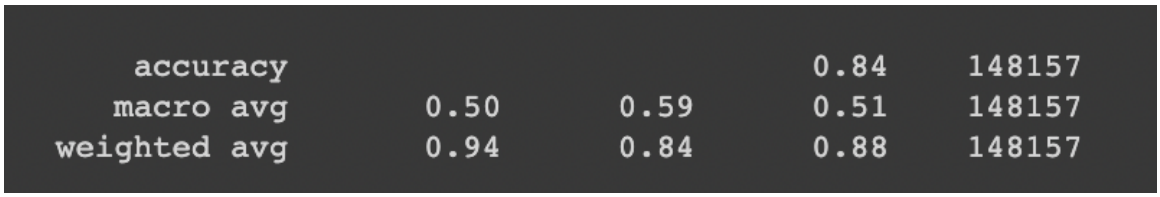
Figure 2: Classification report for baseline on test set

After running the baseline tagger, we try to add *'viterbi_decode'* function in our HMMTagger to implement the Viterbi algorithm. As asked, we have also kept α =1 for additive smoothing.

Classification report for the HMM models with viterbi decoding and α =1 on dev and test set are given in figure 3 and 4 respectively.

accuracy			0.85	171138
macro avg	0.51	0.59	0.51	171138
weighted avg	0.94	0.85	0.89	171138

Figure 3: Classification report for HMM on dev set



accuracy			0.84	148157
macro avg	0.50	0.59	0.51	148157
weighted avg	0.94	0.84	0.88	148157

Figure 4: Classification report for HMM on test set

Alpha	Macro P	Macro R	Macro F1
1	0.50	0.59	0.51
0.01	0.48	0.52	0.49
0.0001	0.62	0.63	0.62
0.00001	**0.66**	**0.67**	**0.66**
0.00005	0.66	0.67	0.66

Table 1: Metric scores with various values of α

4. **Evaluation**

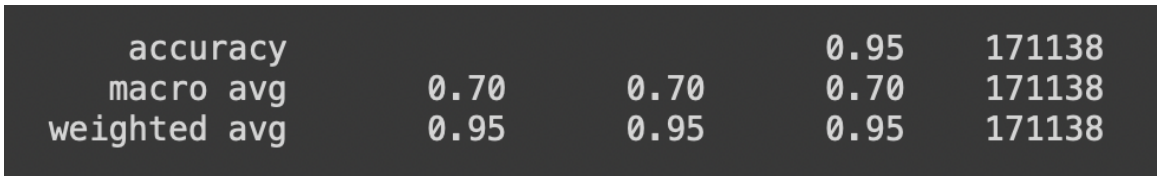
Solution

In this section, we try to fine-tune our hyperparameters (α) on the dev set and see if we get any better results on our test set with a different value. The results of our taggers have been provided in the previous question. We had used the metrics of precision, recall and F1 for each test case and reported the macro-average of the above values. The macro average F1 scores for different values of α have been given in table 1.

The best macro F1 score is highlighted in bold along with its corresponding value of α. Though many iterations of hyperparameter tuning were carried out, we included 5 values including the one giving us the best result.

After getting the ideal POS tagger, we now move to running it on our test dataset. Classification report for the test set is given in figure 5. We also printed the confusion matrix of the tagset to analyse our tagger better. Figure 6 has a small picture of our confusion matrix as the entire set was too big to be included in the report.

In the matrix, we were able to observe the shortcomings of our model with ambiguous tags as well. Though there weren't many such cases, we did see 'DT' tag being confused a lot with 'JJ', and 'JJ' being confused with 'NN'. Figure 7 gives us an example of such sentences where we see the differences between actual tags and predicted tags.



accuracy			0.95	171138
macro avg	0.70	0.70	0.70	171138
weighted avg	0.95	0.95	0.95	171138

Figure 5: Classification report for HMM model with best value of α on test set

After implementing both, a Baseline Tagger and HMM Tagger, we can compare how both models performed on test and dev set through the classification reports figured used in this assignment. If we compare the macro F1 scores in figure 2(Baseline tagger) and figure 4(HMM

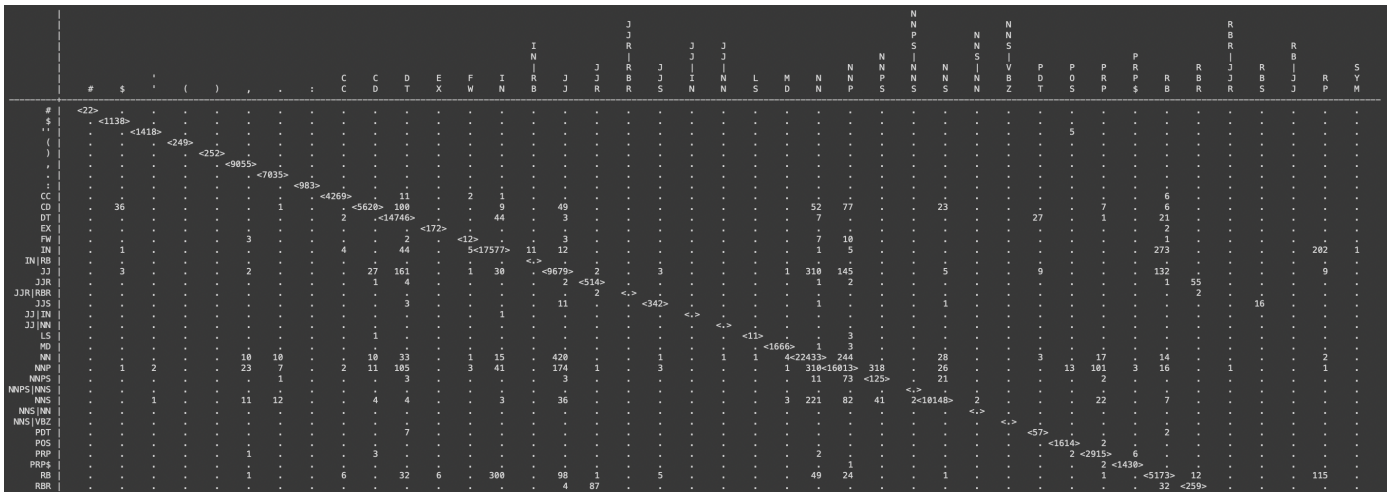


Figure 6: Small portion of the entire confusion matrix of POS tags on test set with the best value of α

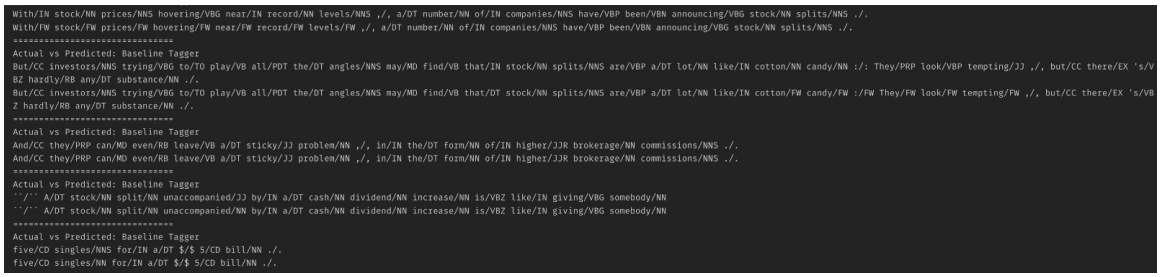


Figure 7: Example values from the dataset, comparing their actual and predicted tags from HMM model with best α score

Tagger with $\alpha = 1$), we can notice that the Baseline tagger clearly outperforms the HMM tagger. However, when playing with the value of α, we saw that HMM trumps baseline model with better F1 score and recall value. This shows the importance of implementing HMM as it provides us with room for enhancing performance with smoothing. We can also tweak the performance further using methods mentioned below.

Under the capacity of this assignment, we were successfully able to build a baseline and a HMM based POS tagger with viterbi algorithm. In our experiments we saw how competitive the results were and the scores weren't improving substantially despite numerous iterations of hyperparameter tuning. In such a case, there are still a lot of things we can do to further enhance the performance of our model. A few of those measures are as follows:

(a) *Implementing trigram-based tagger:* So far in this assignment, we have developed our model based on a bigram tagger. However, for a large dataset like ours, trigram based tagger can provide us with an even higher accuracy as we are considering 2 past words instead of one, which gives us a better contextual understanding. Though for a small training set, bigram would work better.

(b) *Using Latent Annotation:* The objective of training a bigram tagger with latent annotations is to find the transition and emission probabilities associated with the latent tags such that the likelihood of the training data is maximized. Unlike training a standard bigram tagger where the POS tags are observed, in the latent case, the latent tags are

not observable, and so a variant of EM algorithm is used to estimate the parameters.[1]

(c) *Laplace Smoothing (pseudocounts):* Laplace smoothing is a technique where you add a small, non-zero value to all observed counts to offset for unobserved values.

While going through numerous iterations of debugging, one thing that was important to observe was the amount of time required for the model to train and predict. In order to optimise our tagger further, and make it run in less amount of time, one approach that can be proposed and experimented with is **parallel implementation**. This provides a simple yet effective way to enhance our operations and use the entire power of our CPU/GPU. It is possible to use GPUs for throughput sensitive scenarios, where lots of requests can be computed at the same time. A detailed experimental study on this was done by Yuchen Huo and Danhao Guo in implementing a Parallel Hidden Markov Model.[2]

References

[1] Huang, Z., Eidelman, V., Harper, M. (2009, June). Improving a simple bigram hmm part-of-speech tagger by latent annotation and self-training. In Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics, Companion Volume: Short Papers (pp. 213-216).

[2] https://firebb.github.io/parahmm/.