5.1 Baseline

a. **Design a choice of informative word classes to replace rare/unseen words. What is the intuition behind your choice? Given a word, how would you categorize it into classes for your design method?**

For the first Task, I have designed one class of _RARE_ for unseen/rare words. The key idea is that for words when the threshold is less than 5 in the training set i.e. count of words in the training set is less than 5, we label it as _RARE_. The reason behind this is that we need to compute emission probabilities. The emission probabilities can help to represent the tag distribution among words. So, for the words which appear less than 5, we cluster all those words into 1 unknown word _RARE_. Emission probabilities of _RARE_ token will represent all those words that were used less than 5. So, when we test on unseen data it might be possible that some of the words don't appear at all in the emission probability distribution of the training set. So, to deal with these words we can think that since the appearance of these words is very less (in the training set it was not present), so to categorize all these words and to predict the tag we label it in _RARE_ class. In other words, when new words appear for the first time it categorizes their probability distribution similar to _RARE_ class category.

I have also tried another approach to replace infrequent words (less than 5 thresholds). I have created 4 classes:
Class 1: Represents all those words that contain numeric character(_NUME_)
Class 2: Represents all those words that are all-in upper-case letters (_LOWER_)
Class 3: Represents all those words that are all-in lower-case letters (_UPPER_)
Class 4: Represents all those words that don't fall in any of the above categories (_RARE_)

The intuition behind this is that to capture the distribution of emission probability is a detail I have categorized less frequent words into furthermore classes. So basically, in the training set when the count of the word appears less than 5, we classify them into the above categories. So, when we see a word that has appeared less than 5, we take the count from the above categories. The main objective of these classes is to capture the distribution in greater depth. In the previous one we were just categorizing into one label as _RARE_, but over here we are classifying subcategories according to their counts to cluster the distribution of probabilities that have a similar word count. So, to get the probability distribution in detail we can classify it into categories. When the threshold of the word is less than 5, we check whether all the words are in capital, lower, does it contain any numeric, so based on this we can further classify into several categories. The key intuition behind this technique is that when the threshold is less than 5, so probably the importance of word is less. Another intuition behind sub-categories is that we can segregate the words based on the category which it belongs to. This segregation represents the distribution of But in order to handle less frequent words and preserve its emission probability distribution we classify into several categories. The above categories will help to cluster less frequent words based on the categories to form clusters of categories; this cluster of categories will help in sequence tagging of words by finding the probability distribution in which a particular word belongs to.

b. **Evaluate and compare the new baseline model(s) on train and dev sets**

Following are the results obtained when infrequent words were replaced by _RARE_:

	Precision	**Recall**	**F1-Score**
Train Dataset	0.172470	0.701388	0.276861
Dev Dataset	0.158861	0.660436	0.256116

Following are the results obtained when infrequent words were further subcategories based on counts (_NUME_, LOWER_, _UPPER_, _RARE_,):

	Precision	**Recall**	**F1-Score**
Dev Dataset (With category numeric and rare)	0.159913	0.669238	0.258929
Dev Dataset (With category upper and rare)	0.158651	0.661526	0.258247
Dev Dataset (With category lower and rare)	0.158572	0.660101	0.254862
Train Dataset (With category upper, lower, numeric, and rare)	0.173145	0.701419	0.277681
Dev Dataset (With category upper, lower, numeric, and rare)	0.186923	0.660819	0.257823

From the above table as, numeric category is introduced along with rare, we see increase in all the params precision, recall and F1-score. We can see that precision increased from 0.18 to 0.159 when numeric category is introduced along with rare category.

We can see that as we increased the number of classes i.e., subcategorized into more classes we see increase in precision, recall and F1-score. So, there is increase in scores which represents that as we subcategorize _RARE_ into more categories we can see that model is able to understand the probability distribution and can predict the tag of the word. All the categories (Lower, Upper, Numeric and Rare), clusters the word which are less frequent into their corresponding distribution to form clusters of categories. Each category represents their own cluster of distribution which have similar resemblance of tags within the category. Thus, when we see a word which is from less frequent words, we classify its tag according to the category which it belongs to.
In the baseline model emission probability is calculated by below formula:

$$e(x|y) = \frac{\text{Count}(y \rightarrow x)}{\text{Count}(y)}$$

5.2 Trigram HMM

a. What is the purpose of the Viterbi algorithm - dynamic programming vs. brute force vs. greedy?

The main objective of the Viterbi algorithm is to make an inference/prediction on test data from trained model and some observation states. In the Viterbi algorithm we follow the following formula:

$$X^*_{0:T} = \text{argmax}_{X_{0:T}} P[X_{0:T}|Y_{0:T}]$$

Based trigram of the previous states, we need to compute the tag of the current words. For this we need an iterative approach of the kind in belief network with a chain of trigram associated. The key of this algorithm is to process the words from left to right. In the first step we compute the likelihood, we need to compute the tags which are the hidden states based on observed states (given tags). In Viterbi algorithm we tend to find the sequence of tags (I-GENE, O), based on observed tags of the words. For a sentence we are given the observed tags o_1, o_2,…o_n, and we need to find the most probable tags of the sentence h_1, h_2,… h_n. In Viterbi algorithm we need to decode h_1, h_2,… h_n.

For a given sentence Viterbi algorithm can be explained using the following formula:

$$\text{For any } k \in \{1,2,\dots n\}, \text{ for any } u \in S_{k-1} \text{ and } v \in S_k:$$
$$\pi(k,u,v) = \max_{S_{k-2}}(\pi(k\text{-}1,w,u) \times q(v|w,u) \times e(x_k|v))$$

Thus, we can see that current hidden tag is computed based on the previously computed value. The hidden tag for each position based on previous observed tags we iteratively compute most probable tag which can come in hidden state. We iteratively compute the tags for each word, based on these tags generated we take the maximum over these tags for the next state sequence. We compute the current most probable state using the previous states. Over here we use dynamic programming because we need to store the previous hidden as well as observed states to find the current most probable state. Firstly, we compute likelihood of observed output/tags of a given hidden Markov model [2]. Based on these we compute the most probable sequence which led to the generation of the current sequence. In other words, we try to figure out the probability distribution $\pi(k,u,v)$ of the tags for a given sequence using $\max_{S_{k-2}}(\pi(k\text{-}1,w,u) \times q(v|w,u) \times e(x_k|v))$. Finally, we find the probable word tag using argmax which maximize the estimated output from the observes tags $\arg\max_{S_{k-2}}(\pi(k\text{-}1,w,u) \times q(v|w,u) \times e(x_k|v))$ and stores in $bp(k,u,v)$. Using these bp lists we compute $y_k = bp(k+2, y_{k+1}, y_{k+2})$. This is done from right to left using dynamic programming to compute the most probable tag sequence. In the greedy approach we take argmax at each step, because of which it might be possible that it fails to take the best tag sequence overall as it chooses argmax for each word, thus it will fail to converge globally. Dynamic programming will help to keep the track of previous states which will help to compute the most probable current state. We can't use greedy or brute force approach because it will not lead to global optimality of the problem. Another reason for using dynamic based Viterbi approach is to reduce time complexity and to reduce time computations [3].

b. What are the specifics of your implementation: what is the base case; how did you implement the recursive formulation; how did you obtain the joint probability of word sequence and tag sequence; how do you go from this joint probability to final tag sequence, i.e., what path in your dynamic programming table gives you the final tag sequence? If you got stuck and your implementation is not working, describe the key challenges and the issues you faced.

Input: A sentence $x_1, x_2, \ldots x_n$, possible tags(I-GENE, O), parameters q(s|u,v) and e(x|s)

Initialization: Setting $\pi(0, "*", "*") = 1$

Definition: $S_{i-1}=S_0=\{"*"\}$, $S_k=S$, for $k \in \{1,2,\ldots n\}$

Algorithm:

Def Viterbi

For k = 1,2…n

For $u \in S_{k-1}$, $v \in S_k$,

$$\pi(k, u, v) = \max_{S_{k-2}}(\pi(k-1, w, u) \times q(v|w, u) \times e(x_k|v))$$

$$bp(k, u, v) = \arg\max_{S_{k-2}}(\pi(k-1, w, u) \times q(v|w, u) \times e(x_k|v))$$

Set(y_{n-1}, y_n) = $\arg\max_{(u,v)}(\pi(n, w, u) \times q("STOP"|w, u))$

For k = (n-2)…1, y_k=bp(k+2, y_{k+1}, y_{k+2})

Return the tag sequences $y_1, \ldots y_n$

At the end of a sentence, we add "STOP" tag to mark end of sentence, and initially in the start of token we add "*", "*", so this will make things easy to calculate trigram probabilities[1] to generate emission distribution.

For computation of maximum likelihood q(v|w, u) can be computed using the following formula:

$$q(v|w, u) = \frac{\text{Count}(w, u, v)}{\text{Count}(w, u)}$$

For the other term we $e(x_k|v)$ we compute using the following formula:

$$e(x_k|v) = \frac{\text{Count}(v \rightarrow x_k)}{\text{Count}(v)}$$

These above equations will help in generating emission probability distribution, using these equations I have computed the most probable sequence of tags ("I-GENE", "O"). Using these equations I have computed joint probability distribution of the word sequences $x_1, x_2, \ldots x_n$ with its tags $y_1, y_2, \ldots y_n$ using the following equation:

$$P(x_1, x_2, \ldots x_n, y_1, x_2, \ldots y_n) = q(y_1|*,*) \times q(y_2|*, y_1) \times \ldots\ldots \times q(\text{STOP}|y_{n-1}, y_n) \times \prod_{i=3}^{n} q(y_i|y_{i-2}, y_{i-1}) \times \prod_{i=1}^{n} e(x_i|y_i)$$

Thus, using the above equations, we will compute joint probability distribution. For generating the final tag sequence, we need to compute backward using the dynamic storage we have created. In a sentence $u \in S_{k-1}$, $v \in S_k$, I have stored value of the tags in bp(k, u, v) which maximizes probability distribution $\pi(k, u, v)$. Using these bp(k, u, v) we will compute most likely states by backtracking[4]. Using backtracking we will decode the tag sequences. With the help of the bp(k, u, v) we will backtrack and produce tags.

For k = (n-2)…1, y_k=bp(k+2, y_{k+1}, y_{k+2})

Using the above equation, we will backtrack and generate the tag sequence for the dev data.

Using the above algorithm, I have implemented Viterbi algorithm and is producing results as expected.

c. Evaluate the HMM tagger on the dev set, verify the F-1 score; how does it compare to the baseline tagger?

I have implemented HMM tagger on dev set. Following are the results obtained when less frequent words were replaced by _RARE_:

	Precision	Recall	F1-Score
Train Dataset using Baseline (_RARE_)	0.172470	0.701388	0.276861
Dev Dataset using Baseline (_RARE_)	0.158861	0.660436	0.256116
Train Dataset with HMM Trigram (_RARE_)	0.56173	0.350318	0.431892
Dev Dataset with HMM Trigram (_RARE_)	0.541555	0.314642	0.398030

Using these HMM tagger on dev set we can see that there is a huge improvement in the Precision. Recall and F-1 scores obtained. In the baseline model we have obtained precision obtained on dev set is 0.158861, our Trigram HMM model has given precision of 0.541555, so we can improvement in performance of model. Precision signifies that the number of true positives divided by the total number of positive predictions, over here positives are one tag("I-GENE"). In baseline model, recall value obtained is 0.660436, and the recall value obtained on HMM trigram model is 0.314542. Recall value signifies the number of tags samples correctly classified tag to the total number of tags. F-1 Score takes the harmonic mean of precision and recall. F-1 Score tries to embed the information from both precision and recall. On the baseline model I have obtained F-1 Score = 0.256116, I have also computed F-1 Score using HMM Trigram model, the F-1 Score obtained for this model is 0.398030, thus we that there is a huge improvement in the F-1 score using HMM Trigram model. From the above table we can also see that performance is improved on both the datasets train and dev.

d. Evaluate the new HMM model on dev set with informative word classes you designed.

	Precision	Recall	F1-Score
Dev Dataset using Baseline (_RARE_)	0.158861	0.660436	0.256116
Dev Dataset using Baseline (_NUME_+_RARE_+_LOWER_+_UPPER_)	0.186923	0.660819	0.257823
Dev Dataset with HMM Trigram (_RARE_)	0.541555	0.314642	0.398030
Dev Dataset with HMM Trigram (_NUME_+_RARE_+_LOWER_+_UPPER_)	0.51873	0.327431	0.400781

We can see that as the classes of categories are also introduced, we see an improvement in the model even further. When less frequent words were replaced by _NUME_+_RARE_+_LOWER_+_UPPER_, as done in baseline model, we see an improvement in recall and F-1 Score. When Trigram +HMM is applied using the subcategories we can see that recall improved from 0.314642 to 0.327431. Even F-1 score improved from 0.398030 with one category to 0.40078 with 4 subcategories. Thus, we see improvement in recall as well as F-1 Score. Similarly, we see a lot of improvement in scores when compared with first baseline mode, some of the values have improved 3 to 4 times for some of the accuracy params scores.

e. Provide insightful comparison based on these results.

In the baseline model with one category(_RARE_), we have seen that the model has achieved precision 0.158861, recall 0.660436 when subcategories(_NUME_+_RARE_+_LOWER_+_UPPER_) were introduced we see that there is a huge improvement is scores. Similar trend is followed in Trigram HMM, which shows that subcategories play a crucial role in determining the tag sequences of words. Subcategories helps to generate emission probability distribution curve in detail. These categories try to generate the probability distribution in detail, so when we encounter words from dev data which are less frequent words, we try to figure out the category in which this particular word lies, based on this we select the category to get the probability distribution and compute joint probability distribution. It can be seen that hoe F-1 score is improving as we are adding sub-categories like (_NUME_+_RARE_+_LOWER_+_UPPER_) which shows that model is learning the probability distribution of different words. The results obtained from Trigram HMM has significantly increased F-1 scores from 0.256116 to 0.398030 which shows that performance of model has improved using Trigram HMM. When the subcategories were applied on Trigram HMM the precision improved even further justifying the purpose of subcategories. Best performance is achieved on HMM Trigram (Infrequent words less than 5 with (_NUME_+_RARE_+_LOWER_+_UPPER_).

5.3 Extensions

a. Describe the extensions you made to the Trigram HMM tagger, evaluate the extended model and provide analysis.

For the extension I have implemented Laplace Smoothing. Laplace smoothing tries to flatten spiky distributions, so they generalize better, in Laplace smoothing we tend to increase probability of unseen events (which never appeared in training data) and tends to decrease probability of seen data in order to smoothen the probability distribution curve. For computation of maximum likelihood $q(v|w,u)$ can be computed using the following formula:

$$q(v|w,u) = \frac{\text{Count}(w,u,v) + \delta}{\text{Count}(w,u) + \delta \times V}$$

here V is the vocabulary, so we tend to add 1 in the numerator and V in the denominator thus it will smoothen the curve, increases the probability of trigrams which never appeared in the data at the same time reduces the probability of the trigram which has appeared a greater number of times. In this we can determine the amount of smoothing by adding the hyper-parameter δ (delta) which can determine the amount of smoothing, For the other term we $e(x_k|v)$ we compute using the following formula:

$$e(x_k|v) = \frac{\text{Count}(v \rightarrow x_k) + \delta}{\text{Count}(v) + \delta \times V}$$

So, whenever the word exists for one of the tags, in our previous case the model was returning 0, but over here based on Laplace smoothing instead of returning 0, it will smoothen the curve. I have tried Trigram + HMM + Laplace Model, I have conducted experiments with different values of delta, and the optimal delta after running for different value is $\delta = 0.1$

	Precision	**Recall**	**F1-Score**
Dev Dataset with HMM Trigram (Infrequent words less than 5 with _RARE_)	0.541555	0.314642	0.398030
Dev Dataset with HMM Trigram (Infrequent words less than 5 with (_NUME_+_RARE_+_LOWER_+_UPPER_)	0.518733	0.327431	0.40078
Dev Dataset with HMM Trigram + Laplace Smoothing (Infrequent words less than 5 with _RARE_)	0.521831	0.301982	0.391011
Dev Dataset with HMM Trigram + Laplace Smoothing (Infrequent words less than 5 with (_NUME_+_RARE_+_LOWER_+_UPPER_)	0.52114	0.32891	0.399012

b. Specific emphasis is placed on improved performance w.r.t the Trigram HMM model

In the above tables obtained we can see that using Laplace smoothing, there is an improvement in recall, which shows that Laplace smoothing tries to flatten spiky distributions, so they generalize better, in Laplace smoothing we tend to increase probability of unseen events (which never appeared in training data) and tends to decrease probability of seen data in order to smoothen the probability distribution curve.

We can see that recall value increases from 0.327431 to 0.32891 as value of delta is set to 0.1 in Laplace soothing. The above experiment was conducted on different values of delta, ranging from 0.0001, 0.001, 0.01, 0.1, 1.0, the most optimal value of delta obtained is 0.1.

	Precision	**Recall**	**F1-Score**
Dev Dataset with HMM Trigram + Laplace Smoothing (Infrequent words less than 5 with (_NUME_+_RARE_+_LOWER_+_UPPER_)	0.53114	0.32891	0.399012

References:

[1]. Majumder, P., Mitra, M., & Chaudhuri, B. B. (2002, November). N-gram: a language independent approach to IR and NLP. In International conference on universal knowledge and language.

[2]. Sarkar, K., & Gayen, V. (2013). A trigram HMM-based POS tagger for Indian languages. In Proceedings of the international conference on frontiers of intelligent computing: theory and applications (FICTA) (pp. 205-212). Springer, Berlin, Heidelberg.

[3]. Eddy, S. R. (1996). Hidden markov models. Current opinion in structural biology, 6(3), 361-365.

[4]. Ojokoh, B., Zhang, M., & Tang, J. (2011). A trigram hidden Markov model for metadata extraction from heterogeneous references. Information Sciences, 181(9), 1538-1551.