CSE 256: Statistical NLP: Fall 2019
Programming Assignment 3: Sequence Tagging

Kaustav Datta
kdatta@eng.ucsd.edu

University of California San Deigo

Problem

The task was to build a gene sequence tagger, first using a naive baseline tagger, and then using a trigram HMM with Viterbi decoding. We were asked to experiment with different approaches for handling unseen words in test data, and adding other extensions. My approach has been outlined below in each section.

1 Baseline

1.1 Description and Implementation

As a baseline, I implemented a simple baseline tagger, that predicts the tag with the highest emission probability for each word. Every word whose count in the training data was less than 5, was replaced with "_RARE_" in the training data. This was done so as to take care of unseen words in test data by replacing them with _RARE_. The model was then trained on this and the emission probabilities were recorded. For every word in test data, it was first replaced with "_RARE_" if it is not in training data, then the tag resulting in highest emission probability was chosen. This does not take into consideration context at all, and makes a prediction for each word independently. I was able to get the expected results, which is displayed as below.

Found 2669 GENEs. Expected 642 GENEs; Correct: 424.

	precision	recall	F1-Score
Baseline Tagger	0.158861	0.660436	0.256116

1.2 Choices for informative word classes

Replacing every rare word with a single tag "_RARE_" is not a very good choice, and performance can be improved by choosing better word classes. This requires finding patterns that words in the data conform to. I noticed that words in the training data that belong to the I-GENE class shared a few common characteristics. Many of them were composed of uppercase letters with numbers, or they ended with specific suffixes. Other than these, words could also be grouped based on whether they were numbers or punctuation.

Design choice 1
As before, I found all the rare words in my training data (those which had count lesser than 5). Each of these words were then assigned a word class. My first design choice was "onlyNum" as a word class, which contained words which were only numbers. On examining the training data, I noticed there were quite a few words which were numbers, especially among the infrequent words. Mostly in this data domain, different numbers are not very informative, hence the rare numbers can be grouped together into one class. I also included a few other word classes in this design choice like "onlyCaps" (all letters in the word are in uppercase) and "bothLettersAndNum" (the word consists of alphanumeric characters), since I

had noticed words with such patterns in the training data, and they were mostly marked as I-GENE. The rest of the infrequent words were marked as _RARE_. However, I noticed that the effect of "onlyCaps" and "bothLettersAndNum" individually was not much, and it was mostly "onlyNum" that contirbuted to performance improvement, probably because there were more cases of "onlyNum".

The results I got, when compared to the baseline, are shown below. We see that the precision increased only a little, as it now tagged lesser number of words as I-GENE (2644 as compared to 2669 with the baseline tagger). Thus, grouping words in these word classes did help to some extent in avoiding some words being wrongly tagged.

Found 2644 GENEs. Expected 642 GENEs; Correct: 424.

	precision	recall	F1-Score
Baseline with onlyNum+onlyCaps+bothLettersAndNum	**0.160363**	**0.660436**	**0.258065**
Baseline Tagger with only _RARE_	0.158861	0.660436	0.256116

Design choice 2
I noticed that a lot of words tagged as I-GENE ended in suffixes such as "ase", "ases","ate", "tide", "ogen", "lin", "min", "osis", etc. These words were also relatively long words. So I checked whether the length of a word was more than 6 and if it ended with any of the above suffixes. Such words were marked as "endsInSuffixes". Other than this, I also used two other word classes - "initCaps" (words which start with uppercase), and "punct" (words which only consisted of punctuation). The motivation for initCaps was that some rare words which were tagged as I-GENE began with an uppercase on account of it being a proper noun. There were words which were just punctuation, and different punctuations don't really have any meaning in the problem of gene sequence tagging. So such words were classified into one group "punct". However, the results obtained did not improve over the baseline tagger, possibly because such word classes were in such low number, that they were not adequate to influence the F1 score. Even considering each of the three word classes independently, the results were the same.

Found 2669 GENEs. Expected 642 GENEs; Correct: 424.

	precision	recall	F1-Score
Baseline with endsInSuffixes+initCaps+punct	**0.158861**	**0.660436**	**0.256116**
Baseline Tagger with only _RARE_	0.158861	0.660436	0.256116

2 Trigram HMM

2.1 Viterbi Algorithm, and DP vs Greedy vs Brute Force

The purpose of the Viterbi algorithm is during inference, for decoding, i.e, finding the optimal sequence of tags for a particular input sequence. This is done by using dynamic programming to find the sequence of output tags which maximise the joint probability of the input sequence, and the output tag sequence. There are three different approaches to this - Brute Force, Greedy approach, and Viterbi algorithm.

Brute force involves iterating over all possible tag sequences, and calculating the joint probability values for each of them with the input sequence. This is extremely inefficient, as the number of possible tag sequences is exponential in n (input sequence length), and thus it has $O(|K|^n)$ complexity, where K is the set of all possible tags.

Greedy approach on the other hand, involves making a hard decision at each word, by choosing the tag that maximises the product of transmission probability and emission probability of that word with the tag. While this is very fast (with complexity $O(n|K|)$), it is not optimal, and often leads to a drop in performance. This is because the subsequent words have no effect on the hard decision taken at a partciular word.

Viterbi algorithm is the most optimal method out of the three approaches. It has $O(n|K|^3)$ complexity for a trigram HMM. It uses dynamic programming to find the most likely sequence of output tags, and it

is much faster compared to the brute force approach which has complexity that's exponential in n, and arrives at a more optimal solution than the greedy approach.

2.2 Implementation details

I implemented a bottom up approach to the dynamic programming problem, where I created a DP table, in which entries were filled up starting from the base state to the destination state. There were three parameters to this DP problem - k (which runs from 1 to n),u and v. Here, u and v are the last two tags of the sequence of length k.
The base case here is $\pi(0, *, *) = 1$, and in my dp table this is given by $pi\_probs\_table[0]["*"]["*"] = 1$.
The recursive formulation was implemented as such:

$$pi\_probs\_table[k][u][v] = \max_{w \in \kappa_{k-2}} \{pi\_probs\_table[k-1][w][u] * q(v|w,u) * e(x_k|v)\} \quad (1)$$

The joint probability of word sequence and tag sequence is obtained by finding the maximum value of $pi\_probs\_table[n][u][v] * q(STOP|u,v)$ across all possible u and v's. This is formulated as below:

$$\max_{u \in \kappa_{n-1}, v \in \kappa n} \{pi\_probs\_table[n][u][v] * q(STOP|u,v)\} \quad (2)$$

To obtain the final tag sequence, I maintained backpointers, in another table bp_table, which is filled up as pi_probs_table gets filled up, but instead of the max value, the argmax value gets stored in bp_table. Thus it stores the optimal tag in each entry. The final tag sequence of length n was obtained as such -

$$output\_tags[-2], output\_tags[-1] = \operatorname*{argmax}_{u \in \kappa_{n-1}, v \in \kappa n} \{pi\_probs\_table[n][u][v] * q(STOP|u,v)\} \quad (3)$$

For k from ($n-3$ to 0):

$$output\_tags[k] = bp\_table[k+2+1][output\_tags[k+1]][output\_tags[k+2]] \quad (4)$$

2.3 Performance comparison on dev set

The performance of the Trigram HMM with Viterbi decoding was much better as compared to the baseline tagger, and there was an improvement in the F1 score from 0.256 to 0.398. This matched with the expected F1 score as mentioned in the report.

Found 373 GENEs. Expected 642 GENEs; Correct: 202.

	precision	recall	F1-Score
Trigram HMM + Viterbi	**0.541555**	**0.314642**	**0.398030**
Baseline Tagger	0.158861	0.660436	0.256116

2.4 Evaluation with 2 word classes

I employed the following design choices, as described below.

Design Choice 1
I used "onlyNum" and "bothLettersAndNum" this time, which gave me an F1-score of 0.429250. I disregarded "onlyCaps" because the accuracy with that was coming out to be lesser (0.420853). With other cominations of the three word classes, the performance wasn't any better than 0.429250. The results on the dev set are:

Found 411 GENEs. Expected 642 GENEs; Correct: 226.

	precision	recall	F1-Score
Trigram HMM with onlyNum+bothLettersAndNum	0.549878	0.352025	0.429250

Design Choice 2
I tried "endsInSuffixes", "initCaps" and "punct" as before. This time, there was a slight drop in accuracy compared to HMM tagger with only _RARE_ tags, from 0.398030 to 0.394942. I also tried by removing "punct" and it made no difference. Removing "endsInSuffixes" however decreased the F1 score to 0.389105. I tried a couple of other combinations of these, but the best performance was with all 3. The results on the dev set were:

Found 386 GENEs. Expected 642 GENEs; Correct: 203.

	precision	recall	F1-Score
Trigram HMM with endsInSuffixes+initCaps+punct	0.525907	0.316199	0.394942

2.5 Comparative Analysis

	precision	recall	F1-Score
Trigram HMM with onlyNum+bothLettersAndNum	0.549878	0.352025	0.429250
Trigram HMM + _RARE_ (baseline HMM tagger)	0.541555	0.314642	0.398030
Trigram HMM with endsInSuffixes+initCaps+punct	0.525907	0.316199	0.394942
Baseline Tagger with _RARE_	0.158861	0.660436	0.256116

From the above results, we see that Design Choice 1 improved over the baseline HMM tagger (one with no informative word classes), giving 0.42950 over the baseline HMM tagger's 0.398030. It turned out to be better in terms of both precision and recall.
Design 2, on the other hand, had worse precision than the baseline HMM tagger, but only slightly better recall. This reduced the F1-score.
However, when compared with the baseline tagger which was not a HMM, we see that baseline HMM's performace is much better. But we also see that the recall is much worse than the baseline tagger. The high F1 score is due to high precision. Thus, the HMM tagger basically predicted less number of genes (373), but more among them were accurate. The baseline tagger predicted 2669 genes, with only 424 out of them being correct, thus the low precision. However, since it got 424 genes right among the 642 genes, it's recall is high. HMM tagger got only 202 genes right among the total 642.

3 Extensions

I implemented linear interpolation, and computed a weighted sum of the transition probabilities for trigram estimate, bigram estimate, and unigram estimate.

$$q(v|w,u) = \lambda_1 * q_{ML}(v|w,u) + \lambda_2 * q_{ML}(v|u) + \lambda_3 * q_{ML}(v) \tag{5}$$

The values of λ_1, λ_2, λ_3 were chosen via hyperparameter tuning using grid search on the dev set, by varying different values of λ_1 and λ_2, and setting λ_3 as (1 - λ_1 - λ_2). This resulted in the best values of λ_1**=0.6,** λ_2**=0.1,** λ_3**=0.3**.

3.1 Results

The results obtained after choosing the best values of the λ parameters are shown in the table below in bold:

Found 422 GENEs. Expected 642 GENEs; Correct: 236.

	precision	recall	F1-Score
Trigram HMM + Viterbi + LinInterpol	**0.559242**	**0.367601**	**0.443609**
Trigram HMM + Viterbi	0.541555	0.314642	0.398030
Baseline Tagger	0.158861	0.660436	0.256116

After smoothing with linear interpolation, we can see that the F1 score is higher than the trigram HMM with _RARE_ words and Baseline tagger with _RARE_words. The λs that were automatically chosen during tuning suggest that the highest weightage is placed on the trigram estimate, since it considers the most context, and because that is the model our Viterbi algorithm is written based on. Thus that is the result that should be given more attention to, compared to unigram and bigram. Interpolation also ensures that we do not solely rely on the trigram estimate, and that we count some amount of information from both bigram and unigram. This extra information leads to a higher f1 score, and thus we get a smoother estimate.