Programming Assignment 4: Sequence Tagging

Vignesh Nanda Kumar
A59010704

The report presents the task of sequence tagging. First a baseline model is developed which is improved upon using a Hidden Markov Model (HMM) to tag gene names in biological text. The HMM model is implemented along with different decoders and is explained in the following sections.

1. Baseline

(a) The baseline method uses just the emission probabilities for decoding. Now to predict emission probabilities for words in the test data that do not occur in the training data, the first approach used is to map the infrequent words in the training data to a common class and treat unseen words as members of this class. The first class for baseline is mapping infrequent words (Count(x) < 5) to _RARE_ token.

The baseline tagger produces the tag $y^* = \arg\max_y e(x \mid y)$ for each word x in the test sentence. The baseline result is presented in Table 1.

Genes found	Genes expected	Genes correct
2669	642	424

Precision	Recall	F1-score
0.158861	0.660436	0.256116

Table 1: Baseline results on dev set

Now to improve the baseline, 2 main approaches were tried for rare word replacement as detailed below:

1. General word replacement: Since the data is biological data, it is bound to contain proper nouns, abbreviations (biological terms) and numbers. There are 4788 abbreviations, 5941 proper nouns and 777 numbers among the rare words in the corpus. So, the first approach is to replace rare words which are in these categories with their corresponding replacement words _ABBR_, _PROPER_NOUN_ and _NUMBER_. Any rare word not captured in this approach is given the _RARE_ tag.
2. Specific word replacement: To get more specific, the biological data will contain multiple biological terms with common suffixes (like -ze, ble, ies). So each of the rare words with such suffixes can be clubbed together to form a single word class. To get these classes, first I extracted the rare words from the corpus. Now I just targeted the length 2 and length 3 suffixes. After that, I remove the length 2 suffixes that are already part of the length 3 suffixes. Then I have another hyper parameter "Suffix_cutoff" to remove the suffixes with frequency < Suffix_cutoff. Each remaining suffix will be a word class of its own.

Given a word, I will first check for the specific word replacement before the general word replacement. For specific word replacement, I will extract the last two and last three letters (suffixes) and replace the infrequent word with its word class prepended and postpended with underscores.

(b) For evaluation, I check the performance of 1 separately and then combining 1 and 2 for different suffix cutoff values. The results are presented for the experiments performed with baseline decoding technique in Table 2 (evaluated on the dev set).

Rare word technique	Number of classes	Precision	Recall	F1-score
General word	4	0.160302	0.660436	0.257986
Specific word (SC = 30)	161	0.193197	0.663551	0.299262
Specific word (SC = 80)	54	0.181740	0.657321	0.284750
Specific word (SC = 100)	45	0.179574	0.657321	0.282086
Specific word (SC = 150)	30	0.182837	0.660436	0.286390
Specific word (SC = 200)	27	0.186381	0.665109	0.291169
Specific word (SC = 250)	22	0.179714	0.665109	0.282969
Specific word (SC = 500)	12	0.175576	0.665109	0.277814

Table 2: Evaluation with Rare word replacement on dev set

From the results I get a few interesting observations. Firstly the general word replacement technique with 4 classes is able to outperform the baseline approach. Now on adding specific word classes, we see that the more classes we have, the better the F1-score we are getting. There is an increase in performance with 27 classes, that could be because the classes could have been more informative in that case. But in general the trend is to have as many informative classes as possible and the suffix cutoff parameter can be tweaked according to the use case (to have good precision or good recall). As the suffix cutoff goes up, the number of word classes will go down. The best F1-score obtained using this technique is **0.299262** with suffix cutoff = 30 (resulting in 161 word classes).

2. Trigram HMM

(a) We can further improve the baseline by implementing the trigram HMM model along with the Viterbi algorithm for decoding (during prediction). This section will discuss the implementation of Viterbi algorithm and present evaluation results of the algorithm on the dev set.

Viterbi algorithm comes in during the decoding phase. The decoding problem is given an input sentence sequence $x_1, x_2, \ldots, x_n$ find $y_1, y_2, \ldots, y_{n+1}$ that maximizes $p(x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_{n+1})$. A solution for solving this would be to enumerate all such y sequences and pick the best. But this solution will have exponential (in the length of sequence – n) time complexity which is not desirable. This is where Viterbi algorithm comes in. It is a dynamic programming based decoding algorithm used in finding the best tags for a sequence in polynomial time complexity ($O(n|S|^3)$) ($|S|$ is the size of the tag set, n is length of the sequence).

(b) Implementation details:

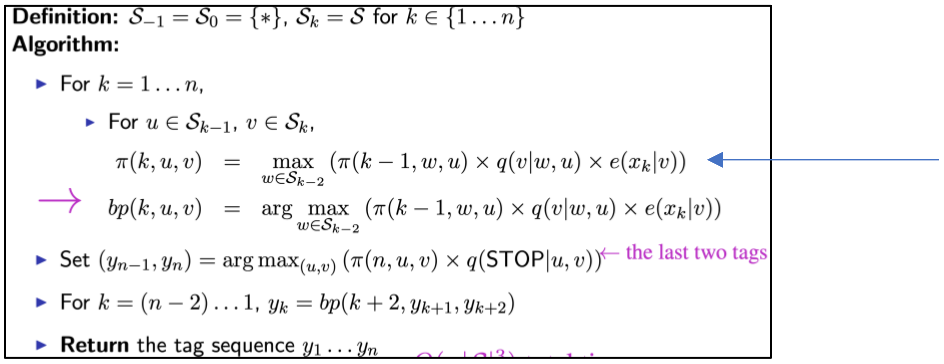
1. I have the emission and the trigram parameters precomputed which will be used in the Viterbi algorithm and also a given sentence S.
2. I initialize a DP matrix (a 3D dictionary) π and set $\pi(0, *, *) = 1.0$ (the base case). The DP matrix π gives the maximum probability of a tag sequence ending in tags u, v at position k. So, since every sentence starts with * , * in the trigram case, $\pi(0, *, *) = 1.0$.

I also store another matrix which is the backing matrix (which stores the max tag in every iteration).

$$r(y_{-1}, y_0, y_1, \ldots, y_k) = \prod_{i=1}^{k} q(y_i|y_{i-2}, y_{i-1}) \prod_{i=1}^{k} e(x_i|y_i)$$

$$\pi(k, u, v) = \max_{\langle y_{-1}, y_0, y_1, \ldots, y_k \rangle : y_{k-1}=u, y_k=v} r(y_{-1}, y_0, y_1 \ldots y_k)$$

Recursive formulation



```
Definition: $\mathcal{S}_{-1} = \mathcal{S}_0 = \{*\}$, $\mathcal{S}_k = \mathcal{S}$ for $k \in \{1 \ldots n\}$
Algorithm:
  ▸ For $k = 1 \ldots n$,
      ▸ For $u \in \mathcal{S}_{k-1}$, $v \in \mathcal{S}_k$,
          $\pi(k, u, v) = \max_{w \in \mathcal{S}_{k-2}} (\pi(k-1, w, u) \times q(v|w, u) \times e(x_k|v))$
          $bp(k, u, v) = \arg\max_{w \in \mathcal{S}_{k-2}} (\pi(k-1, w, u) \times q(v|w, u) \times e(x_k|v))$
  ▸ Set $(y_{n-1}, y_n) = \arg\max_{(u,v)} (\pi(n, u, v) \times q(\text{STOP}|u, v))$ ← the last two tags
  ▸ For $k = (n-2) \ldots 1$, $y_k = bp(k+2, y_{k+1}, y_{k+2})$
  ▸ Return the tag sequence $y_1 \ldots y_n$
```

Figure 1: Iterative formulation

3. We try to estimate the exact value of π using an iterative formulation given in Figure 1 above. Recursive approach is a top down approach which will have multiple overlapping paths which can be avoided using an iterative bottom up approach (dynamic programming) as shown in Figure 1. So we iterate over each word of the sentence, and for each word, we look at the tags of the current word and previous two words based on which we set the values in π and the backing matrix.
4. The equation marked in the blue arrow gives the formulation of computing the joint probability of word sequence and tag sequence upto position k. We iterate over all such tags w for the second previous word to find the best tag w which will be inserted in the backing matrix for position k, and tags u,v. While finding the joint probability value, I need to make sure that emission probability for unseen words is calculated correctly (rare words replacement is done properly). For computing the joint probability, the previous joint probability is required which is present in π, then the emission probability and the trigram probability is required which is all pre-computed.
5. The backing matrix is used to get the final tag sequence. Here first y_{n-1} and y_n is set. Then we iterate over k positions ranging from n-2 to 1 to set the tags of all the other positions using the backing matrix.

Issues that I faced during implementation were matching the correct u, v, w values while building the π matrix. I missed replacing the rare words with their word classes because of which I was getting very low F1-score since the tagging was not done correctly. After correctly replacing the unseen words, the algorithm worked.

(c) Evaluation

Genes found	Genes expected	Genes correct
373	642	202

Precision	Recall	F1-score
0.541555	0.314642	0.398030

Table 3: Baseline results, Trigram using Viterbi decoding

Table 3 shows the results of HMM tagger on the dev set. Compared to the baseline tagger, the precision, recall and F1-score is much better because we are using both the emission probability and the trigram probability during training and also using the exact Viterbi algorithm which will give the most optimal tags as output. Compared to the baseline tagger, the precision is better for the trigram HMM tagger but the recall is worse.

(d) Now, the below results show the performance of the trigram HMM model with the informative word classes designed in section 1.

Rare word technique	Number of classes			
General word	4	384	642	200
Specific word (SC = 30)	161	410	642	221
Specific word (SC = 80)	54	393	642	203
Specific word (SC = 100)	45	391	642	206
Specific word (SC = 150)	30	392	642	206
Specific word (SC = 200)	27	391	642	202
Specific word (SC = 250)	22	389	642	202
Specific word (SC = 500)	12	384	642	203

Rare word technique	Number of classes	Precision	Recall	F1-score
General word	4	0.520833	0.311526	0.389864
Specific word (SC = 30)	161	0.539024	0.344237	0.420152
Specific word (SC = 80)	54	0.516539	0.316199	0.392271
Specific word (SC = 100)	45	0.526854	0.320872	0.398838
Specific word (SC = 150)	30	0.525510	0.320872	0.398453
Specific word (SC = 200)	27	0.516624	0.314642	0.391094
Specific word (SC = 250)	22	0.519280	0.314642	0.391853
Specific word (SC = 500)	12	0.528646	0.316199	0.395712

Table 4: Performance using Trigram HMM model with Viterbi decoding on dev set

(e) We can see good improvement compared to the simple rare word replacement. In this case we observe that with more informative word classes we get the best precision recall and F1-score. On increasing the suffix cutoff, we see that precision and recall don't give a significant difference across different number of classes. We do get significant improvement in F1 score compared to the baseline because of including the transition probability now in our model.

But the F1-score is decreasing a bit on increasing the suffix cutoff. The best result that we get here is again while setting the suffix cutoff to 30 with 161 classes. The model is able to perform better than baseline results of Trigram HMM and is able to tag 221 Genes correctly. One observation here is that compared to the simple baseline model, the recall is lower because out of all words tagged as genes, the words correctly tagged as genes is lesser than the baseline model.

It could be possibly because the trigram HMM model considers only the local context via the transition probability (q) and also the emission probability assumes that the current tag just depends on the current word which are limitations of the trigram HMM model.

3. Extensions:

(a) There can be trigram present in the test data which don't have any existence in the training data. For such trigrams, the q trigram probability will be set to 0. We don't that to occur. To avoid such hard setting of values of 0, we use smoothing. Smoothing is done using the linear interpolation technique using the trigram, bigram and unigram estimate and lamdas for weighting each n-gram estimate. This smoothing technique should help in improving performance of model further.

[λ1, λ2, λ3]	Precision	Recall	F1-score
[0.33, 0.33, 0.34]	0.515152	0.370717	0.431159
[0.1, 0.3, 0.6]	0.550459	0.373832	0.445269
[0.6, 0.3, 0.1]	0.517647	0.411215	0.458333
[0.3, 0.6, 0.1]	0.532151	0.373832	0.439158
[0.1, 0.6, 0.3]	0.555035	0.369159	0.443405
[0.9, 0.05, 0.05]	0.385417	0.345794	0.364532
[0.05, 0.9, 0.05]	0.553957	0.359813	0.436261
[0.05, 0.05, 0.9]	0.504505	0.348910	0.412523

Table 5: Dev set Performance using smoothing

(b) We can see from the performance for different that the best F1-score and recall is coming from the model which is giving the major weightage to unigram estimate. And we get the best precision from the model which gives the most importance to the bigram estimate. But in general, the trend is that when all n-gram estimates are given some significant weightage, we get much better performance compared to the baseline and compared to a single n-gram estimate being given all the weightage.

To get further improvement, neural sequence tagger models can be explored which are able to capture long range dependencies.

References:

1. Lecture slides