Will Powers A59008589

Trigram HMM Sequence Tagging

Introduction

The goal of this paper, given a word/token sequence $w_1 \dots w_n \in \mathcal{W}^n$ from scientific biological texts, is to identify which tokens are referring to a specific DNA gene. Given a tagged training set, we will train a supervised model to produce a tag sequence $s_1 \dots s_n \in \mathcal{S}^n$ given a corpus, to identify if the token at the same index mentions a gene.

Baseline

For our baseline model, we will construct a simple conditional model that independently solves $\underset{s_i}{\operatorname{argmax}} P(w_i|s_i)$ for $i = 1 \dots n$, which using maximum likelihood estimation redefines to $P(w_i|s_i) = \frac{count(s_i \rightsquigarrow w_i)}{count(s_i)}$, where $count(s_i \rightsquigarrow w_i)$ is the word/tag pairing's training set frequency and $count(s_a \dots s_b)$ is a tag sequences' training set frequency.

Further, to handle words unknown to the training set, we will initially convert all words in the training corpus with a frequency less than five to a unique token *_rare_* and for unknown words during evaluation, we will also convert them to this token to share in their probabilities during prediction. The results of the initial model on the dev set are shown to the left.

Genes found	Genes Expected	Genes Correct	Precision	Recall	F1-Score
2669	642	424	0.16	0.66	0.26

To improve our scores, we can sort rare/unseen words into separate classes of tokens, rather than just one. We will be using the tokenization described in *Bikel et. al (1999) An algorithm that learns what's in a name*[1]. For rare/unknown words we will sort and tokenize them into the first valid self-descriptive word class in the following order: *twoDigitNum, fourDigitNum, containsDigitAndAlpha, containsDigitAndDash, containsDigitAndSlash, containsDigitAndComma, containsDigitAndPeriod, othernum, allCaps, capAndPeriod, firstWord, capitalizedWord, lowercaseWord,* and *other.* The results on the model of that classification are to the right. We can see the F1 Score has increased by significantly reducing the number of false positives while only marginally increasing false negatives.

Genes found	Genes Expected	Genes Correct	Precision	Recall	F1-Score
2111	642	403	0.19	0.62	0.29

If we continue to tweak these uniclass and multiclass rare word tokenizers, for both we have the most correct tags and the best F1-score when only considering training words with a

[1] Bikel, Dan & Schwartz, Richard & Weischedel, Ralph. (1999). An Algorithm that Learns What's in a Name. Machine Learning. 34. 10.1023/A:1007558221122.

frequency of one as rare. We can see the results on the right. The question of whether to choose the multiclass or uniclass rare word tokenizers depends on the purpose of the model. To optimize for the total genes identified, we should use the uniclass tokenizer. However, if we care about balancing the costs of false positives and false negatives, the multiclass tokenizer is best.

	Correct	F1 Dev
Multiclass	414	0.3095
Uniclass	428	0.2824

We will optimize further for the F1-score for more accurate tagging. We can improve the F1-score of the tokenizer by adding an additional class after *firstWord* called *WordGreaterThan4*. The threshold for separately tokenizing longer words was designed and tweaked after noticing in the training data many long and complex words tagged as a gene. We also experimented with other classes that tested for the number of unique characters in a word or if it contained certain rare letters in the English language, "Z", "Q", "J" and "X"[2]. Rare letter tokenization did nothing to change scores, suggesting the letters are randomly distributed among the words. The best tokenizer only added a *WordGreaterThan4* class for rare words of frequency one. The full results of that model are below. It improves upon the original multiclass tokenizer of rare words with a frequency of one by decreasing false positives and increasing false negatives only slightly less so.

	F1 Train	F1 Dev
WordGreaterThan4	0.3382	0.3168
WordComplexity GreaterThan4	0.3381	0.3132
containsRareLetter	0.3373	0.3095
WordGreaterThan4 AndContainsRareLetter	0.3373	0.3095

	Genes found	Genes Expected	Genes Correct	Precision	Recall	F1-Score
Train Set	50710	16637	11390	0.22461	0.68461	0.33824
Dev Set	1921	642	406	0.21134	0.63239	0.31681

Trigram HMM

To improve upon our baseline model, we will be using a trigram Hidden Markov Model, and treating individual sentences as sequences. As it is a generative model, we will be solving for $\underset{s_1 \dots s_n}{\operatorname{argmax}} P(w_1 \dots w_n, s_1 \dots s_{n+1},)$, where $s_{n+1} = STOP$, a symbol denoting the end of a sentence.
We assume a word is conditionally dependent on its tag and a tag is conditionally dependent on the two preceding tags. Thus the objective function simplifies to the following:
$\underset{s_1 \dots s_n}{\operatorname{argmax}} P(w_1 \dots w_n, s_1 \dots s_{n+1}) = \underset{s_1 \dots s_n}{\operatorname{argmax}} P(STOP|s_{n-1}, s_n) \prod_{i=1}^{n} P(w_i|s_i) P(s_i|s_{i-2}, s_{i-1})$.
We define, using maximum likelihood estimation $P(w_i|s_i) = \frac{count(s_i \rightsquigarrow w_i)}{count(s_i)}$, $P(s_i|s_{i-2}, s_{i-1}) = \frac{count(s_{i-2}, s_{i-1}, s_i)}{count(s_{i-2}, s_{i-1})}$, where $s_i = *, for\ i < 1$ a symbol denoting the start of a sequence.

Since the tags are still conditionally dependent on others in the sequence, we will not be able to independently maximize subfunctions. Therefore, in the brute force approach, we would need to calculate the objective function for all possible variations of $s_1 \dots s_n$ and then calculate the

[2] http://norvig.com/mayzner.html

maximum result. This would take O($|\mathcal{S}|^n$) runtime, which is unduly costly. The greedy approach is to independently find $\underset{s_i}{\operatorname{argmax}}\, P(w_i|s_i)P(s_i|s_{i-2}, s_{i-1})$ for $i = 1 \dots n$, using the previously solved sequence $s_1 \dots s_{i-1}$ at each iteration. This will save on computing time as the runtime is now $O(n|\mathcal{S}|)$. However, this assumption could prove to produce a poor result. One cannot know the word sequence ahead of time and at further iterations, prior calculated sequences could now be used to produce tags, given new words, that bring down the joint probability of the sequence. Also, it does not consider the cost of $P(STOP|s_{n-1}, s_n)$. So, it could have found incredibly unlikely ending words, which would bring the probability of that sequence way down.

The best approach is by dynamic programming with the Viterbi algorithm, which computes at every iteration $i = 1 \dots n, \forall\, u, v \in \mathcal{S} \cup \{*\}$, $p^*(i, u, v) = \max_{s_1 \dots s_{i-2}} P(w_1 \dots w_i, s_1 \dots s_{i-2}, u, v)$. This is the joint probability of the word sequence up to w_i and its most likely tag sequence ending in u, v. It calculates that efficiently by saving computed values and reusing them at the next set as such: $p^*(i, u, v) = \max_{x \in \mathcal{S}} p^*(i-1, x, u)P(w_i|v)P(v|x, u)$ where $p^*(0,*,*) = 1$. For a base case: $p^*(0, u, v) = 0\ \forall\ u, v \in \mathcal{S} \cup \{*\}, u \neq * \ or\ v \neq *$. And since we want the sequence to take into context the end of the sentence as well, we will additionally find $\max_{u,v \in \mathcal{S}} p^*(n, u, v)P(STOP|u, v)$ to produce the final probability of the most likely sequence. Since the program efficiently reuses variations of the possible tag sequence it has already seen so it can compute the joint probability in $O(n|\mathcal{S}|^3)$ runtime. The cost goes up multiplicatively by $|\mathcal{S}|$ by the size of the n-grams used in its calculation.

To know the full sequence, we save the tag preceding u, v in the most likely tag sequence ending in u, v for $w_1 \dots w_i$, $s^*(i, u, v) = \underset{x \in \mathcal{S}}{\operatorname{argmax}}\, p^*(i-1, x, u)P(w_i|v)P(v|x, u)$ for each $p^*(i, u, v)$. We also solve for $s_{n-1}, s_n = \max_{u,v \in \mathcal{S}} p^*(n, u, v)P(STOP|u, v)$. After solving for last two tags in the sequence, we can backtrack to find $s_i = s^*(i, s_{i+1}, s_{i+2})\ for\ i = n-2 \dots 1$.

This formula was implemented in python 3.8. p^* and s^* were implemented as dictionaries. p^* was implemented as a defaultdict that returned 0 to unseen input so as to save time on iteration. The algorithm could have been implemented recursively by calling p^* as a function and having its probability and its sequence returned (rather than storing the sequence in s^*). However, that appeared cumbersome, slower and could risk stack overflow in long sequences. Also, the iterative version was easily provided.

Tokenizer	Precision	Recall	F1
Uniclass	0.542	0.315	0.398
Multiclass	0.535	0.388	0.450

The dev set results of the trigram HMM model on the dev set with the original uniclass (using frequency less than five) and optimized multiclass rare word tokenizers are shown to the right. With both tokenizers, the HMM model greatly decreased the amount of false positives at the expense of decreasing true positives, as shown in the increased precision and decreased recall when compared to previous models. However, both have a greatly increased F1-Score, which is a testament to the overall accuracy of the HMM model. The multiclass rare word tokenizer when compared to the uniclass tokenizer also

increased the number of true positives, without increasing false positives enough to affect precision much, thus bringing up the F1-score. Therefore, it is optimal to the uniclass tokenizer in terms of accurately tagging words while balancing the number of false positives and false negatives. Using the uniclass tokenizer may return slightly fewer incorrect tags, however it will miss many more words that should have been tagged. For someone/algorithm that must review the resulting words tagged as genes for some purpose, it will be far more beneficial to have many more correct tags, while having to discard only a few more incorrect tags upon examination. The full dev set results of the multiclass tokenizer HMM model are below.

Genes found	Genes Expected	Genes Correct	Precision	Recall	F1-Score
465	642	249	0.535	0.388	0.450

Extensions

To improve the Trigram HMM model; we will extend the Viterbi algorithm to be used for not only trigram buy n-grams of any length $n > 1$ by storing $p^*(n, u_{i-n+1} \dots u_i), u_i \in \mathcal{S} \cup \{*\}$ and reusing them to efficiently solve $\underset{s_1 \dots s_n}{\operatorname{argmax}} P(STOP|s_{n-1}, s_n) \prod_{i=1}^{n} P(w_i|s_i) P(s_i|s_{i-(n-1)} \dots s_{i-1})$. This will help our model use a larger surrounding context when assessing the most appropriate tag to use next. We will also be using the Katz Backoff (KBO) smoothing technique when solving for conditional probabilities. This will allow for the probability distribution to be appropriately designated for n-grams not seen in the original test data that are likely to be seen outside of it. Below we can see the dev set results of models with various improvements.

	Genes found	Genes Expected	Genes Correct	Precision	Recall	F1-Score
3-Gram HMM + KBO w/ .6 discount	465	642	252	0.54	0.40	0.46
4-Gram HMM	474	642	251	0.53	0.40	0.45
4-Gram HMM + KBO w/ .6 discount	482	642	258	0.54	0.40	0.46
7-Gram HMM + KBO w/ .8 discount	486	642	273	0.56	0.43	0.48

After iterating over many possible variations, the 7-Gram HMM model with Katz Backoff using a discount factor of .8 produced the best F1-Score as well as the best precision and recall of all previous models, specifically over the Trigram HMM model. Thus, it produced the truest positives, while having the smallest ratio of retrieved positives that are false ones. Using this model would produce the more correctly identified word tagged as genes and one would not have to sift through the results as much to discard falsely identified ones. Therefore, since it is the most accurate model by all three of our measures it is the most optimal.