Sequence Tagging

Statistical Natural Language Processing - PA3

Saideep Reddy Pakkeer
University of California San Diego
13.05.2019

INTRODUCTION

In this exercise, give a sentence x = ⟨x1, x2, . . . , xn⟩ we will try to tag each word in the sentence. Specifically, we will try to find the most likely sequence of tags $\widehat{y}$ = ⟨y1, y2, . . . , yn⟩. In this corpus, we just have 2 tags {O, I-GENE} to identify each word as a gene or not. First we will analyze a simple counting scheme that looks at how frequent a tag appears for a word without looking at any context and then we will build a HMM model to tag the sequences.

BASELINE

Description and Implementation details:

The baseline tagger is just calculating the highest y^* = arg maxy e(x|y) where the emission parameter e(x|y) = Count(y $\rightsquigarrow$ x)/ Count(y) . So, what the baseline is basically doing is give the tag that has the highest value for the following metric: counting the number of times a tag occurs with a word per occurrence of that tag.

In this implementation, we group the rare (with frequency<5) and unseen words into a seperate class ("_RARE_"). For example, in the new corpus if we are trying to assign a tag to any new word (or rare) we simply assign the tag which has the highest e("_RARE_"/tag). I also experimented with different thresholds (<4, <5, <6) to see how our implementation performs. We will use this implementation and see the results on the dev and train data below along with new techniques (other than _RARE_) to group the rare words.

Result on Dev set:

Frequency of rare words replaced	Precision	Recall	F1- Score
freq<4	0.162193	0.658879	0.260308
freq<5	0.158861	0.660436	0.256116
freq<6	0.153818	0.655763	0.249186

Next we will look at different word classes and their combinations to improve our results - (i) **Numbers** (ii) **All Upper Case** (iii) **Long Words** (iv) **Mix of alphanumeric** - at least one character being alphabet and one being number - we use a regular expression for this

INFORMATIVE WORD CLASSES - I (2-digit, 4-digit Numbers & Long Words)

Motivation/Description:

Grouping all the infrequent words together to _RARE_ is not a very good/informative way to form a word class. Taking cues from the lecture notes and looking at few cases where the model is predicting wrong it seemed like a good idea to group 2 digit numbers and 4-digit numbers as an informative word class. The motivation is of the 314 words coded as _FOURDIGITNUMBER_ almost all of them except 7 have a tag "O". Hence, it became important to keep this class separately. Similarly, all 2-digit numbers are tagged "O". The other class being very long words. So, after checking how the length of words are distributed, I decided to group words with length>15 as one class (_LONGWORD_). 98 words grouped as _LONGWORD_ are tagged I-GENE among 800.

Evaluation on train and dev:

Dataset	Precision	Recall	F1- Score
Train	0.174797	0.693695	0.279234
Dev	0.163517	0.660436	0.262133

So, it definitely improved the performance a little. This is done only on the words with frequency less than 5 to keep the comparisons consistent. As we have seen in the first table, this could be improved for a different threshold.

INFORMATIVE WORD CLASSES - II (All Characters in Upper Case & Other Digit Numbers)

Motivation/Description:

So, after further looking at the cases where the mistakes happen, it seems like words which are all upper case like EGFP, DENTT are all tagged "O" but indeed the correct tag was "I-GENE". Hence, I decided to make another word class that groups them separately. In addition, if the number of digits other than 2-digit & 4-digit are made to another separate class instead of grouping them into _RARE_ class.

Evaluation on train and dev:

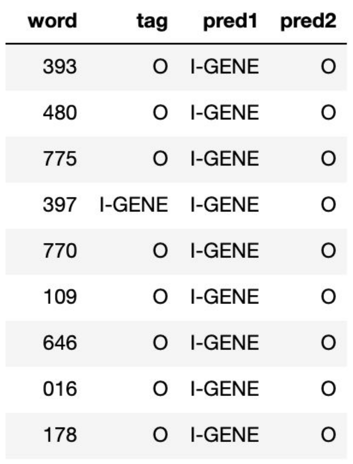
Dataset	Precision	Recall	F1- Score
Train	0.175915	0.693394	0.280633
Dev	0.164660	0.660436	0.263600

This is done on top of the first word class. Hence, you see a further improvement in the scores.

More word classes including the above are discussed after the Viterbi implementation of HMM model.

COMPARATIVE ANALYSIS

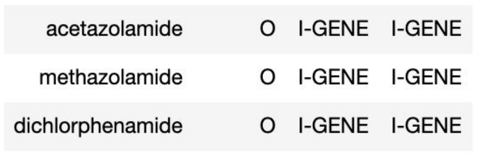
Although, the scores don't seem to improve very much compared to the implementation without any word classes, it seems a reasonable improvement for a model that is just based on counts. So, to compare the two designs we described above, I decided to see where the two designs don't agree in their tags. After doing that analysis I found the following:



word	tag	pred1	pred2
393	O	I-GENE	O
480	O	I-GENE	O
775	O	I-GENE	O
397	I-GENE	I-GENE	O
770	O	I-GENE	O
109	O	I-GENE	O
646	O	I-GENE	O
016	O	I-GENE	O
178	O	I-GENE	O

Here, pred1 is the Informative word class - 1 and pred2 is the tag given by the informative class - 2. So, our idea of grouping the digits of different lengths (other than 2 & 4) has worked out in getting the correct tags.

Next, I looked at the tags that both the designs get wrong. Especially on the long words class that we created. Here is what I found:



acetazolamide	O	I-GENE	I-GENE
methazolamide	O	I-GENE	I-GENE
dichlorphenamide	O	I-GENE	I-GENE

So, here the actual tag was "O" but both our models failed to predict the correct tag. As we have seen before on the dev set, long words are likely to be I-GENE among other kinds of I-GENEs (98 tagged I-GENE only among the infrequent ones).

Trigram HMM - Viterbi algorithm implementation

Motivation behind HMM

It seems rather a naive approach to just use the counts to determine the tags. As it is often said, "you shall know a word by the company it keeps". We intend to use the words in the context to determine the tag for a particular word. In HMM we precisely try to do this by trying to model the joint probability distribution of a tag sequence and a sentence. For tractability and practical purposes, we employ the Markov assumption that the conditional dependence of a word is on the previous few words. In case of a trigram HMM, we make the assumption that it depends on the previous 2 words. In case of 4-gram, we use the previous 3 to model our joint distribution. We will see both the trigram and 4-gram implementations in this exercise. We also do interpolation of all the above which is discussed at the end in the extensions.

Viterbi Algorithm - Purpose

Our problem of modeling joint distribution is as follows: $\arg\max_{y_1 \cdots y_{n+1}} p(x_1 \ldots x_n, y_1 \ldots y_{n+1})$ where x's are our input sequence and y's are our tags. We use maximum likelihood estimate for finding the best tag sequence. Now you can see that if we use brute force to solve this we require $|S|^n$ number of tag sequences where |S| is the total number of tags (S is the tag set). So, we use dynamic programming to solve the subproblems and populate it in a tabular manner, so as to not repeat the calculation of already solved sub-problems. This is the whole purpose of the Viterbi algorithm. In the next section we explain how this algorithm works and how we are avoiding the repeated calculations.

Viterbi Algorithm - Implementation details

Now we will discuss the implementation details of the algorithm. The base implementation is the trigram HMM where we condition the tag sequence probabilities only on the previous two tags. Let us define some terminology before we jump to the implementation. Let n be the length of our input sentence $x_1, x_2, x_3 \ldots x_n$ & S is the tag set {O, I-GENE}. We define S_k as {O, I-GENE} for k = {1,2, .. n} and $S_0 = S_{-1} = \{*\}$. We also define emission parameter e(x/y) which is the same as in the baseline defined in first section and a transition parameter q(s|u,v). It is a parameter which is for any trigram (u, v, s) such that s ∈ S ∪ {STOP}, and u, v ∈ S ∪ {*}, the value for q(s|u, v) can be interpreted as the probability of seeing the tag s immediately after the bigram of tags (u, v). We take the maximum likelihood estimate for q(s/u,v) = Count(u,v,s)/COunt(u,v). Then define a truncated version of p which is a joint distribution of words and tag sequences up to the k^{th} word in the sentence

$$r(y_{-1}, y_0, y_1, \ldots, y_k) = \prod_{i=1}^{k} q(y_i|y_{i-2}, y_{i-1}) \prod_{i=1}^{k} e(x_i|y_i)$$ & $$\pi(k,u,v) = \max_{\langle y_1 \ldots y_k \rangle \in S(k,u,v)} r(y_1 \ldots y_k)$$ where S(k,u,v) are set of all tag sequences of length k, which end in the tag bigram (u, v). Below is our dynamic programming implementation: Base case - π(0,*,*) =1 as every sentence starts with * * and π(0, u, v) = 0 for all (u, v) such that $u \neq *$ or v $v \neq *$. Then we define the recursive definition to get w

For any $k \in \{1 \ldots n\}$, for any $u \in \mathcal{S}_{k-1}$ and $v \in \mathcal{S}_k$:

$$\pi(\underset{\{1,2,\ldots,n\}}{k}, \underset{S_{k-1}}{u}, \underset{S_k}{v}) = \max_{w \in \mathcal{S}_{k-2}} \left(\pi(k-1, w, u) \times q(v|w, u) \times e(x_k|v)\right)$$

Now along with π values, we also keep track of $$bp(k,u,v) \;=\; \arg\max_{w \in \mathcal{S}_{k-2}} \left(\pi(k-1, w, u) \times q(v|w, u) \times e(x_k|v)\right)$$ back pointers to get the tags. Back pointers are nothing but the tags w that maximized our π in the recursive definition. So, the first we get the final two tags (y_{n-1}, y_n) using $\arg\max_{(u,v)} \left(\pi(n,u,v) \times q(\text{STOP}|u,v)\right)$ and then we back calculate other tags using the calculated tags using : $y_k = bp(k+2, y_{k+1}, y_{k+2})$ for k = n-2 to 1.

The way I implemented it is: After I get the trigram, bigram counts from the train data. I read one sentence at a time, run the algorithm and find the sequence of tags and write to the output and then go to the next sentence. Whatever logic I used for creating the word classes, I replicate the same while reading the sentences from dev data.

Result on Dev set

Dataset	Precision	Recall	F1- Score
Dev	0.541555	0.314642	0.398030

INFORMATIVE WORD CLASSES - III (Capitalize everything + check all uppercase + numbers + special character)

Motivation/Description:

As mentioned in the previous word classes we use another combination from 4 unique approaches we gave. Here, we capitalize everything and take the word counts again. Now based on the new word counts we treat the infrequent words (<5 frequency) and apply the following transformations to group the words into classes.

- Check if all the letters are uppercase
- Whether the length of the word>15
- If the word is a 2-digit, 4-digit or any other digit number
- Any special character in the word

Again, additional to the motivation is the same as discussed before, we want to see if any special characters will improve the accuracy. Below are the results for this design

Evaluation on train and dev:

Dataset	Precision	Recall	F1- Score
Train	0.543056	0.361243	0.433872
Dev	0.530414	0.339564	0.414055

INFORMATIVE WORD CLASSES - IV (At least one numeric and one character + all the above)

Motivation/Description:

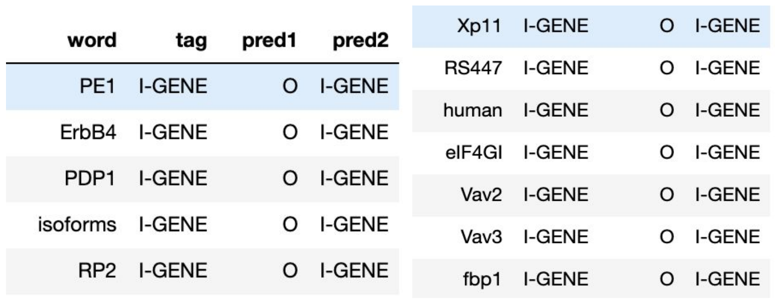
Apart from the above word classes we introduce one more class(_MIXALPHANUM_) which is as follows: We check if the word has at least one number and once alphabet in it using a regular expression - "^(?=.*[a-zA-Z])(?=.*[0-9])". The motivation for this is we see many words tagged I-GENE with a combination of letters and numbers like {C3, D3 etc.}. There are 3283 words grouped as _MIXALPHANUM_ using this design

Evaluation on train and dev:

Dataset	Precision	Recall	F1- Score
Train	0.549813	0.397067	0.461120
Dev	0.540094	0.356698	0.429644

COMPARATIVE ANALYSIS

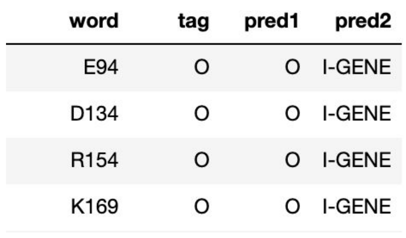
The base case gave a F1-score of 39.8 and we can clearly see the improvement in the approaches we have described above. If we analyze few examples of where the difference between two designs occur we observe:



word	tag	pred1	pred2
PE1	I-GENE	O	I-GENE
ErbB4	I-GENE	O	I-GENE
PDP1	I-GENE	O	I-GENE
isoforms	I-GENE	O	I-GENE
RP2	I-GENE	O	I-GENE
Xp11	I-GENE	O	I-GENE
RS447	I-GENE	O	I-GENE
human	I-GENE	O	I-GENE
eIF4GI	I-GENE	O	I-GENE
Vav2	I-GENE	O	I-GENE
Vav3	I-GENE	O	I-GENE
fbp1	I-GENE	O	I-GENE

where the first row is the actual tag, pred1 is the prediction from word class III and pred2 is the prediction from word class IV. A mix of alphabets and numbers are failed by the first approach where are majority are clearly being captured by our alphanumeric word class.

We also see cases the other way but relatively less in number. Few examples shown below:



word	tag	pred1	pred2
E94	O	O	I-GENE
D134	O	O	I-GENE
R154	O	O	I-GENE
K169	O	O	I-GENE

So, here the actual tag was O but our word class IV gets it wrong

EXTENSIONS

We consider two non-trivial extensions to improve our model. 1) Interpolation 2) 4-gram HMM

Interpolation - Motivation/Description:

The interpolation is an extension of our trigram HMM where we also take bigram and unigram estimate of the transition parameters q. So, our new estimate for q will be

$$
\begin{aligned}
q(\text{Vt} \mid \text{DT, JJ}) \;=\; & \lambda_1 \times \frac{\text{Count(Dt, JJ, Vt)}}{\text{Count(Dt, JJ)}} && \text{Trigram ML Estimate} \\
& +\lambda_2 \times \frac{\text{Count(JJ, Vt)}}{\text{Count(JJ)}} && \text{Bigram ML Estimate} \\
& +\lambda_3 \times \frac{\text{Count(Vt)}}{\text{Count()}} && \text{Unigram ML Estimate}
\end{aligned}
$$

$$\lambda_1 + \lambda_2 + \lambda_3 = 1, \quad \text{and for all } i,\ \lambda_i \geq 0$$

with the rest of the emission parameters being the same. The motivation to do this being it clearly captures the interactions between trigrams, bigrams and unigrams. So, for whatever we have done so far is a subset of this (for $\lambda_1 = 1,\ \lambda_2 = 0,\ \lambda_3 = 0$).

Interpolation - Results:

Here are the results on the dev set for different values of $\lambda_1,\ \lambda_2, \lambda_3$

Lambda values	Precision	Recall	F1- Score
$\lambda_1 = 0.5,\ \lambda_2 = 0.25,\ \lambda_3 = 0.25$	0.541401	0.397196	0.458221
$\lambda_1 = 0.5,\ \lambda_2 = 0.1,\ \lambda_3 = 0.4$	0.543021	0.442368	0.487554
$\lambda_1 = 0.5,\ \lambda_2 = 0.15,\ \lambda_3 = 0.35$	0.558140	0.448598	0.497409

The scores on the train data for the last set of lambda values are the following:

precision	recall	F1-Score
0.557902	0.468534	0.509327

4-gram - Motivation/Description:

For the next extension we go one order higher and implement 4-gram HMM. The motivation is as simple as extending the context that we look into for calculating the transition parameters for our HMM. Apart from that we can interpolate just as we did before including the 4-gram estimates for better results. Here is a brief description of how our implementation gets changed:

We estimate our transition parameters q(v/t,w,u) for 4-grams as Count(t,w,u,v)/Count(t,w,u). So, we extract the 4-gram counts and divide by the trigram counts. In case of interpolation we just add another lambda parameter $\lambda_1 * q_{4gram} + \lambda_2 * q_{trigram} + \lambda_3 * q_{bigram} + \lambda_4 * q_{unigram}$. We iterate over one more tag for populating our backpointer and pi tables.

4-gram - Results:

Below are the results on the dev set after including the 4-gram HMM:

Lambda values	Precision	Recall	F1- Score
$\lambda_1 = 0.15,\ \lambda_2 = 0.4,\ \lambda_3 = 0.1,\ \lambda_4 = 0.35$	0.560878	0.437695	0.491689
$\lambda_1 = 0.05,\ \lambda_2 = 0.45,\ \lambda_3 = 0.15,\ \lambda_4 = 0.35$	0.563953	0.453271	0.502591

The scores on the train data for the last set of lambda values are the following:

precision	recall	F1-Score
0.556727	0.469556	0.509439

REFERENCES:

- http://www.cs.columbia.edu/~mcollins/courses/nlp2011/notes/hmms.pdf
- http://cseweb.ucsd.edu/~nnakashole/teaching/256_sp19.html - lectures