CSE291 Projec2

Shanchuan You

Abstract

In this report, I will discuss the Conditional Random Field applied on the Bi-LSTM RNN. That is, every time we obtain features from LSTM, instead of directly computing the cross entropy between predicted sequence and the ground true tag, we introduce a higher level idea that it no longer treats each word in the sequence independently, it also considers each word's role in the whole sequence.

1 Model Description

In general, the model will be divided into three main parts. The **first** part will be the initialization of everything and adding sentence into the LSTM model to compute and extract features. The **second** part is to compute the negative loss likelihood, which we obtain the score of a provided tag sequence and then minus the partition function, which is simply the proportionality constant (conditioned on input words) to ensure that the total probability over all tags Y correctly adds up to 1. The main purpose of this model is to **minimize** the negative log likelihood. The **last** part is to perform the viterbi algorithm to select the highest probability of final predicted tags.

2 Initialization and LSTM

In general, the initialization is basically the same as the, except that we no longer input the length of the tags, but a dictionary $tagtoix$ instead, where $tagtoix$ = ”O”: 0, ”I-PER”: 1, ”I-ORG”: 2, ’I-LOC’: 3, ’I-MISC’: 4, ’B-MISC’: 5,’B-ORG’: 6,’B-LOC’: 7, START TAG: 8, STOP TAG: 9. The dictionary is used to build our transition matrix where each entry of the transition matrix, namely, Tij, the ith row and the jth coloumn of the matrix, represents the weight from jth tag to ith tag.

Initially, each entry of the transition matrix will be set as number between -1 and 1 selected by normal distribution, as the initial value does not matter since computing negative loss likelihood force the matrix to be learnt, and the values will be adjusted later.

3 Partition function and the Score of the sentence

In this part, We calculate the partition function, forward score, and the score of the sentence with tags provided, which is the gold score. Then we perform forward score - gold score. Since both scores are calculated by the log. The subtraction is used to calculate the negative log likelihood, namely, NLL.

3.1 Score of the sentence

This part computes the score of an input sentence when the tag sequence is given. Note that we are using log with plus/add operation, this is the same that log(a*b) = loga + logb.

The core part of this function is that the input sentence is actually a 2D array, namely, the emission feature computed by the LSTM model, which has dimension [n, 8], where n is the sentence length, and 8 is the probabilities of tag with respect to the current words. In other word, for each word in the sentence, we get a probability list of 8 potential tags.

Then, the computation will would be: for each word in the sentence, score = score + emission of the given tag on this word + weight from previous tag to the current tag.

Finally, we need to add the score of a special transition, that is, from the last tag to the $< Stop >$ tag.

3.2 Partition function

In general, this function is pretty much the same as computing score of a sentence. However, this function compute the complete all possible scores. In other words, it compute the sums of all possible scores, where for each word in the sentence, the function considers all its potential tags, and finally return a ”Total” score which adds up all potential tags score together.

Initially, we calculate transition from the $START$ tag to every potential tag of the first word in the sentence. Then, for each potential tag of the current word, we add up all the previous potential tags, and the emission score of current word with ”this” tag. When the scores of all the tags of current word are computed, we move forward with the next word in the sentence. Finally, at the last word, the function computes the transition score from each of the previous potential tags to the $STOP$ tag. Adding them together, and it returns the final overall scores. Now that we obtain the score of the sentence and the partition score, we can easily compute the negative log likelihood and we use it RNN to minimize the NLL so that our trained model approach to the training set.

4 Viterbi Algorithm

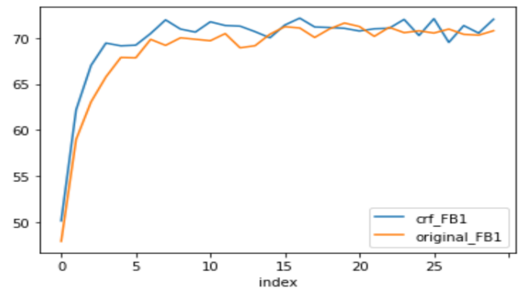
Compared with the greedy algorithm, viterbi algorithms considers more that it no longer consider the possibility of a tag on a specific word. Instead, it considers the overall possibility. For a very simple example, suppose we have a sequence of word that needs to be tagged, with only 2 tags and the word sequence length is 3, [I, love, CS]. Let say I has possibility of tag A is 0.4, and tag B is 0.6. Love has A 0.3, B 0.7. And CS A 0.9, B 0.1. Then for a greedy view, the result should be BBA. However, for viterbi algorithm, not only tag possibility will be considered, but also its previous tag should be considered. That is, given tag sequence, [B,B], we want to compute the possibility of next tag is A. Combine with A’s emission, viterbi, in general, at each word, considers its previous tag and a potential current tag possibility, then it times current word emission when choosing this potential tag. Thus we have the following formula, at each word, we have **transition[from prev, to curr] * emission(word | curr)**. Finally, when we compute the whole sentence’s possibility, we choose the overall largest possibility as our final decision.

The viterbi algorithm uses a dynamic program as it only record the largest possibility in the previous steps. For example, at time t with many predictions of tag O, such as IIO OOO IOO, etc. That is, II has an arrow points to O, OO has an arrow points to O, and IO also has an arrow points to O. The probabilities are 0.3 0.4 0.5, respectively. Then starting from current state O, no matter what tag we choose for it, the pre probability is already defined. That is, IIOI and OOOI and IOOI is predifined as 0.3 * porb of choosing I, 0.4 * prob of choosing I, and 0.5 * prob of choosing I. Since the post prob of these 3 candidates are the same, thus we can assert that 0.5 * prob of choosing I is the highest prob. As a result, there is no need to expand IIO and OOO. As a result, only IOO needs to be expanded at this time. As a result, in our hw case, each time, the previous step will provide 8 possible choices, then each choice expands 8 possibilities. Now that at time t, we have 8 possibilities to choose tag1, 8 possibilities to choose tag2 ... 8 possibilities to choose tag8, we keep the best choice of each one, and thus 8 choices remain. One for each, and we move them into the next step. Thus, the complexity would be the length of the sequence * the number of choices at each time $->$ 64*n, which becomes linear time in our case. This is what we called, dynamic programming, as it contains the best choices at each step and the other choices are killed, which saves lots of computations.

The Viterbi Algorithm is used to compute the highest score of the give sentence and return the argmax tags, respectively.

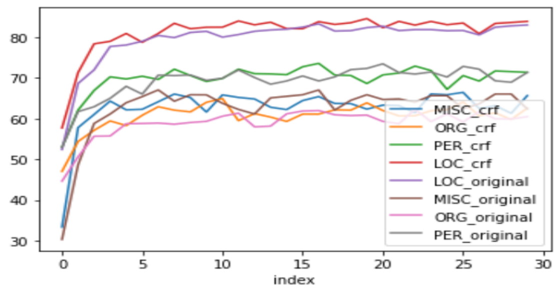
5 Results

The graph below is the comparison between LSTM with crf and the starter code.



Where the Blue line is the performance of crf. In fact, it does have some performance boost, from average around 70 to around 72.

Below is all the other comparisons among crf and the orignal tags.



We can find that tags with crf, in general, performs a little bit better than the original method.

6 Conclusion

In conclusion, crf's overall performance and each tag's performance are all slightly better than than the original method. However, without vectorization of crf, the it cost me around 2 hrs to perform the whole 30 epoches. Even with vectorization, it cost me around 30 mins. No matter which one, the original method only takes 2-3 mins. So, we are sacrificing too much time to exchange for a little bit performance increasing. Maybe we need to find other ways to increase the performance.