Assignment 3: POS tagging

Yongchao Wu
Maria Elena Gambardella

March 9, 2020

1 Naive Tagger

1.1 Learning algorithms, parameters and hyperparameters

The parameters are learned from the data thanks to the method `fit()` defined in the class `NaiveTagger`, which helps us to learn the data to later create a table of all frequencies. In order to achieve that, we create two different lists: one for the unique words (`_unique_words`) and one for the unique tags (`_unique_tags`). We then create two different look-up tables (`_unique_words_lut` and `_unique_tags_lut`), by using the two lists just created, where the word or the tag is the key and its index is the value. Finally, we create a matrix (`frequencies`) to keep count of the frequency for each word. This table shows our parameters, since the elements in it establish the feature table. Since the feature matrix is now fixed, so there is no hyperparameter provided.

1.2 Naive tagger prediciton algorithm

The prediction algorithm defined in the method `predict()` predicts the values for our y's (which are basically the yhats). What the `predict()` method does is taking each list inside the list `X` and it loops though each one of them in order to retrieve each word and compare it to the words present in the look-up table `_unique_words_lut` that we built in the fit method. If the word is in the `_unique_words_lut`, then the tag of that word is retrieved and appended in the list `tags` (which is going to be a list of all the corresponding tags for each word), whereas if the word is not in the `_unique_words_lut`, then the tag with the highest frequency (which is stored in the variable `majority_vote` in the beginning of the method) is appended to the list `tags`. All the tags for each sentence are then appended in another list named ret which is returned at the end of the method, so it can be used for the accuracy function.

1.3 Naive tagger discussion

Although this tagger apparently performs quite well in dealing with POS tagging problems (with accuracy over 94% in the lab), we should take into account the fact that this tagger does not consider any context. This means that every tag is assigned to a word without considering the neighbouring words and corresponding tag.

2 Hidden Markov Model Tagger

Hidden Markov Model(HMM) is a probabilistic sequence model. A sequence model assigns a sequence of labels to each observation in the input sequence. From a probabilistic point of view, HMM computes all probability for all possible sequences of labels and picks the best one. So, a HMM tagger processes sentences which are sequences of words and assign them to best sequences of POS tags.

2.1 learning algorithms, parameters and hyperparameters

In order to discuss HMM tagger learning algorithms, parameters and hyperparameters, and its key components should be illustrated. As discribed in table 1,

Table 1: Key components of HMM tagger

Components	Description
Tags($t_1 t_2 ... t_N$)	A set of N states(tags)
Words($w_1 w_2 ... w_T$)	A sequence of observations(words in sentence)
A($a_1 ... a_{ij} ... a_{NN}$)	Trasition probability matrix A, a_{ij} stands for transition probability from state i to j.
B($b_i(w_t)$)	Emission probability table B, $b_i(w_t)$ stands for probability of a word w_t generated from a tag t_i.
$\Pi(\pi_1, \pi_2, ..., \pi_N)$	Initial probability distribution over states(tags). π_i stands for probability that the Markov chain starts with state i.

The learning process of a HMM tagger is to achieve feature tables. Specifically, from the provided code, the function **def fit(self, X, y)** will take input X and y to achieve the trasition probability A and emission probability table B. After trainning the model with training data, these two feature matrixes/tables will be "learned". Thus, the parameters of HMM taggers are the feature tables A and B. Since the feature matrix/table is fixed, there is no hyperparameters provided.

2.2 HMM tagger prediction algorithms

Given a learned HMM model with transition probability matrix A and emission probability table B, HMM (A, B)) and a sentence noted as w_1^n, the predicted sequence of tags according to a bigram tagger $\hat{t}_1^n$ follows the equation:

$$\hat{t}_1^n = \arg\max_{t_1^n} P(t_1^n | w_1^n) \approx \arg\max_{t_1^n} \prod_{i=1}^{n} P(w_i | t_i) P(t_i | t_{i-1}) \tag{1}$$

where $P(w_i|t_i)$ is the emission probability which can be found in table B and $P(t_i|t_{i-1})$ is the transition probability from state(tag) t_{i-1} to t_i which can be found in table A.

From the provided code, the Viterbi algorithm is implemented. The intuition behind Viterbi algorithm is similar to dynamic programming. The aim of the Viterbi algorithm is to find the best path that indicates a sequence of tags. In the algorithm, a Viterbi matrix $V \in \mathbb{R}^{N \times T}$ is established, where N is the number of states(tags) and T is the observations(words) length. Each cell $v_t(j)$ represents the probability that the state is j after t observations and best state sequence $t_1, ...t_{t-1}$. This probability value is computed by taking best path to this cell recursively, as other dynamic programming, a record of previous computation is reserved.

$$v_t(j) = \max_{t_1^{t-1}} P(t_1^{t-1}, w_1^t, t_t|(A, B)) \tag{2}$$

Suppose we are, at $t-1$ step, v_{t-1} has been already computed, at step t, $v_t(j)$ can be computed as:

$$v_t(j) = \max_{i=1}^{N} v_{t-1}(i) a_{ij} b_j(w_t) \tag{3}$$

Again, transition probability a_{ij} and emission probability $b_j(w_t)$ can be referred to table A and B.

2.3 HMM tagger discussion

Overall, Bigram HMM performs quite well in dealing with POS tagging problems (with accuracy over 94% in the lab). One possible solution to increase the accuracy is to use trigram or "bigger gram" model. However, the computation complexity would increase and much larger feature tables should be maintained. Thus, the short coming for HMM is the constrains of the context from which the information can be extracted.

3 LSTM based tagger

Long Short-Term Memory (LSTM)[1] is a special type of Recurrent Neural Network (RNN). A RNN is a network that contains a cycle within the network connections which means one recurrent connection is included in the hidden layer. Thus, the activation value of the hidden layer depends both on the current input and the activation value of the hidden layer of a previous time step. In simple RNN, there is problem of making use of larger context, which means the information the output of the network tends to be decided by information in a sequence of near distant. LSTM is introduced to address this problem by context management mechanism.

3.1 learning algorithms, parameters and hyperparameters

For simple RNN, suppose we have a sequence of input is $w_1 ... w_t ... w_T$, at time step t, the hidden layer activation value h_t depens on the current input x_t and the hidden layer activation value at previous time step $t-1$ with g as activation function:

$$h_t = g(Uh_{t-1} + Wx_t) \tag{4}$$

and the output vector with f as activation function is:

$$y_t = f(Vh_t) \tag{5}$$

to apply the softmax function over the output vector:

$$y_t = softmax(Vh_t) \tag{6}$$

where, W, U, V are weight matrix that are shared across time. Thus, the model should learn U, V, W of the network from the training data. The process of learning these weight matrixes is similar to Neural Network: through a loss function and backpropagation(gradients descent) to adjust weights in the simple RNN network.

Based on the simple RNN, LSTM adds a context layer that deploys 'gates' to manage context information. These gates have addtional weights and and process the input, previous hidden layer and previous context layers. In the gates, there are sigmoid activation functions that can make binary output 1 or 0. Output 1 means the context information should be kept while output 0 means it should be dropped.

The first gate that we have is a "forget gate". Its purpose is to erase some context information. Specifically, it will process the previous hidden layer and current input through extra weights and sigmoid to get the "gate vector" output and then multiply it by context vector to remove unnecessary information.

$$\begin{aligned} forget_t &= \sigma(U_{forgeth}ht - 1 + W_{forget}x_t) \\ k_t &= c_{t-1}forget_t \end{aligned} \tag{7}$$

Next, the job is to compute the information from previous hidden state and current inputs, and then multiply by a "add gate" to add the information into current context:

$$\begin{aligned} g_t &= \tanh(U_g h_{t-1} + W_g x_t) \\ add_t &= \sigma(U_{add}h_{t-1} + W_{add}x_t) \\ j_t &= g_t add_t \\ c_t &= j_t + k_t \end{aligned} \tag{8}$$

Next, another "output gate" is added to control the output information for the current hidden state:

$$\begin{aligned} output_t &= \sigma(U_{outputh}ht - 1 + W_{output}x_t) \\ h_t &= output_t \tanh(c_t) \end{aligned} \tag{9}$$

From the above explanations, we know that basic weights U, W, V together with context layer with extral weights $U_{forget}, W_{forget}, U_{add}, W_{add}, U_{output}, W_{output}$ are also needed to be learned by training. And the learning process is similar to simple RNN, through a loss function and backpropagation (gradient descent) to adjust weights.

After understanding the learning algorithms, the parameters and hyperparameters are then easier to explain. The parameters of the LSTM tagger are:

$$U, W, V, U_{forget}, W_{forget}, U_{add}, W_{add}, U_{output}, W_{output} \tag{10}$$

the hyperparameters are those that can be tuned to get a better accuracy, like EMBEDDING dimension, HIDDEN dimension loss function, regularization and so on.

3.2 LSTM tagger prediction algorithms

The training process is complex, meanwhile the prediction of LSTM is similar to regular multiclass classification job. Given a learned network, we can follow the process below:

$$h_t = g(Uh_{t-1} + Wx_t) \tag{11}$$

$$y_t = f(Vh_t) \tag{12}$$

$$y_t = softmax(Vh_t) \tag{13}$$

A softmax will help to pick the best result from the output and make a prediction.

3.3 LSTM tagger discussion

As showed in the lab, the LSTM result is underwhelming due to training data size limitations and the fact that they are not tuned. We can see that, LSTM introduced a number of parameters, which will bring high traing cost. To decrease the training cost, one idea is to reduce the number of gates from LSTM, thus Gated Recurrent Units(GRUs)[2] were introduced.Another idea is to reply on attention mechanism to establish global dependencies between input and output[3] .

Reference

[1] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[2] Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, Doha, Qatar, October 2014. Association for Computational Linguistics.

[3] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is All you Need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 5998–6008. Curran Associates, Inc., 2017.