CS134-PA3 Hidden Markov Model

November 7, 2016

Hidden Markov Model (HMM) is a powerful model for a lot NLP tasks because languages can be easily modeled as sequences. The Markov assumption makes the computation during training and inference very efficient. For this assignment, you will be implementing the forward algorithm for computing observation likelihood, the viterbi algorithm for inferring the best label sequence given a sequence of observations, and the training function for fitting Hidden Markov Model parameters. And for the brave among you, the forward-backward (Baum-Welch) algorithm for learning the parameters without the labeled data (unsupervised learning) and in combination with the labeled data (semi-supervised learning).

1 Task: Noun Phrase Chunking with HMM

Your implementation should be generic enough to run on any dataset. But to give you an opportunity to apply your implementation to the real data, we will use HMM to do noun phrase chunking.

Those of you who have taken computational linguistics before can skip this paragraph and go ahead and start implementing. You should be familiar with noun phrase chunking and shallow parsing. For those who come from the computer science background, you might be familiar with the idea of parsing as an expression tree, which is done by compilers. Noun phrase chunking is also known as shallow parsing because it does not induce a tree structure from a given sentence. You will learn later this semester that deep parsing for natural language is difficult to implement, awkward to handle, and expensive to process. Rather we end up with spans demarcating where the noun phrases are. This task can be done with HMM where the hidden states are B(eginning of the noun phrase), I(n the middle of the noun phrase), and O(thers), and the observations are the words and their features. This notation is commonly used in a lot of NLP tasks and are often called *BIO tagging* or *IOB tagging*. For example,

```
fruit flies like bananas
  B     I    O     B
```

We can see that the sentence is no longer syntactically ambiguous after the parse. And we know exactly how many noun phrases are in the sentence. Shallow parsing is substantially easier and cheaper than deep parsing (making a parse tree) and is sometimes sufficient for the task at hand e.g. information extraction (who likes bananas?).

For more detail about noun phrase chunking, please read more about it on chapter 13.5.2 of Martin and Jurafsky. They delineate the theory and practice of noun phrase chunking nicely.

2 Implementation Requirement

2.1 HMM Class

I. Constructor

```
In [ ]: def __init__(self)
```

Initialize any necessary data structures here.

II. Training function

```
In [ ]: def train(self, instance_list)
```

This function should call **_collect_counts** and derive the parameters from the counts. Note that your data instance should now be a sequence of sparse feature vectors. You have to also update the codebooks properly. You can treat every feature as binary. Like Naive Bayes, the model also benefits from smoothing and/or feature selection to some extent. So you are required to do some smoothing during training as well.

III. Collecting counts for fitting parameters

```
In [ ]: def _collect_counts(self, instance_list)
```

This function updates the count tables which are initialized and stored internally in **self.transition_count** and **self.feature_count_table**. It should not return anything.

IV. Forward algorithm and Viterbi decoding algorithm

```
In [ ]: def dynamic_programming_on_trellis(self, instance, run_forward_alg=True)
```

As we've learned in class, the forward algorithm and the Viterbi algorithm are largely the same. In the forward algorithm, the sum operation is used over the previous time step. But in the Viterbi algorithm, the max operation is used. In this implementation, you will convince yourself that this is indeed true. The function will traverse through the trellis and fill it with the appropriate values. You should avoid for loop as much as possible here for efficiency.

When run the forward algorithm model (**run_forward_alg=True**), the function only has to fill up trellis. When run in Viterbi mode (**run_forward_alg=False**), the function has to fill up both **trellis** and **backtrace_pointers** . The function then returns a tuple consisting of the trellis and the backtrace pointers. They should both be m $\times$ t numpy arrays, where m is the number of hidden states, and t is the number of tokens in the sentence.

V. Inferring the best label sequence for a given sequence of observations

```
In [ ]: def classify_instance(self, instance)
```

The only thing you have to do in this function is do use the filled-up trellis and backtrace pointer matrix to recover the best sequence. More precisely, we want to return

$$argmax P(q_1, ..., q_n | o_1, ..., o_n) \tag{1}$$

This operation should be done in linear time since the necessary computation has already been done by the Viterbi algorithm.

VI. (EXTRA CREDIT) Semi-supervised and unsupervised learning for HMM. We will give out generous extra credit for implementing forward-backward algorithm if the implementation is complete and mostly correct.

(a) Forward-backward algorithm

```
In [ ]: def _run_forward_backward(self, instance)
```

This function runs the forward algorithm to obtain the α table and runs the forward algorithm backward to obtain the β table using the notation in the textbook and the slides. You should definitely call the forward algorithm function to accomplish this.

(b) Semi-supervised learning

```
In [ ]: def train_semisupervised(self, unlabeled_instance_list, labeled_instance_list=None)
```

This function implements the Expectation-Maximization (EM) algorithm and wraps around the forward-backward algorithm. It is partially implemented. You need to implement the main three steps in the EM algorithm * Initialization step. Initialize the model uniformly when the labeled data is not provided at all (unsupervised learning). If the labeled data is provided, the model should be initialized with it. * E-Step. Update the (expected) count tables based on the results from the forward-backward algorithm. * M-Step. Combine the expected count tables from the unlabeled data with the observed count tables from the labeled data. Use the combined count tables to reestimate the parameters.

The implementation should emphasize further the importance of efficiency. The EM algorithm is iterative. At each iteration, we need to perform inference on every data point. That's a lot of for loops. If we are not careful, the algorithm becomes impractical.

(c) Determine convergence

```
In [ ]: def _has_converged(self, old_likelihood, likehood)
```

There are a number of ways to determine convergence. Comparing old and new loglike- lihood works quite well in practice.

2.2 Evaluation functions

1. Compute confusion matrix in **evaluator.py**

```
In [ ]: def compute_cm(classifier, test_data)
```

the good news is that this function is fully implemented for you. It will make predictions for the test data and compute the confusion matrix and print it out nicely. It also provides the functions for computing F1 (for each label) and overall accuracy.

> **However,** you still have to modify this function since the **ConfusionMatrix** class requires two parameters: index2label and label2index, basically the classifier should have the two-way mapping from the label string to label index and vice versa. You're also free to implement your own evaluation method.

2.3 Experiments and report

Write a separate script that reads in the training data and train the following HMM models: - 1) HMM with part-of-speech as features, - 2) HMM with words themselves as features, and - 3) HMM with part-of-speech and words as features.

Test these models on the test set and report the result using the functionality provided in ConfusionMatrix class. Analyze the results and include at least one graph or table to accompany your analysis. Describe why certain feature sets might be more useful than others. Should smoothing make a difference at all? Additionally, look at the errors (misclassification) that your best classifier makes. Point out one systematic error and suggest how you can improve the classifier based on this knowledge (e.g. new classes of features, new model, weaker model assumption, etc.) Your write up should be around 1-2 pages single-spaced.

```
In [ ]:
```