Project Report for Assignment 2

B553: Probabilistic Graphical Models

Chaitanya Bilgikar (cbilgika) and Nasheed Moiz (nmoiz)

Step 1: Build up initial Conditional Probability Distributions

In order to do this, we initialize a file reader, that splits the entire content of the corpus by " . . \n", then what we have is a sentence and the tags of its words. Then from each sentence we store in a dictionary the data from each sentence in the format *{index: (word, tag), index: (word, tag)},* then each dictionary representing a sentence is added to a list called *main_list* which is a list of dictionaries. We discussed various alternative options to store the data for future calculation but arrived at this format because it is relatively simple and it makes it easy to access the information we need. The method that does this is called *get_word_list*.

Consequently, we start counting values from this list to build up probability distributions for *P (W), P(S*$_1$*),* and *P(S*$_{i+1}$ *| S*$_i$*).* After initializing dictionaries to store the various CPDs, we iterate through the *main_list* (described above) which has all the sentences, incrementing the respective variables counting variables such as *tag_count*, *word_count*, *total_word* count, occurrences of words as specific tags, and occurrences of specific tags adjacent to other specific tags as we go. Once this iteration is over, we divide these counts by their corresponding total count values to get a probability value as opposed to an integer count.

Step 2: Implement a Naive Bayes Classifier

In this step, we use the probability distributions created in the last step. First we aggregate the data in the format we like. Even though the word, and its true tag value is stored, we do not use it to evaluate, but it comes in handy later on to evaluate the accuracy of our algorithm's own tag assignments. Then as we begin assess each sentence and iterate through it word by word, we create a temporary dictionary that store the probabilities of each word being any of the 12 tags. The Bayesian theorem is used in this stage. From these 12 values for each word, the max is assigned. These assignments are stored in a simple dictionary, with the word as key, and its assigned tag as value.

Step 3: Implement Variable Elimination algorithm to evaluate

Since the Bayesian network described in Fig 1 of the assignment document is in the form of a Markov chain, we have used the forward-backward algorithm to evaluate tags. The meat of this algorithm is enclosed in an outer for loop, which iterates through each of the sentences in the testing data. This outer most for loop has two inner for loops, one for the forward part and one for the backward part. The former starts from first word of the sentence and ends at the last word (1 to n), while the backward does the reverse. Inside each loop, the first (or last) word of the sentence is handled differently than the other words, since the tag node (S1) for first word has no parents.

Inside these innermost loops, we setup *temp_tau*, a dictionary, and add it to the dictionary of dictionaries called *tau_f* for the forward part, and *tau_r* for the reverse part (see code for description). *temp_tau* is a local dictionary that stores probability distributions of any given word being any of the 12 tags. This is calculated by using the CPDs, *P (word | tag)* and *P* $(S_{i+1} \mid S_i)$ from the learning part. After calculating *tau_f* and *tau_r*, we multiply corresponding elements from them to create a new dictionary of similar structure, which stores the products. Then from tau we simply extract the tag with the max value. This value is the predicted value of the word.

Step 4: Sampling

The "sampling" method contains the implementation of the sampling part of the assignment. This method takes a *tau*, which is a dictionary, where each key has a value, which in turn is a dictionary of the format:

{word$_1$*: {tag*$_1$*: *some prob value*, tag*$_2$*: *some prob value*...up to tag*$_{12}$*}, word*$_2$ *: {tag*$_1$*: *some prob value*, tag*$_2$*: *some prob value*...}......up to word*$_n$*}.*

These probability values have already incorporated both the forward and backward part of the forward-backward algorithm implemented in *max_marginal*. In fact, "sampling" is repeatedly called from inside *max_marginal* along with the tau calculated in it. Sampling performs the last step of the inference differently from *max_marginal* in that instead of picking the tag with the maximum probability value for each word, we do a randomly weighted selection of tags. This selection is done using the *random.choice* function in python. Note that the tag of the first word is computed as the most probable tag based on the output of the *max_marginal* function.

One problem we had while implementing this was to scale up the probability values for them to be selected by the random selector.

Step 5: Evaluation

From the main we call the methods *naive_bayes*, *max_marginal* and sampling to initiate the three types of evaluation. It is to be noted however that all three evaluations are not simultaneously conducted on each sentence from the testing set, as shown in the output of the assignment document. For instance a call to *naive_bayes* will perform Naive Bayes classification on the entire testing set, not just one sentence.

Additionally, the main method also includes declarations of the option parser, options for which are described below.

Observations

The accuracies for our Naive Bayes classifier were in the high 70s - low 80s range which we taught was fair based on what we saw on the sample output provided on the assignment.pdf. For the marginal max, we observed that accuracy was strangely a tiny bit higher without the backward part of the forward-backward algorithm implemented (wonder what the reason could be for this, probably the structure of the network that we have used). Note that the efficiency of sampling depends on the prime number that we are using to hash the integer values in the function *sample.*

How to run the code

The file a2.py can be run as

```
python a2.py [options]
```

Options can be:
-c m : running max marginal inference
-c n : running Naive Bayes inference
-h : help on how to run the code

By default, giving no options runs the Naïve Bayes classifier on the data.