Jared von Halle

CSC 594 NLP

HW#3

\*\*\**Jared von Halle and David Chang worked on this together. We have submitted the same code, output, and a similar write-up. The write-up is the same, except for the final section on Overall Comments, which we each completed separately.*

## Instructions

Pre-requisites to running:

1. Python 3 framework must be installed
2. The main python file, ‘hw3.py’ should be in the same directory as any input files

Steps

1. Open a command prompt and navigate to the directory containing the ‘hw3.py’ file
2. Add input files to that folder. This includes the training data as well as the test data.
3. Enter the following command, substituting the name of files for the arguments in brackets:

**python hw3.py <trainFile> <testFile>**

Note, the program may take a couple minutes to complete, depending on the size of the training and test data (mostly dependent on the test data). The output will be written to **output.txt**

## What We did

The “main” function parses the file names (i.e. train, test) from the command line parameters. It then calls a function, *parseTrainingText*, to populate the probabilities needed for the Viterbi matrix.

The first step in calculating probabilities is to obtain the count of each tag, and each word in the training data. Note that we include “start” tags, in order to account for the start of each sentence.

Next, we calculate the probabilities for each word/tag pair. This is done by dividing the number of times each word/tag combination appears in the training set, by the number of times the tag appears in the training set. The results are stored in a dictionary, where the key is the word/tag tuple and the value is the probability.

A similar process is done to calculate the transition probabilities, or in other words, the probability that one tag follows another. For each pair of tags that appear next to each other in the training set, we divide the total number of times that pair was observed, by the number of times that the first tag appeared. Again, we store in a dictionary, where the tag/tag tuple is the key and the probability is the value.

The code then contains some logic used to calculate overall accuracies, and to deal with word/tag pairs and tag/tag pairs that we observe in the test data, but that don’t appear in the training data. We’ll get back to this point later.

The next step is to create the Viterbi matrix. We populate this by considering sentences from the test data one at a time. An array of unique tags provides us a consistent ordering of the tags to use for one dimension of the matrix.

The matrix is created using 3 nested loops. The outermost loop goes through the tags. The next layer goes through the words in the sentence. Finally, the innermost loop goes through the tag array again, for the purpose of considering different prior tags for the transition probabilities. Within that loop, we also keep track of the maximum previous forward path probability, and the index from which that maximum came from. This allows us to backtrack through the matrix in order to find the optimum prediction for tags.

The backtracking occurs by first finding the maximum probability of the final column in the matrix. We use the indices that were stored in each column to populate the array of predicted tags.

Finally, we compare the predicted tags to the actual results, keeping track of the number correct. After calculating the final accuracies, we write out that value, along with the prediction and actual tag for each word in the test set.

## Handling 0 probabilities

There are two cases where we were initially left calculating a 0 probability. The first was when a tag transition (e.g. IN|DT) was observed in the test set, but never in the training set. The second was when a word/tag pair (e.g. DT|word) was observed in the test set but not in the training set.

The first thing we did to help with this was to use logarithmic probabilities. When using regular probabilities (non-logarithmic), you multiply the probabilities of the two cases described above together. This means that if one of those cases is a 0 probability, then the result of multiplying them will also be 0. However, when we switched to log probabilities, we add those values together. Now a 0 value for one of the cases does not affect the other case. Even further, we use those probabilities in calculating probabilities for other Viterbi Matrix paths. Again, the addition instead of multiplication helps to limit the cascading affect of 0 probabilities.

The second, and more impactful, approach that we took was to use a default value other than 0 for the cases where word/tag or tag/tag pairs appeared in the test set but not the training set. The goal was to set these cases to a small probability, but not 0. Therefore, we started with the minimum of all probabilities in each case. For example, if the transition IN|DT was observed in the test data, but not the training data, we set it to the minimum probability that was found for all tag/tag pairs that did occur in the training set. This raised the overall accuracy to ~91%. The final step we took was to choose a value even lower than the minimum for each case. Specifically, we multiplied the minimum value by 2 to achieve our default. Note that since the minimum logarithmic probability was always less than 0, the affect of multiplying by 2 was always to decrease the probability. This raised our overall accuracy to ~94%.

## Results

On the WSJ-test.txt file, we achieved over 94% accuracy. It is interesting to look at some specific cases where we made the wrong prediction. We did not conduct a rigorous analysis of the numbers, but scanning through the results, it looks like many of the errors were on words from the test set that didn’t appear in the training set. For example, we misclassified the word “jetliners”, which did not appear in the training data. However, the word “jetliner” did appear in the training data. If we could have made that connection that “jetliner” and “jetliners” are somehow related, perhaps that could improve accuracy.

We also ran the our program using the held out data, “POS-test”. The results are in the file, “POS output”. Note, we had to modify the code slightly to handle this case where the correct tags are not provided along with the test data. Therefore, running the “POS-test” through our program as it is submitted, will not work.

## Overall Comments