Natural Language Processing Tagging Assignment

Edward Fleri Soler

247696(M)

University of Malta

E-mail: edwardfsoler@gmail.com

Abstract

In this paper, we shall implement a Viterbi tagging algorithm to tag input tokens based off of training data from a tagged corpus. We shall observe the procedure involved in generating probabilistic estimates for a given token based on the previously tagged token and the current observation. We shall then evaluate the performance of this algorithm, and observe the effects that different training data sizes have on accuracy. This algorithm shall be implemented in the Python programming language.

1. Introduction

The Viterbi Algorithm is an algorithm which forms part of the Hidden Markov Model family, and implements dynamic programming principles to select and assign tags to tokens. This algorithm takes into account the probability of a specific observation occurring in a given state, and the probability of transition from one state to the next.

This algorithm is very similar to the Forward HMM algorithm, however is computable in order $O(N^2T)$, as opposed to the $O(N^T)$ of the forward algorithm. This makes this approach much more viable, especially for a larger number of states and observations.

We shall first understand the manner in which this algorithm is implemented, following with an evaluation of the algorithms performance. We shall also discuss the effects of different training data sizes on the input.

2. Implementation

2.1 Data preparation

The first step in the implementation of this algorithm is the preparation of a data set to be used to train and test the system. For the purpose of this assignment, it was decided that the CoNLL 2000 Chunking Data corpus was of the right size and type for this implementation. This corpus is consists of over 200,000 tagged words, which would serve to train and test the system.

The words from this corpus were extracted in a pre-tagged layout, using the NLTK universal tag-set. This tag-set consists of 11 possible tags in their most general form, and does not handle plurality or tense, greatly reducing the possible states which the system must account for.

A **datasets** function was defined to initialise the training and testing data sets. This function takes no arguments and returns two lists. The abovementioned corpus is imported and implemented through the **tagged_words** function, which returns a list of all the words in the corpus, tagged in the universal manner. N-fold validation was used to split the data set into two parts, with 90% assigned for training, and 10% for testing. This process involved the use of random numbers to randomly segment the data set. The two lists were then returned to be later used.

2.2 Training Algorithm

In order to implement the Viterbi algorithm, a dataset containing probabilistic data for the likelihood of a given token being assigned a certain tag was required. The **unigramtrainingset** function was defined to take the training set as an argument. This function then generates a list of tuples, matching each tag with the probability of being assigned to a given token. Therefore, the returned list consisted of a nested tuple-list object, matching each word with the set of tags, which in turn were matched to their probabilistic likelihood as follows:

('policy', [('NOUN', 1.0), ('ADP', 0.0), ('ADV', 0.0), ('NUM', 0.0), ('VERB', 0.0), ('.', 0.0), ('PRON', 0.0), ('DET', 0.0), ('ADJ', 0.0), ('PRT', 0.0), ('CONJ', 0.0)])

This result was obtained by means of the **FreqDist** function, which was used to calculate the frequency of a given tagged word within the dataset. The frequency of each word-tag combination was divided by the word frequency (paying no attention to the tag assigned) to produce the final probabilistic outcome. This result will later be used by the Viterbi algorithm

Another data set required to implement the tagging system is the tag-bigram probability, which rates the probability of a given tag following another. The **bigramTags** function generated a list of all tag bigrams within the data set, and proceeded to calculate the frequency of each tag pair. This was once again reduced to a probabilistic value which was mapped to each bigram and output in a list:

(('NOUN', 'NOUN'), 0.07384073840738407) , (('NOUN', 'ADP'), 0.05393053930539305), …

In total, 121 possible tag bigrams were listed within the list (11^2), together with their respective probability.

2.3 Tagging Algorithm

Once the data sets have been processed and prepared, the implementation of the Viterbi algorithm is all that remains. As was introduced above, the Viterbi algorithm takes into account the likelihood of a given path being the correct one.

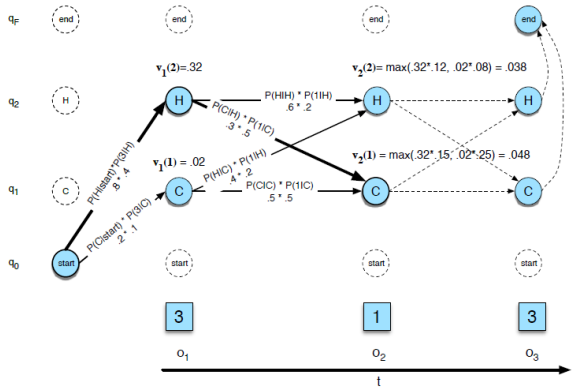

Figure 1. Demonstration of the Viterbi algorithm on a dataset considering frequency of ice creams and weather conditions

Starting at a start state, and ending up in a final state, the Viterbi algorithm selects one of a number of possible paths through a search space by calculating the probability of a given observation (token 'walk') existing in given state (tag 'NOUN') as well as the probability of moving from one state to another ('VERB' to 'NOUN'). These calculations, together with the summation of all previous steps is used to find the most probable path in a search space.

The **viterbiTagger** function accepts a list of bigram tags computed by the **bigramTags** function, a list of unigram words computed by the **unigramtrainingset** function, and the text input to tag.

The first step in this process requires the input text to be tokenised and stored as the observations for the path finding algorithm. Next, two 2-dimensional matrix lists are

defined to store state probabilities and back pointers to the previous state. These lists are initialised with zeros. We now start assigning tags to each token.

Starting with the first token, start by retrieving the word tag-probability data for the current observation (token). Next, we loop through all possible states **s** for the current token, storing this probability in the first column, with each row containing the probability that the given observation is tagged with s specific tag. We then set the back pointer to point to 0, which acts as the start state. With this, the first word is tagged.

Next, we enter a loop which cycles through the remainder of the input tokens, implementing the same process on each token. This process again starts by retrieving the word tag-probability pair from the passed parameter, and looping through the set of possible states. However, before looping, a test is implemented to see whether the current word has indeed been encountered during training. If this is so, control flows as usual; otherwise, this token is tagged with a default tag, so as not to interrupt the control flow. This tag was defined to be a 'NOUN' tag, as it has been observed to be the most common in the data set.

The inner loop, mentioned above, cycles through the possible states of the current observation. This loop contains yet another nested loop, which again cycles to the set of possible states, but this time considering the previous observation. This nested manner gives the algorithm its $O(N^2T)$ running time. Each possible state of the previous observation is matched with the possible state of the current observation, to create a tag bigram. Each tag bigram probability is then multiplied by the final outcome of the given state of the previous observation. These values are all stored in a list.

Outside of the inner most loop, this list is tested for the maximum value, which defines the fittest path to take, in turn definitively deciding the state for the previous observation. This maximum value is further multiplied by the probability of the given observation lying in any possible state, ultimately deciding the likelihood of any tag being assigned to the current word based on the word itself and the word prior to this one.

Finally, the back pointer is set to point to the previous state which was selected above. This back pointer shall be used to traverse through the states again, returning the best path.

This procedure is implemented for all remaining observations constantly surveying all possible paths and selecting the best one. This results in the final column in the 2-dimensional matrix being filled. We are now required to return the path (tags). This process involves traversing the path in reverse by means of the back pointer, adding each state to the front of a list. The list is populated with a given tag based on the row to which the back pointer points to. Upon the complete reversal of this list, an ordered list of tags is returned:

['DET', '.', 'NUM', 'NOUN', 'DET', 'NOUN', '.', 'VERB', 'VERB', '.', 'NOUN', 'ADP', …]

2.4 Bringing it together

The individual functions mentioned above were all thoroughly debugged and tested before stitching them together by means of an intermediary function. The **tag** function takes a string as input, which shall be tagged by the algorithm. This function starts by creating the training and testing data sets through the **datasets** function, and follows by reducing the training data set to a given size, for later testing purposes.

The bigram tag probabilities are then computed through call to the **bigramTags**

function, followed by the computation of the unigram-tag probabilities by calling the **unigramtrainingset** function. Finally, these lists are passed onto the **viterbiTagger** function together with the input, whose result is returned to be output on screen.

Outside of this function, evaluation of the returned tags shall take place, whereby the tags assigned by the algorithm could be compared to the tags assigned by the corpus, to better understand the accuracy of this system. The size of the training set may also be adjusted to understand its effects and improve the accuracy of the system.

3. Evaluation

We shall start by extracting a subset of the training set for testing. The first test shall involve a test input of 200 words and a training data set of 100,000 words. These data sets a derived from the same corpus but do not intersect:

Test 1

Test input size: 200 words

Training data size: 100,000 words

Test string: "a $ 1 toll each way, does have `` loss of income '' insurance to replace lost revenue if the operation of the bridge is interrupted for more than seven days …."

Result: 0.97

Test 2

Test input size: 1,000 words

Training data size: 100,000 words

Test string: "analysis of the compatibility of Wheeler Group's marketing business with other Pitney Bowes operations. Pitney Bowes acquired the core of what evolved into Wheeler Group in 1979 by…"

Result: 0.38661

Test 3

Test input size: 200 words

Training data size: 10,000 words

Test string: "revive the buy-out. Pilot union Chairman Frederick C. Dubinsky said advisers to UAL management and the union will begin meeting in New York today and will work…"

Result: 0.935

Test 4

Test input size: 1000 words

Training data size: 10,000 words

Test string: "1987: Prime Minister Gandhi tells the Indian Parliament, `` ... neither I nor any member of my family has received any consideration in these transactions. That is the…"

Result: 0.29341

Test 5

Test input size: 10% of corpus (~ 27,000 words)

Training data size: 200,000 words

Test string: "again with the same price. " Not-Held Order: This is another timing order. It is a market order that allows floor brokers to take more time to buy or sell an investment , if they…"

Result: 0.2820

Test 6

Test input size: 1,000 words

Training data size: 200,000 words

Test string: “separated from the double-decker roadbed, that was responsible for the collapse. The failure in Oakland of the freeway segment known as the Cypress structure was…”

Result: 0.94

4. Conclusion

Observing the above results, it is apparent that the system performs its designed role but is not without flaws. In 3 of the 6 above tests, the outcome was a +90% accuracy rating. All these cases therefore hold an error rate of roughly 1 out of every 13-17 tags. However, the remaining 3 tests scored low accuracy ratings between 28% and 40%. On observation of the results, including the training and test sizes, one may realise that a trend is present. It is apparent that upon the introduction of large test samples, the error rate drops tremendously. However, for smaller test samples, no matter the training data size, the result is always accurate.

This pitfall is due to the manner in which this algorithm operates. Since a look back of only one observation is performed, one single error could ruin the tagging capabilities of the system, throwing the probabilities off balance and entering a vicious circle, were bad results lead to worse results, turning the system into a random guesser. This occurs whenever a previously unencountered word is present. Whenever an input which was not come across in the training is present, the system assigns the default ‘NOUN’ tag. It has been found that the ‘NOUN’ type is correct by default around 30%-45% of the time. However, if this is not the case, then the system loses accuracy and consistency, resulting in a poor result.

In future implementation, this could be resolved by considering trigrams in the place of bigrams, to base the next tag on a broader domain of values. This would allow the system to better handle unknown words, rather than referring to the default tag. A larger training data set would also be of help, especially if the training set contains a greater variety of words, greatly increasing the dictionary size, and therefore, the recognition rate of the system.

However, the down side of a larger training data set is the increased running time. On average, the system takes from 10 seconds up to 5 minutes to return an output, depending on the set size of the training set. This is due to the O(NT) complexity of the unigram tag probability function, were N is the number of possible tags (11) and T is the training set size. This means that roughly 2.5 million steps are involved in computing the probability of each tag for a training set of 200,000 words. Furthermore, the $O(N^2T)$ complexity of the Viterbi algorithm, were N is the tag size and T is the test input, adds a further 100,000 steps on average.

Therefore it may be concluded, that a training data set size of roughly 100,000 words, and a test input limited to 500 words is recommended for satisfactory tagging rates to be achieved.

5. References

Hidden Markov Models.

Daniel Jurafsky & James H. Martin

Speech and Language Processing

(2014)

Natural Language Toolkit Manual

NLTK 3.0 Documentation

http://www.nltk.org/