Handed out: 2 Oct 2019 **Due:** 13 Nov 2019 (start of lecture)

In this project, you will implement the hidden Markov model's Viterbi algorithm and forward-backward algorithm, and apply them both to Twitter tweets and intraday stock prices. Email your submission to Lim Suang (e0319343@u.nus.edu; prefix your email's subject with [BT3102 Project]). Make sure to answer all the questions, and to attach your code, generated files, and answers.

Question 1. Group Formation (5 points)

This project is to be done in a group of three. Form your group and email Lim Suang with your group members' names by 9 Oct 2019.

1 POS Tagging of Tweets

Twitter (`www.twitter.com`) is a popular microblogging site that contains a large amount of user-generated posts, each called a tweet. By analyzing the sentiment embedded in tweets, we gain precious insights into commercial interests such as the popularity of a brand or the general belief in the viability of a stock[1]. (Many companies earn revenue from providing such tweet sentiment analysis.) In such sentiment analysis, a typical upstream step is the part-of-speech (POS) tagging of tweets, which generates POS tags that are essential for other natural language processing steps downstream.

You are required to build a POS tagging system using the hidden Markov model (HMM). (Please implement your code generically so that it can be easily reused for the next task on intraday stock prices. It would be a really bad (time-consuming, gut-wrenching) idea to implement another HMM for the next task. The files you require are:

(i) `twitter_train.txt`,

(ii) `twitter_train_no_tag.txt`,

(iii) `twitter_dev_no_tag.txt`,

(iv) `twitter_dev_ans.txt`,

(v) `twitter_tags.txt`, and

(vi) `hmm.py`.

`twitter_train.txt` is a labelled training set. Each non-empty line of the labeled data contains one token and its associated POS tag separated by a tab. An empty line separates sentences (tweets). Below is an example with two tweets (there is an empty line after "?? ,").

[1] https://ieeexplore.ieee.org/document/8455771

```
rt ~
@USER_123d5421 @
I O
like V
IPhones N

@USER_2346f3f51 @
wat O
wearin V
4 P
the D
party N
?? ,
```

Both `twitter_dev_no_tag.txt` and `twitter_dev_ans.txt` have the same format as `twitter_train.txt`, except that the former does not contain tags and the latter does not contain tokens. `twitter_dev_ans.txt` contains the tags of the corresponding tokens in `twitter_dev_no_tag.txt`. Both files are used to evaluate your code. (We will evaluate your code with more data from the same domain.) `twitter_train_no_tag.txt` contains the same data as `twitter_train.txt`, but without any tags. `twitter_train*.txt` and `twitter_dev*.txt` contain 1101 and 99 tweets respectively.

`twitter_tags.txt` contains the full set of 25 possible tags. `hmm.py` contains skeleton code, and you have to write your code in this file.

Recall that the joint likelihood of the observed data $x_1, x_2, \dots, x_n$ and their associated tags $y_1, y_2, \dots, y_n$ is given by an HMM as:

$$P(x_1, x_2, \dots, x_n, y_1, y_2, \dots, y_n) = \prod_{t=0}^{n} a_{y_t, y_{t+1}} \prod_{t=1}^{n} b_{y_t}(x_t)$$

where y_0 and y_{n+1} are the START (*) and STOP states respectively, $a_{i,j} = P(y_t = j | y_{t-1} = i)$ is the transition probability, and $b_j(w) = P(x_t = w | y_t = j)$ is the output probability.

Question 2. Naive Approach (5 points)

(a) Write a function to estimate the output probabilities from the training data `twitter_train.txt` using maximum likelihood estimation (MLE), i.e.,

$$P(x = w | y = j) = b_j(w) = \frac{count(y = j \to x = w)}{count(y = j)}$$

where the numerator is the number of times token w is associated with tag j in the training data, and the denominator is the number of times tag j appears. Save these output probabilities into a file named `naive_output_probs.txt`.
A problem you may encounter is that the training data `twitter_train.txt` does not contain some tokens (words) that appear in the test data. You can handle this "unseen token" problem by smoothing the output probability as

$$b_j(w) = \frac{count(y = j \to x = w) + \delta}{count(y = j) + \delta \times (num\_words + 1)},$$

where *num_words* is the number of unique words in the training data, δ is a real number, and the +1 in the denominator accounts for unseen tokens (all unseen tokens "collapse" into a single token). Typical values of δ to try are: 0.01, 0.1, 1, 10. Pick one that works best for you. (This smoothing applies to the other questions too.)

(b) Using these output probabilities, we can naively obtain the best tag j^* for a given token w with the following equation.

$$j^* = \operatorname*{argmax}_j b_j(w) = \operatorname*{argmax}_j P(x = w | y = j)$$

Write a function `naive_predict()` (see `hmm.py`) that uses the output probabilities in `naive_output_probs.txt` to predict the tags of the tweets in `twitter_dev_no_tags.txt` with this naive approach. Write your predictions into a file named `naive_predictions.txt` in the same format as `twitter_dev_ans.txt`.

(c) What is the accuracy of your predictions (number of correctly predicted tags / number of predictions)? (Use the `evaluate` function in `hmm.py`.)

(Remember to submit your output files `naive_output_probs.txt` and `naive_predictions.txt`.)

Question 3. Better But Still Naive Approach (5 points)

A better approach is to estimate j^* using

$$j^* = \operatorname*{argmax}_j P(y = j | x = w).$$

(a) How do you compute the right-hand side of this equation?

(b) Implement this approach in a method `naive_predict2()` (see `hmm.py`) that uses the output probabilities in `naive_output_probs.txt` to predict the tags of the tweets in `twitter_dev_no_tags.txt`. You can also make use of other information in `twitter_train.txt`. Write your predictions into a file named `naive_predictions2.txt` in the same format as `twitter_dev_ans.txt`.

(c) What is the accuracy of your predictions? (Use the `evaluate` function in `hmm.py`.)

(Please remember to submit `naive_predictions2.txt`.)

Question 4. Viterbi algorithm (20 points)

(a) Write a function to compute the output probabilities and transition probabilities from the training data `twitter_train.txt` using the MLE approach, saving the output probabilities to `output_probs.txt`, and the transition probabilities to `trans_probs.txt`. (You may reuse the function you have defined earlier to compute the output probabilities.) Recall that the transition probability is defined as

$$a_{i,j} = P(y_t = j | y_{t-1} = i) = \frac{count(y_{t-1} = i, y_t = j)}{count(y_{t-1} = i)},$$

where $count(y_{t-1} = i, y_t = j)$ is the number of times tag i transitions to tag j in the training data, and $count(y_{t-1} = i)$ is the number of times tag i appears in the training data. You may wish to "smooth" the transition probability in a similar manner as for the output probability.

(b) Write a function `viterbi_predict` that implements the Viterbi algorithm. This function uses the output probabilities and transition probabilities (stored in `output_probs.txt` and `trans_probs.txt` respectively) to predict the tags for tweets in `twitter_dev_no_tag.txt`, and writes the predictions to `viterbi_predictions.txt`. This function also accepts `twitter_tags.txt` so that it knows the full set of tags. Recall that the Viterbi algorithm computes the best tag sequence $y_1^*, y_2^*, \ldots, y_n^*$ for an observed token sequence $x_1, x_2, \ldots, x_n$ in this manner:

$$y_1^*, y_2^*, \ldots, y_n^* = \underset{y_1, y_2, \ldots, y_n}{\operatorname{argmax}} P(x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_n).$$

(c) What is the accuracy of your Viterbi algorithm on `twitter_dev_no_tag.txt`? (Use the `evaluate` function in `hmm.py`.)

(Remember to submit `output_probs.txt`, `trans_probs.txt` and `viterbi_predictions.txt`.)

Question 5. Improvements (10 points)

(a) How can you improve your POS tagger further? Describe your ideas to improve your system. You are not allowed to use any external resources or packages, and must design your new POS tagger within the framework of HMMs. However, you are free to perform any automatic (i.e., not manual) preprocessing of the data. *Hints: Could you better handle unseen words? Could you better model transition probabilities? Could you take advantage of linguistic patterns present in tweets? Could the words be clustered into meaningful groups?*

(b) Write a function `viterbi_predict2` that implements your improved system. This function takes the same input as `viterbi_predict`. If your changes affect the output probabilities and transition probabilities, write those new probabilities into `output_probs2.txt` and `trans_probs2.txt` respectively. If those probabilities are not affected, simply copy your previous files to `output_probs2.txt` and `trans_probs2.txt`. Write your predictions to `viterbi_predictions2.txt`

(c) What is the accuracy of your improved system? (Use the `evaluate` function in `hmm.py`.)

(Remember to submit `output_probs2.txt`, `trans_probs2.txt` and `viterbi_predictions2.txt`.)

Question 6. Forward-Backward Algorithm (20 points)

Till now, we have assumed that the tags are provided in the training data. However, in reality, tags are not available in tweets. We have learned that the forward-backward algorithm can be used to learn the transition and output probabilities when tags are not provided in the training data.

(a) Write a function `forward_backward` to implement the forward-backward algorithm. This function accepts `twitter_train_no_tag.txt` as training data. It also accepts `twitter_tags.txt`. Because the forward-backward algorithm is an iterative algorithm, this function accepts a `max_iter` to control the maximum number of iterations that can be executed. Recall that with each iteration of the algorithm, it updates its output probabilities and transition probabilities so as to increase the likelihood of the observed data, i.e., $\prod_{k=1}^{N} P(x_1^{(k)}, x_2^{(k)}, \ldots, x_n^{(k)})$ where N is the number of training examples. Hence another way to terminate the algorithm is when the fractional change of the (log)likelihoods between consecutive iterations falls below some threshold. Thus, this function accepts such a threshold `thresh` to determine when

the iterations can be terminated. This function also accepts a seed so that you can replicate the initial random initialization of your output probabilities and transition probabilities. Because the performance of the forward-backward algorithm is highly dependent on the initialization of the output probabilities and transition probabilities, and it may take a long time to converge, you are only required to run the algorithm for 10 iterations (i.e., `max_iter=10`; you could set `thresh` to a small value such as 10^{-4}.)
If you find that your implementation contains multiply nested for-loops, think carefully how you could reduce the level of nesting to speed up your code. A little thought could save you an immense amount of time. Also note that it is not a good idea to initialize the output probabilities and transition probabilities as uniform distributions.

(b) Save your output probabilities and transition probabilities right after initialization in `output_probs3.txt` and `trans_probs3.txt`, and use these to run your improved POS tagging system `viterbi_predict2` on `twitter_dev_no_tag.txt`. Save the predicted tags in `fb_predictions3.txt`. What is the accuracy? (Use the `evaluate` function in `hmm.py`.)

(c) Save your output probabilities and transition probabilities right after 10 iterations in `output_probs4.txt` and `trans_probs4.txt`, and use these to run your improved POS tagging system `viterbi_predict2` on `twitter_dev_no_tag.txt`. Save the predicted tags in `fb_predictions4.txt`. What is the accuracy? (Do not be too alarmed if the accuracy did not improve. Note that you have only run the algorithm for a measly 10 iterations and for a single random initialization. We will test the forward-backward more thoroughly in the next application.)

(d) Write the log-likelihoods of the observed data for each of the 10 iterations. What do you observe about the log-likelihoods?

(Remember to submit `output_probs3.txt`, `trans_probs3.txt`, `viterbi_predictions3.txt`, `output_probs4.txt`, `trans_probs4.txt` and `viterbi_predictions4.txt`.)

2 Modeling and Prediction of Intraday Stock Price Movements

Intraday stock price analysis refer to the study of how stock prices change within a day of trading. The short-term intraday behavior of a stock differs greatly from its long-term inter-day behavior. Accurately modeling intraday behavior is an important task in high-frequency trading (HFT)[2] because it confers HFT traders an advantage in forecasting stock movements.

You will reuse the HMM you have developed for tweet POS tagging for modeling intraday trades and predicting stock price movements. (In BT4013, we will learn more sophisticated Bayesian networks for modeling intraday trades.)

The relevant files are:

(i) `cat_price_changes_train.txt`

(ii) `cat_price_changes_dev.txt`

(iii) `cat_price_changes_dev_ans.txt`

(iv) `cat_states.txt`

(v) `hmm.py`

[2] https://www.investopedia.com/terms/h/high-frequency-trading.asp

`cat_price_changes_train.txt` contains the difference in Caterpillar's[3] stock prices between consecutive trades on Jan 4, 2010. Each line contains the change in stock price (in cents) between consecutive trades (i.e.. $Price_{i+1} - Price_i$ where $Price_i$ is price of the i^{th} trade). Consecutive non-empty lines give a sequence of consecutive price changes. An empty line separates sequences. Below is an example with two sequences of price changes (there is an empty line after the last 2).

```
-1
1
3
4
-6

3
0
2

```

`cat_price_changes_dev.txt` and `cat_price_changes_dev_ans.txt` are used to evaluate your code. `cat_price_changes_dev.txt` is in the same format as `cat_price_changes_train.txt`. Each line in `cat_price_changes_dev_ans.txt` contains the price change that occurs after the last change in its associated sequence in `cat_price_changes_dev.txt`. For example, if `cat_price_changes_dev.txt` contains the two sequences above and `cat_price_changes_dev_ans.txt` contains the two integers below, then `-5` occurs after `-6` of the first sequence, and `-4` occurs after `2` in the second sequence.

```
-5
-4
```

You may assume that a price change is always an integer between -6 and 6 (inclusive).

`cat_price_changes_train.txt` contains 1981 sequences. `cat_price_changes_dev*.txt` contains 981 sequences.

`cat_states.txt` contains the set of possible states {`s0, s1, s2`} (excluding the START (*) and STOP states), i.e., the possible values for each y_t. `hmm.py` is where you write your code.

Question 7. Applying Forward-Backward Algorithm on Intraday Data (25 points)

(i) Reuse the `forward-backward` function you have implemented for POS tagging tweets to learn the output probabilities and transition probabilities for the training data in `cat_price_changes_train.txt`. This function also accepts `cat_states.txt`. Set the maximum iteration of the forward-backward algorithm to a large value (e.g., `max_iter = 100000`) and the convergence threshold to a small value (e.g., `thresh = 1e-4`). This allows the forward-backward algorithm to run to convergence to get reasonable values of the output probabilities and transition probabilities. Write the output probabilities and transition probabilities to `cat_output_probs.txt` and `cat_trans_probs.txt` respectively. (The data and set of possible states should be small enough for you to run the algorithm to convergence if you have implemented it efficiently.)

[3] https://www.caterpillar.com/

(ii) Examine the output probabilities. For each state s (`s0, s1, s2`), provide the probabilities that it generates positive integers, zero, and negative integers, i.e.,
$P(0 < x \leq 6|y = s), P(x = 0|y = s), P(-6 \leq x < 0|y = s)$. What semantics has your HMM learned for each state (`s0, s1, s2`), i.e., what does each state represent?

(iii) Examine the transition probabilities. Using the semantics in your answer to the previous question, what has your HMM learned about the transitions from state to state?

(iv) After you have learned the output probabilities and transition probabilities, you are given a sequence of consecutive intraday stock price changes $x_1, x_2, \ldots, x_n$. How can you use these probabilities to predict the next stock price change x_{n+1}?

(v) Write a function `cat_predict` that implements your answer to the previous question. This function takes the probabilites in `cat_output_probs.txt` and `cat_trans_probs.txt`, and predicts the next price change for each sequence in `cat_price_changes_dev.txt`. This function also accepts `cat_states.txt`. Write your predictions to `cat_predictions.txt` with one prediction per line.

(vi) What is the average squared error of your predictions, i.e.,
$\frac{1}{N}\sum_{i=1}^{N}(x_i^{predicted} - x_i^{truth})^2$, where $x_i^{predicted}$ is your prediction for sequence i , x_i^{truth} is the correct next price change for sequence i, and N is the number of sequences? (Use the `evaluate_ave_squared_error` function in `hmm.py`.)
Naively predicting the next price change to be equal to the previous price change gives an average squared error of 3.94. Can you do better?

(Remember to submit `cat_output_probs.txt`, `cat_trans_probs.txt`, and `cat_predictions.txt`.)