# --- 50.007 Machine Learning Report ---
# POS Tagging

**Professor: Lu** Wei

**Ong** Jing Jie (1000722)
**Dai** Gengling (1000444)
**Yam** Gui Peng David (1000517)

## Instructions:

POS:

Run the python program for Part2.py, Part3.py, Part4.py, Part5a.py & Part5b.py in the same folder that the training data ('train', 'train_combined') & testing data ('dev.in ', 'dev.out', 'test.in') are located.

NPC:

Run the python program for Part2.py & Part3.py in the same folder that the training data ('train') & testing data ('dev.in', 'dev.out') are located.

## Functions Created:

---

**Part 2 –**

**init():**

1. reads from 'train' and creates lists [xtrain] & [ytrain]

xset = sorted unique emissions from [xtrain]

yset = sorted unique emissions from [ytrain]

**emission( xtrain, ytrain, xset, yset ):**

1. Trains emission matrix, [e] from [xtrain] & [ytrain]
2. For the case of emissions of words seen in the training set –
   $(y \rightarrow x_k)$: $e(x_k|y) = \frac{count(y \rightarrow x_k)}{count(y)+1}$
3. Includes the cases of an emission of an unknown word –
   $(y \rightarrow x_{uk})$: $e(x_{uk}|y) = \frac{1}{count(y)+1}$
4. Counts the number of times a tag appears: $count(y_i)$
5. Outputs [e], [ycount]

**readin():**

1. Reads from 'dev.in' & 'dev.out' and creates lists [xtest] & [ytest] respectively

**tag( x, e, ycount, xset, yset ):**

1. Predicts the most likely tag based on the probability of emission of a word given that tag
2. Outputs a list [ypred]

**writefile( ypred ):**

1. Writes into 'dev.p2.out' from 'dev.in' and [ypred]

**Part 3 – (functions that are not mentioned again are inherited from the parts above)**

**transition( xset, yset ):**

1. Trains transition matrix, [a] from [xtrain] & [ytrain]
2. For the case of transitions between tags seen in the training set –
   $(y_i \to y_j)$: $a(y_j|y_i) = \frac{count(y_i,y_j)}{count(y_i) + countzero(y_i)}$
3. For the case of transitions between tags not seen in the training set –
4. $(y_i \to y_j)$: $a(y_j|y_i) = \frac{1}{count(y_i) + countzero(y_i)}$
5. Counts the number of times a tag appears: $count(y_i)$
6. Counts the number of times a tag is unable to transition to another tag: $countzero(y_i)$
7. Outputs [a], [ycount]
8. Note: To include the transition between unseen tags was seen as a counterpart to the emission of an unknown word.

**Viterbi( e, a, x, yset ):**

1. Outputs the most likely set of tags for any given sequence given the emission matrix [e], transition matrix [a], the sequence [x] and the possible tags [yset].
2. The Viterbi algorithm works using Dynamic Programming.
3. To avoid numerical underflow we logged the probabilities of emission matrix [e], transition matrix [a] and added it to the previous maximum probability pi(k,u) for tag u at step k.

**tagging( e, a, xtest, yset ):**

1. Predicts the most likely tag sequence $y_1^* \dots y_n^*$ for a given word sequence $x_1 \dots x_n$, using the Viterbi Algorithm. In this case we maximize the log-likelihood (of seeing the joint probability of $x_1 \dots x_n, y_1 \dots y_n$) instead of the likelihood so as to avert numerical underflow. Because log is a monotonous function, maximizing log-likelihood is equivalent to maximizing likelihood
2. Outputs a list [ypred] which contains all the predicted tag sequences

**writefile( ypred ):**

1. Writes into 'dev.p3.out' from 'dev.in' and [ypred]

**Part 4 – (functions that are not mentioned again are inherited from the parts above)**

**Viterbi( e, a, x, yset):**

1. Outputs the ranked top 10 most likely set of tags for any given sequence given the emission matrix [e], transition matrix [a], the sequence [x] and the possible tags [yset].
2. The Viterbi algorithm works using Dynamic Programming.
3. To avoid numerical underflow we logged the probabilities of emission matrix [e], transition matrix [a] and added it to the previous maximum probability pi(k,u) for tag u at step k.
4. This version of the Viterbi Algorithm tracks down the top ten pathways by calculating at each node it's top ten most likely parent nodes/ pathways reaching it

**writefile( ypred ):**

1. Writes into 'dev.p4.out' from 'dev.in' and [ypred]

**Part 5a (Predict POS) – (functions that are not mentioned again are inherited from the parts above)**

**ViterbiAlgoLog( input_string, A, B, Possible_States, Possible_Words):**

1. Outputs the most likely set of tags for any given sequence given the emission matrix [e], transition matrix [a], the sequence [x] and the possible tags [yset].
2. This Viterbi algorithm has a different naming for emission matrix [e] as [B], transition matrix [a] as [A], sequence [x] as [input_string], and possible tags [yset] as [Possible_States].
3. The Viterbi algorithm works using Dynamic Programming.
4. To avoid numerical underflow we logged the probabilities of emission matrix [e], transition matrix [a] and added it to the previous maximum probability pi(k,u) for tag u at step k.

**producetagging( filepath, A, B, Possible_States, Possible_Words ):**

1. Predicts the most likely tag sequence $y_1^* \dots y_n^*$ for a given word sequence $x_1 \dots x_n$, using the Viterbi Algorithm. In this case we maximize the log-likelihood (of seeing the joint probability of $x_1 \dots x_n, y_1 \dots y_n$) instead of the likelihood so as to avert numerical underflow. Because log is a monotonous function, maximizing log-likelihood is equivalent to maximizing likelihood
2. Outputs a list [ypred] which contains all the predicted tag sequences
3. Outputs the whole string which was input into it from whichever filepath

**testaccuracy(optimal_tags, answer_tags):**

1. Checks between the [ypred] and the [yset] to see the accuracy

**writefile( ypred ):**

1. Writes into 'dev.p5.out' from 'dev.in' and [ypred]

**Writefile2( ypred ):**

1. Writes into 'test.p5.out' from 'test.in' and [ypred]

**word_filter( word ):**

1. Filters the word in various circumstances…
    a. If it is a number -> Changes it to '0'
    b. If it is a timing -> Changes it to '0'
    c. If it is a url -> Changes it to 'http'
    d. If it is a @ followed by a username -> Changes it to '@name'
    e. If it is a hash tag -> Changes it to "#name'
    f. Else -> Changes it into lowercase
2. Note: This is done so as to reduce the variance seen for the emission matrix [e] so as to increase accuracy of tagging

**init():** Changes: Filters the word using **word_filter( word )** before making [xtrain]

**readin():** Changes: Filters the word using **word_filter( word )** before making [xtest]

**emission_part5( x, y, xset, yset, de ):**

1. Trains emission matrix, [e] from [xtrain] & [ytrain]
2. Makes use of **Absolute Discounting** to smoothen the Emission parameters
3. More details shown in next section (Absolute Discounting)

**producetransitionmatrix_part5( filepath, dt ):**

1. Trains transition matrix, [a] by opening and reading from the training under filepath
2. Makes use of **Absolute Discounting** to smoothen the Transition parameters
3. More details shown in next section (Absolute Discounting)

**Part 5b (Predict test)– (functions that are not mentioned again are inherited from the parts above)**
**readin():** Changes: trains on 'train_combined' which made up of 'dev.out' and 'train'

**producetransitionmatrix_part5( filepath, dt ):** Changes: trains on 'train_combined' which made up of 'dev.out' and 'train'

---

## Methods to improve accuracy for Challenge:

### The Need for Smoothing in HMMs:

Any form of Smoothing is required in a Hidden Markov Model because the weightage given to instances which are seen is much higher than weightage given to instances which are unseen, which may not represent the true nature of real instances. In our project we chose to use Absolute Discounting because it was shown to have the higher gain in accuracy among the other forms of Smoothing we found: Laplace Smoothing, Good-Turing Estimation & Shrinkage.

### Absolute Discounting:

**Smoothing the Emission Probabilities:**

We use a value $de$ to discount the probability of emissions from a state $y_i$. This discounted value $de$ will then be shared among those words which are given zero probability by Maximum Likelihood Expectation. (i.e. those words have zero probability if they are not seen in the training set to be emissions of that particular tag).

$$\boldsymbol{Probability\ of\ emission}: \left(\boldsymbol{word = w_j}, \quad \boldsymbol{state = y_i}, \quad \boldsymbol{m = total\ \#\ of\ seen\ words}\right)$$

$$\boldsymbol{e(w_j|y_i) = max}\left(\boldsymbol{0}, \frac{\boldsymbol{count(y_i \rightarrow w_j) - de}}{\boldsymbol{count(y_i)}}\right) + \frac{\boldsymbol{de} * \sum_{\boldsymbol{h=1}}^{\boldsymbol{m}} \boldsymbol{1}\{\boldsymbol{count(y_i \rightarrow w_h) > 0}\}}{\boldsymbol{m} * \boldsymbol{count(y_i)}}$$

Note that:

1. This quantity will always be above 0.

   Proof: $\max(0, f(x)) > 0$ for any $f(x)$ & $\frac{de*\sum_{h=1}^{m} 1\{count(y_i \rightarrow w_h)>0\}}{m*count(y_i)} > 0$ **as** $\boldsymbol{de, m, count(y_i)}$**>0**

2. The sum of all words $w_j$ given a state $y_i$ = 1

   Proof: $\sum_{j=1}^{k} max\left(0, \frac{count(y_i \rightarrow w_j) - de}{count(y_i)}\right) + \frac{de*\sum_{h=1}^{m} 1\{count(y_i \rightarrow w_h)>0\}}{m*count(y_i)}$

   $$= \frac{count(y_i) - de * \sum_{j=1}^{m} 1\{count(y_i \rightarrow w_h) > 0\}}{count(y_i)} + \frac{de * \sum_{h=1}^{m} 1\{count(y_i \rightarrow w_h) > 0\}}{count(y_i)} = 1$$

**Smoothing the Transition Probabilities:**

We use a value $dt$ to discount the probability of emissions from a state $y_i$. This discounted value $dt$ will then be shared among those transitions which are given zero probability by Maximum Likelihood Expectation. (i.e. these transitions have zero probability if they are not seen in the training to transition from one to the other).

$$\boldsymbol{Probability\ of\ transition}: \left(\boldsymbol{state\ 1 = y_i}, \quad \boldsymbol{state\ 2 = y_j}, \quad \boldsymbol{k = total\ \#\ of\ seen\ tags}\right)$$

$$\boldsymbol{a(y_j|y_i) = max\left(0, \frac{count(y_i, y_j) - dt}{count(y_i)}\right) + \frac{dt * \sum_{h=1}^{k} 1\{count(y_i, y_h) > 0\}}{k * count(y_i)}}$$

Note that:

1. This quantity will always be above 0.

   Proof: $\max(0, f(x)) > 0$ for any $f(x)$ & $\frac{dt*\sum_{h=1}^{k} 1\{count(y_i,y_h)>0\}}{k*count(y_i)} > 0$ **as** $\boldsymbol{dt, k, count(y_i)}$**>0**

2. The sum of all transitions $y_i, y_j$ given a state $y_i$ = 1

   Proof: $\sum_{j=1}^{m} max\left(0, \frac{count(y_i,y_j)-dt}{count(y_i)}\right) + \frac{dt*\sum_{h=1}^{k} 1\{count(y_i,y_j)>0\}}{k*count(y_i)}$

   $$= \frac{count(y_i) - dt * \sum_{j=1}^{k} 1\{count(y_i, y_j) > 0\}}{count(y_i)} + \frac{dt * \sum_{h=1}^{k} 1\{count(y_i, y_h) > 0\}}{k * count(y_i)} = 1$$

We chose de = 0.03 and dt = 0.675 as the values to discount the emission/ transition rows repectively. These were chosen after a large number of iterations of 'part5.py' on 'dev.in' with various values for de and dt. It was empirically seen to have converged to a maximum at de = 0.03 and dt = 0.675.

**<u>Pre-processing the dataset to reduce variance:</u>**

We pre-processed the training set using a newly created function word_filter( word ) to group various words/ numbers/ patterns of words which have the same meaning in a single word/ item.
(e.g. '123' turns into '0')
Rules:

1. If it is a number -> Changes it to '0'
2. If it is a timing -> Changes it to '0'
3. If it is a url -> Changes it to 'http'
4. If it is a @ followed by a username -> Changes it to '@name'
5. If it is a hash tag -> Changes it to "#name'
6. Else -> Changes it into lowercase

This reduces the random error in predicting certain patterns of words which can actually be grouped together. We also pre-processed the testing dataset so as to have parallel training and testing words.

**<u>Increasing the size of the dataset to reduce variance:</u>**

We combined 'train' and 'dev.out' in part 5b so as to obtain a larger dataset for prediction of tags for 'test.in'. The training file is 'train_combined'

**Maximize log-likelihood to avoid numerical underflow:**

We changed the optimization problem from a maximum of the joint probability to the maximum log-likelihood"

$$\max\{ P(x_1, \dots, x_n, y_0, \dots, y_{n+1}; \Theta) \}$$

$$= \max\{ \prod_{i=1}^{n+1} P(y_i|y_{i-1}) \prod_{j=1}^{n} P(x_j|y_j) \}$$

$$= \max\{ \sum_{i=1}^{n+1} \log P(y_i|y_{i-1}) + \sum_{j}^{n} logP(x_j|y_j) \}$$

---

## Results:

| (All spaces are excluded in score) | POS | NPC |
|---|---|---|
| **Part 2** | 0.600914205345 | 0.700003022335 |
| **Part 3** | 0.665611814346 | 0.7762565358 |
| **Part 4** | 0.658931082982 | NA |
| **Part 5** | 0.790436005626 | NA |

**Comparison of results from Part 3 and Part 5:**

The increase of accuracy from Part 3 to Part 5 can be attributed to the Smoothing of Emission and Transition parameters, Smoothing is required in HMM models due to data scarcity. The amount of data we were given was 6772 lines with 157 sequences of words separated by spaces. This meant that there would be many unseen instances in our training set.

Another factor would be to reduce the random error by pre-processing the training set. This is mentioned in more detail above in (**Pre-processing the dataset to reduce variance).**

**Further possible improvements:**

Some models we considered implementing were the Conditional Random Fields model (CRF), which does not reply on assumptions that words can only be generated from a specific tag in that position and tags are generated from the previous tag. However, based on online literature, we found that a CRF model requires a large amount of data (approximately 5000-10000 sequences) to achieve an increased accuracy over a simple model like HMM.

Another method would be to re-rank the top 10 possible tag sequences from our "part4.py", however we were unable to implement a scoring system to weight/ score the top 10 possible tag sequences. In online literature, it has seen to have caused an increase of 6% accuracy for certain models.