# 01_112_ml_project

Project for SUTD 01.112 Machine Learning Fall 2019

## 1. Group members

Chan Luo Qi 1002983

Eda Tan 1003098

Glenn Chia 1003118

## 2. Explaining the code

### 2.1 Part 2

#### 2.1.1 Part 2 Question 1 Emission probabilities

To work with the **training data**, we wrote a function defined as `readtopdftrain` that reads in the csv and shows the data as a DataFrame with the columns 'words' and 'tags' as shown below.

```python
def readtopdftrain(file_path):
    with open(file_path, encoding="utf8") as f_message:
        temp = f_message.read().splitlines()
        temp = list(filter(None, temp))
    separated_word_tags = [word_tags.split(' ') for word_tags in temp]
    # separated_word_tags = [word.strip() for l in separated_word_tags for word in l]
    df = pd.DataFrame(separated_word_tags, columns=['words', 'tags'])
    df["words"] = [i.strip() for i in df.words]
    return df
```

To work with the **testing data**, we wrote another function called `readtopdftest` that returns a DataFrame with the columns 'words' and 'sentence id'. Words with the same sentence id will be part of the same sentence. This is to make it easier to split the sentences for the output file later.

```
def readtopdftest(file_path):
    with open(file_path, encoding="utf8") as f_message:
        temp = f_message.read().splitlines()

    words = []
    sentenceid = 0
    for word in temp:
        if word != "":
            words.append([word, sentenceid])
        else:
            sentenceid += 1

    df = pd.DataFrame(words, columns=['words', 'sentence id'])
    return df
```

\ To estimate the emission probabilities, in the training dataframe, we follow the following steps:

```
def estimate_emission_parameters(df):
    """
    Calculates the emission probabilities from count of words/ count of tags
    :param df: raw word to tag map
    :return: columns = count of word, count of tags, all emission probabilites of tag --> word
    """
    count_emit = df.groupby(['tags', 'words']).size().reset_index()
    count_emit.columns =["tags", "words", "count_emit"]
    count_tags = df.groupby(["tags"]).size().reset_index()
    count_tags.columns = ["tags", "count_tags"]

    count = pd.merge(count_emit, count_tags, on="tags")
    count["emission"] = count["count_emit"]/count["count_tags"]
    return count.drop(columns=["count_emit", "count_tags"])
```

We decided to use DataFrame as it is a very efficient way of storing the data and there are many ways of transforming the data quickly and efficiently using pandas' functions.

1. Create a new dataframe (`count_emit`) by grouping the rows of the dataframe by their tags, and then the words and finding the size of these groupings
2. Reset the index of `count_emit`.
3. Rename the size column as "count_emit".
4. Create another dataframe (`count_tags`) that counts the number of tags only
5. Lastly, merge the two dataframes together using the "tags" column and calculate the emission.
    - `count['emission']` = $\frac{‘count[”count_{e}mit”]‘}{‘count[”count_{t}ags”]‘}$

- This follows the equation given:

$$e(x|y) = \frac{\text{Count}(y \to x)}{\text{Count}(y)}$$

6. Drop the other two columns and leave the "emission" column.

## 2.1.2 Part 2 Question 2: Smoothing

We smooth the training data set using a function called `smoothingtrain`.

```
def smoothingtrain(data, k=3):
    word_counts = data['words'].value_counts().to_dict()
    data['words'] = data['words'].apply(lambda word: replacewordtrain(word, word_counts, k))
    return data
```

The output would look like this:

```
              words    tags
0             #UNK#    B-NP
1             bonds    I-NP
2               are    B-VP
3         generally  B-ADVP
4                 a  B-ADJP
...             ...     ...
181623     resolved    I-VP
181624           in    B-PP
181625          the    B-NP
181626         West    I-NP
181627          ...       O

[181628 rows x 2 columns]
```

`smoothingtrain` calls a helper function (`replacewordtrain`) that replaces words that occur less than k times with the tag '#UNK#'.
This function is applied to every single row in the dataframe using the `.apply` function.

```
def replacewordtrain(word, word_counts, k):
    if word_counts[word] < k:
        return "#UNK#"
    return word
```

The output looks like this:

```
          words
0           HBO
1           has
2         close
3            to
4            24
...         ...
26126     #UNK#
26127      were
26128        in
26129  Congress
26130         .

[26131 rows x 1 columns]
```

Similarly, for the test data, we smooth it using a function called `smoothingtest`.

```
def smoothingtest(testdata, traindata):
    trainvalues = set(traindata['words'])
    testdata['words'] = testdata['words'].apply(lambda word: replacewordtest(word, trainvalues))
    return testdata
```

This function calls the helper function `replacewordtest` that checks if the word is in the training data set. If it is, return the word. If not, replace it with the tag #UNK#.

```
def replacewordtest(word, train):
    if word in train:
        return word
    return "#UNK#"
```

## 2.1.3 Part 2 Question 3: Simple Sentiment Analysis System

For this sentiment analysis system, we predict using the tag that gives us the maximum emission probability. Firstly, we created a function (`get_emissionlookup`) that will map each word to the tag with the highest emission probability.

- Again, the function finds the indexes where the emission probability are the highest, then stores that in a column in a DataFrame with the words and tags.
- Finally, it returns the dataframe as a dictionary of words and predicted tags.

```
def get_emissionlookup(argmax_emission):
    """
    Map each word to tag of highest emission probability. This ensures lookup is in O(1) time.
    :param argmax_emission: Dataframe with emission probabilities of each tag --> word
    :return: Dictionary of word --> highest e(x|y) tag
    """
    idx = argmax_emission.groupby(['words'])['emission'].transform(max) == argmax_emission['emission']
    argmax_emission = argmax_emission[idx]
    lookup = dict(zip(argmax_emission.words, argmax_emission.tags))
    return lookup
```

Next, the `get_tag_fromemission` function retrieves the tag for each word seen in the test set, and writes it out to a file.

```python
def get_tag_fromemission(lookup, smoothedtest, dataset):
    """

    Retrieve tag for each word seen in testset
    :param lookup: word --> Highest e(y|x) tag
    :param smoothedtest: Processed testset to exclude non-occuring words in trainset
    :param dataset: EN/ CN/ AL/ SG
    :return: output file with allocated tags
    """
    output_file = dataset + "/dev.p2.out"
    with open(output_file, "w", encoding="utf8") as f:
        sentenceid = 0
        for index, row in smoothedtest.iterrows():
            if row['sentence id'] != sentenceid:
                f.write("\n")
                f.write(row['words'] + " " + lookup[row['words']] + "\n")
                sentenceid += 1
            else:
                f.write(row['words'] + " " + lookup[row['words']] + "\n")
    f.close()
```

All of these functions are called in a `sentiment_analysis` function that will read in the data, smooth it, and write the predicted results to a file.

```python
def sentiment_analysis(dataset):
    train_path = dataset + "/train"
    traindf = readtopdftrain(train_path)
    smoothedtrain = smoothingtrain(traindf)
    test_path = dataset + "/dev.in"
    testdf = readtopdftest(test_path)
    smoothedtest = smoothingtest(testdf, smoothedtrain)
    argmax_emission = estimate_emission_parameters(smoothedtrain)
    lookup = get_emissionlookup(argmax_emission)
    get_tag_fromemission(lookup, smoothedtest, dataset)

    print("Done with dataset " + train_path)
```

## 2.2 Part 3

### 2.2.1 Write a function that estimates the transition parameters from the training set using MLE (maximum likelihood estimation): Consider special cases for STOP and START

The `read_to_pdf` function here differs a bit from the ones coded in Part 2. We still create a dataframe, however, in this function, we have to append the 'START' and 'STOP' tags so as to run the Viterbi. Thus, we check for the first and last index, and append 'START' and 'STOP' accordingly. Also if the word is equal to an empty string, we append 'STOP' and then 'START' as it signifies the end of the old sentence and the start of the new one.
Also, we create a new column called 'tags_next' that shows the tag in the sequence after the current one.

```
def read_to_pdf(file_path):
    with open(file_path) as f_message:
        temp = f_message.read().splitlines()
    words = []
    tags = []
    for index, word_tags in enumerate(temp):
        if index == 0:
            words.append('')
            tags.append('START')
        elif index == len(temp) - 1:
            words.append('')
            tags.append('STOP')
        elif word_tags == '':
            words.append('')
            tags.append('STOP')
            words.append('')
            tags.append('START')
        else:
            split_word_tags = word_tags.split(' ')
            words.append(split_word_tags[0])
            tags.append(split_word_tags[1])
    tags_next = tags[1:]
    df = pd.DataFrame(list(zip(words, tags, tags_next)), columns =['words', 'tags', 'tags_next'])
    df['tags_next'] = df['tags_next'].str.replace('START', '')
    return df
```

The output of `readtopdf` is as follows:

```
              words    tags tags_next
0                     START      I-NP
1             bonds    I-NP      B-VP
2               are    B-VP    B-ADVP
3         generally  B-ADVP    B-ADJP
4                 a  B-ADJP    I-ADJP
...             ...     ...       ...
196947     resolved    I-VP      B-PP
196948           in    B-PP      B-NP
196949          the    B-NP      I-NP
196950         West    I-NP         O
196951          ...       O      STOP

[196952 rows x 3 columns]
```

Next, we create the `estimate_transition_parameters` function.
Using the training dataframe passed to us, we group the rows in the dataframe by the tags again and count the number of tags in the dataframe.
Next, we group the rows by the tags and then by tags_next, and count the number of tags_next

per tags (transition of tag -> tag_next).

- Note that we have to locate all of the tags_next that are empty string and replace them with the number zero, as they are not valid transitions. Lastly, we drop columns that we do not require, sort the values and return the dataframe.

The function code is as follows:

```
def estimate_transition_parameters(df):
    """Return a dataframe with
    tag | next_tag | count_tag | count_transition | transition_prob

    Parameters:
    df (DataFrame): Dataframe with word, tags, tags_next

    Returns:
    df (DataFrame)
    """
    df['count_tag'] = df.groupby(['tags']).tags.transform(np.size)
    df['count_transition'] = df.groupby(['tags', 'tags_next']).tags.transform(np.size)
    df.loc[df.tags_next == '', 'count_transition'] = 0
    df['transition_probability'] = df['count_transition'] / df['count_tag']
    df = df.drop_duplicates(subset=['tags', 'tags_next'])
    df = df.drop(['words'], axis=1)
    df = df.sort_values(['tags','tags_next'])
    df = df.reset_index()
    df = df.drop(['index'], axis=1)
    return df
```

The output is:

```
      tags tags_next  count_tag  count_transition  transition_probability
0   B-ADJP    B-ADJP       1751                 2                0.001142
1   B-ADJP    B-ADVP       1751                28                0.015991
2   B-ADJP      B-NP       1751                91                0.051970
3   B-ADJP      B-PP       1751               428                0.244432
4   B-ADJP     B-PRT       1751                 1                0.000571
..     ...       ...        ...               ...                     ...
157  START    B-SBAR       7663               173                0.022576
158  START      B-VP       7663               143                0.018661
159  START      I-NP       7663                 1                0.000130
160  START         O       7663              1087                0.141850
161   STOP                 7662                 0                0.000000

[162 rows x 5 columns]
```

## 2.2.2 Use the estimated transition and emission parameters, implement the Viterbi algorithm to compute the following (for a sentence with n words): Tag sequence

The algorithm is the same as what we were taught in class

**Step 1: Initializing the tree**

```
def build_tree(self):
    """
    initialise viterbi tree (middle layers)
    :return: np.array of size = (t, n); t= rows, n = length
    """
    return np.zeros((len(self.t), len(self.sentence)))
```

```
    def get_score(self, n, t):
        """
        calculates score of a particular node
        :param n: # of words in sentence + 2 (START, STOP)
        :param t: # of states
        :return: array of max 7 scores scores
        """
        if n == 0:
            print("AT START NODE")
            return 1
```

We first initialize all the scores to be zero other than the first score as per the formula

- $\pi(0, u) = 1$ if u = START
- $\pi(0, u) = 0$ otherwise

Step 2: Considering the layer after the START node

```
if i == 0:
    idx = 0
    for j in self.t:
        if ("START", j) not in self.transition or (j, self.sentence[0]) not in self.emission:
            tree[idx, i] = 0.0
        else:
            tree[idx, i] = self.transition[("START", j)] * self.emission[(j, self.sentence[0])]
        idx += 1
```

For the first layer, the only node that it transitions from is the START node, hence there is only one possible score for each node and the formula used is $\pi(0, START) \cdot b_u(x_1) \cdot a_{START,u}$. Hence we don't need to do a max function

**Step 3: Between the START and STOP nodes**

```
else:
    idx = 0
    for j in self.t:
        all_scores = []
        idx_k = 0
        for k in self.t:
            print("Inner: " + k)
            if (k, j) not in self.transition or (j, self.sentence[i]) not in self.emission:
                all_scores.append(0)
            else:
                score = tree[idx_k, i-1] * self.transition[(k, j)] * self.emission[(j, self.sentence[i])]
                all_scores.append(score)
            idx_k += 1
        tree[idx, i] = max(all_scores)
        tag_idx = all_scores.index(max(all_scores))
        tag = self.t[tag_idx]
        # all_nodes[idx][i] = node(idx, i)
        all_nodes[idx][i-1].subpath = tag
        if i != 1:
            all_nodes[idx][i-1].parent = all_nodes[tag_idx][i-2]
        else:
            pass

        idx += 1
```

We iterate through the layers and for each layer, we look at each node. For each node, we compute the scores based on the score from the previous nodes multiplied by the transition probabilities from the previous node to the current node, multiplied by the emission probabilities of the current node. This gives us a list of scores from the nodes from the previous layer to the current node. Subsequently, we find the max of the score in the list and get its index. The index corresponds to the node from the previous layer that produced the best score for each node in the current layer. Since the node corresponds to the tag, we essentially have the best tag from the previous layer from each node. We store this tag in the current node. We also store the identity of the previous node that produced this tag.

This follows the formula taught in class

$max_v\{\pi(j,v)\cdot b_u(x_{j+1})\cdot a_{v,u}\}$

**Step 4: At the STOP node at layer n+1**

```
elif i == len(self.sentence):
    all_scores = []
    idx = 0
    for j in self.t:
        if (j, "STOP") not in self.transition:
            all_scores.append(0)
        else:
            score = tree[idx, -1] * self.transition[(j, "STOP")]
            all_scores.append(score)
        idx += 1
    maxscore = max(all_scores)
    tag_idx = all_scores.index(max(all_scores))
    tag = self.t[tag_idx]
    # all_nodes[0][i] = node(0, i)
    last_node = node(0, i)
    last_node.parent = all_nodes[tag_idx][-1]
    last_node.subpath = tag
```

We reached the last layer which only has the STOP node. We compute the scores based on the score from the previous nodes (in layer n) multiplied by the transition probabilities from the previous node to the STOP node. This gives us a list of scores from the nodes from the previous layer to the STOP node. Subsequently, we find the max of the score in the list and get its index. The index corresponds to the node from the previous layer that produced the best score to the STOP node. Since the node from the previous layer stores the best path up to itself, if the final score at the STOP node is highest, this guarantees that it is the best path. We then store the tag corresponding to the previous node. We also store the identity of the node that produced this tag.

This follows the formula taught in class

$max_v\{\pi(n, v) \cdot a_{v,STOP}\}$

**STEP 5: Working out the path**

```
path = []
current_node = last_node
for i in range(len(self.sentence), 0,  -1):
    path.insert(0, current_node.subpath)
    current_node = current_node.parent
print(path)
```

Since each node stores the best tag of the previous layer, we start from the STOP node and work backwards. The STOP node (which is at layer n+1) will get the tag from layer n. Since we also store the node that produce the best tag, we can use it to find its parent and the associated tag. Working backwards will thus help us find the best path which is the tag sequence.

## 2.3 Part 4: Implement an algorithm to find the 7-th best output sequences. Clearly describe the steps of your algorithm in your report.

### 2.3.1 High level description

Our approach for finding the 7th best path involves finding and storing the paths to the best 7 scores as we traverse between the layers. Hence, for a node `i` in the `j` layer, it will store the best 7 scores from all nodes in the previous layer `j-1` and the paths from `START` to itself that traverses through the best 7 nodes in layer `j-1`. For example, this means that a layer with 10 nodes will have 10*7=70 best paths and scores.

We do this from the START node (The best path 7 paths at this point ) all the way to the end STOP node. At the STOP node, we have all the scores from the previous layer multiplied by the transition probabilities between nodes from the previous layer to the STOP node $a_{u,STOP}$. We sort all the scores and find the 7th best score. This score guarantees that the path from the START node to the STOP node is the 7th best path. Since we are keeping track of the path from the START node to every node in every layer, we can fetch the path corresponding to the 7th best path.

### 2.3.2 Low level description

**Notations**

We modify the scores equation to be an array $\pi(j,u)[k]$ where `j` is the current layer (Note that START starts from layer 0), `u` is the current node and k is the index of the array. if k=0 we are finding the best path. If k=6, we are finding the 7th best path to that node (since index starts from 0).

We let PATH(j,u)[k] where `j` is the current layer and `u` is the current node in that layer. `k` represents the index in the array that contains the path to that node.

**Step 1: Initializing the tree**

To initialise the tree, we instantiate a (n, t, 7) 3D-array with a custom NODE datatype, where n = # of words in the sentence and t = # of tags. Each (n, t) represents a node in the traditional viterbi tree, and each node u is expanded to an array of size 7 to store the 7 best nodes (score, parent, tag) from START to node u. The node datatype in each position stores the score from START to u, as well as the parent node that produced this score, and its corresponding output tag. This is done so backtracking can be done efficiently to find the full path eventually.

**Step 1a: Consider the start layer.**

We initialize the best score of the start layer to be 1.0

$\pi(0, START)[0] = 1.0$

All other scores will be set to 0.0 since there will only be one path from START node to the next layer and we don't need to track the other scores besides the best one for the transition from START node to layer 1.

$\pi(0, START)[h] = 0.0$ where $1 \leq h \leq k-1$

The paths stored at this stage are PATH(0,START)[0] = START and the rest of the paths are PATH(0,START)[0] = NULL

**Step 1b: All other nodes in the tree besides the START node will be initialized to a starting score of 0.0**

**Step 2: Consider the nodes between START and STOP**

For each node in layer `j+1` we find all the scores from the previous layer to this node by taking $\pi(j+1,u)[h] * a_{v,u} * b_u(x_{j+1})$ for every node `v` in the previous layer `j` and every score in node `v`'s score array. We then have all the scores. We then find the best 7 scores and store the indexes of these 7 scores. The indexes give us useful information of which score in which node produced that particular score. In order words, this tells us the node's parent and corresponding tag. We store the 7 best nodes in $\pi(j+1,u)[h]$ for h=0 to h=6.

PATH(j+1, u)[h] = PATH(j, v))[Index which produced one of the 7 best scores] + (v -> u)

where `v` is the previous node in layer `j` with its respective index that produced one of the best 7 scores.

**Step 3: Consider the STOP node at layer n+1**

We compute all the scores from the previous layer n to this node by doing $\pi(n,u)[h] * a_{u,STOP}$ for all nodes u in the previous layer and the 7 scores stored in each node. We now have all the scores. We now find the 7 best scores and store corresponding parent and tag information into the a Node() object.

Assume node q's third score is the overall 7th best score, we have

PATH(n+1, STOP)[7] = PATH(n, q))[3] + (q -> STOP)

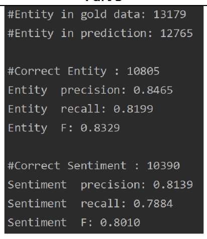
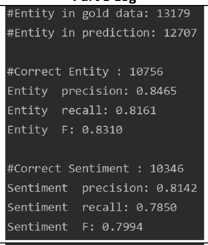
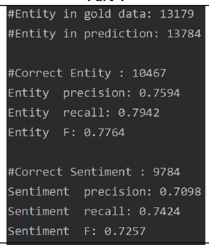
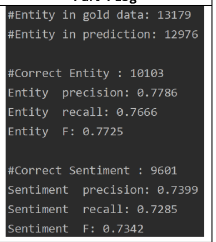
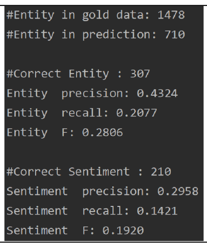
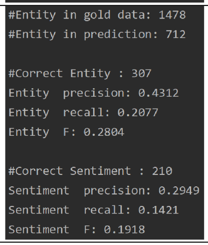
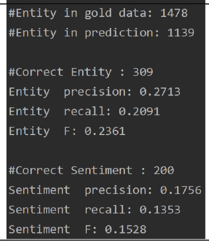
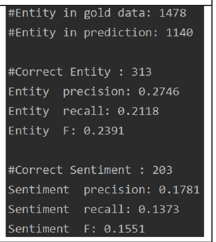
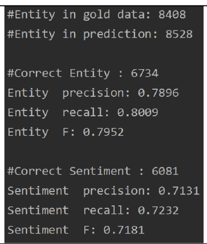
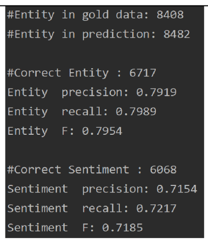
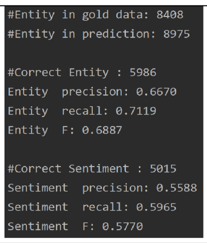
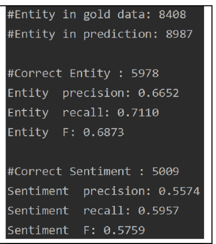
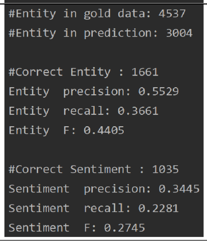
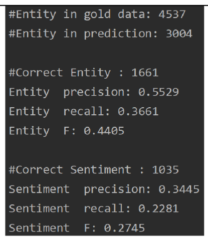
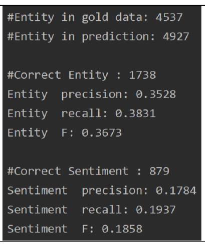
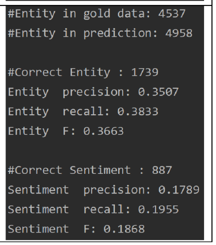
**Step 4: Get 7th best path**

At this stage, we have a completed (n, t, 7) ndarray of Node() objects. We use this array and the final array of 7 best scores from the last STOP layer to find the 7th best path. Take note that each Node() object contains a pointer to its parent and corresponding tag.

Using the 7th best score from the last layer, we implement a simple backtrack to get the tags of each node's parent. We append the tag of last node's parent (already stored in the node from previous iteration of modified viterbi) to a path. We then find this node's parent and do to same by inserting its tag at the front of this path. This is done from layer n+1 to layer 1, and the 7th best path is generated.

## 2.3.3 Results

Please refer to the table below for the results from our viterbi where the row is the type of data and the columns are the questions.

| | Part 3 | Part 3 Log | Part 4 | Part 4 Log |
|---|---|---|---|---|
| **EN** |  #Entity in gold data: 13179 #Entity in prediction: 12765 #Correct Entity : 10805 Entity precision: 0.8465 Entity recall: 0.8199 Entity F: 0.8329 #Correct Sentiment : 10390 Sentiment precision: 0.8139 Sentiment recall: 0.7884 Sentiment F: 0.8010 |  #Entity in gold data: 13179 #Entity in prediction: 12707 #Correct Entity : 10756 Entity precision: 0.8465 Entity recall: 0.8161 Entity F: 0.8310 #Correct Sentiment : 10346 Sentiment precision: 0.8142 Sentiment recall: 0.7850 Sentiment F: 0.7994 |  #Entity in gold data: 13179 #Entity in prediction: 13784 #Correct Entity : 10467 Entity precision: 0.7594 Entity recall: 0.7942 Entity F: 0.7764 #Correct Sentiment : 9784 Sentiment precision: 0.7098 Sentiment recall: 0.7424 Sentiment F: 0.7257 |  #Entity in gold data: 13179 #Entity in prediction: 12976 #Correct Entity : 10103 Entity precision: 0.7786 Entity recall: 0.7666 Entity F: 0.7725 #Correct Sentiment : 9601 Sentiment precision: 0.7399 Sentiment recall: 0.7285 Sentiment F: 0.7342 |
| **CN** |  #Entity in gold data: 1478 #Entity in prediction: 710 #Correct Entity : 307 Entity precision: 0.4324 Entity recall: 0.2077 Entity F: 0.2806 #Correct Sentiment : 210 Sentiment precision: 0.2958 Sentiment recall: 0.1421 Sentiment F: 0.1920 |  #Entity in gold data: 1478 #Entity in prediction: 712 #Correct Entity : 307 Entity precision: 0.4312 Entity recall: 0.2077 Entity F: 0.2804 #Correct Sentiment : 210 Sentiment precision: 0.2949 Sentiment recall: 0.1421 Sentiment F: 0.1918 |  #Entity in gold data: 1478 #Entity in prediction: 1139 #Correct Entity : 309 Entity precision: 0.2713 Entity recall: 0.2091 Entity F: 0.2361 #Correct Sentiment : 200 Sentiment precision: 0.1756 Sentiment recall: 0.1353 Sentiment F: 0.1528 |  #Entity in gold data: 1478 #Entity in prediction: 1140 #Correct Entity : 313 Entity precision: 0.2746 Entity recall: 0.2118 Entity F: 0.2391 #Correct Sentiment : 203 Sentiment precision: 0.1781 Sentiment recall: 0.1373 Sentiment F: 0.1551 |
| **AL** |  #Entity in gold data: 8408 #Entity in prediction: 8528 #Correct Entity : 6734 Entity precision: 0.7896 Entity recall: 0.8009 Entity F: 0.7952 #Correct Sentiment : 6081 Sentiment precision: 0.7131 Sentiment recall: 0.7232 Sentiment F: 0.7181 |  #Entity in gold data: 8408 #Entity in prediction: 8482 #Correct Entity : 6717 Entity precision: 0.7919 Entity recall: 0.7989 Entity F: 0.7954 #Correct Sentiment : 6068 Sentiment precision: 0.7154 Sentiment recall: 0.7217 Sentiment F: 0.7185 |  #Entity in gold data: 8408 #Entity in prediction: 8975 #Correct Entity : 5986 Entity precision: 0.6670 Entity recall: 0.7119 Entity F: 0.6887 #Correct Sentiment : 5015 Sentiment precision: 0.5588 Sentiment recall: 0.5965 Sentiment F: 0.5770 |  #Entity in gold data: 8408 #Entity in prediction: 8987 #Correct Entity : 5978 Entity precision: 0.6652 Entity recall: 0.7110 Entity F: 0.6873 #Correct Sentiment : 5009 Sentiment precision: 0.5574 Sentiment recall: 0.5957 Sentiment F: 0.5759 |
| **SG** |  #Entity in gold data: 4537 #Entity in prediction: 3004 #Correct Entity : 1661 Entity precision: 0.5529 Entity recall: 0.3661 Entity F: 0.4405 #Correct Sentiment : 1035 Sentiment precision: 0.3445 Sentiment recall: 0.2281 Sentiment F: 0.2745 |  #Entity in gold data: 4537 #Entity in prediction: 3004 #Correct Entity : 1661 Entity precision: 0.5529 Entity recall: 0.3661 Entity F: 0.4405 #Correct Sentiment : 1035 Sentiment precision: 0.3445 Sentiment recall: 0.2281 Sentiment F: 0.2745 |  #Entity in gold data: 4537 #Entity in prediction: 4927 #Correct Entity : 1738 Entity precision: 0.3528 Entity recall: 0.3831 Entity F: 0.3673 #Correct Sentiment : 879 Sentiment precision: 0.1784 Sentiment recall: 0.1937 Sentiment F: 0.1858 |  #Entity in gold data: 4537 #Entity in prediction: 4958 #Correct Entity : 1739 Entity precision: 0.3507 Entity recall: 0.3833 Entity F: 0.3663 #Correct Sentiment : 887 Sentiment precision: 0.1789 Sentiment recall: 0.1955 Sentiment F: 0.1868 |

## 2.3.4 Using Log Calculations

While running the viterbi for part 3 and part 4, we realised that there might be undetected underflow problems. That would cause non-zero, but close to zero scores to be mistakenly recorded as zero, and affects the robustness of our model. In an attempt to circumvent this problem, we implement the viterbi using log calculations.

Apart from score calculation, the flow of the viterbi algorithm remains as above.

log_score of Node u = log_score(v) + $log(a_(v, u) + log(b_u(o))$

A summary of all the scores can be found in section 2.3.3 Results.

As expected, the scores for k-best viterbi have improved more than that of regular viterbi. In computing the 7th best viterbi, we are more likely to encounter smaller emission and transition probabilities and fall into the underflow problem.

The output files from log calculations are found in dev_log.pX.out for x = 3, 4.

## 2.4 Part 5: Design challenge

For the design challenge, we tried two different approaches: Structured Perceptron Algorithm and Average Perceptron Algorithm.

### 2.4.1 Structured Perceptron

We created a `StructuredPerceptron` class where we initialised a weights of type dictionary, the number of iterations to run through the sentences for, as well as training data, the output file path and all the possible tags from the training data.

```
class StructuredPerceptron:
    def __init__(self, trainfilepath):
        self.weights = defaultdict(int) # initialise a weight dictionary
        # self.goldfeaturevec = defaultdict(int) # initialise a gold standard feature vector
        self.iterations = 5 # number of iterations to run through all the sentences
        self.traindata = clean_trainset(trainfilepath)
        self.testfilepath = testfilepath
        self.tags = self.traindata.get_tags()
```

The `StructuredPerceptron` class has `train`, `predict`, `updateweights` and `get_structured_perceptron_path` (our version of Viterbi) functions. We will explain the functions in the following sections.

The **training** algorithm for the structured perceptron that we implemented is as follows:

1. For every sentence, run our version of Viterbi with the weights as input.
2. Get the feature vector of the predictions, as well as the feature vector of the actual sentence.
    - The feature vector contains both the tag transitions and emission values.
3. Update the weights.

4. Run through steps 1-3 for a pre-determined number of iterations. The code for the function is below:

```
def train(self):
    """
    Trains and updates the weight using the structured perceptron algorithm.
    :param traindata: Training data sequences (DATA TYPE TBC)
    :return: weights (dictionary)
    """
    traindata = self.traindata.outputsmootheddata()
    for i in range(self.iterations):
        for sentenceindex in range(len(traindata)):
            actualtagsentence = copy.deepcopy(traindata[sentenceindex])
            actualtagtransitions = [(traindata[sentenceindex][index][1], traindata[sentenceindex][index + 1][1])
                                    for index in range(len(traindata[sentenceindex]) - 1)]
```

```
sentencewithouttags = []
for word, tag in traindata[sentenceindex]:
    if tag != 'START':
        sentencewithouttags.append(word)
predtagsentence = self.get_structured_perceptron_path(sentencewithouttags)
# Get the sentence from Viterbi
predtagtransitions = [(predtagsentence[index][1], predtagsentence[index + 1][1])
                      for index in range(len(predtagsentence) - 1)]

# Get prediction feature vector
predfeaturevec = Counter(predtagsentence) + Counter(predtagtransitions)

# Get golden feature vector
goldfeaturevec = Counter(actualtagsentence) + Counter(actualtagtransitions)

self.updateweights(predtagsentence, actualtagsentence, predtagtransitions, actualtagtransitions,
                   predfeaturevec, goldfeaturevec)
```

The **update of weights** algorithm:

1. Iterate through the feature vector of the actual sentence and do the following:
    - If the tag was also in the feature vector of the predicted sentence, then we take value = the counts of the tag in the actual feature vector - counts of the tag in the predicted feature vector
    - If not, then value = count of the tag in the actual feature vector
    - We take the value variable and add it to the tag's weight.
2. We then iterate through the feature vector of the tag sentence and update the weights of any tags that were not in the actual tag sentence.
    - We check if the tag is in actual feature vector, and if it is, we continue on to the next tag so that we do not update the weights again.
    - If it is not, update the weight of the tag to be the weight - count of tag in predict feature vector.

The code for the function is below:

```
def updateweights(self, predtagsentence, actualtagsentence, predtagtransitions, actualtagtransitions,
                  predfeaturevec, goldfeaturevec, learning_rate=0.2):
    for tag in goldfeaturevec.keys():
        if tag in predfeaturevec.keys():
            self.weights[tag] = self.weights[tag] + (goldfeaturevec[tag] - predfeaturevec[tag])
        else:
            self.weights[tag] = self.weights[tag] + goldfeaturevec[tag]

    for tag in predfeaturevec.keys():
        if tag in goldfeaturevec.keys():
            # pass, because we already updated the weights
            continue
        else:
            self.weights[tag] = self.weights[tag] - predfeaturevec[tag]
```

The **Viterbi** algorithm for the structured perceptron is as follows:

The code is as follows:

```
def get_structured_perceptron_path(self, sentence):
    """ Get the best path from START to STOP as we go along

    Parameters:
        tags (list): All tags in the dataset
        sentence (list of str): Words for a specific sentence
        weights (defaultdict): Emissions, transtions and their weights

    Returns:
        predicted_sequence(list of tuples): Tuples contain word and tag for that particular index

    """
    # tags.insert(0,'START')
    tags = copy.deepcopy(self.tags)
    tags.pop(0)
    store_scores = np.zeros((len(tags), len(sentence)))
    predicted_sequence = []
    for index_words, word in enumerate(sentence):
        for index_tags, tag in enumerate(tags):
            if index_words == 0:
                # First word needs to refer to START as the transition to first TAG
                score = self.weights[('START', tag)] + self.weights[(word, tag)]
            else:
                # Take the max of the previous layer as the first tag
                previous_layer_max_value = np.max(store_scores, axis=0)[index_words - 1]
                # Find the index of the max of the previous layer and get its tag
                previous_layer_max_tag = tags[np.argmax(store_scores, axis=0)[index_words - 1]]
                score = previous_layer_max_value + self.weights[(previous_layer_max_tag, tag)] + self.weights[(word, tag)]
            store_scores[index_tags][index_words] = score
        current_layer_max_tag = tags[np.argmax(store_scores, axis=0)[index_words]]
        predicted_sequence.append((word, current_layer_max_tag))
    return predicted_sequence
```

The prediction algorithm is:

1. Run the viterbi algorithm with the latest training weights for each sentence in the test set, and get a list of predicted sentences.
2. Write each predicted sentence to a file.

The code is as follows:

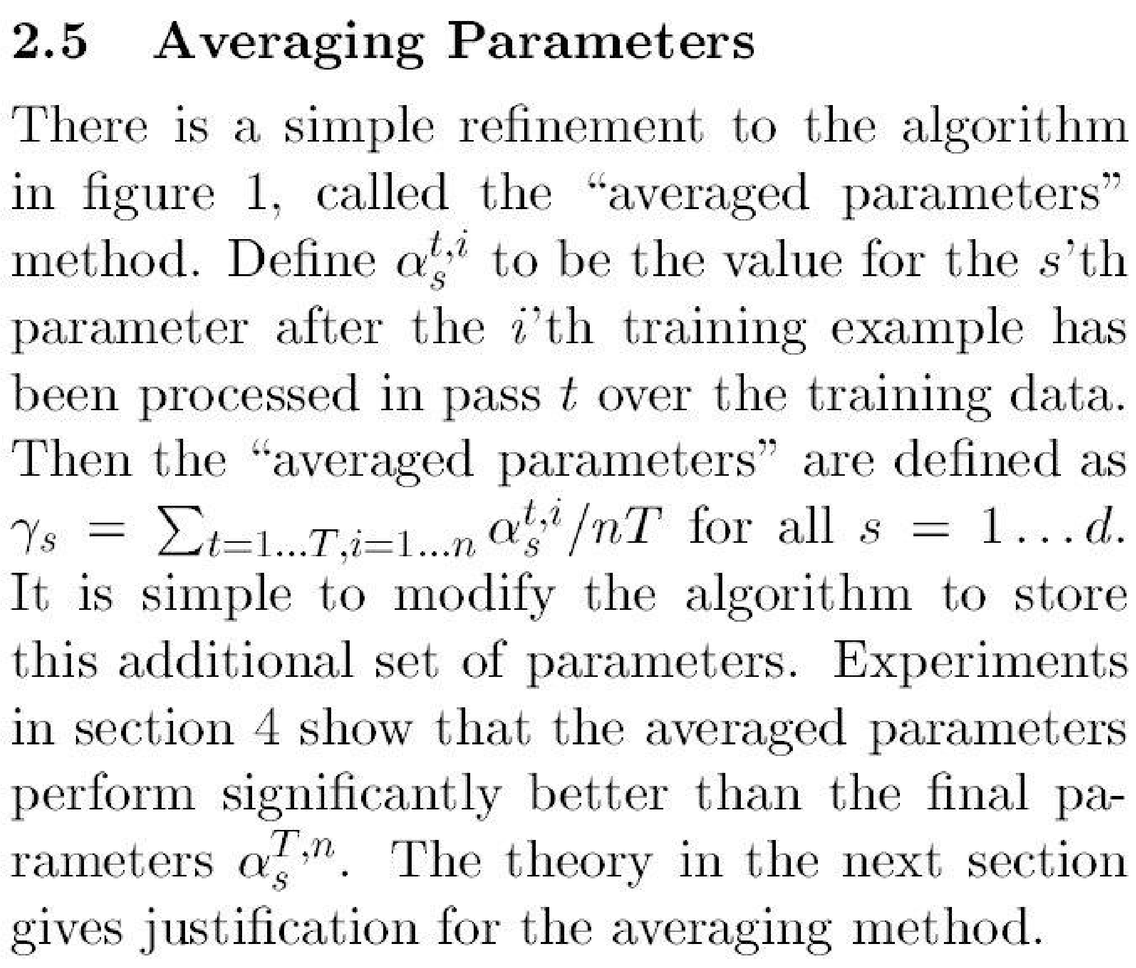
```
def predict(self, testfilepath):
    testdata = clean_testset(testfilepath + '/dev.in', self.traindata.smoothed)
    alltestsentences = testdata.get_all_sentences()
    predictions = []
    for sentence in alltestsentences:
        predictedsentence = self.get_structured_perceptron_path(sentence)
        predictions.append(predictedsentence)

    output_file = self.testfilepath + "/test.p5.out"
    with open(output_file, "w", encoding="utf8") as f:
        for sentence in predictions:
            for predtuple in sentence:
                word, tag = predtuple
                f.write(word + " " + tag + "\n")
            f.write("\n")
    f.close()
```

The limitations of our implementation are:

- We did not implement this for trigrams, only for bigrams, thus the accuracy is acceptable, but lower than Viterbi.
- We did not average the weights

We will address these implementations in our next implementation, averaged perceptron.

### 2.4.2 Average Perceptron



#### 2.5 Averaging Parameters

There is a simple refinement to the algorithm in figure 1, called the "averaged parameters" method. Define $\alpha_s^{t,i}$ to be the value for the $s$'th parameter after the $i$'th training example has been processed in pass $t$ over the training data. Then the "averaged parameters" are defined as $\gamma_s = \sum_{t=1\ldots T,i=1\ldots n} \alpha_s^{t,i}/nT$ for all $s = 1\ldots d$. It is simple to modify the algorithm to store this additional set of parameters. Experiments in section 4 show that the averaged parameters perform significantly better than the final parameters $\alpha_s^{T,n}$. The theory in the next section gives justification for the averaging method.

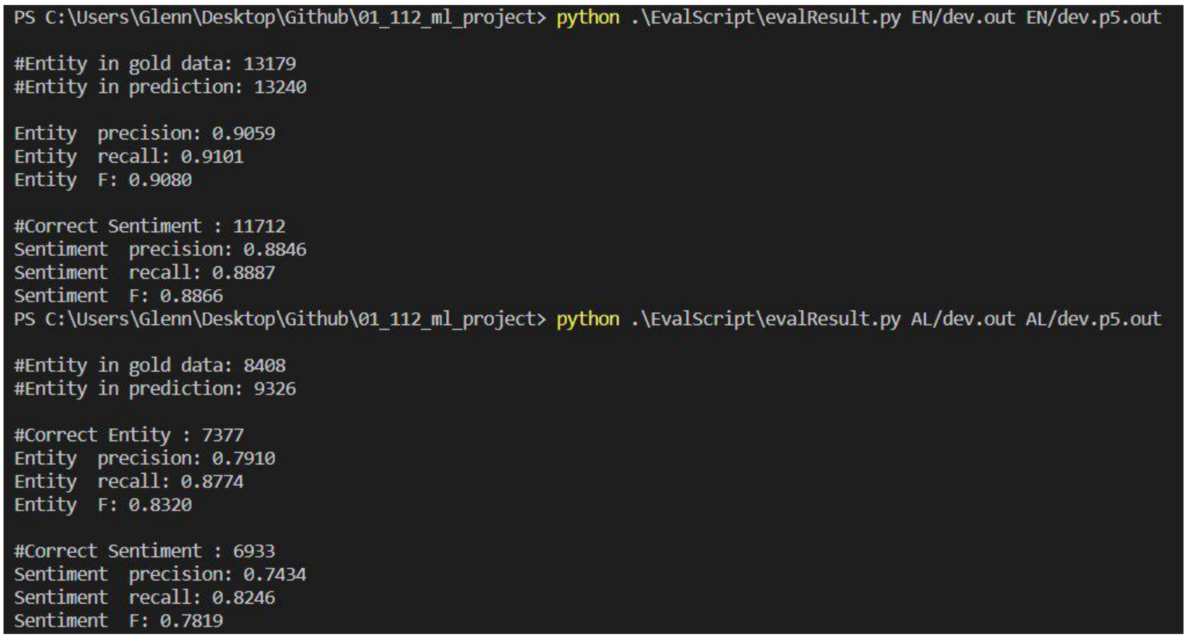
$$\gamma_s = \frac{\sum_{t=1...T, i=1...n} \alpha_s^{t,i}}{nT}$$

According to the literature provided in the project description, the averaging method has been proven to achieve better accuracy. We this added it on top of our existing structured perceptron. We also permutated various features to find the ideal combination of features (We commented out the features that led to a decrease in accuracy). A naive approach would be for each word in the training set, we look at all the features for every word and add it cumulatively. However, this would mean be computationally expensive. Hence, instead, we keep a global count of the total number of updates and a count for when each word's features were updated. Whenever that word is updated, we update the "global weight" of the that word by taking the product of the difference in total time and timestamp multiplied by the weight before the update. After this, we then update the weight according to the standard structured perceptron. Subsequently, at the end of all iterations, we have the global count of the number of updates and we also have the accumulated weight over all words and iterations. A simple division as per the formula above gets the averaged weight which is proven to be more accurate than just using the final weight given by a "structured perceptron" method.

### 2.4.3 Average Perceptron Results

Below are the results from running the average perceptron code:

For EN and then AL dataset:



```
PS C:\Users\Glenn\Desktop\Github\01_112_ml_project> python .\EvalScript\evalResult.py EN/dev.out EN/dev.p5.out

#Entity in gold data: 13179
#Entity in prediction: 13240

Entity  precision: 0.9059
Entity  recall: 0.9101
Entity  F: 0.9080

#Correct Sentiment : 11712
Sentiment  precision: 0.8846
Sentiment  recall: 0.8887
Sentiment  F: 0.8866
PS C:\Users\Glenn\Desktop\Github\01_112_ml_project> python .\EvalScript\evalResult.py AL/dev.out AL/dev.p5.out

#Entity in gold data: 8408
#Entity in prediction: 9326

#Correct Entity : 7377
Entity  precision: 0.7910
Entity  recall: 0.8774
Entity  F: 0.8320

#Correct Sentiment : 6933
Sentiment  precision: 0.7434
Sentiment  recall: 0.8246
Sentiment  F: 0.7819
```

## References

- Collins, M. (2002). Discriminative training methods for hidden Markov models. Proceedings of the ACL-02 Conference on Empirical Methods in Natural Language Processing - EMNLP 02. doi: 10.3115/1118693.1118694
- Boyd-Graber, J. (n.d.). Retrieved from http://users.umiacs.umd.edu/~jbg/teaching/CMSC_726/11d.pdf.