01.112 Machine Learning

Spring 2018

Design Project Report

Chee Rui Yi (1001738)
Maylizabeth (1001818)
Regina Lim (1001789)

In design project, we are required to design the sequence labelling model for informal texts to label each word in texts with a set of labels: O, B-negative, B-positive, B-neutral, I-negative, I-positive, I-neutral. B is used to indicate Beginning of the entities; I is used to indicate the Inside of the entities and O is used to indicate the Outside of any entity. Also, the negative, positive and neutral are used to indicate entities which are associated with the negative, positive and neutral sentiment, respectively.

The team was given four datasets, which for each dataset, a labelled *training* set *train*, an unlabelled development set *dev.in*, and a labelled development set *dev.out*. The labelled data has the format of one word per line with word and tag separated by tab and a single empty line that separates sentences.

The objectives of the project are to:

- Develop two sentiment analysis systems for two different languages from scratch, using own annotations
- Develop another two sentiment analysis systems for two different languages using annotations provided by others

PART 2

In Part two of the project, the team was required to write a function that estimates the emission parameters from the *training* set using MLE (maximum likelihood estimation):

$$e(x|y) = \frac{\texttt{Count}(y \rightarrow x)}{\texttt{Count}(y)}$$

However, as there are some words that appear in the *test* set but do not appear in the *training* set. Hence, this will cause an issue with estimating the emission parameters. A special word token #UNK# is introduced. If the word does not appear in the *training* set, the word is replaced with #UNK# in the *test* set.

```
def modified_test(trainfile, testfile):
    raw_train = open(trainfile, 'r+',
                     encoding="utf8")

    words_in_train = []
    for words2 in raw_train:
        words3 = words2.split('\n')
        for i in range(len(words3)):
            spt = words3[i].split(" ")
            if len(spt) > 3 or len(spt) < 2:
                continue
            # print(spt)
            key1 = spt[0]
            words_in_train.append(key1)

    # print(words_in_train)
    raw_test = open(testfile, 'r+',
                    encoding="utf8")
    words_in_test = []
    read_lines = raw_test.read()
    lines = read_lines.split('\n')
    for words in lines:
        try:
            words_in_test.append(words)
        except:
            words_in_test.append('')
    modified_test = words_in_test

    kcount = 0
    for x in range(len(words_in_test)):
        if words_in_test[x] == '':
            modified_test[x] = modified_test[x]
        elif words_in_test[x] not in words_in_train:
            modified_test[x] = '#UNK#'
            kcount += 1

    return modified_test, kcount
```

The *modified_test* function will take in the *train* file and the *test* file. For the *train* file, the function will strip the sentences in the file using split() and append in every word that is available in the *train* set into the *words_in_train* list. The *words_in_train* list will be used to verify whether the words in the *test* file are available in the *train* file.

For the *test* file, the function will strip the words in the file using split(). The, the words will be appended into the *words_in_test* list where if the words are not in the *words_in_train* list, the words will be replaced by #UNK# in the *words_in_test* list, otherwise, the same words will be appended into the *words_in_test* list. As the empty lines between the sentences are important features because they indicate where the sentences in the file start and end, the ‘ ‘ will be appended into the *words_in_test* list, each time a empty line is spotted. This will help to indicate the start and the end of the sentences in the Viterbi algorithm on the later part. The number of #UNK# is also counted in the loop as kcount where the kcount after the loop is the total number of #UNK# in the entire *test* file.

In the end, the *modified_test* (*words_in_test* list; the list containing all the words and empty lines of the *test* file) and the kcount are returned.

The computation of emission probabilities is modified:

$$e(x|y) = \begin{cases} \frac{\text{Count}(y \to x)}{\text{Count}(y)+k} & \text{If the word token } x \text{ appears in the training set} \\ \frac{k}{\text{Count}(y)+k} & \text{If word token } x \text{ is the special token } \texttt{\#UNK\#} \end{cases}$$

It is assumed that from any label y, there is a certain chance of generating #UNK# as a rare event, and empirically it is also assumed that there are k occurrences of such an event. Basically, k is the number

of #UNK# in the modified *test* set. As mentioned, the *modified_test* function returns the kcount as the number of #UNK# in the modified *test* set.

After that, the *train* set and the kcount that was returned from the *modified_test* function, were inputted into the *emission_par* function which will then calculate the emission probabilities. The *train* set has to be open and split by lines using *get_data* function first before it can be inputted into the *emission_par* function.

```
def get_data(data):
    raw_data = open(data, 'r',
                    encoding="utf8")
    d = raw_data.read()
    words = d.split('/n')
    return words
def emission_par(words,kcount):
    global ggdict
    global nesteddict

    totalcountBpos = 0
    totalcountBneg = 0
    totalcountBneu = 0
    totalcountIpos = 0
    totalcountIneg = 0
    totalcountIneu = 0
    totalcountO = 0
    tag_count = {}
    for words2 in words:
        globaldict = {}
        words3 = words2.split('\n')  ##words3 is a list of pairs of words and their tags
        for i in range(len(words3)):
            spt = words3[i].split(" ")
            if len(spt) > 3 or len(spt) < 2:
                continue
            # print(spt)
            key1 = spt[0]
            label1 = spt[1]
            if key1 not in globaldict:
                globaldict[key1] = nesteddict
                globaldict[key1][label1] += 1
            nesteddict = {"B-positive": 0, "B-negative": 0, "B-neutral": 0, "I-positive": 0, "I-negative": 0,
                          "I-neutral": 0, "O": 0}

            if label1 == 'B-positive':
                totalcountBpos += 1
            elif label1 == 'B-negative':
                totalcountBneg += 1
            elif label1 == 'B-neutral':
                totalcountBneu += 1
            elif label1 == 'I-positive':
                totalcountIpos += 1
            elif label1 == 'I-negative':
                totalcountIneg += 1
            elif label1 == 'I-neutral':
                totalcountIneu += 1
            else:
                totalcountO += 1
```

Three dictionaries are used in the *emission_par* function:

1. nesteddict: a dictionary that take in all the tags as the key and the number of each tag as the value; the initial state of the dictionary is {“B-positive”: 0 , “B-negative”: 0, “B-neutral”: 0, “I-positive”: 0, “I-negative”: 0, “I-neutral”: 0, “O”: 0}
2. globaldict: a subset of ggdict that includes a portion of all the words. We do not know why this happens, but when the code runs, a few dictionaries containing different words were generated.
3. ggdict: a dictionary that take in the words as the key and the nesteddict as the value; eg. {“Netflix”: {“B-positive”: 1 , “B-negative”: 0, “B-neutral”: 0, “I-positive”: 0, “I-negative”: 0, “I-neutral”: 0, “O”: 0},
   “and”: {“B-positive”: 0 , “B-negative”: 0, “B-neutral”: 0, “I-positive”: 0, “I-negative”: 0, “I-neutral”: 0, “O”: 1},

"cats": {"B-positive": 0 , "B-negative": 0, "B-neutral": 0, "I-positive": 0, "I-negative": 0, "I-neutral": 0, "O": 1}}

words3 is the list of pairs of words and their tags: ["Netflix B-positive", "and O", "cats O", …]

spt is obtained after splitting each words3 in a loop. Here are the two examples:

words3	**spt[0]**	**spt[1]**
"Nexflix B-positive"	"Nexflix"	"B-positive"
"and O"	"and"	"O"

The team also need to consider that there are more than one space within a line. The code to take on this argument follows: if (len(spt)) > 3 or len(spt) < 2: continue

The spt[0] is appended into the globaldict as a key if it is not in the globaldict yet, with the nesteddict as the value. Within the globaldict value (globaldict[spt[0]][spt[1]]), the value increases by 1 based on the tag. For example, spt[0] = "Netflix" and spt[1] = "B-positive". So the globaldict[spt[0]] will identify the "Netflix" key if there is and add in 1 to the globaldict[spt[0]][spt[1]] to the "B-positive" key in the nesteddict. The outcome will be: {"Netflix": {"B-positive": 1 , "B-negative": 0, "B-neutral": 0, "I-positive": 0, "I-negative": 0, "I-neutral": 0, "O": 0}}

The total count for each tag is also calculated using spt[1]. If spt[1] is "B-positive", the total count for B-positive will be increased by 1.

```
    for k in globaldict:  ##k refers to the words in the smaller global dictionary
        vdict = globaldict[k]  ##vdict refers to the existing dictionary of label counts for the particular word k
        if k not in ggdict:  ## if the word k does not exist in the global global dictionary
            ggdict[k] = vdict  ## update the global global dictionary with the existing dictionary of label counts
        else:
            for k1, v in vdict.items():
                ggdict[k][k1] += v  ##add the label counts to the global global dictionary
    ggdict['#UNK#']= {"B-positive": kcount, "B-negative": kcount, "B-neutral": kcount, "I-positive": kcount,
                      "I-negative": kcount, "I-neutral": kcount, "O": kcount}
    # Calculating the emission parameters
    for word, tagdict in ggdict.items():
        for tag, count in tagdict.items():
            if tag == 'B-positive':
                ggdict[word][tag] = ggdict[word][tag] / (totalcountBpos +kcount)
            elif tag == 'B-negative':
                ggdict[word][tag] = ggdict[word][tag] / (totalcountBneg +kcount)
            elif tag == 'B-neutral':
                ggdict[word][tag] = ggdict[word][tag] / (totalcountBneu +kcount)
            elif tag == 'I-positive':
                ggdict[word][tag] = ggdict[word][tag] / (totalcountIpos +kcount)
            elif tag == 'I-negative':
                ggdict[word][tag] = ggdict[word][tag] / (totalcountIneg +kcount)
            elif tag == 'I-neutral':
                ggdict[word][tag] = ggdict[word][tag] / (totalcountIneu +kcount)
            else:
                ggdict[word][tag] = ggdict[word][tag] / (totalcountO +kcount)
# print("ggdict: %s" %ggdict)
# build dictionary of total tag counts for each tag
tag_count = {"B-positive": totalcountBpos, "B-negative": totalcountBneg, "B-neutral": totalcountBneu,
                    "I-positive": totalcountIpos, "I-negative": totalcountIneg,
                      "I-neutral": totalcountIneu, "O": totalcountO}
return ggdict, tag_count
```

One puzzling thing that occured was that when the code runs, the globaldict did not include all the 74255 lines. Instead, multiple globaldict dictionaries with different words were generated. Therefore,

there needs to be another bigger dictionary, ggdict (short for global global dictionary) to combine all these smaller dictionaries into one dictionary for the emission parameters.

After obtaining the emission parameters, a simple sentiment analysis system is developed to produce the tag for each word x in the sequence.

$$y^* = \arg\max_y e(x|y)$$

The tag with the maximum emission parameter is assigned to each word.

```
def sentiment_analysis(test_file, emission_params):
  #test_file is dev.in file , output_file is the output of the maximized probabilities of
  #the tags for the words in dev.in, emission_params is the ggdict
    testwords = []
    tag = []
    counter = 0

    for w in test_file:
        counter += 1
        if w in emission_params.keys():
            tagdict = emission_params[w]
            testwords.append(w)
            tag.append(max(tagdict,key=tagdict.get))
        elif w == "#UNK#":
            testwords.append("#UNK#")
            UNKtag = max(emission_params['#UNK#'], key= emission_params['#UNK#'].get)
            tag.append(UNKtag)
        else:
            testwords.append("")
            tag.append("")


    f = open("dev.p2.out", "w+", encoding="utf8")
    for i in range(0,counter):
        f.write("%s %s\n" % (testwords[i],tag[i]))
    f.close()
```

The sentiment analysis function takes the *test* file (*dev.in* file) and the emission parameters dictionary (ggdict) as inputs, and generates a *get_data* file as output. To initialise this, two empty lists are created, testwords for the words in the *test* file, tag to store the label that has the highest probability. Lists were chosen instead of dictionaries because lists are ordered, an important feature because we would want the output file to be in the same order as the input file.

The function will iterate through every line of the *test* file, add the words in the *test* file into the testwords list, then reference the keys of the emission parameters, and append the corresponding label with the highest probability into the tag list.

The function will iterate through every line of the *test* file and does either of the three of the following conditions:

1. If the word is in the ggdict (word is a key in the ggdict), the word is appended into the testwords list, then reference the keys of the emission parameters and append the corresponding tag with the highest probability into the tag list. The code for the tag with the maximum probability is max(tagdict, key=tagdict.get) where the tagdict is the nesteddict, a dictionary of tags {“B-positive”: 0 , “B-negative”: 0, “B-neutral”: 0, “I-positive”: 0, “I-negative”: 0, “I-neutral”: 0, “O”: 1}
2. Else if the word is not in the ggdict and it is #UNK#, the #UNK# is appended into the testwords list and the tag with the maximum probability is appended into the tag list.

3. Else, it is assumed that the words is the empty line and " " is appended into the testwords and tag lists. This is to make sure that he indexes of both input and output files are the same, for accurate calculation of precision.

Finally, the function ends by generating an output file *get_data* with every i^{th} line writing testwords[i] and tag[i], the word and its corresponding label.

For all the four datasets RU, ES, CN and EN, the emission parameters are learned with *train* and the simple sentiment analysis system is evaluated on the development set *dev.in* for each of the dataset. The output is written to *get_data* for each dataset, which it is compared with the gold-standard outputs in dev.out and the following precision, recall and F scores of the system for each dataset follow:

Results

RU

```
#Entity in gold data: 461
#Entity in prediction: 1164

#Correct Entity : 253
Entity  precision: 0.2174
Entity  recall: 0.5488
Entity  F: 0.3114

#Correct Sentiment : 133
Sentiment  precision: 0.1143
Sentiment  recall: 0.2885
Sentiment  F: 0.1637
```

ES

```
#Entity in gold data: 255
#Entity in prediction: 734

#Correct Entity : 135
Entity  precision: 0.1839
Entity  recall: 0.5294
Entity  F: 0.2730

#Correct Sentiment : 63
Sentiment  precision: 0.0858
Sentiment  recall: 0.2471
Sentiment  F: 0.1274
```

CN

```
#Entity in gold data: 349
#Entity in prediction: 1402

#Correct Entity : 153
Entity  precision: 0.1091
Entity  recall: 0.4384
Entity  F: 0.1748

#Correct Sentiment : 77
Sentiment  precision: 0.0549
Sentiment  recall: 0.2206
Sentiment  F: 0.0879
```

EN

```
#Entity in gold data: 878
#Entity in prediction: 2414

#Correct Entity : 486
Entity  precision: 0.2013
Entity  recall: 0.5535
Entity  F: 0.2953

#Correct Sentiment : 216
Sentiment  precision: 0.0895
Sentiment  recall: 0.2460
Sentiment  F: 0.1312
```

PART 3

In part three of the project, the team was required to write a function that estimates the transition parameters from the *training* set using MLE (maximum likelihood estimation):

$$q(y_i|y_{i-1}) = \frac{\text{Count}(y_{i-1}, y_i)}{\text{Count}(y_{i-1})}$$

Special cases are also considered: $q(\text{STOP}|y_n)$ and $q(y_1|\text{START})$.

```
def transition_params(train_file):
    transition_count ={}
    state_count = {}
    u_state = 'START'
    v_state = 'STOP'

    state_count[u_state] = 0
    state_count[v_state] = 0
    transition_count[v_state] = {}

    with open(train_file, encoding='utf=8') as file:
        for line in file:
            word_pair= line.split()
            if len(word_pair) !=0 and len(word_pair) <3:
                value = word_pair[1]
                #print (value)

                #add value into transition count
                if value in transition_count.keys():
                    value_list=transition_count[value]

                    #count occurence of values
                    if u_state in value_list.keys():
                        value_list[u_state] += 1
                    else:
                        value_list[u_state] = 1

                #for new keys(words)
                else:
                    new_value = {}
                    new_value[u_state] = 1
                    transition_count[value] = new_value

                #add start and stop state counts
                if u_state == 'START':
                    state_count[u_state] += 1
                    state_count[v_state] += 1

                #add to state counts
                if value in state_count.keys():
                    state_count[value] += 1
                else:
                    state_count[value] = 1

                u_state = value

            else:
                value_list=transition_count[v_state]
                if u_state in value_list.keys():
                    value_list[u_state] +=1
                else:
                    value_list[u_state] =1
                u_state = 'START'
```

The *transition_params* function initialises the state_count with the ‘START’ and ‘STOP’ to take the special cases into account. The transition parameters consist of the probabilities that there is a state change from ‘START’ state and any tags to any other tags, and state change from any tags to ‘STOP’ state.

For each line in the file, as long as the length of the word pair is 1 or 2, the tag (called as value) is taken and added into the transition_count dictionary under 2 conditions:

1. If the tag is already in the transition_count dictionary as a key, its value, which is a dictionary too, will take in the u_state as a key and the number of occurrences of u_state as the value (so

each time, the value is incremented by one). After which, the u_state is updated by replacing the current u_state with the new state (which is the value)

2. Else, it is assumed that the tag is not in the transition_count dictionary as a key, so new key is created in the transition_count dictionary, with the value being another dictionary of u_state (called as new_value dictionary). After which, the u_state is updated by replacing the current u_state with the new state (which is the value).

The 'Start' and 'STOP' states need to be taken into account in the state_count.

In addition, as long as the length of the word pair is 1 or 2, the tag (called as value) is also added into the state_count dictionary under 2 conditions:

1. If the tag is in state_count dictionary as a key, the value is incremented by one.
2. Else, it is assumed that the tag is not in the state_count dictionary; a new key is added to the dictionary with the value 1.

Else, if the length of the word pair is not 1 or 2, it is assumed that it is the empty line, indicating the end of the sentence. The value of the u_state key in the value_list dictionary (a value of the transition_count dictionary for v_state as a key) is incremented by one if the u_state is in the value_list dictionary. Otherwise, a new key with its value = 1 is created. The u_state is replaced to 'START' because the current sentence ends and a new sentence might appear.

After a series of loops, the probabilities are calculated by dividing all the numerical values in nested dictionary of transition_count dictionary with state_count values for each u_state. The transition parameters dictionary is returned, which contains the states and the probabilities.

The Viterbi algorithm is developed using the estimated transition and emission parameters to compute the following for a sentence with n words.

```
def viterbi_algo(test_file, transition_par, emission_par, tags):
    sentences = []
    input_file = open(test_file, encoding = 'utf-8')
    sentence = []
    # Make use of empty line between sentences to get whole sentence
    for line in input_file:
        if len(line.split()) != 0:
            sentence.append(line.split()[0])
        else:
            sentences.append(sentence)
            sentence = []
    # print(sentences)

    output_file = open("dev.p3.out", "w+", encoding="utf8")
    for s in sentences:
        nodes = calculate_node_scores(s,transition_par,emission_par, tags)
        labelled_sentence = backtracking(s,nodes)

        for word in labelled_sentence:
            output_file.write(word + '\n')
        output_file.write('\n')
    output_file.close()
```

The input to the *viterbi_algo* function is the *dev.in*, transition parameters dictionary returned by the *transition_params* function, emission parameters dictionary returned by the *emission_par* function and the tag_count dictionary returned by the *emission_par* (which is a dictionary that takes the tags as the keys and the total number of tags in the *test* file as value for each key).

In the *viterbi_algo* function, the *dev.in* (*test* file) is stripped into the sentences by taking the empty lines as indicators for the start and end of the sentences. Then the sentence is appended into the the sentence list.

Each word in each sentence is a state/node. For each node, the node scores are calculated using the *calculate_node_scores* function:

```
def calculate_node_scores(s, transition_par, emission_par, tags):
    nodes = {}
    # base case
    nodes[0] = {'START':[1,'Nil']}
    # recursive
    for k in range (1, len(s)+1): # for each word
        X = s[k-1] # previous node
        for V in tags.keys():# for each node
            prev_nodes_dict = nodes[k-1] # access prev nodes
            highest_score = 0 # initiatise
            parent = 'Nil' # initiatise
            # emission parameters
            if X in emission_par.keys():
                # save the emission parameter dictionary for that word as emission_labels
                emission_labels = emission_par[X]
                # if V, the label node, is in emission_labels
                # in case the there is no node V (label) in the emission_label
                if V in emission_labels:
                    b = emission_labels[V]
                else:
                    b = 0
            else:
                # take #UNK# into account as they are missing words
                b = emission_par['#UNK#'][V]

            # transition parameters
            for U in prev_nodes_dict.keys(): # taking the previous node U saved
                prev_states = transition_par[V]
                # if U in prev_states (transition para dictionary): then there is an edge
                if U in prev_states.keys():
                    a = prev_states[U]
                else:
                    # consider the case where U is not in the prev_states; probability = 0
                    a = 0

                # Calculate prev node score
                prev_score = prev_nodes_dict[U][0]
                score = prev_score*a*b
                # Adjust the highest score accordingly
                if score >= highest_score:
                    highest_score = score
                    parent = U # previous node
            if k in nodes.keys():
                nodes[k][V] = [highest_score, parent] # update the highest_score, parent cos iterat
            else:
                new_dict = {V:[highest_score,parent]}
                nodes[k] = new_dict
    # end case
    prev_nodes_dict = nodes[len(s)] # last layer
    highest_score = 0 # initialise end node
    parent = 'None' # initialise end node
    # take the U node in order to do transition parameters
    # transition parameters
    for U in prev_nodes_dict.keys():
        prev_states = transition_par['STOP'] # get the probabilities of tags dictionary
        if U in prev_states.keys():
            a = prev_states[U]
        else:
            a = 0
        # prev node score
        prev_score = prev_nodes_dict[U][0]
        score = prev_score*a # there is no output word
        if score >= highest_score:
            highest_score = score
            parent = U # previous node
    indiv_node = {'STOP': [highest_score, parent]}
    nodes[len(s)+1] = indiv_node

    return nodes
```

The *calculate_node_scores* take in the sentence (called as s), the transition parameter dictionary returned by the *transition_params* function, the emission parameter dictionary returned by the *emission_par* function and the tags (which is a dictionary that takes the tags as the keys and the total number of tags in the *test* file as value for each key). The initialisation is {'START': [1,'Nil']} where the 'START' state will always be the first state of the entire sequence, with no state before it as indicated by Nil.

The iteration through the sentence s is done where the k is the word in the sentence s that is in being iterated. Then the keys in the tag dictionary is iterated in order to take all the tags (B-positive, B-negative, B-neutral, I-positive, I-negative, I-neutral, O) into consideration. The previous state (called as X in the code) in the sentence s is taken while we focus on the second word of the sentence s. This is the initialisation, with the highest_score = 0 and parent (state) = 'Nil'.

Then the code for the calculation of variable b based on the emission parameters starts, which consists of two conditions (with the tags iteration: for V in tags.keys():):

1. If the previous state is in the emission parameters dictionary, there are two conditions:
    a. If the V is in the nested dictionary (this dictionary is the value of the emission parameters dictionary with the X as the key.), the variable b will be the value of this nested dictionary (which is the emission probability for that word).
    b. Else, the variable b will have a value of 0.
2. Else, it is assumed that the word is #UNK# and the variable b will be the emission probability for #UNK#.

Then the calculation of the variable a based on the transition parameters follows. Within the loop, the score for that state is calculated based on the previous score, the variable a and the variable b.
For each previous state U saved, two conditions in the calculation of the variable a are:

1. If the U is in the nested dictionary (this dictionary is the value of the transition parameters dictionary with the V as the key where V is in the tag iteration), the variable a will be the value of this nested dictionary (which is the transition probability of that previous state).
2. Else, it is assumed that U is not in the prev_states; the variable a is zero,

This is followed by the calculation of the score of the previous state. If this score is higher than the highest score, the highest score will be updated by being replaced by the new highest score. Lastly, the highest score and the parent state is updated/created in the nodes dictionary.

Finally, the end case needs to be considered and will be different from the 'START' case and middle case. For the end case, the initialisation occurs with the prev_nodes_dict dictionary taking the value of the last layer key, the highest_score = 0 and the parent state is 'None'. We take in the previous state to calculate the variable a from the transition probabilities. For each U in the prev_nodes_dict.keys(), two conditions are followed:

1. If U is in the nested dictionary (this dictionary is the value of the transition parameters dictionary with the 'STOP' state as the key.), the variable a will be the value of this nested dictionary (which is the transition probability of the previous state.)
2. Else, it is assumed that U is not in the nested dictionary; the variable a is zero.

Then the score of the previous state is calculated. If this score is higher than the highest score, the highest score will be updated by being replaced by the new highest score. Lastly, the highest score and the parent state for the 'STOP' state is updated/created in the nodes dictionary.

The nodes dictionary containing all the states, their highest scores and their parent states is returned from the *calculate_node_scores* function.

Going back to the *viterbi_algo* function, each word in the sentence will be labelled through the backtracking function which takes the sentence s and the nodes dictionary returned by the *calculate_node_scores* function. The function will take the 'STOP' state first then iterate the sequence backwards, one state at a time. This assigns the tag with the highest score to each word. Lastly, the sentence s list (containing the words and their assigned tags) is returned as the labelled_sentence in the *viterbi_algo* function.

Finally, the labelled_sentence list is written to the *dev.p3.out* file.

For all datasets, the model parameters are learned with *train*. Then the Viterbi algorithm is ran on the development set *dev.in* using the learned models and output is written to *dev.p3.out* for the four datasets respectively.

Results

RU

```
#Entity in gold data: 461
#Entity in prediction: 445

#Correct Entity : 190
Entity  precision: 0.4270
Entity  recall: 0.4121
Entity  F: 0.4194

#Correct Sentiment : 119
Sentiment  precision: 0.2674
Sentiment  recall: 0.2581
Sentiment  F: 0.2627
```

ES

```
#Entity in gold data: 255
#Entity in prediction: 279

#Correct Entity : 107
Entity  precision: 0.3835
Entity  recall: 0.4196
Entity  F: 0.4007

#Correct Sentiment : 57
Sentiment  precision: 0.2043
Sentiment  recall: 0.2235
Sentiment  F: 0.2135
```

CN

```
#Entity in gold data: 349
#Entity in prediction: 505

#Correct Entity : 113
Entity  precision: 0.2238
Entity  recall: 0.3238
Entity  F: 0.2646

#Correct Sentiment : 78
Sentiment  precision: 0.1545
Sentiment  recall: 0.2235
Sentiment  F: 0.1827
```

EN

```
#Entity in gold data: 878
#Entity in prediction: 809

#Correct Entity : 285
Entity  precision: 0.3523
Entity  recall: 0.3246
Entity  F: 0.3379

#Correct Sentiment : 161
Sentiment  precision: 0.1990
Sentiment  recall: 0.1834
Sentiment  F: 0.1909
```

PART 4

In part 4, the team has modified the viterbi algorithm to store the top k pi scores, the respective parents node and the index of the score under the *viterbi_kbest* function. The *viterbi_kbest* function takes in the input file, output file, transition and emission parameters, list of tags from part 3 and additional 2 arguments: top_k and i_th.

Top_k corresponds to the number of best paths to store (eg. top_5 corresponds to the top 5 paths that return the max score in that order) and i_th corresponds to the output of the ith best output from the top k paths calculated. The output file will be in the form of *dev.p4.out*.

```
def viterbi_kbest(test_file, transition_par, emission_par, tags, topk, kth):
    sentences = []
    input_file = open(test_file, encoding = 'utf-8')
    sentence = []
    # Make use of empty line between sentences to get whole sentence
    for line in input_file:
        if len(line.split()) != 0:
            sentence.append(line.split()[0])
        else:
            sentences.append(sentence)
            sentence = []

    output_file = open("dev.p4.out", "w+", encoding="utf8")
    for s in sentences:
        nodes = calculate_node_scores(s,transition_par,emission_par, tags, topk)
        labelled_sentence = backtracking(s,nodes, kth)

        for word in labelled_sentence:
            output_file.write(word + '\n')
        output_file.write('\n')
    output_file.close()
```

In general, the logic for *viterbi_kbest* function goes as follows:

1. Initialize empty dictionary for nodes with nodes = {k: {tags: [[score, parent, index] , …], … }, … }
2. Read the file, append each sentence as a list, into a list of sentences
3. For each sentence, apply Viterbi algorithm and backtrack to find the optimal sentiments for the words
4. Output each labelled sentence into the output file

```
def calculate_node_scores(s, transition_par, emission_par, tags, topk):
    nodes = {}
    # base case
    nodes[0] = {'START':[[1,'Nil',0]]}
    # recursive
    for k in range (1, len(s)+1): # for each word
        X = s[k-1] # previous node
        for V in tags.keys():# for each node
            prev_nodes_dict = nodes[k-1] # access prev nodes
            # emission parameters
            if X in emission_par.keys():
                # save the emission parameter dictionary for that word as emission_labels
                emission_labels = emission_par[X]
                # if V, the label node, is in emission_labels
                # in case the there is no node V (label) in the emission_label
                if V in emission_labels:
                    b = emission_labels[V]
                else:
                    b = 0
            else:
                # take #UNK# into account as they are missing words
                b = emission_par['#UNK#'][V]

            scores =[]

            # transition parameters
            for U in prev_nodes_dict.keys(): # taking the previous node U saved
                prev_states = transition_par[V]
                # if U in prev_states (transition para dictionary): then there is an edge
                if U in prev_states.keys():
                    a = prev_states[U]
                else:
                    # consider the case where U is not in the prev_states; probability = 0
                    a = 0
```

We calculate each node score using the function *calculate_node_scores*, where the base case and the forward recursive segment also remains the same as part 3, and we calculate the value of the transition and emission parameter respectively. (same as part 3)

```
                index = 0
                # Calculate prev node score
                for prev_k_nodes in prev_nodes_dict[U]:
                    prev_score = prev_k_nodes[0]
                    score = prev_score*a*b
                    scores.append([score, U, index])
                    index += 1

            # take top k scores
            scores.sort(key=lambda x:x[0], reverse=True)
            topk_scores = scores[:topk]
            if k in nodes.keys():
                nodes[k][V] = topk_scores
            else:
                new_dict = {V:topk_scores}
                nodes[k] = new_dict
```

The modification starts from calculating the previous node score segment, where we (#take top k scores):

- Initialise index=0 and scores= [] to store the scores of each nodes with iterable index
- For every prev_k_nodes loop through in the prev_nodes_dict, the previous node score will be assigned as prev_k_nodes score
- Compute the score for that particular node and the score obtained is appended into the score list and the index is also incremented by 1.
- Sort through the scores in ascending order.

If the word is in the key lists of the node dictionary, update the score to its highest score for every iteration. If not, add the word and its respective values.

For the end case, obtain the transition probability from the end layer for each nodes to ‘STOP’ node. If there is no transition from a node to the ‘STOP’ node, the value of the transition probability is 0. To calculate the score of that particular last layer, we multiply the score of that node to the transition probability to ‘STOP’ node and increment the index by 1 for every words. Sort through the values in

ascending order, pick the top_k score and store it together with the respective parents and index in the node dictionary.

```
    # end case
    prev_nodes_dict = nodes[len(s)] # last layer
    scores = []
    # take the U node in order to do transition parameters
    # transition parameters
    for U in prev_nodes_dict.keys():
        prev_states = transition_par['STOP'] # get the probabilities of tags dictionary
        if U in prev_states.keys():
            a = prev_states[U]
        else:
            a = 0
        index = 0
        # prev node score
        for prev_k_nodes in prev_nodes_dict[U]:
            score = prev_k_nodes[0] *a
            scores.append([score, U, index])
            index += 1
    scores.sort(key=lambda x: x[0], reverse=True)
    topk_scores = scores[:topk]
    indiv_node = {'STOP': topk_scores}
    nodes[len(s)+1] = indiv_node

    return nodes

def backtracking(s, nodes,kth):
    prev_state = 'STOP'
    prev_index = 0
    for i in range(len(s)+1, 1, -1): # Start at the end, and iterate backwards one node at a time
        if i == len(s) +1:
            prev_node = nodes[i][prev_state][kth-1]
        else:
            prev_node = nodes[i][prev_state][prev_index]
        prev_state = prev_node[1]
        prev_index = prev_node[2]
        s[i-2] += " " + prev_state # save the prev node
    return s
```

As for the end case, we calculate the top k score using the backtracking function. We find the top ith path and select the best ith parent from the top state and recursively backtrack to find the entire path using the stored parents and indexes.

Results

ES

```
#Entity in gold data: 255
#Entity in prediction: 2568

#Correct Entity : 171
Entity  precision: 0.0666
Entity  recall: 0.6706
Entity  F: 0.1211

#Correct Sentiment : 127
Sentiment  precision: 0.0495
Sentiment  recall: 0.4980
Sentiment  F: 0.0900
```

RU

```
#Entity in gold data: 461
#Entity in prediction: 3679

#Correct Entity : 289
Entity  precision: 0.0786
Entity  recall: 0.6269
Entity  F: 0.1396

#Correct Sentiment : 191
Sentiment  precision: 0.0519
Sentiment  recall: 0.4143
Sentiment  F: 0.0923
```

PART 5

Summary of Multiclass Perceptron Algorithm

For part 5 of the project, we utilized multi-class perceptron algorithm to *train* and predict the labels. The general idea of this is that we iterate through every label for example, if we are iterating through the label "B-positive", the classifications will be "B-positive" and "*not* B-positive". For this to happen, we need the algorithm to learn **7 different theta vectors** as the weights for the prediction, instead of just 1 theta vector in the normal case of a dual classification perceptron algorithm.
Then in the *test*ing phase, for every word in the *test* file, the algorithm will return the label that has the highest value when X is multiplied (dot product) with the corresponding label theta vector.

Formatting the Data

The format is very different from the other parts of the project. Instead of having a nested dictionary of words and a dictionary of their label counts, the *train*_data will be in the form of a nested list with a dictionary of words and their counts for each label. Here is an example of what *train*_data list would look like.

*Train*_data = [['B-positive', {'digital':1, 'Sens':1, 'Mathews': 1}], ['B-negative', {'digital':1, 'Sens':1, 'Mathews': 1}]]

A function *modifytestfile* takes in the *train* and *test* files as inputs. It firsts opens the file and split the words

```
def modifytestfile(trainfile, testfile):
    raw_train = open(trainfile, 'r+',
                     encoding="utf8")  # r+ is Special read and write mode, which is used to handle both actions when working with a file
    words_in_train = [] #this is a list of all the words X_i from the training data
    train_data = [["B-positive",{}], ["B-negative",{}], ["B-neutral",{}], ["I-positive",{}], ["I-negative",{}], ["I-neutral",{}], ["O",{}]]
    for words2 in raw_train:
        words3 = words2.split('\n')  ##words3 is a list of pairs of words and their labels

        for i in range(len(words3)):
            spt = words3[i].split(" ")

            if len(spt) > 3 or len(spt) < 2:
                continue


            for i in range(len(train_data)):
                if train_data[i][0] == spt [1]: #if the label in the training_data list is the same as the label of the training data
                    if spt[0] in train_data[i][1]:
                        train_data[i][1][spt[0]]+=1
                    else:
                        train_data[i][1][spt[0]] = 1
                else:
                    train_data[i][1][spt[0]]=0
            key1 = spt[0] #'Trump', 'tour', 'bus'
            words_in_train.append(key1)
```

Words3 is a list of pairs of the words and its label, and they are being split by their spaces. The new list is named spt.

words3 appears like this
['for O', '']
['8 O', '']
['years O', '']

print(spt)
['Trump', 'B-neutral']
['tour', 'O']

Then, iterating through all the *training* data, if the label in the *train*_data list is the same as the label of the *training* data, then the word count in the dictionary gets updated for that particular label. The *words_in_train* list also gets appended with all the words in *train*. This is to help find and index the lists later on when doing *test*ing.

Following which, we open the *test* file and modify it in such a way where the words are arranged in the same way as with the *train* file. In other words, words that are in the *test* file that exists in the *train* file will have a counter in a list of the same index as the word in the *train* file. If the word in the *test* file exists, then the *modified_test* list will append the counts of it occurring in the *test* file, in the same index as the word that appear in the *train* file.

```
    raw_test = open(testfile, 'r+', encoding="utf8")
    words_in_test = []
    read_lines =raw_test.read()
    lines = read_lines.split('\n')
    for words in lines:
        try:
            words_in_test.append(words)
        except:
            words_in_test.append('')

    modified_test = []
    # modified_test is a list that counts the number of times the words appear in test file, in the same order as the words in trainfile
    for word in words_in_train:
        count = 0
        if word in words_in_test:
            count += words_in_test.count(word)
        modified_test.append(count)

    return words_in_train, train_data, modified_test, words_in_test
```

At the end of this function, the outputs will be *words_in_train*, a list of words that occur in the *training* set, *train_data*, a nested list containing the label and dictionary of words as pairs in a list. We also have *modified_test*, a list of just numbers corresponding to the counts of the words in the *train* file, arranged in the same order as the words in the *train* file. Finally *words_in_test* is a list that stores the words that occur in the *train* file.

Training Phase

Because many things can go wrong with a perceptron algorithm, multiple iterations is required to for the machine to fully learn the data and come up with theta vectors that can accurately predict the labels. In this project, the iteration number is added as a parameter in the *training* function.
The *train* function takes *words_in_train*, *train_data*, (both of which comes from the *modifytestfile* function), number of *iterations* and theta0 as the inputs.

```
def training(words_in_train,train_data,iterations,theta0): #takes training data as input, returns theta vector
    label_list = ["B-positive", "B-negative","B-neutral","I-positive","I-negative","I-neutral","O"]

    '''
    # initializes a vector of zeroes for the different labels, +1 to accomodate theta0
    {'B-positive': array([0, 0, 0, ..., 0, 0, 0]), 'B-negative': array([0, 0, 0, ..., 0, 0, 0]), 'B-neutral': array([0, 0, 0, ..., 0, 0, 0]),
    '''

    for i in range(iterations):
        for label,word_dict in train_data:
            word_count_list = []  # word count of all the words in the trainfile for the particular label
            '''
            # print(word_dict)
            #label would show B-positive, B-negative
            #word_dict shows a dictionary of all the 69k words {"Donny's": 1, 'Sticks': 1, 'Tesco': 2, 'parasites': 1}
            '''
            for i in range(len(words_in_train)): # iterate through 69k times
                word = words_in_train[i]
                if word in word_dict: # if for that particular label, the word is in the word dictionary
                    word_count_list.append(word_dict[word])
                    # appends the count for that partcular word in the corresponding label
            word_count_list.append(theta0)
            # word_count_list is a list of the counts of words [1, 1, 1, 1, 6, 15, 6, 2, 6, 1, 6, 6, 1, 1, 3, 1
            train_array = np.array(word_count_list)
            argmax = 0 #initialize the argmax
            predicted_label = label_list[0]
             #initialise the first label to be "B-positive"

            theta_vector = {c: np.array([0 for i in range(len(word_count_list))]) for c in label_list}
```

The *label_list* is defined in this function as a list of all the different labels

- **["B-positive", "B-negative","B-neutral","I-positive","I-negative","I-neutral","O"]**

Next, the function will then iterate through every label and corresponding word dictionary (words as keys and counts as value), a *word_count_list* is created. This *word_count_list* is a list of counts, to count the number of times the word appears in the *train* file. *Word_dict*[word] is the counts for the number of times the specific word appears in the *train* file, and the count number will be appended into the *word_count_list*.

To initialise the learning, we initialise the *argmax* to be 0, and the predicted label to be *label_list*[0], in this case initialised to be "B-positive". *theta_vector* is an array of zeros for the 7 theta vectors for the 7 labels.

```
        for i in range(len(label_list)):

            key1 = label_list[i] # key1 = "B-positive"
            thetav = theta_vector[key1] # the specific theta for the label
            learning = np.dot(word_count_list, thetav) # this is the learning phase, theta dot x = y

            if learning >= argmax:
                argmax = learning
                predicted_label = label_list[i]

                if label != predicted_label: # if the label is wrongly predicted, update the theta
                    theta_vector[label] += word_count_list # add vector to the correct weight
                    theta_vector[predicted_label] -= word_count_list # minus the vector from the wrong weight
                    # this allows for a more accurate learning of theta
    return theta_vector
```

For every label in the *label_list*, we assign *key1* to be the label, then use *key1* to access the theta vector for the specific label. The list for the specific label is *thetav*, which is *theta_vector*[key1].

Learning is the value obtained from multiplying (dot product) the *word_count_list* with the *thetav* list. If the value is greater than the threshold value stipulated by *argmax*, then *argmax* will be updated to be the learning value, and the *predicted_label* will be updated to become the label.

After iterating through the 7 labels, if the label is wrongly predicted, the theta gets updated. The *theta_vector* of the correct label will have the *word_count_list* added to it, and the *theta_vector* of the wrong label will have the *word_count_list* subtracted from it. This is to properly calibrate the theta vectors so that it will get closer to the correct theta after every iteration for a more accurate learning of theta.

At the end of this function, it will output the *theta_vector* array which has 7 different theta for the 7 labels. This will be the parameters we will use to predict the labels for the *test* file.

Testing Phase

The next part of this is the *predict* function. The predict function takes *count_words_in_test*, which is the same as the *modified_list* output in the *modifytestfile* function, is a list of counts of words in the *test* file as one of its inputs. The *theta_vector* list from the output of the *training* function is also another input of the *predict* function. Other inputs include *words_in_test* list, and *words_in_train* lists of words from the *test* file and *train* file obtained as outputs from the *modifytestfile* function. *Theta0* is an input decided by the user. The *theta0* chosen should be the same as the theta0 input in the *training* function.

```
def predict(count_words_in_test, theta_vector, words_in_test, words_in_train, theta0):
    # initialisation
    count_words_in_test.append(theta0)
    output_file = open("dev.p5.out", "w+", encoding="utf8")
    testwords = []
    tag = []
    counter = 0

    for word in words_in_test: # iterate through every word in the testfile
        counter += 1
        # output_file.write(word + " ")
        word_vector = [0] * len(count_words_in_test)
        argmax = 0
        predicted_label = "B-positive"

        if word in words_in_train:
            testwords.append(word)
            index = words_in_train.index(word)
            word_vector[index] = 1

            for k in theta_vector.keys():
                # iterate with the different label thetas as well. k is the labels "B-positive", "B-negative" etc
                threshold = np.dot(word_vector,theta_vector[k])
                # print(threshold)
                if threshold >= argmax:
                    argmax = threshold
                    predicted_label = k
            tag.append(predicted_label)

        elif word == "":
            testwords.append("")
            tag.append("")
        else:
            testwords.append(word)
            tag.append("")

    for i in range(0,counter):
        output_file.write("%s %s\n" % (testwords[i], tag[i]))

    output_file.close()
    return output_file
```

First we add theta0 into the count_words_in_test list so that it would be the same length as the theta vector. Argmax is initialised as 0.

We iterate through all the words in the *test* file, and write the words into an output file. For every iteration, we will create a word vector of all zeros except for 1. The number 1 will be located at the same index of the word_vector as the index of the same word that occured in the *train* file. For example, if the word “pink” is in the n^{th} index of the *words_in_train* file, then the word vector for this iteration will be a vector of all zeros except for the value at the n^{th} index taking the value 1.

After generating the specific word vector, it will iterate through the 7 weight vectors for the 7 labels, and the threshold value is the dot product of the word vector and the specific weight vector for the label. If the threshold value is greater than the argmax, the argmax will be updated with the threshold value, and the predicted_label becomes k, the label of that iteration. At the end of the 7 iterations, the predicted_label will be the label with the highest threshold value, and it will be returned as a label prediction for that particular word.

After iterating through all the words and labels, the function then returns the output file of the words and their predicted labels as *dev.p5.out*.

Results

There was an insufficient time to finish running the code.

Analysis of the Multiclass Perceptron Algorithm

The multiclass perceptron algorithm is inefficient and less accurate compared to the HMM algorithm. It is less accurate because with the size of the dataset, the vector is very large, and for the machine to accurately learn the weights, it has to go through many iterations, which will take up a lot of time considering the size of the word vector. HMM just takes into account the emission and transition probabilities through the counts of the occurrence of the words and states. Running the perceptron took an approximately 6 minutes, while it only takes HMM 1 minute for it to come up with the *dev.out* for RU dataset.

This is also because, in perceptron, the *test* file has to be formatted into the same way as the *train* file, and each time it goes through an iteration, it needs to do a dot product which is much slower than just taking the max of the probabilities in HMM.