Machine_Learning_Report

December 6, 2017

Name: Amish Bhandari Student ID: 1001614 Name: Yim Tat Yuen Bernard Student ID: 1001542 Name: Suen Hung Ying Sidney Student ID: 1001525

0.1 Machine Learning Design Project

0.1.1 Part 2

Disclaimer: please refer to the README.txt for instructions on how to run the code. The purpose of this report is to explain our implementation.

Before answering the questions, it is important to discuss how the parameters are calculated and stored as data. For example, the image below is a representation of the nested dictionary used to store the emission parameters: For example, we have a dictionary structured like the diagram above, and it is named *emissionParameters*. To access the number of times an observation is emitted from a state, we can access it via in this manner (note that this is pseudocode):

$$Count(tag) = tagCount[tag]$$

$$Count(tag \rightarrow observation) = emissionParameters[observation][count][tag]$$

$$emissionParameters[x][''parameters''][y] = \frac{emissionParameters[x][''count''][y]}{tagCount[y]}$$

This dictionary is constructed by the usage of the helper functions *nestedDictProcess*

(5 pts) Write a function that estimates the emission parameters from the training set using MLE (maximum likelihood estimation):

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

```
In [ ]: # Returns a dictionary with the emission parameters, structured in the manner as shown
        # in the diagram above
        def computeEmissions(fileDir, tagCount):
            with open('{0}\modifiedTrain.txt'.format(fileDir), 'r',encoding='utf-8') as modTra
                trainSetString = modTrainSet.read()
            emissionParameters = {}
            # Compute the emission counts
            trainSetLines = trainSetString.splitlines(True)
            for i in trainSetLines:
                data = i.rsplit(" ",1)
                if(len(data)==2):
```

```
                word = data[0]
                tag = data[1].rstrip('\n')
                if(word == '' or tag not in sentimentSets):
                    print("Corrupted data detected: {0}".format(i))
                else:
                    nestedDictProcess(emissionParameters,word,tag)# Builds up the dictiona
            elif(i == '\n'):
                pass
            else:
                print("Corrupted data detected: {0}".format(i))

        # Compute the observation parameters
        emitParams = buildEmissionParameters(emissionParameters,tagCount)#Builds up the pa
        return emitParams

    def buildEmissionParameters(dictionary, tagCount):
        for key, value in dictionary.items():
            parameters = {}
            for subKey,subvalue in value["count"].items():
                parameters[subKey] = subvalue/tagCount[subKey]
            dictionary[key]["parameters"] = parameters
        return dictionary
```

(10 pts) One problem with estimating the emission parameters is that some words that appear in the test set do not appear in the training set. One simple idea to handle this issue is as follows. First, replace those words that appear less than k times in the training set with a special token #UNK# before training. This leads to a "modified training set". We then use such a modified training set to train our model. During the testing phase, if the word does not appear in the "modified training set", we replace that word with #UNK# as well. Set k to 3, implement this fix into your function for computing the emission parameters.

```
In [2]: sentimentSets = ["START","STOP","O","B-positive","I-positive","B-neutral","I-neutral",'

    #This function is run first to get all the training parameters
    def preprocess(fileDir,kVal=3):
        #Read the designated files first
        outliers = {} # A dictionary created to store any outliers we might encounter
        tagCount ={}# tagCount = {tag: Count(tag)}
        trainWords={}# trainWords = {word: Count(word)}
        modtrainWords = {}# same as trainWords, but after converting rare words into #UNK#

        # Read the train file as a string. The train file's extension had to be changed to
        # .txt format as python could not read it. Utf-8 encoding was specified to deal wi
        # characters
        with open('{0}\\train.txt'.format(fileDir), 'r',encoding='utf-8') as trainSet:
            trainSetString = trainSet.read()

        print("Processing tagcounts and train word counts")
```

```
trainSetLines = trainSetString.splitlines(True)# Split the training set line by li

for i in trainSetLines:
    data = i.rsplit(" ",1)# each line is in the format of "word tag", rsplit is to
    if(len(data)==2):
        word = data[0]
        tag = data[1].rstrip('\n')# remove the newline character
        # Checking for anomalous data
        if(word == '' or tag not in sentimentSets):
            print("Corrupted data detected: {0}").format(i)
        else:
            dictProcess(tagCount, tag)#A helper function to tally up the counts
            dictProcess(trainWords,word)
    elif(i == '\n'):
        dictProcess(tagCount,"START")
        dictProcess(tagCount,"STOP")
    else:
        print("Corrupted data detected: {0}".format(i))

print("Replacing words in the training set that appear less than k times with #UNK#
# Replace the words in the training set that appear less than k times with #UNK#
wordDict = {k:v for (k,v) in trainWords.items() if v < kVal}
modifiedString = ""

# parsing through the training set, changing the words into unknowns and
# appending each line into a string. The final string will be written to
# a file titled "modifiedTrain.txt"
for i in trainSetLines:
    data = i.rsplit(" ",1)
    if(len(data)==2):
        word = data[0]
        tag = data[1].rstrip('\n')
        if(word in wordDict):
            words = "#UNK# "+tag+"\n"
            modifiedString = modifiedString+ words
        elif(word not in wordDict):
            # print("Word not in the dictionary just add as usual: {0}".format(i))
            modifiedString = modifiedString+i
        else:
            print("I have no idea what this is: {0}").format(i)
            modifiedString = modifiedString+i
    elif(i == '\n'):
        # print("Just a new line")
        modifiedString = modifiedString+i
    else:
        print("Corrupted data detected: {0}".format(i))
        modifiedString = modifiedString+i
```

```python
    # Building modtrainWords from the modified training set:
    for i in modifiedString.splitlines(True):
        data = i.rsplit(" ",1)
        if(len(data)==2):
            word = data[0]
            tag = data[1].rstrip('\n')
            if(word == '' or tag not in sentimentSets):
                print("Corrupted data detected: {0}").format(i)
            else:
                dictProcess(modtrainWords,word)
        elif(i == '\n'):
            # print("Just a new line")
            pass
        else:
            print("Corrupted data detected: {0}".format(i))

    # Reading the words inside the testSet that do not appear in the training set
    testWords = {}# a dictionary to record all the words in the training set
    with open('{0}\dev.in'.format(fileDir), 'r',encoding='utf-8') as testSet:
        testSetString = testSet.read()
    testSetLines = testSetString.splitlines()# This converts all the '\n' to ''
    for i in testSetLines:
        if(i!=''):
            dictProcess(testWords,i)

    wordsNotInTrainingSet = set(testWords) - set(modtrainWords)

    # Iterate over the lines of the test file. If any words occur inside the
    # set of words not inside the training set, replace with "#UNK#"
    modifiedTestString = ""
    for i in testSetLines:
        if (i != ''):
            if i in wordsNotInTrainingSet:
                modifiedTestString = modifiedTestString+"#UNK#\n"
            else:
                modifiedTestString = modifiedTestString+ i+'\n'
        else:
            modifiedTestString = modifiedTestString+ '\n'
    #Write the modified strings to a file
    with open('{0}\modifiedTrain.txt'.format(fileDir), 'w',encoding='utf-8') as outputF
        outputFile.write(modifiedString)
    with open('{0}\modifiedTest.txt'.format(fileDir), 'w',encoding='utf-8') as outputTe
        outputTestFile.write(modifiedTestString)
    return tagCount

# Addition operation for a normal dictionary
def dictProcess(dictionary, key):
    dictionary[key] = dictionary.get(key,0)+1
```

```
# Helper function to tally up the counts for the nested dictionary structure
def nestedDictProcess(dictionary,key,subKey):
    if key not in dictionary:
        dictionary[key]={}
        dictionary[key]["count"] = {subKey:1}
    else:
        dictionary[key]["count"][subKey] = dictionary[key]["count"].get(subKey,0)+1 #I

# A helper function designed to detect anomalies
def detectAnomalies(fileDir):
    with open('{0}\modifiedTrain.txt'.format(fileDir), 'r',encoding='utf-8') as modTra
        trainSetString = modTrainSet.read()
    trainSetLines = trainSetString.splitlines(True)
    sentences = 0
    index = 0
    indices = ""
    for i in trainSetLines:
        index+= 1
        data = i.split(" ")# Such was a wrong way of splitting
        if(len(data)==2):
            word = data[0]
            tag = data[1].rstrip('\n')
            if(word == '' or tag not in sentimentSets):
                print("Corrupted data detected: {0}".format(i))
                indices= indices +"{0} {1}\n".format(i,index)
        elif(i == '\n'):
            sentences +=1
        else:
            print("Corrupted data detected: {0}".format(i))
            indices= indices +"{0} {1}\n".format(i,index)
```

```
C:\Users\Sidney
```

(10 pts) Implement a simple sentiment analysis system that produces the tag

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

for each word x in the sequence. For all the four datasets EN, FR, CN, and SG, learn these parameters with train, and evaluate your system on the development set dev.in for each of the dataset. Write your output to dev.p2.out for the four datasets respectively. Compare your outputs and the gold-standard outputs in dev.out and report the precision, recall and F scores of such a baseline system for each dataset.

To find y_*, $emissionParameters[observation]["parameters"]$ can be used to access the dictionary whose values contain the emission values for each tag for a given observation. The tag with the maximum value would be extracted quite easily.

```
In [ ]: #Part 2.3
        def simpleSentimentAnalysis(fileDir,emissionParameters):
            with open('{0}\modifiedTest.txt'.format(fileDir), 'r',encoding='utf-8') as modTestS
                testSetString = modTestSet.read()

            testSetLines = testSetString.splitlines()
            devp2out = ""
            for i in testSetLines:
                if (i != ''):
                    paramList = emissionParameters[i]["parameters"]
                    tag = max(paramList, key=paramList.get)
                    # This was why the emission parameters were structured this way
                    # Just perform one extract-max function
                    devp2out = devp2out +i +" "+tag +"\n"
                else:
                    devp2out = devp2out+ '\n'
            with open('{0}\dev.p2.out'.format(fileDir), 'w',encoding='utf-8') as outputFile:
                outputFile.write(devp2out)
```

Set name	**CN Set:**	**FR set:**	**SG set:**	**EN set:**
#Entity in gold data:	362	223	1382	226
#Entity in prediction:	3318	1149	6599	1201
#Correct Entity :	183	182	794	165
Entity precision:	0.0552	0.1584	0.1203	0.1374
Entity recall:	0.5055	0.8161	0.5745	0.7301
Entity F:	0.0995	0.2653	0.1990	0.2313
#Correct Sentiment :	57	68	315	71
Sentiment precision:	0.0172	0.0592	0.0477	0.0591
Sentiment recall:	0.1575	0.3049	0.2279	0.3142
Sentiment F:	0.0310	0.0991	0.0789	0.0995

dev.p2.out scores:

0.1.2 Part 3:

(5 pts) Write a function that estimates the transition parameters from the training set using MLE (maximum likelihood estimation):

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

Please make sure the following special cases are also considered: $q(STOP|y_n)$ and $q(y_1|START)$.

Above is the structure of the dictionary used to contain the the transition parameters. For example, if the dictionary's name is *transitionParameters*, the transition parameters can be accessed in this manner:

$$a_{u,v} = transitionParameters[u]["parameters"][v]$$

$$Count(u) = tagCount[u]$$

$$Count(u,v) = transitionParameters[u]["count"][v]$$

$$transitionParameters[u]["parameters"][v] = \frac{transitionParameters[u]["count"][v]}{tagCount[v]}$$

```
In [ ]: from collections import deque
        import sys
        import math
        from math import inf

        sentimentSets = ["START","STOP","O","B-positive","I-positive","B-neutral","I-neutral","

        def computeSentences(fileDir):
            with open('{0}\modifiedTest.txt'.format(fileDir), 'r',encoding='utf-8') as modTestS
                testSetString = modTestSet.read()
            sentences= []# an array of sentence
            sentence = []# an array of words
            testSetLines = testSetString.splitlines()
            for i in testSetLines:
                if (i != ''):
                    sentence.append(i)
                else:
                    #End of sentence reached
                    sentences.append(sentence)
                    sentence = []
            #The function save_obj saves the variable as a .pkl file that can be used later
            save_obj(sentences,fileDir,"sentences")


        def computeTransitions(fileDir,tagCount):
            with open('{0}\modifiedTrain.txt'.format(fileDir), 'r',encoding='utf-8') as modTrai
                trainSetString = modTrainSet.read()
            transitionParameters = {}

            #Transition parameters are calculated using y_prev and y_next
            y_prev = "START"
            y_next = ""

            trainSetLines = trainSetString.splitlines(True)
            #Parse through the lines in the training set
            for i in trainSetLines:
                data = i.rsplit(" ",1)
                if(len(data)==2):
                    word = data[0]
                    tag = data[1].rstrip('\n')
                    #Build up the counts of the transitions
                    if(word == '' or tag not in sentimentSets):
                        print("Corrupted data detected: {0}".format(i))
                    else:
                        y_next = tag
                        nestedDictProcess(transitionParameters,y_prev,y_next)
                        y_prev = tag

                elif(i == '\n'):
                    y_next = "STOP"
                    nestedDictProcess(transitionParameters,y_prev,y_next)
                    y_prev = "START"
                else:
                    print("Corrupted data detected: {0}".format(i))
            transParams = buildTransitionParameters(transitionParameters,tagCount)#Builds up t
            return transParams

        # You now have the dictionary containing the counts of the tag u, now just divide
        # number of transitions from u to v by Count(u) to obtain a_u,v
        def buildTransitionParameters(dictionary, tagCount):
            for y_prev, value in dictionary.items():
                parameters = {}
                for y_next,count in value["count"].items():
                    parameters[y_next] = count/tagCount[y_prev]
                dictionary[y_prev]["parameters"] = parameters
            return dictionary

        import pickle
        def save_obj(obj, fileDir, fileName ):
            with open('{0}\\variables\{1}.pkl'.format(fileDir,fileName),'wb') as f:
                pickle.dump(obj, f, pickle.HIGHEST_PROTOCOL)

        def load_obj(fileDir, fileName):
            with open('{0}\\variables\{1}.pkl'.format(fileDir,fileName), 'rb') as f:
                return pickle.load(f)
```

(15 pts) Use the estimated transition and emission parameters, implement the Viterbi algorithm to compute the following (for a sentence with n words):

$$y_i^*, ..., y_n^* = \arg\max_{y_1,..,y_n} p(x_1, .., x_n | y_1, .., y_n)$$

For all datasets, learn the model parameters with train. Run the Viterbi algorithm on the development set dev.in using the learned models, write your output to dev.p3.out for the four datasets respectively. Report the precision, recall and F scores of all systems. *Note: in case you encounter potential numerical underflow issue, think of a way to address such an issue in your implementation.* Any number smaller than $2.2250738585072014 * 10^{-308}$ will become 0, hence the values of the nodes will instead be the logarithm of the actual values. As logarithm of 0 is undefined, the value negative infinity (-inf) will instead be assigned.

The value of the nodes will be stored as an array of dictionaries, which is named markovTable. The array index refers to the position in the sequence. The value of $\pi(i, v)$ is represented by markovTable[i][v].

```
In [ ]: #Helper functions to obtain transmission and emission parameters
        def aUV(transitionParameters,prev_tag,tag):
            dic = transitionParameters[prev_tag]["parameters"]
            return dic.get(tag,0)# If the transition is not observed in training set, return 0

        def bVxi(emissionParameters,observation,tag):
            dic= emissionParameters[observation]["parameters"]
            return dic.get(tag,0)# If the emission is not observed in training set, return 0.

        # This function calls all Vitterbi for every sentence in the testing set.
        def decodeAllSentences(sentences, fileDir, tP, eP):
            fileString = ""
            for sentence in sentences:
                fileString = fileString+viterbiAlgorithm(sentence,tP,eP)+"\n"
            with open('{0}\\dev.p3.out'.format(fileDir), 'w',encoding='utf-8') as outputFile:
                outputFile.write(fileString)

        def viterbiAlgorithm(sentence_array, transitionParameters, emissionParameters):
            tP = transitionParameters
            eP = emissionParameters
            terminalValue = 0
            tagSets = {"O":0,"B-positive":0,"I-positive":0,"B-neutral":0,"I-neutral":0,"B-nega
            markovTable = []
            prev_tag = "START"

            trans =0
            emit =0
            # Computing the values of the nodes
            for i in range(0,len(sentence_array)):
                # The table is continuously built, with the node values intialised to 0.
                tagSets = {"O":0,"B-positive":0,"I-positive":0,"B-neutral":0,"I-neutral":0,"B-
                markovTable.append(tagSets)
                observation = sentence_array[i]
                if i == 0:
                    print("Base case")
                    for tag in markovTable[i]:
                        trans = aUV(tP,prev_tag,tag)
                        emit = bVxi(eP,observation,tag)
                        if(trans == 0 or emit ==0):
                            markovTable[i][tag] = -inf
                        else:
                            markovTable[i][tag] = math.log10(emit*trans)
                else:
                    #Recursive case
                    for tag in markovTable[i]:
                        # To get the tag which produces the larget value, we first
                        # create an array to store the values. We calculate the values
                        # for each different preceding tag and store the values in the array
                        # then we extract the maximum value and assign it to the current node
                        values = []
                        for prev_tag in markovTable[i-1]:
                            prev_node_val = markovTable[i-1][prev_tag]
                            trans = aUV(tP,prev_tag,tag)
                            emit = bVxi(eP,observation,tag)

                            #Check the values
                            if(prev_node_val is -inf or trans == 0 or emit ==0):
                                tempVal = -inf
                            else:
                                #Need to reassign a new value based on the log space laws
                                #The value in the nodes are already in log base 10
                                tempVal = prev_node_val+math.log10(trans*emit)
                            values.append(tempVal)
                        markovTable[i][tag] = max(values)

            # Terminal case (when the end of the sentence is reached):
            lastTag = "STOP"
            values = []
            observation = sentence_array[-1]
            for prev_tag in markovTable[-1]:
                prev_node_val = markovTable[-1][prev_tag]
                trans = aUV(tP,prev_tag,lastTag)
                if(prev_node_val is -inf or trans == 0):
                    tempVal = -inf
                else:
                    tempVal = prev_node_val+math.log10(trans)
                values.append(tempVal)
            terminalValue = max(values) #This is pi(n+1,STOP)

            #Backtracking
            print("Commencing back trekking with terminal value: {}".format(terminalValue))
            from collections import deque
            sequenceList = deque()
            latestTag = "STOP"
            for i in range(len(markovTable)-1, -1, -1):
                observation = sentence_array[i]

                #Iterate over all the tags in the previous table, multiplying with the
                #corresponding transition parameter and selecting the max as y^*_i
                for prev_tag,pi in markovTable[i].items():
                    transition = aUV(tP,prev_tag,latestTag)
                    if (transition ==0 or pi is -inf):
                        markovTable[i][prev_tag] = -inf
                    else:
                        markovTable[i][prev_tag] = pi+math.log10(transition)
                parent = max(markovTable[i], key=markovTable[i].get)
                sequenceList.appendleft(parent)
                latestTag = parent

            #Return the observation-state string
            obs_statePair = ""
            for i in range(0, len(sentence_array)):
                word = sentence_array[i]
                tag = sequenceList[i]
                obs_statePair = obs_statePair+"{0} {1}\n".format(word, tag)
            return obs_statePair
```

Set name	**CN Set:**	**FR Set:**	**SG set:**	**EN set:**
#Entity in gold data:	362	223	1382	226
#Entity in prediction:	159	166	724	162
#Correct Entity :	64	112	386	104
Entity precision:	0.4025	0.6747	0.5331	0.6420
Entity recall:	0.1768	0.5022	0.2793	0.4602
Entity F:	0.2457	0.5758	0.3666	0.5361
#Correct Sentiment :	47	72	244	64
Sentiment precision:	0.2956	0.4337	0.3370	0.3951
Sentiment recall:	0.1298	0.3229	0.1766	0.2832
Sentiment F:	0.1804	0.3702	0.2317	0.3299

dev.3.out scores:

0.2 Part 4 (20 points)

(20 pts) Use the estimated transition and emission parameters, implement the alternative max-marginal decoding algorithm. Clearly describe the steps of your algorithm in your report. Run the algorithm on the development sets EN/dev.in and FR/dev.in only. Write the outputs to EN/dev.p4.out and FR/dev.p4.out. Report the precision, recall and F scores for the outputs for both languages. *Hint: the max-marginal decoding involves the implementation of the forward-backward algorithm.*

Forward algorithm:

$$Base\ case : \alpha_u(1) = a_{START,u}$$

$$Recursive\ case : \alpha_v(j+1) = \sum_u \alpha_u(j) * a_{u,v} * b_u(x_j)$$

#### Backward algorithm:

$$Base\ case : \beta_u(n) = a_{u,STOP} * b_u(x_n)$$

$$Recursive\ case: \beta_u(j) = \sum_v \beta_v(j+1) * a_{u,v} * b_u(x_j)$$

#### Max-marginal decoding: By computing the α and β parameters, we can thus find the most likely tag at a given position in a setence.This can be achieved by finding

$$y_i^* = \arg\max_{y_i} P(y_i = u|x)$$

$$= \arg\max_u \frac{P(y_i = u, x)}{P(x)}$$

$$= \arg\max_u P(x_1, .., x_{j-1}, y_i = u) * P(x_j, .., x_n|y_j = u)$$

$$= \arg\max_u \alpha_u(j) * \beta_u(j)$$

For every observation in each sentence of the test set. Implementation of the above steps will be discussed in the code below. *Note: while running the code on the EN and FR test sets, there were no cases of underflow detected, hence there is no scaling used in this implementation of the forward backward algorithm*

```
In [1]: # This function takes in an array of sentences (arrays of words) and calls the
        # function maxMarginal on each sentence. maxMarginal takes in the sentence, as
        # well as the transition and emission parameters learnt from the training set
        # and outputs the sequence of words and tags in the same format as train.txt
        def main(sentences, fileDir, tP, eP):
            fileString = ""
            for sentence in sentences:
                fileString = fileString+maxMarginal(sentence,tP,eP)+"\n"
            with open('{0}\\dev.p4.out'.format(fileDir), 'w',encoding='utf-8') as outputFile:
                outputFile.write(fileString)

        # The alpha and beta parameters are stored as a array of dictionaries, with
        # each dictionary containing the tags as keys and the corresponding alphas/betas
        # as the values. This is a similar strategy to the one used in the Vitterbi algorithm
        # alpha_u(1) = alpha[1][u],
        # likewise for beta:
        # beta_u(1) =beta[1][u]
        def maxMarginal(sentence,tP,eP):
            n = len(sentence)
            alpha = deque()
            firstTag = "START"
            #The forward algorithm
            for index in range(0,n):
                observation = sentence[index]
                tagSets = {"O":0,"B-positive":0,"I-positive":0,"B-neutral":0,"I-neutral":0,"B-
                alpha.append(tagSets)# Continuously build the table of alpha values

                if (index == 0):
                    # Base case
                    for current_tag in alpha[index]:
                        # print(current_tag)
                        transition = aUV(tP,firstTag,current_tag)
                        alpha[0][current_tag] = transition
                else:
                    # Iterate over every tag in the current set
                    previous_observation = sentence[index-1]
                    for current_tag in alpha[index]:
                        runningTotal = 0
                        # Summation over every alpha from the previous set
                        for previous_tag in alpha[index-1]:
                            alpha_u_n = alpha[index-1][previous_tag]
                            trans = aUV(tP,previous_tag,current_tag)
                            emit = bVxi(eP,previous_observation, previous_tag)
                            runningTotal = runningTotal + alpha_u_n*trans*emit
                            # Because scaling is not used, this if statement is there to check
                            # if any underflow might occur
                            if(runningTotal < 1e-300 and runningTotal!= 0):
                                print("Danger! {}".format(runningTotal))
                        alpha[index][current_tag] = runningTotal

            # The backward algorithm: because beta values are calulated from position n to 1
            # a deque is used to allow insertions from the left
            beta = deque()
            lastTag = "STOP"

            for i in range(0,n):
                tagSets = {"O":0,"B-positive":0,"I-positive":0,"B-neutral":0,"I-neutral":0,"B-
                beta.append(tagSets)

            for index in range(n-1,-1,-1):
                observation = sentence[index]

                if (index == n-1):
                    #base case
                    for current_tag in beta[index]:
                        transition = aUV(tP,current_tag,lastTag)
                        emission = bVxi(eP,observation,current_tag)
                        beta_u_n = transition*emission
                        beta[index][current_tag] = beta_u_n

                #Recursive case
                else:
                    for current_tag in beta[index]:
                        runningTotal = 0
                        for previous_tag in beta[index+1]:
                            beta_u_n = beta[index+1][previous_tag]
                            trans = aUV(tP,current_tag,previous_tag) #a_v,u
                            emit = bVxi(eP,observation, current_tag)
                            runningTotal = runningTotal+ beta_u_n*trans*emit

                            if(runningTotal < 1e-300 and runningTotal!= 0):
                                print("Danger! {}".format(runningTotal))

                        beta[index][current_tag] = runningTotal

            obs_statePair = ""
            #Max marginal decoding
            for index in range(0,n):
                word = sentence[index]
                dict_alpha = alpha[index]
                dict_beta =  beta[index]
                alpha_x_beta = {}
                for sentiment in tagSets:
                    alpha_x_beta[sentiment] = dict_alpha[sentiment]*dict_beta[sentiment]
                #Calculate argmax(u) alpha_u(j)* beta_u(j)
                tag = max(alpha_x_beta, key=alpha_x_beta.get)
                obs_statePair = obs_statePair +word +" "+tag +"\n"
            return obs_statePair

        def aUV(transitionParameters,prev_tag,tag):
            dic = transitionParameters[prev_tag]["parameters"]
            return dic.get(tag,0)

        def bVxi(emissionParameters,observation,tag):
            dic= emissionParameters[observation]["parameters"]
            return dic.get(tag,0)
```

Set name	**EN Set:**	**FR Set:**
#Entity in gold data:	226	223
#Entity in prediction:	175	173
#Correct Entity :	108	113
Entity precision:	0.6171	0.6532
Entity recall:	0.4779	0.5067
Entity F:	0.5387	0.5707
#Correct Sentiment :	69	73
Sentiment precision:	0.3943	0.4220
Sentiment recall:	0.3053	0.3274
Sentiment F:	0.3441	0.3687

dev.p4.out scores: Part 5: In order to improve the sentiment analysis results, we attempted to train and use perceptrons in conjunction with the Viterbi algorithm. The improved transition and emission parameters will then be used in the maxMarginal method, in order to generate a predictive output on the test data.

```
We adapted a perceptron training model from a reference slide from Cambridge University's depa

From the reference, the algorithm for training looks like this.
```

In words: Assume n tagged sentences for training Initialise weights to zero Do L passes over the training data For each tagged sentence in the training data, find the highest scoring tag sequence using the current weights If the highest scoring tag sequence matches the gold, move to next sentence If not, for each feature in the gold but not in the output, add 1 to its weight; for each feature in the output but not in the gold, take 1 from its weight Return weights

```
We adapted a few things from this reference slide, as we are making use of the existing code. 

Firstly, our gold standard was the tagged training data, and the respective training data is t

The perceptrons and their weights are also implemented in another way. We choose to modify the

At the end of the perceptron training, we have a final transmission and emission parameter, wh

Our modified algorithm looks like this:
    Generate tagCount (counts the number of tags for all possible states, eg I-negative, I-pos
    Assume n tagged sentences for training
    Do 10 passes over the training data
        Initialise feature counters for each feature to 0
        For each tagged sentence in the training data, use the existing viterbi algorithm in p
            If the generated sequence matches the gold, move to next sentence
            If not, for each feature in the gold but not in the output, add 1 to the feature c

            For all non-zero feature-counters, we deduct from tagCount their respective count.
            We then regenerate the transition and emission parameters from the new tagCounts, 

A problem with the implementation is that we modify the tagCount directly, instead of choosing

From the results (shown above), we can see a marked improvement in both precision and recall f

Future works can be to:
    I) change the implementation of the perceptron to affect the transition and emission param
    II) Consider and reduce the cases of perceptron overfitting, by averaging the weights out 



Reference:
https://www.cl.cam.ac.uk/teaching/1213/L101/clark_lectures/lect5.pdf
```

```
In [ ]: # -*- coding: utf-8 -*-


        from collections import deque
        import math
```

```
from math import inf
import pickle

#part 5 changes much of part2-4 code to prevent writing to files. Instead, strings are
#Because of this, you must run the entire code from the beginning to completion even i
#Note: This file takes damn long to run (about 30 minutes).


#CONSTANTS, change here for settings

TEST=True
#if TEST is true, read and write from test.in. Otherwise, it will be from dev.in
#writes to dev.p5.out (or test.p5.out)
#should not write to anything else

fileDir = "EN"
#Specifies language to use. choose between EN and FR

NUMOFITER = 1
#Number of times to run perceptron. 3 is fine, 10 takes a while
TRAIN = True
#To train the model on train.txt. If false, loads from pickle file

### PART 4 modified
def part4Vit(sentences, fileDir, tP, eP):
    #Runs maxMarginal on all sentences, and writes to test.p5.out (or dev.p5.out)
    fileString = ""
    for sentence in sentences:
        maxMarginal(sentence,tP,eP)
        fileString = fileString+maxMarginal(sentence,tP,eP)+"\n"

    if TEST:
        with open('{0}\\test.p5.out'.format(fileDir), 'w',encoding='utf-8') as outputFi
            outputFile.write(fileString)
    else:
        with open('{0}\\dev.p5.out'.format(fileDir), 'w',encoding='utf-8') as outputFil
            outputFile.write(fileString)

def maxMarginal(sentence,tP,eP):
    n = len(sentence)
    alpha = deque()
    firstTag = "START"

    for index in range(0,n):
        observation = sentence[index]
        tagSets = {"O":0,"B-positive":0,"I-positive":0,"B-neutral":0,"I-neutral":0,"B-r
        alpha.append(tagSets)
    if (index == 0):
        #Base case
        for current_tag in alpha[index]:
            #print(current_tag)
            transition = aUV(tP,firstTag,current_tag)
            alpha[0][current_tag] = transition
    else:
        #Iterate over every tag in the current set
        previous_observation = sentence[index-1]
        for current_tag in alpha[index]:
            runningTotal = 0
            #Summation over the previous set
            for previous_tag in alpha[index-1]:
                alpha_u_n = alpha[index-1][previous_tag]
                trans = aUV(tP,previous_tag,current_tag)
                emit = bVxi(eP,previous_observation, previous_tag)
                runningTotal = runningTotal + alpha_u_n*trans*emit
                if(runningTotal < 1e-300 and runningTotal!= 0):
                    print("Danger! {}".format(runningTotal))

            alpha[index][current_tag] = runningTotal



beta = deque()
lastTag = "STOP"

for i in range(0,n):
    tagSets = {"O":0,"B-positive":0,"I-positive":0,"B-neutral":0,"I-neutral":0,"B-
    beta.append(tagSets)

for index in range(n-1,-1,-1):
    observation = sentence[index]

    if (index == n-1):
        #base case
        for current_tag in beta[index]:
            transition = aUV(tP,current_tag,lastTag)
            emission = bVxi(eP,observation,current_tag)
            #print("word: {} tag:{} trans: {}, emiss: {}".format(observation, curr
            beta_u_n = transition*emission
            beta[index][current_tag] = beta_u_n
            #print(beta[0])
    else:
        for current_tag in beta[index]:
            runningTotal = 0
            for previous_tag in beta[index+1]:
                beta_u_n = beta[index+1][previous_tag]
```

```
                    trans = aUV(tP,current_tag,previous_tag)#Reversed flow
                    emit = bVxi(eP,observation, current_tag)
                    runningTotal = runningTotal+ beta_u_n*trans*emit

                    if(runningTotal < 1e-300 and runningTotal!= 0):
                        print("Danger! {}".format(runningTotal))

                beta[index][current_tag] = runningTotal

    obs_statePair = ""
    for index in range(0,n):
        word = sentence[index]
        dict_alpha = alpha[index]
        dict_beta =  beta[index]
        #alpha_x_beta = {k:dict_alpha[k]*dict_beta[k] for k in tagSets}
        alpha_x_beta = {}
        for sentiment in tagSets:
            alpha_x_beta[sentiment] = dict_alpha[sentiment]*dict_beta[sentiment]
        tag = max(alpha_x_beta, key=alpha_x_beta.get)
        #print("word: {}, tag: {}".format(word,tag))
        obs_statePair = obs_statePair +word +" "+tag +"\n"
    return obs_statePair

def aUV(transitionParameters,prev_tag,tag):
    dic = transitionParameters[prev_tag]["parameters"]
    return dic.get(tag,0)

def bVxi(emissionParameters,observation,tag):
    dic= emissionParameters[observation]["parameters"]
    return dic.get(tag,0)

### END PART 4



### PART 2 mod

sentimentSets = ["START","STOP","O","B-positive","I-positive","B-neutral","I-neutral",
def preprocess(fileDir,kVal):
    #returns tuple of (tagCount,cleanedTrainString,cleanedTestString)
    #Read the designated files first
    tagCount ={}
    trainWords={}
    modtrainWords = {}

    with open('{0}\\train.txt'.format(fileDir), 'r',encoding='utf-8') as trainSet:
        trainSetString = trainSet.read()
```

```
print("Processing tagcounts and train word counts")
#Parse through the training set
trainSetLines = trainSetString.splitlines(True)
#dictProcess(tagCount,"START")
for i in trainSetLines:
    data = i.rsplit(" ",1)
    if(len(data)==2):
        word = data[0]
        tag = data[1].rstrip('\n')
        if(word == '' or tag not in sentimentSets):
            print("Corrupted data detected: {0}").format(i)
        else:
            dictProcess(tagCount, tag)#A helper function to tally up the counts
            dictProcess(trainWords,word)
    elif(i == '\n'):
        dictProcess(tagCount,"START")
        #print("Just a new line")
        dictProcess(tagCount,"STOP")
    else:
        print("Corrupted data detected: {0}".format(i))


print("Replacing words in the training set that appear less than k times with #UNK
#Replace the words in the training set that appear less than k times with #UNK#
wordDict = {k:v for (k,v) in trainWords.items() if v < kVal}
modifiedString = ""
for i in trainSetLines:
    data = i.rsplit(" ",1)
    #print(data)
    #What about cases where there is a word without a sentiment?
    #TODO: account for cases where there is corrupted data.
    if(len(data)==2):
        word = data[0]
        tag = data[1].rstrip('\n')
        if(word in wordDict):
            words = "#UNK# "+tag+"\n"
            modifiedString = modifiedString+ words
        elif(word not in wordDict):
            #print("Word not in the dictionary just add as usual: {0}".format(i))
            modifiedString = modifiedString+i
        else:
            print("I have no idea what this is: {0}").format(i)
            modifiedString = modifiedString+i
    elif(i == '\n'):
        #print("Just a new line")
        modifiedString = modifiedString+i
    else:
        print("Corrupted data detected: {0}".format(i))
```

```
            modifiedString = modifiedString+i

    #Building modtrainWords:
    #TODO: find a way to streamline this computation
    for i in modifiedString.splitlines(True):
        data = i.rsplit(" ",1)
        if(len(data)==2):
            word = data[0]
            tag = data[1].rstrip('\n')
            if(word == '' or tag not in sentimentSets):
                print("Corrupted data detected: {0}").format(i)
            else:
                dictProcess(modtrainWords,word)
        elif(i == '\n'):
            #print("Just a new line")
            pass
        else:
            print("Corrupted data detected: {0}".format(i))

    #Reading the words inside the testSet that do not appear in the training set
    testWords = {}
    testSetString=""
    if TEST:
        with open('{0}\\test.in'.format(fileDir), 'r',encoding='utf-8') as testSet:
            testSetString = testSet.read()
    else:
        with open('{0}\\dev.in'.format(fileDir), 'r',encoding='utf-8') as testSet:
            testSetString = testSet.read()

    testSetLines = testSetString.splitlines()#This converts all the '\n' to ''
    for i in testSetLines:
        if(i!=''):
            dictProcess(testWords,i)

    wordsNotInTrainingSet = set(testWords) - set(modtrainWords)

    modifiedTestString = ""

    for i in testSetLines:
        if (i != ''):
            if i in wordsNotInTrainingSet:
                modifiedTestString = modifiedTestString+"#UNK#\n"
            else:
                modifiedTestString = modifiedTestString+ i+'\n'
        else:
            modifiedTestString = modifiedTestString+ '\n'

    return (tagCount,modifiedString,modifiedTestString)
```

```
#    with open('{0}\modifiedTest.txt'.format(fileDir), 'w',encoding='utf-8') as output
#        outputTestFile.write(modifiedTestString)
#
#
#    with open('{0}\modifiedTrain.txt'.format(fileDir), 'w',encoding='utf-8') as outpu
#        outputFile.write(modifiedString)
#


def dictProcess(dictionary, key):
    dictionary[key] = dictionary.get(key,0)+1


#Returns a dictionary with the emission parameters
def computeEmissions(fileDir, tagCount, modifiedTrainingString):
#    with open('{0}\modifiedTrain.txt'.format(fileDir), 'r',encoding='utf-8') as modTr
#        trainSetString = modTrainSet.read()
    trainSetString = modifiedTrainingString
    emissionParameters = {}
    #Compute the emission counts
    trainSetLines = trainSetString.splitlines(True)
    for i in trainSetLines:
        data = i.rsplit(" ",1)
        if(len(data)==2):
            word = data[0]
            tag = data[1].rstrip('\n')
            if(word == '' or tag not in sentimentSets):
                print("Corrupted data detected: {0}".format(i))
            else:
                nestedDictProcess(emissionParameters,word,tag)#Builds up the dictionar
        elif(i == '\n'):
            #print("Just a new line")
            pass
        else:
            print("Corrupted data detected: {0}".format(i))

    #Compute the observation parameters
    emitParams = buildEmissionParameters(emissionParameters,tagCount)#Builds up the pa
    return emitParams

def nestedDictProcess(dictionary,key,subKey):
    if key not in dictionary:
        dictionary[key]={}
        dictionary[key]["count"] = {subKey:1}
    else:
        dictionary[key]["count"][subKey] = dictionary[key]["count"].get(subKey,0)+1 #I

def buildEmissionParameters(dictionary, tagCount):
    for key, value in dictionary.items():
        parameters = {}
        for subKey,subvalue in value["count"].items():
            parameters[subKey] = subvalue/tagCount[subKey]
        dictionary[key]["parameters"] = parameters
    return dictionary


def save_obj(obj, fileDir, fileName ):
    with open('{0}\\variables\{1}.pkl'.format(fileDir,fileName),'wb') as f:
        pickle.dump(obj, f, pickle.HIGHEST_PROTOCOL)


def load_obj(fileDir, fileName):
    with open('{0}\\variables\{1}.pkl'.format(fileDir,fileName), 'rb') as f:
        return pickle.load(f)



def detectAnomalies(fileDir):
    with open('{0}\modifiedTrain.txt'.format(fileDir), 'r',encoding='utf-8') as modTra
        trainSetString = modTrainSet.read()
    trainSetLines = trainSetString.splitlines(True)
    sentences = 0
    index = 0
    indices = ""
    for i in trainSetLines:
        index+= 1
        data = i.split(" ")#Such was a wrong way of splitting
        if(len(data)==2):
            word = data[0]
            tag = data[1].rstrip('\n')
            if(word == '' or tag not in sentimentSets):
                print("Corrupted data detected: {0}".format(i))
                indices= indices +"{0} {1}\n".format(i,index)
        elif(i == '\n'):
            sentences +=1
        else:
            print("Corrupted data detected: {0}".format(i))
            indices= indices +"{0} {1}\n".format(i,index)

### END PART 2


### PART 3

sentimentSets = ["START","STOP","O","B-positive","I-positive","B-neutral","I-neutral",'
```

```
#TODO: Compute the sentences of the test-Set
def computeSentences(fileDir,cleanedTestString):
#    with open('{0}\modifiedTest.txt'.format(fileDir), 'r',encoding='utf-8') as modTes
#        testSetString = modTestSet.read()
    testSetString=cleanedTestString
    sentences= []
    sentence = []
    testSetLines = testSetString.splitlines()
    for i in testSetLines:
        if (i != ''):
            #Valid ; choose "||" as delimiter
            sentence.append(i)
        else:
            #End of sentence reached
            sentences.append(sentence)
            sentence = []
    return sentences
    #save_obj(sentences,fileDir,"sentences")

def computeTransitions(fileDir,tagCount,modifiedTrainingString):
#    with open('{0}\modifiedTrain.txt'.format(fileDir), 'r',encoding='utf-8') as modTr
#        trainSetString = modTrainSet.read()
    trainSetString = modifiedTrainingString
    transitionParameters = {}
    y_prev = "START"
    y_next = ""
    #Compute the emission counts
    #TODO: deal with the transition parameters
    #TODO: deal with anomalous data
    trainSetLines = trainSetString.splitlines(True)
    for i in trainSetLines:

        data = i.rsplit(" ",1)
        if(len(data)==2):
            word = data[0]
            tag = data[1].rstrip('\n')
            if(word == '' or tag not in sentimentSets):
                print("Corrupted data detected: {0}".format(i))
            else:
                y_next = tag
                nestedDictProcess(transitionParameters,y_prev,y_next)
                y_prev = tag

        elif(i == '\n'):
            #print("Just a new line")
            y_next = "STOP"
            nestedDictProcess(transitionParameters,y_prev,y_next)
            y_prev = "START"
        else:
            print("Corrupted data detected: {0}".format(i))

    transParams = buildTransitionParameters(transitionParameters,tagCount)#Builds up t
    return transParams

def buildTransitionParameters(dictionary, tagCount):
    for y_prev, value in dictionary.items():
        parameters = {}
        for y_next,count in value["count"].items():
            parameters[y_next] = count/tagCount[y_prev]#You must reference the count o
        dictionary[y_prev]["parameters"] = parameters
    return dictionary


def decodeAllSentences(sentences, fileDir, tP, eP):
    fileString = ""
    for sentence in sentences:
        fileString = fileString+viterbiAlgorithm(sentence,tP,eP)+"\n"
    with open('{0}\\dev.p3.out'.format(fileDir), 'w',encoding='utf-8') as outputFile:
        outputFile.write(fileString)


#TODO: Account for log-space
def viterbiAlgorithm(sentence_array, transitionParameters, emissionParameters):
    tP = transitionParameters
    eP = emissionParameters
    terminalValue = 0
    tagSets = {"O":0,"B-positive":0,"I-positive":0,"B-neutral":0,"I-neutral":0,"B-nega
    markovTable = []
    prev_tag = "START"

    trans =0
    emit =0
    #print("Commencing forward computation")
    for i in range(0,len(sentence_array)):
        tagSets = {"O":0,"B-positive":0,"I-positive":0,"B-neutral":0,"I-neutral":0,"B-
        markovTable.append(tagSets)
        observation = sentence_array[i]
        if i == 0:
            #print("Base case")
            for tag in markovTable[i]:
                trans = aUV(tP,prev_tag,tag)
                emit = bVxi(eP,observation,tag)
                if(trans == 0 or emit ==0):
                    markovTable[i][tag] = -inf
                else:
                    markovTable[i][tag] = math.log10(emit*trans)
            #print(markovTable)

        else:
            #print("entering recursive case")
            for tag in markovTable[i]:
                values = []
                #print(markovTable[i-1])
                for prev_tag in markovTable[i-1]:
                    prev_node_val = markovTable[i-1][prev_tag]
                    trans = aUV(tP,prev_tag,tag)
                    emit = bVxi(eP,observation,tag)

                    #Check the values
                    if(prev_node_val is -inf or trans == 0 or emit ==0):
                        tempVal = -inf
                    else:
                        #Need to reassign a new value based on the log space laws
                        #The value in the nodes are already in log base 10
                        tempVal = prev_node_val+math.log10(trans*emit)
                    values.append(tempVal)
                #Set to None if no values available

                markovTable[i][tag] = max(values)



    #print("terminal case")
    lastTag = "STOP"
    values = []
    observation = sentence_array[-1]
    for prev_tag in markovTable[-1]:
        prev_node_val = markovTable[-1][prev_tag]
        trans = aUV(tP,prev_tag,lastTag)
        #emit = bVxi(eP,observation,lastTag) STOP emits NOTHING
        #print("tag: {} trans: {}, prev: {}".format(prev_tag,trans, prev_node_val ))

        if(prev_node_val is -inf or trans == 0):
            tempVal = -inf
        else:
            #Need to reassign a new value based on the log space laws
            #The value in the nodes are already in log base 10
            tempVal = prev_node_val+math.log10(trans)
        values.append(tempVal)
    terminalValue = max(values)

    #Backtracking
    #print("Commencing back trekking with terminal value: {}".format(terminalValue))
    sequenceList = deque()
    latestTag = "STOP"
    for i in range(len(markovTable)-1, -1, -1):
        observation = sentence_array[i]

        for prev_tag,pi in markovTable[i].items():
            transition = aUV(tP,prev_tag,latestTag)
            if (transition ==0 or pi is -inf):
                markovTable[i][prev_tag] = -inf
            else:
                markovTable[i][prev_tag] = pi+math.log10(transition)
        #validEntries = {prev_tag:pi for (prev_tag,pi) in markovTable[i].items() if pi
        parent = max(markovTable[i], key=markovTable[i].get)
        #print("Parent found: {}".format(parent))
        sequenceList.appendleft(parent)
        latestTag = parent

    #Return the observation-state string
    obs_statePair = ""
    for i in range(0, len(sentence_array)):
        word = sentence_array[i]
        tag = sequenceList[i]
        obs_statePair = obs_statePair+"{0} {1}\n".format(word, tag)
    return obs_statePair

### END PART 3



### PART 5
#Part 2 cleans train and test data. Test data is further parsed in part 3, into 'sente
#Otherwise, test data ('sentences') is not touched, and train data is used to generate
#part 3 also uses viterbi to train the params. We modify this to train perceptrons



if TRAIN:
    tagCount,modifiedTrainingString,modifiedTestString = preprocess(fileDir,3)
    #save_obj(tagCount, fileDir, "tagCount")
    emissionParameters = computeEmissions(fileDir,tagCount, modifiedTrainingString)
    #save_obj(emissionParameters, fileDir, "emissionParameters")

    transitionParameters = computeTransitions(fileDir,tagCount, modifiedTrainingString
    #save_obj(transitionParameters,fileDir,"transitionParameters")

    #use training data without tags to fit perceptrons, by using viterbi to guess the

    #trainingSentences = []
    #with open(fileDir+"\\modifiedTrain.txt",'r',encoding='utf-8') as f:
```

```
#    sentence = []
#    for line in f:
#        if line.strip() != "":
#            sentence.append(line.strip().split(' ')[0])
#        else:
#            trainingSentences.append(sentence)
#            sentence =[]
#
##tagged data from the training set.
#trainedSentences = []
#with open(fileDir+"\\modifiedTrain.txt",'r',encoding='utf-8') as f:
#    sentence = []
#    for line in f:
#        if line.strip() != "":
#            sentence.append(line.strip())
#        else:
#            trainedSentences.append(sentence)
#            sentence =[]



#first we change modifiedTrainingString into a list of sentences
listOfTaggedData = modifiedTrainingString.splitlines()
trainedSentences=[]
sentence = []
for line in listOfTaggedData:
    if line.strip() != "":
        sentence.append(line.strip())
    else:
        trainedSentences.append(sentence)
        sentence =[]

#we do the same but strip off the tag for training perceptron
trainingSentences = []
sentence = []
for line in listOfTaggedData:
    if line.strip() != "":
        sentence.append(line.strip().split(' ')[0])
    else:
        trainingSentences.append(sentence)
        sentence =[]

for i in range(NUMOFITER):
    print("AT stage "+str(i)+" out of "+str(NUMOFITER))
    #For each tagged sentence in the training data, find the highest scoring tag s
    numberOfSentences = len(trainingSentences)
    for sentenceIndex in range(numberOfSentences):
        viterOut = viterbiAlgorithm(trainingSentences[sentenceIndex],transitionPar
        #If the highest scoring tag sequence matches the gold, move to next senten
        #   If not, for each feature in the gold but not in the output, add 1 to i
        #           for each feature in the output but not in the gold, take 1 fro

        trainedSentence = trainedSentences[sentenceIndex]
        sentCount = {"O":0,"B-positive":0,"I-positive":0,"B-neutral":0,"I-neutral"

        for j in range(len(trainedSentence)):   #check each word
            viterSent = viterOut[j].split(" ")[1]
            trainedSent = trainedSentence[j].split(" ")[1]
            if viterSent == trainedSent:    #correct match
                continue
            else:   #wrong match, must change weight
                sentCount[viterSent] -= 1
                sentCount[trainedSent] += 1
                #print("  mismatch found: "+viterSent+" : "+trainedSent + "  at li

        change = True #sanity check
        #modify tagCount according to weight (in doing so, effectively change weig
        for senti, value in sentCount.items():
            if value != 0:
                if tagCount[senti]-value <= 0:
                    change = False
        if change:
            for senti, value in sentCount.items():
                if value != 0:
                    #print("  modifying with perceptron")
                    tagCount[senti]-=value

        emissionParameters = computeEmissions(fileDir,tagCount, modifiedTrainingSt
        transitionParameters = computeTransitions(fileDir,tagCount, modifiedTrainin

save_obj(transitionParameters, fileDir, "PercepTrainedTrans")
save_obj(emissionParameters, fileDir, "PercepTrainedEmi")



#apply max-min with our new Tp Eps on "sentences". It will be dev.in for EN and FR for
sentences = computeSentences(fileDir,modifiedTestString)
#sentences = load_obj(fileDir,"sentences")
transitionParameters = load_obj(fileDir, "PercepTrainedTrans")
emissionParameters = load_obj(fileDir, "PercepTrainedEmi")
part4Vit(sentences,fileDir,transitionParameters,emissionParameters)
```

part4 #Entity in gold data: 226 #Entity in prediction: 175

```
#Correct Entity : 108
```

```
Entity  precision: 0.6171
Entity  recall: 0.4779
Entity  F: 0.5387

#Correct Sentiment : 69
Sentiment  precision: 0.3943
Sentiment  recall: 0.3053
Sentiment  F: 0.3441
```

Modifying tagCount: ITER = 1, both trans and emi #Entity in gold data: 226 #Entity in prediction: 213

```
    #Correct Entity : 127
    Entity  precision: 0.5962
    Entity  recall: 0.5619
    Entity  F: 0.5786

    #Correct Sentiment : 73
    Sentiment  precision: 0.3427
    Sentiment  recall: 0.3230
    Sentiment  F: 0.3326

ITER = 3, trans and emi

    #Entity in gold data: 226
    #Entity in prediction: 223

    #Correct Entity : 132
    Entity  precision: 0.5919
    Entity  recall: 0.5841
    Entity  F: 0.5880

    #Correct Sentiment : 79
    Sentiment  precision: 0.3543
    Sentiment  recall: 0.3496
    Sentiment  F: 0.3519

ITER = 10, trans and emi
    #Entity in gold data: 226
    #Entity in prediction: 227

    #Correct Entity : 133
    Entity  precision: 0.5859
    Entity  recall: 0.5885
    Entity  F: 0.5872

    #Correct Sentiment : 80
    Sentiment  precision: 0.3524
    Sentiment  recall: 0.3540
    Sentiment  F: 0.3532
```

FR: Part4 #Entity in gold data: 223 #Entity in prediction: 173

```
#Correct Entity : 113
Entity  precision: 0.6532
Entity  recall: 0.5067
Entity  F: 0.5707

#Correct Sentiment : 73
Sentiment  precision: 0.4220
Sentiment  recall: 0.3274
Sentiment  F: 0.3687
```

Part 5 ITER = 3, both trans and emi #Entity in gold data: 223 #Entity in prediction: 225

```
    #Correct Entity : 147
    Entity  precision: 0.6533
    Entity  recall: 0.6592
    Entity  F: 0.6563

    #Correct Sentiment : 91
    Sentiment  precision: 0.4044
    Sentiment  recall: 0.4081
    Sentiment  F: 0.4062

ITER = 10, both trans and emi
    #Entity in gold data: 223
    #Entity in prediction: 225

    #Correct Entity : 147
    Entity  precision: 0.6533
    Entity  recall: 0.6592
    Entity  F: 0.6563

    #Correct Sentiment : 91
    Sentiment  precision: 0.4044
    Sentiment  recall: 0.4081
    Sentiment  F: 0.4062
```