ISTD 50.007 Machine Learning, Fall 2020

1D Machine Learning Group Project

Design Project

Team 2-7

1004581	Toh Kai Feng
1004288	Bryan Phengan Hengardi
1004241	Amrish Dev Sandhu

Part 1:
We omitted this part since this was individual and done on a different platform with reference to the “Annotation Guideline”.

Part 2:
For this question. We needed to calculate the emission probability of every state using the following formula: Count (S->O)/ Count (S) Where S is the State and O is the Observation as provided in the “ML Project”:

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

We implemented this using 2 python dictionaries to store:
1. the count/ number of occurrences for the emission of State Si To Observation O for every emission observed in the dataset.
2. The count/ number of occurrences of every state observed

```
3.    ycount={}
4.    y_to_x_count={}
```

We also accounted for the possibility of observations that might show up in the test dataset but not in the training dataset using the following formula:

```
    for key in y_to_x_count.keys():
        emission_x_given_y[key]=y_to_x_count[key]/(ycount[key[0]]+k)
    for key in ycount.keys():
        transition=tuple((key,"#UNK#"))
        emission_x_given_y[transition]=k/(ycount[key]+k)
```

As seen above, we added additional emission probabilities from every state to an unknown observation “#UNK#”. This will be used later when running the estimation on the test set to estimate the likelihood of that given observation in the event a new observation (that did not show up in the training dataset) appears.

Part 2 results are as follows:

```
#results for EN

#Entity in gold data: 13179
#Entity in prediction: 18650

#Correct Entity : 9542
Entity  precision: 0.5116
Entity  recall: 0.7240
Entity  F: 0.5996

#Correct Sentiment : 8456
Sentiment  precision: 0.4534
Sentiment  recall: 0.6416
Sentiment  F: 0.5313

#results for CN
#Entity in gold data: 700
#Entity in prediction: 4248

#Correct Entity : 345
Entity  precision: 0.0812
Entity  recall: 0.4929
Entity  F: 0.1395

#Correct Sentiment : 167
Sentiment  precision: 0.0393
Sentiment  recall: 0.2386
Sentiment  F: 0.0675

#results for SG
#Entity in gold data: 4301
#Entity in prediction: 12237

#Correct Entity : 2386
Entity  precision: 0.1950
Entity  recall: 0.5548
Entity  F: 0.2885

#Correct Sentiment : 1531
Sentiment  precision: 0.1251
Sentiment  recall: 0.3560
Sentiment  F: 0.1851
```

Part 3:
For this part, we did the counting of transitions from one state to another in a similar fashion. Our first step was to run a python script to generate all the states for the EN, CN and SG data. We also initialised the tuples emission counter and emission parameter, which are nested dictionaries containing the counts and emission probabilities for each respective states. Transmission_parameter stores a len(states) + 1 X len(states) array to store the transition probabilities.

```
FILE = training_set(language)
ycount = transition_estimator(FILE)
states = list(ycount.keys())

emission_counter = tuple([{} for i in range(len(states))])
emission_parameter = tuple([{} for i in range(len(states))])
transmission_parameter = [[0]*(len(states)+1) for i in range(len(states))]
states.append("STOP")
counter = 0
dic = {}
for state in states:
    dic[state] = counter
    counter +=1
dic["STOP"] = counter
u = 'START'
observation_space = set()
count = [0] * (len(states)-1)
```

For the rest of this part in the report, we will be referencing the code we used for the EN data set. The implementation for the CN and SG data set would be under the else branch.

We implemented the the same algorithm to store:

1. the count/number of occurrences of each transition
2. And the count/number of occurrences of each state

For the viterbi algorithm, the input would be a list of observations, and the output would be a list of states that would be tagged to each observation. We use a nested dictionary to store the emission parameters, as well as a list to store the counts. For the first transition (Start → 1), we will first need to iterate through all the states and calculate the respective probabilities for each respective transition, and store it in the layers list.

```
for j in range(1, num_of_states):
        if (x in observation_space):
            b = emission_parameter[j][x]
        else:
            b = 1.0 / count[j]
        probability = transmission_parameter[0][j] * b
        prev_layer.append((probability, 0))
    layers = [[(1,-1)],prev_layer]
```

We then need to define a forward function that calculates the maximum score for the respective emission/transitions and append it to each layer. This layer will subsequently be appended on to the outer layers list.

```
layer = []
    for i in range(1, num_of_states):
        temp_score = []

        if (x in observation_space):
            b = emission_parameter[i][x]
        else:
            b = 1.0 / count[i]
        for j in range(1, num_of_states):
            j_score = preScore[j-1][0] * (transmission_parameter[j][i]) * b  #
trans 1~7 -> 1-7
            temp_score.append(j_score)
        max_value = max(temp_score)
        max_index = temp_score.index(max_value)
        layer.append((max_value, max_index))
    return layer
```

For the subsequent transitions all the way up to n, we will implement the forward function to recursively calculate the probabilities for each transition, and append them to the respective layers. The idea is similar for the last layer.

```
for i in range(1, n):  # 1 -> n-1
            score = forward(layers[i], X[i], type)
            layers.append(score)
```

We will subsequently calculate the score for each layer and append it to the variable tempscore, and subsequently extract the maximums respectively to the variable max_score.

```
temp_score = []
   for j in range(1, num_of_states):
       t_score = layers[n][j-1][0] * (transmission_parameter[j][num_of_states])
       temp_score.append(t_score)
   max_value = max(temp_score)
   max_index = temp_score.index(max_value)
   layers.append([(max_value, max_index)])
```

For backtracking, we will insert the state with the maximum score for each layer into the optimum list of states, and return it at the end of the function.

```
   parent = 0
   for i in range(n+1, 1, -1):
       parent = layers[i][parent][1]
       Final_sequence.insert(0, states[parent + 1])
   return Final_sequence
```

Part 3 results are as follows:

```
#results for EN
#Entity in gold data: 13179
#Entity in prediction: 14340

#Correct Entity : 10631
Entity  precision: 0.7414
Entity  recall: 0.8067
Entity  F: 0.7726

#Correct Sentiment : 9756
Sentiment  precision: 0.6803
Sentiment  recall: 0.7403
Sentiment  F: 0.7090

#results for CN
#Entity in gold data: 700
#Entity in prediction: 806

#Correct Entity : 202
Entity  precision: 0.2506
Entity  recall: 0.2886
Entity  F: 0.2683

#Correct Sentiment : 120
Sentiment  precision: 0.1489
Sentiment  recall: 0.1714
Sentiment  F: 0.1594

#results for SG
#Entity in gold data: 4301
#Entity in prediction: 4108

#Correct Entity : 2264
Entity  precision: 0.5511
Entity  recall: 0.5264
Entity  F: 0.5385

#Correct Sentiment : 1841
Sentiment  precision: 0.4481
Sentiment  recall: 0.4280
Sentiment  F: 0.4379
```

Part 4:

In order to obtain the top-k best sequences, we need to implement two-dimensional arrays in order to store the additional information. For our code, we initialised an additional variable states in order to do this. Previously, each layer variable would only require one dimensionality in order to store the scores. Now, the scores of each state will be appended onto the layer variable as well to store the top-k best states.

```
    layer = []
    temp_score = []
    viterbi_states = []
    failed = False
    for j in range(1, num_of_states):
        for sub in range(0, len(layers[n][0])):
            t_score = layers[n][j - 1][sub][0] *
(transmission_parameter[j][num_of_states])
            temp_score.append([t_score, j - 1, sub])

    temp_score.sort(key=lambda tup: tup[0], reverse=True)
    for sub in range(0, k):  # get top k best
        viterbi_states.append(temp_score[sub])
    layer.append(viterbi_states)
    layers.append(layer)
```

The implementation will be similar to part 3, with the main difference being the extra states variable that stores the top-k best states.

Part 4 results are as follows for EN data set:

#Entity in gold data: 13179
#Entity in prediction: 14586

#Correct Entity : 10292
Entity precision: 0.7056
Entity recall: 0.7809
Entity F: 0.7414

#Correct Sentiment : 9367
Sentiment precision: 0.6422
Sentiment recall: 0.7108
Sentiment F: 0.6747

Part 5:

We took inspiration from our very first topic in machine learning, the perceptron update algorithm, as well as the question from homework 3, where we learned the possibility of using 2nd order Hidden Markov Models.

For our design, we implement perceptrons and a 2nd order Hidden Markov Model in order to improve the results. As such, two additional states,’START1’ and ‘STOP2’ have to be added in order to account for the 2nd order.

The main concept is to carry out the following tasks at every iteration:

1. Compare the output sequence generation from viterbi with the actual output sequence in the training set
2. Increase the likelihood of transitions and emissions observed in the trainingset
3. Decrease the likelihood of transitions and emissions observed in the output sequence generated by the viterbi algorithm

Note that if the sequence of states generated by the viterbi algorithm is equal to the sequence of states observed in the training set, no updates would be made.

In the train function of our Part 5, we initialise a list YGOLD filled with the ideal states that are tagged onto the observation, and use it to compare with another list of states YTRAIN that is generated by our Viterbi algorithm.

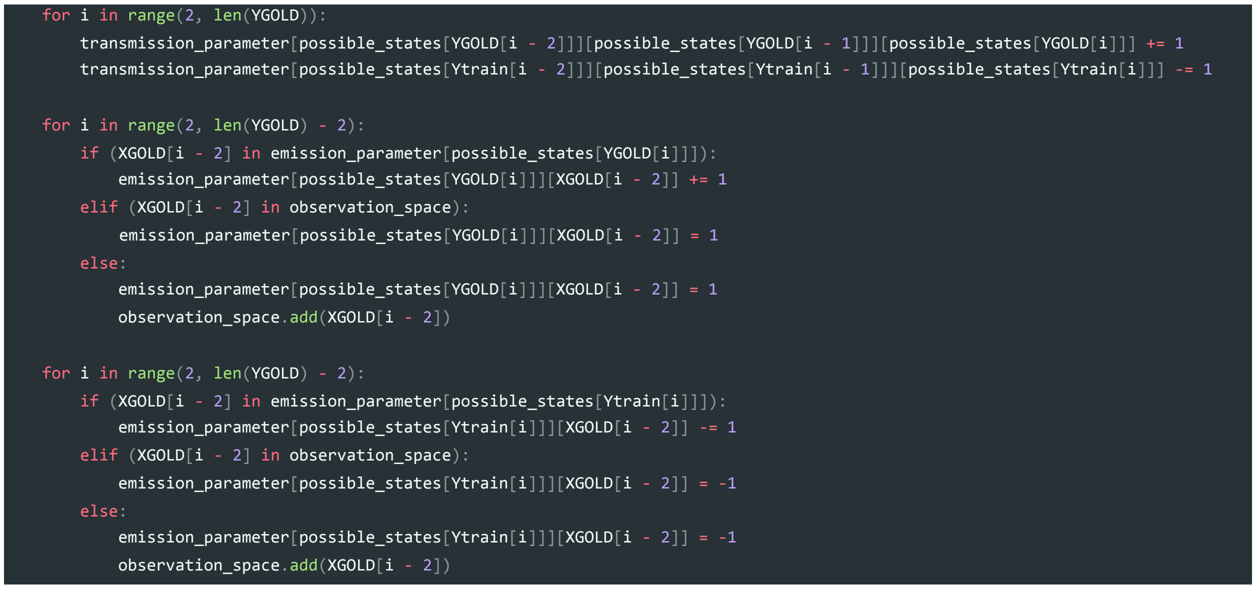
```
                Ygold.extend(['STOP', 'POSTSTOP'])
                Ytrain = ['PRESTART', 'START']
                Ytrain.extend(viterbiAlgo(X))
                Ytrain.extend(['STOP', 'POSTSTOP'])

                update_parameters(Sentence, Ygold, Ytrain)

                # reset
                Ygold = ['PRESTART', 'START']
                Sentence = []
```

We then take these two variables and input them into updateParam, which updates our emission and transition parameters to skew the parameters towards YGOLD. This is done by adding the updating the i,j,k th entry corresponding to the observed sequence of the transmission parameter table by adding 1. The i,j,k th entry refers to the transition from state i, to state j, to state k. The i,j th entry of the emission parameter table refers to the emission from state i to observation j.

```
def update_parameters(XGOLD, YGOLD, Ytrain):
```



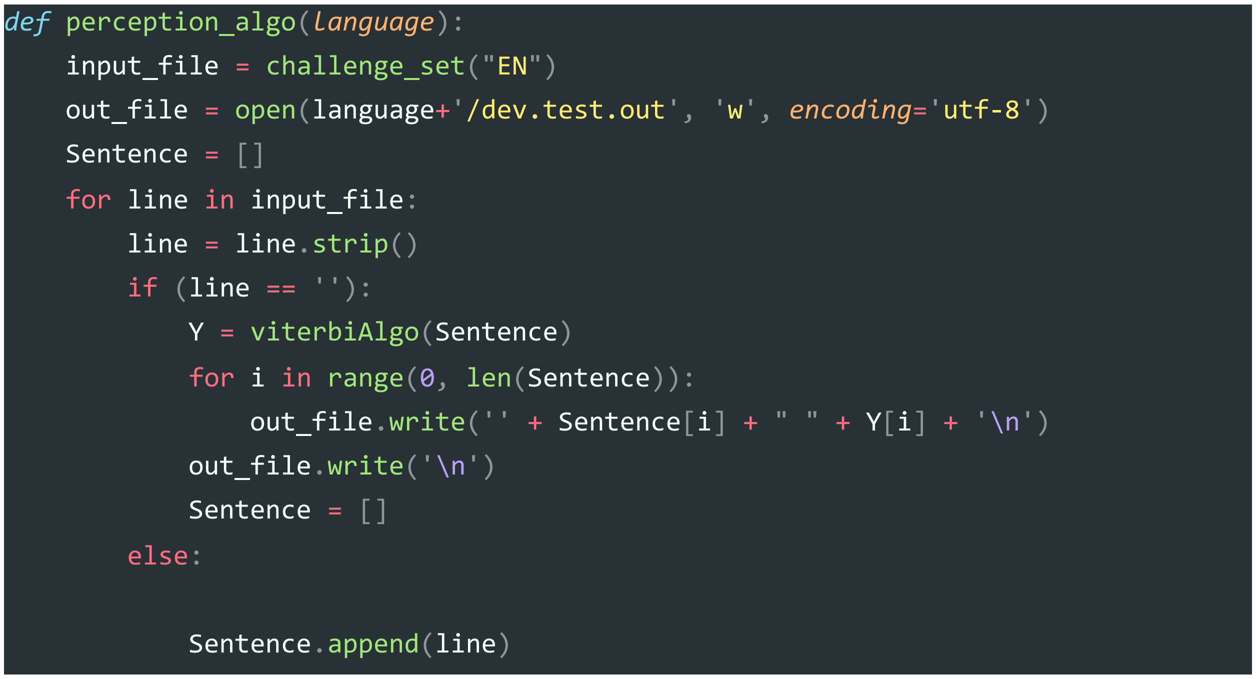
```
    for i in range(2, len(YGOLD)):
        transmission_parameter[possible_states[YGOLD[i - 2]]][possible_states[YGOLD[i - 1]]][possible_states[YGOLD[i]]] += 1
        transmission_parameter[possible_states[Ytrain[i - 2]]][possible_states[Ytrain[i - 1]]][possible_states[Ytrain[i]]] -= 1

    for i in range(2, len(YGOLD) - 2):
        if (XGOLD[i - 2] in emission_parameter[possible_states[YGOLD[i]]]):
            emission_parameter[possible_states[YGOLD[i]]][XGOLD[i - 2]] += 1
        elif (XGOLD[i - 2] in observation_space):
            emission_parameter[possible_states[YGOLD[i]]][XGOLD[i - 2]] = 1
        else:
            emission_parameter[possible_states[YGOLD[i]]][XGOLD[i - 2]] = 1
            observation_space.add(XGOLD[i - 2])

    for i in range(2, len(YGOLD) - 2):
        if (XGOLD[i - 2] in emission_parameter[possible_states[Ytrain[i]]]):
            emission_parameter[possible_states[Ytrain[i]]][XGOLD[i - 2]] -= 1
        elif (XGOLD[i - 2] in observation_space):
            emission_parameter[possible_states[Ytrain[i]]][XGOLD[i - 2]] = -1
        else:
            emission_parameter[possible_states[Ytrain[i]]][XGOLD[i - 2]] = -1
            observation_space.add(XGOLD[i - 2])
```

We can decide the number of iterations to repeat the function train, which will run updateParams multiple times more in order to optimise our transition and emission parameters towards YGold.

After obtaining our ideal emission and transition parameters, we can start outputting to our files using our updated parameters.

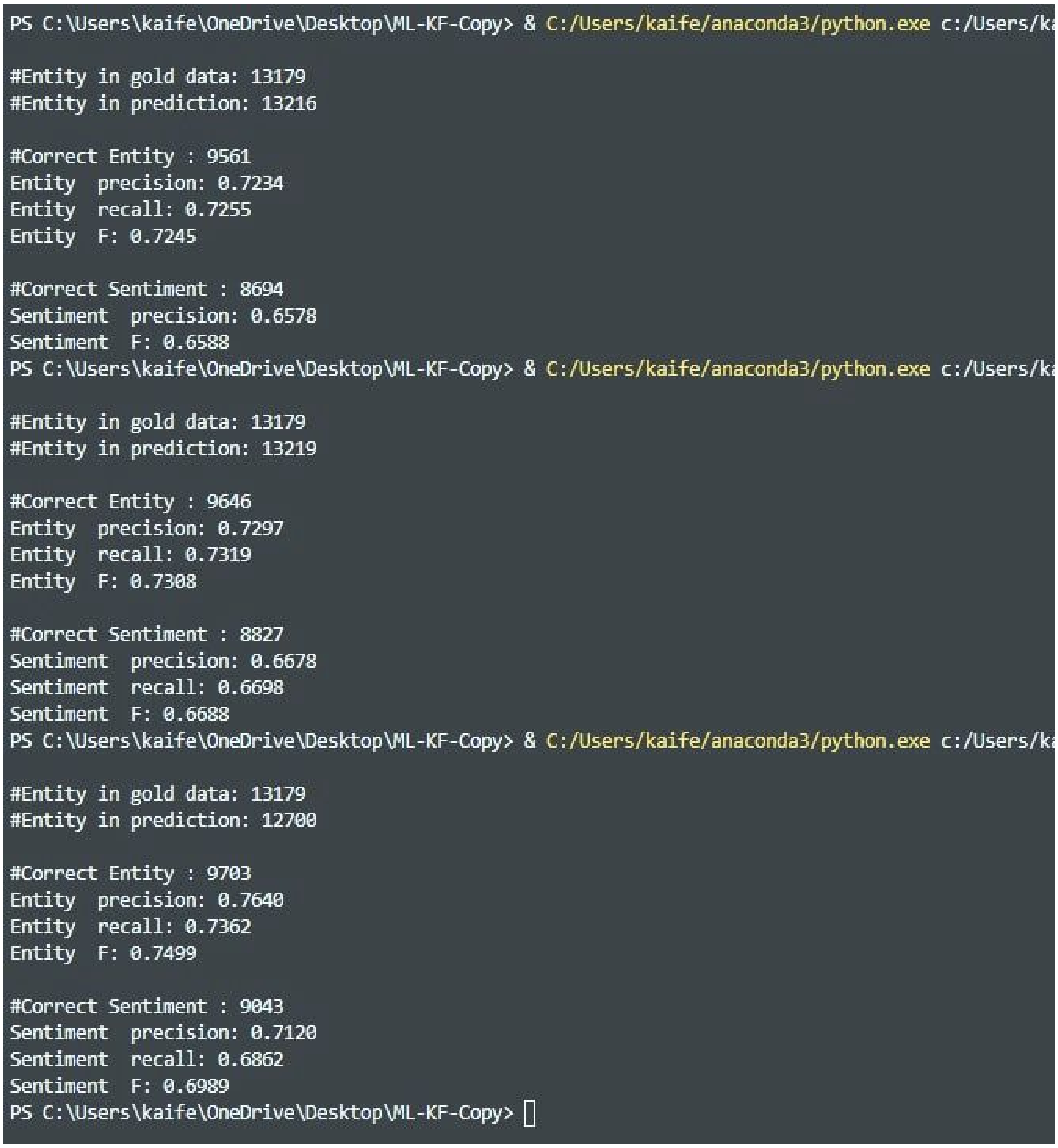


```
def perception_algo(language):
    input_file = challenge_set("EN")
    out_file = open(language+'/dev.test.out', 'w', encoding='utf-8')
    Sentence = []
    for line in input_file:
        line = line.strip()
        if (line == ''):
            Y = viterbiAlgo(Sentence)
            for i in range(0, len(Sentence)):
                out_file.write('' + Sentence[i] + " " + Y[i] + '\n')
            out_file.write('\n')
            Sentence = []
        else:

            Sentence.append(line)
```

For the given test: we ran 15 iterations
For the given dev.in: we ran 2,15 and 30 iterations.

Part 5 results for 2,15,30:



```
PS C:\Users\kaife\OneDrive\Desktop\ML-KF-Copy> & C:/Users/kaife/anaconda3/python.exe c:/Users/ka

#Entity in gold data: 13179
#Entity in prediction: 13216

#Correct Entity : 9561
Entity  precision: 0.7234
Entity  recall: 0.7255
Entity  F: 0.7245

#Correct Sentiment : 8694
Sentiment  precision: 0.6578
Sentiment  F: 0.6588
PS C:\Users\kaife\OneDrive\Desktop\ML-KF-Copy> & C:/Users/kaife/anaconda3/python.exe c:/Users/ka

#Entity in gold data: 13179
#Entity in prediction: 13219

#Correct Entity : 9646
Entity  precision: 0.7297
Entity  recall: 0.7319
Entity  F: 0.7308

#Correct Sentiment : 8827
Sentiment  precision: 0.6678
Sentiment  recall: 0.6698
Sentiment  F: 0.6688
PS C:\Users\kaife\OneDrive\Desktop\ML-KF-Copy> & C:/Users/kaife/anaconda3/python.exe c:/Users/ka

#Entity in gold data: 13179
#Entity in prediction: 12700

#Correct Entity : 9703
Entity  precision: 0.7640
Entity  recall: 0.7362
Entity  F: 0.7499

#Correct Sentiment : 9043
Sentiment  precision: 0.7120
Sentiment  recall: 0.6862
Sentiment  F: 0.6989
PS C:\Users\kaife\OneDrive\Desktop\ML-KF-Copy> 
```