ML Project Report

Members:

1. Loy Pek Yong 1004475

2. Sim Reynard Simano 1006307

3. Lim Yongqing 1005955

4. Aravind Gopinath Nair 1006057

How to Run: Run all in Code.ipynb

Question 1:

For part 1 of the emission parameters. We first iterated through the training set to get all the counts of the different labels. While doing so, we also counted all the (word, label) pairs there were. Then, we calculated the emission parameters.

For part 2. We looped through the dev.in words and associated those words that did not pop up in the training dataset with the special word #UNK#. And then recalculated the emission parameters with the new formula provided.

For part 3. In the simple sentiment analysis, we loop through the words in dev.in. If the word exists in the emission parameter, we assign that word the max value of the (word, label) emission label value possible for that word. If the word is an unseen word, we assign that unseen word the max emission value of the (#UNK#, label) emission parameters.

Results for part 1:

```
ES result is:

#Entity in gold data: 229
#Entity in prediction: 1144

#Correct Entity : 181
Entity  precision: 0.1582
Entity  recall: 0.7904
Entity  F: 0.2637

#Correct Sentiment : 120
Sentiment  precision: 0.1049
Sentiment  recall: 0.5240
Sentiment  F: 0.1748
```

```
RU result is:

#Entity in gold data: 389
#Entity in prediction: 1467

#Correct Entity : 288
Entity  precision: 0.1963
Entity  recall: 0.7404
Entity  F: 0.3103
...
Sentiment  precision: 0.1186
Sentiment  recall: 0.4473
Sentiment  F: 0.1875
```

Question 2

For this question, we first initialize a transition dictionary, which is all the transitions available, and equate it all to 0. From there, we created a function to calculate the transition probability and update the initial transition dictionary. (This is a part of the dictionary):

```
transit_dict = {
    ("O", "O"): 0,
    ("O", "B-positive"):0,
    ("O", "B-negative"):0,
    ("O", "B-neutral"):0,
    ("O", "I-positive"):0,
    ("O", "I-negative"):0,
    ("O", "I-neutral"):0,
    ("O", "START"):0,
    ("O", "STOP"):0,

    ("B-positive", "O"):0,
    ("B-positive", "B-positive"):0,
    ("B-positive", "B-negative"):0,
    ("B-positive", "B-neutral"):0,
    ("B-positive", "I-positive"):0,
    ("B-positive", "I-negative"):0,
    ("B-positive", "I-neutral"):0,
    ("B-positive", "START"):0,
    ("B-positive", "STOP"):0,


    ("B-negative", "O"):0,
    ("B-negative", "B-positive"):0,
    ("B-negative", "B-negative"):0,
    ("B-negative", "B-neutral"):0,
    ("B-negative", "I-positive"):0,
    ("B-negative", "I-negative"):0,
    ("B-negative", "I-neutral"):0,
    ("B-negative", "START"):0,
    ("B-negative", "STOP"):0,

    ("B-neutral", "O"):0,
    ("B-neutral", "B-positive"):0,
    ("B-neutral", "B-negative"):0,
    ("B-neutral", "B-neutral"):0,
    ("B-neutral", "I-positive"):0,
    ("B-neutral", "I-negative"):0,
    ("B-neutral", "I-neutral"):0,
    ("B-neutral", "START"):0,
    ("B-neutral", "STOP"):0,

    ("I-positive", "O"):0,
    ("I-positive", "B-positive"):0,
    ("I-positive", "B-negative"):0,
```

Then we take in the input as a list (by converting them first) and using the emission parameter we obtained from question 1, we start the viterbi algorithm. The algorithm itself is divided into several parts/functions.

1. We get the next possible states given the current word
2. The Viterbi algorithm itself
3. Using our Viterbi algorithm, we then predict the input sequence by finding the highest log value meaning the highest probability. This step here is what gives our words the correct label
4. We then convert back to its original input format

The scoring is as follows:

```
ES result is:

#Entity in gold data: 229
#Entity in prediction: 1014

#Correct Entity : 139
Entity  precision: 0.1371
Entity  recall: 0.6070
Entity  F: 0.2237

#Correct Sentiment : 86
Sentiment  precision: 0.0848
Sentiment  recall: 0.3755
Sentiment  F: 0.1384
```

```
RU result is:

#Entity in gold data: 389
#Entity in prediction: 1392

#Correct Entity : 234
Entity  precision: 0.1681
Entity  recall: 0.6015
Entity  F: 0.2628
...
Sentiment  precision: 0.1078
Sentiment  recall: 0.3856
Sentiment  F: 0.1684
```

Question 3

To obtain the k^{th} best sequence, we will use the same algorithm that we use for question 2 for most of the workings except for the function **processing_pred(pred),** where we modify the function to tailor the mechanism behind it, creating numerous sequences.

E

```python
import heapq

def mod_processing_pred(pred, k):
    all_pred_ls = []
    for seq in pred:
        # Initialize priority queue with the highest value sequence
        queue = []
        grouped_dict = {}
        for key, value in seq.items():
            var = key[0]
            if var in range(0, len(seq)):
                if var not in grouped_dict or value > grouped_dict[var][1]:
                    grouped_dict[var] = (key, value)
        initial_sequence = [item[0] for item in grouped_dict.values()]
        initial_value = sum(item[1] for item in grouped_dict.values())
        heapq.heappush(queue, (-initial_value, initial_sequence))

        # Generate top 10 sequences to limit memory
        pred_ls = []
        for _ in range(10):
            if not queue:
                break
            value, sequence = heapq.heappop(queue)
            pred_ls.append(sequence)

            # Generate new sequences by replacing one element at a time
            for i in range(len(sequence)):
                replaced_key = sequence[i]
                remaining_keys = [key for key in seq.keys() if key[0] == replaced_key[0] and key != replaced_key]
                for new_key in remaining_keys:
                    new_sequence = sequence[:i] + [new_key] + sequence[i+1:]
                    new_value = value - seq[replaced_key] + seq[new_key]
                    heapq.heappush(queue, (-new_value, new_sequence))

        all_pred_ls.append(pred_ls)

    kth_sequences = [seq[min(k-1, len(seq)-1)] for seq in all_pred_ls ]


    return kth_sequences
```

Above is the modified function (you can also see directly in the code we give) where we modify the function as follows:

1. For the first part, it will have the same workings as the original function, returning the best sequence
2. When the best sequence, we would then use the **heapq** module to store the sequence inside
3. We then will generate up to number (in this case 10 to limit memory as well as since we want to obtain the 2nd and 8th) of different sequences given the input
4. The last loop is used to calculate the next maximum sequence until 10 sequences for each given input is generated, and all will be combined in a list
5. Then the last will be using our argument **k** to obtain the kth best sequence

P.S: we can obtain more than 10 sequences for each given input, so adjustments are easy

The rest of the steps are the same working as question 2 and we obtain the following scores:

```
ES result is:

#Entity in gold data: 229
#Entity in prediction: 1044

#Correct Entity : 139
Entity  precision: 0.1331
Entity  recall: 0.6070
Entity  F: 0.2184

#Correct Sentiment : 69
Sentiment  precision: 0.0661
Sentiment  recall: 0.3013
Sentiment  F: 0.1084
```

```
RU result is:

#Entity in gold data: 389
#Entity in prediction: 1424

#Correct Entity : 230
Entity  precision: 0.1615
Entity  recall: 0.5913
Entity  F: 0.2537
...
Sentiment  precision: 0.0955
Sentiment  recall: 0.3496
Sentiment  F: 0.1500
```

Question 4

To improve on our results, we look at possible reasons as to why our previous scores are low

We found that the emission probability is very low since we are looking at the probability of the word given the state, which is very low.

Given that there are a lot of state 'O's, P(word|"O") will be low. In addition, if we look at each particular word in the training set, most of them should have label "O". Hence if we look at P(word|"O"), it will most likely be the lowest out of all the other states due to number of state "O"s being a lot greater, diluting the probability. In other words, we end up predicting the state of the word to be B-neg, B-pos, I-pos etc. much, much more than predicting the state of the word to be "O".

To solve this issue, we decided to look at the probability of the state given the word, rather than the word given the state. We first modify two functions for our emission parameter generation.

```
def train_estimate_emission_parameters_word(train):
    word_count = {}
    emission_count = {}
    label_count = {}

    for row in range(train.shape[0]):
        word = train.iloc[row, 0]
        label = train.iloc[row, 1]
        if (word not in word_count) and (word != "null"):
            word_count[word] = 1
        elif ( word != "null" ):
            word_count[word] += 1

        if (label not in label_count) and (label != "null"):
            label_count[label] = 1
        elif (label != "null"):
            label_count[label] += 1

        if ((word, label) not in emission_count) and (word != "null"):
            emission_count[(word, label)] = 1
        elif (word != "null"):
            emission_count[(word, label)] += 1

    emission_parameter = {}
    for word, label in emission_count:
        emission_parameter[(word, label)] = emission_count[(word, label)] / word_count[word]

    return emission_parameter, emission_count, label_count
```

238] ✓ 0.0s

```
def upgraded_estimate_emission_parameters_word(test_set, emission_parameter, emission_counts, label_counts):
    # predict the label of the test set
    k = 0
    special_word = '#UNK#'
    test_copy = test_set.copy()
    #train word list is all the first argument in keys of emission counts
    train_word_list= [word for word, label in emission_counts.keys()]
    for i in range(len(test_copy["word"])):
        word = test_copy["word"][i]
        if word not in train_word_list:
            test_copy["word"][i] = special_word
            k+=1

    for word in test_copy["word"]:
        for label in label_counts:
            if (word == special_word) and ((word, label) not in emission_parameter):
                emission_parameter[(word, label)] = (k*(label_counts.get(label)/sum(label_counts.values())))/k
                # print((word, label), emission_parameter.get((word, label)))

    return emission_parameter
```

244] ✓ 0.0s

We then redo the training using the Viterbi algorithm that we have implemented and do another round of predictions

As shown in the code, results have improved.

Possible Improvements:

- Use an ensemble model incorporating k-nearest neighbors to generate a set of tags associated with each word.
- By first extracting features (e.g. gender, singular/plural form) of words from the training data and creating language specific feature vectors of such attributes for both training and test data, we can calculate the Euclidean distances of test data words from the training data feature vectors and tag the test words similarly to training data points which they are closest to (by Euclidean distance).
- This could add more language-specific tagging capabilities to the model instead of relying only on the order and frequency of words present.