ML Report

Kishen Durairajoo	1005885
Kelvin Thian	1006020
Kevin Teng	1005262
Ong Zheng Han	1005867

Pretext

Before starting to perform any functions, we start by creating helper functions to read the dev.in files needed for our training.

```
def read_train(filename):
    data = []
    with open(filename, 'r', encoding='utf-8') as file:
        token_seq , tag_seq , current_token , current_tag = [],[],[],[]
        for line in file:
            token_tag = line.strip().split(" ")

            if len(token_tag) == 2:
                current_token += [token_tag[0]]
                current_tag += [token_tag[1]]
            else:
                token_seq += [current_token]
                tag_seq += [current_tag]
                current_token = []
                current_tag = []
        if (len(current_token) != 0):
            token_seq += [current_token]
            tag_seq += [current_tag]
    return token_seq, tag_seq
```

We also developed a read_test function such that we can use the training file for testing as well by extracting only the sequence

```
#Helper function read input test files
def read_test(filename):
    data = []
    with open(filename, 'r', encoding='utf-8') as file:
        token_seq , current_token = [],[]
        for line in file:
            token_test = line.strip()

            if len(token_test) !=0:
                current_token += [token_test]

            else:
                token_seq += [current_token]
                current_token = []

        if (len(current_token) != 0):
            token_seq += [current_token]

    return token_seq
```

We then had to format the possible out put with another helper function that generates a similar output file given a sequence and its corresponding tag

```
#Helper function to save data
def writeoutput(filename, tokens, predicted_tags):
  with open(filename, 'w', encoding='utf-8') as file:

    for i in range(0,len(tokens)):
      for j in range(0,len(tokens[i])):
        file.write(tokens[i][j] + " " + predicted_tags[i][j] + "\n" )

      file.write("\n")
```

Part 1 : Emission Parameter and Sentiment Analysis

Approach for MLE Emission Parameter

Our approach to generating the mission parameter through MLE is to count the occurrence of each token given each tag and then finding the probability of each token occurring before storing it in a dictionary with the key as the emission given the state and the value as the probability. The code can be seen in the ipynb .

```python
def create_emission_dictionary(token_seq, tag_seq,k):
    emission_dictionary = {}
    count_tag = {} #number of words tagged with tag
    count_token_tagged_tag = {} #number of times a token is tagged with tag
    for i in range(0,len(token_seq)):
        for j in range(0, len(token_seq[i])):
            x = token_seq[i][j]
            y = tag_seq[i][j]
            if not (y in count_tag.keys()):
                count_tag[y] = 0

            count_tag[y] += 1

            if not (x in count_token_tagged_tag.keys()):
                count_token_tagged_tag[x] = {}

            if not (y in count_token_tagged_tag[x].keys()):
                count_token_tagged_tag[x][y] = 0

            count_token_tagged_tag[x][y] += 1

    for i in range(0, len(token_seq)):
        for j in range(0, len(token_seq[i])):
            x = token_seq[i][j]
            y = tag_seq[i][j]
            if not(x in emission_dictionary.keys()):
                emission_dictionary[x] = {}
            emission_dictionary[x][y] = (count_token_tagged_tag[x][y]) / (count_t
            emission_dictionary[x]["START"] = 0
            emission_dictionary[x]["STOP"] = 0
    # Part 1ii) Account for unknown words
    emission_dictionary["#UNK#"] = {}
    for tag in count_tag.keys():
        emission_dictionary["#UNK#"][tag] = k / (count_tag[tag] + k)
    uniquetags = list(count_tag.keys())
    return emission_dictionary , uniquetags
```

Approach for MLE for #UNK#

Following which, we then implemented an edge case whereby we replace unknown words with the probability k/count(y)+k as per requirement. The values are stored in the dictionary of emission probabilities as well. All words in the sequence that is not in the training set will then reference this probability during the decoding/ sentiment analysis process.

```python
    emission_dictionary["#UNK#"] = {}
    for tag in count_tag.keys():
        emission_dictionary["#UNK#"][tag] = k / (count_tag[tag] + k)
    uniquetags = list(count_tag.keys())
    return emission_dictionary , uniquetags
```

Approach for Sentiment Analysis

For sentiment analysis, our function takes in a test token in a list and takes the argmax of emission probability to find its most probable tag. This can also be observed in the ipynb

```python
def simple_sentiment_analysis(test_token,emission_dictionary, uniquetags):
    predicted_tags = []
    for token_seq in test_token:
        current_token = []
        for token in token_seq:
            predicted_tag = ""
            max_prob = 0
            for tag in uniquetags:
                prob = emission(token, tag, emission_dictionary, uniquetags)
                if prob > max_prob:
                    max_prob = prob
                    predicted_tag = tag
            current_token += [predicted_tag]
        predicted_tags += [current_token]
        current_token = []
    return predicted_tags
```

Part 2 : Transition Parameter and Viterbi Decoding

Approach for MLE for transition Dictionary

We first accounted for the start and stop tags which are not in the sentences by adding them as new keys to the transition dictionary. We create the transition dictionary for each previous tag transition to the current tag in the form of a tuple key by counting the occurrence of one tag given the previous state over the total occurrence of that previous state.

```python
#Creates Transition Probability Dictionary
def create_transition_dictionary(tag_seq, unique_tags):
    transition_dictionary = {}

    for first_tag in unique_tags:
        for second_tag in unique_tags:
            count, total = 0, 0
            for i in tag_seq:
                total += len(i) - 1
                for j in range(len(i) - 1):
                    if i[j] == first_tag and i[j+1] == second_tag:
                        count += 1
            if count != 0:
                transition_dictionary[(first_tag, second_tag)] = count / total

    start_dict = Counter(i[0] for i in tag_seq)
    stop_dict = Counter(i[-1] for i in tag_seq)

    total_sentences = len(tag_seq)

    for tag, count in start_dict.items():
        transition_dictionary[('START', tag)] = count / total_sentences

    for tag, count in stop_dict.items():
        transition_dictionary[(tag, 'STOP')] = count / total_sentences

    return transition_dictionary
```

Approach for Viterbi Decoding

We start by creating helper functions to retrieve transition and emission probabilities frm the dictionary given 2 tags for transition or 1 tag and 1 token for emission.

```python
# Helper function to retrieve transition probability
def transition(transition_dict, previous_tag, current_tag):
    transition_key = (previous_tag, current_tag)

    if transition_key not in transition_dict:
        transition_dict[transition_key] = math.log(1)

    return transition_dict[transition_key]

# Helper function to retrieve emission probability
def emission(token, tag, emission_dictionary, uniquetags):
    if tag not in uniquetags:
        return math.log(1)
    elif token not in emission_dictionary:
        return emission_dictionary["#UNK#"][tag]
    elif tag not in emission_dictionary[token]:
        return math.log(1)
    else:
        return emission_dictionary[token][tag]
```

We account for edge cases of assign 0 if the tags are not in our unique list of tag which is all the possible tags + stop + start.

Then we created the viterbi function which has the rationale as follows.

1) Create the first layer after start

```python
def viterbi(test_token_seq_sentence, unique_tags, emission_dictionary, transition_dictionary):
    viterbi_probs = {0: {"START": math.log(1)}, **{i: {tag: -math.inf for tag in unique_tags} for i in range(1, len(test_token_seq_sentence)

    # Calculate probabilities for each tag in the first word
    for tag in unique_tags:
        emission_prob = emission(test_token_seq_sentence[0], tag, emission_dictionary, unique_tags)
        transition_prob = transition(transition_dictionary, "START", tag)

        if emission_prob == 0 or transition_prob == 0:
            viterbi_probs[1][tag] = -math.inf
        else:
            viterbi_probs[1][tag] = math.log(emission_prob) + math.log(transition_prob) + viterbi_probs[0]["START"]
```

This layer is separately initiated as it does not have a prior score.

2) Then calculate the hidden layers scores based on the viterbi algorithm by taking the max of the previous layer scores multiplied with the transition probability from prev to current tag which if we take log values, will be each log adding up together.

```
# Calculate probabilities for each tag in the rest of the words
for i in range(1, len(test_token_seq_sentence)):
    next_token = test_token_seq_sentence[i]
    for j in unique_tags:
        max_prob = -math.inf
        for k in unique_tags:
            emission_prob = emission(next_token, j, emission_dictionary, unique_tags)
            transition_prob = transition(transition_dictionary, k, j)

            if emission_prob == 0 or transition_prob == 0:
                prob = -math.inf
            else:
                prob = math.log(emission_prob) + math.log(transition_prob) + viterbi_probs[i][k]

            if prob > max_prob:
                max_prob = prob
        viterbi_probs[i+1][j] = max_prob
```

3) After that, we calculate the last layer's transition to the stop tag

```
# Calculate probabilities for transitioning to the "STOP" tag
max_prob = -math.inf
for tag in unique_tags:
    previous_prob = viterbi_probs[len(test_token_seq_sentence)][tag]
    transition_prob = transition(transition_dictionary, tag, "STOP")

    if transition_prob == 0:
        prob = -math.inf
    else:
        prob = previous_prob + math.log(transition_prob)

    if prob > max_prob:
        max_prob = prob
viterbi_probs[len(test_token_seq_sentence)+1] = {}
viterbi_probs[len(test_token_seq_sentence)+1]["STOP"] = max_prob
```

4) We then backtracked our viterbi to derive the best paths

```
# Backtracking to find the best path
best_path = []
argmax = -math.inf
currentmax = -math.inf
argmax_tag = "NULL"

for tag in unique_tags:
    prob = viterbi_probs[len(test_token_seq_sentence)][tag]
    transition_prob = transition(transition_dictionary, tag, "STOP")
    if transition_prob != 0:
        currentmax = prob + math.log(transition_prob)

        if currentmax > argmax:
            argmax = currentmax
            argmax_tag = tag

best_path.append(argmax_tag)
```

```
    # Backpropagation
    for i in range(len(test_token_seq_sentence), 1, -1):
        argmax = -math.inf
        currentmax = math.log(1)

        for tag in unique_tags:
            prob = viterbi_probs[i-1][tag]
            transition_prob = transition(transition_dictionary, tag, best_path[-1])

            if transition_prob == 0:
                currentmax = -math.inf
            else:
                currentmax = prob + math.log(transition_prob)

            if currentmax > argmax:
                argmax = currentmax
                argmax_tag = tag
        best_path.append(argmax_tag)

    best_path.reverse()
    return best_path
```

5) As we know there are more then 1 sequence, we also created a loop to loop through all and clean the ones with null inputs.

```
def viterbi_loop(test_token_seq, unique_tags, emission_dictionary, transition_dictionary):
    result = []

    for sentence in test_token_seq:
        viterbi_tags = viterbi(sentence, unique_tags, emission_dictionary, transition_dictionary)
        updated_tags = handle_null_tags(sentence, viterbi_tags, unique_tags, emission_dictionary)
        result.append(updated_tags)

    return result

# Helper function to account for null tags in dataset
def handle_null_tags(sentence, viterbi_tags, unique_tags, emission_dictionary):
    updated_tags = []

    for i in range(len(sentence)):
        if viterbi_tags[i] == "NULL":
            word = sentence[i]
            max_emission_value = -math.inf
            best_emission_tag = None

            for tag in unique_tags:
                emission_prob = emission(word, tag, emission_dictionary, unique_tags)
                if emission_prob > max_emission_value:
                    max_emission_value = emission_prob
                    best_emission_tag = tag

            if best_emission_tag is not None:
                updated_tags.append(best_emission_tag)
            else:
                # If no valid emission tag found, keep the original "NULL" tag
                updated_tags.append(viterbi_tags[i])
        else:
            updated_tags.append(viterbi_tags[i])

    return updated_tags
```

Part 3: Kbest Viterbi

To form the K best path, we first have to perform the normal viterbi algorithm and modifying it at each token to maintain a list of the k most probable paths leading to each tag. These paths are sorted by their probabilities in descending order. If more than k paths are available,

it keeps only the top-k paths with the highest probabilities, discarding the rest. In the backtracking step, we collect probabilities for all tags at the last word position. It then sorts these probabilities and associated paths in descending order. From this sorted list, it selects the top-k paths with the highest probabilities as the k best paths and returns it to the loop.

We then created the helper function to write the k-best path into dev.p3.out

```
# Helper to write output into file
def writeoutput_k_viterbi(filename, tokens, predicted_tags, k):
    with open(filename, 'w', encoding='utf-8') as file:
        for i in range(len(tokens)):
            if k > 0 and len(predicted_tags[i]) >= k:
                tags_to_write = predicted_tags[i][k - 1]  # k-1 index for 0-based indexing
            else:
                # If there are fewer than k sequences available, write the last one available
                tags_to_write = predicted_tags[i][-1] if predicted_tags[i] else []


            for j in range(len(tokens[i])):
                if j < len(tags_to_write):
                    file.write(tokens[i][j] + " " + tags_to_write[j] + "\n")
                else:
                    file.write(tokens[i][j] + " " + "NULL" + "\n")
            file.write("\n")
```

It ensures that if a sequence cannot be permutated k times for whatever reason it will just return the last highest possible k. It also ensures to ignore and only track the k value that matters and discards all the smaller k values.

Part 4 : Improved Model

To make a better model, we decided to do the Expectation-Maximisation method whereby in the E-step, we calculate the transition and emission probabilities details in the ipynb.

E-Step

For emission, we calculate the expected counts of emission probabilities for each token and tag combination.
For transition, we calculate the expected transition counts between consecutive tags in the token sequences.

M-Step

For emission, we update the emission probabilities based on the expected counts computed in the E-step and the new emission probabilities are calculated as the ratio of the expected counts to the total count.

For transition, we update the transition probabilities based on the expected transition counts computed in the E-step and also add a smoothing factor to avoid division by zero.

EM Iterations

We then combined the E-step and M-step for both emissions and transitions in an iterative Expectation-Maximization process. It updates emission and transition dictionaries for a specified number of iterations which we set to 200. It uses deepcopy to ensure that the original dictionaries remain unchanged during the iterations. The unk_prob parameter is used for handling unknown tokens in the E-step for emissions which we default to 1 as per part 1.

By doing so we can get a better F1 score than before.

Results

Question	Espanol			Russian		
Scores	Precision	Recall	F1	Precision	Recall	F1
Part 1 Entity	0.1214	0.7773	0.2100	0.1465	0.6838	0.2413
Part 1 Sentiment	0.0662	0.4236	0.1145	0.0710	0.3316	0.1170
Part 2 Entity	0.44	0.0480	0.0866	0.4553	0.1440	0.2187
Part 2 Sentiment	0.4	0.0437	0.0787	0.3496	0.1105	0.168
Part 3 Entity k=2	0.332	0.3712	0.3505	0.347	0.3702	0.3582
Part 3 Sentiment k=2	0.2461	0.2751	0.2598	0.241	0.257	0.2488
Part 3 Entity k=8	0.282	0.4236	0.3386	0.2344	0.329	0.2738
Part 3 Sentiment k=8	0.1512	0.2271	0.1815	0.1172	0.1645	0.1369
Part 4 Entity	0.1241	0.7729	0.2139	0.1473	0.6838	0.2424
Part 4 Sentiment	0.0722	0.4498	0.1245	0.0792	0.3676	0.1303