50.007 Machine Learning

Design Project

Team 25

Ang Song Gee 1004589
Lim Jun Wei 1004379
Dharmapuri Krishna Sathvik 1004286

Part 1

Approach

The function that estimates the emission probability of every state using:

$$e(x|y) = \frac{count(y \to x)}{count(y)}$$

where y represents the tags and x represents the word tokens
The function for the special token #UNK#:

$$e(x|y) = \frac{k}{count(y)+k}$$

where k = 1

```
def get_emission_using_MLE(training, k=1):
    tags = {}
    tags_to_word = {}
    emission = {}
    for data in training:
        word, tag = data[0], data[1]
        if tag in tags:
            tags[tag] += 1
        else:
            tags[tag] = 1

        tag_to_word = (tag, word)
        if tag_to_word in tags_to_word:
            tags_to_word[tag_to_word] += 1
        else:
            tags_to_word[tag_to_word] = 1

    for key in tags_to_word.keys():
        emission[key] = tags_to_word[key] / (tags[key[0]] + k)
    for key in tags.keys():
        transition = (key, UNK_WORD)
        emission[transition] = k / (tags[key] + k)

    return emission
```

To account for when the word does not appear in the training set, we calculated all the emission probabilities from all the tags to #UNK#.

For our simple sentiment analysis system, we compare all emissions for a word and pick the most probable tag emitted from the word.

```
def get_most_probable_tag(emission):
    highest_prob = {}
    output = {}
    for key, prob in emission.items():
        tag, word = key[0], key[1]
        if word not in highest_prob:
            highest_prob[word] = prob
            output[word] = tag
        else:
            if prob > highest_prob[word]:
                highest_prob[word] = prob
                output[word] = tag

    return output
```

Results

Scores for ES Dataset

Entity in gold data: 255
Entity in prediction: 1733

Correct Entity : 205
Entity precision: 0.1183
Entity recall: 0.8039
Entity F: 0.2062

Correct Sentiment : 113
Sentiment precision: 0.0652
Sentiment recall: 0.4431
Sentiment F: 0.1137

Scores for RU Dataset

Entity in gold data: 461
Entity in prediction: 2089

Correct Entity : 335
Entity precision: 0.1604
Entity recall: 0.7267

Entity F: 0.2627

Correct Sentiment : 136
Sentiment precision: 0.0651
Sentiment recall: 0.2950
Sentiment F: 0.1067

Part 2

Approach

To get the transmission parameter, we first pre-processed the training files by extracting the tags and words into separate lists and also creating a new list of tags with ‘START’ and ‘STOP’. Words were extracted from test files as well.

$$a_{u,v} = \frac{count(u; v)}{count(u)}$$

```
def get_transition_pairs(tags):
    transition_pair_count = {}

    for tag in tags:
        # yi-1 and yi tuples
        for tag1, tag2 in zip(tag[:-1], tag[1:]):
            transition_pair = (tag1, tag2)
            if transition_pair in transition_pair_count:
                transition_pair_count[transition_pair] += 1
            else:
                transition_pair_count[transition_pair] = 1

    return transition_pair_count
```

```
def count_y(tag, tags):
    tags_flattened = list(itertools.chain.from_iterable(tags))
    return tags_flattened.count(tag)
```

We created a count dictionary of y_{i-1} and y_i tuples (previous tag and tag) and created a function to collect count of y_i.

With both information collected we then created the parameters dictionary and divided the transition count of y_{i-1} to y_i with count y_i to get the transmission parameters for each tag pair.

```
# divide transition_count by count_yi, to get probability
 for u, transition_row in transition.items():
     count_yi = count_y(u, tags_with_start_stop)
     # words in training set
     for v, transition_count in transition_row.items():
         if count_yi == 0:
             transition[u][v] = 0.0
         else:
             transition[u][v] = transition_count / count_yi
```

For the Viterbi algorithm, in the forward function, we extracted the maximum log value of emission probability of word multiplied by transition probability of tag to next tag all the way to n.

$$\pi(k, v) = \max_{u \in \mathcal{T}} \{\pi(k-1, u) \cdot a_{u,v} \cdot b_v(x_k)\}$$

In our code, pi values are collected using the viterbi_values dictionary.

- Base case:

$$\pi(0, v) = \begin{cases} 1 & \text{if } v = \text{START (starting state, no observations)} \\ 0 & \text{otherwise} \end{cases}$$

Base case is first calculated as shown below in the generate_viterbi_values function and recursively generated till n in the start_viterbi function.

```
if n == 1:
     try:
         if word_list[n - 1] in words_unique:
             try:
                 current_max_viterbi_value = math.log(
                     emission_params[(current_tag, word_list[n - 1])]
                     * transmission_params[START_TAG][current_tag]
                 )
             except KeyError:
                 current_max_viterbi_value = -sys.float_info.max
         else:
             current_max_viterbi_value = math.log(
                 emission_params[(current_tag, UNK_WORD)]
                 * transmission_params[START_TAG][current_tag]
             )
```

```
        except ValueError:
            current_max_viterbi_value = -sys.float_info.max

        viterbi_values[(n, current_tag)] = current_max_viterbi_value
        return
```

```
   for tag in tags_unique:
        # Here, we use a try-except block because our emission parameters only
contain emissions which appeared in our datasets
        # Thus, any unobserved emission will throw a KeyError, however it's value
should be -inf, so we just catch the Error and proceed to the next tag
        # If transmission_params gives 0, math.log will throw a valueError, thus we
catch it and skip the current tag since 0 means we should never consider it
        try:
            if word_list[n - 1] in words_unique:
                try:
                    value = viterbi_values[(n - 1, tag)] + math.log(
                        emission_params[(current_tag, word_list[n - 1])]
                        * transmission_params[tag][current_tag]
                    )
                except KeyError:
                    continue
            else:
                value = viterbi_values[(n - 1, tag)] + math.log(
                    emission_params[(current_tag, UNK_WORD)]
                    * transmission_params[tag][current_tag]
                )
        except ValueError:
            continue

        current_max_viterbi_value = max(current_max_viterbi_value, value)

    viterbi_values[(n, current_tag)] = current_max_viterbi_value
```

At n, to transition to ‘STOP’ we replicate the equation below with just the transition probability.

$$\max_{y_1,\ldots,y_n} p(x_1,\ldots,x_n,y_0=\text{START},y_1,\ldots,y_n,y_{n+1}=\text{STOP}) = \max_{v\in\mathcal{T}}\{\pi(n,v)\cdot a_{v,\text{STOP}}\}$$

```
def start_viterbi(
    word_list, words_unique, tags_unique, emission_params,
transmission_params
):
    global viterbi_values
    max_final_viterbi_value = -sys.float_info.max

    n = len(word_list)

    # Recursive call to generate viterbi_values for (n, tag)
    for tag in tags_unique:
        generate_viterbi_values(
            n,
            tag,
            word_list,
            words_unique,
            tags_unique,
            emission_params,
            transmission_params,
        )

    # Use viterbi values from n to generate viterbi value for (n+1, STOP)
    for tag in tags_unique:
        try:
            value = viterbi_values[(n, tag)] + math.log(
                transmission_params[tag][STOP_TAG]
            )
        except ValueError:
            continue
        max_final_viterbi_value = max(max_final_viterbi_value, value)

    viterbi_values[(n + 1, STOP_TAG)] = max_final_viterbi_value
```

We took the log values instead of absolute values to speed up calculation and solve the potential numerical underflow issue.

To get the predictions, we have to backtrack from ‘STOP’ and take the max argument which is the tag from the tags_unique dictionary. Otherwise the default tag assigned would be ‘O’.

$$y_n^* = \arg\max_v \{\pi(n, v) \cdot a_{v,\text{STOP}}\}$$

```
generated_tag_list = ["" for _ in range(n)]

 # Compute tag for n
 current_best_tag = "O"
 current_best_tag_value = -sys.float_info.max

 for tag in tags_unique:
     try:
         value = viterbi_values[(n, tag)] + math.log(
             transmission_params[tag][STOP_TAG]
         )
     except ValueError:
         continue
     if value > current_best_tag_value:
         current_best_tag = tag
         current_best_tag_value = value

 generated_tag_list[n - 1] = current_best_tag
```

For n-1 all the way to ‘START’, we follow the same equation.

$$y_{n-1}^* = \arg\max_u \{\pi(n-1, u) \cdot a_{u,y_n^*}\}$$

```
for i in range(n - 1, 0, -1):
     current_best_tag = "O"
     current_best_tag_value = -sys.float_info.max

     for tag in tags_unique:
         try:
             value = viterbi_values[(i, tag)] + math.log(
                 transmission_params[tag][generated_tag_list[i]]
             )
         except ValueError:
             continue
         if value > current_best_tag_value:
             current_best_tag = tag
             current_best_tag_value = value

     generated_tag_list[i - 1] = current_best_tag
```

Results

Score for ES Dataset

Entity in gold data: 255
Entity in prediction: 551

Correct Entity : 131
Entity precision: 0.2377
Entity recall: 0.5137
Entity F: 0.3251

Correct Sentiment : 104
Sentiment precision: 0.1887
Sentiment recall: 0.4078
Sentiment F: 0.2581

Score for RU Dataset

Entity in gold data: 461
Entity in prediction: 533

Correct Entity : 219
Entity precision: 0.4109
Entity recall: 0.4751
Entity F: 0.4406

Correct Sentiment : 144
Sentiment precision: 0.2702
Sentiment recall: 0.3124
Sentiment F: 0.2897

Part 3

Approach

To obtain the 5-th best output sequences, our group used a Python dictionary (hashmap) to keep track of every possible sequence and score. As we work backwards from STOP, new tags will be appended to the front of the tag sequence. The dictionary will keep sorting for top 5 best sequences as it iterates. After retrieving the top 5 best sequences, we will return the sequence with the 5-th best score.

```
def generate_predictions_viterbi(word_list, tags_unique, transmission_params):
    global viterbi_values
    total_viterbi_scores = {}
    n = len(word_list)

    for tag in tags_unique:
        try:
            value = viterbi_values[(n, tag)] + math.log(
                transmission_params[tag][STOP_TAG]
            )
            total_viterbi_scores[value] = [tag]
        except ValueError:
            continue

    total_viterbi_scores = get_top_scores_from_dictionary(total_viterbi_scores)

    # Generate predictions starting from n-1 going down to 1
    for i in range(n - 1, 0, -1):
        link = {}
        for tags in total_viterbi_scores.values():

            for tag in tags_unique:
                try:
                    value = viterbi_values[(i, tag)] + math.log(
                        transmission_params[tag][tags[0]]
                    )  # we take the first tag because we are working backwards
                    link[value] = [tag] + tags
                except ValueError:
                    continue

        total_viterbi_scores = get_top_scores_from_dictionary(link)
    return list(total_viterbi_scores.values())[-1]
```

Results

Score for ES Dataset

Entity in gold data: 255
Entity in prediction: 2385

Correct Entity : 124
Entity precision: 0.0520
Entity recall: 0.4863
Entity F: 0.0939

Correct Sentiment : 20
Sentiment precision: 0.0084
Sentiment recall: 0.0784
Sentiment F: 0.0152

Score for RU Dataset

Entity in gold data: 461
Entity in prediction: 1532

Correct Entity : 83
Entity precision: 0.0542
Entity recall: 0.1800
Entity F: 0.0833

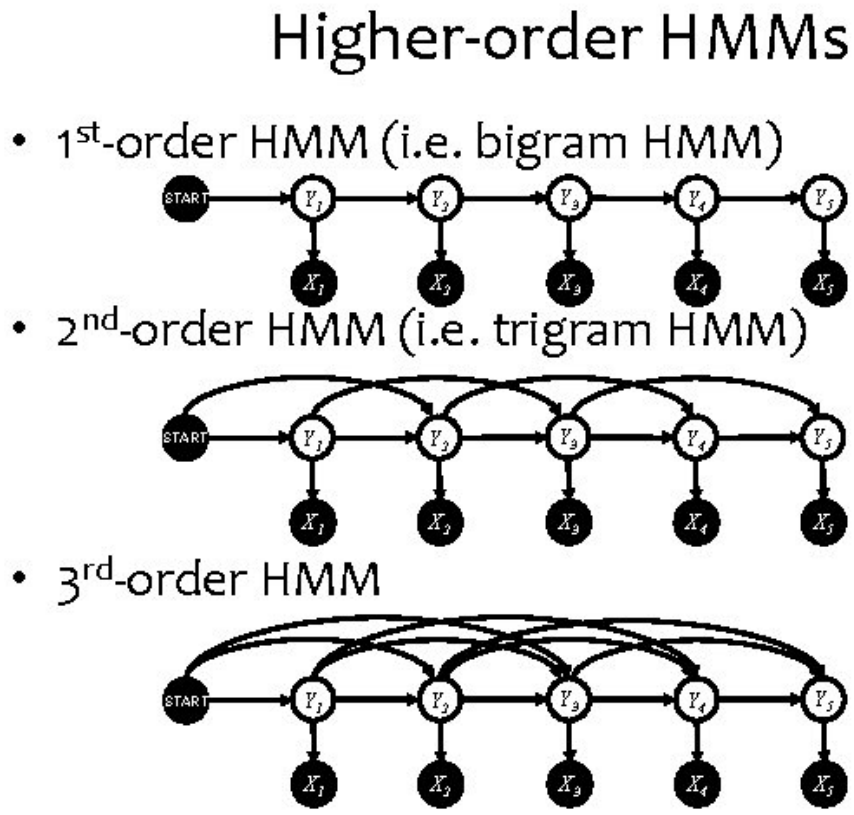
Correct Sentiment : 32
Sentiment precision: 0.0209
Sentiment recall: 0.0694
Sentiment F: 0.0321

Part 4

Approach

Trigram HMM Model (part4b.py)

Firstly, we decided to adopt a 2nd order Trigram HMM model with transition triplets used within the viterbi algorithm. In Part 2, we had already implemented the 1st order bigram HMM model, and we would like to attempt to improve the prediction model by considering the previous 2 transitions.



We still need to generate transition pairs, which are used to predict the tag for the first word in the sequence. For the subsequent words, the generated transition triplets are used for prediction. Firstly, we obtain the count of each transition triplet using get_transition_triplets(), followed by calculating the probabilities in get_transition_triplets_using_MLE().

```python
def get_transition_triplets(tags):
    transition_triplet_count = {}

    for tag in tags:
        for tag1, tag2, tag3 in zip(tag[:-2], tag[1:-1], tag[2:]):
            transition_triplet = (tag1, tag2, tag3)
            if transition_triplet in transition_triplet_count:
                transition_triplet_count[transition_triplet] += 1
            else:
                transition_triplet_count[transition_triplet] = 1

    return transition_triplet_count
```

```python
def get_transition_triplets_using_MLE(
    unique_tags, transition_triplet_count, tags_with_start_stop
):
    transition = {}
    unique_tags = [START_TAG] + unique_tags + [STOP_TAG]

    # Tags (u -> v -> w)
    for u in unique_tags[:-1]:  # omit STOP for first tag
        transition_first_tag = {}
        for v in unique_tags[1:-1]:  # omit START and STOP for second tag
            transition_second_tag = {}
            for w in unique_tags[1:]:  # omit START for third tag
                transition_second_tag[w] = 0.0
            transition_first_tag[v] = transition_second_tag
        transition[u] = transition_first_tag

    # fill up transition parameters
    for u, v, w in transition_triplet_count:
        transition[u][v][w] += transition_triplet_count[(u, v, w)]

    # divide transition_count by count_yi, to get probability
    for u, transition_first_tag in transition.items():
        for v, transition_second_tag in transition_first_tag.items():
            count_yi = count_y(u, tags_with_start_stop)

            for w, instance_in_training_set in transition_second_tag.items():
                if count_yi == 0:
                    transition[u][v][w] = 0.0
                else:
                    transition[u][v][w] = instance_in_training_set / count_yi

    return transition
```

Following that, we use a modified version of our previous Viterbi algorithm, which now takes into account that the first tag in the transition triplet for n=2 is ‘START’.

```
# Handle n == 2 where first tag in triplet is START
    if n == 2:
        for tag in tags_unique:
            try:
                if word_list[n - 1] in words_unique:
                    try:
                        value = viterbi_values[(n - 1, tag)] + math.log(
                            emission_params[(current_tag, word_list[n - 1])]
                            *
transmission_triplet_params[START_TAG][tag][current_tag]
                        )
                    except KeyError:
                        continue
                else:
                    value = viterbi_values[(n - 1, tag)] + math.log(
                        emission_params[(current_tag, UNK_WORD)]
                        * transmission_triplet_params[START_TAG][tag][current_tag]
                    )
            except ValueError:
                continue
            current_max_viterbi_value = max(current_max_viterbi_value, value)

        viterbi_values[(n, current_tag)] = current_max_viterbi_value
        return
```

The rest of the viterbi pi-values are computed similarly to the Bigram model in Part 2, with a small change being that we now loop over 2 sets of tags when computing the max viterbi value for that particular (n, current_tag).

```
# Use viterbi values from n-1 to generate current viterbi value
    for tag1 in tags_unique:
        for tag2 in tags_unique:
            try:
                if word_list[n - 1] in words_unique:
                    try:
                        value = viterbi_values[(n - 1, tag2)] + math.log(
                            emission_params[(current_tag, word_list[n - 1])]
                            * transmission_triplet_params[tag1][tag2][current_tag]
                        )
                    except KeyError:
                        continue
                else:
```

```
                    value = viterbi_values[(n - 1, tag2)] + math.log(
                        emission_params[(current_tag, UNK_WORD)]
                        * transmission_triplet_params[tag1][tag2][current_tag]
                    )
                except ValueError:
                    continue

                current_max_viterbi_value = max(current_max_viterbi_value, value)
        viterbi_values[(n, current_tag)] = current_max_viterbi_value
```

In terms of modifying the prediction generation for Viterbi, we had to handle the edge case where the length of the word is 1, which requires us to fix the first tag in the transition triplet as ‘START’, and the final tag as ‘STOP’.

```
# Handle case where word is length 1
    if n == 1:
        for tag in tags_unique:
            try:
                value = viterbi_values[(1, tag)] + math.log(
                    transmission_triplet_params[START_TAG][tag][STOP_TAG]
                )
            except ValueError:
                continue
            if value > current_best_tag_value:
                current_best_tag = tag
                current_best_tag_value = value

        generated_tag_list[0] = current_best_tag
        return generated_tag_list
```

Otherwise, we proceed to generate predictions for i from n-1 to 1 using the same Viterbi algorithm as in Part 2, with one change being that we now loop over 2 sets of tags, and we use transition triplets instead of pairs.

However, after generating the output file and evaluating our results, we saw a sharp drop in all metrics as compared to the Bigram HMM.

A possible reason why the trigram model performed worse may be due to the small size of the dataset. A smaller dataset could create unseen trigrams, which will cause issues when tagging the test set. Also in our dataset, the trigram probability for two words could be the same even though the bigram probability has a huge difference between them. Hence, there may be an overfitting of our small dataset, causing test data to perform worse.

Results (For Trigram HMM)

Score for ES dataset

Entity in gold data: 255
Entity in prediction: 79

Correct Entity : 34
Entity precision: 0.4304
Entity recall: 0.1333
Entity F: 0.2036

Correct Sentiment : 30
Sentiment precision: 0.3797
Sentiment recall: 0.1176
Sentiment F: 0.1796

Score for RU dataset

Entity in gold data: 461
Entity in prediction: 286

Correct Entity : 129
Entity precision: 0.4510
Entity recall: 0.2798
Entity F: 0.3454

Correct Sentiment : 88
Sentiment precision: 0.3077
Sentiment recall: 0.1909
Sentiment F: 0.2356

Add-1 Estimate for transition pairs (part4.py)

https://digitalscholarship.unlv.edu/cgi/viewcontent.cgi?article=2008&context=thesesdissertations

Due to the previous Trigram HMM Model not succeeding in improving the performance of the Viterbi Algorithm, we proceeded to implement Add-1 Estimate (or Laplace Smoothing) in the calculation of our transition parameters. Prior to this, we noticed that our transition parameters contained some 0 values, which completely removes the possibility of generating those transitions. Adding one to all the count of observed transitions gives unseen events a non-zero probability, which gives the Viterbi algorithm a chance of outputting that transition.

Smoothed unigram probabilities:

$$P(w_x) = \frac{count(w_x) + 1}{N + V}$$

where V is the vocab size

Below shows our modified code for obtaining the transition pair parameters using add-1 estimate. We set all transition counts to 1 at the start to account for the extra observed event, and when computing the count for each tag, we add an amount equal to the extra observations. This allowed us to obtain transition pair parameters with non-zero values as shown in the 2nd code block below.

```
def get_transition_using_add1_estimate(
    unique_tags, transition_pair_count, tags_with_start_stop
):
    unique_tags = [START_TAG] + unique_tags + [STOP_TAG]
    transition = {}
    for u in unique_tags[:-1]:  # omit STOP
        transition_row = {}
        for v in unique_tags[1:]:  # omit START
            transition_row[v] = 1.0
        transition[u] = transition_row

    # populate transition parameters with counts
    for u, v in transition_pair_count:
        transition[u][v] += transition_pair_count[(u, v)]

    # divide transition_count by count_yi, to get probability
    for u, transition_row in transition.items():
        count_yi = count_y(u, tags_with_start_stop) + len(unique_tags) - 1
        # words in training set
        for v, transition_count in transition_row.items():
            if count_yi == 0:
                transition[u][v] = 0.0
            else:
                transition[u][v] = transition_count / count_yi
```

```
{'START': {'B-positive': 0.053545586107091175, 'O': 0.9257115291847564,
'B-negative': 0.013506994693680656, 'I-neutral': 0.000482392667631452, 'B-neutral':
0.005306319343945972, 'I-negative': 0.0004823 0.00078003120124805, 'B-neutral':
0.0015600624024961, 'I-negative': 0.00078003120124805, 'I-positive':
0.12714508580343215, 'STOP': 0.0078003120124804995}, 'O': {'B-positive':
0.03676307886834203, 'O': 0.8831989884621464, 'B-negative': 0.012739054844318002,
'I-neutral': 3.16105579263474e-05, 'B-neutral': 0.002370791844476055, 'I-negative':
3.16105579263474e-05, 'I-positive': 3.16105579263474e-05, 'STOP':
0.06483325430693852}, 'B-negative': {'B-positive': 0.002288329519450801, 'O':
0.7963386727688787, 'B-negative': 0.002288329519450801, 'I-neutral':
0.002288329519450801, 'B-neutral': 0.002288329519450801, 'I-negative':
0.18077803203661327, 'I-positive': 0.002288329519450801, 'STOP':
0.011441647597254004}, 'I-neutral': {'B-positive': 0.019230769230769232, 'O':
0.3269230769230769, 'B-negative':
0.019230769230769232, 'I-neutral': 0.5576923076923077, 'B-neutral':
0.019230769230769232, 'I-negative': 0.019230769230769232, 'I-positive':
0.019230769230769232, 'STOP': 0.019230769230769232}, 'B-neutr04219409282700422,
'B-neutral': 0.004219409282700422, 'I-negative': 0.6413502109704642, 'I-positive':
0.004219409282700422, 'STOP': 0.004219409282700422}, 'I-positive': {'B-positive':
0.0024509803921568627, 'O': 0.3946078431372549, 'B-negative':
0.0024509803921568627, 'I-neutral': 0.0024509803921568627, 'B-neutral':
0.0024509803921568627, 'I-negative': 0.0024509803921568627, 'I-positive':
0.5857843137254902, 'STOP': 0.007352941176470588}}
```


Following that, we used the same Viterbi algorithm we had implemented in Part 2 to generate our output. The results for our dev dataset are shown below in the Results section. Generally, the results obtained were similar to those in Part 2, with a slight decrease in all 3 metrics overall.

Given more time, we would attempt to try out other methods of smoothing such as Good-Turing Smoothing, which seem to show better results in practice as compared to Add-1 Smoothing.

Results (For Add-1 Estimate)

Score for ES dataset

Entity in gold data: 255
Entity in prediction: 576

Correct Entity : 131
Entity precision: 0.2274
Entity recall: 0.5137
Entity F: 0.3153

Correct Sentiment : 104

Sentiment precision: 0.1806
Sentiment recall: 0.4078
Sentiment F: 0.2503

Score for RU dataset

Entity in gold data: 461
Entity in prediction: 552

Correct Entity : 222
Entity precision: 0.4022
Entity recall: 0.4816
Entity F: 0.4383

Correct Sentiment : 145
Sentiment precision: 0.2627
Sentiment recall: 0.3145
Sentiment F: 0.2863

All outputs are in their respective folders, ES and RU.

Shortcomings of dataset

Our group finds that a possible reason our models performed poorly could be due to the small training dataset provided. The training dataset was also imbalanced because most of the word tokens are tagged with O. If there was an equal distribution of tags for the training word tokens, the model might have performed better.