50.007 Machine Learning Design Project

Group Members: **2**
Instructions to Run Code **2**
Part 1(Refer to the part1.ipynb file) **3**
Subpart a (5 points) 3
Approach 3
Subpart b (10 points) 4
Approach 4
Subpart c (10 points) 4
Approach
In the testing phase, we intend to label the given test set dev.in by following the steps below 4
ES and RU Evaluation Results 5
Part 2 (Refer to the part2.ipynb file) **6**
Subpart a (10 points) 6
Approach 6
Subpart b (15 points) 7
Approach 7
ES and RU Evaluation Results 8
Part 3 (Refer to the part3.ipynb file) **9**
Subpart a (25 points) 9
Approach 9
Algorithm Overview: 9
Data Structure: 11
Main Loop: 11
ES and RU Evaluation Results 12
Part 4 (Refer to the part4.ipynb file) **15**
Subpart a (10 points) 15
Approach 15
General Algorithm: 15
Smoothing Techniques Used: 16
Laplace Smoothing: A simple additive smoothing by adding counts to “pseudocounts” to ensure no probability is 0. 16
Witten-Bell Smoothing: 16
Absolute Discounting: 17
Stages for our new system: 17

1. Data Preparation 17
2. Hyperparameter Tuning 17
3. Model Training 17
4. Prediction & Cross Validation 18
5. Final Model Selection & Testing 18
Subpart b (15 points) 19
Results -ES and RU Evaluation 19
Other considerations: 20
Bidirectional HMMs 20
Second-order HMM 20

Group Members:

Toh Hengyi Lucas 1006061

Mohammed Ansar Ahmed 1006015

Wang Jun Long Ryan 1005923

Jone Chong 1006338

Instructions to Run Code

You may clone this Git repo: https://github.com/a-nnza-r/ML-proj.git
Or
Refer to the submitted zip file

Each Python notebook represents Q1, Q2, Q3 and Q4 respectively. Run all cells for each notebook to get the results as listed in this document.

Part 1(Refer to the part1.ipynb file)

Subpart a (5 points)

Approach

For Part 1, our approach to estimate the emission parameters from the training set using MLE follows these steps:

1. We first read the training contents file and split by “\n” to separate each of the sentences.
2. Next, for each word-label pair, we split the line by spaces to separate the word and the label. This step also considers the presence of spaces in the words itself such as in the case of the RU training set where there are words with spaces.

   For e.g word = “. ..”, label = “O”

   ```
   очень O
   вкусно O
   . .. O
   ```

3. After separating the words and the labels, we can start the supervised learning process by calculating the emission parameters as follows :

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

   Calculation of the emission parameter for all labels and the words they are calculated by counting the number of times the label is observed to emit word over the total occurrences of the label in the training set.

Subpart b (10 points)

Approach

For Part 2, our fix to this function for computing the emission parameters would be as follows:

We make the following modifications from Part 1 to the emission parameters used.

1. Recalculate all emission parameters for every word that exists in the dataset by adding 1 to the denominator (i.e the number of occurrences of label + 1)
2. Calculate the emission parameters e("#UNK# | y) = 1/(count(y)+1) for all labels y.

Subpart c (10 points)

Approach

In the testing phase, we intend to label the given test set dev.in by following the steps below

1. Split the test set dev.in by "\n" to separate the words.
2. For each test word in the test words:
    1. We check if the test word exists in the training set.
        a. If the test word exists, then we will compare the emission probability of that test word by the labels that emit it and pick the label that has the highest emission probability of that test word.
        b. If the test word DOES not exist, then we will compare the emission probability of the word "#UNK#" by all the labels and pick the label that has the highest emission probability of "#UNK#".
3. Now for each test word, there is an associated predicted label attached to it based on the above method.
4. We create dev.p1.out by creating the file where each line represents the test word and the predicted label, placing spaces where necessary to match the format of dev.out.

ES and RU Evaluation Results

```
# Evaluation of ES
!python EvalScript/evalResult.py Data/ES/dev.out Data/ES/dev.p1.out
```

```
#Entity in gold data: 229
#Entity in prediction: 1466

#Correct Entity : 178
Entity  precision: 0.1214
Entity  recall: 0.7773
Entity  F: 0.2100

#Correct Sentiment : 97
Sentiment  precision: 0.0662
Sentiment  recall: 0.4236
Sentiment  F: 0.1145
```

```
# Evaluation of RU
!python EvalScript/evalResult.py Data/RU/dev.out Data/RU/dev.p1.out
```

```
#Entity in gold data: 389
#Entity in prediction: 1816

#Correct Entity : 266
Entity  precision: 0.1465
Entity  recall: 0.6838
Entity  F: 0.2413

#Correct Sentiment : 129
Sentiment  precision: 0.0710
Sentiment  recall: 0.3316
Sentiment  F: 0.1170
```

Part 2 (Refer to the part2.ipynb file)

Subpart a (10 points)

Approach

For Part 1, we will estimate the transition parameters from the training set using MLE as follows:

1. First, we split the training file contents by “\n” to separate each word and label, following the steps as in Question 1 to deal with cases where there are spaces in the word.
2. Next, we extract only the labels from each word-label pair and append it to training labels according to the sequence in the training set. In lines where there is no word-label pair (i.e a space), we will instead substitute it with the labels : STOP,START where STOP indicates the end of the previous sentence and START indicates the start of the next sentence.
3. We add START to the index 0 position of the training labels list as well as ensure that the last label is a STOP instead of a STOP,START.
4. Now we have prepared the training labels set, we can start calculating the transition probabilities is follows :

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

5. For each label y_i in the training labels set, we will look at the label y_j directly in front of it. We add 1 to the numerator count from y_i to y_j. We repeat this for every label in the training labels set. Next, we calculate the transition probabilities by dividing all the numerator counts (number of transition from y_i to y_j) over the denominator count which is the occurrence of the label (number of y_i) for all i and all j.

Subpart b (15 points)

Approach

For Part 2, we intend to use the estimated transition parameters from Q2 and estimated emission parameters from Q1 to perform prediction on the test set dev.in using the Viterbi algorithm.

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

We can assume that the model parameters are already learnt in the previous questions. To run the Viterbi algorithm on the dev.in, we follows these steps:

1. Create a Viterbi matrix where the rows denoted by index v represent the different possible states (e.g START, O, B-positive …) and the columns denoted by index k represent the words/observations at indexes 1 to n, where index 0 represents the start and index n+1 would represent end state as per the Viterbi matrix shown in class. Each entry in this Viterbi matrix stores the highest score/probability at a particular state for a particular word.
2. Create a parent list to store the parent that resulted in the largest probability for every column j in the Viterbi matrix.
3. To begin populating the matrix, we will first initialise the first column where the row that represents START will be set to 1.
4. Next, we will populate the matrix in the subsequent rows in a bottom-up approach.
5. We use v to represent the row index to represent states and k to represent the column index represent the words.
6. For all k from 1 to n+1, for each v for all states excluding START and STOP:
    a. Initialise a max score and a max parent variable.
    b. For each state u for all states, calculate viterbi_matrix[u][k-1] * emission probability at state v for word at index k-1 * transition probability from state u at k-1 to state v at k.

    c. Keep track of the u that leads to the highest value.
    d. Keep track of the highest value for each u.
    e. After exiting the for loop at b, set the max score at index k and state v to the highest value for all the u. Also, set the max parent at index k and state v to be the u that leads to the above highest value.
7. When reaching the STOP state, populate viterbi_matrix[STOP state index][n+1] by looking for the highest value of viterbi_matrix[v][n] * transition probability from state v at column k=n to STOP for all v. Once again, we will keep track of the v* that led to the highest value and store this v* as the max parent at index n.
8. Now, we can use the parent list that stores the max parents at each k to trace the optimal sequence of v.

Note : To account for underflow issues in Part 2, we take log for any transition or emission probability. Hence, in the above multiplications of probabilities, they become addition instead as shown in the code.

ES and RU Evaluation Results

```
# Evaluation of ES
!python EvalScript/evalResult.py Data/ES/dev.out Data/ES/dev.p2.out
```

```
#Entity in gold data: 229
#Entity in prediction: 542

#Correct Entity : 134
Entity  precision: 0.2472
Entity  recall: 0.5852
Entity  F: 0.3476

#Correct Sentiment : 97
Sentiment  precision: 0.1790
Sentiment  recall: 0.4236
Sentiment  F: 0.2516
```

```
# Evaluation of RU
!python EvalScript/evalResult.py Data/RU/dev.out Data/RU/dev.p2.out
```

```
#Entity in gold data: 389
#Entity in prediction: 484

#Correct Entity : 188
Entity  precision: 0.3884
Entity  recall: 0.4833
Entity  F: 0.4307

#Correct Sentiment : 129
Sentiment  precision: 0.2665
Sentiment  recall: 0.3316
Sentiment  F: 0.2955
```

Part 3 (Refer to the part3.ipynb file)

Subpart a (25 points)

Approach

The `k_best_viterbi` algorithm is an extended version of the traditional Viterbi algorithm, designed to find the top k best paths in a Hidden Markov Model (HMM). It maintains a data structure called the `scores` dictionary, which stores k-best scores and backpointers for each position and state, enabling efficient path reconstruction.

Algorithm Overview:

1. Input:
   - `y_count`: Count of states in the HMM.
   - `emission_count`: Emission count statistics.
   - `transition_counts`: Transition count statistics.
   - `training_observations_x`: Training observations.
   - `x_input_seq`: Input sequence for which paths are to be found.
   - `k`: The number of top paths to be found (default is 1).

2. Initialization:
   - Initialize the `states` list containing all possible states.
   - Initialize the `scores` dictionary to store k-best scores and backpointers.

3. Position and State Initialization:
   - Iterate over positions `i` from 0 to `n`, and for each state `state`:
     - Initialize a list of k tuples in `scores[(i, state)]`, each containing (score, parent state, index in parent score list).

4. Initialization of STOP State:
   - Set `scores[(n+1, "STOP")]` to `None`, as no transitions lead from STOP.

5. Initialization of START State:
   - Initialize `scores[(0, "START")]` with a single tuple [(0.0, None, None)].
   - This marks the initial state with a score of 0.0 and no parent state.

6. Dynamic Programming Step:
   - Iterate over positions `t` from 1 to `n`, and for each state `v`:
     - Initialize an empty list `all_scores`.
     - For each previous state `u`:
       - Calculate emission and transition probabilities.
       - Compute the current state's score considering all possible paths from `u`.
       - Append the calculated score, `u`, and index in parent score list to `all_scores`.
     - Sort `all_scores` in descending order and store the top k scores in `scores[(t, v)]`.

7. Transition to STOP State:

- Similar to the dynamic programming step, calculate scores for transitions from all states to the STOP state.
- Store the top k scores in `scores[(n+1, "STOP")]`.

8. Backtracking and Path Reconstruction:
- Initialize an empty list `k_best_paths`.
- For each index `idx_in_STOP_list` in the range of k:
  - Initialize an empty list `path`.
  - Retrieve the score, parent state, and index from `scores[(n+1, "STOP")]` for the current index.
  - Backtrack from STOP to START using parent states and indices, constructing the path.
  - Insert the constructed path at the beginning of `k_best_paths`.

9. Return Paths and Scores:
- Return the `k_best_paths` list containing the top k best paths from START to STOP, and the `scores` dictionary containing k-best scores and backpointers.

Data Structure:

The core data structure in the algorithm is the `scores` dictionary. It has the following structure:
- Key: Tuple `(position, state)`
- Value: List of k tuples `(score, parent state, index in parent score list)`

This data structure efficiently stores the k-best scores and their associated information for each position and state, enabling path reconstruction.

Main Loop:

The main loop iterates through each position in the input sequence and each state, performing the following steps:
1. Initialize `all_scores` list.

2. Calculate scores for transitions from previous states to the current state.
3. Store the top k best scores in the `scores` dictionary for the current position and state.

The main loop's purpose is to systematically calculate and store the k-best scores, facilitating the subsequent reconstruction of the top k best paths in the HMM.

Overall, the `k_best_viterbi` algorithm is a complex extension of the Viterbi algorithm that efficiently finds the k-best paths in an HMM, making use of dynamic programming and careful data structure management.

ES and RU Evaluation Results

```
# ES Evaluation (2nd)
!python EvalScript/evalResult.py ./Data/ES/dev.out ./Data/ES/dev.p3.2nd.out
```

✓ 0.0s

```
#Entity in gold data: 229
#Entity in prediction: 436

#Correct Entity : 117
Entity  precision: 0.2683
Entity  recall: 0.5109
Entity  F: 0.3519

#Correct Sentiment : 65
Sentiment  precision: 0.1491
Sentiment  recall: 0.2838
Sentiment  F: 0.1955
```

```
# ES Evaluation (8th)
!python EvalScript/evalResult.py ./Data/ES/dev.out ./Data/ES/dev.p3.8th.out
```

✓ 0.1s

```
#Entity in gold data: 229
#Entity in prediction: 438

#Correct Entity : 102
Entity  precision: 0.2329
Entity  recall: 0.4454
Entity  F: 0.3058

#Correct Sentiment : 56
Sentiment  precision: 0.1279
Sentiment  recall: 0.2445
Sentiment  F: 0.1679
```

```
# RU Evaluation (2nd)
!python EvalScript/evalResult.py ./Data/RU/dev.out ./Data/RU/dev.p3.2nd.out
```
✓ 0.0s

```
#Entity in gold data: 389
#Entity in prediction: 696

#Correct Entity : 202
Entity  precision: 0.2902
Entity  recall: 0.5193
Entity  F: 0.3724

#Correct Sentiment : 124
Sentiment  precision: 0.1782
Sentiment  recall: 0.3188
Sentiment  F: 0.2286
```

```
# RU Evaluation (8th)
!python EvalScript/evalResult.py ./Data/RU/dev.out ./Data/RU/dev.p3.8th.out
```
✓ 0.0s

```
#Entity in gold data: 389
#Entity in prediction: 703

#Correct Entity : 172
Entity  precision: 0.2447
Entity  recall: 0.4422
Entity  F: 0.3150

#Correct Sentiment : 94
Sentiment  precision: 0.1337
Sentiment  recall: 0.2416
Sentiment  F: 0.1722
```

Part 4 (Refer to the part4.ipynb file)

Subpart a (10 points)

Approach

We focused on increasing the F scores for the development set as defined in the question (and provided in the evaluation script). To do so, we use various smoothing techniques such as Laplace smoothing, Witten Bell Smoothing and Absolute Discounting to handle zero probabilities (which will affect the probabilities for unseen words in the test data set).

General Algorithm:

1. Define necessary constants such as results (dictionary) and k_values (a list with a range of hyperparameters to try) to be used later.
2. Data Preparation stage
3. Estimate transmission probabilities.
4. Model Training: Estimate emission probabilities with different smoothing techniques and store their results into the results dictionary.
5. Prediction and Validation: Predict using viterbi's algorithm and select the best smoothing method (highest entity F scores)
6. Use the selected models[1] to predict tags on the development set (which we also used as the validation set) as well as the given test set.

[1] The selected models (emission and transmission probabilities) in our system are different for both languages

We came up with this general algorithm after considering other methods and realized that the most (time and meaningfully) efficient way (within our capabilities) to improve F scores without drastically changing the system entirely is to train models using various smoothing methods.

Hence, we explored three different smoothing techniques and experimented with the hyperparameters (if any) with the objective of achieving a robust system. We considered the nature of the project[2], and realized that there can be a lot of unknown words in any test data set. As such, we wanted the new system to train a model to be as robust as possible by accounting for 0 probabilities and unknown words.

Smoothing Techniques Used:

Laplace Smoothing: A simple additive smoothing by adding counts to "pseudocounts" to ensure no probability is 0.

The probability is given by: $P(x) = \frac{x+k}{N+V \cdot k}$

x: Count of the occurrence for the word.

k: Smoothing constant (default value in the implementation is 1) to avoid 0 probabilities.

N: Total count of all occurrences (words).

V: Number of possible unique observations.

Witten-Bell Smoothing:

The emission probability is given by: $P(x) = \lambda \cdot \frac{x}{N} + (1 - \lambda) \cdot \frac{1}{V}$.

x: Count of the occurrence for the word for each each state

T: Number of unique observations (unique words) for each state.

N: Total count of all occurrences for the word for each state.

λ: $\frac{T}{T+N}$

V: Number of unique words in the data set.

[2] The project is concerned with correctly identifying sentiments and entities in tweets which can be highly variable due to other factors such as slangs. Hence, the data used to train on is expected to be noisy and the test data is expected to have a significant number of unknown words.

Absolute Discounting:

redistributing of weightage from observed words to unknown words with a constant d - a hyperparameter to be tuned in the validation step.
The probability is given by: $P(x) = \frac{max(x-d, 0)}{N} + \frac{P(UNK)}{V}$ where $P(UNK) = \frac{d \cdot T}{N}$.

x: Count of the occurrence for the word for each state.

d: A constant (discount) - a hyperparameter.

N: Total count of all occurrences for the word for each state.

T: Total count of unique observations (words) for each state.

V: Total count of unique observations (words) for the entire data set.

Stages for our new system:

1. Data Preparation

- prepare_data(file_path): Prepares the data including mappings for states and observations.

2. Hyperparameter Tuning

- Laplace Smoothing: Iterates through k values (from 0.1 to 1) to optimize emission probabilities.
- Absolute Discounting: Iterates through d values (from 0.01 to 1) to optimize emission probabilities using absolute discounting.
- Witten-Bell Smoothing: Applies Witten-Bell smoothing to emission probabilities.

3. Model Training

- estimate_transmission_parameters(): Estimates transition probabilities.
- estimate_emission_parameters(): Estimates emission probabilities specific to Laplace smoothing.
- estimate_emission_parameters_absolute_discounting(): Specific to Absolute Discounting.

- estimate_emission_parameters_witten_bell(): Specific to Witten-Bell smoothing.

4. Prediction & Cross Validation

- viterbi(): Runs the Viterbi algorithm on validation data (dev data set).
- write_predictions_to_file(): Writes the predicted tags to a file.
- compute_scores(): Computes metrics such as precision, recall, and F1 score for both entities and sentiments by importing the given evaluation script.

5. Final Model Selection & Testing

- best_results(): Identifies the best smoothing method and corresponding hyperparameters.
- Retraining the model with the best method.
- Prediction on the validation set and writing predictions (dev.p4.out).
- Prediction on the test set with viterbi() and writing predictions to file. (test.p4.out)

Subpart b (15 points)

Results -ES and RU Evaluation

```
# ES Evaluation
!python EvalScript/evalResult.py ./Data/ES/dev.out ./Data/ES/dev.p4.out
```

✓ 0.1s

```
#Entity in gold data: 229
#Entity in prediction: 214

#Correct Entity : 136
Entity  precision: 0.6355
Entity  recall: 0.5939
Entity  F: 0.6140

#Correct Sentiment : 106
Sentiment  precision: 0.4953
Sentiment  recall: 0.4629
Sentiment  F: 0.4786
```

```
# RU Evaluation
!python EvalScript/evalResult.py ./Data/RU/dev.out ./Data/RU/dev.p4.out
```

✓ 0.1s

```
#Entity in gold data: 389
#Entity in prediction: 279

#Correct Entity : 186
Entity  precision: 0.6667
Entity  recall: 0.4781
Entity  F: 0.5569

#Correct Sentiment : 134
Sentiment  precision: 0.4803
Sentiment  recall: 0.3445
Sentiment  F: 0.4012
```

Other considerations:

Bidirectional HMMs

In our research, we explored the potential of employing bidirectional Hidden Markov Models (HMMs) to enhance our performance. This approach aims to leverage contextual information by considering both preceding and subsequent observations. We implemented two distinct variants of bidirectional HMMs:

In the first variant, we utilized a standard HMM to calculate forward probabilities and a separate HMM to compute backward probabilities. By comparing the tag label discrepancies between these two sets of probabilities, we aimed to determine the most probable label.

In the second variant, we employed a single HMM to calculate transition probabilities from both the beginning to the end and from the end to the beginning. This allowed us to capture information from both directions.

Upon closer examination, we realized that when employing the same training data for both the forward and backward transition probabilities, the resulting labels were essentially the same, but in reverse order. This observation led us to the conclusion that there were no conflicts in tag labels, and the bidirectional HMM approach was yielding labels identical to those obtained through a unidirectional HMM.

Second-order HMM

We also explored an alternative approach known as the second-order Hidden Markov Model (HMM). The second-order HMM introduces the following key advantages:

1. Capturing Longer Dependencies: Unlike first-order HMMs, second-order HMMs take into account not only the current state but also the state immediately preceding it during transitions. This expanded consideration of context enables these models to capture longer dependencies within a sequence. This capability is particularly valuable in tasks where the context extends beyond just the immediate previous state.
2. Enhanced Representation: Second-order HMMs offer an improved representation of the interactions and influences between language elements. By incorporating information from the previous state, these models can potentially provide a more accurate portrayal of the underlying dynamics, leading to enhanced tagging accuracy.

However, it's important to note that due to time constraints, we encountered challenges in fully debugging and validating this approach as a functional model. On top of that, while second-order HMMs offer promising advantages, we were unable to definitively ascertain their efficacy. This uncertainty stems from a lack of confirmation regarding both the quality and quantity of available training data, which can significantly impact the performance of such models. It is most unfortunate that we weren't able to extensively test the second-order HMM approach within our given timeframe.

There were plenty of other approaches that we were keen to try, like the multi-layered perceptron neural network, but due to time constraints, we have decided to keep within the scope of HMMs.

Aside from changing the models, we also explored other options as given in the hints. For instance, we tried different smoothing methods for our emission parameters. One of them was the Good-Turing smoothing, a technique used to estimate the probabilities of unseen events in a dataset by leveraging the frequencies of events that have been observed. The formula is given by:

$$P_{GT}(r) = \frac{(r+1)\,N_{r+1}}{N_r}$$

where:

- $P_{GT}(r)\ is\ the\ Good - Turing\ smoothed\ probability\ for\ an\ event\ with\ observed\ frequency\ r$
- $N_r\ is\ the\ number\ of\ events\ that\ occurred\ r\ times\ in\ the\ dataset$
- $N_{r+1}\ is\ the\ number\ of\ events\ that\ occurred\ r+1\ times\ in\ the\ dataset.$

This formula essentially estimates the probability of an event with frequency r as the ratio of the number of events that occurred $r+1$ times to the number of events that occurred r times, adjusted by a factor of $r+1$.

However, our empirical results have shown that this was computationally intensive and the results were less than satisfactory.