ASSIGNMENT-2

COURSE ID: CS 563

Natural Language Processing

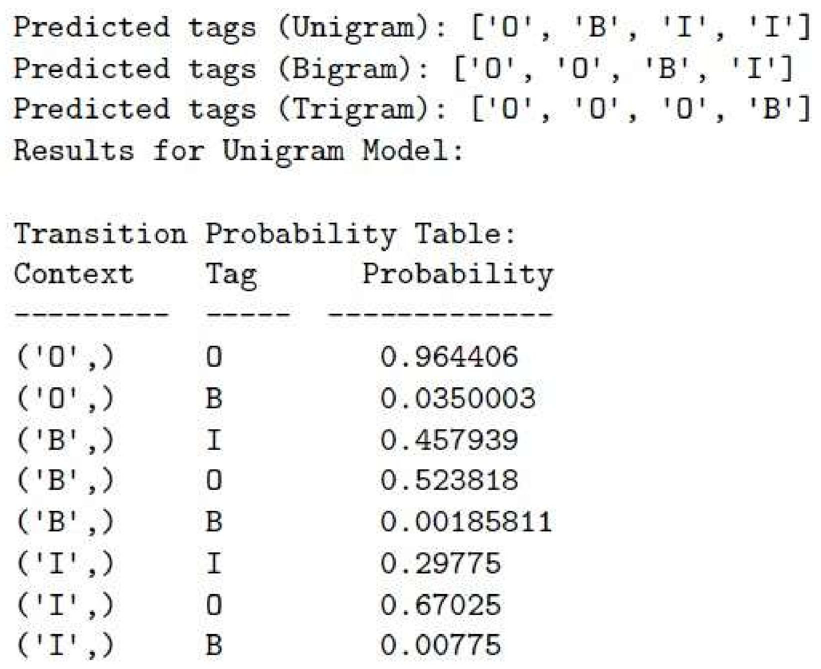
Submitted by:

Ankit Anand

Roll No: 2311MC04

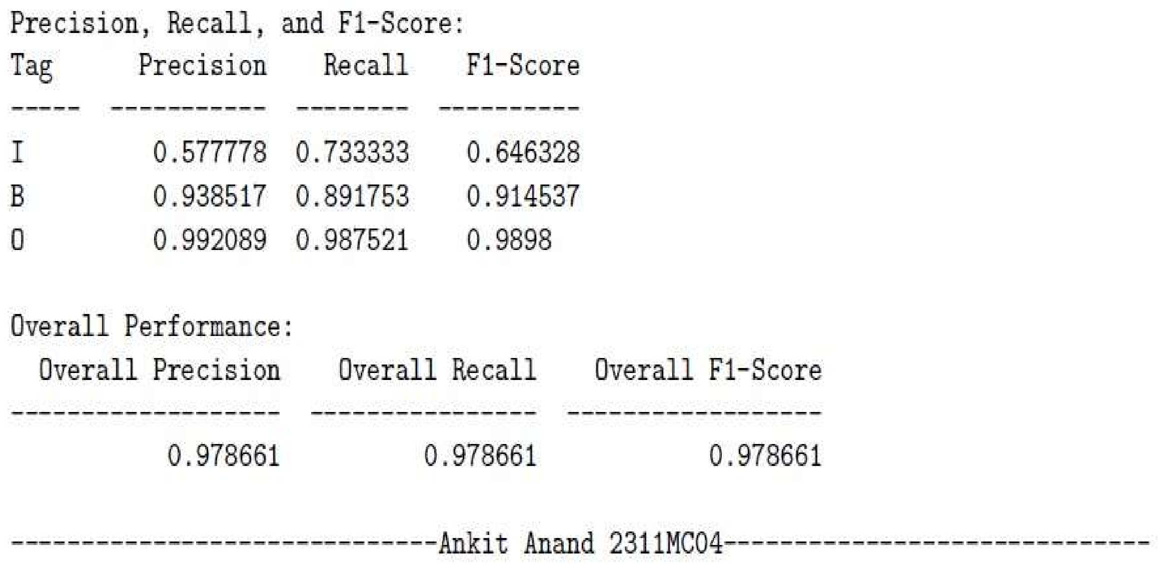
<u>OUTPUT AS PER OUR MODEL</u>

HMM based model:



```
Predicted tags (Unigram): ['O', 'B', 'I', 'I']
Predicted tags (Bigram): ['O', 'O', 'B', 'I']
Predicted tags (Trigram): ['O', 'O', 'O', 'B']
Results for Unigram Model:

Transition Probability Table:
Context    Tag      Probability
---------  -----  -------------
('O',)     O         0.964406
('O',)     B         0.0350003
('B',)     I         0.457939
('B',)     O         0.523818
('B',)     B         0.00185811
('I',)     I         0.29775
('I',)     O         0.67025
('I',)     B         0.00775
```

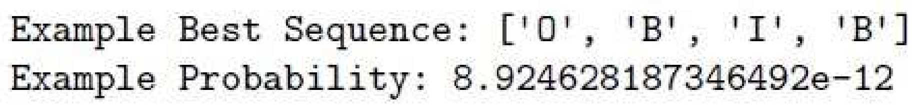


```
Precision, Recall, and F1-Score:
Tag      Precision    Recall    F1-Score
-----  -----------  --------  ----------
I         0.577778  0.733333    0.646328
B         0.938517  0.891753    0.914537
O         0.992089  0.987521    0.9898

Overall Performance:
  Overall Precision    Overall Recall    Overall F1-Score
-------------------  ----------------  ------------------
           0.978661          0.978661            0.978661

------------------------------Ankit Anand 2311MC04------------------------------
```

Viterbi based model:



```
Example Best Sequence: ['O', 'B', 'I', 'B']
Example Probability: 8.924628187346492e-12
```

RNN based Model:

● Explain and draw the architecture of RNN that you are proposing with justification:

A Recurrent Neural Network (RNN) is a type of neural network particularly well-suited for sequential data processing. It maintains a hidden state that is updated at each time step, allowing it to capture dependencies and patterns in sequential data.

Here's a basic overview of the architecture:

1. Input Layer: The input layer takes input data at each time step. For sequential data like text or time series, each input would correspond to a specific element in the sequence (e.g., a word in a sentence).

2. Recurrent Layer: This layer is where the magic of RNNs happens. It maintains a hidden state that captures information from previous time steps. At each time step, the input is combined with the previous hidden state to produce a new hidden state, which is then used for the next time step.

3. Output Layer: The output layer takes the hidden state and produces an output for each time step. For example, in a language modeling task, the output might be the probability distribution over the next word in the sequence.

4. Optional Additional Layers: Depending on the specific task, you might add additional layers such as fully connected layers or convolutional layers after the recurrent layer to perform further processing or feature extraction.

Justification:

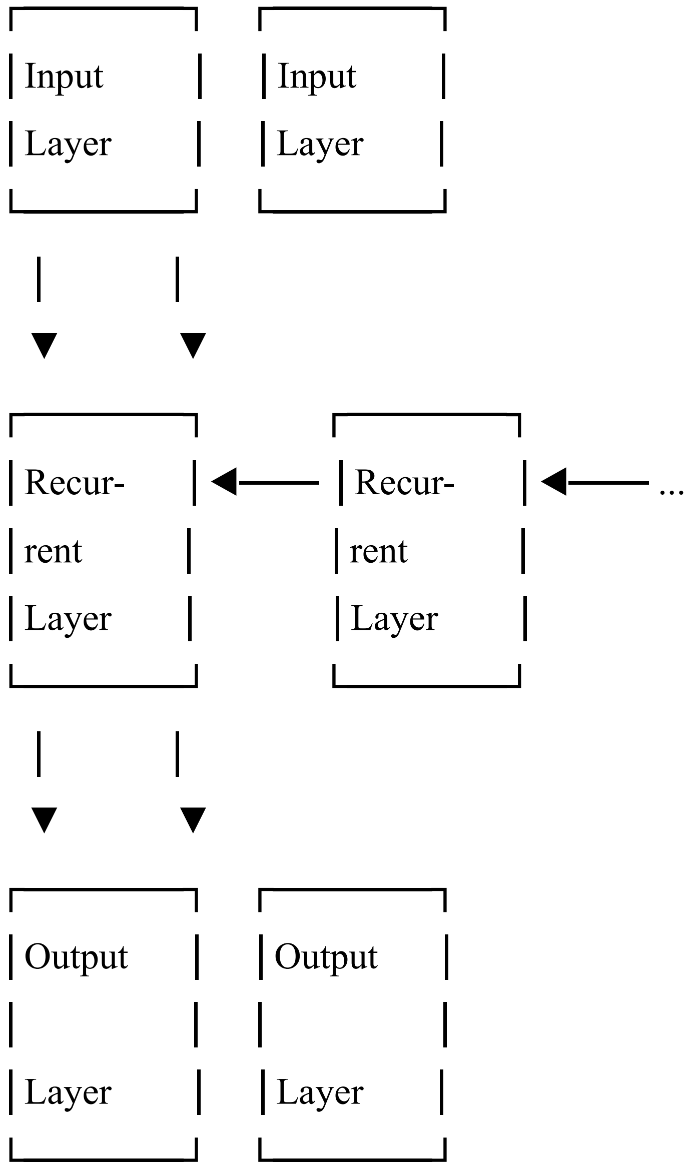
1. Sequential Data Processing: RNNs are designed specifically for processing sequential data, making them a natural choice for tasks like language modeling, time series prediction, and sequence generation.

2. Variable-Length Inputs: RNNs can handle inputs of variable length, which is important for tasks where the length of the input sequence can vary.

3. Capturing Temporal Dependencies: The recurrent connections in RNNs allow them to capture long-range dependencies in sequential data, making them powerful for tasks where context over time is important.

4. Efficient Training: RNNs can be trained efficiently using backpropagation through time (BPTT), which extends the backpropagation algorithm to sequential data by unrolling the network through time.

Architecture Diagram:



This diagram illustrates the flow of data through the layers of the RNN. At each time step, the input is passed through the input layer and combined with the previous hidden state in the recurrent layer to produce a new hidden state. The hidden state is then used to generate an output at that time step. This process repeats for each time step in the input sequence.

● <u>**Describe the features of RNN:**</u>

Recurrent Neural Networks (RNNs) are a class of neural networks specifically designed to handle sequential data. Here are some key features of RNNs:

1. Temporal Dynamics: RNNs are capable of capturing temporal dependencies in sequential data. They can model the context of previous elements in a sequence, making them suitable for tasks where the order of data points matters, such as time series prediction, language modeling, and speech recognition.

2. Recurrent Connections: Unlike feedforward neural networks, which have connections only in the forward direction, RNNs have recurrent connections that allow information to persist over time. This enables them to maintain a memory of previous time steps and use it to inform predictions at the current time step.

3. Variable-Length Inputs: RNNs can handle input sequences of variable length. This flexibility is crucial for tasks where the length of the input sequence may vary, such as natural language processing tasks where sentences can have different numbers of words.

4. Parameter Sharing: In RNNs, the same set of weights is applied at each time step, which allows them to efficiently handle sequences of arbitrary length. This parameter sharing enables the network to generalize well across different time steps.

5. Backpropagation Through Time (BPTT): RNNs are trained using the backpropagation algorithm, extended through time. This means that the gradient of the loss function is backpropagated not only through the layers of the network but also through the time steps, allowing the network to learn from past interactions.

6. Vanishing and Exploding Gradients: One challenge with training RNNs is the issue of vanishing or exploding gradients, which can occur when backpropagating

gradients through many time steps. Techniques such as gradient clipping and using specialized activation functions like LSTM (Long Short-Term Memory) and GRU (Gated Recurrent Unit) cells have been developed to mitigate these issues.

7. Flexibility in Architecture: RNNs can be extended with various architectural modifications to address specific challenges. For example, LSTM and GRU cells introduce gating mechanisms to better control the flow of information through the network, making them more effective at capturing long-term dependencies.

Named Entity Recognition

Named Entity Recognition (NER) is a crucial task in natural language processing (NLP) that involves identifying and classifying entities such as persons, organizations, locations, and more within a text. NER plays a vital role in various NLP applications, including information retrieval, question answering, and sentiment analysis. One approach to tackling the NER problem is by using Hidden Markov Models (HMMs), which leverage probabilistic methods to infer the most likely sequence of named entities given observed words in a text.

Named Entity Recognition (NER) is the process of identifying and classifying named entities in unstructured text data. Named entities typically include entities such as persons, organizations, locations, dates, and more. NER is a fundamental task in natural language processing (NLP) and serves as a building block for various downstream applications such as information extraction, question answering, and sentiment analysis.

Hidden Markov Models (HMMs)

Hidden Markov Models (HMMs) are probabilistic models widely used for sequence modeling tasks. In the context of NER, an HMM can be employed to model the sequence of named entity tags corresponding to a sequence of observed words in a text. The key components of an HMM include:

- States: Represent the possible named entity tags that each word in the text can belong to.

- Transition Probabilities: Define the probabilities of transitioning from one named entity tag to another.

- Emission Probabilities: Define the probabilities of observing a word given a named entity tag.

Solving NER with HMMs

To solve the NER problem using HMMs, the following steps are typically followed:

1. Data Preprocessing: The dataset is preprocessed to extract word-tag pairs from annotated text data.

2. Model Initialization: The HMM is initialized with the specified order (unigram, bigram, or trigram) and smoothing parameters.

3. Training: The HMM is trained on the preprocessed dataset to estimate the transition and emission probabilities.

4. Prediction: The most likely sequence of named entity tags is predicted for a given sequence of words using the Viterbi algorithm.

5. Evaluation: The performance of the HMM-based NER system is evaluated using metrics such as accuracy, precision, recall, and F1-score.

Viterbi Algorithm

Viterbi algorithm is a dynamic programming algorithm used to find the most likely sequence of hidden states (named entity tags) in a sequence of observed events (words). In the context of NER, the Viterbi algorithm efficiently computes the probability of all possible tag sequences and determines the sequence with the highest probability.

The steps involved in the Viterbi algorithm for NER using HMMs are as follows:

1. Initialization: Initialize the trellis matrix and back pointers for dynamic programming.

2. Forward Pass: Compute the probabilities of reaching each state (tag) at each time step based on the previous state probabilities and transition probabilities.

3. Backward Pass: Trace back through the trellis matrix to find the most likely sequence of hidden states (named entity tags) by following the highest probability path.

Named Entity Recognition (NER) is a critical task in natural language processing (NLP) with numerous applications across various domains. Hidden Markov

Models (HMMs) provide an effective framework for solving the NER problem by modeling the sequence of named entity tags in text data. The Viterbi algorithm, a dynamic programming technique, plays a central role in efficiently computing the most likely sequence of named entity tags given observed words. By leveraging HMMs and the Viterbi algorithm, NER systems can accurately identify and classify named entities in text data, thus facilitating more advanced NLP applications and insights.