# Habib University

CS/CE 458/463-L1

Course: Natural Language Processing

Name: Kushal Chandani (kc07535)

Assignment: Homework 2 Report

Date: 15/10/2024

## TABLE OF CONTENTS

## Question 1: Implementing an N-Gram Language Model and Testing Perplexity [20].

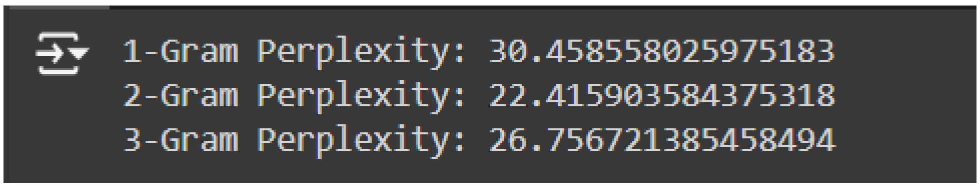
### 1. Approach to Implementing the N-Gram Model

The task was to implement an N-Gram language model to calculate the perplexity of different n values (unigram, bigram, trigram). Here's approach:

- **Data Loading and Preprocessing**:
  - The text from train.txt and test.txt was loaded from the provided URLs.
  - Preprocessing involved converting the text to lowercase and removing non-alphabetic characters and special symbols to maintain only words.
  - The text was tokenized by splitting it into words (tokens).
- **N-Gram Model**:
  - A function to generate n-grams was implemented. This function takes a sequence of tokens and forms n-grams by taking n consecutive tokens as a tuple.
  - An N-Gram model was initialized, which involved maintaining counts for each n-gram and its context (the preceding n-1 tokens). These were stored in dictionaries where contexts mapped to word counts.
  - During training, each n-gram's counts were updated, and a vocabulary of all the words was created.
- **Smoothing**:
  - To handle unseen words in the test set, add-one smoothing (Laplace smoothing) was applied. This ensures that even unseen words get a small non-zero probability by adding 1 to the count of each word in the context and adjusting for the size of the vocabulary.
- **Perplexity Calculation**:
  - Perplexity measures how well the language model predicts a sample. It was calculated using the logarithm of the probability of each word given its context (derived from the n-grams).
  - The sum of the log probabilities was computed, and then exponentiation and normalization were applied to obtain the perplexity score.
  - This process was repeated for unigram, bigram, and trigram models, with perplexity compared for each.

### 2. Results of Perplexity Calculations for Different N-Grams

The N-Gram model was tested with three different n values: unigram (n=1), bigram (n=2), and trigram (n=3). The perplexity values for each model were as follows:



```
1-Gram Perplexity: 30.458558025975183
2-Gram Perplexity: 22.415903584375318
3-Gram Perplexity: 26.756721385458494
```

- **1-Gram Perplexity (Unigram)**: 30.46
- **2-Gram Perplexity (Bigram)**: 22.42
- **3-Gram Perplexity (Trigram)**: 26.76

### 3. Discussion of How N Impacts Model Performance and Perplexity

- **Unigram Model**: The unigram model, which ignores context entirely, has the highest perplexity (30.46). This is expected because it treats each word as independent of the previous words, failing to capture any structure or relationships in the text.
- **Bigram Model**: The bigram model considers one preceding word, making it better at predicting word sequences compared to the unigram model. The perplexity significantly decreases to 22.42, showing that it captures word dependencies, improving prediction accuracy.
- **Trigram Model**: The trigram model, which looks at two preceding words, initially seems promising, but the perplexity increases slightly to 26.76. This suggests that while the trigram model captures more context, it may overfit due to sparse data in longer sequences, leading to slightly worse performance compared to the bigram model.

## Question 2: Implementing a Naive Bayes Classifier for Sentiment Analysis [40].

In this task, I implemented a Naive Bayes classifier from scratch for binary sentiment classification (positive/negative) using the IMDB movie reviews dataset. The classifier was trained on a subset of 1000 reviews (500 positive, 500 negative) and evaluated on a test set of 200 reviews (100 positive, 100 negative).

### 1. Preprocessing and Feature Extraction

- **Text Cleaning:** Each review was cleaned by converting to lowercase, removing punctuation, and eliminating stopwords using **NLTK's** English stopwords.
- **Tokenization:** Cleaned reviews were split into individual tokens.
- **Vocabulary Building:** A vocabulary was created from the training data by extracting unique words across all positive and negative reviews.
- **Feature Extraction:** For each review, the frequency of each word in the vocabulary was calculated, forming a feature vector used in model training.

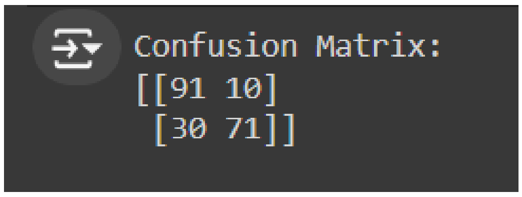
### 2. Naive Bayes Classifier

- **Prior Probability:** The prior probability for each class (positive and negative) was calculated based on the frequency of reviews in the training set.
- **Likelihood Estimation:** The likelihood of each word appearing in positive or negative reviews was estimated using Laplace smoothing, ensuring that even words absent in the training data received a small probability.
- **Classification:** For each test review, posterior probabilities for both classes were calculated, and the class with the higher probability was chosen as the predicted sentiment.

## 3. Evaluation

The model was evaluated using accuracy, precision, recall, and F1-score. Additionally, a confusion matrix was generated to observe the classifier's performance in distinguishing between positive and negative reviews.

- **Confusion Matrix:**

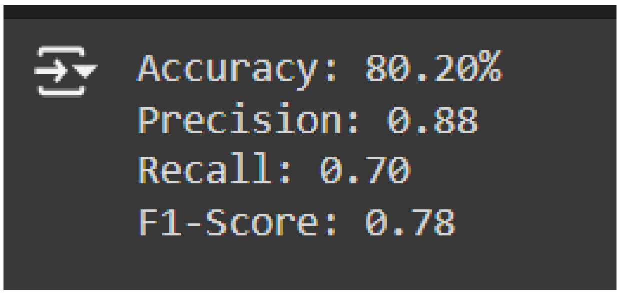


```
Confusion Matrix:
[[91 10]
 [30 71]]
```

**[[True Negative, False Positive], [False Negative, True Positive]]**

[[91 10], [30 71]]

- **Performance Metrics:**



```
Accuracy: 80.20%
Precision: 0.88
Recall: 0.70
F1-Score: 0.78
```

- **Accuracy:** 80.20%
- **Precision:** 0.88
- **Recall:** 0.70
- **F1-Score:** 0.78

### 4. Challenges

1. **Text Preprocessing:** Removing stopwords and punctuation was straightforward, but ensuring the model generalizes well was challenging due to the small vocabulary size.
2. **Laplace Smoothing:** Handling words that didn't appear in the training data required careful application of Laplace smoothing to avoid zero probabilities.
3. **Feature Selection:** Building a representative vocabulary while balancing performance was crucial.

### 5. How can we Improve?

1. **N-Grams:** Incorporating bi-grams or tri-grams could help capture context better, especially for negations (e.g. not good).
2. **Text Preprocessing:** Advanced techniques such as lemmatization or stemming could improve the generalization by reducing word variants (e.g. running vs. run).
3. **Dimensionality Reduction:** Using techniques like TF-IDF to prioritize more informative words may reduce noise and improve accuracy.
4. **Word Embeddings:** Using word embeddings (e.g., Word2Vec) could significantly improve the model's understanding of word semantics.

## Question 3: Implementing an ANN for Sentiment Classification [40].

In this task, I implemented an Artificial Neural Network (ANN) from scratch for a sentiment classification task. The goal was to classify reviews as positive or negative using one-hot encoded vectors as input. The dataset contained 5,000 training and 500 test samples.

### 1. Data Preprocessing

- **Text Cleaning:** The text was converted to lowercase, and all punctuation and special characters were removed.
- **Tokenization:** Each review was tokenized into individual words.
- **Vocabulary Building:** A vocabulary was built containing all unique words in the training data. Each word was assigned a unique index.
- **One-Hot Encoding:** Each tokenized review was converted into a one-hot encoded vector, with the length equal to the size of the vocabulary.

### 2. Artificial Neural Network Implementation

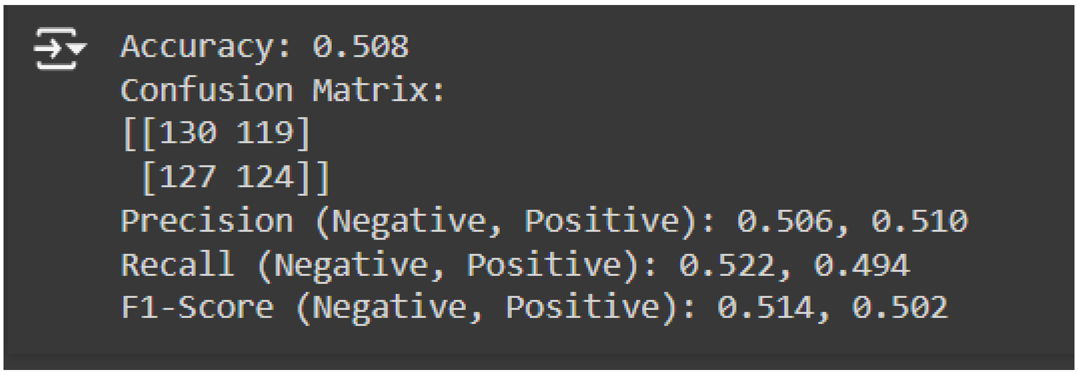
- **Input Layer:** The input size was set to the size of the vocabulary.

- **Hidden Layer:** A fully connected hidden layer with 128 neurons was implemented, using the ReLU activation function.
- **Output Layer:** A single neuron was used in the output layer, with a sigmoid activation function to classify the sentiment (0 for negative, 1 for positive).

## 3. Forward propagation, Back propagation and Training

- **Forward propagation:** The input (one-hot encoded vector) passes through the network layers. First, it's multiplied by the weights and biases in the hidden layer, followed by applying the ReLU activation function.
- **Back propagation:** The backpropagation algorithm was implemented to update weights and biases. Gradients were computed for both the hidden and output layers.
- **Training:** The model was trained on the 5,000 training samples over 20 epochs, with a learning rate of 0.01.

## 4. Results



```
Accuracy: 0.508
Confusion Matrix:
[[130 119]
 [127 124]]
Precision (Negative, Positive): 0.506, 0.510
Recall (Negative, Positive): 0.522, 0.494
F1-Score (Negative, Positive): 0.514, 0.502
```

- **Accuracy:** The model achieved an accuracy of **0.51 or 51%** on the test set.
- **Confusion Matrix: [[True Negative, False Positive], [False Negative, True Positive]]** = **[[130 119] [127 124]]**
- **Precision (Negative, Positive): 0.506, 0.510**
- **Recall (Negative, Positive): 0.522, 0.494**
- **F1-Score (Negative, Positive): 0.514, 0.502**

## 5. Challenges Encountered

- **Efficient Vocabulary Building:** While managing the vocabulary size for the one-hot encoding process was time consuming.
- **Weight Initialization:** Proper initialization of weights was crucial to avoid exploding gradients.
- **Training Time:** Training the ANN without using machine learning libraries led to increased computation time, particularly with a large vocabulary size (300).

### 6. Potential Improvements

- **More Hidden Layers:** Introducing additional hidden layers could improve the model's ability to learn complex patterns.
- **Batch Normalization:** When I was researching, this is one of the methods that could be implemented to improve the model's stability and convergence.
- **Different Activation Functions:** Using activation functions like eLU & Tanh could improve learning by addressing issues with ReLU, such as dying neurons.

## References

1) Natural Language Processing Book
2) https://www.geeksforgeeks.org/implementing-ann-training-process-in-python/
3) https://www.kaggle.com/code/prashant111/naive-bayes-classifier-in-python#1.-Introduction-to-Naive-Bayes-algorithm-