A close-up of a logo

Description automatically generated 

**INOFRMATICS INSTITUTE OF TECHNOLOGY**

**In collaboration with**

ROBERT GORDEN UNIVERSITY ABERDEEN

**Department of Computing**

|  |  |  |
| --- | --- | --- |
| Name | IIT NO | RGU NO |
| Oshan Rathnayake | **20211079** | **2119146** |
| Sandushke De Alwis | **20200244** | **2117852** |
| Nadun Senarathne | **20210488** | **2117538** |

**Module: CM3604 Deep Learning**

**Module Leader: Mr. Prasan Yapa**

**Handout Date: 30th of September 2023**

**Submission Date: 2nd of December 2023**

**Acknowledge**

We would like to take this opportunity to thank all the IIT staff for giving us the opportunity to follow the degree Artificial Intelligence & Data Science. We specially thank our lecturer and tutor Mr. Prasan Yapa for giving us the knowledge to do this report. Rather than the academic knowledge we got a huge support and guidance while researching and finding information to complete this project. At last, we would like to take the opportunity to thank for all the feedback that you provided us. Thank you.

Table of Contents

[**Solution Methodology** 4](#_Toc152321911)

[**Evaluation Criteria** 14](#_Toc152321912)

[**Model Evaluation** 16](#_Toc152321913)

[**Experimental Results** 18](#_Toc152321914)

[**Limitations & Future enhancements** 19](#_Toc152321915)

[**GitHub Repository URL** 19](#_Toc152321916)

[**Appendix** 20](#_Toc152321917)

# **Solution Methodology**

The below section explains the different models that were used and how it was used to perform the sentiment analysis using the yelp review dataset.

BERT – ‘BERT’ model was chosen as the supervised model to perform sentiment analysis using the yelp review dataset. This model has the capability of examining the whole context of a given sentence. Therefore, it was quite effective to use as the supervised model to perform the sentimental analysis task customer reviews. BERT model identifies the complex connections between the words and their meanings by improving its pretrained contextual embeddings.

XLNet – XLNet is an unsupervised model which expands the capabilities of the BERT’s pre-trained model. This model has the capability to learn and understand the structure of a context by just predicting the missing words in in particular space. Once the model is pre-trained, it can be fine-tuned for purposes such as sentiment analysis which can even be a strong tool in unsupervised learning applications.

**BERT:**

Out of the whole dataset, 10,000 data were taken from the star rating between 1.0-2.0, 10,000 data from the star rating of 3.0 and another 10,000 data from star the rating between 4.0-5.0. The total of 30,000 data were then merged together to perform the whole sentiment analysis task.

yelp\_data = pd.concat([yelp\_data\_0, yelp\_data\_1, yelp\_data\_2])

# Resetting the index of the combined DataFrame

yelp\_data.reset\_index(drop=True, inplace=True)

# Shuffle the combined DataFrame

yelp\_data = yelp\_data.sample(frac=1).reset\_index(drop=True)

# Display the first few rows of the shuffled DataFrame

yelp\_data.head()

As the next step, after shuffling all the data which we had, we checked if there were any data duplications, missing values etc.

# Check for duplicates

duplicates = yelp\_data.duplicated()

print("Total duplicate rows:", duplicates.sum())

# Check for missing values

missing\_values = yelp\_data.isnull().sum()

print('\n',missing\_values)

Next, to conduct the feature selection task, we selected the features ‘text’ and ‘stars’ from the dataset. Also, using the lambda mapping method we created a new feature called ‘sentiment’ where it maps 1.0-2.0-star rating as 0, star rating 3 as 1 and star rating between 4.0-5.0 as 2. These 0,1 and 2 values were assigned as negative, neutral, and positive sentiments respectively.

# Select the necessary features

yelp\_data = yelp\_data[['text', 'stars']]

# Map star ratings to sentiment labels (0 for 1-2 stars, 1 for 3 stars, 2 for 4-5 stars)

yelp\_data['sentiment'] = yelp\_data['stars'].apply(lambda x: 0 if x <= 2 else (1 if x == 3 else 2))

As the next step the data splitting part was done where 80% of data were divided as training data, 20% data as testing data. Out of the 20% of test data, 10% of data were divided as validation data and the remaining 10% as the test data.

# Split data into train, validation, and test sets

train\_data, test\_data = train\_test\_split(yelp\_data, test\_size=0.2, random\_state=42)

val\_data, test\_data = train\_test\_split(test\_data, test\_size=0.5, random\_state=42)

After this task was done, we load the pre-trained model which is BERT, and the number of labels were assigned as 3 since we are predicting 3 labels. In the model architecture, a dropout layer was used to prevent the model being overfitting.

dropout\_prob = 0.15

model.dropout = torch.nn.Dropout(dropout\_prob)

The review and sentiment data in the dataset were obtained using an iterative loop and they were being then tokenized using the tokenizer.encode\_plus method. The tokenized data were stored in input\_ids and the attention mask stored all the important featured that were obtained when tokenizing the data. Moreover, the labels too were obtained, and were stored as a list called labels. The three lists ‘inputs\_ids’, ‘attention masks’ and ‘labels’ were next converted into pytorch tensors. The goal of converting these data into pytorch tensors were to use it when training and testing the model.

def tokenize\_data(yelp\_data, max\_length=128):

    input\_ids = []

    attention\_masks = []

    labels = []

    for index, row in yelp\_data.iterrows():

        review = row['text']

        label = row['sentiment']

        try:

            encoded\_data = tokenizer.encode\_plus(

                review,

                add\_special\_tokens=True,

                max\_length=max\_length,

                padding='max\_length',

                return\_attention\_mask=True,

                return\_tensors='pt',

                truncation=True

            )

            input\_ids.append(encoded\_data['input\_ids'])

            attention\_masks.append(encoded\_data['attention\_mask'])

            labels.append(label)

        except Exception as e:

            # Handle tokenization errors

            print(f"Error tokenizing the following text: {review}")

            print(f"Error details: {e}")

    if not input\_ids or not attention\_masks or not labels:

        raise ValueError("No valid data after tokenization. Check your input.")

    input\_ids = torch.cat(input\_ids, dim=0)

    attention\_masks = torch.cat(attention\_masks, dim=0)

    labels = torch.tensor(labels)

    return input\_ids, attention\_masks, labels

# Usage

train\_input\_ids, train\_attention\_masks, train\_labels = tokenize\_data(train\_data)

val\_input\_ids, val\_attention\_masks, val\_labels = tokenize\_data(val\_data)

test\_input\_ids, test\_attention\_masks, test\_labels = tokenize\_data(test\_data)

batch\_size = 32

train\_dataset = TensorDataset(train\_input\_ids, train\_attention\_masks, train\_labels)

train\_sampler = RandomSampler(train\_dataset)

train\_dataloader = DataLoader(train\_dataset, sampler=train\_sampler, batch\_size=batch\_size)

val\_dataset = TensorDataset(val\_input\_ids, val\_attention\_masks, val\_labels)

val\_sampler = SequentialSampler(val\_dataset)

val\_dataloader = DataLoader(val\_dataset, sampler=val\_sampler, batch\_size=batch\_size)

test\_dataset = TensorDataset(test\_input\_ids, test\_attention\_masks, test\_labels)

test\_sampler = SequentialSampler(test\_dataset)

test\_dataloader = DataLoader(test\_dataset, sampler=test\_sampler, batch\_size=batch\_size)

As the optimizer we used the ‘AdamW’ by giving a default value for learning rate.

# Check if M1 GPU is available; if not, use CPU

device = torch.device("mps" if torch.backends.mps.is\_available() else "cpu")

# device = torch.device("cpu")

print(f"Using device: {device}")

# Define optimizer and loss function

optimizer = AdamW(model.parameters(), lr=2e-5, eps=1e-8)

# Move the model to the device

model.to(device)

After these steps were conducted, we loaded the fine-tuned bert-base model using the data we wanted. The model training was done as the next step where it sent the data in batchwise (32 mini batches). Techniques such gradient loss and backpropagation were applied in the training loop to observe how the model behaves and to observe the final output of the model. The average loss and training loss were also calculated to observe the trend of the results. From the model.eval it will load the validated data. Through attention\_mask and labels we check if the obtained results which we trained using training data were validated properly as we predicted. We used an early stopping to stop the model without leading it to get overfitted.

train\_losses = []

val\_losses = []

train\_accuracies = []

val\_accuracies = []

num\_epochs = 4

# Early stopping parameters

early\_stopping\_patience = 3 # Number of epochs to wait after last time validation loss improved.

best\_val\_loss = float('inf') # Initialize the best validation loss as infinity.

early\_stopping\_counter = 0 # Counter for how many epochs without improvement.

for epoch in range(num\_epochs):

model.train()

total\_loss = 0

correct = 0

total = 0

for batch in tqdm(train\_dataloader, desc=f'Epoch {epoch + 1}'):

batch = tuple(t.to(device) for t in batch)

input\_ids, attention\_mask, labels = batch

optimizer.zero\_grad()

outputs = model(input\_ids, attention\_mask=attention\_mask, labels=labels)

loss = outputs.loss

total\_loss += loss.item()

loss.backward()

optimizer.step()

\_, predicted = torch.max(outputs.logits, 1)

correct += (predicted == labels).sum().item()

total += labels.size(0)

average\_loss = total\_loss / len(train\_dataloader)

train\_accuracy = correct / total

train\_losses.append(average\_loss)

train\_accuracies.append(train\_accuracy)

# Validation

model.eval()

val\_loss = 0

val\_correct = 0

val\_total = 0

for batch in tqdm(val\_dataloader, desc='Validation'):

batch = tuple(t.to(device) for t in batch)

input\_ids, attention\_mask, labels = batch

with torch.no\_grad():

outputs = model(input\_ids, attention\_mask=attention\_mask, labels=labels)

if isinstance(outputs, tuple):

loss = outputs[0]

elif isinstance(outputs, dict):

loss = outputs['loss']

else:

raise ValueError("Loss not found in model outputs.")

val\_loss += loss.item()

\_, predicted = torch.max(outputs.logits, 1)

val\_correct += (predicted == labels).sum().item()

val\_total += labels.size(0)

val\_average\_loss = val\_loss / len(val\_dataloader)

val\_accuracy = val\_correct / val\_total

val\_losses.append(val\_average\_loss)

val\_accuracies.append(val\_accuracy)

print(f'Epoch {epoch + 1} - Train Loss: {average\_loss:.4f} - Val Loss: {val\_average\_loss:.4f} - Train Acc: {train\_accuracy:.4f} - Val Acc: {val\_accuracy:.4f}')

# Early stopping check

if val\_average\_loss < best\_val\_loss:

best\_val\_loss = val\_average\_loss

early\_stopping\_counter = 0

else:

early\_stopping\_counter += 1

if early\_stopping\_counter >= early\_stopping\_patience:

print(f'Early stopping triggered. Stopping at epoch {epoch + 1}.')

break

The final saved model will be then used for evaluation purposes such as to plot the confusion matrix graph, to obtain the classification report etc.

torch.save(model, 'bert\_model.pth')

model = torch.load('bert\_model.pth')

model.to(device)

**XLNet:**

For the XLNet model we took the same number of Dataset that we used for BERT, which is 30000. Since this is Unsupervised learning method, we took only the “text” feature from each Data and processed into preprocessing.in the preprocessing part firstly we checked if there is any duplication data and eliminated them. Then we checked if there were any missing.

yelp\_data = yelp\_data[['text']]

duplicates = yelp\_data.duplicated()

print("Total duplicate rows:", duplicates.sum())

# Check for missing values

missing\_values = yelp\_data.isnull().sum()

print('\n',missing\_values)

Then we load the XLNet prebuild model and tokenizer for tokenization.

tokenizer = XLNetTokenizer.from\_pretrained('xlnet-base-cased')

model = XLNetModel.from\_pretrained('xlnet-base-cased')

Then we used function to tokenize the ‘text’ data.

def tokenize\_data(yelp\_data):

    input\_ids = []

    attention\_masks = []

    for index, row in yelp\_data.iterrows():

        review = row['text']

        encoded\_data = tokenizer.encode\_plus(

            review,

            add\_special\_tokens=True,

            max\_length=128,

            padding='max\_length',

            return\_attention\_mask=True,

            return\_tensors='pt',

            truncation=True

        )

        input\_ids.append(encoded\_data['input\_ids'])

        attention\_masks.append(encoded\_data['attention\_mask'])

    input\_ids = torch.cat(input\_ids, dim=0)

    attention\_masks = torch.cat(attention\_masks, dim=0)

    return input\_ids, attention\_masks

train\_input\_ids, train\_attention\_masks = tokenize\_data(yelp\_data)

batch\_size = 32

train\_dataset = TensorDataset(train\_input\_ids, train\_attention\_masks)

train\_sampler = RandomSampler(train\_dataset)

train\_dataloader = DataLoader(train\_dataset, sampler=train\_sampler, batch\_size=batch\_size)

We import relevant libraries during the model training process, such as TQDM for progress tracking. We limit the number of epochs to ten and define early termination criteria, initializing the best loss to positive infinity and tracking epochs that do not improve. then We enter the training loop by putting the model in training mode and iterating over batches with the training data loader. Gradients are wiped out, and model outputs are obtained, with the loss (in this case, the mean of the last hidden state) calculated. We collect the loss and use backpropagation to update gradients via optimization.

After finishing each epoch, we compute and display the average loss. By comparing the current average loss to the best loss, we look for early halting. If the current loss is less than the best loss, we update the best loss and reset the number of epochs without improvement. Otherwise, the count is increased.

Then we save the model, and We use torch.load() to load the saved model. We use the to(device) method to ensure that the model is on the correct device (GPU). This phase is essential for maintaining consistency between the training and inference settings.

import numpy as np

from tqdm import tqdm

# Model training

num\_epochs = 10  #

early\_stopping\_patience = 3

best\_loss = float('inf')

epochs\_without\_improvement = 0

for epoch in range(num\_epochs):

    model.train()

    total\_loss = 0

    for batch in tqdm(train\_dataloader, desc=f'Epoch {epoch + 1}'):

        batch = tuple(t.to(device) for t in batch)

        input\_ids, attention\_mask = batch

        optimizer.zero\_grad()

        outputs = model(input\_ids, attention\_mask=attention\_mask)

        loss = outputs.last\_hidden\_state.mean()  # You can use a different reconstruction loss here

        total\_loss += loss.item()

        loss.backward()

        optimizer.step()

    average\_loss = total\_loss / len(train\_dataloader)

    print(f'Epoch {epoch + 1} - Average Loss: {average\_loss:.4f}')

    # Early Stopping Check

    if average\_loss < best\_loss:

        best\_loss = average\_loss

        epochs\_without\_improvement = 0

    else:

        epochs\_without\_improvement += 1

    if epochs\_without\_improvement >= early\_stopping\_patience:

        print(f'Early stopping after {epoch + 1} epochs without improvement.')

        break

Following the generation of embeddings from the trained model, we proceed to the clustering phase, which employs the scikit-learn library's KMeans method. To begin, we switch the model to evaluation mode and establish an empty list, 'embeddings,' in which to store the feature vectors.

We use the learned model to cycle through batches of the training dataset in a no-grad scenario. After converting it to a NumPy array, we retrieve the input\_ids and attention\_mask from each batch, collect model outputs, and append the mean of the latest concealed state to the 'embeddings' list.

We concatenate all the embeddings along the specified axis to generate a thorough feature matrix.

Moving on to the clustering evaluation, we set the maximum number of clusters (max\_clusters) to 6, though this can be changed based on the needs. We apply KMeans clustering on the embeddings and iterate over potential cluster numbers. For each cluster arrangement, silhouette scores, a measure of cluster quality, are computed and saved.

Finally, we use Matplotlib to display the silhouette scores by plotting them against the number of clusters. The resulting figure aids in choosing the best number of clusters for the given data, with a higher silhouette score suggesting more well-defined clusters. Adjustments to the 'max\_clusters' option can be made to investigate various clustering scenarios depending on specific requirements.

from sklearn.cluster import KMeans

from sklearn.metrics import silhouette\_score

import matplotlib.pyplot as plt

model.eval()

embeddings = []

with torch.no\_grad():

    for batch in tqdm(train\_dataloader, desc='Generating Embeddings'):

        batch = tuple(t.to(device) for t in batch)

        input\_ids, attention\_mask = batch

        outputs = model(input\_ids, attention\_mask=attention\_mask)

        embeddings.append(outputs.last\_hidden\_state.mean(dim=1).cpu().numpy())

embeddings = np.concatenate(embeddings, axis=0)

# Evaluate Clustering for Different Numbers of Clusters

max\_clusters = 6 # You can adjust this based on your requirements

silhouette\_scores = []

for num\_clusters in range(2, max\_clusters + 1):

    kmeans = KMeans(n\_clusters=num\_clusters, random\_state=42)

    cluster\_labels = kmeans.fit\_predict(embeddings)

    silhouette\_avg = silhouette\_score(embeddings, cluster\_labels)

    silhouette\_scores.append(silhouette\_avg)

# Plot Silhouette Scores

plt.plot(range(2, max\_clusters + 1), silhouette\_scores, marker='o')

plt.xlabel('Number of Clusters')

plt.ylabel('Silhouette Score')

plt.title('Silhouette Score vs. Number of Clusters')

plt.show()

Based on the silhouette we went with 3 clusters

A graph with a line

Description automatically generated

Then we use the scikit-learn t-SNE algorithm to reduce the dimensionality of the embeddings to two dimensions, allowing for easier display. The generated 2D embeddings are saved in the variable 'embeddings\_2d'.

We use Matplotlib and Seaborn to visually evaluate the clustering result. The t-SNE-transformed embeddings are shown in a scatter plot, with points colored according to the cluster labels supplied to them. For clarity, the 'viridis' color palette is used, and a legend is given to identify different clusters.

The resulting picture, dubbed 't-SNE picture of Clusters,' sheds light on the spatial distribution of data points in a reduced-dimensional environment. Patterns and separations among clusters emerge, assisting in the qualitative evaluation of the clustering algorithm's efficacy.

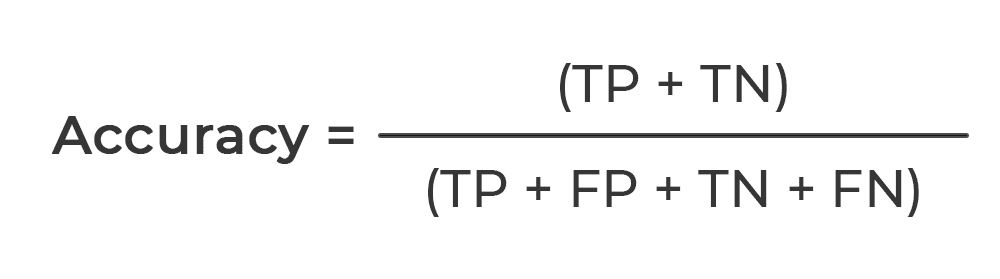
The models BERT and XLNet were used to perform sentiment analysis for the yelp review dataset. As the supervised model, BERT was used to leverage the understanding of complex contexts within the reviews. In addition to the supervised model, XLNet was chosen as the unsupervised model to improve the understanding of sequential dependencies of review data. Each of these models were selected based on their strengths, which could help to identify the overall sentiment.

# **Evaluation Criteria**

Evaluation criteria uses critical tools to observe the performance and effectiveness of models in variety of fields. These metrices helps to assess the model’s ability to achieve a specified task. Selecting the appropriate evaluation metrices depends on the characteristics of the dataset and the nature of problem that was chosen. Below are some of the evaluation metrices that were selected to capture and predict text data in sentiment analysis.

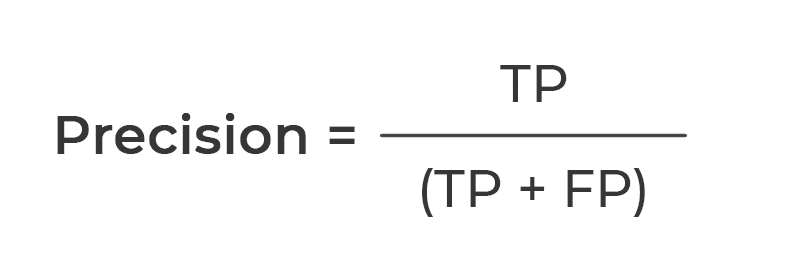
**Accuracy :**

* Definition: Accuracy simply means the ratio of correctly predicting the total instances. It gives the overall evaluation of the model’s accuracy.
* How it is being applied: The proportion of correctly identified positive and negative sentiments are measured by the accuracy.



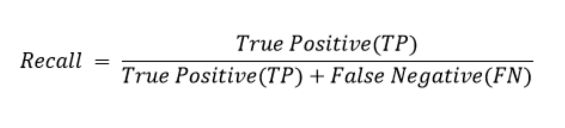
**Precision:**

* Definition: Precision is the ratio of correctly predicted positive instances out of the total predicted positives. Its goal is to find how many positive predictions are actually positive.
* How it has being applied: According to the sentiment analysis task, precision is the fraction which accurately predicts the detected positive reviews out of all the reviews that are expected to be positive.



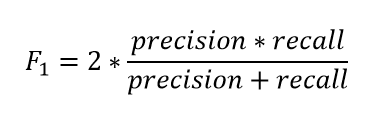
**Recall:**

* Definition: Recall is defined as the ratio of correctly predicted positive observations to all the actual positives. Its goal is to identify how many actual positive instances were identified by the model.
* How it has being applied: According to the sentiment analysis scenario, recall gives the percentage of accurately detected positive reviews out of all actual reviews.

****

**F1-Score:**

* Definition: F1 score is the weighted average of precision and recall which is in the range of between 0 and 1. It gives a balance between precision and recall.
* How it has being applied: According to the sentiment analysis scenario, F1 score considers the accuracy of both positive predictions of precision and recall.



Evaluation for XLNet model was done using following Criteria:

Inertia

Inertia refers to the total within-cluster variance, also known as the within-cluster sum of squares. It is frequently used as a statistic to assess the performance of clustering algorithms like KMeans.

In our case Inertia score is 15.5207 which means the clusters are more compact, which means that the data points inside each cluster are closer to the centroid. This is often beneficial in clustering because it suggests well-defined and distinct data categories.

# **Model Evaluation**

**BERT Model:**

Train – 80 %,

Test – 10%

Validation - 10%

**Classification Report (BERT)**

A screenshot of a computer screen

Description automatically generated

**Confusion Matrix (BERT)**

A blue squares with white text

Description automatically generated

**Bar chart (Predicted Sentiment Distribution) - (BERT)**

A graph of negative and negative

Description automatically generated

**XLNet Model (3D model)**

A graph with a colorful sphere

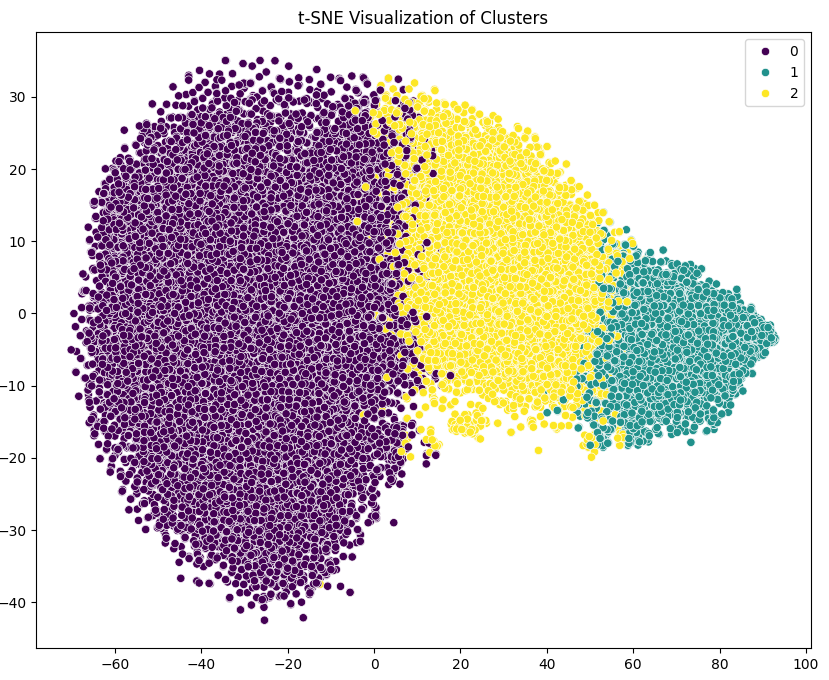
Description automatically generated with medium confidence

# **Experimental Results**

Following graphs shows the result after evaluating result for BERT model.



Following graphs shows the result for XLNet after evaluating result



# **Limitations & Future enhancements**

Limitations:

* The biggest challenge we faced as team was it was a very tedious task to run the deep learning models since it requires high computing power. This took hours to run even one epoch value (approx.: 2 hrs. for one epoch) due to the low processing power.
* Some sentiment reviews contained sensitive, complicated thoughts which were difficult to capture properly.

Ways to overcome the limitations:

* Do more hyperparameter tunning to find out the most optimal solution.
* Do more experiments on transformers or ensemble models to capture the complex sentimental reviews.

Future enhancements:

* Trying out multiple testings with other models which belong to supervised, unsupervised, semi-supervised and reinforcement categories and observe and compare the results that were obtained with other models that were done before.
* Trying out with different other pre-trained models or embeddings to obtain more benefits when working with larger datasets.

# **GitHub Repository URL**

<https://github.com/Sandushke/Deep-Learning-Coursework>

# **Appendix**

**BERT Model:**

import pandas as pd

import numpy as np

import json

from sklearn.model\_selection import train\_test\_split, StratifiedKFold

from sklearn.metrics import accuracy\_score, classification\_report, confusion\_matrix

from transformers import BertTokenizer, BertForSequenceClassification, AdamW, get\_linear\_schedule\_with\_warmup, BertConfig

from torch.utils.data import TensorDataset, DataLoader, RandomSampler, SequentialSampler

from tqdm import tqdm

import matplotlib.pyplot as plt

import seaborn as sns

from wordcloud import WordCloud

import torch

# Function to get the total number of lines in the file

def get\_total\_lines(filename):

    with open(filename, 'r', encoding='utf-8') as file:

        return sum(1 for \_ in file)

# Function to load rows where 'stars' feature is 1.0 or 2.0

def load\_selected\_data\_0(filename):

    data = []

    count = 0

    with open(filename, 'r', encoding='utf-8') as file:

        for line in file:

            row = json.loads(line)

            if row.get('stars') in [1.0, 2.0]:  # Check for 1.0 or 2.0 stars

                data.append(row)

                count += 1

                if count == 10000:  # Stop after collecting 10000 rows

                    break

    return data

# Function to load rows where 'stars' feature is 3.0

def load\_selected\_data\_1(filename):

    data = []

    count = 0

    with open(filename, 'r', encoding='utf-8') as file:

        for line in file:

            row = json.loads(line)

            if row.get('stars') == 3.0:

                data.append(row)

                count += 1

                if count == 10000:  # Stop after collecting 10000 rows

                    break

    return data

# Function to load rows where 'stars' feature is 4.0 or 5.0

def load\_selected\_data\_2(filename):

    data = []

    count = 0

    with open(filename, 'r', encoding='utf-8') as file:

        for line in file:

            row = json.loads(line)

            if row.get('stars') in [4.0, 5.0]:  # Check for 4.0 or 5.0 stars

                data.append(row)

                count += 1

                if count == 10000:  # Stop after collecting 10000 rows

                    break

    return data

# File path

file\_path = 'yelp\_dataset/yelp\_academic\_dataset\_review.json'

# Load the data

yelp\_data\_0 = pd.DataFrame(load\_selected\_data\_0(file\_path))

yelp\_data\_1 = pd.DataFrame(load\_selected\_data\_1(file\_path))

yelp\_data\_2 = pd.DataFrame(load\_selected\_data\_2(file\_path))

yelp\_data\_0.head()

yelp\_data\_1.head()

yelp\_data\_2.head()

yelp\_data = pd.concat([yelp\_data\_0, yelp\_data\_1, yelp\_data\_2])

# Resetting the index of the combined DataFrame

yelp\_data.reset\_index(drop=True, inplace=True)

# Shuffle the combined DataFrame

yelp\_data = yelp\_data.sample(frac=1).reset\_index(drop=True)

# Display the first few rows of the shuffled DataFrame

yelp\_data.head()

# Printing the shape

print("DataFrame Shape:", yelp\_data.shape)

# Check for duplicates

duplicates = yelp\_data.duplicated()

print("Total duplicate rows:", duplicates.sum())

# Check for missing values

missing\_values = yelp\_data.isnull().sum()

print('\n',missing\_values)

# Select the necessary features

yelp\_data = yelp\_data[['text', 'stars']]

# Map star ratings to sentiment labels (e.g., 0 for 1-2 stars, 1 for 3 stars, 2 for 4-5 stars)

yelp\_data['sentiment'] = yelp\_data['stars'].apply(lambda x: 0 if x <= 2 else (1 if x == 3 else 2))

yelp\_data.head()

# Printing the shape

print("DataFrame Shape:", yelp\_data.shape)

# Split data into train, validation, and test sets

train\_data, test\_data = train\_test\_split(yelp\_data, test\_size=0.2, random\_state=42)

val\_data, test\_data = train\_test\_split(test\_data, test\_size=0.5, random\_state=42)

# Load pre-trained BERT tokenizer and model

tokenizer = BertTokenizer.from\_pretrained('bert-base-uncased')

model = BertForSequenceClassification.from\_pretrained('bert-base-uncased', num\_labels=3)

# # Define a new configuration with reduced complexity

# reduced\_config = BertConfig.from\_pretrained('bert-base-uncased', num\_labels=3, num\_hidden\_layers=6, hidden\_size=768)

# # Load the model with the reduced configuration

# model = BertForSequenceClassification(reduced\_config)

dropout\_prob = 0.15  # You can adjust the dropout probability

model.dropout = torch.nn.Dropout(dropout\_prob)

def tokenize\_data(yelp\_data, max\_length=128):

    input\_ids = []

    attention\_masks = []

    labels = []

    for index, row in yelp\_data.iterrows():

        review = row['text']

        label = row['sentiment']

        try:

            encoded\_data = tokenizer.encode\_plus(

                review,

                add\_special\_tokens=True,

                max\_length=max\_length,

                padding='max\_length',

                return\_attention\_mask=True,

                return\_tensors='pt',

                truncation=True

            )

            input\_ids.append(encoded\_data['input\_ids'])

            attention\_masks.append(encoded\_data['attention\_mask'])

            labels.append(label)

        except Exception as e:

            # Handle tokenization errors (e.g., if a text is too long)

            print(f"Error tokenizing the following text: {review}")

            print(f"Error details: {e}")

    if not input\_ids or not attention\_masks or not labels:

        raise ValueError("No valid data after tokenization. Check your input.")

    input\_ids = torch.cat(input\_ids, dim=0)

    attention\_masks = torch.cat(attention\_masks, dim=0)

    labels = torch.tensor(labels)

    return input\_ids, attention\_masks, labels

# Usage

train\_input\_ids, train\_attention\_masks, train\_labels = tokenize\_data(train\_data)

val\_input\_ids, val\_attention\_masks, val\_labels = tokenize\_data(val\_data)

test\_input\_ids, test\_attention\_masks, test\_labels = tokenize\_data(test\_data)

batch\_size = 32

train\_dataset = TensorDataset(train\_input\_ids, train\_attention\_masks, train\_labels)

train\_sampler = RandomSampler(train\_dataset)

train\_dataloader = DataLoader(train\_dataset, sampler=train\_sampler, batch\_size=batch\_size)

val\_dataset = TensorDataset(val\_input\_ids, val\_attention\_masks, val\_labels)

val\_sampler = SequentialSampler(val\_dataset)

val\_dataloader = DataLoader(val\_dataset, sampler=val\_sampler, batch\_size=batch\_size)

test\_dataset = TensorDataset(test\_input\_ids, test\_attention\_masks, test\_labels)

test\_sampler = SequentialSampler(test\_dataset)

test\_dataloader = DataLoader(test\_dataset, sampler=test\_sampler, batch\_size=batch\_size)

# Check if M1 GPU is available; if not, use CPU

device = torch.device("mps" if torch.backends.mps.is\_available() else "cpu")

# device = torch.device("cpu")

print(f"Using device: {device}")

# Define optimizer and loss function

optimizer = AdamW(model.parameters(), lr=2e-5, eps=1e-8)

# Move the model to the appropriate device

model.to(device)

train\_losses = []

val\_losses = []

train\_accuracies = []

val\_accuracies = []

num\_epochs = 4

# Early stopping parameters

early\_stopping\_patience = 3 # Number of epochs to wait after last time validation loss improved.

best\_val\_loss = float('inf') # Initialize the best validation loss as infinity.

early\_stopping\_counter = 0 # Counter for how many epochs without improvement.

for epoch in range(num\_epochs):

model.train()

total\_loss = 0

correct = 0

total = 0

for batch in tqdm(train\_dataloader, desc=f'Epoch {epoch + 1}'):

batch = tuple(t.to(device) for t in batch)

input\_ids, attention\_mask, labels = batch

optimizer.zero\_grad()

outputs = model(input\_ids, attention\_mask=attention\_mask, labels=labels)

loss = outputs.loss

total\_loss += loss.item()

loss.backward()

optimizer.step()

\_, predicted = torch.max(outputs.logits, 1)

correct += (predicted == labels).sum().item()

total += labels.size(0)

average\_loss = total\_loss / len(train\_dataloader)

train\_accuracy = correct / total

train\_losses.append(average\_loss)

train\_accuracies.append(train\_accuracy)

# Validation

model.eval()

val\_loss = 0

val\_correct = 0

val\_total = 0

for batch in tqdm(val\_dataloader, desc='Validation'):

batch = tuple(t.to(device) for t in batch)

input\_ids, attention\_mask, labels = batch

with torch.no\_grad():

outputs = model(input\_ids, attention\_mask=attention\_mask, labels=labels)

if isinstance(outputs, tuple):

loss = outputs[0]

elif isinstance(outputs, dict):

loss = outputs['loss']

else:

raise ValueError("Loss not found in model outputs.")

val\_loss += loss.item()

\_, predicted = torch.max(outputs.logits, 1)

val\_correct += (predicted == labels).sum().item()

val\_total += labels.size(0)

val\_average\_loss = val\_loss / len(val\_dataloader)

val\_accuracy = val\_correct / val\_total

val\_losses.append(val\_average\_loss)

val\_accuracies.append(val\_accuracy)

print(f'Epoch {epoch + 1} - Train Loss: {average\_loss:.4f} - Val Loss: {val\_average\_loss:.4f} - Train Acc: {train\_accuracy:.4f} - Val Acc: {val\_accuracy:.4f}')

# Early stopping check

if val\_average\_loss < best\_val\_loss:

best\_val\_loss = val\_average\_loss

early\_stopping\_counter = 0

else:

early\_stopping\_counter += 1

if early\_stopping\_counter >= early\_stopping\_patience:

print(f'Early stopping triggered. Stopping at epoch {epoch + 1}.')

break

torch.save(model, 'bert\_model.pth')

model = torch.load('bert\_model.pth')

model.to(device)

model.eval()

predictions = []

true\_labels = []

for batch in tqdm(val\_dataloader, desc='Validation'):

    batch = tuple(t.to(device) for t in batch)

    input\_ids, attention\_mask, labels = batch

    with torch.no\_grad():

        outputs = model(input\_ids, attention\_mask=attention\_mask)

    logits = outputs.logits

    predicted\_labels = torch.argmax(logits, dim=1).tolist()

    predictions.extend(predicted\_labels)

    true\_labels.extend(labels.tolist())

accuracy = accuracy\_score(true\_labels, predictions)

report = classification\_report(true\_labels, predictions)

print(f'Validation Accuracy: {accuracy:.4f}')

print(report)

# Create a confusion matrix

confusion\_mat = confusion\_matrix(true\_labels, predictions)

# Plot the confusion matrix

plt.figure(figsize=(8, 6))

sns.heatmap(confusion\_mat, annot=True, fmt='d', cmap='Blues')

plt.xlabel('Predicted Labels')

plt.ylabel('True Labels')

plt.title('Confusion Matrix')

plt.show()

# Count the predicted sentiment labels

predicted\_counts = [predictions.count(0), predictions.count(1), predictions.count(2)]

# Define sentiment labels

sentiment\_labels = ['Negative', 'Neutral', 'Positive']

# Create a bar chart

plt.figure(figsize=(8, 6))

plt.bar(sentiment\_labels, predicted\_counts, color=['red', 'gray', 'green'])

plt.xlabel('Sentiment')

plt.ylabel('Count')

plt.title('Predicted Sentiment Distribution')

plt.show()

def analyze\_sentiment(comment):

    # Tokenize and preprocess the comment

    inputs = tokenizer.encode\_plus(

        comment,

        add\_special\_tokens=True,

        max\_length=128,  # can adjust the maximum sequence length

        padding='max\_length',

        return\_attention\_mask=True,

        return\_tensors='pt',

        truncation=True

    )

    # Perform inference on GPU

    input\_ids = inputs['input\_ids'].to(device)

    attention\_mask = inputs['attention\_mask'].to(device)

    with torch.no\_grad():

        outputs = model(input\_ids, attention\_mask=attention\_mask)

    logits = outputs.logits

    predicted\_label = torch.argmax(logits, dim=1).item()

    # Define sentiment labels

    sentiment\_labels = {0: 'Negative', 1: 'Neutral', 2: 'Positive'}

    # Get the sentiment label and score

    sentiment\_label = sentiment\_labels[predicted\_label]

    sentiment\_score = torch.softmax(logits, dim=1)[0][predicted\_label].item()

    return sentiment\_label, sentiment\_score, input\_ids

comment = "A spacious restaurant with. A good mix of flavors. Price is average.The place is smelly as you are in the Carpark. Inside it is okay. The picture attached shows the palate which the burger served which is not so appealing. I saw they serve drinks from a container which is not a reusable plastic, which is not a hygienic option.Staff are not trained professionals but a set of youngsters who is not educated to serve in a restaurant."

sentiment\_label, sentiment\_score, logits = analyze\_sentiment(comment)

print(f"Sentiment: {sentiment\_label}")

print(f"Sentiment Score: {sentiment\_score:.4f}")

**XLNet Model:**

import pandas as pd

import numpy as np

import json

import random

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import accuracy\_score, classification\_report

from transformers import XLNetTokenizer, XLNetModel,XLNetForSequenceClassification,AdamW

from torch.utils.data import TensorDataset, DataLoader, RandomSampler, SequentialSampler

from tqdm import tqdm

import seaborn as sns

import torch

import matplotlib.pyplot as plt

# Function to get the total number of lines in the file

def get\_total\_lines(filename):

    with open(filename, 'r', encoding='utf-8') as file:

        return sum(1 for \_ in file)

# Function to load rows where 'stars' feature is 1.0 or 2.0

def load\_selected\_data\_0(filename):

    data = []

    count = 0

    with open(filename, 'r', encoding='utf-8') as file:

        for line in file:

            row = json.loads(line)

            if row.get('stars') in [1.0, 2.0]:  # Check for 1.0 or 2.0 stars

                data.append(row)

                count += 1

                if count == 10000:  # Stop after collecting 10000 rows

                    break

    return data

# Function to load rows where 'stars' feature is 3.0

def load\_selected\_data\_1(filename):

    data = []

    count = 0

    with open(filename, 'r', encoding='utf-8') as file:

        for line in file:

            row = json.loads(line)

            if row.get('stars') == 3.0:

                data.append(row)

                count += 1

                if count == 10000:  # Stop after collecting 10000 rows

                    break

    return data

# Function to load rows where 'stars' feature is 4.0 or 5.0

def load\_selected\_data\_2(filename):

    data = []

    count = 0

    with open(filename, 'r', encoding='utf-8') as file:

        for line in file:

            row = json.loads(line)

            if row.get('stars') in [4.0, 5.0]:  # Check for 4.0 or 5.0 stars

                data.append(row)

                count += 1

                if count == 10000:  # Stop after collecting 10000 rows

                    break

    return data

# File path

file\_path = 'yelp\_dataset/yelp\_academic\_dataset\_review.json'

# Load the data

yelp\_data\_0 = pd.DataFrame(load\_selected\_data\_0(file\_path))

yelp\_data\_1 = pd.DataFrame(load\_selected\_data\_1(file\_path))

yelp\_data\_2 = pd.DataFrame(load\_selected\_data\_2(file\_path))

yelp\_data\_0.head()

yelp\_data\_1.head()

yelp\_data\_2.head()

yelp\_data = pd.concat([yelp\_data\_0, yelp\_data\_1, yelp\_data\_2])

# Resetting the index of the combined DataFrame

yelp\_data.reset\_index(drop=True, inplace=True)

# Shuffle the combined DataFrame

yelp\_data = yelp\_data.sample(frac=1).reset\_index(drop=True)

# Display the first few rows of the shuffled DataFrame

yelp\_data.head()

# Printing the shape

print("DataFrame Shape:", yelp\_data.shape)

duplicates = yelp\_data.duplicated()

print("Total duplicate rows:", duplicates.sum())

# Check for missing values

missing\_values = yelp\_data.isnull().sum()

print('\n',missing\_values)

# Select the necessary features

yelp\_data = yelp\_data[['text']]

tokenizer = XLNetTokenizer.from\_pretrained('xlnet-base-cased')

model = XLNetModel.from\_pretrained('xlnet-base-cased')

dropout\_prob = 0.15  # You can adjust the dropout probability

model.dropout = torch.nn.Dropout(dropout\_prob)

model.to(device)

def tokenize\_data(yelp\_data):

    input\_ids = []

    attention\_masks = []

    for index, row in yelp\_data.iterrows():

        review = row['text']

        encoded\_data = tokenizer.encode\_plus(

            review,

            add\_special\_tokens=True,

            max\_length=128,

            padding='max\_length',

            return\_attention\_mask=True,

            return\_tensors='pt',

            truncation=True

        )

        input\_ids.append(encoded\_data['input\_ids'])

        attention\_masks.append(encoded\_data['attention\_mask'])

    input\_ids = torch.cat(input\_ids, dim=0)

    attention\_masks = torch.cat(attention\_masks, dim=0)

    return input\_ids, attention\_masks

train\_input\_ids, train\_attention\_masks = tokenize\_data(yelp\_data)

batch\_size = 32

train\_dataset = TensorDataset(train\_input\_ids, train\_attention\_masks)

train\_sampler = RandomSampler(train\_dataset)

train\_dataloader = DataLoader(train\_dataset, sampler=train\_sampler, batch\_size=batch\_size)

# Check if M1 GPU is available; if not, use CPU

device = torch.device("mps" if torch.backends.mps.is\_available() else "cpu")

# device = torch.device("cpu")

print(f"Using device: {device}")

# Define optimizer and loss function

optimizer = AdamW(model.parameters(), lr=2e-5, eps=1e-8)

# Move the model to the appropriate device

model.to(device)

model = torch.load('XLnet.pth')

model.to(device)

from sklearn.cluster import KMeans

from sklearn.metrics import silhouette\_score

import matplotlib.pyplot as plt

model.eval()

embeddings = []

with torch.no\_grad():

    for batch in tqdm(train\_dataloader, desc='Generating Embeddings'):

        batch = tuple(t.to(device) for t in batch)

        input\_ids, attention\_mask = batch

        outputs = model(input\_ids, attention\_mask=attention\_mask)

        embeddings.append(outputs.last\_hidden\_state.mean(dim=1).cpu().numpy())

embeddings = np.concatenate(embeddings, axis=0)

# Evaluate Clustering for Different Numbers of Clusters

max\_clusters = 6 # You can adjust this based on your requirements

silhouette\_scores = []

for num\_clusters in range(2, max\_clusters + 1):

    kmeans = KMeans(n\_clusters=num\_clusters, random\_state=42)

    cluster\_labels = kmeans.fit\_predict(embeddings)

    silhouette\_avg = silhouette\_score(embeddings, cluster\_labels)

    silhouette\_scores.append(silhouette\_avg)

# Plot Silhouette Scores

plt.plot(range(2, max\_clusters + 1), silhouette\_scores, marker='o')

plt.xlabel('Number of Clusters')

plt.ylabel('Silhouette Score')

plt.title('Silhouette Score vs. Number of Clusters')

plt.show()

# Apply Clustering (Example: K-Means)

num\_clusters = 3  # You can adjust this based on your requirements

kmeans = KMeans(n\_clusters=num\_clusters, random\_state=42)

cluster\_labels = kmeans.fit\_predict(embeddings)

# Evaluate Clustering (Example: Silhouette Score)

silhouette\_avg = silhouette\_score(embeddings, cluster\_labels)

print(f'Silhouette Score: {silhouette\_avg:.4f}')

from sklearn.manifold import TSNE

tsne = TSNE(n\_components=2, random\_state=42)

embeddings\_2d = tsne.fit\_transform(embeddings)

# Visualize Clusters

import matplotlib.pyplot as plt

import seaborn as sns

plt.figure(figsize=(10, 8))

sns.scatterplot(x=embeddings\_2d[:, 0], y=embeddings\_2d[:, 1], hue=cluster\_labels, palette='viridis', legend='full')

plt.title('t-SNE Visualization of Clusters')

plt.show()

inertia = kmeans.inertia\_

print(f'Inertia: {inertia:.4f}')

tsne\_3d = TSNE(n\_components=3, random\_state=42)

embeddings\_3d = tsne\_3d.fit\_transform(embeddings)

# Visualize Clusters in 3D

from mpl\_toolkits.mplot3d import Axes3D

fig = plt.figure(figsize=(12, 10))

ax = fig.add\_subplot(111, projection='3d')

scatter = ax.scatter(

    embeddings\_3d[:, 0],

    embeddings\_3d[:, 1],

    embeddings\_3d[:, 2],

    c=cluster\_labels,

    cmap='viridis'

)

ax.set\_title('t-SNE 3D Visualization of Clusters')

ax.set\_xlabel('Dimension 1')

ax.set\_ylabel('Dimension 2')

ax.set\_zlabel('Dimension 3')

# Add a colorbar

colorbar = plt.colorbar(scatter)

colorbar.set\_label('Cluster Labels')

plt.show()

def predict\_text(input\_text):

    model.eval()

    tokenizer = XLNetTokenizer.from\_pretrained('xlnet-base-cased')

    encoded\_data = tokenizer.encode\_plus(

        input\_text,

        add\_special\_tokens=True,

        max\_length=128,

        padding='max\_length',

        return\_attention\_mask=True,

        return\_tensors='pt',

        truncation=True

    )

    input\_ids = encoded\_data['input\_ids'].to(device)

    attention\_mask = encoded\_data['attention\_mask'].to(device)

    with torch.no\_grad():

        outputs = model(input\_ids, attention\_mask=attention\_mask)

    embeddings = outputs.last\_hidden\_state.mean(dim=1).cpu().numpy()

    cluster\_label = kmeans.predict(embeddings.reshape(1, -1))[0]

    return cluster\_label

# Example usage

new\_text = "this place is okay"

predicted\_cluster = predict\_text(new\_text)

print(f"Predicted Cluster: {predicted\_cluster}")