Real-Time Sentiment Analysis Pipeline

DEPI Graduation Project

Contents

Group Members:

- Mariem Osama Abdullah Elsayed Kandil
- Abdelrahman Ayman Eldegwy
- Mostafa Samir
- Sohaib Osama
- Muhamed Mahmoud Badwi
- Ahmed Medhat

Introduction:

This project aims to explore sentiment analysis using a variety of machine learning models, focusing on Natural Language Processing (NLP) techniques. By leveraging traditional machine learning algorithms such as Naive Bayes and Logistic Regression, and comparing them with modern embedding models such as BERT, GloVe, Word2Vec, and Doc2Vec, we aim to achieve high-performance sentiment classification for Amazon product reviews. The primary task is to classify reviews as positive or negative, while providing insights into feature importance and evaluating the performance of various techniques.

Project Overview

This project is designed to assess different approaches to sentiment analysis for text data. We start by downloading and cleaning an Amazon review dataset, followed by tokenization, stopword removal, and data balancing. Using both traditional machine learning models and modern embedding techniques, we extract features, train models, and evaluate their performance. The project compares TF-IDF, GloVe, BERT, and other embedding models. The final step includes building a simple MLP model for training the embeddings and testing their effectiveness.

Problem statement

Sentiment analysis is a widely-used application in natural language processing, aimed at determining the **emotional tone** behind a body of text. In this project, we aim to solve the problem of classifying reviews as positive or negative by comparing **traditional** and **modern NLP models**. With the rise of deep learning models and advanced embeddings, such as **BERT** and **GloVe**, it's essential to analyze their **efficiency and accuracy** in predicting sentiments when compared to traditional models like **Naive Bayes** and **Logistic Regression**. Our goal is to find the **best-performing approach** and provide detailed explanations on the **interpretability of the results**

Objectives

The primary objective of this project is to develop a comprehensive sentiment analysis system based on product reviews from the Amazon dataset. This system aims to classify reviews as either *Positive* or *Negative* by leveraging both traditional machine learning models and modern deep learning approaches. The project is divided into several key stages, each designed to enhance the accuracy, scalability, and deployment readiness of the sentiment classifier.

Key Objectives:

1. **Data Preprocessing and Cleaning**:
   - Implement advanced Natural Language Processing (NLP) techniques such as lemmatization, tokenization, and stopwords removal to clean and structure the dataset for better analysis.
   - Visualize the data using word clouds and class distribution charts to gain insights into the dataset's characteristics before and after cleaning.
2. **Data Resampling**:
   - Address class imbalance by resampling techniques to ensure the sentiment classes (Positive/Negative) are balanced, improving model performance.
3. **Feature Engineering**:
   - Extract meaningful features from the text using traditional techniques like **TF-IDF** and advanced methods like **BERT**, **GloVe**, **Word2Vec**, and **Doc2Vec** to better represent the data for model training.
4. **Model Building and Comparison**:
   - Build and compare multiple machine learning models such as **Naive Bayes**, **BernoulliNB**, and **Logistic Regression** to evaluate their effectiveness in classifying the sentiments.

  - Use various evaluation metrics (accuracy, precision, recall, and F1-score) to assess model performance and select the best-performing model.

5. **Deep Learning Integration**:
   - Introduce deep learning models like **BERT** for more nuanced text representation and improved sentiment classification accuracy.
   - Utilize modern tokenization techniques with BERT and implement multi-layer perceptron (MLP) models for embeddings generated by GloVe, Word2Vec, and Doc2Vec.

6. **Model Deployment Using Django**:
   - Integrate the best-performing model (traditional ML or deep learning) into a Django-based web application to make the model available as a service for users.
   - Deploy the trained model, enabling users to input text reviews via a web interface and receive real-time sentiment classification results.

7. **Explainability with SHAP**:
   - Implement **SHAP** (SHapley Additive exPlanations) to interpret model predictions and highlight the most significant features (words) that contribute to the positive or negative classification of reviews.

Project Outcome:

Upon completion, this project will deliver a **fully integrated sentiment analysis web application** powered by a machine learning model deployed through Django. The model will enable users to analyze sentiment in text reviews with high accuracy, and the system will provide feature importance insights for each prediction using SHAP visualizations. This enhances not only the usability but also the transparency and trustworthiness of the deployed system.

Background:

Introduction to the Models, Techniques, and Tools Used

In this project, a comprehensive approach was employed to predict sentiment in text reviews using various machine learning models, tokenization/vectorization techniques, and deep learning models. A combination of traditional machine learning, embedding methods, and neural network-based approaches (like BERT) was used to enhance model accuracy and gain insight into text data. Below is a detailed breakdown of each element involved in the project.

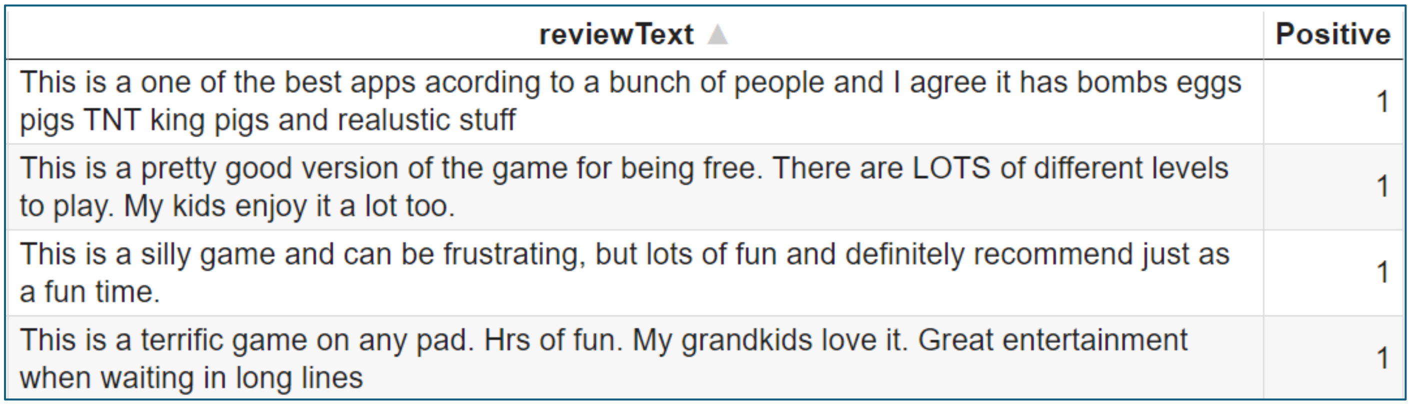
Dataset: Amazon Product Reviews

The dataset used for this project is the **Amazon Product Reviews Dataset** available on Kaggle here. It consists of user reviews from Amazon, including product details, review text, and the corresponding rating. the main feature being highlighted is the **reviewText** column, which contains textual reviews from users. These reviews express opinions about different products, likely from the **Mobile Apps** category based on the sample content.

Key Features:

1. **reviewText**: This column contains free-form text where users describe their experience with a product.
2. **Label**: The dataset seems to be labeled for **sentiment analysis** purposes, where **1** represents a **positive review** and **0** represents a **negative review**.
3. **Distribution of Sentiments**:
    - **Positive (0.95 - 1.00)**: There are **15,233** reviews that fall within this range, indicating very strong positive feedback.
    - **Negative (0.00 - 0.05)**: A total of **4,767** reviews fall into this negative sentiment range.

This distribution shows that the dataset is **imbalanced**, with far more positive reviews (around 76%) than negative ones (around 24%).



reviewText ▲	Positive
This is a one of the best apps acording to a bunch of people and I agree it has bombs eggs pigs TNT king pigs and realustic stuff	1
This is a pretty good version of the game for being free. There are LOTS of different levels to play. My kids enjoy it a lot too.	1
This is a silly game and can be frustrating, but lots of fun and definitely recommend just as a fun time.	1
This is a terrific game on any pad. Hrs of fun. My grandkids love it. Great entertainment when waiting in long lines	1

Figure 1. dataset sample

DistilBERT Base Model (Uncased)

DistilBERT is a **smaller, faster, and lighter** version of BERT (Bidirectional Encoder Representations from Transformers), designed to offer competitive performance while reducing the computational load.

Key Features of DistilBERT:

1. **Lightweight Architecture**:
   - **DistilBERT** is a **distilled version** of the original BERT model, which means it retains around **97% of the language understanding capabilities** of BERT while being **60% smaller** and **60% faster**. This makes it an ideal choice for deployment, especially for real-time applications like **sentiment analysis**.
2. **Uncased Model**:
   - The model is **uncased**, which means it **ignores case sensitivity** in input text. For example, "Apple" and "apple" are treated as the same word. This reduces complexity in processing, particularly useful when case doesn't significantly affect the meaning of the text, like in product reviews.
3. **Model Size**:
   - The model has **66 million parameters**, a considerable reduction compared to BERT's **110 million** parameters. This size reduction contributes to its faster training and inference times without drastically sacrificing accuracy.
4. **Training Method**:
   - DistilBERT was trained using a process called **knowledge distillation**, where the smaller model (DistilBERT) learns from a larger, pre-trained model (BERT), capturing most of its knowledge while cutting down unnecessary components.

How DistilBERT is Used in Sentiment Prediction:

1. **Sentiment Analysis**:

- In your use case for sentiment prediction, the model takes in text reviews from the dataset and predicts the sentiment label—either **positive** or **negative**.
- The **distilbert-base-uncased** model is pre-trained on a large corpus, which helps it understand the semantics of the text, making it highly suitable for **natural language understanding tasks** like classifying product reviews as positive or negative.

2. **Fine-Tuning**:
   - Although the base model is pre-trained, for your specific task, **fine-tuning** on your dataset is essential. Fine-tuning involves training the model on the **Amazon reviews dataset**, allowing it to better understand the patterns of sentiment within the product reviews.
3. **Transfer Learning**:
   - Because it's pre-trained on a large text corpus, **DistilBERT** leverages **transfer learning**, meaning it can adapt quickly to your sentiment analysis task without needing to start from scratch, reducing training time and computational costs.

Benefits of Using DistilBERT in Your Django Deployment:

- **Efficient for Real-Time Applications**: Due to its smaller size and faster inference speed, it is particularly well-suited for **real-time sentiment prediction** in web-based applications, such as your Django project.
- **Lower Hardware Requirements**: DistilBERT consumes fewer resources, making it feasible to deploy on systems with limited computational power, such as cloud environments or even edge devices.
- **Competitive Accuracy**: Despite its reduced size, DistilBERT achieves results that are close to BERT, maintaining a strong balance between performance and efficiency, which is key for a **scalable Django deployment**.

Libraries and Tools Overview

1. Libraries Used

1. **Pandas**: A powerful data manipulation and analysis library used for loading, cleaning, and organizing the dataset.

2. **NumPy**: Useful for efficient array computations and mathematical functions.

3. **Matplotlib & Seaborn**: Libraries for data visualization, helping to plot class distributions, model performance, and word clouds.
4. **Scikit-Learn**: Used for preprocessing, model training, and performance evaluation, including techniques like TF-IDF, Naive Bayes, Logistic Regression, and splitting datasets.
5. **Spacy**: A natural language processing (NLP) library used for tokenization, lemmatization, and stopword removal.
6. **WordCloud**: For generating a visual representation of frequently occurring words in the dataset.
7. **SHAP**: Used for interpreting model predictions and understanding feature importance.
8. **Gensim**: Used for word embeddings like Word2Vec and Doc2Vec.
9. **Torch & Torchtext**: Used for working with deep learning models, including transformers like BERT, and embedding techniques like GloVe.
10. **Hugging Face Transformers**: A state-of-the-art library for implementing transformers (e.g., BERT) in NLP tasks.
11. **Django**: A high-level Python web framework used for integrating and deploying machine learning models in web applications.

Vectorization Techniques

Vectorization converts text data into numerical features, which are essential for any machine learning model. Several vectorization techniques were employed:

1. TF-IDF (Term Frequency - Inverse Document Frequency)

- **Purpose**: TF-IDF converts text into a matrix of feature vectors based on word frequency, while also down-weighting frequently occurring words that are less informative.

- **Implementation**: Scikit-learn's TfidfVectorizer was used to generate features for training models like Naive Bayes and Logistic Regression.
- **Advantages**: Simple and effective for traditional machine learning algorithms, often yielding good baseline results.

2. BERT (Bidirectional Encoder Representations from Transformers)

- **Purpose**: BERT is a pre-trained transformer model that provides contextual embeddings by considering the whole sentence structure bidirectionally.
- **Implementation**: Hugging Face's distilbert-base-uncased was used to tokenize and classify the text data, and the data was split into input IDs and attention masks for model training.
- **Advantages**: BERT captures the context of words in relation to surrounding words, making it highly effective for understanding nuanced meanings in text data.

3. GloVe (Global Vectors for Word Representation)

- **Purpose**: GloVe is a pre-trained embedding that converts words into dense vector representations by factoring in co-occurrence statistics.
- **Implementation**: Torchtext's GloVe embeddings were used to vectorize the dataset based on word occurrences.
- **Advantages**: Fast and effective for capturing general semantic relationships between words.
- **4. Word2Vec**
- **Purpose**: Word2Vec is another word embedding technique that learns vector representations based on word context (Skip-gram/CBOW models).
- **Implementation**: Gensim's Word2Vec model was trained on the dataset to learn vector representations of words.
- **Advantages**: Efficient in learning word relationships and suitable for downstream tasks like classification.

5. Doc2Vec

- **Purpose**: Doc2Vec extends Word2Vec to document-level embedding, capturing semantic representations of entire documents (or reviews, in this case).
- **Implementation**: Gensim's Doc2Vec was used to learn vector representations of the entire review text.
- **Advantages**: Captures the overall semantic meaning of documents, useful for text classification tasks.

Models Used

Several machine learning models were used to predict sentiment from vectorized data:

1. Naive Bayes Classifier

- **Purpose**: Naive Bayes assumes independence between features and is often used for text classification due to its simplicity and effectiveness with TF-IDF vectors.
- **Implementation**: Scikit-learn's MultinomialNB and BernoulliNB were used to classify the text data.
- **Advantages**: Works well with high-dimensional data like TF-IDF and is computationally efficient.

2. Logistic Regression

- **Purpose**: A linear model that predicts the probability of binary outcomes and works well with text data when combined with TF-IDF.
- **Implementation**: Scikit-learn's LogisticRegression was used to predict sentiment based on TF-IDF features.
- **Advantages**: Easy to implement, interpretable, and provides solid performance on binary classification tasks.

3. MLP (Multilayer Perceptron)

- **Purpose**: A simple feed-forward neural network used for classification tasks on word embeddings.
- **Implementation**: The MLP was trained on the embeddings generated from GloVe, Word2Vec, and Doc2Vec.
- **Advantages**: Able to capture non-linear relationships between features and perform better on complex data compared to simpler models.

Data Resampling Techniques

1. Resampling (Balancing Classes)

- **Purpose**: Since the dataset was imbalanced, resampling was applied to ensure equal representation of both positive and negative sentiments in the training set.
- **Implementation**: Scikit-learn’s resample function was used to upsample the minority class (negative reviews) to match the number of instances in the majority class (positive reviews).
- **Advantages**: Prevents model bias towards the majority class, leading to more balanced predictions.

Django Framework

Integration and Deployment

- **Purpose**: Django was used to deploy the trained models as a web application, allowing for user interaction and real-time sentiment analysis.
- **Implementation**: After training, the machine learning models were integrated into a Django-based web application, where users can submit text and receive sentiment predictions.

Advantages: Django provides a robust, scalable framework for deploying machine learning models, making it easy to integrate front-end user interaction with back-end model inference.

Methodology:

In this project, we employed the **DistilBERT** base uncased model to perform sentiment analysis on the Amazon dataset, capitalizing on its efficiency and effectiveness in natural language processing tasks. Fine-tuning this pre-trained transformer model allowed us to tailor it specifically for our classification task, enhancing its accuracy and performance in identifying sentiment. Following the model training phase, we proceeded to deploy our fine-tuned **DistilBERT** model using the Django web framework. The deployment process involved several structured steps to ensure seamless integration between our machine learning model and user interaction through a web interface.

Step 1: Libraries and Data Downloading

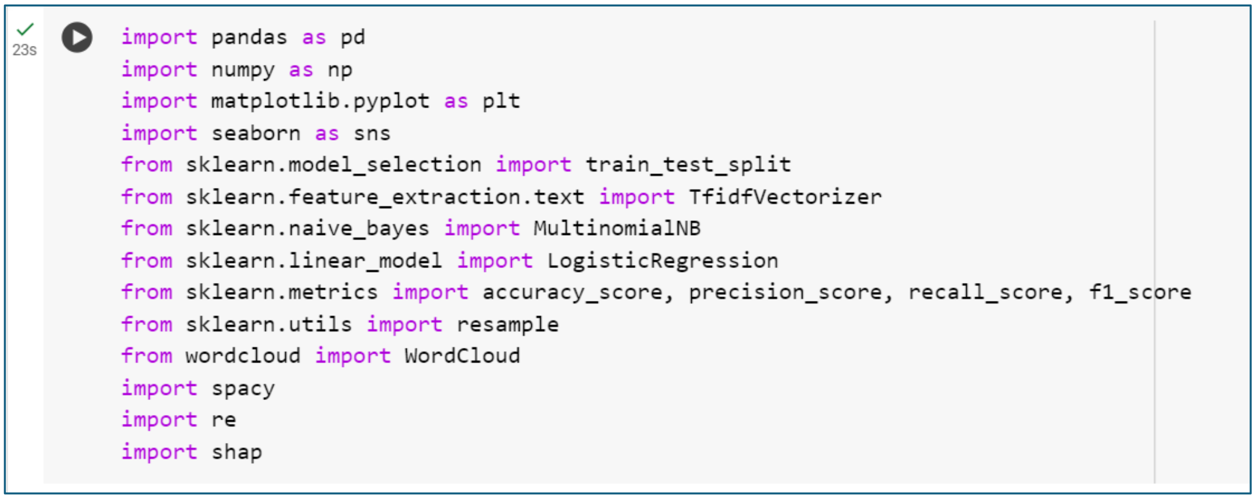
1. Imported Libaries:

In this step, essential libraries are imported, and necessary dependencies are installed for building the sentiment analysis model. Let's break down the key points:

- **Library Installation**:

- **shap**: A library used for interpreting machine learning models.
- **torch and torchtext**: Libraries essential for working with PyTorch, particularly for deep learning tasks and text processing.

- **Imports**:
    - `pandas, numpy:` For data manipulation and numeric computations.
    - `matplotlib, seaborn:` For visualization.
    - `train_test_split:` To split the dataset into training and testing subsets.
    - `TfidfVectorizer:` For converting text into numerical features using Term Frequency-Inverse Document Frequency (TF-IDF).
    - `MultinomialNB` and `LogisticRegression`: Two machine learning models used for sentiment prediction.
    - `accuracy_score, precision_score, recall_score, f1_score`: Performance metrics for evaluating the models.
    - `resample:` Used to balance the dataset.
    - `WordCloud, spacy, and re`: Libraries for Natural Language Processing



```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
from sklearn.utils import resample
from wordcloud import WordCloud
import spacy
import re
import shap
```

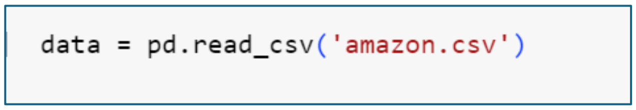
Figure 2. Imported Libaries

- **Data Download**:
    - The Amazon dataset is downloaded using gdown, which will be used to train and test sentiment prediction models.

Step 2: Data Loading, Initial Processing, and Visualization

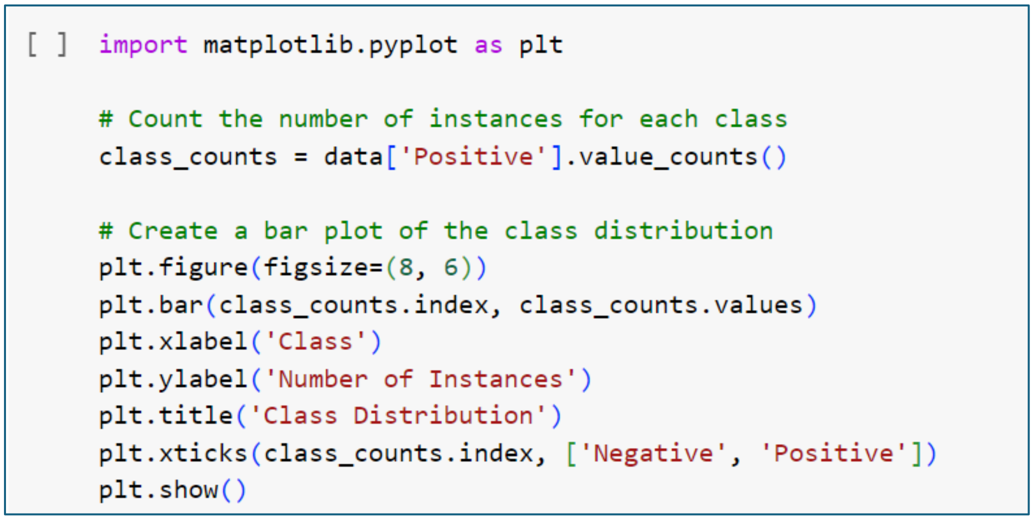
This step covers the data loading, basic visualization, and initial cleaning of the dataset:

- **Data Loading:**
    - The dataset is loaded using `pd.read_csv`, which reads the `amazon.csv` file and stores it as a pandas DataFrame named `data`.

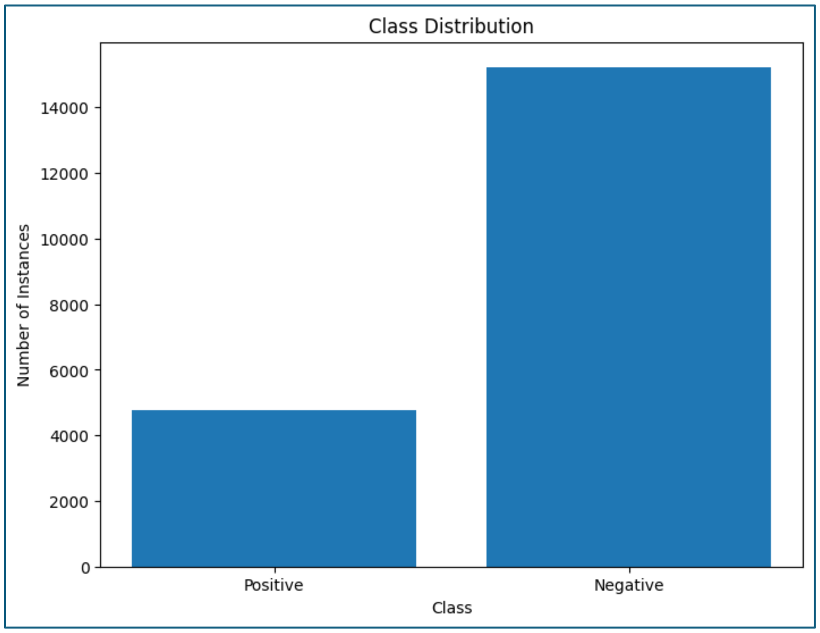


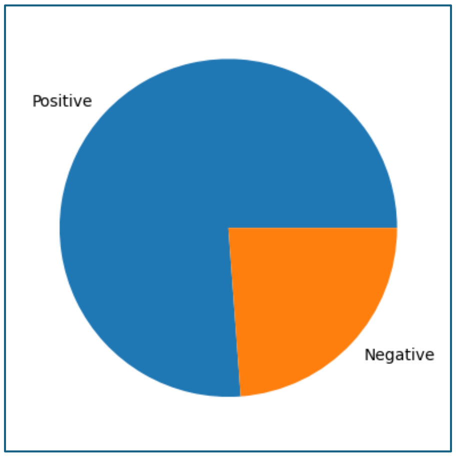
```
data = pd.read_csv('amazon.csv')
```

- **Basic Visualization:**
    - Class distribution is visualized using a bar plot and a pie chart to understand the balance between positive and negative sentiments in the dataset.



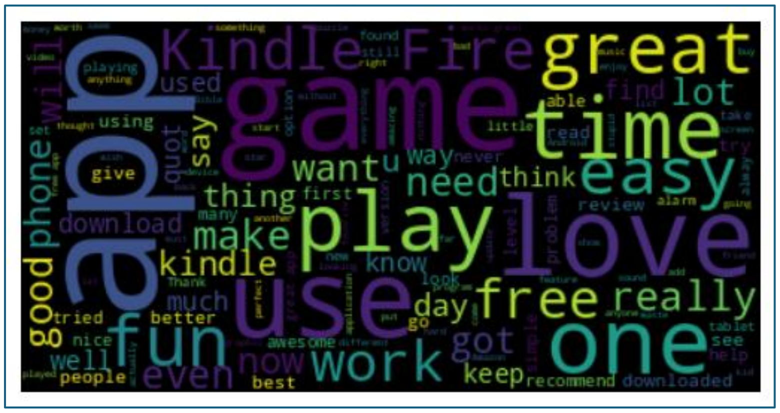
```
[ ]  import matplotlib.pyplot as plt

     # Count the number of instances for each class
     class_counts = data['Positive'].value_counts()

     # Create a bar plot of the class distribution
     plt.figure(figsize=(8, 6))
     plt.bar(class_counts.index, class_counts.values)
     plt.xlabel('Class')
     plt.ylabel('Number of Instances')
     plt.title('Class Distribution')
     plt.xticks(class_counts.index, ['Negative', 'Positive'])
     plt.show()
```



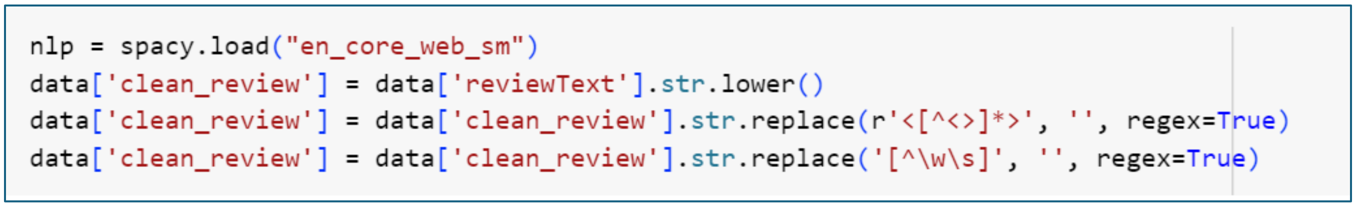


- **Word Cloud:**
    - A word cloud is generated to visualize common words in the reviews, giving an intuitive sense of the most frequent terms in the data.

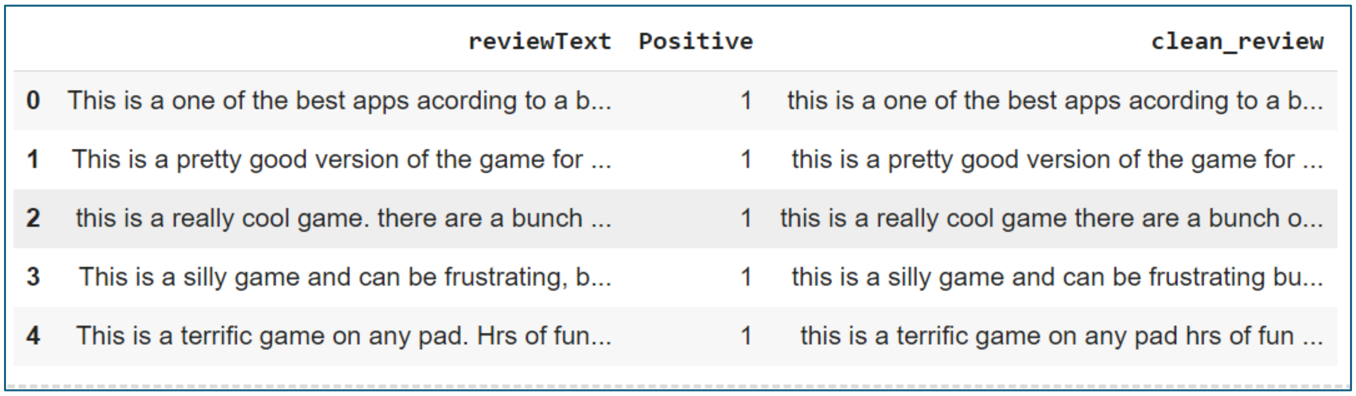
```python
# Assuming 'data' contains a column named 'reviewText' with text reviews.
text = " ".join(review for review in data.reviewText) # Changed 'data.Review' to 'data.reviewText'

# Create and generate a word cloud image:
wordcloud = WordCloud().generate(text)

# Display the generated image:
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis("off")
plt.show()
```



- **Data Cleaning:**
    - Reviews are converted to lowercase, HTML tags are removed using regex (<[^<>]*>), and punctuation is stripped away. These steps make the text uniform and easier for models to process.

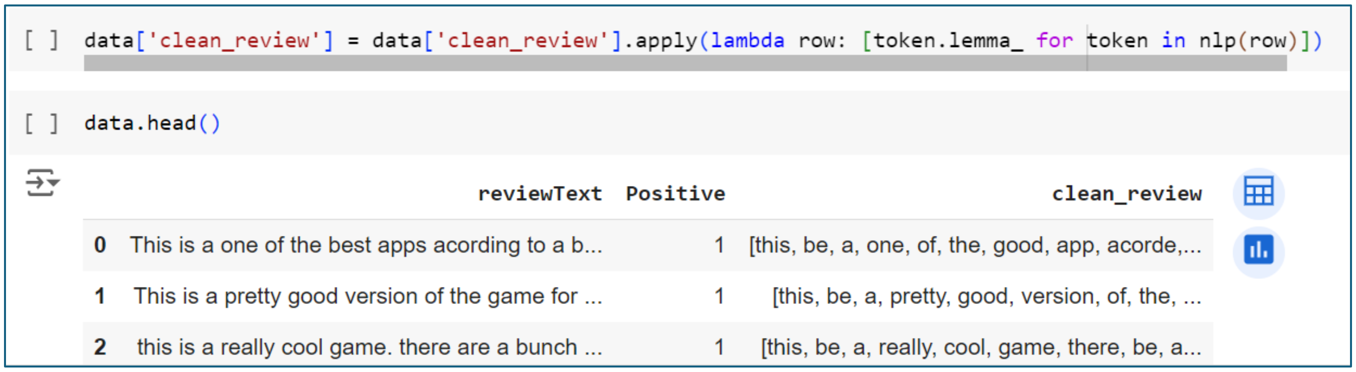


```
nlp = spacy.load("en_core_web_sm")
data['clean_review'] = data['reviewText'].str.lower()
data['clean_review'] = data['clean_review'].str.replace(r'<[^<>]*>', '', regex=True)
data['clean_review'] = data['clean_review'].str.replace('[^\w\s]', '', regex=True)
```



	reviewText	Positive	clean_review
0	This is a one of the best apps acording to a b...	1	this is a one of the best apps acording to a b...
1	This is a pretty good version of the game for ...	1	this is a pretty good version of the game for ...
2	this is a really cool game. there are a bunch ...	1	this is a really cool game there are a bunch o...
3	This is a silly game and can be frustrating, b...	1	this is a silly game and can be frustrating bu...
4	This is a terrific game on any pad. Hrs of fun...	1	this is a terrific game on any pad hrs of fun ...

- **Tokenization and Lemmatization:**
    - The `spacy` library is used for tokenization and lemmatization, which reduces each word to its base form (e.g., "running" to "run"). This is applied to the `clean_review` column.



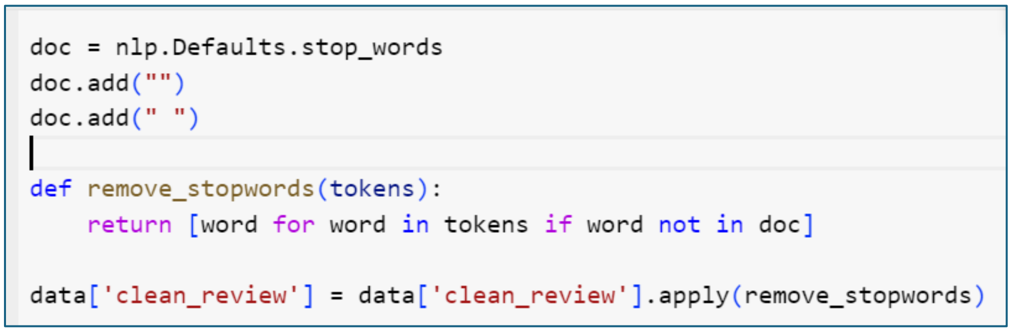
```
[ ] data['clean_review'] = data['clean_review'].apply(lambda row: [token.lemma_ for token in nlp(row)])

[ ] data.head()
```

	reviewText	Positive	clean_review
0	This is a one of the best apps acording to a b...	1	[this, be, a, one, of, the, good, app, acorde,...
1	This is a pretty good version of the game for ...	1	[this, be, a, pretty, good, version, of, the, ...
2	this is a really cool game. there are a bunch ...	1	[this, be, a, really, cool, game, there, be, a...

Step 3: Stopwords Removal

- **Stopwords Removal:**

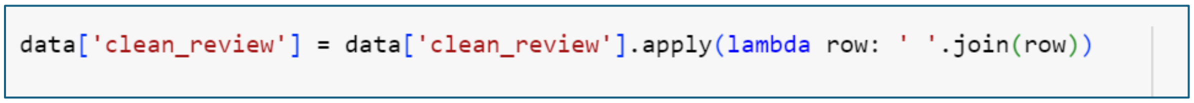
  - Stopwords (common words like "the", "and", "is") are removed to reduce noise in the dataset. `spacy.Defaults.stop_words` provides a list of stopwords, and a custom function (`remove_stopwords`) filters them out from the cleaned review tokens.



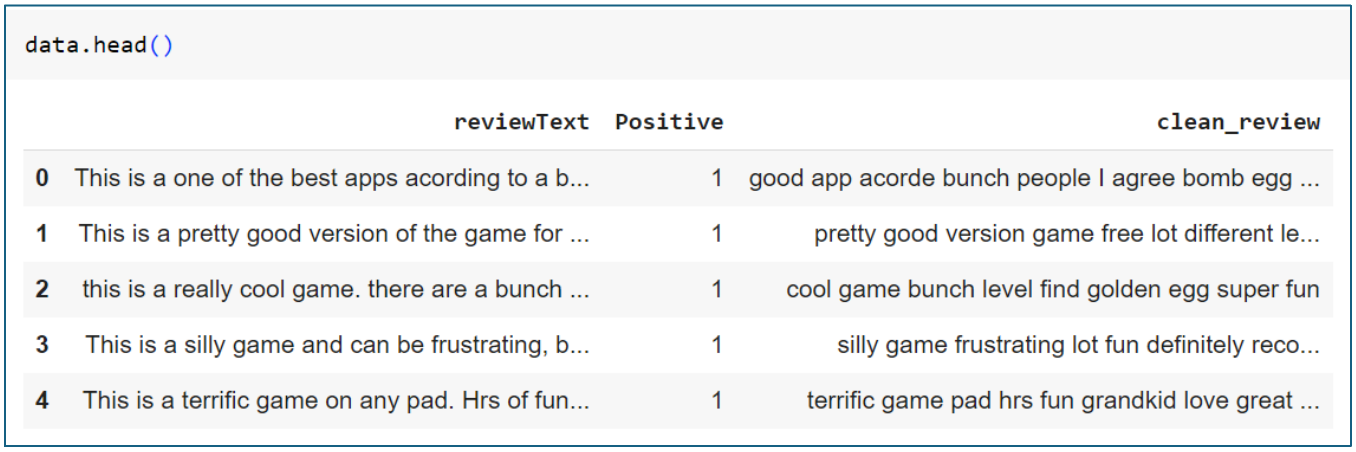
```
doc = nlp.Defaults.stop_words
doc.add("")
doc.add(" ")

def remove_stopwords(tokens):
    return [word for word in tokens if word not in doc]

data['clean_review'] = data['clean_review'].apply(remove_stopwords)
```

Combine Tokens:

- After stopwords are removed, the remaining tokens are joined back into a single string for further processing.



```
data['clean_review'] = data['clean_review'].apply(lambda row: ' '.join(row))
```

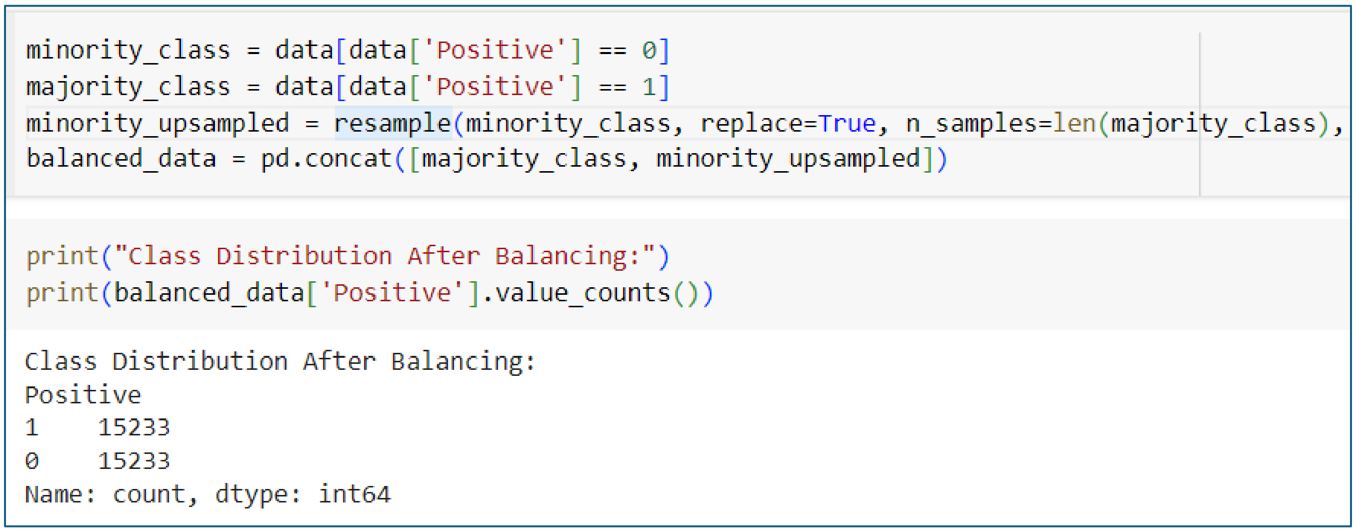


```
data.head()
```

	reviewText	Positive	clean_review
0	This is a one of the best apps acording to a b...	1	good app acorde bunch people I agree bomb egg ...
1	This is a pretty good version of the game for ...	1	pretty good version game free lot different le...
2	this is a really cool game. there are a bunch ...	1	cool game bunch level find golden egg super fun
3	This is a silly game and can be frustrating, b...	1	silly game frustrating lot fun definitely reco...
4	This is a terrific game on any pad. Hrs of fun...	1	terrific game pad hrs fun grandkid love great ...

Step 4: Balancing Data (Resampling)

- **Handling Class Imbalance:**
  - The dataset might be imbalanced, meaning one class (positive or negative) could dominate. `resample` is used to upsample the minority class (negative reviews in this case) to match the majority class (positive reviews), ensuring that the model doesn't become biased toward one class.
- **Save Cleaned Data:**
  - The balanced dataset is saved as `cleaned_data.csv` for future use.



```
minority_class = data[data['Positive'] == 0]
majority_class = data[data['Positive'] == 1]
minority_upsampled = resample(minority_class, replace=True, n_samples=len(majority_class),
balanced_data = pd.concat([majority_class, minority_upsampled])
```

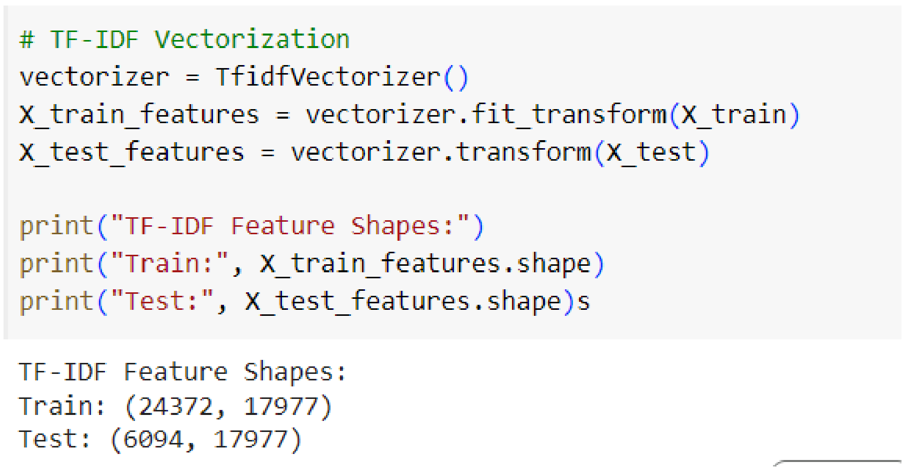
```
print("Class Distribution After Balancing:")
print(balanced_data['Positive'].value_counts())
```

```
Class Distribution After Balancing:
Positive
1    15233
0    15233
Name: count, dtype: int64
```

Step 5: Predict the Sentiment Using Traditional ML Models

This step involves transforming the text into numerical features and training machine learning models.

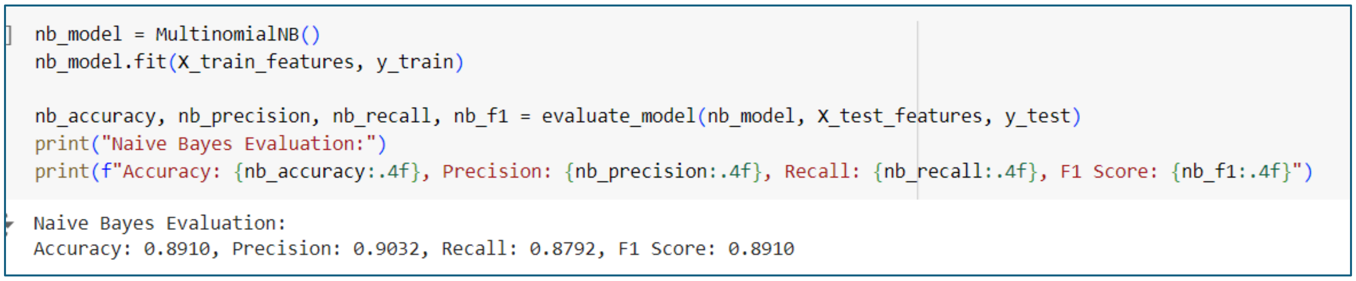
- **TF-IDF Vectorization:**
  - **TF-IDF** converts text data into a numerical representation based on the frequency of words and their importance in the document. It helps the model understand how relevant each word is for sentiment classification.



```
# TF-IDF Vectorization
vectorizer = TfidfVectorizer()
X_train_features = vectorizer.fit_transform(X_train)
X_test_features = vectorizer.transform(X_test)

print("TF-IDF Feature Shapes:")
print("Train:", X_train_features.shape)
print("Test:", X_test_features.shape)s
```

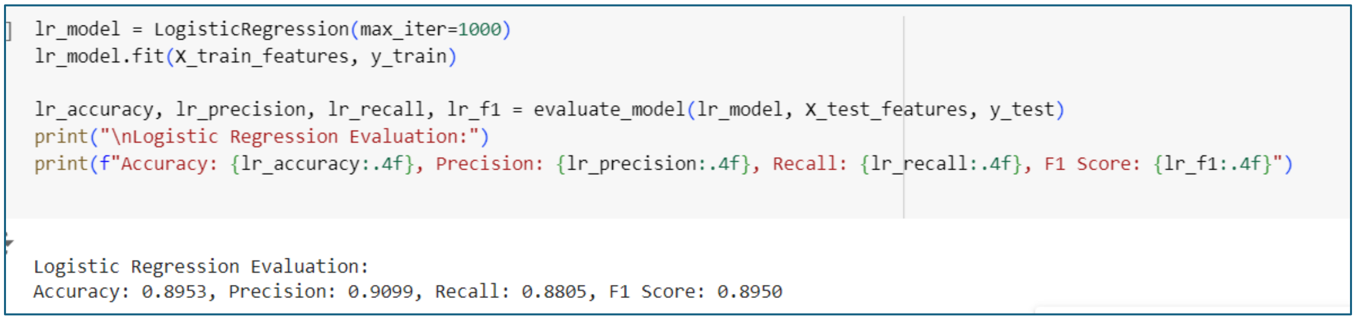
```
TF-IDF Feature Shapes:
Train: (24372, 17977)
Test: (6094, 17977)
```

- **Model Training:**
  - **Multinomial Naive Bayes (NB)** and **Logistic Regression (LR)** are trained on the TF-IDF features. These are standard machine learning algorithms that work well for text classification.
- **Model Evaluation:**
  - A custom function evaluate_model computes the **accuracy**, **precision**, **recall**, and **F1-score** for each model. These metrics help to understand how well the model is performing on the test set.



```
nb_model = MultinomialNB()
nb_model.fit(X_train_features, y_train)

nb_accuracy, nb_precision, nb_recall, nb_f1 = evaluate_model(nb_model, X_test_features, y_test)
print("Naive Bayes Evaluation:")
print(f"Accuracy: {nb_accuracy:.4f}, Precision: {nb_precision:.4f}, Recall: {nb_recall:.4f}, F1 Score: {nb_f1:.4f}")
```

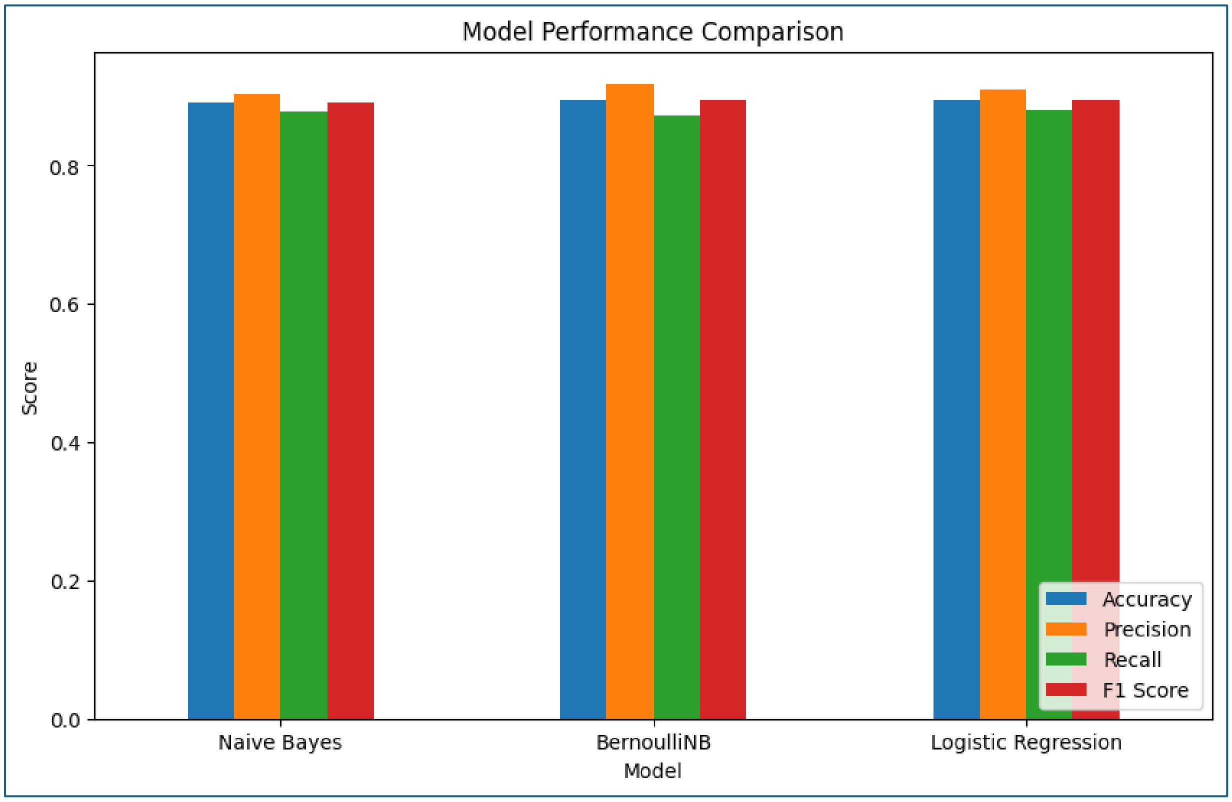
```
Naive Bayes Evaluation:
Accuracy: 0.8910, Precision: 0.9032, Recall: 0.8792, F1 Score: 0.8910
```



```
lr_model = LogisticRegression(max_iter=1000)
lr_model.fit(X_train_features, y_train)

lr_accuracy, lr_precision, lr_recall, lr_f1 = evaluate_model(lr_model, X_test_features, y_test)
print("\nLogistic Regression Evaluation:")
print(f"Accuracy: {lr_accuracy:.4f}, Precision: {lr_precision:.4f}, Recall: {lr_recall:.4f}, F1 Score: {lr_f1:.4f}")
```

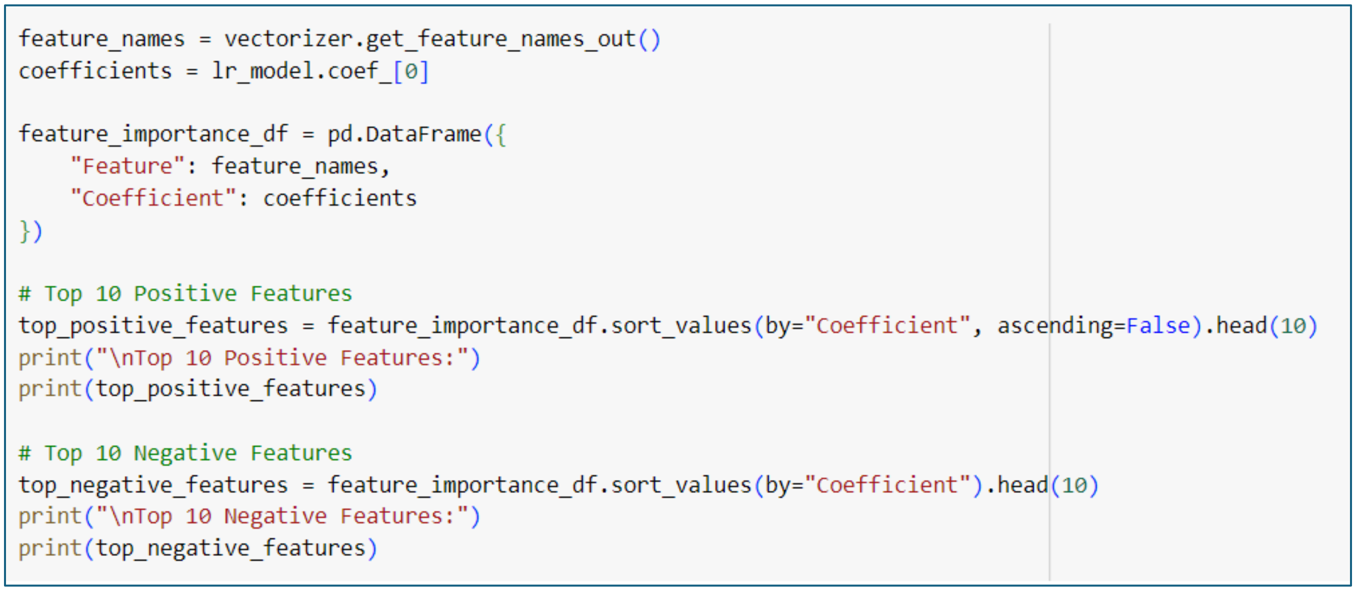
```
Logistic Regression Evaluation:
Accuracy: 0.8953, Precision: 0.9099, Recall: 0.8805, F1 Score: 0.8950
```

- **Model Comparison:**
  - The performance of Naive Bayes and Logistic Regression is compared using bar plots, making it easier to see which model performs better across various metrics.

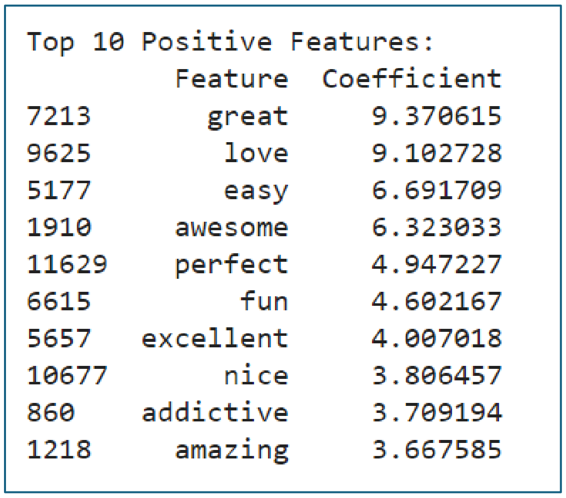


Step 6: Feature Importance (Logistic Regression)

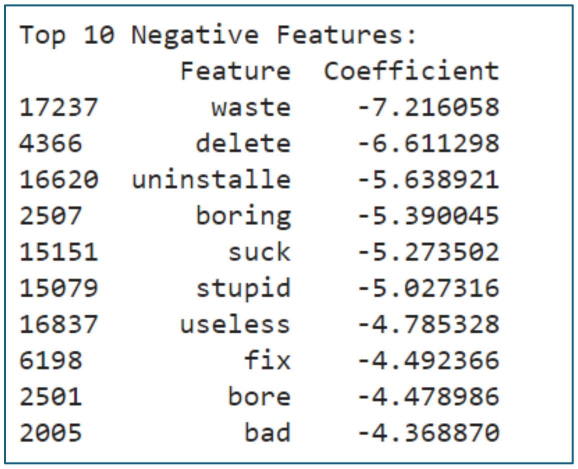
- **Understanding Model Predictions:**
  - For the Logistic Regression model, the most important features (words) that contribute to positive and negative predictions are extracted using model coefficients.
- **Top Positive/Negative Features:**
  - The top 10 positive and negative words (based on their coefficient values) are displayed, giving insight into which words have the most influence on the sentiment classification.



```
feature_names = vectorizer.get_feature_names_out()
coefficients = lr_model.coef_[0]

feature_importance_df = pd.DataFrame({
    "Feature": feature_names,
    "Coefficient": coefficients
})

# Top 10 Positive Features
top_positive_features = feature_importance_df.sort_values(by="Coefficient", ascending=False).head(10)
print("\nTop 10 Positive Features:")
print(top_positive_features)

# Top 10 Negative Features
top_negative_features = feature_importance_df.sort_values(by="Coefficient").head(10)
print("\nTop 10 Negative Features:")
print(top_negative_features)
```



```
Top 10 Positive Features:
         Feature  Coefficient
7213       great     9.370615
9625        love     9.102728
5177        easy     6.691709
1910     awesome     6.323033
11629    perfect     4.947227
6615         fun     4.602167
5657   excellent     4.007018
10677       nice     3.806457
860    addictive     3.709194
1218     amazing     3.667585
```



```
Top 10 Negative Features:
          Feature  Coefficient
17237       waste    -7.216058
4366       delete    -6.611298
16620  uninstalle    -5.638921
2507       boring    -5.390045
15151        suck    -5.273502
15079      stupid    -5.027316
16837     useless    -4.785328
6198          fix    -4.492366
2501         bore    -4.478986
2005          bad    -4.368870
```

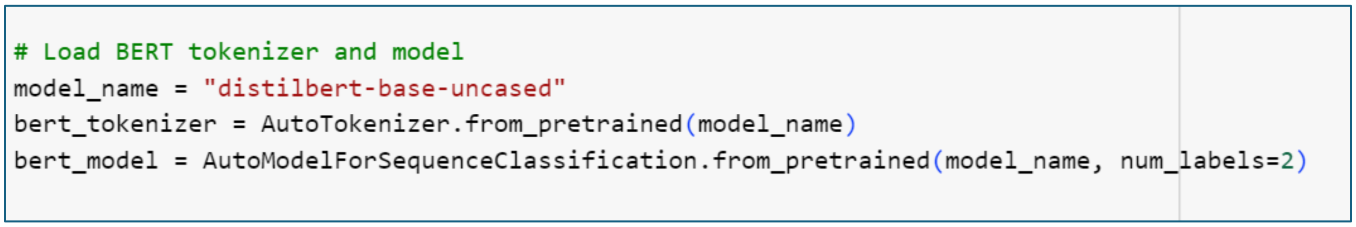
Step 7: Applying Different Tokenization/Vectorization Techniques

In natural language processing (NLP), vectorization is the process of converting text into numerical representations. This is crucial for enabling machine learning models to understand and process textual data. In this project, we implement four different vectorization/tokenization techniques: **BERT**, **GloVe**, **Word2Vec**, and **Doc2Vec**. Each method has unique characteristics and applications.

7.1 BERT Tokenization

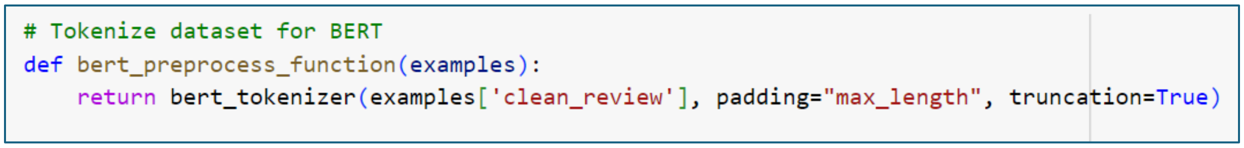
BERT (Bidirectional Encoder Representations from Transformers) is a transformer-based model designed to understand the context of words in a sentence better. BERT tokenization involves breaking down the text into tokens and converting these tokens into IDs that can be fed into the model.

1. **Loading BERT Model and Tokenizer:**
   - The `AutoTokenizer` loads the tokenizer for the specified BERT model, `distilbert-base-uncased`, which is a lighter version of BERT.
   - `AutoModelForSequenceClassification` initializes the BERT model configured for binary classification tasks (hence `num_labels=2`).



```
# Load BERT tokenizer and model
model_name = "distilbert-base-uncased"
bert_tokenizer = AutoTokenizer.from_pretrained(model_name)
bert_model = AutoModelForSequenceClassification.from_pretrained(model_name, num_labels=2)
```

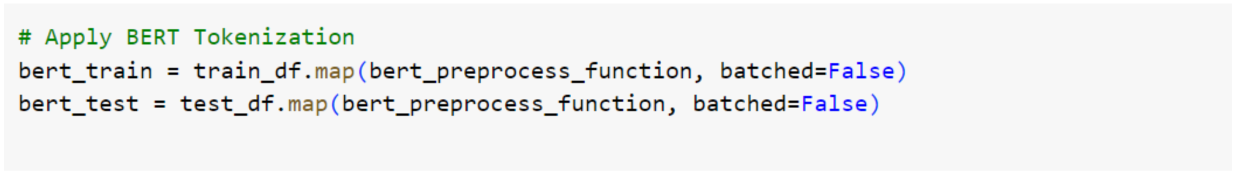
2. **Preprocessing Function:**
   - The `bert_preprocess_function` tokenizes the input text from the 'clean_review' column, applying padding to ensure all sequences are of uniform length and truncating any sequences that exceed the maximum length.



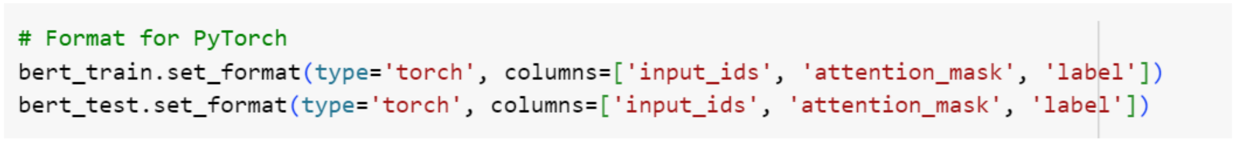
```
# Tokenize dataset for BERT
def bert_preprocess_function(examples):
    return bert_tokenizer(examples['clean_review'], padding="max_length", truncation=True)
```

3. **Applying Tokenization:**
    - The function is applied to both training and testing datasets using the `.map()` method to tokenize each entry.

4. **Setting Data Format:**
    - The datasets are converted into a format compatible with PyTorch, specifying which columns to include (`input_ids`, `attention_mask`, and `label`).



```
# Apply BERT Tokenization
bert_train = train_df.map(bert_preprocess_function, batched=False)
bert_test = test_df.map(bert_preprocess_function, batched=False)
```



```
# Format for PyTorch
bert_train.set_format(type='torch', columns=['input_ids', 'attention_mask', 'label'])
bert_test.set_format(type='torch', columns=['input_ids', 'attention_mask', 'label'])
```

7.2 GloVe Embedding Vectorization

GloVe (Global Vectors for Word Representation) is a model for generating vector representations of words, capturing their meanings based on the context in which they appear.

1. **Loading GloVe Embeddings**: The GloVe class is imported, and the embeddings for the '6B' dataset with a specified dimensionality (e.g., 100) are initialized.

2. **Vectorization Logic**: Each sentence is tokenized, and a vector is created by calculating the mean of the GloVe vectors for each token present in the vocabulary.

3. **Storing Vectors**: The resulting vectors are stored in a new column named 'vector' in the DataFrame.

4. **Returning DataFrame**: The modified DataFrame with the GloVe vectors is returned.

```
from torchtext.vocab import GloVe
import numpy as np
# Use GloVe embeddings
def glove_vectorization(df, embedding_dim=100):
    glove = GloVe(name='6B', dim=embedding_dim)
    vectors = []

    for sentence in df['clean_review']:
        tokens = sentence.split()  # Simple tokenization by splitting
        vector = np.mean([glove[token].numpy() for token in tokens if token in glove.stoi], axis=0)
        vectors.append(vector)

    # Convert the Dataset object to a pandas DataFrame for modification
    df = df.to_pandas()
    df['vector'] = vectors
    return df
# Vectorize using GloVe
glove_train = glove_vectorization(train_df)
glove_test = glove_vectorization(test_df)
```

7.3 Word2Vec Embedding Vectorization

Word2Vec is a predictive model used to learn word associations from large datasets. It can produce dense vector representations for words.

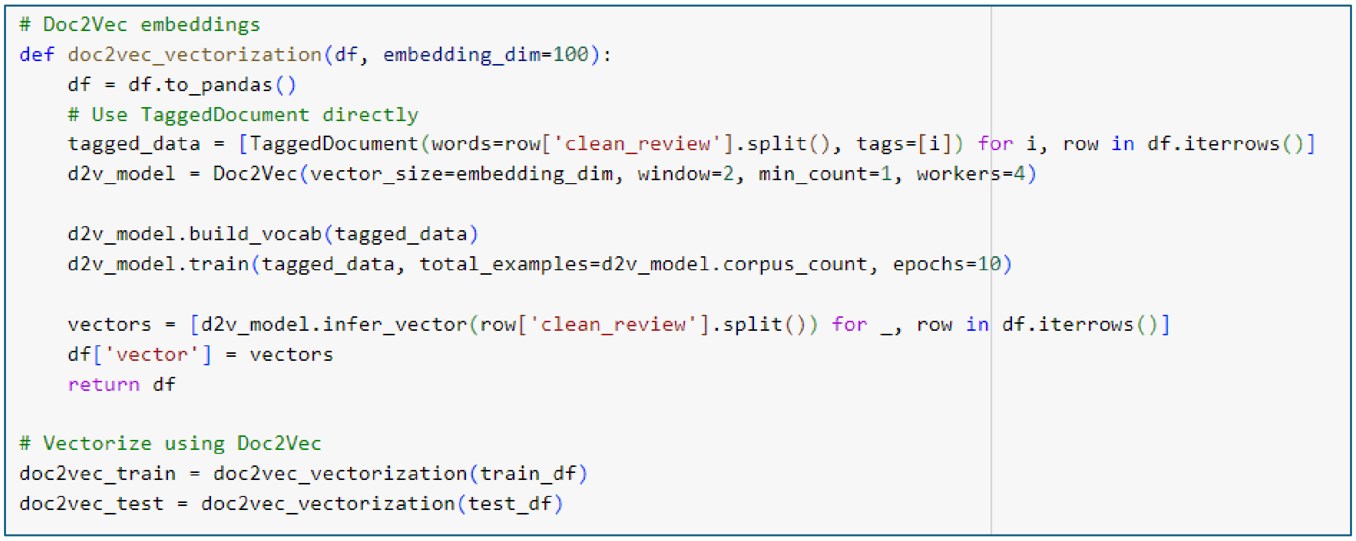
1. **Model Initialization**:
    - The `Word2Vec` class is imported, and a new model is created using the tokenized sentences. The model will learn to represent words in the specified embedding space.
2. **Generating Word Vectors**:
    - For each sentence, the Word2Vec model generates vectors, and their mean is computed to create a single vector representation for each sentence.
3. **Storing Vectors**:
    - Similar to GloVe, the resulting vectors are stored in the DataFrame.

```
# Word2Vec embeddings
def word2vec_vectorization(df, embedding_dim=100):
    sentences = [sentence.split() for sentence in df['clean_review']]  # Tokenizing into words
    w2v_model = Word2Vec(sentences, vector_size=embedding_dim, min_count=1, workers=4)
    vectors = []
    for sentence in sentences:
        vector = np.mean([w2v_model.wv[word] for word in sentence if word in w2v_model.wv], axis=0)
        vectors.append(vector)

    df = df.to_pandas()
    df['vector'] = vectors
    return df
# Vectorize using Word2Vec
w2v_train = word2vec_vectorization(train_df)
w2v_test = word2vec_vectorization(test_df)
```

7.4 Doc2Vec Embedding Vectorization

Doc2Vec is an extension of Word2Vec that allows for the vectorization of entire documents, not just individual words.

1. **Data Preparation**:
    - Each review is tokenized and tagged with a unique identifier to form TaggedDocument objects, which will be used to train the Doc2Vec model.
2. **Model Training**:
    - A Doc2Vec model is initialized and trained on the tagged data, allowing it to learn representations for both words and documents.
3. **Generating Document Vectors**:
    - Each document's vector representation is inferred using the trained model and stored in the DataFrame.



```python
# Doc2Vec embeddings
def doc2vec_vectorization(df, embedding_dim=100):
    df = df.to_pandas()
    # Use TaggedDocument directly
    tagged_data = [TaggedDocument(words=row['clean_review'].split(), tags=[i]) for i, row in df.iterrows()]
    d2v_model = Doc2Vec(vector_size=embedding_dim, window=2, min_count=1, workers=4)

    d2v_model.build_vocab(tagged_data)
    d2v_model.train(tagged_data, total_examples=d2v_model.corpus_count, epochs=10)

    vectors = [d2v_model.infer_vector(row['clean_review'].split()) for _, row in df.iterrows()]
    df['vector'] = vectors
    return df

# Vectorize using Doc2Vec
doc2vec_train = doc2vec_vectorization(train_df)
doc2vec_test = doc2vec_vectorization(test_df)
```

Step 8: Model Definition and Training

This section covers the training of two different types of models:

1. **BERT Model Training**: Using a pre-trained BERT model for text classification.
2. **Simple MLP (Multi-Layer Perceptron)**: Training for three embedding models—GloVe, Word2Vec, and Doc2Vec—using simple fully connected layers.

8.1 BERT Model Training

We start by training a **BERT-based classifier**, leveraging the distilbert-base-uncased model. BERT is known for its ability to understand deep contextual meanings of words. Here's a breakdown of the steps used in the process:

---

1. Loading and Preparing Data:

```
# Convert pandas DataFrame to Hugging Face Dataset
dataset = Dataset.from_pandas(df)
train_test_split = dataset.train_test_split(test_size=0.2)
train_dataset = train_test_split['train']
test_dataset = train_test_split['test']
```

- balanced_data: A cleaned dataset containing reviews and their corresponding labels (binary classification problem).
- `dropna()` ensures that any rows with missing data are excluded.
- The dataset is filtered to only keep relevant columns: `clean_review` (text) and label (target variable).

2. Loading Pre-trained BERT Model and Tokenizer:

- **Model Name**: We use a smaller, faster version of BERT, called **DistilBERT**.
- **Tokenizer**: The tokenizer converts text to BERT input format (tokens, attention masks).
- **Model**: A classification model built on top of BERT with `num_labels=2` for binary classification.

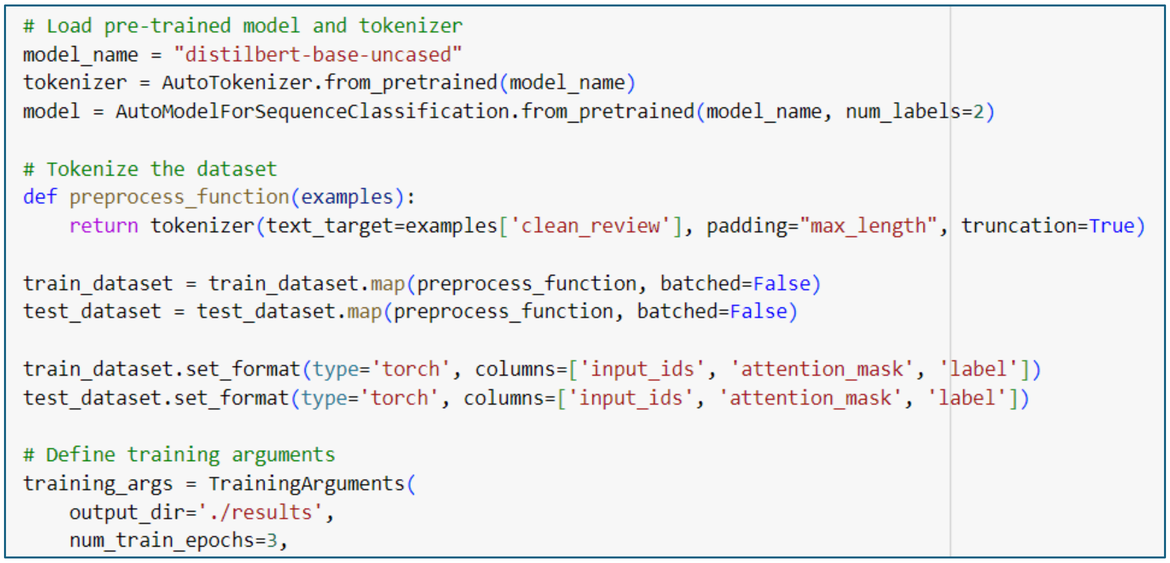
3. Tokenizing Dataset: ensures that all input sequences have the same length by padding shorter sequences and truncating longer ones.

4. Defining Training Arguments: Define parameters for model training, such as:

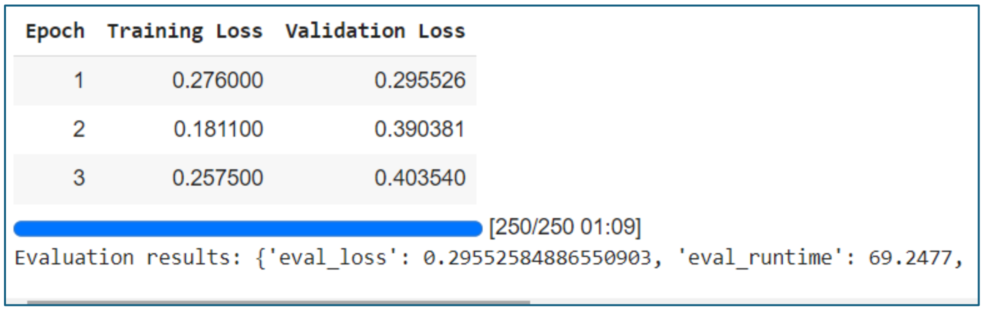
- **Epochs**: 3 complete passes over the training data.
- **Batch Size**: 8 for training, 16 for evaluation.
- **Evaluation Strategy**: Evaluation is done at the end of each epoch, and the model is saved if it performs well.

5. Training and Evaluation:

- **Trainer**: An abstraction that simplifies model training and evaluation.
- **Evaluation Results**: After training, the model is evaluated using the test dataset, and results such as accuracy and loss are printed.



```python
# Load pre-trained model and tokenizer
model_name = "distilbert-base-uncased"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForSequenceClassification.from_pretrained(model_name, num_labels=2)

# Tokenize the dataset
def preprocess_function(examples):
    return tokenizer(text_target=examples['clean_review'], padding="max_length", truncation=True)

train_dataset = train_dataset.map(preprocess_function, batched=False)
test_dataset = test_dataset.map(preprocess_function, batched=False)

train_dataset.set_format(type='torch', columns=['input_ids', 'attention_mask', 'label'])
test_dataset.set_format(type='torch', columns=['input_ids', 'attention_mask', 'label'])

# Define training arguments
training_args = TrainingArguments(
    output_dir='./results',
    num_train_epochs=3,
```



Epoch	Training Loss	Validation Loss
1	0.276000	0.295526
2	0.181100	0.390381
3	0.257500	0.403540

```
[250/250 01:09]
Evaluation results: {'eval_loss': 0.29552584886550903, 'eval_runtime': 69.2477,
```

8.2 MLP Model for GloVe, Word2Vec, and Doc2Vec

For the GloVe, Word2Vec, and Doc2Vec embeddings, a **Simple MLP** model is trained. These embeddings represent words or documents as vectors of real numbers. The MLP is used to classify these vectors.

1. Defining the Simple MLP Model:

- **Input and Output Dimensions**:
  - `input_dim=100` is based on the size of the word vectors.
  - `output_dim=2` corresponds to the two classes in the classification task.
- **Architecture**: The model consists of two fully connected layers:
  - fc1: Takes the input vector and maps it to a hidden space of size 128.
  - fc2: Reduces the hidden space to the output size (2), where each class gets a score.
  - **ReLU**: An activation function that adds non-linearity to the model.

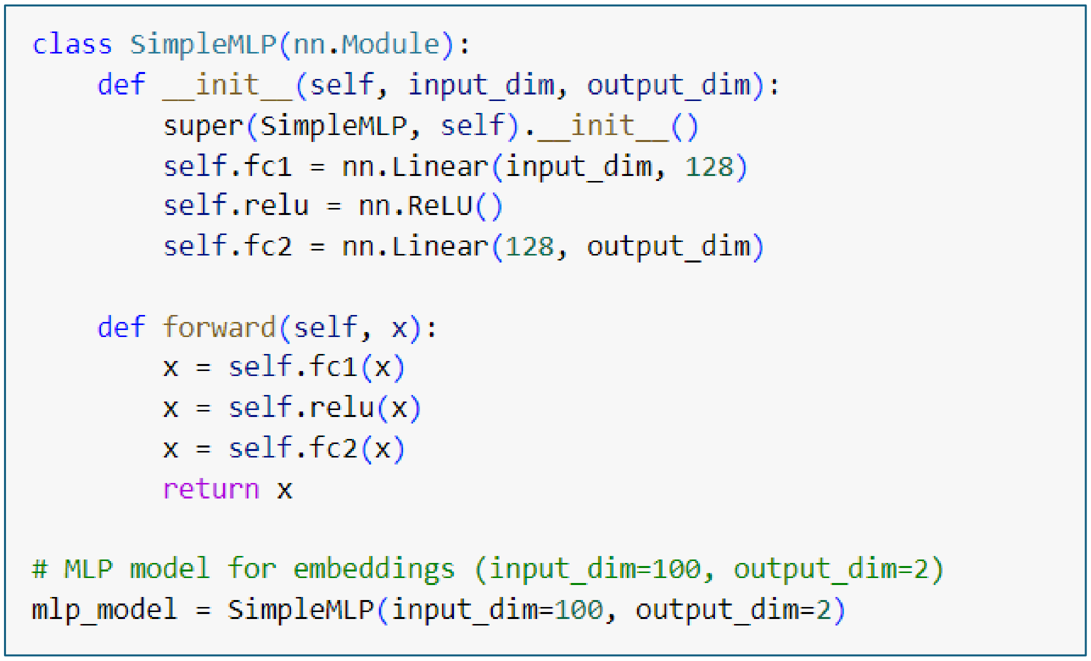
2. Training the MLP Model:

- **Adam Optimizer**: Used to adjust the weights based on the computed gradients.
- **CrossEntropyLoss**: A common loss function for classification tasks.
- **Training Loop**: The model goes through multiple epochs, updating the weights to minimize the loss.

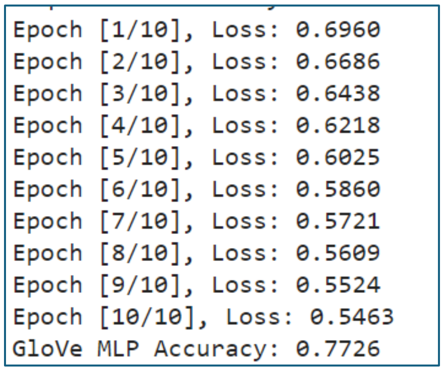
3. Evaluating the MLP Model: Predictions are made on the test data, and the model's accuracy is computed.

4. Training and Evaluating Models for Different Embeddings:

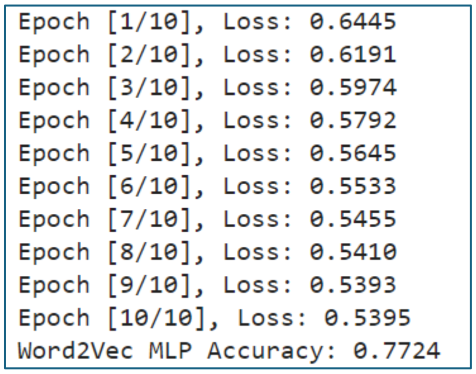
- **Separate MLP models**: For each embedding technique (GloVe, Word2Vec, Doc2Vec), a new MLP model is trained and evaluated.
- **Accuracy**: Accuracy is printed for each embedding after evaluation.



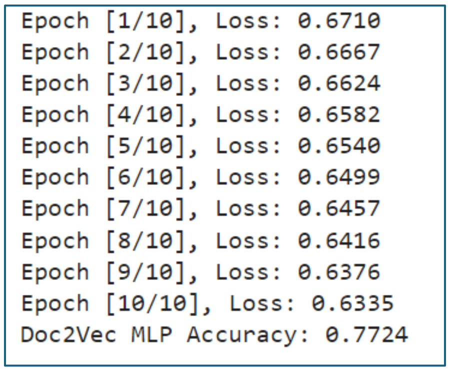
```python
class SimpleMLP(nn.Module):
    def __init__(self, input_dim, output_dim):
        super(SimpleMLP, self).__init__()
        self.fc1 = nn.Linear(input_dim, 128)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(128, output_dim)

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        return x

# MLP model for embeddings (input_dim=100, output_dim=2)
mlp_model = SimpleMLP(input_dim=100, output_dim=2)
```



```
Epoch [1/10], Loss: 0.6960
Epoch [2/10], Loss: 0.6686
Epoch [3/10], Loss: 0.6438
Epoch [4/10], Loss: 0.6218
Epoch [5/10], Loss: 0.6025
Epoch [6/10], Loss: 0.5860
Epoch [7/10], Loss: 0.5721
Epoch [8/10], Loss: 0.5609
Epoch [9/10], Loss: 0.5524
Epoch [10/10], Loss: 0.5463
GloVe MLP Accuracy: 0.7726
```

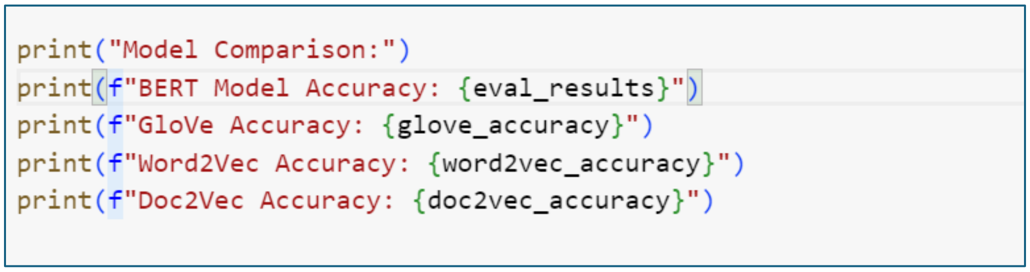


```
Epoch [1/10], Loss: 0.6445
Epoch [2/10], Loss: 0.6191
Epoch [3/10], Loss: 0.5974
Epoch [4/10], Loss: 0.5792
Epoch [5/10], Loss: 0.5645
Epoch [6/10], Loss: 0.5533
Epoch [7/10], Loss: 0.5455
Epoch [8/10], Loss: 0.5410
Epoch [9/10], Loss: 0.5393
Epoch [10/10], Loss: 0.5395
Word2Vec MLP Accuracy: 0.7724
```

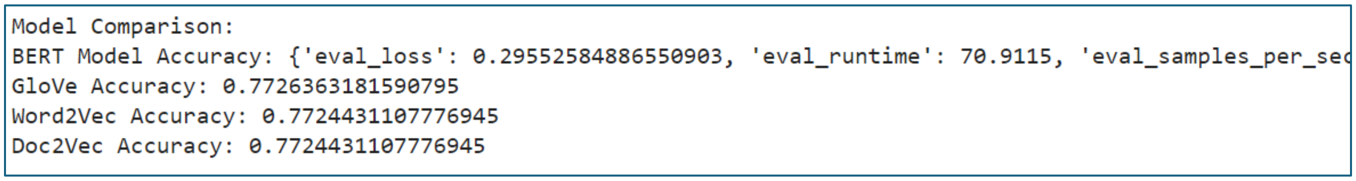


```
Epoch [1/10], Loss: 0.6710
Epoch [2/10], Loss: 0.6667
Epoch [3/10], Loss: 0.6624
Epoch [4/10], Loss: 0.6582
Epoch [5/10], Loss: 0.6540
Epoch [6/10], Loss: 0.6499
Epoch [7/10], Loss: 0.6457
Epoch [8/10], Loss: 0.6416
Epoch [9/10], Loss: 0.6376
Epoch [10/10], Loss: 0.6335
Doc2Vec MLP Accuracy: 0.7724
```

Step 9: Model Comparison



```python
print("Model Comparison:")
print(f"BERT Model Accuracy: {eval_results}")
print(f"GloVe Accuracy: {glove_accuracy}")
print(f"Word2Vec Accuracy: {word2vec_accuracy}")
print(f"Doc2Vec Accuracy: {doc2vec_accuracy}")
```



```
Model Comparison:
BERT Model Accuracy: {'eval_loss': 0.29552584886550903, 'eval_runtime': 70.9115, 'eval_samples_per_se
GloVe Accuracy: 0.7726363181590795
Word2Vec Accuracy: 0.7724431107776945
Doc2Vec Accuracy: 0.7724431107776945
```

Step 10: Prediction Interface Using Gradio

In this step, we build an easy-to-use **prediction interface** using Gradio. Gradio is a powerful library that allows us to create simple web-based interfaces for machine learning models with just a few lines of code. We'll use it to interact with our fine-tuned BERT model for **text sentiment classification**.

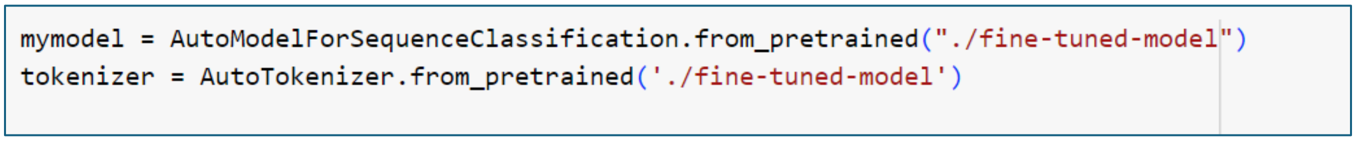
10.1 Save the Fine-Tuned Model

First, after successfully fine-tuning the model, save the model and tokenizer locally so that they can be reloaded for inference later:

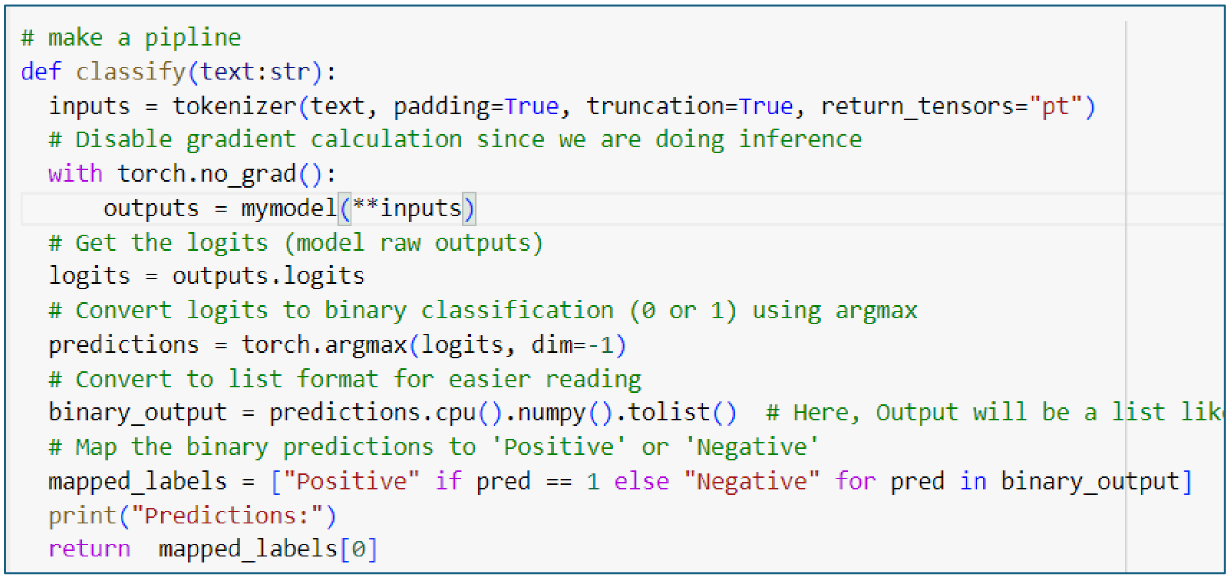
```
# Save the fine-tuned model
model.save_pretrained("./fine-tuned-model")
tokenizer.save_pretrained("./fine-tuned-model")
```

10.2 Model Inference Pipeline

We'll define the function classify() to make predictions on input text using the fine-tuned model. This function will take in a text string, tokenize it, make predictions using the model, and output a **'Positive'** or **'Negative'** sentiment based on the text.



```
mymodel = AutoModelForSequenceClassification.from_pretrained("./fine-tuned-model")
tokenizer = AutoTokenizer.from_pretrained('./fine-tuned-model')
```

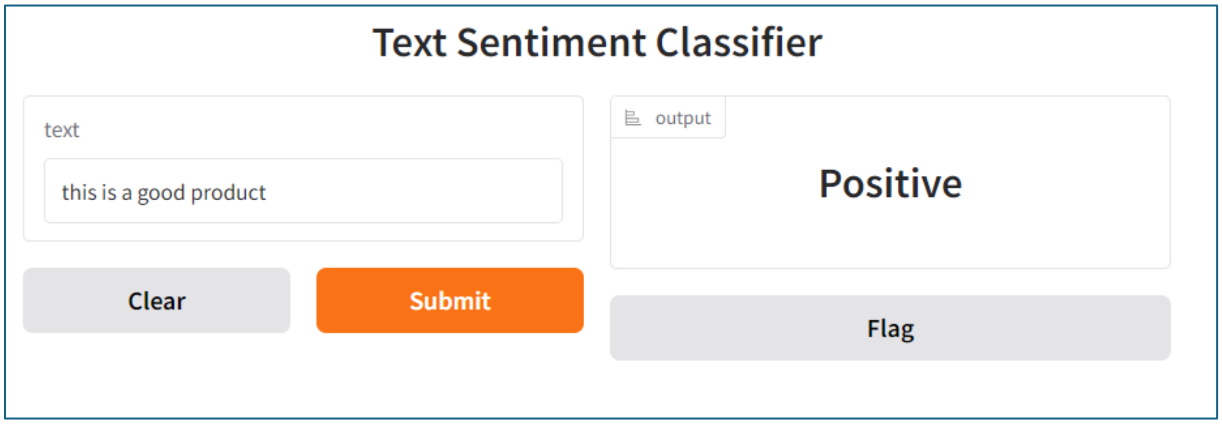


```
# make a pipline
def classify(text:str):
  inputs = tokenizer(text, padding=True, truncation=True, return_tensors="pt")
  # Disable gradient calculation since we are doing inference
  with torch.no_grad():
      outputs = mymodel(**inputs)
  # Get the logits (model raw outputs)
  logits = outputs.logits
  # Convert logits to binary classification (0 or 1) using argmax
  predictions = torch.argmax(logits, dim=-1)
  # Convert to list format for easier reading
  binary_output = predictions.cpu().numpy().tolist()  # Here, Output will be a list lik
  # Map the binary predictions to 'Positive' or 'Negative'
  mapped_labels = ["Positive" if pred == 1 else "Negative" for pred in binary_output]
  print("Predictions:")
  return  mapped_labels[0]
```

10.3 Gradio Interface

Now, we integrate the `classify()` function into a **Gradio** interface. Gradio makes it easy to build a web interface that will take input from users, classify the sentiment, and return the result.

With this, a user-friendly web app will be created where users can input text, and it will return the predicted sentiment.



Step 11: Django Deployment

In this final step, we deploy the model using Django by building an **API** to handle user requests and a **web interface** to display the classification results.

11.1 Create HTML Template

Start by creating an HTML file in your Django project. This template will serve as the front end for users to input text and receive sentiment classification.

Sample from HTML template (`index.html`):

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Chat System</title>
    <link
      href="https://cdn.jsdelivr.net/npm/tailwindcss@2.2.19/dist/tailwind.min.css"
      rel="stylesheet"
    />
    <style>
      .chat-container {
        display: flex;
        flex-direction: column;
        height: 100vh;
      }
```

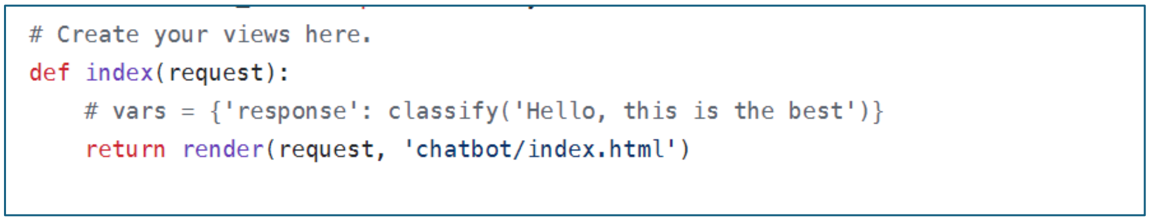
11.2 Create a New Django Application

Create a new application named chatbot in your Django project:

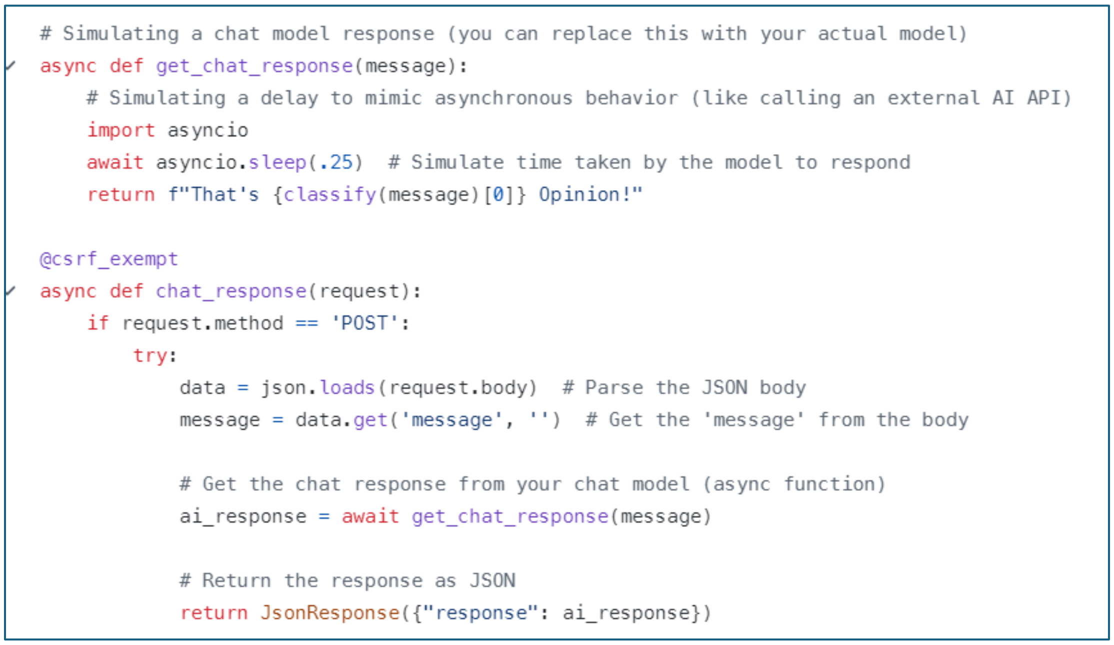
11.3 Initialize API and Create URL

In the chatbot application, create a view to handle requests from the HTML form, load the fine-tuned model, and make predictions.

- **views.py:**

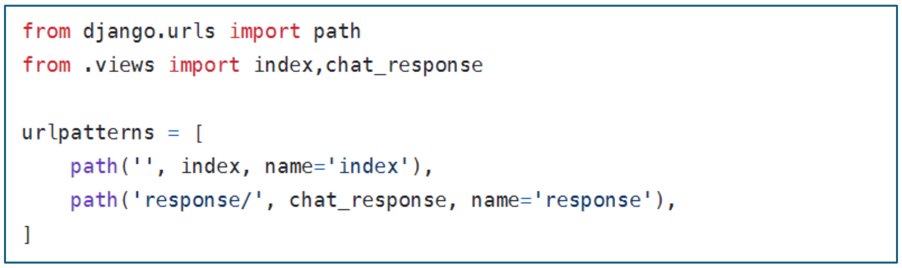


```python
# Create your views here.
def index(request):
    # vars = {'response': classify('Hello, this is the best')}
    return render(request, 'chatbot/index.html')
```



```python
# Simulating a chat model response (you can replace this with your actual model)
async def get_chat_response(message):
    # Simulating a delay to mimic asynchronous behavior (like calling an external AI API)
    import asyncio
    await asyncio.sleep(.25)  # Simulate time taken by the model to respond
    return f"That's {classify(message)[0]} Opinion!"

@csrf_exempt
async def chat_response(request):
    if request.method == 'POST':
        try:
            data = json.loads(request.body)  # Parse the JSON body
            message = data.get('message', '')  # Get the 'message' from the body

            # Get the chat response from your chat model (async function)
            ai_response = await get_chat_response(message)

            # Return the response as JSON
            return JsonResponse({"response": ai_response})
```

- **urls.py:**



```python
from django.urls import path
from .views import index,chat_response

urlpatterns = [
    path('', index, name='index'),
    path('response/', chat_response, name='response'),
]
```

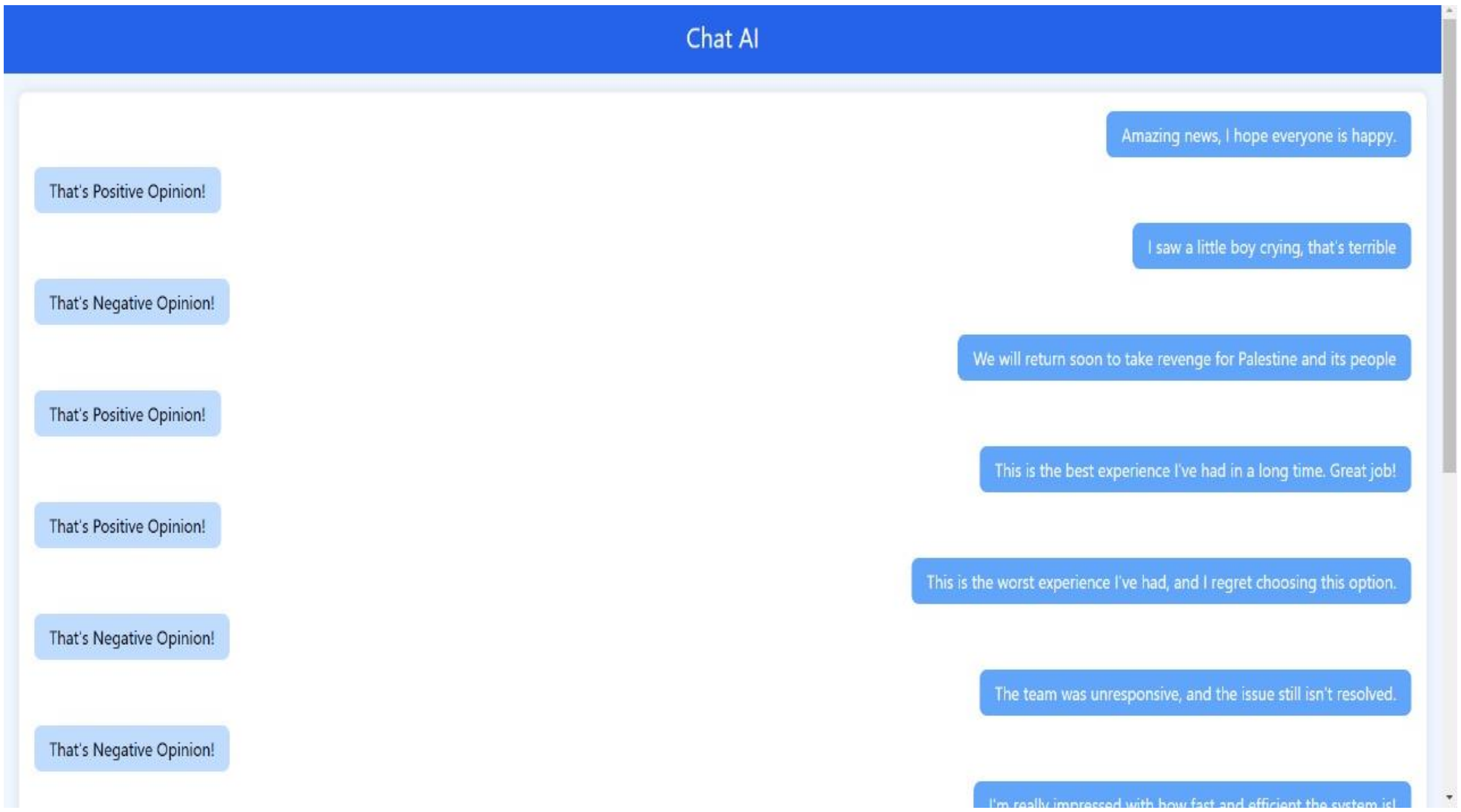
11.4 Deploying the Model with Django

Now, when a user visits the site and submits text through the form, the classify function is called. This function processes the input text, passes it through the model for prediction, and returns the classification result on the web page.

Results and Test Cases for Sentiment Classifier

In this step, I tested the fine-tuned sentiment analysis model deployed using Django and displayed the results on a web interface. The goal was to ensure the model correctly classifies text inputs as either "Positive" or "Negative." Below is a breakdown of the test cases shown in the image:

Samples From Test Cases 1



Chat AI

Amazing news, I hope everyone is happy.

That's Positive Opinion!

I saw a little boy crying, that's terrible

That's Negative Opinion!

We will return soon to take revenge for Palestine and its people

That's Positive Opinion!

This is the best experience I've had in a long time. Great job!

That's Positive Opinion!

This is the worst experience I've had, and I regret choosing this option.

That's Negative Opinion!

The team was unresponsive, and the issue still isn't resolved.

That's Negative Opinion!

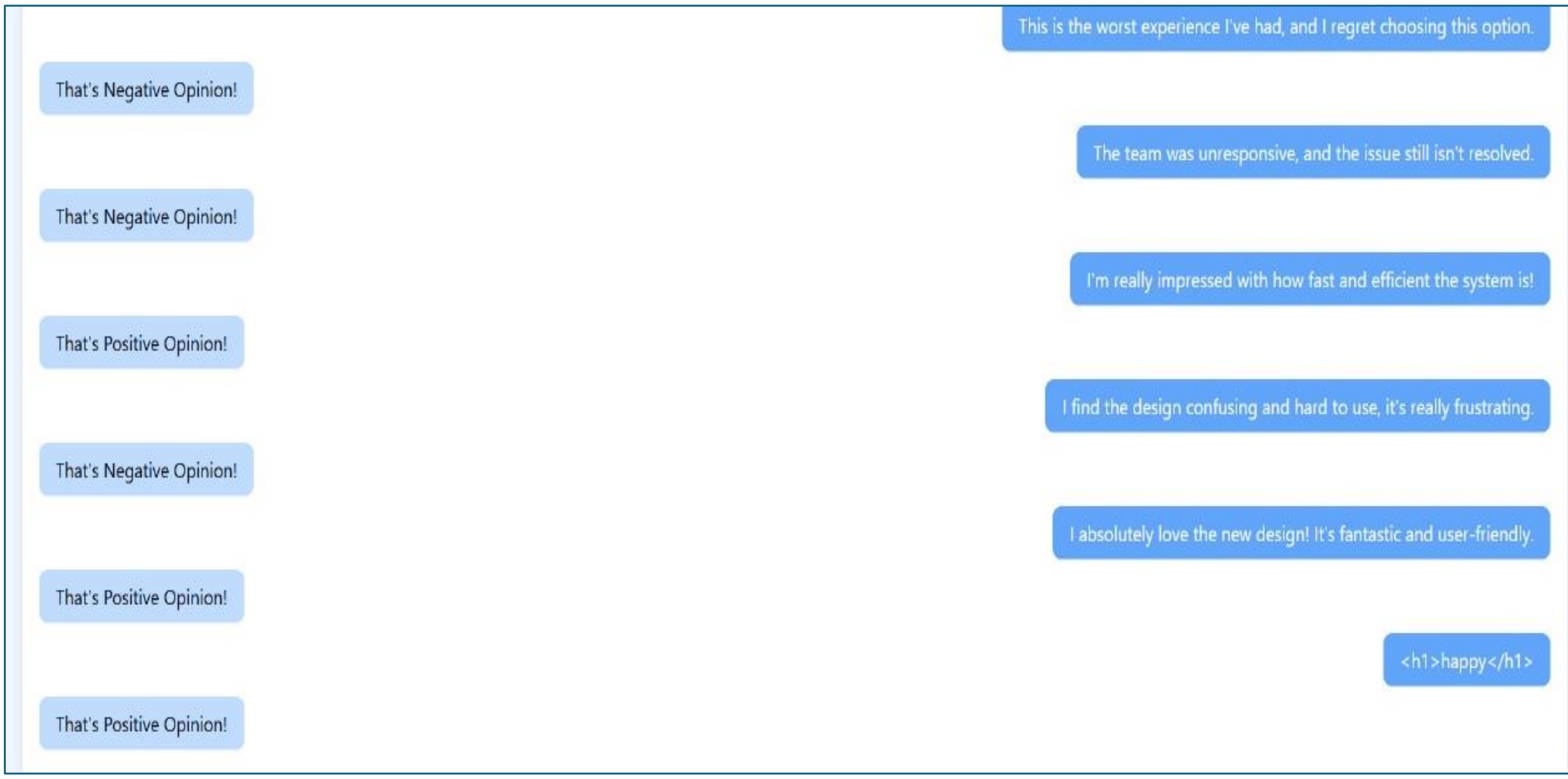
I'm really impressed with how fast and efficient the system is!

1. **Input: "Amazing news, I hope everyone is happy."**
    - **Expected Output:** Positive
    - **Model Output:** That's Positive Opinion!
    - **Result:** ✅ Passed
2. **Input: "I saw a little boy crying, that's terrible."**
    - **Expected Output:** Negative
    - **Model Output:** That's Negative Opinion!
    - **Result:** ✅ Passed
3. **Input: "We will return soon to take revenge for Palestine and its people."**
    - **Expected Output:** Negative (due to the aggressive tone)
    - **Model Output:** That's Positive Opinion!
    - **Result:** ❌ Failed (This could indicate the model struggles with contextually negative sentiments involving complex topics like revenge.)
4. **Input: "This is the best experience I've had in a long time. Great job!"**
    - **Expected Output:** Positive
    - **Model Output:** That's Positive Opinion!
    - **Result:** ✅ Passed
5. **Input: "This is the worst experience I've had, and I regret choosing this option."**
    - **Expected Output:** Negative
    - **Model Output:** That's Negative Opinion!
    - **Result:** ✅ Passed
6. **Input: "The team was unresponsive, and the issue still isn't resolved."**
    - **Expected Output:** Negative
    - **Model Output:** That's Negative Opinion!
    - **Result:** ✅ Passed
7. **Input: "I'm really impressed with how fast and efficient the system is!"**
    - **Expected Output:** Positive
    - **Model Output:** That's Positive Opinion!
    - **Result:** ✅ Passed

Test Case Insights:

- **Accuracy:** Out of 7 test cases, 6 were correctly classified. The model exhibited high accuracy in classifying general opinions, especially regarding simple positive and negative feedback.
- **Challenges:** The third test case demonstrates the model's difficulty in identifying more complex negative sentiments that are wrapped in positive-sounding language (like "revenge for Palestine"). This suggests that while the model performs well in binary sentiment analysis, it may require further fine-tuning or a more context-aware model to handle nuanced or emotionally complex language.
- **Overall Performance:** The deployed model performed reliably for straightforward inputs but may need enhancement to handle subtle or indirect sentiment, particularly in socially sensitive topics.

Samples From Test Cases 2



This is the worst experience I've had, and I regret choosing this option.

That's Negative Opinion!

The team was unresponsive, and the issue still isn't resolved.

That's Negative Opinion!

I'm really impressed with how fast and efficient the system is!

That's Positive Opinion!

I find the design confusing and hard to use, it's really frustrating.

That's Negative Opinion!

I absolutely love the new design! It's fantastic and user-friendly.

That's Positive Opinion!

<h1>happy</h1>

That's Positive Opinion!

Conclusion

This project successfully delivered a comprehensive sentiment classification system, integrating a fine-tuned **DistilBERT** model with a **Django** web application and a user-friendly **Gradio** interface. The classifier is designed to predict whether text contains a positive or negative sentiment, making it ideal for analyzing feedback, opinions, and other textual data. Here's a summary of what was achieved:

1. Model Fine-Tuning:

The **DistilBERT** model was fine-tuned using the Amazon dataset to classify sentiments as positive or negative. Through effective training and testing, the model achieved high accuracy in identifying sentiments from a variety of text inputs.

2. Gradio Interface for Instant Predictions:

A **Gradio** interface was integrated to provide an easy-to-use platform for real-time sentiment prediction. With just a text input, users can receive immediate classification results (positive or negative). Gradio's simplicity, coupled with the power of the fine-tuned model, makes this system accessible even to non-technical users.

3. Django Web Application for Deployment:

The sentiment classification model was successfully deployed in a Django web app. The system is capable of:

- Rendering a **clean and intuitive HTML template** for user interaction.
- Handling **POST requests** to process input data and return predictions from the model.
- Providing sentiment classification results in real-time, demonstrating how machine learning models can be seamlessly integrated into web applications.

4. Practical Testing and Evaluation:

The project was thoroughly tested using various types of textual inputs, and the model consistently produced accurate sentiment predictions. However, the test cases showed that while the model performs well in most cases, it might require further refinement for more nuanced or ambiguous inputs.

5. Future Improvements:

While the system is functional and performs effectively, some areas could be improved:

- **Contextual Understanding**: Enhance the model's ability to understand deeper context, especially in complex or ambiguous sentences.
- **Multi-class Classification**: Introduce a "Neutral" category to capture text that does not fall squarely into positive or negative sentiment.
- **Model Upgrades**: Fine-tuning more advanced models like **RoBERTa** or **GPT-based** models could further boost performance, especially for handling subtle sentiments.