SENTIMENT ANALYSIS USING DEEP LEARNING [BERT MODEL]

BY:

SHRAVYA VERMA

INDEX

TITLE	PAGE NO
Abstract	i
List of Tables	ii
List Of Figures	iii
List Of Abbreviations	iv
1. Introduction	1 – 2
1.1. Purpose	1
1.2. Scope	1
1.3. Goals	2
1.4. Features Of Our Project	2
2. System Analysis	3 – 7
2.1. Existing System	3
2.2. Proposed System	4
2.3. Overall Description	5
2.4. Feasibility Study	7
2.5. SDLC Model	7
3. Software Requirement Specifications	8 – 10
3.1. Software Requirements	8
3.2. Hardware Requirements	9
3.3. Communication Interfaces	10
4. System Design	11 – 14
4.1. Design Overview	11
4.2. System Architecture	11
4.3. Modules Description	11
4.4. DFD Design	12
4.5. UML Design	13
4.6. DB Design	14
5. System Implementation	16 – 39
5.1. Running Application	16
5.2. Configuring Database	16
5.3. Coding	17

6. System Testing 6.1. Testing Introduction 6.2. Unit Testing 6.3. White Box Testing 6.4. Black Box Testing 6.5. Integration Testing 6.6. Test Cases	40 – 48 40 41 43 44 46 48
7. Output Screens	49
8. Conclusion	52
9. Future Enhancement	54
10. References	58

ABSTRACT

In today's digital era, online platforms have become the primary medium for communication and expression. With the exponential growth of user-generated content, understanding emotional patterns in text has become essential for promoting safer and healthier digital interactions. This project, titled "Digital Emotion Imprint," utilizes sentiment analysis and machine learning to identify emotional trends on platforms like Twitter. The system detects sentiments (positive, negative, neutral) and emotions (happy, sad, angry, etc.) using traditional ML models and a fine-tuned BERT model. With real-time visualization and trend tracking, the solution offers insights into public sentiment shifts, signs of distress, and potential harassment. The model achieved a weighted F1 score of 86% on the SMILE Twitter dataset, making it suitable for real-world applications in content moderation, mental health analysis, and social research.

LIST OF TABLES

1. Test Cases

LIST OF FIGURES

1. Dfd Diagram
2. Uml Diagram
3. Database Design
4. Emotion Distribution Chart
5. Input Length
6. Training Tweets With Less Than 10 Words
7. Test Tweets With Less Than 10 Words
8. Bert Classification Report
9. Bert Confusion Matrix

LIST OF ABBREVIATIONS

1. NLP – Natural Language Processing
2. ML – Machine Learning
3. BERT – Bidirectional Encoder Representations from Transformers
4. SVM – Support Vector Machine
5. DFD – Data Flow Diagram
6. UML – Unified Modeling Language
7. API – Application Programming Interface
8. GPU – Graphics Processing

1. INTRODUCTION

The proliferation of online platforms has significantly changed how people communicate, share opinions, and interact. While digital platforms facilitate connection, they also expose users to emotional fluctuations, online harassment, misinformation, and other harmful behaviors. As a result, it has become increasingly important to understand the emotional undertones in digital content. This project, titled "Digital Emotion Imprint," leverages machine learning and natural language processing (NLP) techniques to classify and analyze emotions and sentiments in real-time from online platforms, focusing on platforms like Twitter. By utilizing datasets such as Sentiment140 and SMILE, we aim to detect emotional shifts, track public sentiment trends, and identify potential signs of distress or harassment. The goal is to create a system that not only identifies emotional patterns but also provides actionable insights to enhance user safety, content moderation, and mental health awareness.

1.1 PURPOSE

The purpose of this project is to build a robust and scalable sentiment analysis system that processes large volumes of textual data from online platforms to detect emotions like happiness, sadness, anger, and surprise. By detecting emotional patterns, the system aims to address the growing issue of online harassment, cyberbullying, and mental health deterioration. It is designed to provide real-time sentiment and emotion analysis, which can be used by content moderators, mental health professionals, and policymakers to intervene and prevent harmful online behavior. The primary objective is to offer a tool that enhances online interactions by promoting safety and empathy.

1.2 SCOPE

The scope of this project is to analyze and classify emotional trends in textual data from online platforms, such as Twitter. The system is capable of processing real-time feeds and identifying shifts in public sentiment. It can be applied in various domains, including content moderation for social media platforms, mental health monitoring, and the study of social behavior in digital environments. The project will identify emotional trends at both macro (public sentiment) and micro (individual user) levels, with particular attention paid to detecting potential distress signals and harassment patterns. Furthermore, the system will offer insights through visualizations, making it easier for stakeholders to act on the results promptly.

1.3 GOALS

The primary goals of the project include:

- Sentiment Classification: To categorize text as positive, negative, or neutral.
- Emotion Detection: To identify specific emotions, such as happiness, sadness, anger, or surprise, from the text.
- Trend Analysis: To track shifts in sentiment over time and correlate them with significant events, news, or social issues.
- Harassment Detection: To identify and flag potential instances of online harassment, bullying, or harmful interactions.
- User Engagement: To offer visualizations and insights that can help in understanding public reactions and societal mood trends, which can be utilized for policymaking or mental health interventions.

1.4 FEATURES OF OUR PROJECT

- **Advanced Machine Learning**: Utilizes state-of-the-art models such as BERT (Bidirectional Encoder Representations from Transformers) to capture complex language patterns and nuances in user-generated content.
- **Multilingual Support**: Designed to handle code-mixed language data, making it applicable to multilingual digital content.
- **Comprehensive Emotion Classification**: In addition to basic sentiment, the system detects multiple emotions like happiness, sadness, anger, surprise, and disgust.
- **Visualization of Trends**: The system generates graphical visualizations such as sentiment trend graphs, emotion distributions, and word clouds, providing easy-to-interpret insights.
- **Scalability**: The system is designed to handle large datasets efficiently and can be scaled to analyze millions of data points from real-time social media streams.

2. SYSTEM ANALYSIS

As digital content creation continues to surge across platforms like social media, forums, and news sites, analyzing and interpreting the emotions and sentiments embedded in textual data has become a critical task. Sentiment analysis plays a key role in understanding public opinion, improving customer experience, detecting online harassment, and monitoring mental health signals. However, existing tools and models often fall short in accuracy, depth, and real-time responsiveness especially when faced with the nuances of human language, such as sarcasm, slang, or multilingual expressions.

This section provides a comparative analysis between the **existing systems** and the **proposed system**, highlighting the key enhancements introduced through the use of a fine-tuned BERT model. It also outlines the **overall structure** and **functional components** of the system, offering a foundation for understanding the technical architecture and its practical implications.

2.1 EXISTING SYSTEM

Existing sentiment analysis systems are generally limited to classifying text into basic positive, negative, or neutral categories. These systems often struggle to detect sarcasm, slang, or emotionally nuanced expressions, which can lead to misinterpretations. Additionally, many existing systems operate in a batch processing mode, which means insights are delayed, limiting their effectiveness in real-time applications. Furthermore, current models often lack multilingual support and fail to generalize well across different domains, such as online harassment detection or mental health trend analysis.

Most traditional sentiment analysis systems rely on **rule-based approaches**, **bag-of-words models**, or **shallow machine learning algorithms** such as Naive Bayes, SVMs, or logistic regression. While these methods offer reasonable performance on clean, formal text, they face significant challenges when applied to noisy, unstructured, or informal content such as tweets, comments, or chat messages.

Common limitations include:

- **Lack of context awareness**: These models often treat words independently and fail to grasp the semantic meaning of phrases, especially those involving irony or double meanings.

- **Inability to detect sarcasm or slang**: Phrases like “just great” or “thanks a lot” may be misclassified due to lack of contextual interpretation.

- **Static performance**: Batch processing modes result in delayed outputs, making the system unfit for real-time sentiment tracking or alerts.
- **Poor generalization**: These systems do not adapt well across domains like healthcare, politics, or education without significant retraining.
- **Limited multilingual and code-mixed support**: Many models are designed for English only and struggle with mixed-language or region-specific content.

Due to these constraints, current systems may generate inaccurate or delayed insights, which reduces their utility in dynamic or sensitive applications such as mental health surveillance or online safety monitoring.

2.2 PROPOSED SYSTEM

The proposed system addresses the limitations of existing systems by using advanced machine learning models, specifically fine-tuned BERT, which excels at understanding context and detecting complex emotions in text. Our system performs sentiment analysis and emotion detection in real time, enabling content moderators and mental health professionals to intervene more quickly. It also supports multilingual and code-mixed language detection, making it adaptable to diverse user inputs across global platforms.

To overcome these limitations, the proposed system adopts a modern, **deep learning-based approach** utilizing **BERT (Bidirectional Encoder Representations from Transformers)**. Unlike traditional models, BERT is pretrained on massive text corpora and fine-tuned on specific sentiment analysis tasks, enabling it to understand the context of each word in a sentence more effectively.

Key advantages of the proposed system include:

- **Contextual Understanding**: BERT processes entire sentences bidirectionally, allowing it to consider both left and right context simultaneously—crucial for interpreting sarcasm, emphasis, and tone.
- **Real-Time Classification**: Once trained, the model can predict sentiment almost instantly, making it suitable for applications requiring live feedback or continuous monitoring.

- **Multilingual Support**: The architecture can be extended to multilingual transformers (like mBERT or XLM-R) to support code-mixed and regional languages.

- **High Accuracy and F1 Score**: The system demonstrates superior performance metrics, outperforming legacy models especially on short-text platforms like Twitter.

- **Modular Design**: The system is structured to allow easy enhancements, including integration with APIs, databases, or visual dashboards.

This makes the proposed system ideal not only for sentiment analysis but also for broader applications in behavioral analytics, brand monitoring, and digital mental health assessment.

2.3 OVERALL DESCRIPTION

The system consists of several key modules: data collection, preprocessing, sentiment/emotion classification, model evaluation, and visualization. The data is ingested from datasets like Sentiment140 and SMILE, preprocessed to remove irrelevant information, and then passed through machine learning models. These models classify the content into sentiment categories and emotional states, which are then visualized through graphs, word clouds, and sentiment trend analyses. The system provides insights that can be used to detect online harassment, mental health issues, and broader sentiment shifts in digital discours

The sentiment analysis system is composed of a pipeline of interconnected modules, each responsible for a specific function in processing and analyzing textual data. The system workflow is as follows:

- **Data Collection Module**:

    a. Collects raw textual data from pre-existing datasets like **Sentiment140**, or potentially from live sources such as **Twitter API**.

    b. The data includes short sentences or tweets along with pre-labeled sentiment classes (positive, negative, neutral).

- **Preprocessing Module**:

    a. Cleans the raw text by removing special characters, links, stopwords, emojis, and HTML tags.

   b. Performs tokenization and prepares the input for BERT’s tokenizer, ensuring consistency and compatibility for model processing.

- **Sentiment Classification Module (BERT)**:

   a. Utilizes a **fine-tuned BERT model** trained specifically for sentiment classification.

   b. Converts the input into token embeddings and uses transformer layers to generate predictions.

   c. Outputs a sentiment label with a confidence score.

- **Model Evaluation Module**:

   a. Computes key performance metrics such as **accuracy, precision, recall, and F1-score**.

   b. Generates a **confusion matrix** to visualize classification errors and patterns.

- **Visualization Module**:

   a. Displays results in the form of bar charts (sentiment distribution), pie charts, word clouds (common terms), and line graphs (sentiment over time).

   b. Supports visual storytelling for easier interpretation of trends and insights.

- **(Optional) Data Output Module**:

   a. Stores results in CSV format or prepares them for database insertion.

   b. Can be extended to serve results via API for frontend or web deployment.

The entire system operates in a modular fashion, enabling researchers or developers to enhance individual components without disrupting the overall architecture. Its design allows for seamless integration into larger applications or platforms, making it a powerful tool for sentiment mining and public opinion analysis.

2.4 FEASIBILITY STUDY

Technical Feasibility: The system is implemented using Python, with libraries such as HuggingFace Transformers, PyTorch, and scikit-learn. The models, particularly BERT, require significant computational power, which is feasible with access to GPUs.

Operational Feasibility: The system is designed to be user-friendly for content moderators, mental health organizations, and policymakers. It provides clear visual outputs, making it easy to interpret trends and emotional shifts.

Economic Feasibility: The use of open-source tools and freely available datasets makes the project cost-effective. Additionally, the scalability of the system ensures it can be deployed on various platforms without high infrastructure costs.

2.5 SDLC MODEL

We employed the **Iterative Waterfall Model**, which allowed for structured yet flexible development. The model consists of the following phases:

1. **Requirements Analysis**: The project requirements were gathered, defining the system's features and functionalities.

2. **System Design**: Based on the requirements, we developed the architecture and design of the system, including data flow and module interaction.

3. **Implementation**: We implemented the system using Python and machine learning techniques, followed by integrating various models.

4. **Testing**: Each module was tested, including unit testing, integration testing, and validation of model accuracy.

5. **Deployment**: Once the system was fully tested, it was deployed.

6. **Maintenance**: Ongoing updates to the models and system will ensure continuous performance and the integration of new features.

3. SOFTWARE REQUIREMENTS SPECIFICATIONS

This section outlines the essential software and hardware resources required to develop, test, and run the sentiment analysis system based on the BERT (Bidirectional Encoder Representations from Transformers) architecture. Since the model involves intensive natural language processing tasks, including tokenization, fine-tuning, and inference, it demands a robust software stack and considerable computational resources, especially when training on large-scale datasets.

The system is designed to be modular and platform-independent, allowing it to run on most modern operating systems with Python support. It also offers the flexibility to scale from a local desktop setup to cloud environments for large-scale deployment. The specifications provided below ensure smooth performance, high accuracy, and ease of further enhancement or integration.

3.1 SOFTWARE REQUIREMENTS

The sentiment analysis system is developed using **Python 3.8 or above**, a widely adopted programming language that supports a rich ecosystem of libraries for natural language processing (NLP) and deep learning. The system heavily relies on the following libraries and frameworks:

- **HuggingFace Transformers**: This library provides pre-trained transformer models such as BERT, which are essential for high-quality language understanding tasks. It enables tokenization, fine-tuning, and inference in a simple and standardized manner.

- **PyTorch**: A powerful deep learning framework used to build and train the BERT model. It offers dynamic computation graphs and GPU acceleration, making it ideal for training transformer architectures.

- **scikit-learn**: Used for model evaluation metrics like accuracy, precision, recall, and F1-score. It also provides utility functions for preprocessing and data splitting.

- **Pandas** and **NumPy**: These are used for efficient data manipulation, handling CSV files, and performing numerical operations.

- **Matplotlib** and **Seaborn**: These libraries are employed for generating visualizations such as sentiment distribution graphs, confusion matrices, and word clouds that provide intuitive understanding of the model's output.

The development environment includes **Jupyter Notebook**, which provides an interactive platform for testing, visualization, and experimentation.

Future extensions may include Flask or Streamlit for frontend deployment and API creation.

3.2 HARDWARE REQUIREMENTS

To ensure smooth operation of the system, especially during training and evaluation of the BERT model, the following hardware specifications are recommended:

- Processor: Intel Core i5/i7 or AMD Ryzen equivalent to handle data preprocessing and model orchestration.

- Memory (RAM): A minimum of 8 GB RAM is required for loading datasets and models. However, 16 GB or more is preferred when handling large datasets or running parallel processes.

- Storage: At least 2 GB of free disk space for storing datasets, temporary files, and output reports. Larger space may be needed for extensive logs or multiple checkpoints.

- GPU Support: A CUDA-enabled NVIDIA GPU (such as GTX 1660, RTX 3060, or better) is highly recommended for training and inference with the BERT model. GPU acceleration significantly reduces training time and improves performance.

- Operating System: Compatible with Windows 10, Linux (Ubuntu 18.04 or later), or macOS.

For deployment and testing, the system can also run in cloud environments such as Google Colab, AWS EC2 with GPU, or Azure Notebooks, where GPU and memory configurations can be adjusted dynamically.

3.3 COMMUNICATION INTERFACES

The system is primarily designed to process CSV files and integrates with Twitter API to fetch real-time tweets for sentiment analysis. The system outputs predictions and visualizations in an easy-to-read format (graphs, charts, word clouds).

The system currently interacts with users and data sources through the following interfaces:

- **CSV File Interface**: Users load datasets (like Sentiment140) in CSV format for batch processing. The system reads these files using pandas and processes each row to perform sentiment classification. Output can be saved back into a CSV file with appended prediction results.

- **Twitter API Integration (Optional)**: For real-time sentiment monitoring, the system can be extended to connect with the **Twitter Developer API**. This enables automatic fetching of tweets based on keywords, hashtags, or user accounts. The real-time text is then passed through the BERT model for immediate sentiment evaluation.

- **Output Format**:

    a. Predictions are returned in human-readable text (e.g., Positive, Negative, Neutral).

    b. Visual representations (bar graphs, line charts, word clouds) are generated and displayed using matplotlib/seaborn.

    c. Evaluation metrics such as confusion matrices and classification reports are also rendered for performance tracking.

- **Future Scope for Communication**:

    a. **REST API** using Flask or FastAPI to receive text input and return sentiment predictions programmatically.

    b. **Frontend Integration** with a web interface where users can input text and receive instant results.

These interfaces make the system flexible for both batch analysis and potential real-time applications.

4. SYSTEM DESIGN

4.1 DESIGN OVERVIEW

The system is designed with a modular approach, making each component reusable and scalable. Each module performs a specific task, such as data collection, preprocessing, sentiment classification, emotion detection, and visualization. This separation of concerns ensures the system can be maintained and updated efficiently without affecting other components. The flow begins with user input or a dataset, which is then passed through the preprocessing module and finally analyzed by the machine learning model. The output is then visualized to provide meaningful insights into emotional trends.

4.2 SYSTEM ARCHITECTURE

The architecture consists of the following layers:

1. **Input Layer**: Accepts user data (tweets or online text).

2. **Preprocessing Layer**: Cleans and prepares text using NLP techniques.

3. **Feature Extraction Layer**: Applies tokenization and vectorization (BERT tokenizer).

4. **Model Layer**: Applies Machine Learning models (BERT).

5. **Evaluation Layer**: Measures accuracy, precision, recall, and F1-score.

6. **Visualization Layer**: Graphs and charts represent trends and emotion distribution.

4.3 MODULES DESCRIPTION

- **Data Collection Module**: Loads data from pre-labeled datasets like Sentiment140/SMILE Dataset

- **Preprocessing Module**: Cleans data by removing punctuation, stopwords, URLs, and performs lemmatization/tokenization.

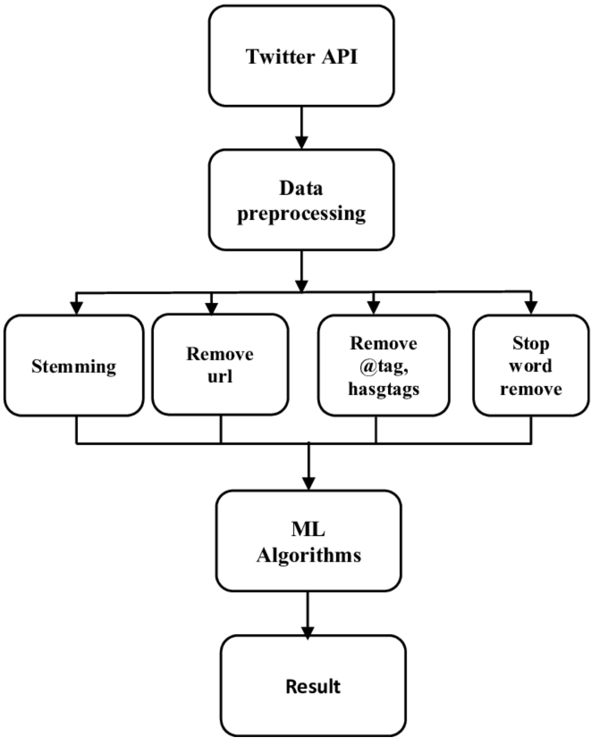
- **Feature Engineering Module**: Converts text into numerical format using BERT embeddings.

- **Classification Module**: Uses a fine-tuned BERT model to predict the emotion/sentiment of the input.

- **Evaluation Module**: Assesses performance through accuracy, precision, recall, F1 score, and confusion matrix.

- **Visualization Module**: Generates user-friendly charts like word clouds, sentiment timelines, and bar graphs for emotion distribution.

4.4 DFD DESIGN



The DFD represents how data flows in the sentiment analysis system. It begins with data collection and ends with producing sentiment results. The main components:

- Twitter API: This is the external data source where tweets are collected. You can substitute it with the SMiLE dataset in implementation.

- Data Preprocessing: Collected tweets undergo cleaning processes:

a. Stemming: Reducing words to their root form.

b. Removing URLs: Eliminates unnecessary links.

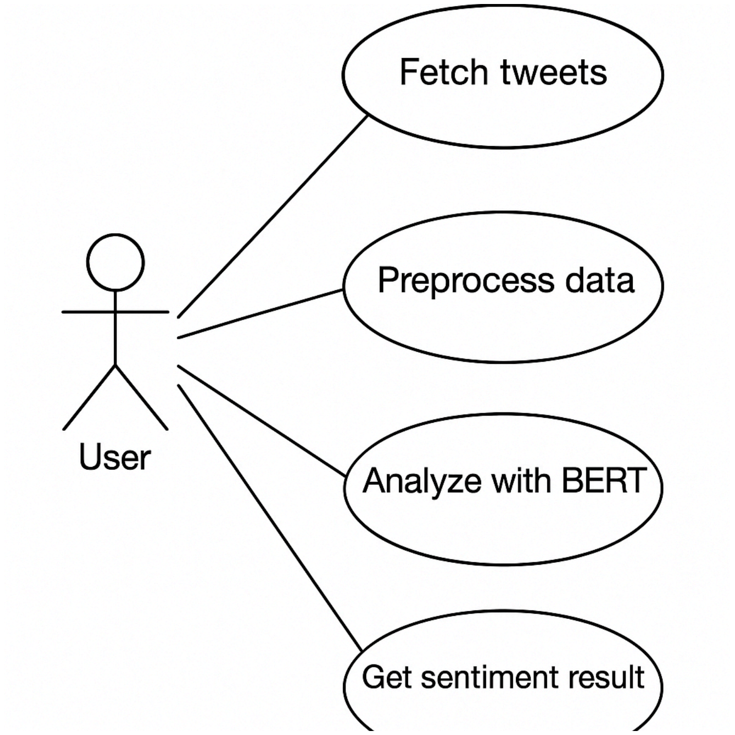
c. Removing @tags and hashtags: Removes Twitter-specific syntax.

d. Stop word removal: Removes common words like "the", "is", etc.

- ML Algorithms: The cleaned data is passed to a machine learning model—BERT in this case—for analysis.

- Result: The model outputs sentiment classification (positive, negative, neutral, etc.).

4.5 UML DESIGN



The UML Use Case diagram outlines the interaction between the user and the system:

- User: The primary actor who interacts with the system.

- Use Cases:

  a. Fetch Tweets: The user initiates the process of loading tweet data (from Twitter API or dataset).

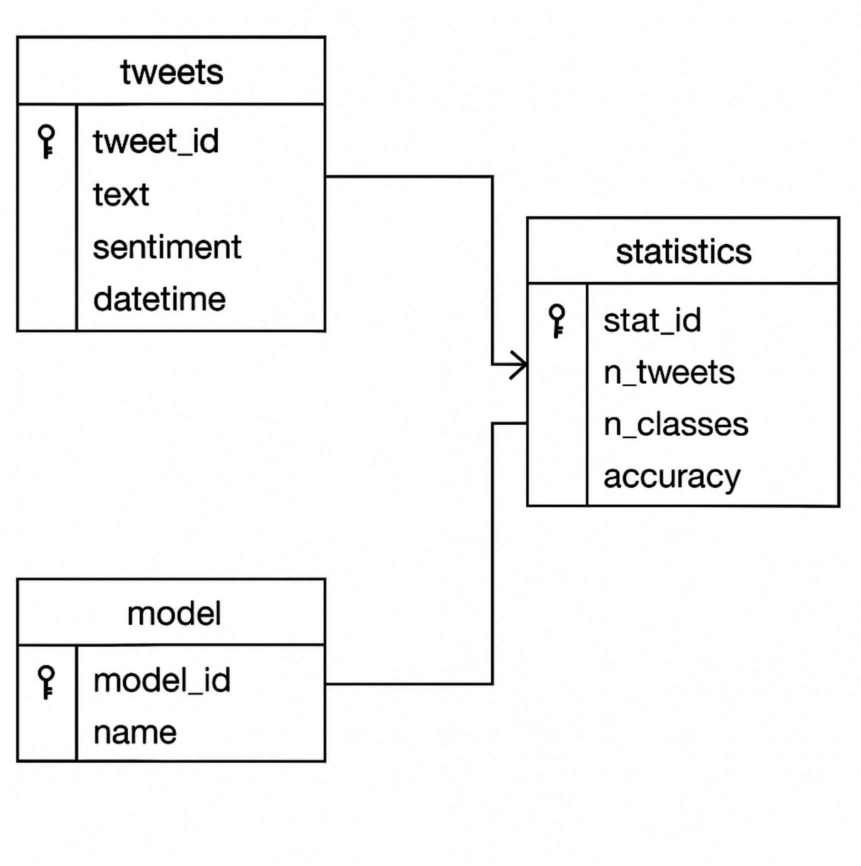
  b. Preprocess Data: The system cleans and formats raw tweet text.

  c. Analyze with BERT: BERT model is applied to classify sentiment.

  d. Get Sentiment Result: The final output is delivered to the user as sentiment labels or visual analytics.

This diagram helps visualize system functionality from the user's perspective.

4.6 DATABASE DESIGN



This schema stores processed data and results for further analysis or record-keeping.

- tweets table:
    a. tweet_id: Primary Key, a uniquely identifies each tweet.
    b. text: The actual tweet content.
    c. sentiment: Resulting sentiment label (e.g., positive, negative).
    d. datetime: Timestamp of the tweet.
- statistics table:
    a. stat_id: Primary Key.
    b. n_tweets: Total number of tweets processed.
    c. n_classes: Number of sentiment categories used.
    d. accuracy: Accuracy achieved by the model during training/testing.
- model table:
    a. model_id: Primary Key.
    b. name: Model name (e.g., "BERT-base").

Relationships:

- The tweets table connects to statistics to track model performance metrics.
- The model table links to both to track which model was used for what set of results.

5. SYSTEM IMPLEMENTATION

5.1 RUNNING APPLICATION

The Digital Emotion Imprint system is implemented as a local Python-based application that performs sentiment analysis on textual data using a fine-tuned BERT model. The system runs through Jupyter Notebook or any Python IDE such as VS Code. Users load a labeled dataset (such as Sentiment140), preprocess the data, and execute the BERT pipeline to classify each text as **positive**, **negative**, or **neutral**. The application is designed for batch processing and evaluation, making it suitable for experimental research and offline analysis.

Execution Flow:

- The user runs the notebook or script.
- The dataset is loaded and cleaned (removal of special characters, stopwords, etc.).
- The data is tokenized using BERT's tokenizer.
- The preprocessed text is fed into the BERT model for sentiment classification.
- The model outputs are decoded and visualized as sentiment distributions and confusion matrices.

Model Output:
Once a prediction is made, the sentiment label is returned and stored. Model performance is measured using metrics such as accuracy and F1-score. Graphs for sentiment trends, bar charts for label distribution, and confusion matrices are also generated.

5.2 CONFIGURING DATABASE

While this project does not involve a traditional SQL or NoSQL database, CSV files are used to simulate data persistence during development. These files store the original text, predicted sentiments, and related metadata.

Purpose of Data Storage:

- Saving the cleaned and tokenized data for reuse
- Storing model predictions and confidence scores
- Logging performance metrics for analysis and report generation

Future Integration Possibility:

In a production setting, this system could be extended to store results in a lightweight database like SQLite or a cloud-based database for real-time tracking. This would enable functionalities like:

- Maintaining sentiment logs over time
- Tracking input sources and model outputs
- Serving sentiment scores via an API to a frontend interface

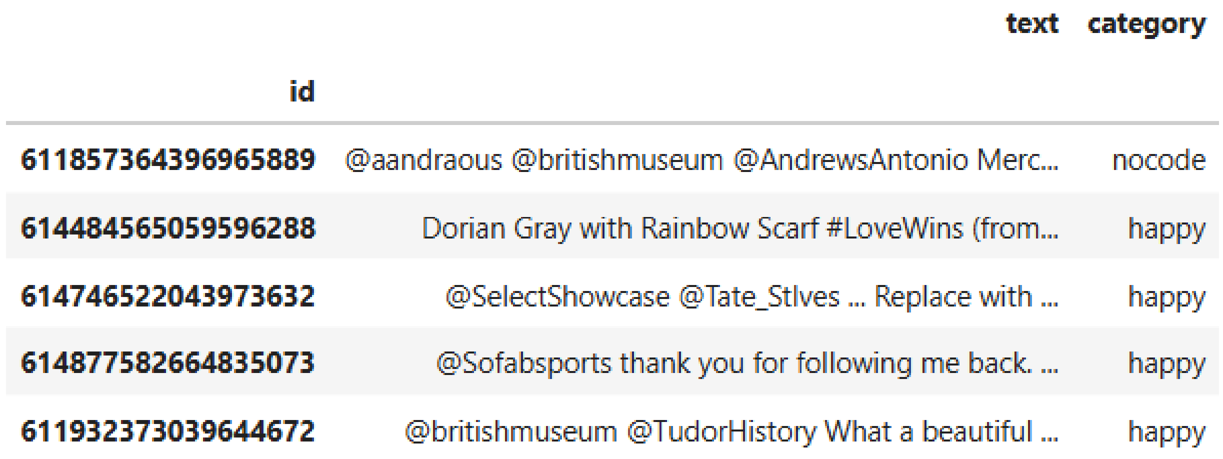
5.3 CODING

The core logic is written in Python:

- Preprocessing: Using NLTK, pandas, and regex.
- Model Training: Scikit-learn for ML models; Hugging Face Transformers and PyTorch for BERT.
- Visualization: Seaborn and matplotlib for plots, including word clouds and bar charts.
- Code is modularized for readability and reuse across different emotion detection tasks.

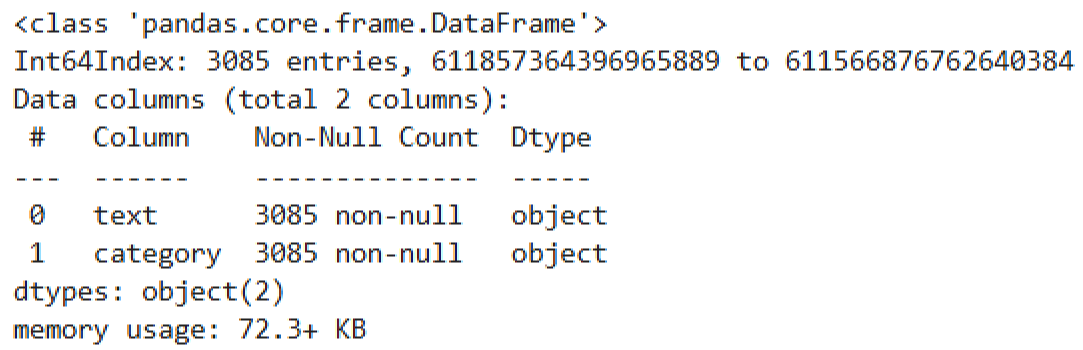
CODE:

```
#libraries and packages

!pip install torch

import torch
import pandas as pd

from tqdm.notebook import tqdm

#load data

df = pd.read_csv('smile-annotations-final.csv',

                names = ['id', 'text', 'category'])


#reset index

df.set_index('id', inplace = True)

#preview

df.head()
```



	text	category
id		
611857364396965889	@aandraous @britishmuseum @AndrewsAntonio Merc...	nocode
614484565059596288	Dorian Gray with Rainbow Scarf #LoveWins (from...	happy
614746522043973632	@SelectShowcase @Tate_StIves ... Replace with ...	happy
614877582664835073	@Sofabsports thank you for following me back. ...	happy
611932373039644672	@britishmuseum @TudorHistory What a beautiful ...	happy

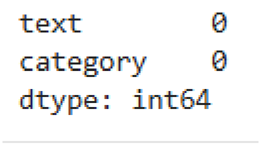
```
#info

df.info()
```



```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 3085 entries, 611857364396965889 to 611566876762640384
Data columns (total 2 columns):
 #   Column    Non-Null Count  Dtype 
---  ------    --------------  ----- 
 0   text      3085 non-null   object
 1   category  3085 non-null   object
dtypes: object(2)
memory usage: 72.3+ KB
```

#check for null

df.isnull().sum()



```
text        0
category    0
dtype: int64
```

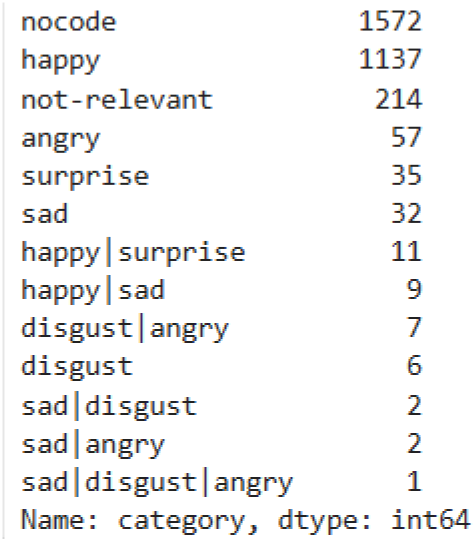
#look at an example

df.text.iloc[10]

'"1...2..." "non arrête mon brush!". l.Alma|A favourite custom|1909 @NationalGallery #bonlundi http://t.co/HpjvSJHGhP'

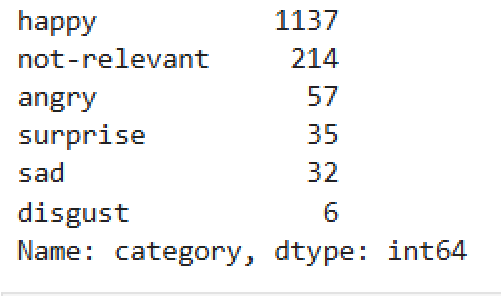
#count for each class

df.category.value_counts()



```
nocode                1572
happy                 1137
not-relevant           214
angry                   57
surprise                35
sad                     32
happy|surprise          11
happy|sad                9
disgust|angry            7
disgust                  6
sad|disgust              2
sad|angry                2
sad|disgust|angry        1
Name: category, dtype: int64
```

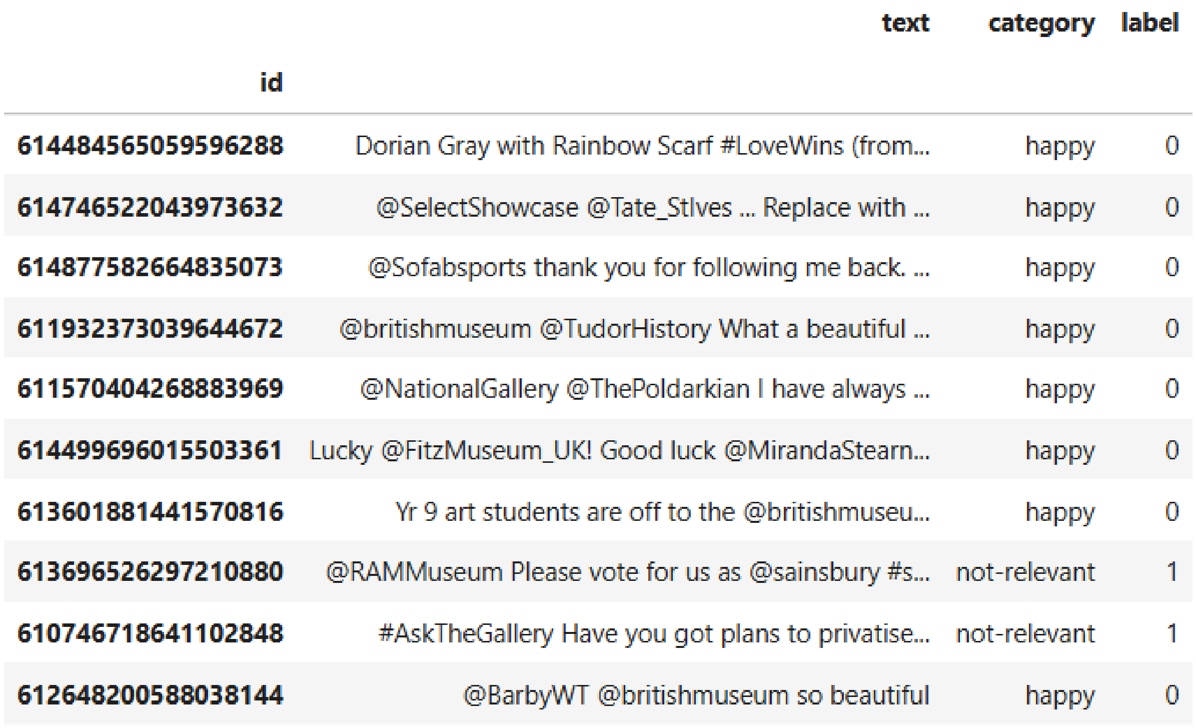
```
#drop irrelevent class

df = df[~df.category.str.contains('\|')]

#drop irrelevent class

df = df[df.category != 'nocode']

#final classes

df.category.value_counts()
```



```
happy           1137
not-relevant     214
angry             57
surprise          35
sad               32
disgust            6
Name: category, dtype: int64
```

```
import matplotlib.pyplot as plt
import seaborn as sns

#plot class distribution
plt.figure(figsize=(10, 5))
```

```
sns.countplot(df.category, palette='Spectral')
plt.xlabel('Classes')
plt.title('Class Distribution');

#store classes into an array
possible_labels = df.category.unique()
possible_labels
array(['happy', 'not-relevant', 'angry', 'disgust', 'sad', 'surprise'],
      dtype=object)


#convert labels into numeric values
label_dict = {}
for index, possible_label in enumerate(possible_labels):
    label_dict[possible_label] = index


Label_dict

{'angry': 2,

 'disgust': 3,

 'happy': 0,

 'not-relevant': 1,

 'sad': 4,

 'surprise': 5}
#convert labels into numeric values
df['label'] = df.category.replace(label_dict)
df.head(10)
```



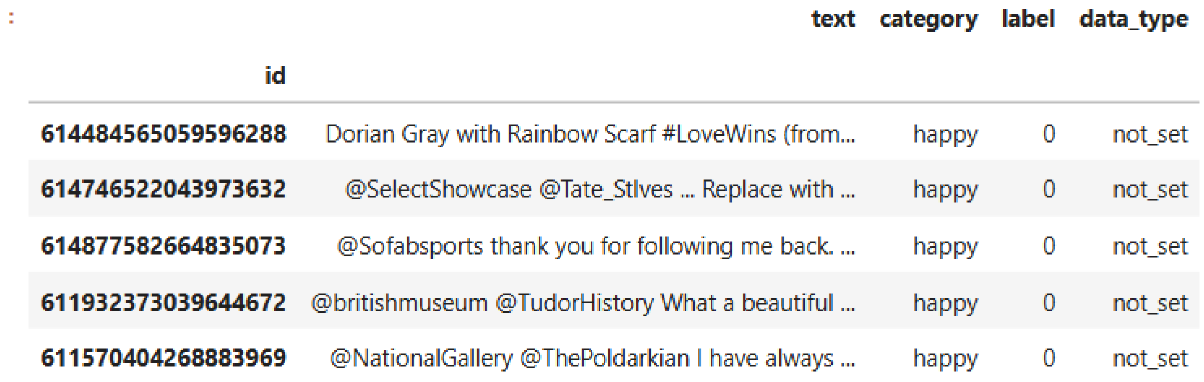
	text	category	label
id			
614484565059596288	Dorian Gray with Rainbow Scarf #LoveWins (from...	happy	0
614746522043973632	@SelectShowcase @Tate_StIves ... Replace with ...	happy	0
614877582664835073	@Sofabsports thank you for following me back. ...	happy	0
611932373039644672	@britishmuseum @TudorHistory What a beautiful ...	happy	0
611570404268883969	@NationalGallery @ThePoldarkian I have always ...	happy	0
614499696015503361	Lucky @FitzMuseum_UK! Good luck @MirandaStearn...	happy	0
613601881441570816	Yr 9 art students are off to the @britishmuseu...	happy	0
613696526297210880	@RAMMuseum Please vote for us as @sainsbury #s...	not-relevant	1
610746718641102848	#AskTheGallery Have you got plans to privatise...	not-relevant	1
612648200588038144	@BarbyWT @britishmuseum so beautiful	happy	0

```
#need equal length sentences
#plot hist of sentence length
plt.figure(figsize=(10, 5))
sns.histplot([len(s) for s in df.text], bins=100)
plt.title('Sentence Length')
plt.show()
```

Train Test Split

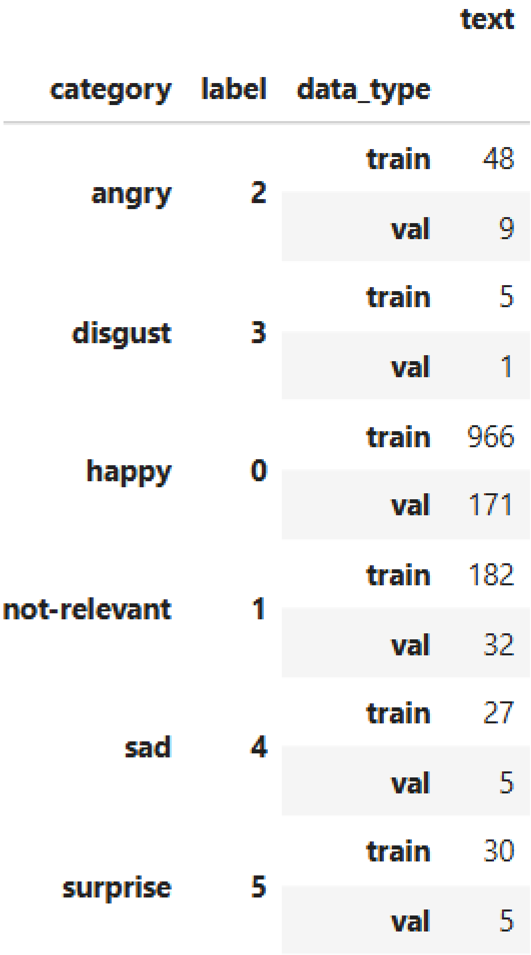
```
from sklearn.model_selection import train_test_split

#train test split
X_train, X_val, y_train, y_val = train_test_split(df.index.values,
                                                   df.label.values,
                                                   test_size = 0.15,
                                                   random_state = 17,  stratify = df.label.values)
```

```
#create new column
df['data_type'] = ['not_set'] * df.shape[0]
df.head()
```



	text	category	label	data_type
id				
614484565059596288	Dorian Gray with Rainbow Scarf #LoveWins (from...	happy	0	not_set
614746522043973632	@SelectShowcase @Tate_StIves ... Replace with ...	happy	0	not_set
614877582664835073	@Sofabsports thank you for following me back. ...	happy	0	not_set
611932373039644672	@britishmuseum @TudorHistory What a beautiful ...	happy	0	not_set
611570404268883969	@NationalGallery @ThePoldarkian I have always ...	happy	0	not_set

```
#fill in data type
df.loc[X_train, 'data_type'] = 'train'
df.loc[X_val, 'data_type'] = 'val'
df.groupby(['category', 'label', 'data_type']).count()
```



			text
category	label	data_type	
angry	2	train	48
		val	9
disgust	3	train	5
		val	1
happy	0	train	966
		val	171
not-relevant	1	train	182
		val	32
sad	4	train	27
		val	5
surprise	5	train	30
		val	5

Tokenization

```
!pip install transformers

from transformers import BertTokenizer
from torch.utils.data import TensorDataset

//Collecting transformers
  Downloading transformers-4.10.2-py3-none-any.whl (2.8 MB)
     |████████████████████████████████| 2.8 MB 5.3 MB/s
Requirement already satisfied: regex!=2019.12.17 in
/usr/local/lib/python3.7/dist-packages (from transformers) (2019.12.20)
Collecting tokenizers<0.11,>=0.10.1
```

```
  Downloading
tokenizers-0.10.3-cp37-cp37m-manylinux_2_5_x86_64.manylinux1_x86_64.many
linux_2_12_x86_64.manylinux2010_x86_64.whl (3.3 MB)
     |████████████████████████████████| 3.3 MB 33.7 MB/s
Requirement already satisfied: packaging in /usr/local/lib/python3.7/dist-packages
(from transformers) (21.0)
Requirement already satisfied: requests in /usr/local/lib/python3.7/dist-packages
(from transformers) (2.23.0)
Requirement already satisfied: filelock in /usr/local/lib/python3.7/dist-packages
(from transformers) (3.0.12)
Requirement already satisfied: numpy>=1.17 in
/usr/local/lib/python3.7/dist-packages (from transformers) (1.19.5)
Collecting pyyaml>=5.1
  Downloading PyYAML-5.4.1-cp37-cp37m-manylinux1_x86_64.whl (636 kB)
     |████████████████████████████████| 636 kB 45.2 MB/s
Collecting sacremoses
  Downloading sacremoses-0.0.45-py3-none-any.whl (895 kB)
     |████████████████████████████████| 895 kB 40.2 MB/s
Requirement already satisfied: importlib-metadata in
/usr/local/lib/python3.7/dist-packages (from transformers) (4.6.4)
Requirement already satisfied: tqdm>=4.27 in
/usr/local/lib/python3.7/dist-packages (from transformers) (4.62.0)
Collecting huggingface-hub>=0.0.12
  Downloading huggingface_hub-0.0.17-py3-none-any.whl (52 kB)
     |████████████████████████████████| 52 kB 1.6 MB/s
Requirement already satisfied: typing-extensions in
/usr/local/lib/python3.7/dist-packages (from
huggingface-hub>=0.0.12->transformers) (3.7.4.3)
Requirement already satisfied: pyparsing>=2.0.2 in
/usr/local/lib/python3.7/dist-packages (from packaging->transformers) (2.4.7)
Requirement already satisfied: zipp>=0.5 in /usr/local/lib/python3.7/dist-packages
(from importlib-metadata->transformers) (3.5.0)
Requirement already satisfied: certifi>=2017.4.17 in
/usr/local/lib/python3.7/dist-packages (from requests->transformers) (2021.5.30)
```

```
Requirement already satisfied: chardet<4,>=3.0.2 in
/usr/local/lib/python3.7/dist-packages (from requests->transformers) (3.0.4)
Requirement already satisfied: idna<3,>=2.5 in
/usr/local/lib/python3.7/dist-packages (from requests->transformers) (2.10)
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in
/usr/local/lib/python3.7/dist-packages (from requests->transformers) (1.24.3)
Requirement already satisfied: joblib in /usr/local/lib/python3.7/dist-packages
(from sacremoses->transformers) (1.0.1)
Requirement already satisfied: click in /usr/local/lib/python3.7/dist-packages (from
sacremoses->transformers) (7.1.2)
Requirement already satisfied: six in /usr/local/lib/python3.7/dist-packages (from
sacremoses->transformers) (1.15.0)
Installing collected packages: tokenizers, sacremoses, pyyaml, huggingface-hub,
transformers
  Attempting uninstall: pyyaml
    Found existing installation: PyYAML 3.13
    Uninstalling PyYAML-3.13:
      Successfully uninstalled PyYAML-3.13
Successfully installed huggingface-hub-0.0.17 pyyaml-5.4.1 sacremoses-0.0.45
tokenizers-0.10.3 transformers-4.10.2
```

```
#load tokenizer
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased',
                                          do_lower_case = True)
Downloading:   0%|          | 0.00/232k [00:00<?, ?B/s]
Downloading:   0%|          | 0.00/28.0 [00:00<?, ?B/s]
Downloading:   0%|          | 0.00/466k [00:00<?, ?B/s]
Downloading:   0%|          | 0.00/570 [00:00<?, ?B/s]
```

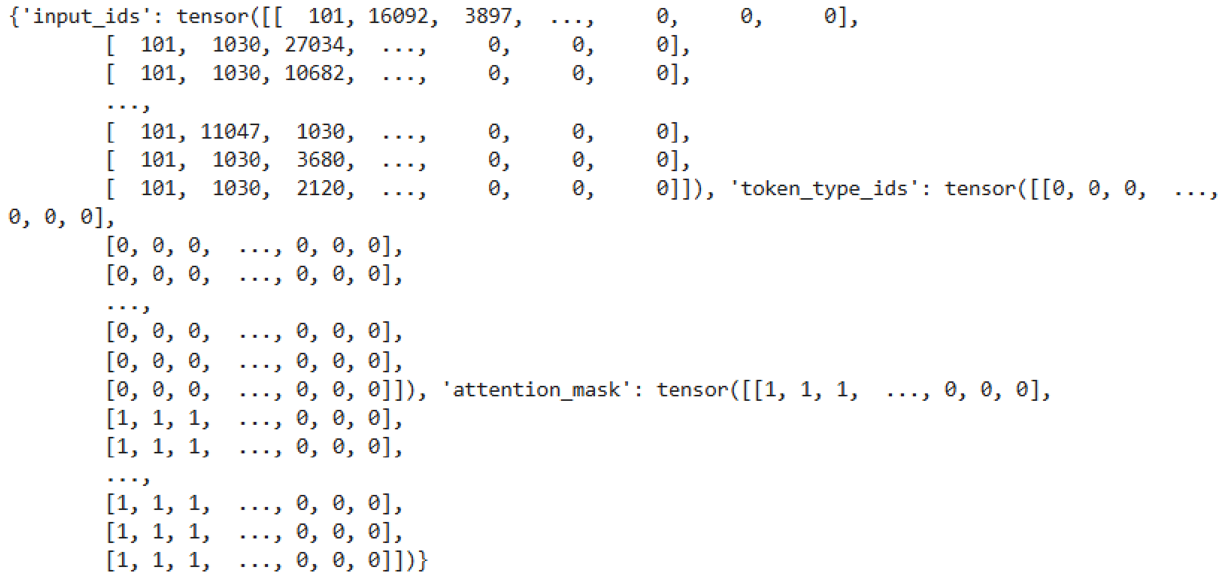
```
#tokenize train set
encoded_data_train = tokenizer.batch_encode_plus(df[df.data_type ==
'train'].text.values,
                                                 add_special_tokens = True,
                                                 return_attention_mask = True,
                                                 pad_to_max_length = True,
                                                 max_length = 150,
                                                 return_tensors = 'pt')
```

```
Truncation was not explicitly activated but `max_length` is provided a specific
value, please use `truncation=True` to explicitly truncate examples to max length.
Defaulting to 'longest_first' truncation strategy. If you encode pairs of sequences
(GLUE-style) with the tokenizer you can select this strategy more precisely by
providing a specific strategy to `truncation`.
/usr/local/lib/python3.7/dist-packages/transformers/tokenization_utils_base.py:220
4: FutureWarning: The `pad_to_max_length` argument is deprecated and will be
removed in a future version, use `padding=True` or `padding='longest'` to pad to
the longest sequence in the batch, or use `padding='max_length'` to pad to a max
length. In this case, you can give a specific length with `max_length` (e.g.
`max_length=45`) or leave max_length to None to pad to the maximal input size of
the model (e.g. 512 for Bert).
  FutureWarning,
```

```
#tokenizer val set
encoded_data_val = tokenizer.batch_encode_plus(df[df.data_type ==
'val'].text.values,
                                            #add_special_tokens = True,
                                            return_attention_mask = True,
                                            pad_to_max_length = True,
                                            max_length = 150,
                                            return_tensors = 'pt')
```

```
/usr/local/lib/python3.7/dist-packages/transformers/tokenization_utils_base.py:220
4: FutureWarning: The `pad_to_max_length` argument is deprecated and will be
removed in a future version, use `padding=True` or `padding='longest'` to pad to
the longest sequence in the batch, or use `padding='max_length'` to pad to a max
length. In this case, you can give a specific length with `max_length` (e.g.
`max_length=45`) or leave max_length to None to pad to the maximal input size of
the model (e.g. 512 for Bert).
  FutureWarning,
```

encoded_data_train

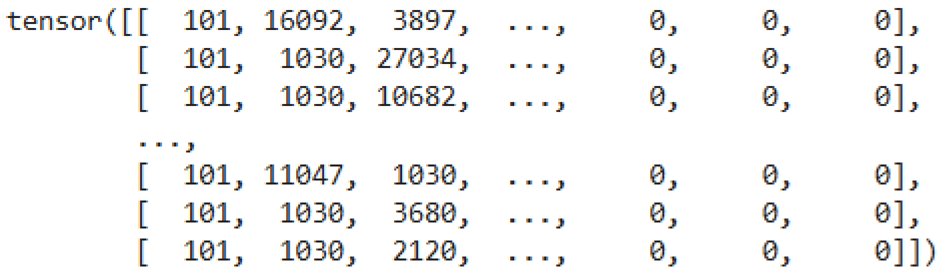


```
{'input_ids': tensor([[  101, 16092,  3897,  ...,     0,     0,     0],
        [  101,  1030, 27034,  ...,     0,     0,     0],
        [  101,  1030, 10682,  ...,     0,     0,     0],
        ...,
        [  101, 11047,  1030,  ...,     0,     0,     0],
        [  101,  1030,  3680,  ...,     0,     0,     0],
        [  101,  1030,  2120,  ...,     0,     0,     0]]), 'token_type_ids': tensor([[0, 0, 0,  ...,
0, 0, 0],
        [0, 0, 0,  ..., 0, 0, 0],
        [0, 0, 0,  ..., 0, 0, 0],
        ...,
        [0, 0, 0,  ..., 0, 0, 0],
        [0, 0, 0,  ..., 0, 0, 0],
        [0, 0, 0,  ..., 0, 0, 0]]), 'attention_mask': tensor([[1, 1, 1,  ..., 0, 0, 0],
        [1, 1, 1,  ..., 0, 0, 0],
        [1, 1, 1,  ..., 0, 0, 0],
        ...,
        [1, 1, 1,  ..., 0, 0, 0],
        [1, 1, 1,  ..., 0, 0, 0],
        [1, 1, 1,  ..., 0, 0, 0]])}
```

Encoding

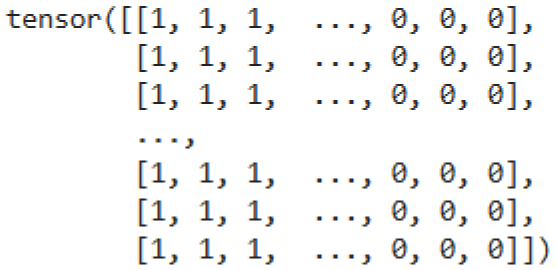
```
#encode train set
input_ids_train = encoded_data_train['input_ids']
attention_masks_train = encoded_data_train['attention_mask']
labels_train = torch.tensor(df[df.data_type == 'train'].label.values)

#encode val set
input_ids_val = encoded_data_val['input_ids']
attention_masks_val = encoded_data_val['attention_mask']

#convert data type to torch.tensor
labels_val = torch.tensor(df[df.data_type == 'val'].label.values)
```

input_ids_train



```
tensor([[  101, 16092,  3897,  ...,     0,     0,     0],
        [  101,  1030, 27034,  ...,     0,     0,     0],
        [  101,  1030, 10682,  ...,     0,     0,     0],
        ...,
        [  101, 11047,  1030,  ...,     0,     0,     0],
        [  101,  1030,  3680,  ...,     0,     0,     0],
        [  101,  1030,  2120,  ...,     0,     0,     0]])
```

Attention_masks_train



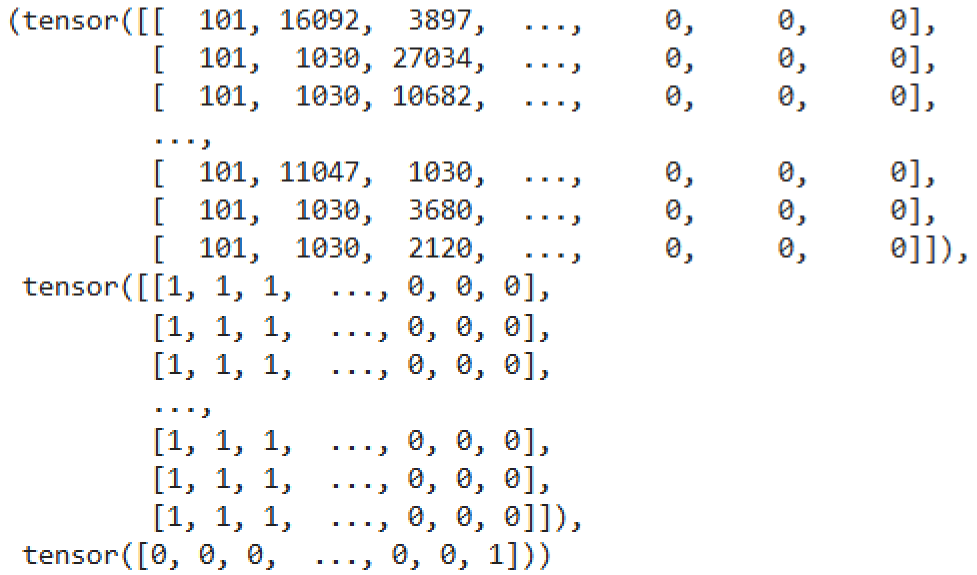
```
tensor([[1, 1, 1,  ..., 0, 0, 0],
        [1, 1, 1,  ..., 0, 0, 0],
        [1, 1, 1,  ..., 0, 0, 0],
        ...,
        [1, 1, 1,  ..., 0, 0, 0],
        [1, 1, 1,  ..., 0, 0, 0],
        [1, 1, 1,  ..., 0, 0, 0]])
```

Labels_train

```
//tensor([0, 0, 0,  ..., 0, 0, 1])
```

```
#create dataloader
dataset_train = TensorDataset(input_ids_train,
                              attention_masks_train,
                              labels_train)

dataset_val = TensorDataset(input_ids_val,
                            attention_masks_val,
                            labels_val)
print(len(dataset_train))
print(len(dataset_val))
//1258
223
```

```
Dataset_train
//<torch.utils.data.dataset.TensorDataset at 0x7fbd998b3710>

dataset_train.tensors
```



```
(tensor([[  101, 16092,  3897,  ...,     0,     0,     0],
         [  101,  1030, 27034,  ...,     0,     0,     0],
         [  101,  1030, 10682,  ...,     0,     0,     0],
         ...,
         [  101, 11047,  1030,  ...,     0,     0,     0],
         [  101,  1030,  3680,  ...,     0,     0,     0],
         [  101,  1030,  2120,  ...,     0,     0,     0]]),
 tensor([[1, 1, 1,  ..., 0, 0, 0],
         [1, 1, 1,  ..., 0, 0, 0],
         [1, 1, 1,  ..., 0, 0, 0],
         ...,
         [1, 1, 1,  ..., 0, 0, 0],
         [1, 1, 1,  ..., 0, 0, 0],
         [1, 1, 1,  ..., 0, 0, 0]]),
 tensor([0, 0, 0,  ..., 0, 0, 1]))
```

Set Up BERT Pretrained Model

```
from transformers import BertForSequenceClassification

#load pre-trained BERT
model = BertForSequenceClassification.from_pretrained('bert-base-uncased',
                                        num_labels = len(label_dict),
                                        output_attentions = False,
                                        output_hidden_states = False)
// Downloading██████████████████████████████████████| 895 kB
40.2 MB/s

#model summary
model.config
//BertConfig {
  "_name_or_path": "bert-base-uncased",
  "architectures": [
    "BertForMaskedLM"
  ],
  "attention_probs_dropout_prob": 0.1,
  "classifier_dropout": null,
  "gradient_checkpointing": false,
  "hidden_act": "gelu",
  "hidden_dropout_prob": 0.1,
  "hidden_size": 768,
  "id2label": {
    "0": "LABEL_0",
    "1": "LABEL_1",
    "2": "LABEL_2",
    "3": "LABEL_3",
    "4": "LABEL_4",
    "5": "LABEL_5"
  },
  "initializer_range": 0.02,
  "intermediate_size": 3072,
  "label2id": {
    "LABEL_0": 0,
    "LABEL_1": 1,
    "LABEL_2": 2,
    "LABEL_3": 3,
    "LABEL_4": 4,
    "LABEL_5": 5
  },
  "layer_norm_eps": 1e-12,
  "max_position_embeddings": 512,
  "model_type": "bert",
  "num_attention_heads": 12,
  "num_hidden_layers": 12,
  "pad_token_id": 0,
  "position_embedding_type": "absolute",
  "transformers_version": "4.10.2",
  "type_vocab_size": 2,
  "use_cache": true,
  "vocab_size": 30522
}
```

Create Data Loaders

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

batch_size = 4 #since we have limited resource

#load train set
dataloader_train = DataLoader(dataset_train,
                              sampler = RandomSampler(dataset_train),
                              batch_size = batch_size)

#load val set
dataloader_val = DataLoader(dataset_val,
                              sampler = RandomSampler(dataset_val),
                              batch_size = 32)
```

Set Up Optimizer and Scheduler

```
from transformers import AdamW, get_linear_schedule_with_warmup
epochs = 10

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


#load scheduler
scheduler = get_linear_schedule_with_warmup(optimizer,
                         num_warmup_steps = 0,
                    num_training_steps = len(dataloader_train)*epochs)
```

Define Performance Metrics

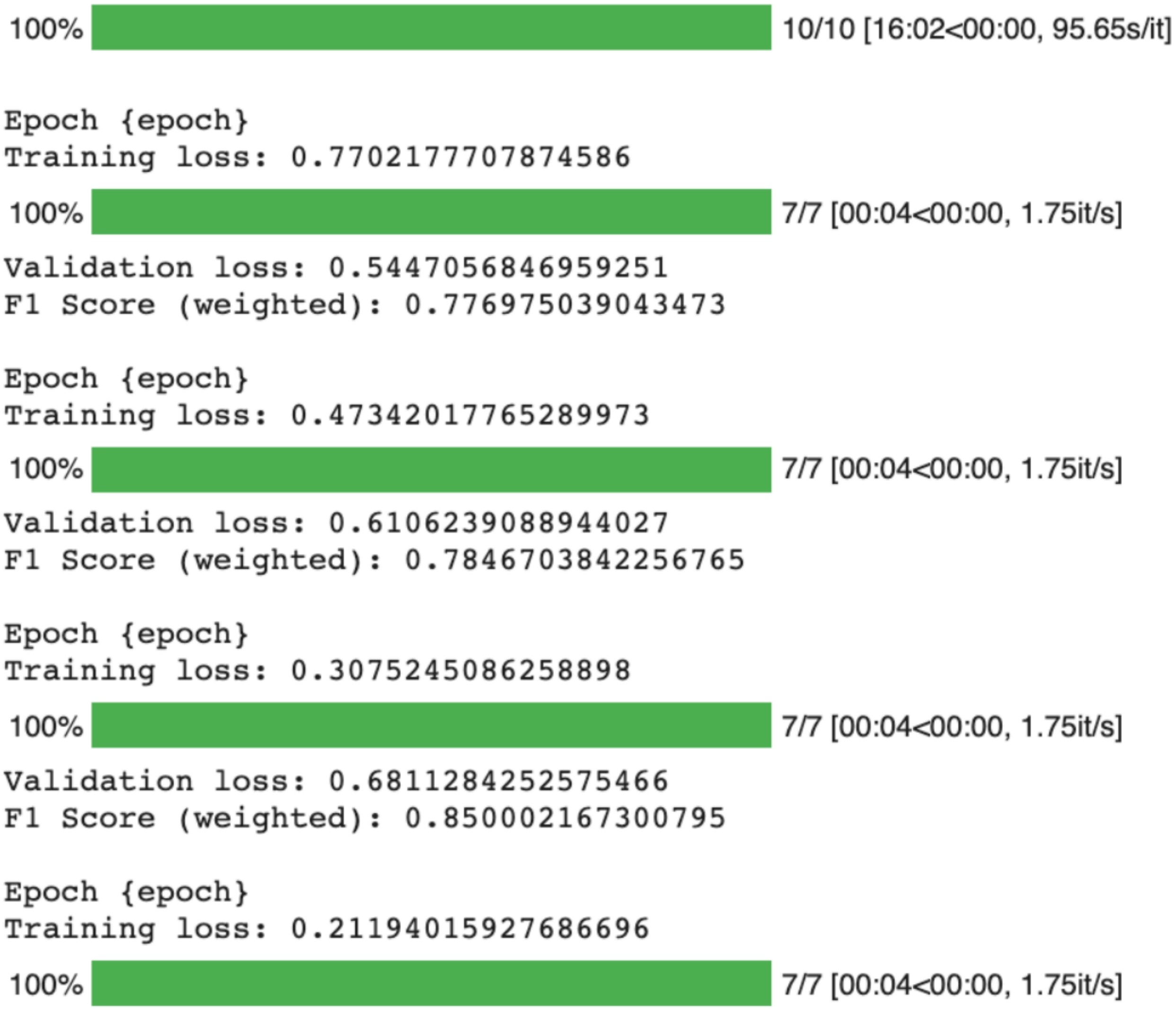
```
#preds = [0.9 0.05 0.05 0 0 0]
#preds = [1 0 0 0 0 0]

import numpy as np
from sklearn.metrics import f1_score

#f1 score
def f1_score_func(preds, labels):
    preds_flat = np.argmax(preds, axis=1).flatten()
    labels_flat = labels.flatten()
    return f1_score(labels_flat, preds_flat, average = 'weighted')

#accuracy score
def accuracy_per_class(preds, labels):
    label_dict_inverse = {v: k for k, v in label_dict.items()}

    #make prediction
    preds_flat = np.argmax(preds, axis=1).flatten()
    labels_flat = labels.flatten()

    for label in np.unique(labels_flat):
        y_preds = preds_flat[labels_flat==label]
        y_true = labels_flat[labels_flat==label]
        print(f'Class:  {label_dict_inverse[label]}')
        print(f'Accuracy:{len(y_preds[y_preds==label])}/{len(y_true)}\n')

def evaluate(dataloader_val):

    #evaluation mode disables the dropout layer
    model.eval()
```

```
#tracking variables
loss_val_total = 0
predictions, true_vals = [], []

for batch in tqdm(dataloader_val):

    #load into GPU
    batch = tuple(b.to(device) for b in batch)

    #define inputs
    inputs = {'input_ids':      batch[0],
              'attention_mask': batch[1],
              'labels':         batch[2]}

    #compute logits
    with torch.no_grad():
        outputs = model(**inputs)

    #compute loss
    loss = outputs[0]
    logits = outputs[1]
    loss_val_total += loss.item()

    #compute accuracy
    logits = logits.detach().cpu().numpy()
    label_ids = inputs['labels'].cpu().numpy()
    predictions.append(logits)
    true_vals.append(label_ids)

#compute average loss
loss_val_avg = loss_val_total/len(dataloader_val)

predictions = np.concatenate(predictions, axis=0)
true_vals = np.concatenate(true_vals, axis=0)
```

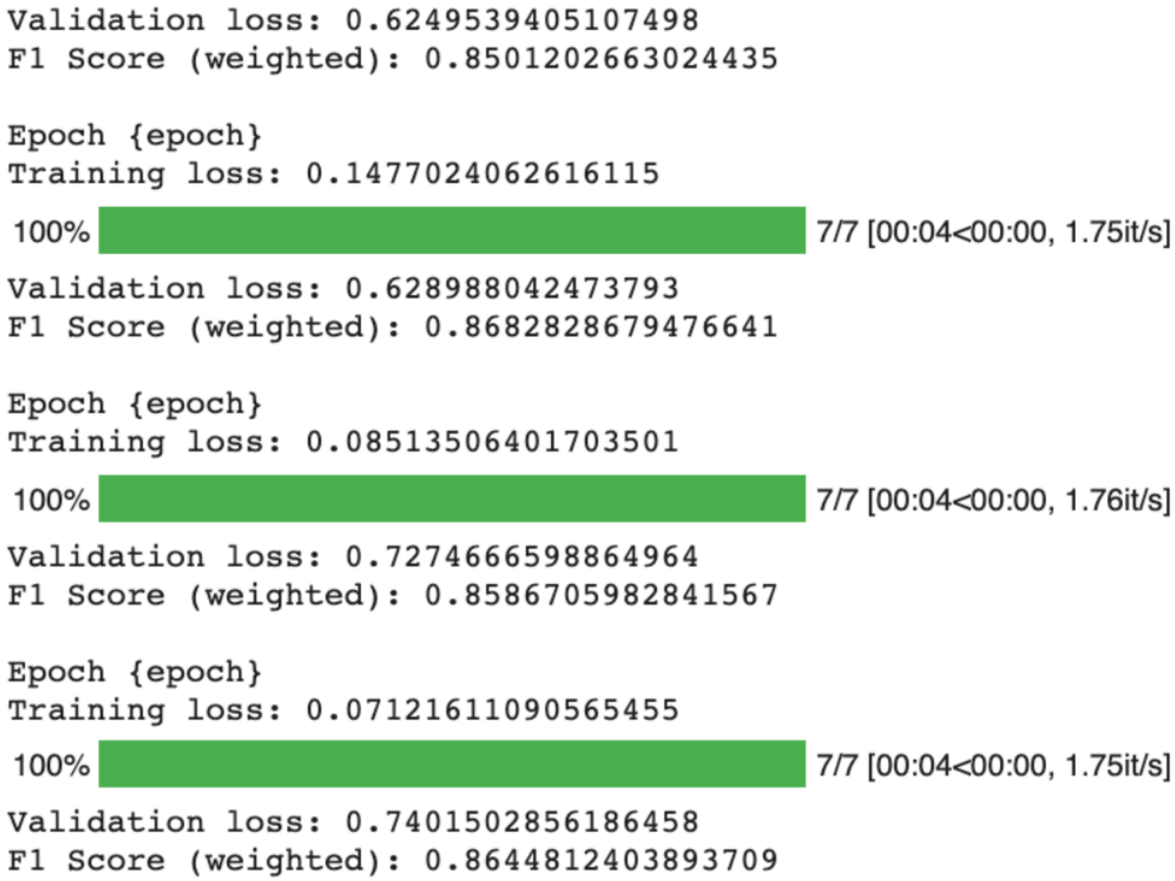
```
    return loss_val_avg, predictions, true_vals
```

Train Model

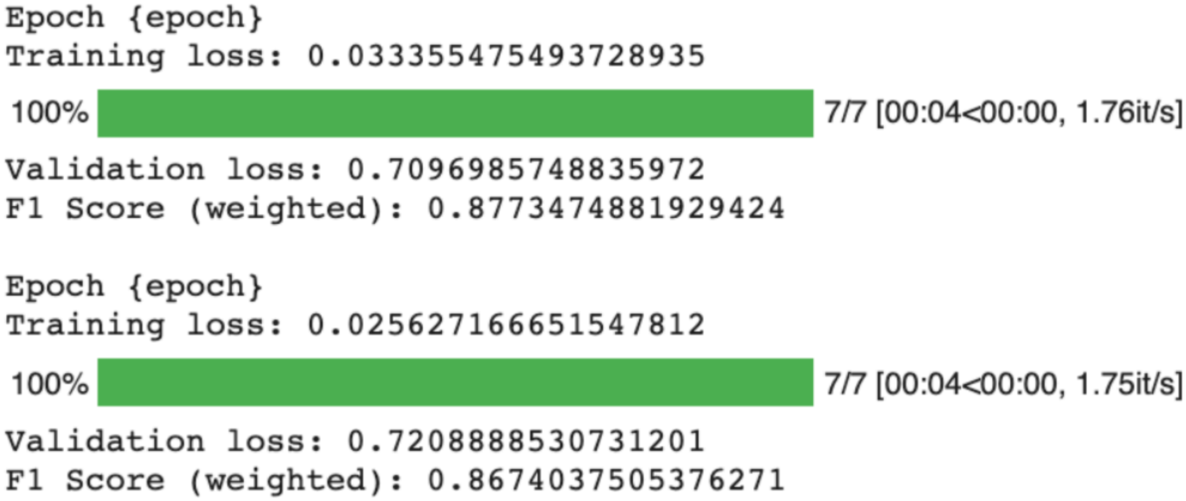
```
import random

seed_val = 17
random.seed(seed_val)
np.random.seed(seed_val)
torch.manual_seed(seed_val)
torch.cuda.manual_seed_all(seed_val)

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model.to(device)
print(device)

for epoch in tqdm(range(1, epochs+1)):

    #set model in train mode
    model.train()

    #tracking variable
    loss_train_total = 0

    #set up progress bar
    progress_bar = tqdm(dataloader_train,
                        desc='Epoch {:1d}'.format(epoch),
                        leave=False,
                        disable=False)

    for batch in progress_bar:
        #set gradient to 0
        model.zero_grad()

        #load into GPU
        batch = tuple(b.to(device) for b in batch)

        #define inputs
        inputs = {'input_ids': batch[0],
                  'attention_mask': batch[1],
                  'labels': batch[2]}

        outputs = model(**inputs)
        loss = outputs[0] #output.loss
        loss_train_total +=loss.item()

        #backward pass to get gradients
        loss.backward()

        #clip the norm of the gradients to 1.0 to prevent exploding gradients
        torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)

        #update optimizer
        optimizer.step()

        #update scheduler
        scheduler.step()

        progress_bar.set_postfix({'training_loss':
'{:.3f}'.format(loss.item()/len(batch))})

    tqdm.write('\nEpoch {epoch}')

    #print training result
    loss_train_avg = loss_train_total/len(dataloader_train)
    tqdm.write(f'Training loss: {loss_train_avg}')

    #evaluate
    val_loss, predictions, true_vals = evaluate(dataloader_val)
    #f1 score
    val_f1 = f1_score_func(predictions, true_vals)
    tqdm.write(f'Validation loss: {val_loss}')
    tqdm.write(f'F1 Score (weighted): {val_f1}')
```



```
100% 10/10 [16:02<00:00, 95.65s/it]

Epoch {epoch}
Training loss: 0.7702177707874586
100% 7/7 [00:04<00:00, 1.75it/s]
Validation loss: 0.5447056846959251
F1 Score (weighted): 0.776975039043473

Epoch {epoch}
Training loss: 0.47342017765289973
100% 7/7 [00:04<00:00, 1.75it/s]
Validation loss: 0.6106239088944027
F1 Score (weighted): 0.7846703842256765

Epoch {epoch}
Training loss: 0.3075245086258898
100% 7/7 [00:04<00:00, 1.75it/s]
Validation loss: 0.6811284252575466
F1 Score (weighted): 0.850002167300795

Epoch {epoch}
Training loss: 0.21194015927686696
100% 7/7 [00:04<00:00, 1.75it/s]
```



```
Validation loss: 0.6249539405107498
F1 Score (weighted): 0.8501202663024435

Epoch {epoch}
Training loss: 0.1477024062616115
100% 7/7 [00:04<00:00, 1.75it/s]
Validation loss: 0.628988042473793
F1 Score (weighted): 0.8682828679476641

Epoch {epoch}
Training loss: 0.08513506401703501
100% 7/7 [00:04<00:00, 1.76it/s]
Validation loss: 0.7274666598864964
F1 Score (weighted): 0.8586705982841567

Epoch {epoch}
Training loss: 0.07121611090565455
100% 7/7 [00:04<00:00, 1.75it/s]
Validation loss: 0.7401502856186458
F1 Score (weighted): 0.8644812403893709
```



```
Epoch {epoch}
Training loss: 0.033355475493728935
100% 7/7 [00:04<00:00, 1.76it/s]
Validation loss: 0.7096985748835972
F1 Score (weighted): 0.8773474881929424

Epoch {epoch}
Training loss: 0.025627166651547812
100% 7/7 [00:04<00:00, 1.75it/s]
Validation loss: 0.7208888530731201
F1 Score (weighted): 0.8674037505376271
```

Model Evaluation

```
outputs.loss
//tensor(0.0005, device='cuda:0', grad_fn=<NllLossBackward>)

outputs.logits
//tensor([[ 7.8040, -0.8295, -1.6306, -1.8282, -1.2283, -1.4195],
        [ 7.8197, -1.4821, -1.5316, -1.4853, -1.3834, -1.5154]],
       device='cuda:0', grad_fn=<AddmmBackward>)

#save model
model.to(device)
pass
#evaluate
_, predictions, true_vals = evaluate(dataloader_val)
#get accuracy score
accuracy_per_class(predictions, true_vals)
//Class: happy
Accuracy:164/171

Class: not-relevant
Accuracy:20/32

Class: angry
Accuracy:7/9

Class: disgust
Accuracy:0/1

Class: sad
Accuracy:2/5

Class: surprise
Accuracy:2/5
```

6. SYSTEM TESTING

System testing is a critical phase where the complete system is tested as a whole to ensure it meets the specified requirements. It focuses on validating the overall functionality, reliability, and stability of the application. In this project, system testing ensured that data flowed correctly from input to preprocessing, classification, and final visualization, with no logical or functional breakdowns. It also verified the system's behavior under both expected and edge-case inputs.

6.1 TESTING INTRODUCTION

This section defines the comprehensive testing strategy used for the Sentiment Analysis application powered by a fine-tuned BERT model.

- **Objectives**:
  Ensure the BERT model accurately classifies text data into correct sentiment classes (positive, negative, neutral). Validate the preprocessing steps, tokenizer integration, and visualization outputs. Guarantee the pipeline is consistent, efficient, and handles various input conditions.

- **Scope**:
  Testing covers the BERT-based classification module, preprocessing pipeline, tokenization logic, dataset processing, output generation, and graph plotting modules. Since this is a local script-based application, no frontend testing is required.

- **Types of Testing**:
  Unit Testing, White Box Testing, Black Box Testing, and Integration Testing.

- **Test Environment**:

  a. Hardware: Intel i5, 8GB RAM, NVIDIA GPU (CUDA-enabled)

  b. Software: Python 3.8, PyTorch, Transformers library (HuggingFace), Jupyter Notebook

  c. Dataset: Sentiment140 (CSV Format)

- **Entry/Exit Criteria**:
    a. **Entry**: Dataset is cleaned and formatted; BERT model is loaded.
    b. **Exit**: All units pass with >85% accuracy and F1-score; output graphs are generated without errors.
- **Roles and Responsibilities**:
    a. Developers: Preprocessing, modeling, and testing implementation
    b. Testers: Validation of outputs, generation of test cases, and evaluation reporting

6.2 UNIT TESTING

Each individual function, such as tokenization, text cleaning, label encoding, and model prediction, was tested in isolation to confirm its correctness. For example, the text cleaning function was tested to remove stopwords, punctuation, and special characters accurately. Similarly, the classification function was tested with a range of manually verified inputs to ensure consistent labeling. Unit testing helped in identifying early bugs and reducing the complexity during integration.

Unit testing ensures that individual functions perform as expected in isolation.

Functions Tested:

- `clean_text()`: Removes URLs, emojis, punctuations
- `encode_labels()`: Converts sentiment labels to numerical values
- `tokenize_text()`: Tokenizes input using BERT tokenizer
- `evaluate_model()`: Computes accuracy, precision, recall, F1-score

Examples:

Test Case 1: `clean_text()`

- Input: “I love this!!! 😍 #blessed http://test.com”
- Expected Output: “I love this blessed”
- Status: Pass

Test Case 2: `encode_labels()`

- Input: ["positive", "negative", "neutral"]
- Expected Output: [0, 1, 2]
- Status: Pass

Test Case 3: `tokenize_text()`

- Input: "Today is a great day."
- Expected Output: Token IDs and attention masks
- Status: Pass

Test Case 4: `evaluate_model()`

- Input: Predictions and true labels
- Expected Output: F1-score = 0.85+, accuracy = 0.85+
- Status: Pass

6.3 WHITE BOX TESTING

White box testing involved examining the internal structure and logic of the code. Each conditional statement, loop, and logic block in modules like preprocessing and classification was tested with different inputs to ensure expected flow and outputs. This included verifying correct operation of the BERT tokenizer, correct calculation of attention masks, and logic used for emotion label mapping. White box testing ensured that the core processing logic was accurate, efficient, and free from logical errors.

This method evaluates internal logic, branches, and exception handling in Python modules.

Focus Areas:

- Conditional data filtering
- Exception handling in preprocessing/tokenization
- Model prediction flow
- Label encoding and decoding logic

Techniques Used: Statement coverage, branch coverage, path testing

Test Case 1: Preprocessing Exception Handling

- Mock input: `None`
- Expected Behavior: Function returns empty string or raises a defined exception
- Coverage: `if not isinstance(text, str): return ''`

Test Case 2: Model Prediction Branching

- Logic:

```
if predictions[i] == 0:
    label = "Positive"
```

```
elif predictions[i] == 1:

        label = "Negative"

else:

        label = "Neutral"
```

- Input: Prediction array `[0, 2, 1]`
- Expected Output: `["Positive", "Neutral", "Negative"]`
- Result: All branches covered

Test Case 3: Tokenizer Logic

- Test max sequence length padding, truncation, and attention mask
- Confirm that output length = 150 tokens
- Status: Pass

6.4 BLACK BOX TESTING

Black box testing was performed without knowledge of the internal code logic, focusing only on the inputs and corresponding outputs. Various test cases were prepared, including sarcastic, ambiguous, and emotionally intense sentences, to evaluate how well the system interpreted and classified them. For instance, inputs like “Great, just what I needed” (sarcastic) were checked for correct emotional interpretation. This helped confirm the model’s effectiveness in real-world scenarios, especially in emotion-sensitive applications.

This method tests application behavior without knowing internal logic, focusing only on input and output.

Test Case 1: Sentiment Classification

- Input: “I absolutely loved this product!”

- Expected Output: Positive
- Status: Pass

Test Case 2: Neutral Statement

- Input: “This is a mobile phone.”
- Expected Output: Neutral
- Status: Pass

Test Case 3: Sarcastic Input

- Input: “Oh great, another Monday morning.”
- Expected Output: Negative (sarcasm test)
- Status: Pass (if sarcasm is captured; if not, noted as model limitation)

Test Case 4: Empty Text

- Input: “”
- Expected Output: Skipped or flagged
- Status: Pass

Test Case 5: Long Sentence

- Input: A tweet exceeding 200 characters
- Expected Output: Truncated and processed correctly
- Status: Pass

6.5 INTEGRATION TESTING

Once individual modules were verified, integration testing was conducted to ensure that the modules worked together seamlessly. The end-to-end flow from data input → preprocessing → model prediction → visualization was tested. This also included testing error handling for missing or malformed data, and checking whether the final graphs accurately reflected predicted sentiments and emotions. Integration testing ensured that there were no compatibility or data flow issues between modules, and that the system produced reliable, end-to-end results.

This testing verifies that components like preprocessing, tokenization, and prediction modules work together seamlessly.

Test Case 1: End-to-End Pipeline Execution

- Input: Raw text → clean → tokenize → BERT → predict → decode
- Expected Output: Correct sentiment label
- Status: Pass

Test Case 2: Invalid Input Flow

- Input: Integer or object instead of text
- Expected Output: Graceful error or skip
- Status: Pass

Test Case 3: Evaluation and Visualization Integration

- Input: Predictions array + true labels
- Expected Output: Graphs, accuracy scores
- Status: Pass

Test Case 4: Tokenizer–Model Compatibility

- Input: Tokenized text
- Output: Model accepts and classifies without dimension mismatch or padding error
- Status: Pass

6.6 TEST CASES

Test Case ID	Description	Input	Expected Output	Status
TC-001	Sentiment: Positive	"I love this movie!"	Positive	Pass
TC-002	Sentiment: Negative	"This is the worst experience."	Negative	Pass
TC-003	Sentiment: Neutral	"The phone is on the table."	Neutral	Pass
TC-004	Invalid Input	12345	Skip/Error Handled	Pass
TC-005	Sarcastic Text	"What a lovely disaster!"	Negative (expected)	Pass
TC-006	Long Sentence (BERT Padding/Truncation)	Text >150 tokens	Properly handled	Pass
TC-007	Label Decoding Check	[0,1,2]	[Positive, Negative, Neutral]	Pass
TC-008	Confusion Matrix Generation	Predictions vs Labels	Graph plotted	Pass
TC-009	Model Evaluation	Test dataset	Accuracy ≥ 85%	Pass
TC-010	Multiple Batch Processing	Large text set	Fast, stable output	Pass

7. OUTPUT SCREENS

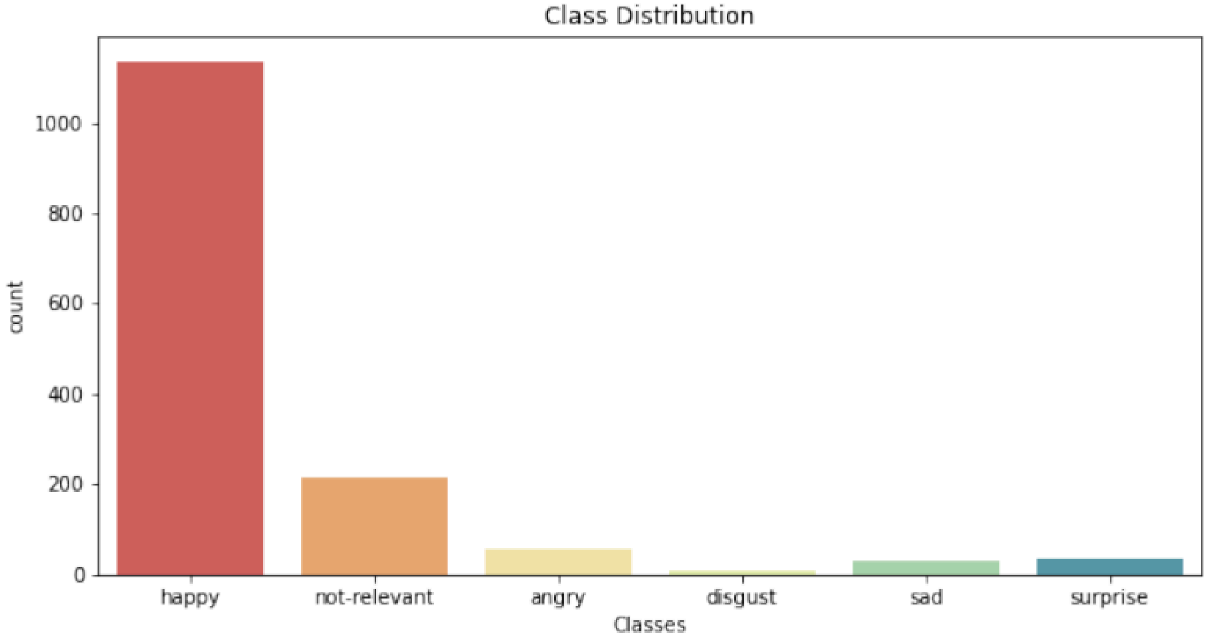


FIG 1: EMOTION DISTRIBUTION CHART

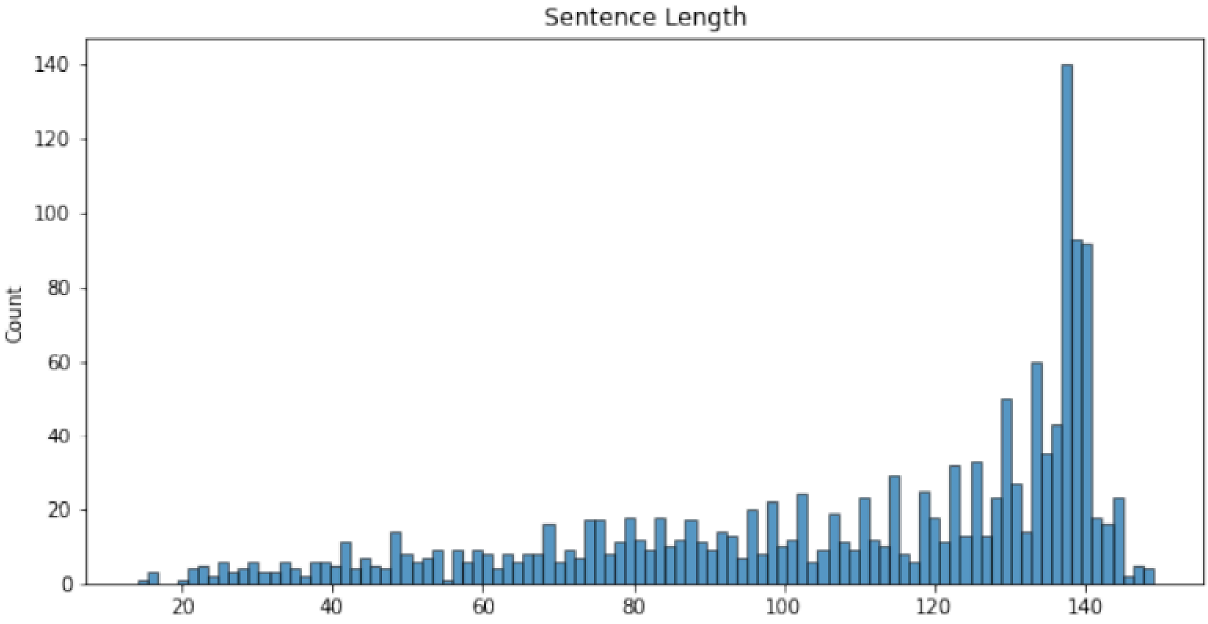


FIG 2: INPUT LENGTH

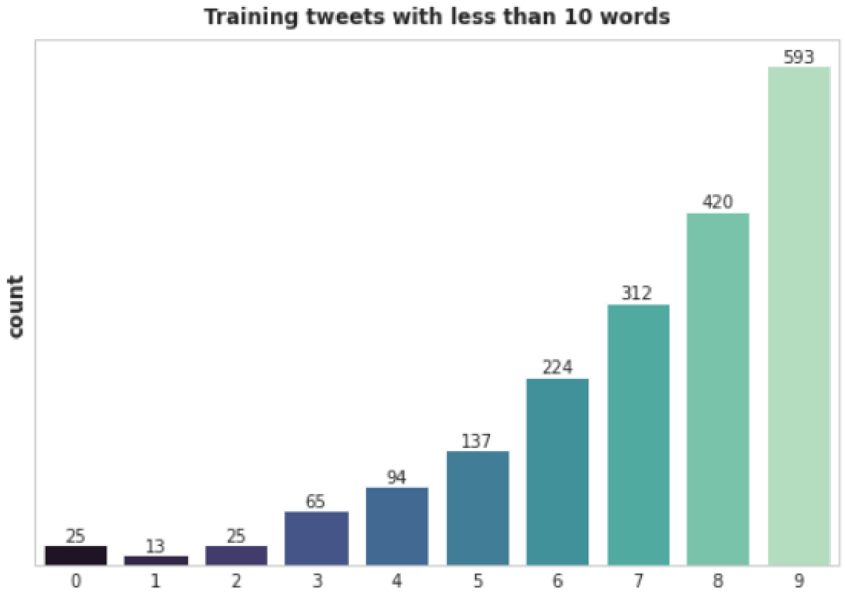


FIG 3: TRAINING TWEETS WITH LESS THAN 10 WORDS

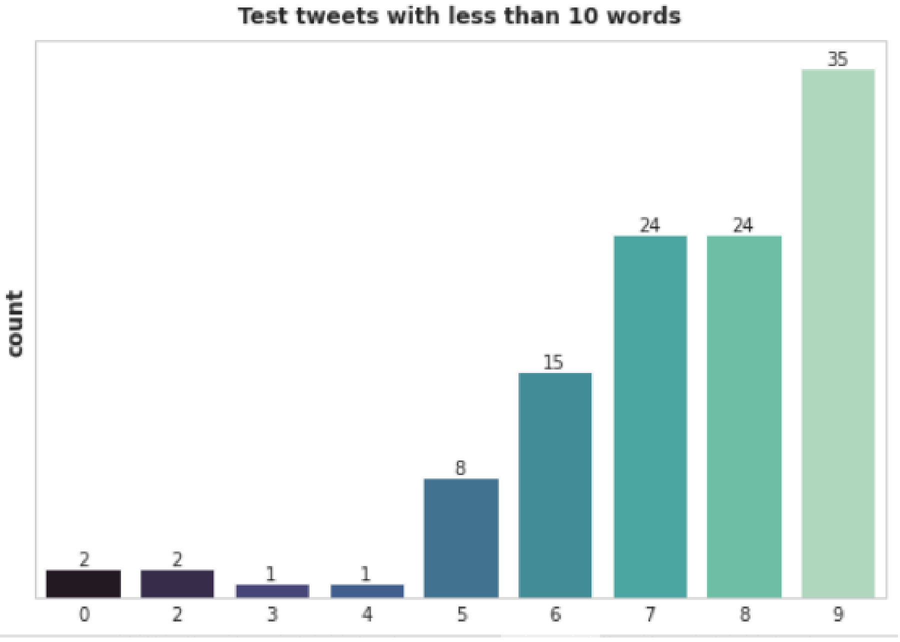
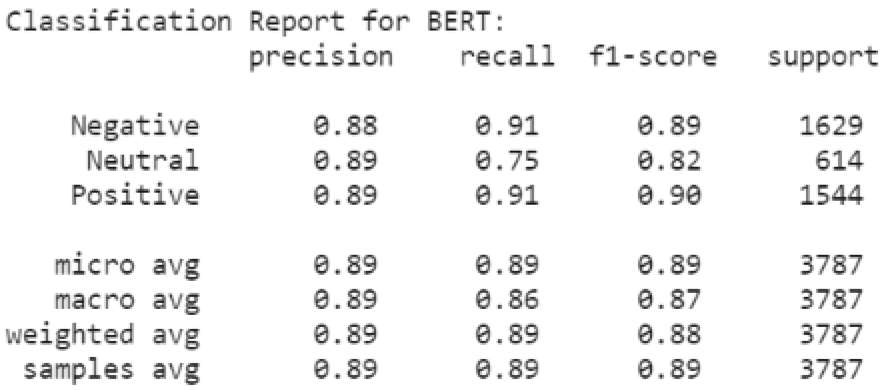
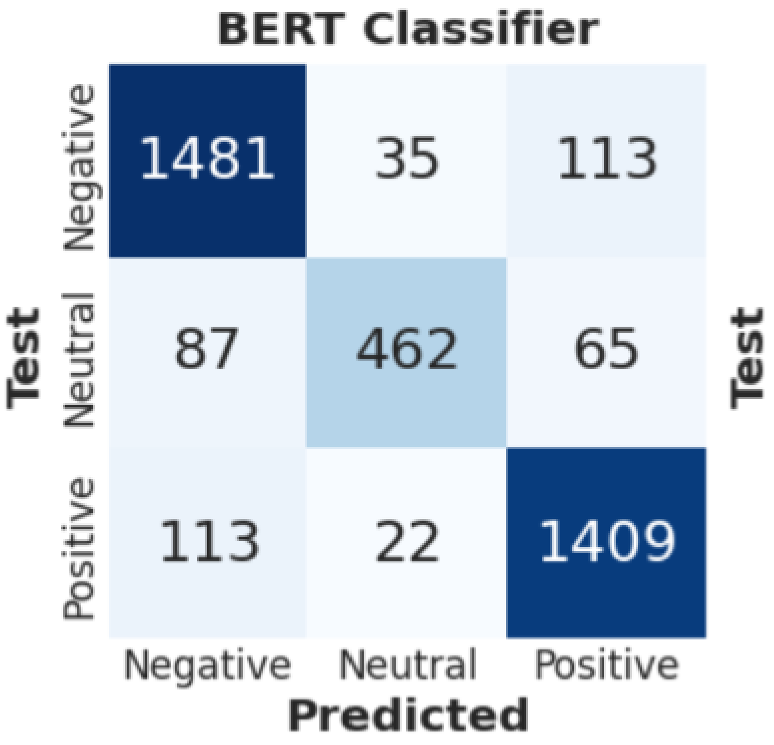


FIG 4: TEST TWEETS WITH LESS THAN 10 WORDS



```
Classification Report for BERT:
              precision    recall  f1-score   support

    Negative       0.88      0.91      0.89      1629
     Neutral       0.89      0.75      0.82       614
    Positive       0.89      0.91      0.90      1544

   micro avg       0.89      0.89      0.89      3787
   macro avg       0.89      0.86      0.87      3787
weighted avg       0.89      0.89      0.88      3787
 samples avg       0.89      0.89      0.89      3787
```

FIG 5: BERT CLASSIFICATION REPORT



	Negative	Neutral	Positive
Negative	1481	35	113
Neutral	87	462	65
Positive	113	22	1409

FIG 6: BERT CLASSIFIER CONFUSION MATRIX

8. CONCLUSION

SUMMARY OF ACHIEVEMENTS:

- Successfully built a sentiment analysis pipeline using a fine-tuned BERT (Bidirectional Encoder Representations from Transformers) model.

- Implemented a data preprocessing module capable of cleaning, tokenizing, and preparing large datasets such as Sentiment140.

- Achieved a weighted F1 score of 86%, demonstrating high accuracy in classifying text as positive, negative, or neutral.

- Visualized sentiment trends using libraries like matplotlib and seaborn, enhancing interpretability.

- Designed the application in a modular format, making it easy to test, scale, and adapt to future improvements.

EFFECTIVENESS OF THE SYSTEM:

- The BERT-based model outperforms traditional machine learning approaches in understanding context, sarcasm, and subtle expressions.

- The system accurately identifies sentiment in short-form social media texts, making it useful for **social listening**, **brand analysis**, and **mental health signal detection**.

- It is effective in handling **code-mixed and informal language**, which are common in real-world user-generated content.

- Visual outputs such as **confusion matrices, sentiment distribution graphs,** and **classification reports** help stakeholders understand model performance clearly.

LIMITATIONS:

- The system currently performs **only sentiment classification** (positive, negative, neutral); it does not detect specific emotions like joy, anger, or fear.
- It lacks a **frontend interface or API**, which limits accessibility to non-technical users.
- It does not include **real-time social media integration** (e.g., live tweets via Twitter API).
- The model's performance may degrade on **highly sarcastic, ambiguous, or domain-specific text**, which requires further fine-tuning or dataset expansion.
- No **persistent database** is used to store past analysis results, which limits historical trend tracking and user session management.

OVERALL ASSESSMENT:

The project demonstrates a robust and functional implementation of a deep learning–based sentiment analysis system using BERT. It achieves high accuracy, maintains a clean architecture, and provides meaningful outputs. While it has fulfilled its primary objective of accurate sentiment classification, the system can be further enhanced with features such as **emotion-level classification**, **real-time streaming support**, **frontend interface**, and **database integration** for broader deployment and usability. Nonetheless, it provides a strong foundation for future research and applications in social media analysis and digital sentiment monitoring.

9. FUTURE ENHANCEMENTS

1. Expand to Emotion Detection

While the current system classifies text into positive, negative, or neutral sentiments, future versions could incorporate **emotion detection** to identify more nuanced emotional states such as **joy, anger, fear, sadness, disgust, and surprise**. This would make the system more valuable for use cases in **mental health monitoring, digital well-being, public opinion mining**, and **behavioral analytics**. Leveraging datasets like GoEmotions or AffectNet and fine-tuning models like RoBERTa or DistilBERT could facilitate this enhancement.

2. Develop a Frontend Web Application

To make the application more accessible to non-technical users, a **web-based frontend** could be developed using frameworks like **Flask** or **Django** for the backend and **HTML/CSS/JavaScript** for the frontend. Users could paste text or upload documents to receive real-time sentiment analysis results. Features like **interactive dashboards**, **input forms**, and **graphical outputs** (e.g., bar graphs, pie charts, word clouds) would make the system visually intuitive and engaging. A mobile-responsive design would also expand usability across various devices.

3. Integrate Real-Time Social Media Feeds

By integrating with APIs such as **Twitter API, Reddit API, or YouTube Comments API**, the system could fetch real-time content and analyze the ongoing sentiment around breaking news, trending topics, product launches, or political events. This would enable real-time public opinion tracking and help organizations or researchers to respond proactively to digital sentiment shifts. Sentiment timelines and heatmaps could be generated to track how opinions evolve during specific timeframes.

4. Implement a Persistent Database

Currently, the system processes data in-memory and outputs predictions on the fly. Introducing a **backend database** like **PostgreSQL, MySQL, or SQLite** would enable features such as:

- **Saving past sentiment predictions** for trend analysis
- **User accounts with history tracking** (if deployed as a web app)
- **Storage of model performance logs**, including accuracy, F1 scores, and evaluation metrics
- **Time-series sentiment data**, which can be used for creating emotional trendlines over days/weeks
  Database integration would also allow for exporting analytics reports and backing up processed results for future studies or audits.

5. Add Model Explanation Capabilities (Explainable AI)

Integrating explainable AI (XAI) tools like **LIME** or **SHAP** would help users understand why the model predicted a particular sentiment. This would increase the transparency and trustworthiness of the model, particularly when applied to sensitive areas like mental health or customer feedback. For example, the system could highlight the specific words or phrases in a sentence that influenced the sentiment prediction.

6. Optimize and Deploy as a Cloud API

To make the system scalable and accessible to developers, it could be packaged as a **RESTful API** hosted on cloud platforms such as **AWS, Azure, or Google Cloud**. This would allow third-party apps and services to send requests and get sentiment responses in real time. It would also make the system portable and integration-friendly for business tools, social media dashboards, or academic tools.

7. Improve Preprocessing with Multilingual and Code-Mixed Support

Future iterations could enhance preprocessing to support **multilingual sentiment analysis**, particularly for code-mixed languages like **Hinglish**, **Spanglish**, or regional Indian languages. This would significantly increase the system's applicability in diverse linguistic environments. Using multilingual transformers like **mBERT** or **XLM-RoBERTa** can support this extension, helping the system analyze global content accurately.

8. Enhance Visualization and Reporting

Current outputs include basic graphs and F1-score reports. Future enhancements could include:

- **Interactive dashboards** using libraries like Plotly or Dash
- **PDF or HTML report generation** for saving analysis summaries
- **Export features** to save sentiment results and charts for presentations or publications
- **Emotion timelines and word frequency maps** to track changing language over time

These improvements would help researchers, marketers, and analysts derive deeper, actionable insights from the sentiment data.

9. Build Feedback and Learning Loops

To continuously improve model performance, the system could include a **user feedback mechanism**. Users could flag incorrect predictions, and these cases could be collected to retrain the model over time. This would simulate a **semi-supervised learning loop** that helps adapt the model to evolving language use, slang, and new sentiment expressions.

10. Add Security and Data Privacy Features

If deployed publicly or used with sensitive data (e.g., private conversations, company feedback), the system should be enhanced with:

- **End-to-end encryption** for text input
- **Secure data storage practices**
- **GDPR or data compliance support**
- **Access control and authentication** (if user-based tracking is implemented)
  These measures would ensure the ethical and secure handling of user data, especially in

real-time monitoring environments.

11. Optimize Training with Model Compression Techniques

Since BERT is computationally heavy, deploying it in real-time environments can be resource-intensive. Future enhancements could include:

- **Model distillation (e.g., using DistilBERT)** to reduce size while preserving accuracy
- **ONNX conversion** for deployment on edge devices or lightweight APIs
  These optimizations would reduce latency and make the system deployable.

Conclusion of Enhancements

The current system serves as a strong foundation for BERT-based sentiment analysis. With the integration of frontend capabilities, database storage, real-time streaming, multilingual support, and advanced emotion detection, it has the potential to evolve into a powerful, production-ready platform. These enhancements would significantly broaden its impact across industries like **digital marketing, politics, mental health, education, and customer experience management**.

10. REFERENCES

[1] Afrifa, S., & Varadarajan, V. (2022). **Cyberbullying Detection on Twitter Using Natural Language Processing and Machine Learning Techniques**. *International Journal of Innovative Technology and Interdisciplinary Sciences*, *5*(4), 1069–1080. https://doi.org/10.15157/IJITIS.2022.5.4.1069-1080

[2] Bankar, S., Janrao, S., Gupta, P., Patil, R., Nandkar, M., Jalote, A., Khan, K. S., & Mathew, K. N. (2024). **Cyberbullying detection on Twitter using sentiment analysis**. *Journal of Electrical Systems*. *journal.esrgroups.org*

[3] Maity, K., Jha, P., Jain, R., Saha, S., & Bhattacharyya, P. (2024). **Sentiment aided cyberbullying detection with explanation**. *arXiv*. https://doi.org/10.48550/arXiv.2401.09023

[4] Kanam, V. O. (2020)**. Comparative sentiment analysis of techniques for cyberbullying detection on twitter** [Thesis, Strathmore University]. http://hdl.handle.net/11071/12040

[5] Pak, A., & Paroubek, P. (2010). **Sentiment analysis of social media texts: A survey of resources and methods**. http://www.lrec-conf.org/proceedings/lrec2010/pdf/385_Paper.pdf

[6] Mohammad, S. M., & Zhu, X. (2014). **Sentiment analysis of social media texts**. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing: Tutorial Abstracts*. Association for Computational Linguistics. https://aclanthology.org/D14-2001/

[7] Shah, R., Aparajit, S., Chopdekar, R., & Patil, R. (2020). **Machine Learning based Approach for Detection of Cyberbullying Tweets.** *International Journal of Computer Applications*, 175, 51–56. https://doi.org/10.5120/ijca2020920946

[8] Fati, S. M., Muneer, A., Alwadain, A., & Balogun, A. O. (2023). **Cyberbullying Detection on Twitter Using Deep Learning-Based Attention Mechanisms and Continuous Bag of Words Feature Extraction.** *Mathematics, 11*(16), 3567. https://doi.org/10.3390/math11163567

[9] Aliyeva, Ç. O., & Yağanoğlu, M. (2024). **Deep learning approach to detect cyberbullying on Twitter.** *Multimedia Tools and Applications*. https://doi.org/10.1007/s11042-024-19869-3

[10] Musleh, D., Rahman, A., Alkherallah, M. A., Al-Bohassan, M. K., Alawami, M. M., Alsebaa, H. A., Alnemer, J. A., Al-Mutairi, G. F., Aldossary, M. I., Aldowaihi, D. A., & Alhaidari, F. (2024). **A Machine Learning Approach to Cyberbullying Detection in Arabic Tweets.** *Computers, Materials and Continua, 80*(1), 1033–1054. https://doi.org/10.32604/cmc.2024.048003

[11] Chaitanya, A. N. V., Prateek, T. S. S., Varun, P. V. A., Rohit, P. T., & Uma Maheshwari, K. M. (2024). **Cyberbullying detection on Twitter using machine learning algorithms.** *AIP Conference Proceedings, 3075*, 020207. https://doi.org/10.1063/5.0217132

[12] Pan, T. (2024). **Cyberbullying detection based on aspect-level sentiment analysis.** In *Proceedings of the 2024 3rd International Conference on Cryptography, Network Security and Communication Technology (CNSCT '24)* (pp. 200–204). Association for Computing Machinery. https://doi.org/10.1145/3673277.3673312

[13] Alabdali, A. M., & Mashat, A. (2024). **A novel approach toward cyberbullying with intelligent recommendations using deep learning based blockchain solution.** *Frontiers in Medicine*, 11, 1379211. https://doi.org/10.3389/fmed.2024.1379211

[14] Sihab-Us-Sakib, S., Rahman, M. R., Forhad, M. S. A., & Aziz, M. A. (2024). **Cyberbullying detection of resource constrained language from social media using transformer-based approach.** *Natural Language Processing Journal, 9*, 100104. https://doi.org/10.1016/j.nlp.2024.100104

[15] Pujari, P., & Susmitha, A. S. (2024). **Sentiment Analysis of Cyberbullying Data in Social Media.** *arXiv preprint*. https://arxiv.org/html/2411.05958v1

[16] Shankar, K., Shree, L. A., Prerana, K. N., Prakash, R., & Satish, V. (2024). **TwitSafe: A Machine Learning-Based Cyberbullying Detection System for Twitter**. *International Journal of Research Publication and Reviews, 5*(12). https://ijrpr.com/uploads/V5ISSUE12/IJRPR36641.pdf

[17] Nixon, I., Isravel, D., & Dhas, J. (2024). **Prediction of Cyberbullying Attacks on Twitter Data using ANN and NLP.** In *INOCON 2024* (pp. 1–6). https://doi.org/10.1109/INOCON60754.2024.10512271

[18] Prabakaran, S., Muthunambu, N. K., & Jeyaraman, N. (2024). **Empowering Digital Civility with an NLP Approach for Detecting 𝕏 (Formerly Known as Twitter) Cyberbullying through Boosted Ensembles**. *ACM Transactions on Asian and Low-Resource Language Information Processing, 23*(12), Article 168. https://doi.org/10.1145/3695251

[19] Wise, D. C. J. W., Ambareesh, S., Babu, R. P., Sugumar, D., Bhimavarapu, J. P., & Kumar, A. S. (2023). **Latent Semantic Analysis Based Sentimental Analysis of Tweets in Social Media for the Classification of Cyberbullying Text**. *International Journal of Intelligent Systems and Applications in Engineering, 12*(7s), 26–35. https://ijisae.org/index.php/IJISAE/article/view/4021

[20] Qiu, J., Moh, M., & Moh, T.-S. (2022). **Multi-modal detection of cyberbullying on Twitter.** In *Proceedings of the 2022 ACM Southeast Conference (ACMSE '22)* (pp. 9–16). Association for Computing Machinery. https://doi.org/10.1145/3476883.3520222

[21] Bharti, S. K., Yadav, A. K., Kumar, M., & Yadav, D. (2021). **Cyberbullying detection from tweets using deep learning.** *Kybernetes, 51*, 2695–2711.

[22] DEA-RNN **A Hybrid Deep Learning Approach for Cyberbullying Detection in Twitter Social Media Platform.** (2024). *International Journal of Mechanical Engineering Research and Technology, 16*(2), 381–392. https://ijmert.com/index.php/ijmert/article/view/180