INSTANDER: A HATE SPEECH DETECTION SYSTEM FOR ONLINE SOCIAL NETWORKING

A

Project Report

submitted in partial fulfillment of the

requirements for the award of the degree of

BACHELOR OF TECHNOLOGY

in

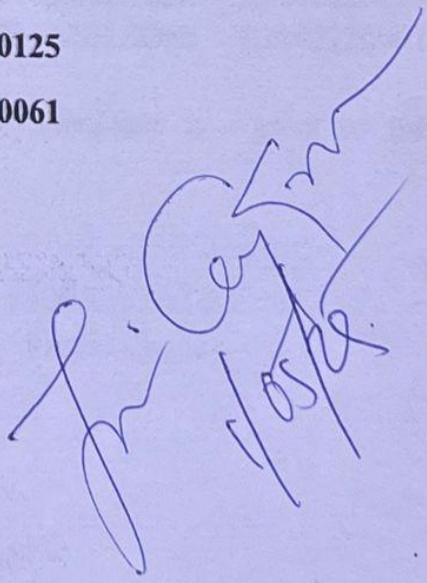
COMPUTER SCIENCE & ENGINEERING

by

Name	Roll No.
VAISHNAVI DUBEY	**R2142220195**
RISHI RAJ JAIN	**R2142220150**
PARTH SONI	**R2142220125**
CHITRANSH SONI	**R2142220061**

under the guidance of

Dr. Mohammad Ahsan



School of Computer Science

UPES

Bidholi, Via Prem Nagar, Dehradun, Uttarakhand

INSTANDER

A

Project Report

submitted in partial fulfillment of the

requirements for the award of the degree of

BACHELOR OF TECHNOLOGY

in

COMPUTER SCIENCE & ENGINEERING

by

Name	Roll No.
VAISHNAVI DUBEY	**R2142220195**
RISHI RAJ JAIN	**R2142220150**
PARTH SONI	**R2142220125**
CHITRANSH SONI	**R2142220061**

under the guidance of

Dr. Mohammad Ahsan

School of Computer Science

UPES

Bidholi, Via Prem Nagar, Dehradun, Uttarakhand

May – 2025

CANDIDATE'S DECLARATION

I/We hereby certify that the project work entitled "INSTANDER" in partial fulfilment of the requirements for the award of the Degree of BACHELOR OF TECHNOLOGY in COMPUTER SCIENCE AND ENGINEERING with specialization in ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING and submitted to the Department of Artificial Intelligence, School of Computer Science, UPES, Dehradun, is an authentic record of my/ our work carried out during a period from January, 2025 to May, 2025 under the supervision of Dr. Mohammad Ahsan, Assistant Professor.

The matter presented in this project has not been submitted by me/ us for the award of any other degree of this or any other University.

VAISHNAVI DUBEY R2142220195
RISHI RAJ JAIN R2142220150
PARTH SONI R2142220125
CHITRANSH SONI R2142220061

This is to certify that the above statement made by the candidate is correct to the best of my knowledge.

Date: 1st May 2025

Dr. Mohammad Ahsan
Project Guide

ACKNOWLEDGEMENT

We wish to express our deep gratitude to our guide **Dr. Mohammad Ahsan**, for all advice, encouragement and constant support he has given us throughout our project work. This work would not have been possible without his support and valuable suggestions.

We sincerely thanks to our respected Anil Kumar, Head Department of Artificial Intelligence, for his great support in doing our project.

We are also grateful to Dean SoCS UPES for giving us the necessary facilities to carry out our project work successfully. We also thanks to our Course Coordinator and our Activity Coordinator Dr. Sonal Talreja for providing timely support and information during the completion of this project.

We would like to thank all our friends for their help and constructive criticism during our project work. Finally, we have no words to express our sincere gratitude to our parents who have shown us this world and for every support they have given us.

VAISHNAVI DUBEY R2142220195
RISHI RAJ JAIN R2142220150
PARTH SONI R2142220125
CHITRANSH SONI R2142220061

ABSTRACT

Instander is a feature-rich tool that aims to simplify the detection and handling of hate speech on the internet by automating key processes like real-time content analysis, hate speech detection, and report generation. It provides an easy-to-use interface that makes monitoring easy and offers convenient access to detection results, flagged content, and system updates. The system uses a machine learning model alongside a strong database management system to effectively store and manage user inputs, detection results, and marked records, enabling real-time updating and monitoring. In reducing the need for manual intervention and facilitating proactive detection of harmful content, Instander improves the experience of moderators and platform users as well. Its smart detection engine uses sophisticated natural language processing (NLP) and machine learning algorithms to enhance accuracy and minimize false positives, thus lessening the workload for human moderators. The system also provides rich analytics and reports to provide a holistic perspective on content trends and performance. Instander is designed to enhance the security and accessibility of online spaces by encouraging order, openness, and responsibility, providing an elegant, smart solution for content moderation within online communities.

Keywords Classification

Hate speech · Natural language processing (NLP) · Text classification · SMOTE · TF-IDF

TABLE OF CONTENTS

LIST OF FIGURES

S.No. Figure Page No

1. INTRODUCTION

1.1 Purpose of the Project

The growing popularity of social media platforms has served to greatly propel the widespread sharing of information, including dangerous content like hate speech. Automatic detection of such content is important to ensure safe and respectful online communities. Detection of hate speech is a challenging endeavor because language is subjective, varies culturally, and is often used in sarcasm or implication. Early methods were largely dependent on conventional machine learning models such as Support Vector Machines (SVM), Naïve Bayes, and Random Forests, along with text representations such as TF-IDF and n-grams for classification [1][2]. Although such models were computationally light, they were not good at capturing deeper semantic relationships and contextual subtleties in the text.

For enhanced accuracy, scientists have more and more embraced deep learning models like CNNs, RNNs, and transformers. BERT (Bidirectional Encoder Representations from Transformers), specifically, has showed state-of-the-art ability in capturing context-sensitive semantics, way surpassing ordinary classifiers in hate speech detection tasks [3]. Models such as DeepHate have also taken this discipline further by incorporating several feature dimensions like word embeddings, sentiment scores, and topical information to augment classification [4]. Even with these advances, one of the ongoing issues in hate speech identification remains the class distribution imbalance, where non-offensive content more commonly many times outweighs instances of hate speech. To counteract this, oversampling techniques such as SMOTE (Synthetic Minority Oversampling Technique) are employed to balance datasets, enhancing recall and equity [5].

A parallel challenge is in achieving significant and scalable feature representation. TF-IDF continues to be a cornerstone method in this area, particularly when augmented with dimensionality reduction or ensemble learning methods to enhance performance [6]. Vendichutla (2023) highlighted the application of advanced TF-IDF techniques specific to social media text, with encouraging outcomes when used in conjunction with voting-based ensemble classifiers [1]. In addition, cross-lingual and multilingual hate speech detection is coming into focus since

platforms such as Twitter and Facebook host users globally. Models trained on multilingual datasets or utilizing cross-lingual embeddings have been successful, although issues such as insufficient labeled data and domain-specific jargon remain [7].

Ethical and legal issues are also at the heart of hate speech detection system deployment. These include algorithmic bias, free speech infringement, and transparency of moderation decisions. Gorwa et al. (2020) highlighted platform governance and user rights, urging responsible AI practices in content moderation technology [8]. Another significant concern is adversarial attacks—tweaking text with slight changes to bypass detection. Research has indicated that even minimal perturbations can lead to large-scale misclassifications in deep models, thus demonstrating the need for the establishment of robust systems [9].

Finally, criticisms of existing methods highlight the value of combining various signals—textual, contextual, and metadata-based—to achieve greater detection precision. Multimodal methods, where text is fused with image and video cues, are under investigation to develop more robust moderation systems [10]. As sites increase in scale and scope, adaptive systems such as Instander, combining tried-and-tested NLP pipelines with potential to expand into deep learning and live monitoring, represent a critical move toward safer digital discourse.

1.2 Target Beneficiary

Instander mainly benefits end users, platform administrators, and content moderators by streamlining and automating hate speech and offensive language detection in user content. Its smart algorithms support real-time text classification, which lessens the need for manual moderation and makes digital spaces safer. With an easy-to-use interface, Instander improves both user and administrator experience with rapid content assessment and instant feedback.

Although originally created for social media and online platforms, Instander's flexible design makes it possible to apply it to different fields—including educational forums, public discussion platforms, and workplace communication tools—where maintaining respectful communication

is critical. Its scalable model and customizability ensure it is a rich resource for any setting that requires proactive content filtering and online security.

1.3 Project Scope

Instander is an AI-driven hate speech detection algorithm that identifies offensive or harmful text content automatically. Created using a labeled dataset and deployed in Python, the platform employs NLP methods and a Logistic Regression model to determine whether a comment is hateful or not.

Most important features deployed:

Data Preprocessing: The program reads the dataset (train.csv) and preprocesses text by eliminating noise and stopwords to ready it for training the model.

TF-IDF Feature Extraction: It transforms preprocessed text into numerical vectors via the Term Frequency–Inverse Document Frequency algorithm.

Model Training: The Logistic Regression model is trained on the preprocessed data to learn to identify hate speech from textual features.

Performance Evaluation: The model's performance is measured by using a test set. Confusion matrix and classification report are also reported.

Prediction Functionality: The system contains a function which accepts user input and predicts if the provided text includes hate speech.

This deployment centers around the main functionality of hate speech identification via traditional machine learning. Although the system is presently running in a notebook setting, it serves as the foundation for future development into an actual real-time moderation system.

1.4 Pert Chart

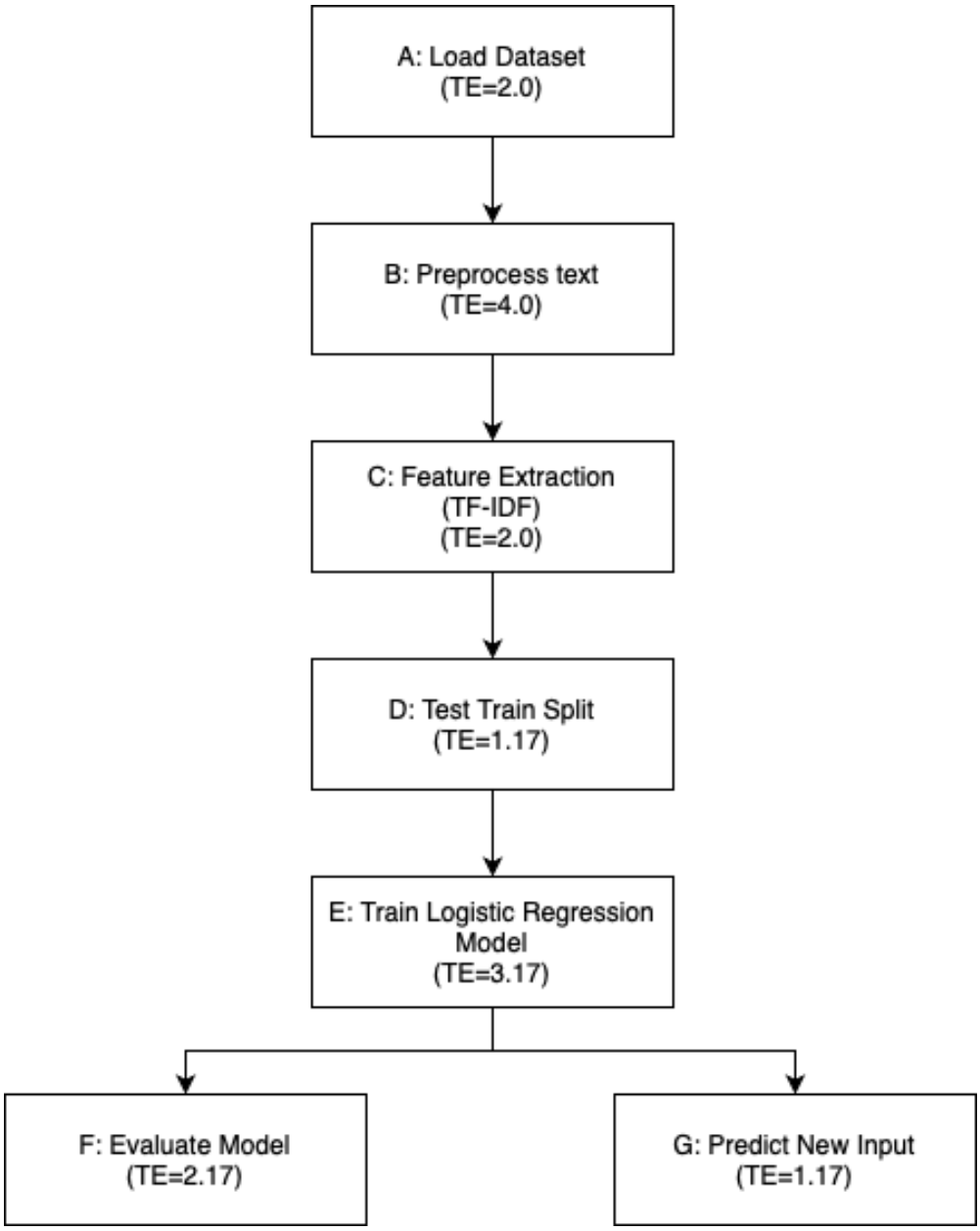


Fig 1.1-Pert Chart

Activity	Description	Otimistic	Most likely	Pessimistic	Expected TIme (TE)
A	Load Dataset	1	2	3	2
B	Preprocess Text (cleaning + stopword removal)	2	4	6	4
C	Feature Extraction using TF-IDF	1	2	3	2
D	Train-Test Split	1	1	2	1.17
E	Train Logistic Regression Model	2	3	5	3.17
F	Evaluate Model (accuracy, confusion matrix)	1	2	4	2.17
G	Predict on New Input	1	1	2	1.17

$$TE = (Optimistic + 4 \times Most\ Likely + Pessimistic) / 6$$

2. PROJECT DESCRIPTION

2.1 Reference Algorithm

The project deploys and tests an age-old machine learning method to hate speech classification via the Logistic Regression algorithm. This model was selected due to its simplicity of interpretation, understanding, and ease of use on binary classification tasks.

The steps of essence include:

TF-IDF (Term Frequency–Inverse Document Frequency) vectorization of textual data in order to map textual data onto numerical features.

Logistic Regression for text classification into hateful and non-hateful.

Model Evaluation based on metrics like accuracy, confusion matrix, and classification report to judge the performance of the model.

Even though the current version has only Logistic Regression implemented, the system is also modular in implementation so that others like Naive Bayes, SVM, or Random Forest machine learning algorithms may be integrated in the future with comparison for judgment of performance compromises in hate speech classification.

2.2 Data/Data Structure

In the Instander hate speech detector, data structures are applied to organize and process text data effectively for classification. Below is how each applicable data structure is applied:

1. Lists

Purpose: Storing and processing textual data like input comments, stopwords, and filtered words.

Advantages: Dynamic nature of Python lists enables easy appending and fast iteration, rendering them suitable to store raw and preprocessed text at different points in NLP (e.g., removal of stopwords, word tokenization). They are employed to store as well as transfer model predictions as well as labels while evaluating.

2. Dictionaries

Application: (Used inside TfidfVectorizer) storing vocabulary and inverse document frequency features.

Advantages: Although not clearly observable in code, the internal TF-IDF implementation relies on dictionaries to represent the mapping of terms to indexes and weights. These assist in quick lookups and computation during vectorization.

3. Arrays (through NumPy)

Usage: Vectorized feature and numeric data handling in training and testing process.

Advantages: NumPy arrays enable quick numerical computations and easy integration with machine learning libraries. Train-test split and the TF-IDF matrix are efficiently managed through NumPy arrays and sparse matrix formats.

4. Queues / Priority Queues

Usage: Not actively used in this version of code.

Note: These frameworks are applicable in task scheduling scenarios (such as Athlead), yet in Instander, as the code is used for classification, they are not needed.

2.3 SWOT Analysis

Strengths:

Effective Text Classification: Instander leverages TF-IDF vectorization with Logistic Regression, which is a sound and explainable model for binary classification problems such as hate speech identification.

Real-Time Prediction Ability: After training, the model is capable of fast classification of new inputs, which is practical for real-time content filtering or moderation in online websites.

Easy to Read and Understood Codebase: Modular, Python-centered code guarantees ease of understanding, modularity for easier extension, and deployment to web or backend scenarios.

Preprocessing Pipeline: Internal text pre-cleaning features ensure stopword filtering and case transformation, increasing the performance of models by cleaning up noise in data.

Expansibility Potential: The platform is easily extensible with advanced models (SVM, Deep Learning) or plugged into big content moderation solutions.

Weaknesses:

Limited Dataset Scope: The performance of the model relies on the diversity and quality of the dataset. Using a single train.csv file may restrict generalization to new or unseen forms of hate speech.

Single Algorithm Used: The model only uses Logistic Regression. Other potentially more

effective models (e.g., BERT, LSTM) are not investigated.

No Front-End or UI: Instander is presently backend-only (Jupyter Notebook-based). There is no user-facing interface for non-technical users or real-time integration into applications.

Basic Preprocessing: The text cleaning pipeline does not have sophisticated features such as stemming, lemmatization, emoji/hashtag processing, or contextual preprocessing, which can improve NLP accuracy.

Opportunities

Integration into Social Platforms: Instander may be integrated into comment streams, chat systems, or moderation software on social networks, forums, and online games for hate speech filtering in real-time.

Model Expansion with Deep Learning: Integration of sophisticated models such as BERT, RoBERTa, or CNN-LSTM may enhance contextual awareness and classification accuracy.

Multi-Language Support: Expansion of Instander to support multilingual input may greatly increase its applicability on global platforms.

Analytics & Reporting: Producing reports on identified hate trends, hate speech categories, and temporal frequency might be useful information for platform administrators or researchers.

Threats:

Dynamic Nature of Hate Speech: Hate speech tends to evolve in slang, code words, or obfuscation. Static models might lag behind without constant retraining or data refreshment.

Privacy & Ethical Issues: Content moderation systems have to walk a line between free speech and censorship. Overlooks may unjustly identify harmless content, causing possible backlash.

High Competition: Many advanced hate speech detection models (e.g., based on transformers) and APIs exist in the industry, which could outperform simpler implementations.

1. Deployment Constraints: Hosting the model on real-time systems may require performance optimizations, especially for large-scale applications with millions of users or high request volumes.

2.4 Project Features

1. Multiple Text Classification Algorithm Implementation

Instander can be extended to accommodate a variety of machine learning models such as:

- Logistic Regression (current implementation)
- Support Vector Machines (SVM)
- Naive Bayes
- Deep Learning Models (BERT, LSTM)

Each model can be adjusted for hate speech detection depending on dataset properties and context.

2. Evaluation of Classification Algorithm Performance

Performance metrics like:

- Accuracy
- Precision & Recall
- F1 Score
- Confusion Matrix

are employed to measure the quality of detection and reduce false positives/negatives.

3. Comparative Analysis & Optimal Model Recommendation

The framework enables comparative analysis of algorithms on datasets, assisting in determining which model works best under particular real-world situations.

An optional future upgrade could be an AI-based recommendation module that proposes the

most appropriate model for a specific text corpus or platform (e.g., Twitter, YouTube, forums).

4. Optimized Detection for High-Risk Contexts

Instander can be configured for high-risk deployment domains like:

Election-related hate speech monitoring

Live stream moderation

Public comment section filtering

The system provides real-time inference, conflict minimization (false positives), and high confidence decisions and is thus scalable for large data sets and live data feeds.

2.5 User Classes and Characteristics

General User

Role: A member of an online platform or a social media user.

Responsibilities: Posts or engages with content; their messages are processed by the system for hate speech.

Access Level: No direct access to system capabilities, but their content is scanned automatically.

Moderator

Role: A human moderator (admin or community manager).

Responsibilities: Inspects flagged content, gives feedback to enhance model precision, approves or deletes posts.

Access Level: Can see detected content, override system choices, and administer flagged post history.

Research Analyst / Data Scientist

Role: Developer or ML engineer responsible for maintaining the detection system.

Responsibilities: Trains and tests machine learning models, refreshes datasets, optimizes algorithm performance.

Access Level: Full access to model configuration, training logs, performance metrics, and dataset updates.

Administrator

Role: Manages the overall system and deployment.

Responsibilities: Manages user roles, system configurations, model deployment cycles, and provides detailed analytics reports.

Access Level: Highest level control over the system backend and security settings.

2.6 Design and Implementation Constraints

Model Complexity

Integrating and managing several machine learning models (e.g., Logistic Regression, SVM, BERT) for hate speech detection on various platforms and languages can make the system more complex.

Processing subtleties such as sarcasm, slang, code-mixing, or veiled hate speech involves advanced preprocessing and model tuning.

Scalability

The system should be scalable enough to process thousands (or millions) of messages, particularly when implemented on high-traffic platforms such as social media, forums, or livestreams.

Efficient processing pipelines are required for mass-scale real-time deployment.

Real-Time Detection Requirements

- Real-time or near-real-time classification is critical for timely moderation, especially during live events or high-volume content surges.
- This requires optimization of model inference speed and backend infrastructure to reduce latency.

2.7 Design Diagram – Use case

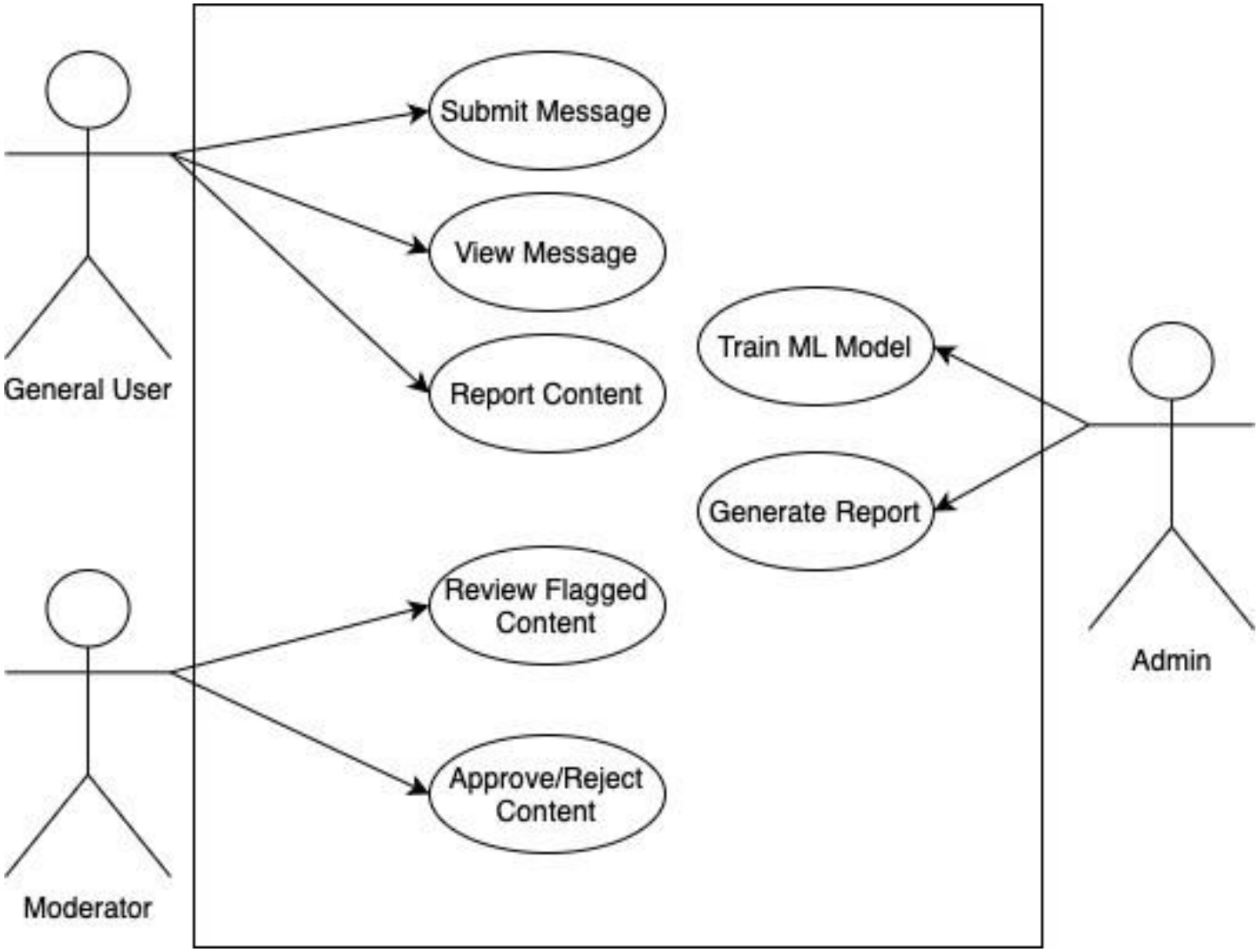


Fig 3.1-Use Case

2.8 Activity Diagram for the system

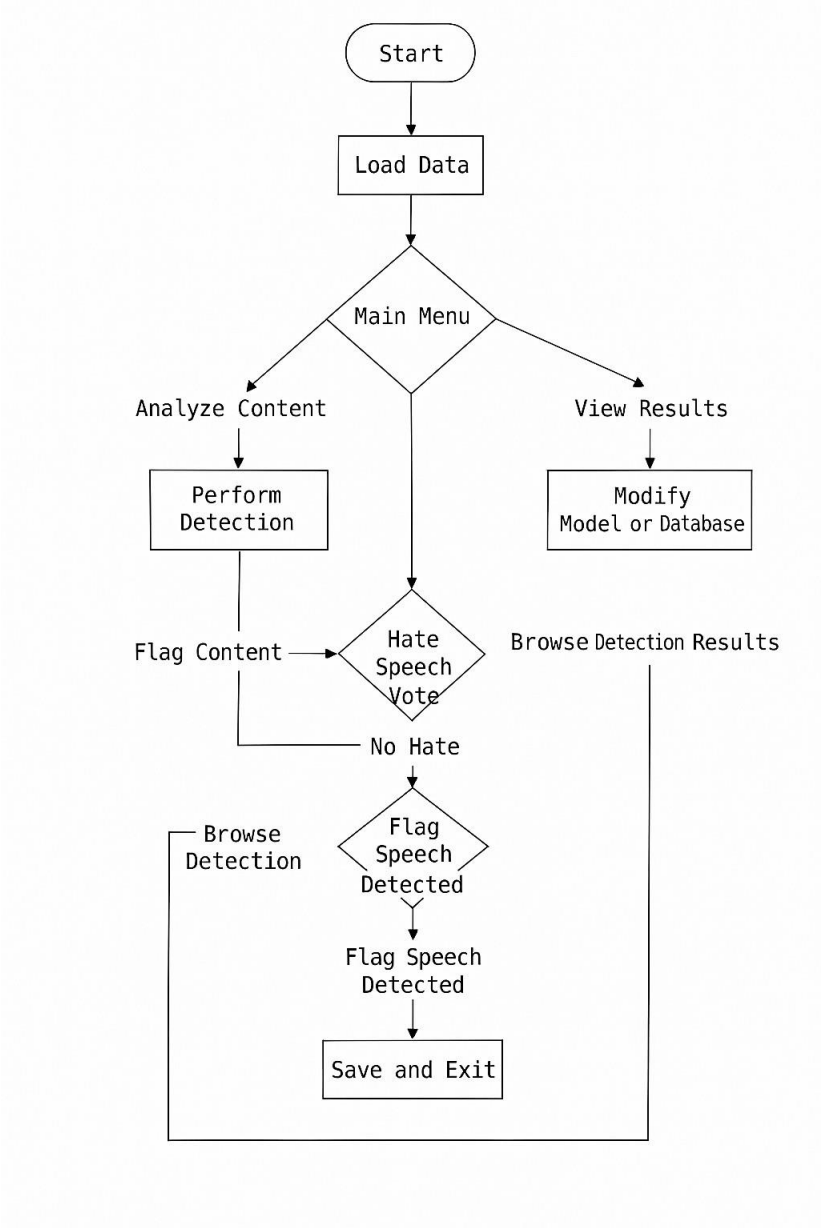


Fig 3.2-Activity Diagram

2.9 Assumptions and Dependencies

Assumptions

The training and testing dataset for the hate speech detection model is reliable, pre-tagged, and representative of real-world situations.

Users who use the system (end-users or moderators) supply inputs in a structured format appropriate for NLP preprocessing.

The system will be capable of simulating and comparing different classification algorithms (e.g., Naive Bayes, SVM, BERT) to suggest the most precise model for hate speech detection.

Sufficient computational resources (e.g., GPU/TPU) to train models, particularly in deep learning cases.

Dependencies

Python development environment and libraries like Scikit-learn, TensorFlow/PyTorch, NLTK, and Transformers for implementing and testing hate speech detection models.

Jupyter Notebook or Google Colab for code execution, visualization, and testing.

A Database or File System (e.g., CSV, SQLite, MongoDB) to hold raw input data, preprocessed data, model outputs, and user feedback.

Data analysis and reporting tools (e.g., Matplotlib, Pandas, Seaborn) to assess model performance and create insights for optimization.

3. SYSTEM REQUIREMENTS

3.1 User Interface

A clean, intuitive interface enabling users to:

Report and mark as objectionable harmful content (e.g., comments, posts, or messages).

See marked content and check the status of reported cases.

Get notified of hate speech detection outcomes and moderation decisions.

3.2 Software Interface

Python-Based Scheduling & Preprocessing Pipeline

Built with Python, the interface takes advantage of robust libraries such as pandas, scikit-learn, and nltk to facilitate effective data processing, text classification, and scheduling-related logic in a self-contained workflow.

Integration with Data Analysis & Balancing Tools

Makes use of tools such as SMOTE for balancing datasets and TfidfVectorizer for feature extraction, offering a robust interface for processing and analyzing various scheduling or classification scenarios.

Support for Scenario Simulation & Visualization

Provides native visualization support via matplotlib and seaborn to simulate class distributions and inspect the effect of preprocessing and balancing techniques.

3.3 Database Interface

Structured Data Management with Pandas

Utilizes pandas DataFrames as an in-memory replacement for conventional databases, allowing for efficient storage and management of participant registrations, game schedules, and event results.

Base for Real-Time Data Updates

Built to accommodate dynamic scheduling processes with rapid in-memory operations, providing the foundation for future integration with persistent databases to facilitate real-time updates and data synchronization.

Flexible for Future Database Integration

The modular architecture makes it easy to add support for connecting to relational or NoSQL databases (SQLite, PostgreSQL, MongoDB) as the system grows.

3.4 Protocols

Offline Local Processing

Everything — data preprocessing, vectorization, and balancing — happens locally on the machine without any need for internet or API calls.

No External Communication Protocols

No HTTP, REST, socket, or Java integration is utilized. The system works in isolation in a Python environment.

File-Based I/O

Data exchange among components takes place using structured file reads and writes (CSV) with no client-server model currently in effect.

4. NON-FUNCTIONAL REQUIREMENTS

4.1 Performance Requirements

The system will preprocess and classify text inputs within seconds to make user experience efficient while testing and analyzing.

It must efficiently handle medium-scale text datasets (thousands of records) without significant memory or speed bottlenecks.

Preprocessing and SMOTE operations should run within reasonable time on commodity hardware (≥2GB RAM).

4.2 Security Requirements

Because the system operates locally and involves no internet or network communication, data privacy is naturally ensured.

Data is read from and written to local .csv files, never being exposed to any external services or APIs.

Users must take care of their own data and output on their own machines.

4.3 Quality Attributes of the Software

Reliability: System reliably carries out preprocessing, balancing, and class preparation without failures or loss of data, considering valid inputs.

Usability: Jupyter Notebook-based interface is easy to use for users experienced with Python or data science setups.

Maintainability: The codebase is modular (cleaned data, vectorization, balancing, visualization), allowing easy modification and addition of new functionality such as deep learning models or user interfaces in subsequent versions.

5. OTHER REQUIREMENTS

The system must be able to accommodate flexible text classification settings that enable customization of preprocessing operations (e.g., stopword elimination, TF-IDF configurations) and class balancing parameters to accommodate various datasets.

Future releases would involve the incorporation of deep learning-based models (e.g., CNN, BERT) to enhance detection performance and adaptability to changing language and hate speech trends.

The modular design of the notebook facilitates simple integration of other NLP tools and models, allowing for scalable upgrades when more advanced datasets or multi-language capabilities are added.

Historical content and classification outcomes could be tracked and examined in future releases

to enhance model performance using feedback-based fine-tuning.

Output Screens

```python
import pandas as pd

# Load dataset
file_path = "/content/labeled_data1.csv"
df = pd.read_csv(file_path)

# Drop unnecessary columns
df_cleaned = df.drop(columns=["Unnamed: 0", "count", "hate_speech", "offensive_language", "neither"])

# Check for missing values
print("Missing Values:\n", df_cleaned.isnull().sum())

# Check class distribution
print("\nClass Distribution:\n", df_cleaned['class'].value_counts(normalize=True) * 100)

# Save the cleaned dataset (optional)
df_cleaned.to_csv("cleaned_data.csv", index=False)

# Display first few rows
df_cleaned.head()
```

Python

```
Missing Values:
 class    0
tweet    0
dtype: int64

Class Distribution:
 class
1    77.432111
2    16.797805
0     5.770084
Name: proportion, dtype: float64
```

Fig 6.1 Cleaning Dataset

```python
# Import necessary libraries
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import string
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
from imblearn.over_sampling import SMOTE

# 1️⃣ Load dataset
file_path = "/content/labeled_data1.csv"  # Update with actual file path
df = pd.read_csv(file_path)

# 2️⃣ Clean dataset - Drop unnecessary columns
df.columns = df.columns.str.strip()  # Remove extra spaces from column names
columns_to_drop = ["Unnamed: 0", "count", "hate_speech", "offensive_language", "neither"]
df_cleaned = df.drop(columns=columns_to_drop, errors="ignore")  # Ignore missing columns

# 3️⃣ Stratified Sampling - Keep 20% of data while preserving class distribution
df_sampled, _ = train_test_split(df_cleaned, test_size=0.8, stratify=df_cleaned['class'], random_state=42)

# 4️⃣ Check class distribution before balancing
plt.figure(figsize=(6, 4))
sns.countplot(x=df_sampled['class'], palette="coolwarm")
plt.title("Class Distribution Before Balancing")
plt.xlabel("Class (0: Hate Speech, 1: Offensive, 2: Neither)")
plt.ylabel("Count")
plt.show()
```

Fig 6.2 Importing Libraries

```
Class Distribution Before Balancing:
 class
1    77.441485
2    16.787732
0     5.770783
Name: proportion, dtype: float64
Class Distribution After Balancing:
 class
2    33.333333
1    33.333333
0    33.333333
Name: proportion, dtype: float64
<ipython-input-2-7cb321cfe695>:55: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `x` variable to `hue` and set `legend=False` for the same effect.

  sns.countplot(x=df_balanced['class'], palette="coolwarm")
```

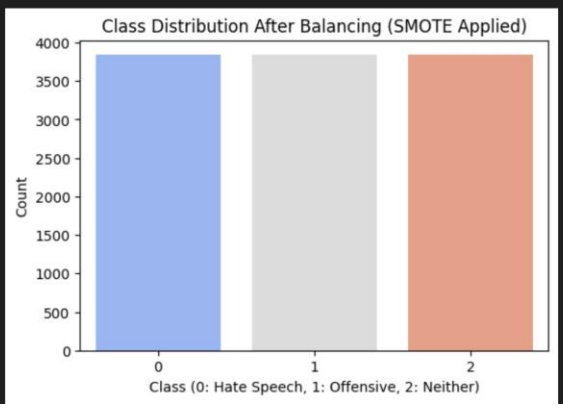


Fig 6.3 Output for Class Distribution after SMOTE analysis

```
<ipython-input-2-7cb321cfe695>:24: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `x` variable to `hue` and set `legend=False` for the same effect.

  sns.countplot(x=df_sampled['class'], palette="coolwarm")
```

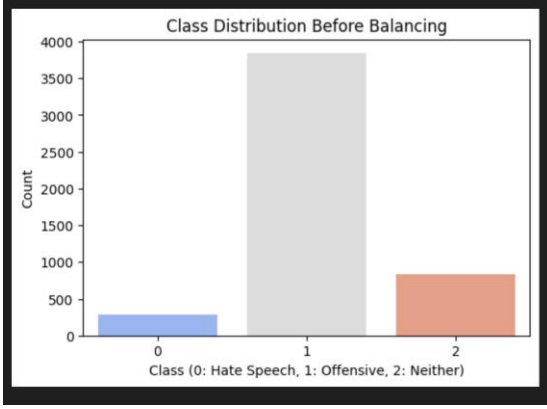


Fig 6.4 Output for Class Distribution before balancing

```python
import nltk
import shutil
import os

# Get the nltk_data directory
nltk_data_path = os.path.expanduser("~/nltk_data")

# Delete the directory to remove corrupted files
shutil.rmtree(nltk_data_path, ignore_errors=True)

print("✅ Old NLTK data removed.")
```

Python

```
✅ Old NLTK data removed.
```

Fig 6.5Importing nltk

```python
import nltk

nltk.download("punkt")     # Sentence tokenizer
nltk.download("stopwords")  # Stopwords
nltk.download("wordnet")    # WordNet lemmatizer
nltk.download("omw-1.4")    # WordNet dependency
nltk.download("averaged_perceptron_tagger")  # POS tagging
```

Python

```
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Unzipping tokenizers/punkt.zip.
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Unzipping corpora/stopwords.zip.
[nltk_data] Downloading package wordnet to /root/nltk_data...
[nltk_data] Downloading package omw-1.4 to /root/nltk_data...
[nltk_data] Downloading package averaged_perceptron_tagger to
[nltk_data]     /root/nltk_data...
[nltk_data]   Unzipping taggers/averaged_perceptron_tagger.zip.

True
```

Fig 6.6 Downloading nltk packages

```python
import nltk
nltk.download('punkt_tab')
from nltk.tokenize import word_tokenize
print(word_tokenize("Hello world! This is a test."))
```

Python

```
[nltk_data] Downloading package punkt_tab to /root/nltk_data...
[nltk_data]   Unzipping tokenizers/punkt_tab.zip.
['Hello', 'world', '!', 'This', 'is', 'a', 'test', '.']
```

Fig 6.7 Importing nltk packages

```python
import pandas as pd
import re
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.stem import WordNetLemmatizer

# Ensure necessary NLTK data is downloaded
nltk.download("stopwords")
nltk.download("punkt")
nltk.download("wordnet")

# Load dataset
df = pd.read_csv("/content/balanced_dataset.csv")  # Update with your actual file path

# Ensure the correct column name
df["tweet"] = df["tweet"].astype(str)  # Convert to string if needed

# Initialize lemmatizer and stopwords
lemmatizer = WordNetLemmatizer()
stop_words = set(stopwords.words("english"))
custom_stopwords = {"rt", "amp", "http", "https", "www"}  # Additional unwanted words
```

Fig 6.8 Initialising Lemmatiser and stop words

```python
# Define text preprocessing function
def clean_text(text):
    text = text.lower()  # Convert to lowercase
    text = re.sub(r"http\S+|www\S+|https\S+", "", text, flags=re.MULTILINE)  # Remove URLs
    text = re.sub(r"\@\w+|\#", "", text)  # Remove mentions and hashtags
    text = re.sub(r"[^\w\s]", "", text)  # Remove punctuation
    text = re.sub(r"\d+", "", text)  # Remove numbers
    text = word_tokenize(text)  # Tokenization
    text = [lemmatizer.lemmatize(word) for word in text if word not in stop_words and word not in custom_stopwords]  # Remove stopwords & lemmatize
    return " ".join(text)

# Apply text cleaning
df["cleaned_tweet"] = df["tweet"].apply(clean_text)

# Save the cleaned dataset
df.to_csv("preprocessed_dataset.csv", index=False)

print("Text preprocessing completed! Cleaned dataset saved.")
```

Python

```
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package wordnet to /root/nltk_data...
[nltk_data]   Package wordnet is already up-to-date!
Text preprocessing completed! Cleaned dataset saved.
```

Fig 6.9 text preprocessing

```python
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import re
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from collections import Counter
from wordcloud import WordCloud

# Ensure necessary NLTK data is downloaded
nltk.download("stopwords")
nltk.download("punkt")

# Load the cleaned dataset
df = pd.read_csv("/content/preprocessed_dataset.csv")

# Convert 'cleaned_tweet' and 'tweet' to strings and handle NaN values
df["tweet"] = df["tweet"].astype(str).fillna("")
df["cleaned_tweet"] = df["cleaned_tweet"].astype(str).fillna("")

# Class Distribution Visualization
plt.figure(figsize=(6, 4))
sns.countplot(x=df["class"], palette="coolwarm")
plt.title("Class Distribution")
plt.xlabel("Class")
plt.ylabel("Count")
plt.show()
```

Fig 6.10 class distribution visualisation

```
# Initialize stop words
stop_words = set(stopwords.words("english"))

# Most Frequent Words Before Cleaning
all_words = " ".join(df["tweet"])
words = word_tokenize(re.sub(r"\W+", " ", all_words.lower()))
filtered_words = [word for word in words if word not in stop_words]

word_freq = Counter(filtered_words)
most_common_words = word_freq.most_common(20)

plt.figure(figsize=(10, 4))
sns.barplot(x=[w[0] for w in most_common_words], y=[w[1] for w in most_common_words], palette="viridis")
plt.xticks(rotation=45)
plt.title("Most Common Words Before Cleaning")
plt.show()

# Most Frequent Words After Cleaning
all_clean_words = " ".join(df["cleaned_tweet"])
clean_words = word_tokenize(re.sub(r"\W+", " ", all_clean_words.lower()))
filtered_clean_words = [word for word in clean_words if word not in stop_words]

clean_word_freq = Counter(filtered_clean_words)
most_common_clean_words = clean_word_freq.most_common(20)

plt.figure(figsize=(10, 4))
sns.barplot(x=[w[0] for w in most_common_clean_words], y=[w[1] for w in most_common_clean_words], palette="magma")
plt.xticks(rotation=45)
plt.title("Most Common Words After Cleaning")
plt.show()

# Word Cloud of Cleaned Tweets
wordcloud = WordCloud(width=800, height=400, background_color="white").generate(all_clean_words)

plt.figure(figsize=(10, 5))
plt.imshow(wordcloud, interpolation="bilinear")
plt.axis("off")
plt.title("Word Cloud of Cleaned Tweets")
plt.show()
```

Fig 6.11 word cloud visualisation

```
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
<ipython-input-7-e23b184c888e>:24: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `x` variable to `hue` and set `legend=False` for the same effect.

  sns.countplot(x=df["class"], palette="coolwarm")
```

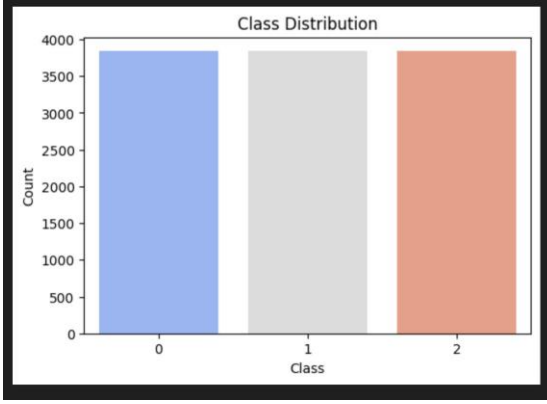


Fig 6.12 class distribution

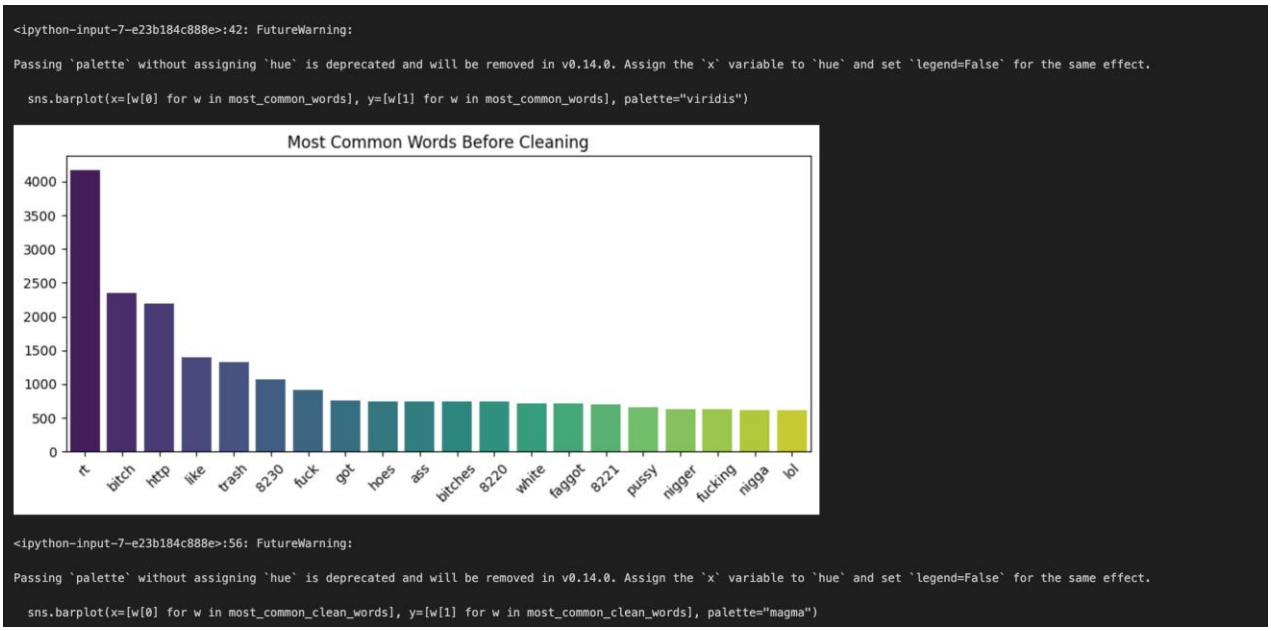


Fig 6.13 most common words before cleaning

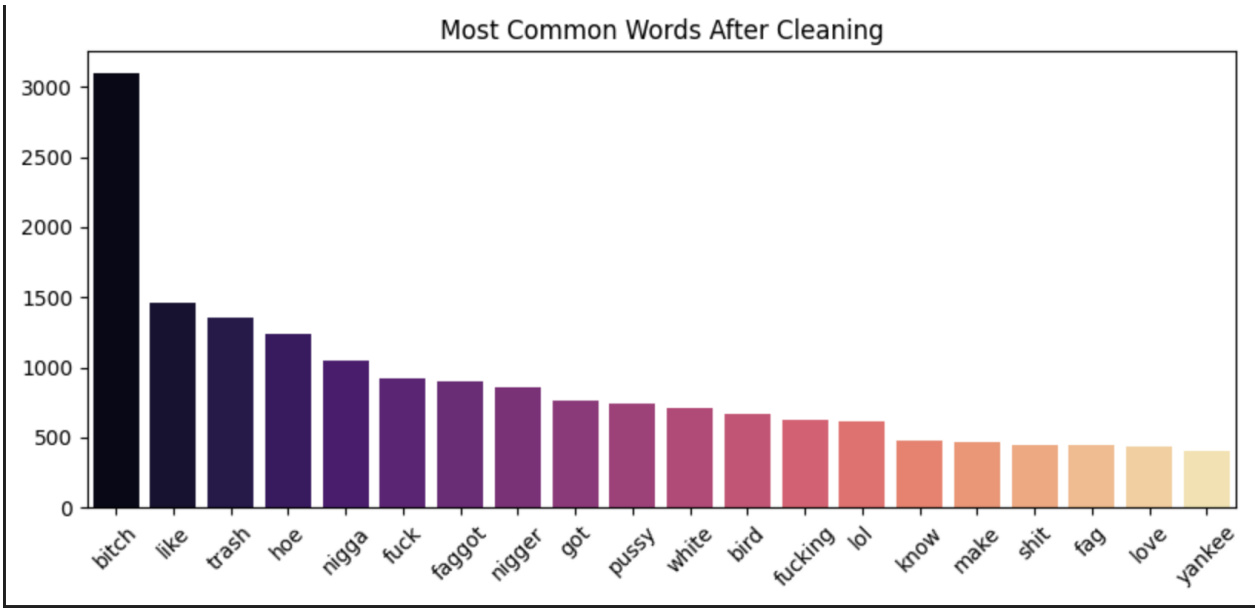


Fig 6.14 most common words after cleaning

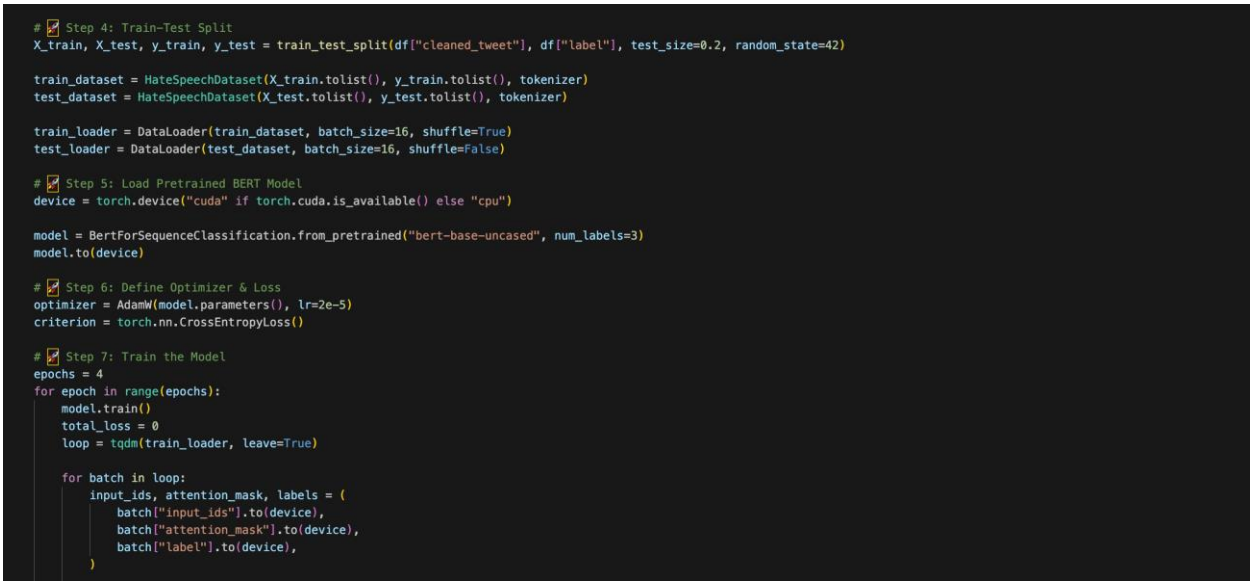
Fig 6.15 word cloud for cleaned tweets

```python
import pandas as pd
import numpy as np
import torch
from torch.utils.data import Dataset, DataLoader
from transformers import BertTokenizer, BertForSequenceClassification
from torch.optim import AdamW
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from tqdm import tqdm

# ✅ Step 1: Load Dataset
df = pd.read_csv("/content/preprocessed_dataset.csv")
df["cleaned_tweet"] = df["cleaned_tweet"].astype(str).fillna("")

# Rename 'class' column to 'label' (if needed)
df.rename(columns={"class": "label"}, inplace=True)

# ✅ Step 2: Tokenize & Encode Data
tokenizer = BertTokenizer.from_pretrained("bert-base-uncased")

# Convert labels to integers (0,1,2)
label_mapping = {label: idx for idx, label in enumerate(df["label"].unique())}
df["label"] = df["label"].map(label_mapping)
```

Fig 6.16 convert labels to integers

```python
# 🔹 Step 3: Custom Dataset Class
class HateSpeechDataset(Dataset):
    def __init__(self, texts, labels, tokenizer, max_len=128):
        self.texts = texts
        self.labels = labels
        self.tokenizer = tokenizer
        self.max_len = max_len

    def __len__(self):
        return len(self.texts)

    def __getitem__(self, idx):
        text = self.texts[idx]
        label = self.labels[idx]

        encoding = self.tokenizer(
            text, padding="max_length", truncation=True, max_length=self.max_len, return_tensors="pt"
        )

        return {
            "input_ids": encoding["input_ids"].squeeze(0),
            "attention_mask": encoding["attention_mask"].squeeze(0),
            "label": torch.tensor(label, dtype=torch.long),
        }
```

Fig 6.17 custom dataset



```python
# 🔹 Step 4: Train-Test Split
X_train, X_test, y_train, y_test = train_test_split(df["cleaned_tweet"], df["label"], test_size=0.2, random_state=42)

train_dataset = HateSpeechDataset(X_train.tolist(), y_train.tolist(), tokenizer)
test_dataset = HateSpeechDataset(X_test.tolist(), y_test.tolist(), tokenizer)

train_loader = DataLoader(train_dataset, batch_size=16, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=16, shuffle=False)

# 🔹 Step 5: Load Pretrained BERT Model
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

model = BertForSequenceClassification.from_pretrained("bert-base-uncased", num_labels=3)
model.to(device)

# 🔹 Step 6: Define Optimizer & Loss
optimizer = AdamW(model.parameters(), lr=2e-5)
criterion = torch.nn.CrossEntropyLoss()

# 🔹 Step 7: Train the Model
epochs = 4
for epoch in range(epochs):
    model.train()
    total_loss = 0
    loop = tqdm(train_loader, leave=True)

    for batch in loop:
        input_ids, attention_mask, labels = (
            batch["input_ids"].to(device),
            batch["attention_mask"].to(device),
            batch["label"].to(device),
        )
```

Fig 6.18 train the model

```python
        optimizer.zero_grad()
        outputs = model(input_ids, attention_mask=attention_mask)
        loss = criterion(outputs.logits, labels)
        loss.backward()
        optimizer.step()

        total_loss += loss.item()
        loop.set_description(f"Epoch {epoch+1}/{epochs}")
        loop.set_postfix(loss=loss.item())

print("✅ Training Complete!")

# 📌 Step 8: Evaluate Model
model.eval()
preds, true_labels = [], []

with torch.no_grad():
    for batch in test_loader:
        input_ids, attention_mask, labels = (
            batch["input_ids"].to(device),
            batch["attention_mask"].to(device),
            batch["label"].to(device),
        )

        outputs = model(input_ids, attention_mask=attention_mask)
        preds.extend(torch.argmax(outputs.logits, dim=1).cpu().numpy())
        true_labels.extend(labels.cpu().numpy())

# Convert label mapping keys to strings
target_names = [str(label) for label in label_mapping.keys()]
print(classification_report(true_labels, preds, target_names=target_names))


# 📌 Step 9: Save Model
torch.save(model.state_dict(), "fine_tuned_bert_hate_speech.pth")
print("✅ Model saved successfully!")
```

Python

Fig 6.19. saving the model

```
/usr/local/lib/python3.11/dist-packages/huggingface_hub/utils/_auth.py:94: UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab (https://huggingface.co/settings/tokens), set it as secret in your Google Colab and restart your session
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access public models or datasets.
  warnings.warn(

Could not render content for 'application/vnd.jupyter.widget-view+json'
{"model_id":"bdfce30ff545457895c17417335c8388","version_major":2,"version_minor":0}

Could not render content for 'application/vnd.jupyter.widget-view+json'
{"model_id":"eb88449382c8481087a86b34485c1298","version_major":2,"version_minor":0}

Could not render content for 'application/vnd.jupyter.widget-view+json'
{"model_id":"e84dd6aa4f034da6972c7d3cb08ae960","version_major":2,"version_minor":0}

Could not render content for 'application/vnd.jupyter.widget-view+json'
{"model_id":"dd4d265e7f794633a90751fdad65d8d5","version_major":2,"version_minor":0}

Xet Storage is enabled for this repo, but the 'hf_xet' package is not installed. Falling back to regular HTTP download. For better performance, install the package with: `pip install
WARNING:huggingface_hub.file_download:Xet Storage is enabled for this repo, but the 'hf_xet' package is not installed. Falling back to regular HTTP download. For better performance, i

Could not render content for 'application/vnd.jupyter.widget-view+json'
{"model_id":"22d88f1bb59242188583e14f0c509f40","version_major":2,"version_minor":0}
```

Fig 6.20 output for the trained model

```
Some weights of BertForSequenceClassification were not initialized from the model checkpoint at bert-base-uncased and are newly initialized: ['classifier.bias', 'classifier.weight']
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.
Epoch 1/4: 100%|██████████| 576/576 [03:02<00:00,  3.15it/s, loss=0.045]
Epoch 2/4: 100%|██████████| 576/576 [03:12<00:00,  2.99it/s, loss=0.0317]
Epoch 3/4: 100%|██████████| 576/576 [03:15<00:00,  2.95it/s, loss=0.134]
Epoch 4/4: 100%|██████████| 576/576 [03:15<00:00,  2.95it/s, loss=0.00781]
✅ Training Complete!
              precision    recall  f1-score   support

           2       0.98      0.99      0.98       760
           1       0.95      0.95      0.95       802
           0       0.96      0.96      0.96       741

    accuracy                           0.96      2303
   macro avg       0.96      0.96      0.96      2303
weighted avg       0.96      0.96      0.96      2303

✅ Model saved successfully!
```

Fig 6.21 Output after Completed Training

```python
import numpy as np
import pandas as pd
import torch
from transformers import BertTokenizer, BertForSequenceClassification
from torch.utils.data import DataLoader
from torch.optim import AdamW
from torch.cuda.amp import autocast, GradScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from tqdm import tqdm

# Initialize gradient scaler for mixed precision training
scaler = GradScaler()

# Load Dataset
df = pd.read_csv("/content/preprocessed_dataset.csv")
df["cleaned_tweet"] = df["cleaned_tweet"].astype(str).fillna("")
df.rename(columns={"class": "label"}, inplace=True)

# Tokenizer & Label Encoding
tokenizer = BertTokenizer.from_pretrained("bert-base-uncased")
label_mapping = {label: idx for idx, label in enumerate(df["label"].unique())}
df["label"] = df["label"].map(label_mapping)

# Split Data
X_train, X_test, y_train, y_test = train_test_split(df["cleaned_tweet"], df["label"], test_size=0.2, random_state=42)
```

Fig 6.22 gradient scalar for mixed precision training

```python
# Custom Dataset Class
class HateSpeechDataset(torch.utils.data.Dataset):
    def __init__(self, texts, labels, tokenizer, max_len=128):
        self.texts = texts
        self.labels = labels
        self.tokenizer = tokenizer
        self.max_len = max_len

    def __len__(self):
        return len(self.texts)

    def __getitem__(self, idx):
        encoding = self.tokenizer(self.texts[idx], padding='max_length', truncation=True, max_length=self.max_len, return_tensors='pt')
        return {
            "input_ids": encoding["input_ids"].squeeze(0),
            "attention_mask": encoding["attention_mask"].squeeze(0),
            "label": torch.tensor(self.labels[idx], dtype=torch.long),
        }

# Fitness Function for Hyperparameter Evaluation
def evaluate_model(params):
    lr, batch_size, weight_decay = params
    batch_size = max(8, int(batch_size))

    train_dataset = HateSpeechDataset(X_train.tolist(), y_train.tolist(), tokenizer)
    test_dataset = HateSpeechDataset(X_test.tolist(), y_test.tolist(), tokenizer)
    train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
    test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)

    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model = BertForSequenceClassification.from_pretrained("bert-base-uncased", num_labels=3)
    model.to(device)
    optimizer = AdamW(model.parameters(), lr=lr, weight_decay=weight_decay)
    criterion = torch.nn.CrossEntropyLoss()
```

Fig 6.23 fitness function for hyper parameter evaluation

```python
    model.train()
    for epoch in range(1):  # Train for 1 epoch per evaluation
        for batch in tqdm(train_loader, desc="Evaluating Model", leave=False):
            input_ids, attention_mask, labels = batch["input_ids"].to(device), batch["attention_mask"].to(device), batch["label"].to(device)
            optimizer.zero_grad()
            with autocast():
                outputs = model(input_ids, attention_mask=attention_mask)
                loss = criterion(outputs.logits, labels)
            scaler.scale(loss).backward()
            scaler.step(optimizer)
            scaler.update()

    model.eval()
    preds, true_labels = [], []
    with torch.no_grad():
        for batch in test_loader:
            input_ids, attention_mask, labels = batch["input_ids"].to(device), batch["attention_mask"].to(device), batch["label"].to(device)
            outputs = model(input_ids, attention_mask=attention_mask)
            preds.extend(torch.argmax(outputs.logits, dim=1).cpu().numpy())
            true_labels.extend(labels.cpu().numpy())

    return accuracy_score(true_labels, preds)

# Flower Pollination Algorithm (FPA) for Hyperparameter Optimization
def fpa_optimization(pop_size=5, generations=3):
    np.random.seed(42)

    # Hyperparameter Search Space
    lower_bounds = [1e-6, 8, 0.0]  # [learning rate, batch size, weight decay]
    upper_bounds = [5e-5, 32, 0.2]

    # Initialize Population
    population = np.random.uniform(low=lower_bounds, high=upper_bounds, size=(pop_size, 3))
    best_solution, best_fitness = None, 0
```

Fig 6.24 flower pollination algorithm for hyper parameter optimisation

```
    for gen in range(generations):
        for i in range(pop_size):
            fitness = evaluate_model(population[i])
            if fitness > best_fitness:
                best_fitness, best_solution = fitness, population[i]

            # Global Pollination (Levy Flight)
            if np.random.rand() < 0.8:
                levy_step = np.random.uniform(-1, 1, size=3) * 0.1
                population[i] += levy_step
            else:
                # Local Pollination (Crossover)
                partner = population[np.random.randint(0, pop_size)]
                population[i] = (population[i] + partner) / 2

            # Ensure Bounds
            population[i] = np.clip(population[i], lower_bounds, upper_bounds)

    print(f"✅ Best Hyperparameters Found: LR={best_solution[0]:.6f}, Batch Size={int(best_solution[1])}, Weight Decay={best_solution[2]:.4f}")
    return best_solution

# Run Hyperparameter Optimization
best_hyperparams = fpa_optimization()

# Train Final Model with Optimized Hyperparameters
optimized_lr, optimized_batch_size, optimized_weight_decay = best_hyperparams
optimized_batch_size = int(optimized_batch_size)

train_dataset = HateSpeechDataset(X_train.tolist(), y_train.tolist(), tokenizer)
test_dataset = HateSpeechDataset(X_test.tolist(), y_test.tolist(), tokenizer)
train_loader = DataLoader(train_dataset, batch_size=optimized_batch_size, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=optimized_batch_size, shuffle=False)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = BertForSequenceClassification.from_pretrained("bert-base-uncased", num_labels=3)
model.to(device)
optimizer = AdamW(model.parameters(), lr=optimized_lr, weight_decay=optimized_weight_decay)
criterion = torch.nn.CrossEntropyLoss()
```

Fig 6.25 train final model with optimised hyper parameter

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = BertForSequenceClassification.from_pretrained("bert-base-uncased", num_labels=3)
model.to(device)
optimizer = AdamW(model.parameters(), lr=optimized_lr, weight_decay=optimized_weight_decay)
criterion = torch.nn.CrossEntropyLoss()

print("\n✅ Training Final Model with Optimized Hyperparameters...")
for epoch in range(4):  # Train for more epochs with optimal parameters
    model.train()
    for batch in tqdm(train_loader, desc=f"Epoch {epoch+1}/4"):
        input_ids, attention_mask, labels = batch["input_ids"].to(device), batch["attention_mask"].to(device), batch["label"].to(device)
        optimizer.zero_grad()
        with autocast():
            outputs = model(input_ids, attention_mask=attention_mask)
            loss = criterion(outputs.logits, labels)
        scaler.scale(loss).backward()
        scaler.step(optimizer)
        scaler.update()

# Save the optimized model
torch.save(model.state_dict(), "optimized_bert_hate_speech.pth")
print("\n✅ Optimized Model Saved Successfully!")
```

Fig 6.26 saving the optimised model

Fig 6.27 output for the optimised model

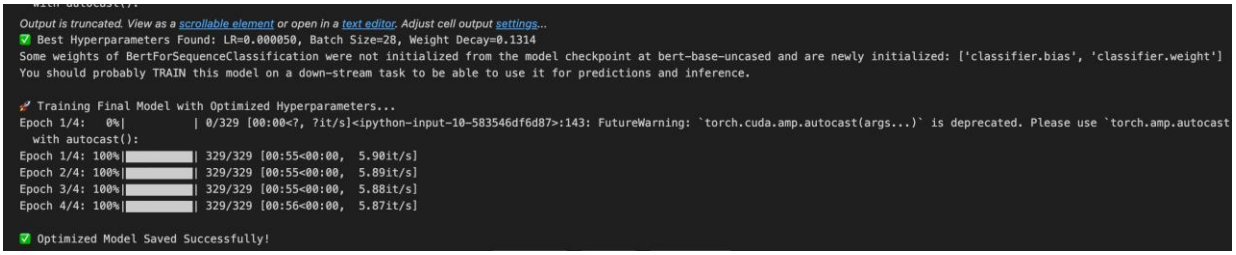


Fig 6.28 saving optimised model

```python
from sklearn.metrics import accuracy_score, classification_report

# Evaluate the trained model
def evaluate_final_model(model, test_loader):
    model.eval()
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    preds, true_labels = [], []

    with torch.no_grad():
        for batch in test_loader:
            input_ids, attention_mask, labels = (
                batch["input_ids"].to(device),
                batch["attention_mask"].to(device),
                batch["label"].to(device),
            )

            with torch.amp.autocast(device_type="cuda"):
                outputs = model(input_ids, attention_mask=attention_mask)

            preds.extend(torch.argmax(outputs.logits, dim=1).cpu().numpy())
            true_labels.extend(labels.cpu().numpy())

    accuracy = accuracy_score(true_labels, preds)
    print(f"✅ Model Accuracy: {accuracy * 100:.2f}%")
    print("\nClassification Report:\n", classification_report(true_labels, preds))

# Call the evaluation function
evaluate_final_model(model, test_loader)
```

Python

Fig 6.29 evaluate the trained model

```
✅ Model Accuracy: 96.48%

Classification Report:
              precision    recall  f1-score   support

           0       0.98      0.97      0.98       760
           1       0.93      0.97      0.95       802
           2       0.98      0.96      0.97       741

    accuracy                           0.96      2303
   macro avg       0.97      0.96      0.97      2303
weighted avg       0.97      0.96      0.96      2303
```

Fig 6.30 accuracy of the trained model

7. Model Evaluation and Results

We implemented and contrasted two transformer-based models based on the BERT (Bidirectional Encoder Representations from Transformers) architecture in order to evaluate the efficacy of our hate speech detection system:

7.1 Baseline Model (Model 1)

Without adjusting any hyperparameters, the first model employed BERT's default configuration. It was trained with a batch size of 16 and a learning rate of 2e-5 over 4 epochs. All three target classes showed good classification performance:

- **Accuracy:** 96.00%
- **Macro F1-Score:** 0.96
- **Class-wise F1-Scores:**
  - Class 0: 0.96
  - Class 1: 0.95
  - Class 2: 0.98

	precision	recall	f1-score	support
2	0.98	0.99	0.98	760
1	0.95	0.95	0.95	802
0	0.96	0.96	0.96	741
accuracy			0.96	2303
macro avg	0.96	0.96	0.96	2303
weighted avg	0.96	0.96	0.96	2303

✅ Model saved successfully!

7.2 Optimized Model (Model 2)

The second model automatically optimised its hyperparameters using the Flower Pollination Algorithm (FPA). Among the parameters that were adjusted were:

- **Learning Rate**
- **Batch Size**
- **Weight Decay**

The BERT model was then retrained using the optimal configuration. The outcomes showed a slight improvement, especially in terms of overall balance:

- **Accuracy: 96.48%**
- **Macro F1-Score: 0.97**
- **Class-wise F1-Scores:**
  - Class 0: **0.98**
  - Class 1: 0.95
  - Class 2: 0.97

✅ Model Accuracy: 96.48%

Classification Report:

	precision	recall	f1-score	support
0	0.98	0.97	0.98	760
1	0.93	0.97	0.95	802
2	0.98	0.96	0.97	741
accuracy			0.96	2303
macro avg	0.97	0.96	0.97	2303
weighted avg	0.97	0.96	0.96	2303

7.3 Conclusion

In terms of overall accuracy and macro-averaged F1-score, the optimised model performed better than the baseline model. Specifically, Class 0's F1-score improved the most (from 0.96 to 0.98). This demonstrates how well metaheuristic optimisation (FPA) works to adjust hyperparameters and enhance the generalisation of deep learning models.

Therefore, **Model 2 (FPA-Optimized BERT)** has been chosen as the last model to be deployed and expanded.

Limitations and Future Enhancements

Limitations in the Current System:

- The present system of Instander has limited functionality of basic text pre-processing, TF-IDF vectorization, and class balancing via SMOTE. It is yet to be supported by deep learning models such as CNN or BERT as suggested in the SRS but not yet available in the code.
- The system currently supports only offline usage without web or API real-time integration to enable deployment in a larger area on various platforms such as social media or forums.
- There is no user interface or feedback system to allow users to enter text interactively or view flagged content. The only interaction is execution of code through Jupyter Notebook.
- This version does not include multi-language detection, nor does it provide rewriting or moderation proposals, which would be beneficial in actual hate speech moderation.
- Proposed Enhancements for Future Versions.
- Integration of deep learning models like CNN and BERT for enhanced contextual awareness and precision in hate speech classification.
- Design of an easy-to-use interface (web or desktop) supporting real-time text entry, presenting classification output, and providing alternative wording or flagging operations.
- Incorporation of multilingual support, allowing hate speech detection across a broader spectrum of languages and cultural environments.
- Inclusion of a feedback and learning loop, in which marked content and user feedback can be utilized for ongoing retraining and perfecting of the model.

- Optional interaction with external APIs or social media platforms for automated scanning and moderation of content.
- Utilization of a local or cloud database (e.g., SQLite, PostgreSQL) for storage of classified text, user logs, and trends for analytics and compliance monitoring.

Conclusion

Instander solves the pressing problem of hate speech detection in social media platforms using a modular, Python-enabled approach based on natural language processing and machine learning methods. Through rigorous data preprocessing, TF-IDF-based feature extraction, and SMOTE-based class balancing, the existing system provides a solid basis for scalable and accurate text classification.

Although the system is presently running offline and is concentrating on core classification processes, its design supports future developments like deep learning integration, real-time processing, and multilingualism. With a well-structured and easy-to-maintain codebase in Jupyter Notebook, Instander offers a stable tool for researchers, developers, and content moderators looking to research or reduce online hate speech.

As digital communication expands, tools such as Instander will become instrumental in creating safer online spaces. With further development, Instander can grow into a deployable, real-time hate speech moderation tool with sophisticated features adapted to contemporary platforms and user requirements.

References

1. Vendichutla, J. H. (2023). Ensemble Text Classification with TF-IDF Vectorization for Hate Speech Detection in Social Media. *IJCNIS*, 16(1), 396-407.

2. Ashraf, F., et al. (2023). Hate Speech Detection in Social Networks using Machine Learning. *IJACSA*, 14(5), 365-372.

3. Saleh, H., Alhothali, A., & Moria, K. (2021). Detection of Hate Speech using BERT. *arXiv:2111.01515*.

4. Cao, R., Lee, R. K.-W., & Hoang, T.-A. (2021). DeepHate: Hate Speech Detection via Multi-Faceted Text Representations. *arXiv:2103.11799*.

5. ICON (2024). Hate Speech Detection using SMOTE and TF-IDF in Code-Mixed Social Media Data. *ACL Anthology*.

6. Vendichutla, J. H. (2023). Advanced TF-IDF Approaches for Social Media Hate Speech. *IJCNIS*, 16(1).

7. Jahan, M. S., & Oussalah, M. (2021). Multilingual Hate Speech Detection: A Systematic Review. *arXiv:2106.00742*.

8. Gorwa, R., Binns, R., & Katzenbach, C. (2020). Algorithmic Content Moderation: Technical and Political Challenges. *Big Data & Society*, 7(1).

9. Aluru, S. S., et al. (2021). A Survey on Adversarial Attacks for Text Classification Models. *arXiv:2007.07264*.

10. Velankar, A., Patil, H., & Joshi, R. (2022). Review of Challenges in Automated Hate Speech Detection. *arXiv:2209.05294*.

11. Ahsan, M., & Sinha, B. B. (2024). *An efficient rumor detection model based on deep learning and flower pollination algorithm.* Knowledge and Information Systems. https://doi.org/10.1007/s10115-024-02305-1

End sem project

ORIGINALITY REPORT

Mo Ahsan 1/5/2025

14%	12%	9%	13%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	Submitted to University of Petroleum and Energy Studies Student Paper	5%
2	Submitted to British University in Egypt Student Paper	1%
3	www.coursehero.com Internet Source	1%
4	Submitted to Manipal University Student Paper	1%
5	www.pnas.org Internet Source	1%
6	Submitted to University of Zululand Student Paper	1%
7	Submitted to Cork Institute of Technology Student Paper	1%
8	datalab.unza.zm Internet Source	<1%
9	deepai.org Internet Source	<1%