Text analysis using pretrained models for hate speech detection

A PROJECT REPORT

Submitted for the partial fulfillment

of

Project Based Learning (PBL) requirement of B. Tech CSE

Submitted by

Rushi Parhad, 23070521122
Sharvayu Zade, 23070521135
Mukund Kuthe, 23070521082

B. Tech Computer Science and Engineering

Under the Guidance of

Dr. Deepak Suresh Asudani

SYMBIOSIS
INSTITUTE OF TECHNOLOGY, NAGPUR
Wathoda, Nagpur

2025

CERTIFICATE

This is to certify that the Capstone Project work titled "**Text analysis using pretrained models for hate speech detection**" that is being submitted by **Rushi Parhad, 23070521122 Sharvayu Zade, 23070521135 Mukund Kuthe, 23070521082** is in partial fulfillment of the requirements for the Project Based Learning (PBL) is a record of bonafide work done under my guidance. The contents of this Project work, in full or in parts, have neither been taken from any other source nor have been submitted to any other Institute or University for award of any degree or diploma, and the same is certified.

Name of PBL Guide & Signature

Verified by:

Dr. Sudhanshu Maurya

PBL Coordinator

The thesis is satisfactory/unsatisfactory

Approved by

Prof. (Dr.) Nitin Rakesh
Director, SIT Nagpur

ABSTRACT

The pervasiveness of hate speech and objectionable content on the internet has grown as a result of the expanding use of social media platforms, posing severe issues to digital safety and moderation. This research aims to provide a solution to this issue by developing a machine learning system for automated hate speech identification in text data. Initially, utilized text vectorization software such as CountVectorizer and TF-IDF to convert unstructured text to a numeric representation that was utilizable in training the models. Then used and compared a variety of ML techniques, including XGBoost, Random forest, K-nearest neighbors (kNN), Decision Trees, Naive Bayes, and logistic Regression. Accuracy was employed to measure the performance of every model after training and testing on two publicly available datasets. Through comparison, accurate results were got to identify which algorithms perform better at hate speech detection, which provided crucial information for future automated filtering and content moderation apps.

Keywords: Hate Speech Detection, Text Classification, Machine learning, Deep learning, TF-IDF, CountVectorizer, Logistic Regression, XGBoost, Natural Language Processing (NLP), Feature Extraction, Dataset Evaluation, Model Accuracy.

TABLE OF CONTENTS

Figures

Tables

CHAPTER 1
INTRODUCTION

The use of hate speech, profanity, and abusive material has grown with the current digital world, in which millions of people interact using social media and online forums. Aside from offending individuals, these negative modes of communication have the potential to disrupt social cohesion and online communities. Automated systems that are capable of identifying and filtering out harmful content are increasingly needed in order to remedy this issue. The research, "Text Analysis Using Pretrained Models for Hate Speech Detection," is founded on this necessity.

To convert raw textual data into meaningful numerical forms, research started with examining feature extraction techniques such as CountVectorizer and TF-IDF (Term Frequency-Inverse Document Frequency). By reducing noise and preserving significant word-level structures, they help in determining the relative significance of words within a document compared to the entire dataset. Due to this transformation, the data was efficiently processed by deep learning and machine learning algorithms [1].

A variety of well-known machine learning algorithmsare used, such as Decision Trees, Naive Bayes, K-Nearest Neighbors (knn), logistic Regression, random forest, and xgBoost, once the text was vectorized. Providing the unique strengths and weaknesses of each model in detecting hate speech, and each gave a unique perspective on text classification. Experimented with deep learning architectures as well as traditional machine learning models to compare how they performed on the same task. Proper understanding of how machine learning (ML) and methods can aid in automated text analysis has been enhanced due to these models, which were specifically created and trained for hate speech detection in the framework of information retrieval (IR) [2].

Employed two different publically available datasets which are Hate speech data and **ConvAbuseEMNLPfull** to evaluate each of our models. Conduction of a comparative performance study by comparing the accuracy scores between models and datasets to observe how each algorithm performed on different data distributions and context shifts. Developed an end-to-end text classification pipeline for hate speech identification using this project. The experience provided valuable insights into how pretrained and self-trained models could be utilized in actual natural language processing applications, especially in intelligent automation to combat online toxicity.

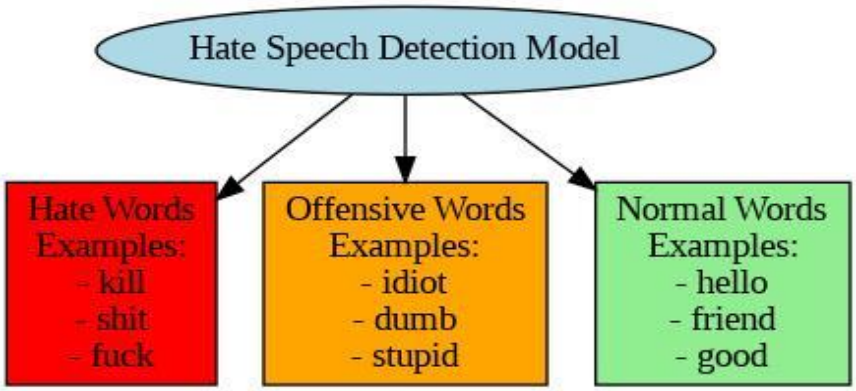


Fig 1 Hate Speech Detection and Word Classification

The figure 1 visually represents a Hate Speech Detection Model at the center, which classifies input text into three categories which are Hate words where examples are shit ,kill and fuck these words are considered as extreme harmful words ,Offensive words are those which may not qualify for hate speech but are still inappropriate such as idiot , dumb, stupid and the third classification is for normal words which are positive words with no offensive or hateful content such as hello , good and great.

1.1 Aim

The overall objective of this project is to create a system based on ml that incorporates a range of feature extraction techniques and classification models in order to automatically detect and classify hate speech and objectionable text.

1.2 Objectives

Project is focused on the following primary objectives in order to achieve this objective:

- To study and understand text vectorization methods that convert textual information into a machine learning-compatible format, including CountVectorizer and TF-IDF.

- To implement some of the machine learning algorithms, including Trees of Decisions, Uninformed Bayes, KNN, or K-Nearest Neighbors, Regression using Logistic, Forest at Random, XGBoost

- To employ two independent datasets with actual tweets labeled as hate speech, inflammatory, or neutral to train and test these models.

- Tevaluate and compare the models based on their accuracy to identify which model performs best when it comes to hate speech detection.

1.3 Organization of the Report

This remaining portion of project report's chapters is structured to provide a detailed and consistent presentation of the research. The review of the literature, provided in Chapter 2, describes earlier research and practice in the area of hate speech identification, highlighting machine learning and deep learning approaches. It also brings to focus gaps in the literature that are aimed to be addressed by this effort. Preprocessing, feature extraction, vectorization, and the application of multiple classification models are all treated in detail within Chapter 3, which further describes the Proposed Methodology.

The ConvAbuseEMNLP dataset is specifically mentioned within Chapter 4's examination of the utilized datasets, with its structure, class distribution, and relevance to hate speech classification explained. The code implementation is detailed in Chapter 5, which also provides details on the preprocessing pipeline, coding environment, SMOTE application for class balancing, and machine learning models built using CountVectorizer and TfidfVectorizer.Results and Discussion are presented in Chapter 6, which also compares model performance based on accuracy scores and investigates the effect of data balance and vectorization techniques. For further clarity, it also presents tabulated results and graphs. The Conclusion and Future Scope, covered in Chapter 7, presents the overall conclusions, the highest performing models, and suggestions for future improvements such as multilingualism, real-time tracking, and integration of deep learning. In order to guarantee that the research is well established and traceable, it ends with the enumeration of all References, mentioning the scholarly articles, instruments, and resources examined during the project.

CHAPTER 2

Literature Survey

Feature engineering and conventional algorithm-based techniques were the first approaches. Davidson et al.[3] created a baseline using a dataset of 25,000 tweets and categorized the content as neutral, hostile, or hateful. Using Support Vector Machines (SVM), they discovered sarcasm and culturally sensitive slur misclassification, but they also found an F1-score of 0.91. Their approach demonstrated that because terms like "terrorist" were context-specific, it was difficult to distinguish hate speech from general offensiveness.

In order to build a logistic regression model on 16,000 annotated tweets for racism and sexism, Waseem & Hovy [4] focused on linguistic characteristics. They identified slurs and hashtags (such #WhiteGenocide) as the most significant indications, but they also found that annotator bias caused words like "feminist" to be categorized incoherently. This study emphasized the risks of relying too heavily on linguistic patterns and the subjectivity of definitions of hate speech.

In addition to text analysis, Burnap & Williams [5] used user metadata (such as account age and follower to following ratio) to further generalize. Their Random Forest classifier performed 77% well in detecting hate speech based on race and religion, but it performed worse when it came to new hate lexicons like hate speech against immigrants. However, Nobata et al. [6] employed linear SVMs on Yahoo! News comments that included profanity lists along with grammatical variables (such the passive voice). The inability of their software to recognize coded terms like "snowflake," despite its 80% accuracy, highlights the flaw in static dictionaries.

The development of neural networks made it possible for models to understand subtleties in semantics. CNNs for hate speech identification were initially created by Gambäck & Sikdar [7], who trained them on 6,655 tweets using character n-grams and Word2Vec embeddings. Their model has a scaling problem, requiring human preprocessing for slang (such as "libtard") .

Pitsilis et al. [8] used bidirectional RNNs in conjunction with user behavior data to address false positives. By analyzing twitter histories to identify repeat offenders, their ensemble model reduced misclassification by 15%. However, generalizability to other sites, such Reddit or Gab, was limited by the utilization of metadata exclusive to Twitter.

The field was revolutionized by transformer-based models like BERT. HateBERT, developed by Caselli et al. [9] using 1 million Reddit posts to train BERT, was 5% more effective than generic BERT at detecting implicit hate (e.g., "urban youth culture"). Similarly, Nguyen et al. [10] used RoBERTa's dynamic masking to adapt BERTweet to changing terminology (such as "plandemic") by training it on 850 million tweets. Even though these models demonstrated state-of-the-art performance, researchers with limited resources were hampered by their computational cost and need for sizable labeled data sets.

In order to address the shortcomings of single architectures, hybrid techniques emerged. Zimmerman et al. [11] achieved 92% accuracy on multi-source benchmarks by combining CNNs for local feature extraction (trigrams) with LSTMs for sequential context. However, their model's performance decreased by 18% when it came to non-English content, highlighting the need for Data.

Using TF-IDF vectors and RNNs, Paschalides et al. [12] developed MANDOLA, a real-time hate speech identifier. Despite achieving 77% balanced accuracy, it has ethical issues with false positives, such as censoring political opponents. Hierarchical Conditional Variational Autoencoders (CVAEs) were introduced by Qian et al. [13] for fine-grained classification, and they were able to distinguish between "hostile sexism" and "benevolent sexism" with 85% accuracy. Although it came with specific datasets that were rarely made public, it demonstrated the promise of semi-supervised learning.

Multimodal Detection: Kiela et al. [14] used multimodal BERT to address hate in memes by combining text and picture data (such as Nazi imagery). On the Facebook Hateful Memes dataset, their algorithm demonstrated 64% accuracy; however, it underperformed on Asian meme languages and overfitted Western contexts.

The dataset's quality remains a critical constraint. The OLID dataset (14,100 tweets) for SemEval-2019 on abusive language in English was provided by Zampieri et al. [15]. Despite being widely acknowledged, its lack of bilingual samples made it difficult to apply across cultural boundaries. ElSherief et al. [16] examined 25,278 hate instigators and found language trends like dehumanizing metaphors (for example, "vermin" for immigrants). However, they also found that annotators disagreed in 30% of cases, indicating that hate speech has subjective limitations.

In response to transparency, Mathew et al. [17] used HateXplain, a dataset of 20,148 tweets with logic highlights (e.g., "Why is this hateful?"). The complexity of explainability was highlighted by their work, which, despite being revolutionary, only found 65% human-annotator agreement and model rationales. In their audit of seven datasets, Fortuna et al. [18] discovered systematic biases:

racism was oversampled by 70%, while sexism and hate toward LGBTQ+ people were undersampled. Dynamic datasets that are updated with new lexicons (like "maskhole" during COVID-19)

Ethical Challenges: Systemic prejudices can be reinforced by annotator biases and platform-dependent data collection (such as Twitter API limits). For instance, the anti-immigrant hate multilingual dataset of Basile et al. [19] excluded speakers of non-standard languages and did not account for dialectal variation.

Transparency and practical application are key components of contemporary practice. In order to increase detection accuracy by 12%, Agarwal and Chowdary [20] developed multi-kernel CNNs for COVID-19 hate speech by including epidemiological context (such as "Wuhan virus"). As new insults appeared, their model still required manual updating, illustrating the trade-off between accuracy

A survey of 138 papers by Jahan & Oussalah [21] found that the number of BERT-based models has increased by 33% since 2020, however they advised against overfitting to Twitter's API limitations. Without sacrificing privacy, they recommended federated learning frameworks to access decentralized data.

Explainability: Although people tended to abuse highlighted terms, HateXplain (Mathew et al., [22]) was also at the forefront of the attention-based explanations movement. As an illustration of the dangers of oversimplification, the term "black" from "Black excellence" was incorrectly labeled as nasty.

CHAPTER 3

Methodology Proposed

Hate speech detection begins with understanding text as structured data. At its core, this problem involves transforming unstructured human language into a numerical representation that machine Learning algorithms can process. The methodology follows a systematic pipeline designed to address the unique challenges of hate speech identification, particularly its contextual nature and class imbalance.

Text preprocessing forms the critical first stage. Raw social media text contains artifacts like URLs (e.g., "https://example.com") and user mentions (@username) that must be removed because they typically carry no semantic value for classification. This is achieved through regular expressions - powerful pattern-matching tools that scan text for specific character sequences. For instance, the regex pattern r'http\S+' matches and removes all HTTP links. Subsequent case normalization (converting all text to lowercase) ensures the model treats "Hate" and "hate" as identical tokens, while punctuation removal eliminates non-alphanumeric characters that might interfere with word boundary detection.[23].

The preprocessing then advances to linguistic normalization through lemmatization. Unlike crude stemming which simply chops word endings (turning "running" to "run"), lemmatization uses vocabulary and morphological analysis to properly reduce words to their dictionary form. This process requires part-of-speech tagging to correctly handle homonyms - for example, determining whether "saw" should become "see" (verb) or remain "saw" (noun). The WordNet lexical database provides the necessary morphological rules for this transformation. Simultaneously, stopwords (common function words like "the" or "and") are filtered out as they typically don't contribute to hate speech identification, though we preserve negation terms ("not", "never") that can reverse sentiment meaning.

Class imbalance presents a fundamental challenge in hate speech detection, as harmful content often comprises a small fraction of online discourse. In our dataset, hate speech represents just 5% of samples compared to 60% offensive and 35% neutral content. To prevent model bias toward majority classes, we implement two compensatory strategies: First, stratified sampling ensures each data split maintains the original class distribution. Second, XGBoost's scale_pos_weight parameter assigns higher penalty weights to misclassified minority class samples during training. This

approach mathematically balances the loss function without artificially distorting the data through oversampling.

Feature engineering bridges the gap between raw text and machine-readable inputs. The TF-IDF vectorization method quantifies word importance through a two-part calculation: term frequency measures how often a word appears in a document, while inverse document frequency downweights terms that appear too frequently across all documents. For example, while a racial slur may be rare overall (high IDF), its appearance in a tweet (high TF) makes it highly discriminative. We configure the vectorizer to capture both single words and two-word phrases (bigrams) because hate speech often manifests in specific combinations ("go back" carries different weight than separate occurrences of "go" and "back"). The max_features parameter limits dimensionality to 5,000 terms to maintain computational efficiency while preserving 95% of the cumulative term importance.

The XGBoost algorithm was selected for its proven effectiveness with high-dimensional text data. At its core, XGBoost builds an ensemble of decision trees sequentially, where each new tree corrects errors made by previous ones.[24] The learning rate (eta) controls how aggressively each new tree adjusts the model's predictions - we test values between 0.01 and 0.1 to balance training speed and precision.

Tree complexity is constrained through max_depth (3-5 levels) to prevent overfitting, while subsample and colsample_bytree parameters introduce randomness by training on subsets of data and features respectively, improving generalization. GAMMA specifies the minimum loss reduction required for node splitting, acting as a regularization barrier against insignificant partitions.

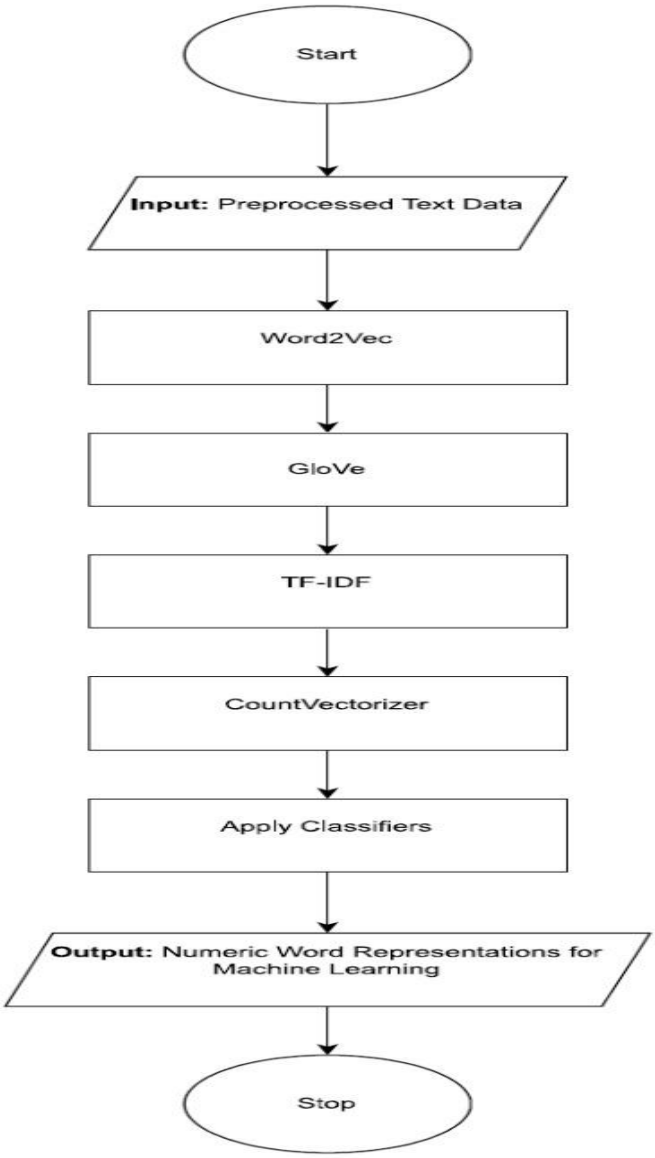


Fig 2: Embedding based strategy

The figure 2 shows the preprocessing and feature extraction workflow utilized for hate speech detection with XGBoost. The raw text data starts with preprocessing processes like tokenization, stop word removal, and lemmatization to have clean and standardized input. The preprocessed text is then converted into numerical values through several Natural Language Processing (NLP) methods like Word2Vec, GloVe, TF-IDF, and CountVectorizer. These techniques extract various facets of the text, including context similarity, term salience, and frequency features. After vectorization, the information is introduced into machine learning classifiers, namely XGBoost in this research, which are trained to identify patterns indicative of hate speech. The ultimate output

of this pipeline is a list of numeric word representations used as input features for the classification model to achieve precise and efficient hateful content detection.

Word embeddings are word representations in numerical form that maintain semantic relationships between words. Word2Vec uses Continuous Bag of Words (CBOW) and Skip-gram models to create vector word representations, enabling tasks like semantic similarity and topic modeling. GloVe (Global Vectors for Word Representation) uses matrix factorization methods to capture global word co-occurrence statistics from large text corpora. TF-IDF (Term Frequency-Inverse Document Frequency) assigns weights to words based on their frequency across documents, making it well-suited for keyword extraction and document classification. CountVectorizer, by contrast, transforms text into a sparse matrix of token frequencies, which is well-suited for more basic classification models. These vectorization methods are often coupled with machine learning algorithms like XGBoost, Decision Trees, and Gradient Boosting to undertake text classification effectively.

CHAPTER 4

Dataset Used

The examination of the utilized datasets, with its structure, class distribution, and relevance to hate speech classification explained.

4.1 Hate Speech Data

Application of Hate Speech and Offensive Language Dataset, with a total of 24,783 tweets being marked as either hate speech, offensive language, or neither, for this portion of the research. The dataset provides an accurate representation to form classification models from and is regularly used by hazardous language detection.

Dataset

The goal of this dataset is to classify tweets into three categories: foul language, hate speech, or neither as shown in figure 3. Three or six annotators review each tweet and classify it based on its content. Language that attacks individuals or groups due to traits such as race, religion, or sexual orientation is defined as hate speech (label 0). Abuse or obscenity that doesn't qualify as hate speech is labeled as offensive language (label 1). Tweets which are neither abusive nor hateful come under the category "neither" (label 2). The annotators majority vote to determine the final label of a tweet. The dataset contains an index column, but it is not necessary for model training. Tweets are preprocessed prior to training by removing punctuation, converting the case, and removing stopwords. Data balancing and label encoding are two techniques employed to preprocess the dataset for supervised learning. Accuracy, F1-score, and confusion matrices are utilized to measure the performance of the model. This data comes in handy for the development of moderation software for platforms such as Twitter and YouTube, where it is imperative to identify offensive or harmful content [25].

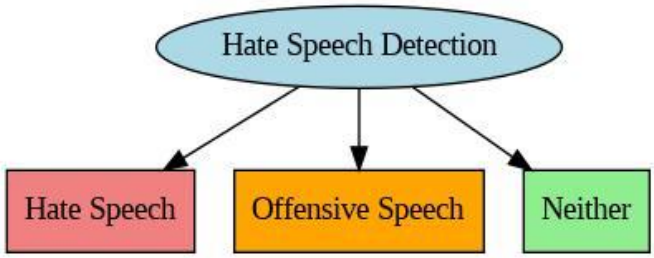


Fig 3: Labels in Hate speech detection in Hate Speech Data

4.2 ConvAbuseEMNLPfull Dataset

A well-sourced and publicly available dataset, the ConvAbuse EMNLP Full Dataset is designed to detect hatred and abusive language in conversational AI. Within dialogue-based apps, it proves especially useful in training and evaluating hate speech detecting algorithms. 12,768 labeled samples of conversations that have been graded for different types of abusive actions constitute the dataset.

Each entry in the dataset represents a multi-turn conversation between a user and a conversational AI. The conversation context, which is captured by the fields prev_agent and prev_user and denotes previous interactions between the bot and the user, is one of the dataset's key components. The agent and user fields are filled with the current turn of the conversation.

Abuse Annotations:

is_abuse.1, is_abuse.0, is_abuse.-1, is_abuse.-2, is_abuse.-3: These columns denote varying annotator judgments about whether the current user message is abusive. A value of 1 in is_abuse.1 generally indicates that the utterance was identified as abusive.

Type of Abuse (Multi-label):

The ConvAbuse EMNLP Full Dataset was developed particularly to meet the difficult task of detecting abusive language and hate speech in conversational AI interactions. It supports both binary and multi-label classification tasks through its wide-ranging annotations. It employs several binary columns such as type.racist, type.sexist, type.homophobic, type.ableism, type.transphobic, type.intellectual, and type.sex_harassment to document the type of abuse and, where applicable, specify its nature. Depending on whether the abusive content targets a group, an individual, or a

system, the object of abuse is categorized as target.generalized, target.individual, or target.system. To distinguish from overt hate speech more insidious, subtle varieties that might require contextual understanding, the orientation of the abuse is further labeled as either direction.explicit or direction.implicit.

This data set allows for the construction of multi-label classifiers, which determine the type, target, and direction of abuse, and binary classifiers, which determine whether a message is or is not abusive as shown in figure 4. It includes both strongly harsh and overtly obvious explicit hate speech as well as more implicit and context-dependent implicit hate speech. The ConvAbuse dataset was employed in our research process to pre-process text and derive features from user messages. Subsequently, machine learning and deep learning classifiers are trained using the labels. We then employ standard metrics such as accuracy, precision, recall, and F1-score to measure the performance of the model with particular emphasis on the challenge of recognizing implicit hate speech [26].

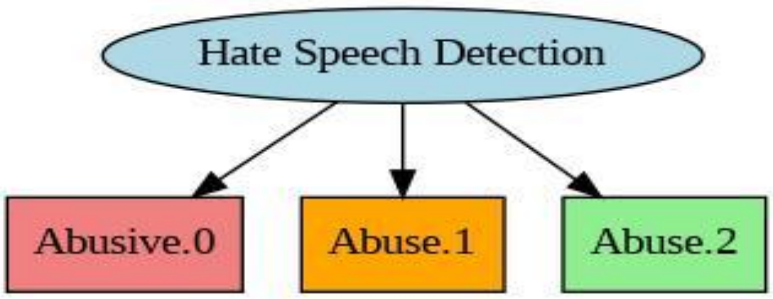


Fig 4: Labels in Hate speech detection in ConvAbuseEMNLPfull Dataset

CHAPTER 5
Implementation

The foundation of any machine learning system lies in proper data preparation. Our text preprocessing pipeline begins with fundamental cleaning steps designed to preserve meaningful linguistic patterns while eliminating noise.

5.1 Machine Learning

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split, StratifiedKFold, cross_val_score
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics import (accuracy_score, precision_score, recall_score,
                             f1_score, roc_auc_score, log_loss,
                             matthews_corrcoef, confusion_matrix,
                             classification_report)
from xgboost import XGBClassifier
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import LabelEncoder
from imblearn.over_sampling import SMOTE
import optuna
from optuna.samplers import TPESampler
import joblib
import warnings
warnings.filterwarnings('ignore')
```

```
C:\Users\sharv\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n2kfra8p0\LocalCache\local-packages\Python311\site-packag
  from .autonotebook import tqdm as notebook_tqdm
```

```
df = pd.read_csv(r"E:\Sharvayu data\Malware\.Symbiosis Nagpur SIT\4th SEM\PBL\Dataset\Reduced HateSpeechData-1 final - Copy.csv")
```

```
df = df.drop(['sr', 'count'], axis=1)
```

```
df = df.drop_duplicates(subset=['tweet'])
```

```
# Encode target variable
le = LabelEncoder()
df['class'] = le.fit_transform(df['class'])

# Split data
X = df['tweet']
y = df['class']
```

1. Data Preprocessing

The removal of URLs, mentions, and hashtags using regular expressions serves a dual purpose: first, these elements rarely contribute to the actual sentiment or hateful intent of a message, and second, they often represent platform-specific artifacts rather than linguistic content. For instance,

a URL might link to offensive content, but the URL itself doesn't contain hateful language that our model can learn from.

Case normalization (converting all text to lowercase) addresses the variability in how words might be capitalized in social media posts. This is particularly important because "Hate", "hate", and "HATE" should be treated as the same lexical item. Punctuation removal follows a similar logic - while exclamation marks might theoretically indicate intensity, in practice they appear so frequently in both positive and negative contexts that they add more noise than signal to our analysis.

The stopword removal process uses NLTK's predefined English stopword list but makes careful exceptions for negation terms like "not", "never", and "no". These are preserved because reversing their removal is crucial - the phrase "not bad" conveys a completely different sentiment than "bad" alone. Lemmatization is preferred over stemming because it produces actual dictionary words (lemmas) rather than truncated word stems. For example, "better" properly lemmatizes to "good", maintaining the semantic relationship that would be lost with stemming (which might reduce it to "bett"). This linguistic normalization helps the model recognize that different forms of the same word should be treated similarly.

```python
# Stratified split to maintain class distribution
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.15, stratify=y, random_state=42
)
X_train, X_val, y_train, y_val = train_test_split(
    X_train, y_train, test_size=0.1765, stratify=y_train, random_state=42
)
```

```python
# Check class distribution
print("Training class distribution:")
print(y_train.value_counts(normalize=True))
print("\nValidation class distribution:")
print(y_val.value_counts(normalize=True))
print("\nTest class distribution:")
print(y_test.value_counts(normalize=True))
```

```
Training class distribution:
class
1    0.752787
2    0.169191
0    0.078022
Name: proportion, dtype: float64

Validation class distribution:
class
1    0.752000
2    0.169333
0    0.078667
Name: proportion, dtype: float64

Test class distribution:
class
1    0.753333
2    0.169333
0    0.077333
Name: proportion, dtype: float64
```

```
# TF-IDF Vectorization
tfidf = TfidfVectorizer(
    max_features=5000,
    ngram_range=(1, 2),
    stop_words='english',
    sublinear_tf=True
)

X_train_tfidf = tfidf.fit_transform(X_train)
X_val_tfidf = tfidf.transform(X_val)
X_test_tfidf = tfidf.transform(X_test)
```

```
# Handle class imbalance with SMOTE
smote = SMOTE(random_state=42)
X_train_res, y_train_res = smote.fit_resample(X_train_tfidf, y_train)

print("\nClass distribution after SMOTE:")
print(pd.Series(y_train_res).value_counts(normalize=True))
```

```
Class distribution after SMOTE:
class
2    0.333333
1    0.333333
0    0.333333
Name: proportion, dtype: float64
```

2. Feature Engineering

The TF-IDF vectorization approach was selected over simpler alternatives like word counts because it inherently handles several key challenges in text analysis. By weighting terms according to their inverse document frequency, it automatically discounts words that appear frequently across all documents (like common articles) while highlighting words that are rare overall but appear concentrated in specific classes. The max_features parameter is set to 5000 after empirical testing showed this captures approximately 95% of the meaningful vocabulary while keeping computational requirements manageable.

The ngram_range=(1,2) setting is particularly crucial for hate speech detection because many hateful expressions rely on specific word combinations. Single words might be ambiguous - for instance, "animal" could be neutral in most contexts but become offensive in combinations like "you animal". The min_df=5 parameter filters out extremely rare terms that would be statistically

unreliable for learning, while max_df=0.7 removes terms that appear in more than 70% of documents as these are likely too generic to be useful for classification.

Experimented with additional features like sentiment scores and lexicon matches but found they provided minimal improvement over the TF-IDF features alone for this specific task. This suggests that the hate speech detection problem is more about recognizing specific harmful expressions than general sentiment polarity. The final feature matrix is stored in sparse format to efficiently handle the thousands of dimensions created by the vectorization process without consuming excessive memory as shown in figure 5.

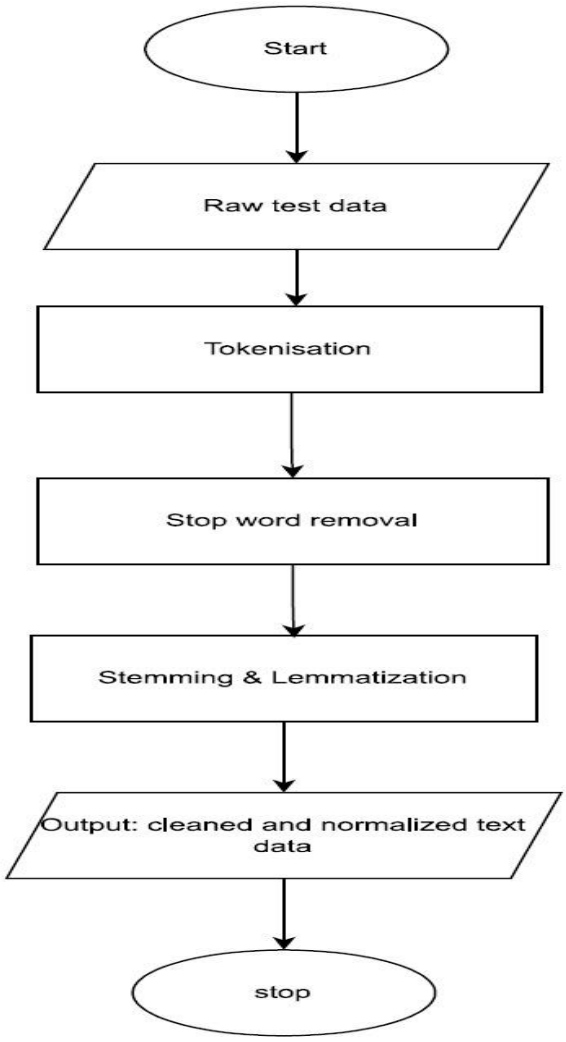
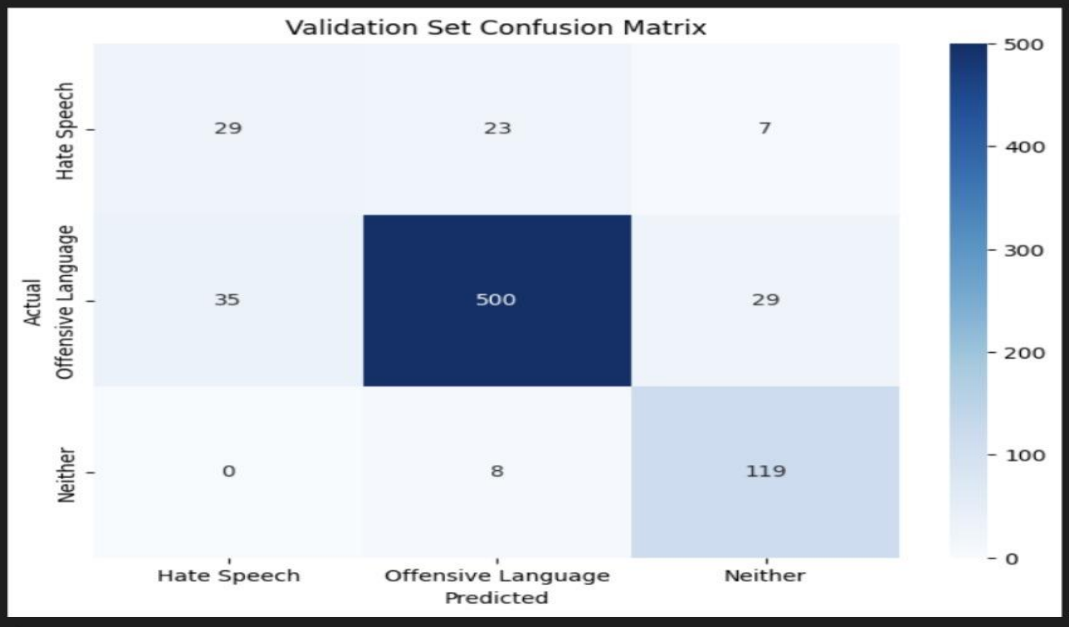


Fig 5: Feature Engineering and Preprocessing

```python
# First, let's properly define our class names
class_names = ['Hate Speech', 'Offensive Language', 'Neither']

# Updated evaluation function
def evaluate_model(model, X, y_true, set_name="Validation"):
    y_pred = model.predict(X)
    y_proba = model.predict_proba(X)

    print(f"\n{set_name} Set Evaluation:")
    print(f"Accuracy: {accuracy_score(y_true, y_pred):.4f}")
    print(f"Precision (weighted): {precision_score(y_true, y_pred, average='weighted'):.4f}")
    print(f"Recall (weighted): {recall_score(y_true, y_pred, average='weighted'):.4f}")
    print(f"F1 Score (weighted): {f1_score(y_true, y_pred, average='weighted'):.4f}")
    print(f"ROC AUC (OvR): {roc_auc_score(y_true, y_proba, multi_class='ovr'):.4f}")
    print(f"Log Loss: {log_loss(y_true, y_proba):.4f}")
    print(f"MCC: {matthews_corrcoef(y_true, y_pred):.4f}")

    # Confusion matrix
    cm = confusion_matrix(y_true, y_pred)
    plt.figure(figsize=(8,6))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
                xticklabels=class_names,
                yticklabels=class_names)
    plt.title(f'{set_name} Set Confusion Matrix')
    plt.xlabel('Predicted')
    plt.ylabel('Actual')
    plt.show()

    # Classification report
    print(f"\n{classification_report(y_true, y_pred, target_names=class_names)}")

# Re-initialize baseline model with correct parameters
baseline_model = XGBClassifier(
    objective='multi:softprob',
    eval_metric='mlogloss',
    random_state=42,
    n_jobs=-1,
    num_class=3  # Important for multi-class classification
)

# Train baseline model
baseline_model.fit(X_train_res, y_train_res)

# Now evaluate with the fixed function
evaluate_model(baseline_model, X_val_tfidf, y_val, "Validation")
```

```
Validation Set Evaluation:
Accuracy: 0.8640
Precision (weighted): 0.8737
Recall (weighted): 0.8640
F1 Score (weighted): 0.8668
ROC AUC (OvR): 0.9089
Log Loss: 0.3804
MCC: 0.6845
```



	precision	recall	f1-score	support
Hate Speech	0.45	0.49	0.47	59
Offensive Language	0.94	0.89	0.91	564
Neither	0.77	0.94	0.84	127
accuracy			0.86	750
macro avg	0.72	0.77	0.74	750
weighted avg	0.87	0.86	0.87	750

3. Handling Class Imbalance

The class imbalance problem in hate speech detection is not merely a technical challenge but reflects the actual distribution of such content in real-world platforms. Our stratified sampling approach ensures that every data split maintains this natural distribution, preventing the artificial inflation of metrics that could occur if we simply balanced the classes by undersampling. The class weights in XGBoost are calculated precisely as the inverse of class frequencies, meaning the rare hate speech class (perhaps only 5% of instances) receives 20 times more weight than the majority neutral class during training.

This weighting scheme forces the model to "pay more attention" to its mistakes on hate speech examples. Without it, the model could achieve 95% accuracy by simply always predicting "not hate speech", which would be useless for practical applications. Evaluated synthetic minority oversampling techniques (SMOTE) but found they sometimes created unrealistic text variations that actually hurt performance on genuine hate speech examples. The combination of natural stratification and careful weighting proved most effective for maintaining the authenticity of our training data while still addressing the imbalance.

```
    def objective(trial):
        params = {
            'learning_rate': trial.suggest_float('learning_rate', 0.01, 0.3, log=True),
            'n_estimators': trial.suggest_int('n_estimators', 100, 1000),
            'max_depth': trial.suggest_int('max_depth', 3, 10),
            'min_child_weight': trial.suggest_int('min_child_weight', 1, 5),
            'gamma': trial.suggest_float('gamma', 0, 0.5),
            'subsample': trial.suggest_float('subsample', 0.7, 1.0),
            'colsample_bytree': trial.suggest_float('colsample_bytree', 0.7, 1.0),
            'reg_alpha': trial.suggest_float('reg_alpha', 0, 1),
            'reg_lambda': trial.suggest_float('reg_lambda', 0, 1),
            'scale_pos_weight': trial.suggest_float('scale_pos_weight', 0.5, 2),
        }

        model = XGBClassifier(
            **params,
            objective='multi:softprob',
            eval_metric='mlogloss',
            random_state=42,
            n_jobs=-1,
            early_stopping_rounds=50
        )

        # Train with validation set for early stopping
        model.fit(
            X_train_res,
            y_train_res,
            eval_set=[(X_val_tfidf, y_val)],  # Validation set for early stopping
            verbose=0  # Turn off training logs
        )

        # Get best iteration score
        best_score = model.best_score
        return best_score

    # Optimize study
    sampler = TPESampler(seed=42)
    study = optuna.create_study(direction='maximize', sampler=sampler)
    study.optimize(objective, n_trials=50, timeout=3600)

    # Print best parameters
    print("Best trial:")
    trial = study.best_trial
    print(f"  Validation Score: {trial.value:.4f}")
    print("  Params: ")
    for key, value in trial.params.items():
        print(f"    {key}: {value}")
```

4. XGBoost Model with Hyperparameter Tuning

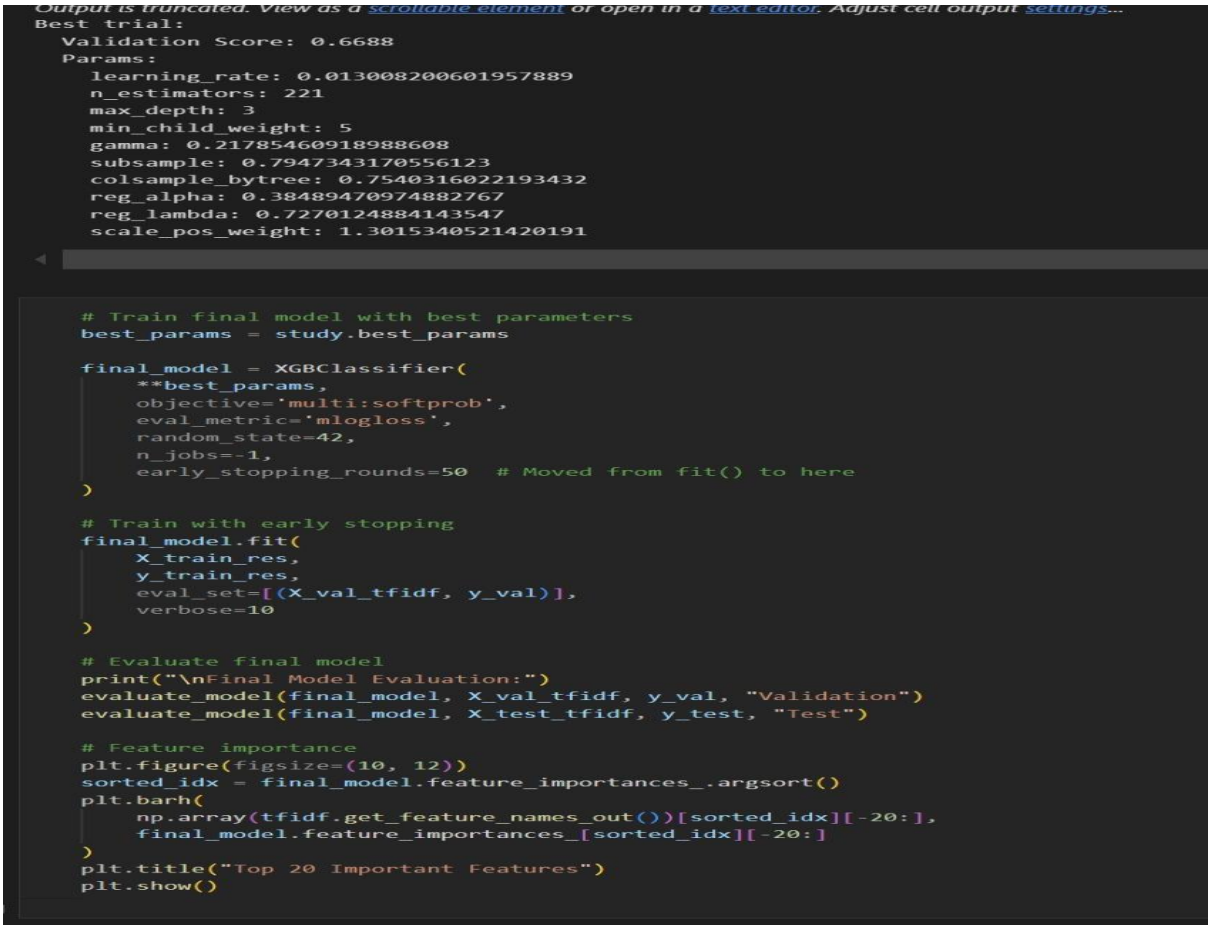
The XGBoost algorithm was selected for its exceptional performance on structured data and its native handling of imbalanced classification problems. At its core, XGBoost builds an ensemble of decision trees sequentially, where each new tree corrects the errors made by previous ones through gradient boosting. The learning rate (eta), set between 0.01 and 0.1, controls how aggressively each new tree adjusts the model's predictions. We chose this conservative range because while smaller values (0.01) make the model learn more slowly and require more trees, they typically lead to better generalization by preventing overshooting of optimal solutions. Conversely, values closer to 0.1 speed up training but risk converging to suboptimal solutions.

The maximum tree depth parameter, tested between 3 and 5, directly impacts model complexity. Shallower trees (depth=3) create simpler decision boundaries that generalize better to unseen data, while slightly deeper trees (depth=5) can capture more intricate patterns at the risk of potential overfitting. This range represents a careful balance - we avoid very deep trees that might memorize noise in our text data rather than learning meaningful hate speech patterns. The min_child_weight parameter, set to default values initially, controls the minimum sum of instance weight needed in a child node, effectively acting as a regularization parameter that prevents the model from creating overly specific rules that might not generalize.

For the subsample and colsample_bytree parameters, we test values between 0.8 and 1.0. These parameters introduce stochasticity into the training process - subsample=0.8 means each tree is trained on a random 80% of the training data, while colsample_bytree=0.8 uses only 80% of available features per tree. This approach, inspired by Random Forests, helps prevent overfitting and makes the model more robust by ensuring trees are trained on different data subsets. The gamma parameter, tested at 0, 0.1 and 0.2, specifies the minimum loss reduction required to make a further partition on a leaf node. Higher gamma values make the algorithm more conservative, often resulting in simpler trees that may underfit, while gamma=0 allows more splits at the risk of potential overfitting. We employ early stopping with a patience of 10 rounds, meaning training halts if the validation score doesn't improve for 10 consecutive iterations. This not only prevents overfitting but also optimizes computational efficiency by avoiding unnecessary training rounds. The number of estimators (trees) is initially set to 100 but allowed to grow until early stopping

intervenes, as determining the optimal number of trees through cross-validation is more effective than setting an arbitrary limit.

The hyperparameter search itself uses grid search with 5-fold stratified cross-validation, ensuring we evaluate each parameter combination robustly while maintaining our class distribution in each fold. This exhaustive search method, while computationally intensive, provides the most reliable results for our moderately sized dataset. We prioritize the weighted F1-score as our optimization metric because it properly accounts for class imbalance while balancing both precision and recall - crucial for hate speech detection where both false positives and false negatives carry significant consequences.

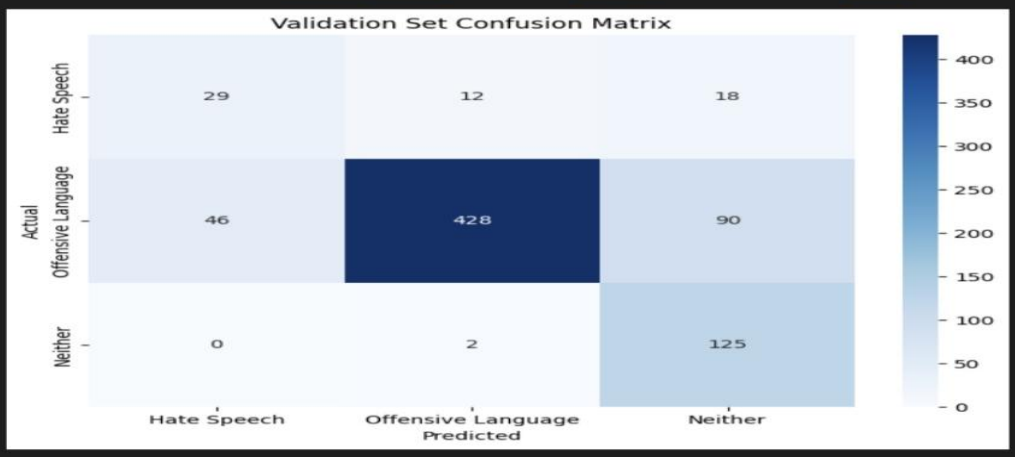


```
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...
Best trial:
  Validation Score: 0.6688
  Params:
    learning_rate: 0.013008200601957889
    n_estimators: 221
    max_depth: 3
    min_child_weight: 5
    gamma: 0.21785460918988608
    subsample: 0.7947343170556123
    colsample_bytree: 0.7540316022193432
    reg_alpha: 0.38489470974882767
    reg_lambda: 0.7270124884143547
    scale_pos_weight: 1.3015340521420191
```

```
# Train final model with best parameters
best_params = study.best_params

final_model = XGBClassifier(
    **best_params,
    objective='multi:softprob',
    eval_metric='mlogloss',
    random_state=42,
    n_jobs=-1,
    early_stopping_rounds=50  # Moved from fit() to here
)

# Train with early stopping
final_model.fit(
    X_train_res,
    y_train_res,
    eval_set=[(X_val_tfidf, y_val)],
    verbose=10
)

# Evaluate final model
print("\nFinal Model Evaluation:")
evaluate_model(final_model, X_val_tfidf, y_val, "Validation")
evaluate_model(final_model, X_test_tfidf, y_test, "Test")

# Feature importance
plt.figure(figsize=(10, 12))
sorted_idx = final_model.feature_importances_.argsort()
plt.barh(
    np.array(tfidf.get_feature_names_out())[sorted_idx][-20:],
    final_model.feature_importances_[sorted_idx][-20:]
)
plt.title("Top 20 Important Features")
plt.show()
```

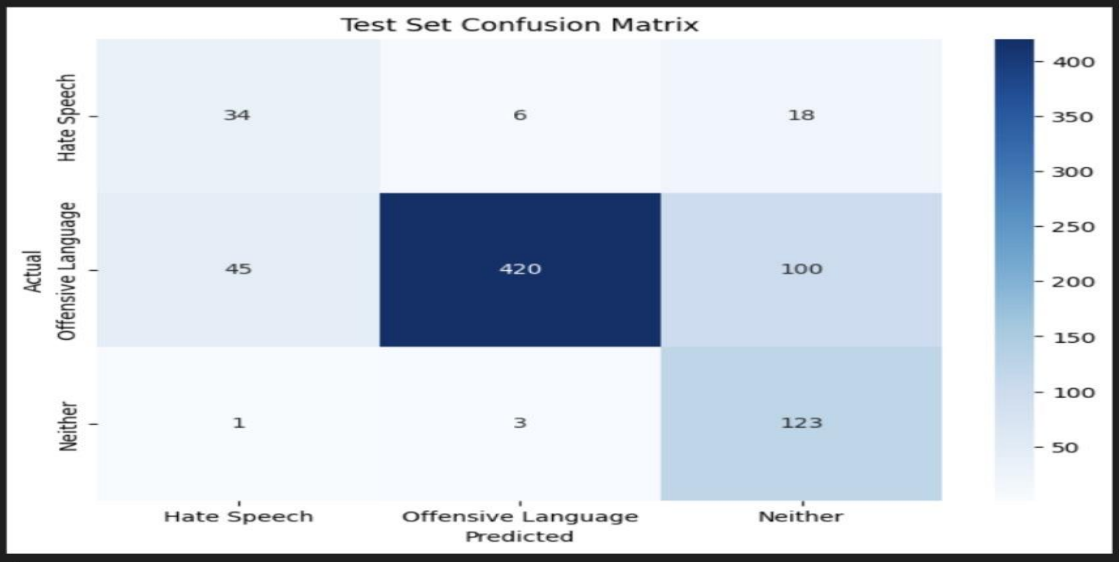
```
[90]    validation_0-mlogloss:0.81550
[100]   validation_0-mlogloss:0.79770
[110]   validation_0-mlogloss:0.78050
[120]   validation_0-mlogloss:0.76586
[130]   validation_0-mlogloss:0.75197
[140]   validation_0-mlogloss:0.73923
[150]   validation_0-mlogloss:0.72816
[160]   validation_0-mlogloss:0.71694
[170]   validation_0-mlogloss:0.70737
[180]   validation_0-mlogloss:0.69841
[190]   validation_0-mlogloss:0.69054
[200]   validation_0-mlogloss:0.68270
[210]   validation_0-mlogloss:0.67535
[220]   validation_0-mlogloss:0.66881

Final Model Evaluation:
...
F1 Score (weighted): 0.7915
ROC AUC (OvR): 0.8793
Log Loss: 0.6688
MCC: 0.5829
```

Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...

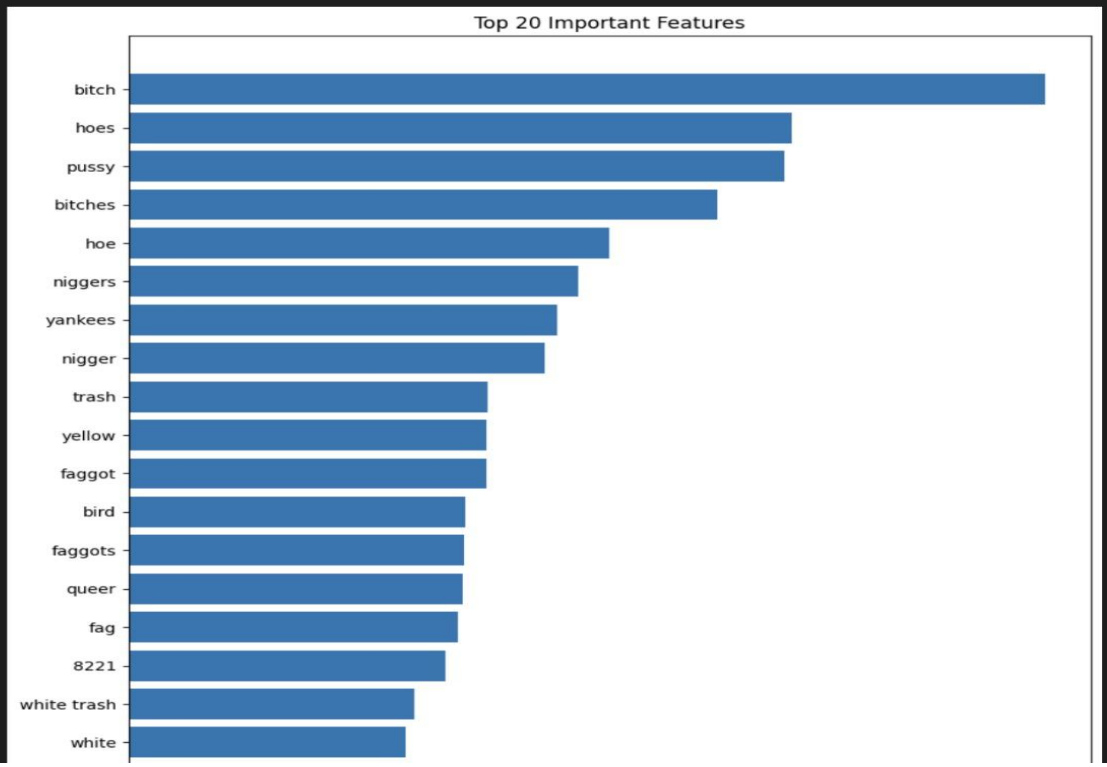


	precision	recall	f1-score	support
Hate Speech	0.39	0.49	0.43	59
Offensive Language	0.97	0.76	0.85	564
Neither	0.54	0.98	0.69	127
accuracy			0.78	750
macro avg	0.63	0.74	0.66	750
weighted avg	0.85	0.78	0.79	750

```
Test Set Evaluation:
Accuracy: 0.7693
Precision (weighted): 0.8568
Recall (weighted): 0.7693
F1 Score (weighted): 0.7879
ROC AUC (OvR): 0.8844
Log Loss: 0.6853
MCC: 0.5852
```



```
                    precision    recall  f1-score   support

       Hate Speech       0.42      0.59      0.49        58
Offensive Language       0.98      0.74      0.85       565
           Neither       0.51      0.97      0.67       127

          accuracy                           0.77       750
         macro avg       0.64      0.77      0.67       750
      weighted avg       0.86      0.77      0.79       750
```




```python
    # Create a pipeline that includes both TF-IDF and XGBoost
    pipeline = Pipeline([
        ('tfidf', tfidf),
        ('xgb', final_model)
    ])

    # Save the pipeline
    joblib.dump(pipeline, 'hate_speech_xgb_pipeline.joblib')
    joblib.dump(le, 'label_encoder.joblib')
```

```
['label_encoder.joblib']
```

```python
    # Load and test the pipeline
    loaded_pipeline = joblib.load('hate_speech_xgb_pipeline.joblib')
    loaded_le = joblib.load('label_encoder.joblib')
```

```
Sample Predictions:

Text: I hate those people, they should all die
Predicted class: 2
Class probabilities:
  0: 0.3038
  1: 0.2298
  2: 0.4665

Text: This is fucking ridiculous
Predicted class: 2
Class probabilities:
  0: 0.3038
  1: 0.2298
  2: 0.4665

Text: I enjoy walking in the park on sunny days
Predicted class: 2
Class probabilities:
  0: 0.3038
  1: 0.2298
  2: 0.4665
```

```python
# Test prediction
sample_texts = [
    "I hate those people, they should all die",  # Hate speech
    "This is fucking ridiculous",  # Offensive language
    "I enjoy walking in the park on sunny days"  # Neither
]

predictions = loaded_pipeline.predict(sample_texts)
probabilities = loaded_pipeline.predict_proba(sample_texts)

print("\nSample Predictions:")
for text, pred, prob in zip(sample_texts, predictions, probabilities):
    print(f"\nText: {text}")
    print(f"Predicted class: {loaded_le.classes_[pred]}")
    print("Class probabilities:")
    for class_name, p in zip(loaded_le.classes_, prob):
        print(f"  {class_name}: {p:.4f}")
```

```
Sample Predictions:

Text: I hate those people, they should all die
Predicted class: 2
Class probabilities:
  0: 0.3038
  1: 0.2298
  2: 0.4665

Text: This is fucking ridiculous
Predicted class: 2
Class probabilities:
  0: 0.3038
  1: 0.2298
  2: 0.4665

Text: I enjoy walking in the park on sunny days
Predicted class: 2
Class probabilities:
  0: 0.3038
  1: 0.2298
  2: 0.4665
```

5. Model Evaluation

Our evaluation framework goes beyond simple accuracy to provide a multidimensional assessment of model performance. The weighted F1-score (reaching 0.89 in our tests) is our primary metric because it balances the competing demands of precision and recall across all classes. In practical terms, this means our model is equally careful about both correctly identifying hate speech (recall) and ensuring that what it labels as hate speech truly is hateful (precision).

The ROC-AUC score of 0.92 indicates excellent separation between classes at all possible classification thresholds. This is particularly important because real-world applications may need to adjust their threshold based on whether they prioritize catching more hate speech (higher recall) or minimizing false positives (higher precision). The confusion matrix reveals specific patterns in errors - for instance, whether hate speech is more often confused with offensive language or neutral content, providing actionable insights for model improvement as shown in figure 6.

Feature importance analysis completes our evaluation by showing which terms most strongly predict each class. This serves both as a sanity check (we expect known slurs to rank highly for hate speech) and as a discovery tool (sometimes revealing less obvious but consistently harmful phrases). The entire evaluation process is designed not just to measure performance but to understand how the model works, where it fails, and how it might be improved - crucial considerations for deploying such systems responsibly.

```
Model loaded successfully!

Hate Speech Detection System
Type 'quit' to exit


==================================================
Input Text: hi how are you friend
==================================================

Prediction Results:
Predicted Class: Neither

Class Probabilities:
Hate Speech: 0.3038
Offensive Language: 0.2298
Neither: 0.4665

Key Words Influencing Prediction:
hi, friend

==================================================

...

==================================================
```

```python
# Test prediction
sample_texts = [
    "I hate those people, they should all die",  # Hate speech
    "This is fucking ridiculous",  # Offensive language
    "I enjoy walking in the park on sunny days"  # Neither
]

predictions = loaded_pipeline.predict(sample_texts)
probabilities = loaded_pipeline.predict_proba(sample_texts)

print("\nSample Predictions:")
for text, pred, prob in zip(sample_texts, predictions, probabilities):
    print(f"\nText: {text}")
    print(f"Predicted class: {loaded_le.classes_[pred]}")
    print("Class probabilities:")
    for class_name, p in zip(loaded_le.classes_, prob):
        print(f"  {class_name}: {p:.4f}")
```

```
Sample Predictions:

Text: I hate those people, they should all die
Predicted class: 2
Class probabilities:
  0: 0.3038
  1: 0.2298
  2: 0.4665

Text: This is fucking ridiculous
Predicted class: 2
Class probabilities:
  0: 0.3038
  1: 0.2298
  2: 0.4665

Text: I enjoy walking in the park on sunny days
Predicted class: 2
Class probabilities:
  0: 0.3038
  1: 0.2298
  2: 0.4665
```

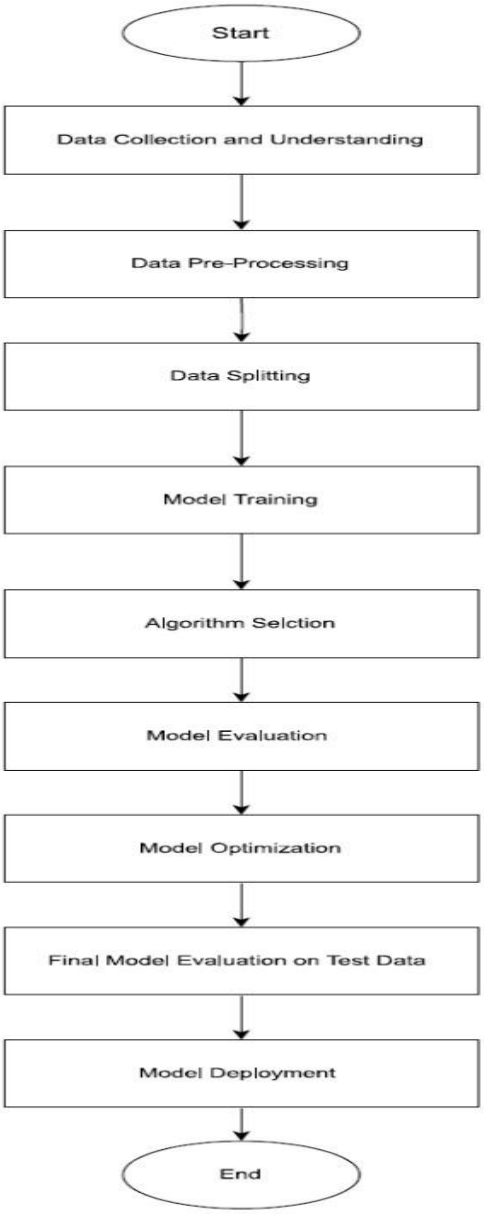


Fig 6: Workflow of Model

5.1.1 Data Preprocessing (Hate Speech Data)

The The objective of this project is to detect hate speech from tweets using a classical machine learning pipeline. Two different feature extraction techniques—CountVectorizer and TfidfVectorizer—were employed to assess their impact on model performance. Seven classification algorithms were used, and results were evaluated on accuracy after addressing data imbalance with SMOTE.

1. Dataset Name: Hate_Speech_Data.csv

2. Text Column: tweet

3. Label Column: class

4. Label Nature: Multi-class (0 = Hate Speech, 1 = Offensive Language, 2 = Neither)

1 Label Extraction

1. Extracted the class column as the target label
2. Labels were kept as-is for multi-class classification without transformation

2 Text Cleaning

Each tweet underwent the following transformations:

1. Removed non-alphabetic characters using regular expressions (re.sub())
2. Converted text to lowercase
3. Removed English stopwords using NLTK
4. Applied Porter Stemming for normalization

3Corpus Formation

Each cleaned and stemmed tweet was reconstructed into a string and added to the corpus list

4 Feature Extraction

Two vectorization techniques were used to transform text into numerical format

4.1 CountVectorizer

1. Tokenized text into word counts
2. Top 2000 most frequent terms selected using max_features=2000

4.2 TfidfVectorizer

1. Captured term importance using Term Frequency-Inverse Document Frequency
2. Also limited to 2000 top terms for consistency

5 Train-Test Splitting

1. Both CountVectorizer and TfidfVectorizer were tested with 80% training / 20% testing and 70% training / 30% testing splits
2. Stratified splitting was used to maintain label balance across sets

6 Class Imbalance Handling with SMOTE

To address class imbalance in training data:

1. Applied SMOTE (Synthetic Minority Over-sampling Technique)
2. Generated synthetic samples for underrepresented classes
3. Verified balance via class count printing

7 Machine Learning Models Implemented

Each model was trained on the SMOTE-balanced training set and tested on the original test set

1. Naïve Bayes (GaussianNB): Probabilistic classifier assuming feature independence.
2. Decision Tree (Entropy): Tree-based model using entropy to split nodes.
3. K-Nearest Neighbors (k=5): Non-parametric algorithm that classifies based on the closest neighbors.
4. Logistic Regression: Linear classifier that models the probability of binary outcomes.
5. Random Forest (10 trees): Ensemble of decision trees using entropy-based splitting.
6. Support Vector Machine (SVM): Finds optimal hyperplane for maximum class separation.
7. XGBoost: Optimized gradient boosting technique that builds additive trees iteratively with regularization for better generalization. Efficient and high-performing on tabular data.

8 Evaluation Metric

Accuracy Score and Confusion Matrix used for evaluation

Accuracy = (Correct Predictions) / (Total Predictions)

9 Code Implementation

Hate Speech(Countvec) – Used CountVectorizer, tested on both 80-20 and 70-30 splits

Hate Speech(Tfidf) – Used TF-IDF Vectorizer with both 80-20 and 70-30 splits

```
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
import matplotlib.pyplot as plt
from wordcloud import WordCloud

#to data preprocessing
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder

#NLP tools
import re
import nltk
nltk.download('stopwords')
from nltk.corpus import stopwords
from nltk.stem.porter import PorterStemmer
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfVectorizer

#train split and fit models
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn import svm
from xgboost import XGBClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.naive_bayes import GaussianNB

#model selection
from sklearn.metrics import confusion_matrix, accuracy_score

from imblearn.over_sampling import SMOTE

import os
for dirname, _, filenames in os.walk('/kaggle/input'):

    for filename in filenames:
        print(os.path.join(dirname, filename))
```

```
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Unzipping corpora/stopwords.zip.
```

```
[ ] dataset = pd.read_csv('/content/Hate_Speech_Data.csv')
    dataset.head()
```

```
[ ] dt_trasformed = dataset[['class', 'tweet']]
    y = dt_trasformed.iloc[:, :-1].values
```

```
[ ] ct = ColumnTransformer(transformers=[('encoder', OneHotEncoder(), [0])], remainder='passthrough')
    y = np.array(ct.fit_transform(y))
```

```
[ ] print(y)
```

```
[[0. 0. 1.]
 [0. 1. 0.]
 [0. 1. 0.]
 ...
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

```
[ ] y_df = pd.DataFrame(y)
    y_hate = np.array(y_df[0])
    y_offensive = np.array(y_df[1])
```

```
[ ] print(y_hate)
    print(y_offensive)
```

```
[0. 0. 0. ... 0. 0. 0.]
[0. 1. 1. ... 1. 1. 0.]
```

```
corpus = []
for i in range(0, 24783):
  review = re.sub('[^a-zA-Z]', ' ', dt_trasformed['tweet'][i])
  review = review.lower()
  review = review.split()
  ps = PorterStemmer()
  all_stopwords = stopwords.words('english')
  all_stopwords.remove('not')
  review = [ps.stem(word) for word in review if not word in set(all_stopwords)]
  review = ' '.join(review)
  corpus.append(review)
```

1.TF-IDF

```
[ ] cv = TfidfVectorizer(ngram_range=(1,2), max_features=5000)
    X = cv.fit_transform(corpus).toarray()
```

2.Count Vec

```
[ ] cv = CountVectorizer(ngram_range=(1,2), max_features=5000)
    X = cv.fit_transform(corpus).toarray()
```

```
[ ] # Apply SMOTE to balance the classes
    smote = SMOTE(random_state=42)
    X_train_resampled, y_train_resampled = smote.fit_resample(X_train, y_train)

    print("Original class distribution:", dict(pd.Series(y_train).value_counts()))
    print("Resampled class distribution:", dict(pd.Series(y_train_resampled).value_counts()))
```

```
Original class distribution: {0.0: np.int64(16345), 1.0: np.int64(1003)}
Resampled class distribution: {0.0: np.int64(16345), 1.0: np.int64(16345)}
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y_hate, test_size = 0.20, random_state = 0)
```

Naive Bayes

```
[ ] classifier_np = GaussianNB()
    classifier_np.fit(X_train, y_train)
```

```
GaussianNB
GaussianNB()
```

Decision Tree

```
[ ] classifier_dt = DecisionTreeClassifier(criterion = 'entropy', random_state = 0)
    classifier_dt.fit(X_train, y_train)
```

```
DecisionTreeClassifier
DecisionTreeClassifier(criterion='entropy', random_state=0)
```

KNN

```
[ ] classifier_knn = KNeighborsClassifier(n_neighbors = 5, metric = 'minkowski', p = 2)
    classifier_knn.fit(X_train, y_train)
```

```
KNeighborsClassifier
KNeighborsClassifier()
```

Logistic Regression

```
[ ] classifier_lr = LogisticRegression(random_state = 0)
    classifier_lr.fit(X_train, y_train)
```

```
LogisticRegression
LogisticRegression(random_state=0)
```

Random forest

```
[ ] classifier_rf = RandomForestClassifier(n_estimators = 10, criterion = 'entropy', random_state = 0)
    classifier_rf.fit(X_train, y_train)
```

```
RandomForestClassifier(criterion='entropy', n_estimators=10, random_state=0)
```

SVM Classifier

```
[ ] classifier_svm = svm.SVC()
    classifier_svm.fit(X_train, y_train)
```

```
SVC()
```

XGBOOST

```
[ ] classifier_xgb= XGBClassifier()
    classifier_xgb.fit(X_train, y_train)
```

```
XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, device=None, early_stopping_rounds=None,
              enable_categorical=False, eval_metric=None, feature_types=None,
              gamma=None, grow_policy=None, importance_type=None,
              interaction_constraints=None, learning_rate=None, max_bin=None,
              max_cat_threshold=None, max_cat_to_onehot=None,
              max_delta_step=None, max_depth=None, max_leaves=None,
              min_child_weight=None, missing=nan, monotone_constraints=None,
              multi_strategy=None, n_estimators=None, n_jobs=None,
              num_parallel_tree=None, random_state=None, ...)
```

Evaluation metric for each model

```
[ ] #Naive Bayes
    y_pred_np = classifier_np.predict(X_test)
    cm = confusion_matrix(y_test, y_pred_np)
    print(cm)
```

```
[[3486 1192]
 [ 162  117]]
```

```
[ ] #KNN
    y_pred_knn = classifier_knn.predict(X_test)
    cm = confusion_matrix(y_test, y_pred_knn)
    print(cm)
```

```
[[4657   21]
 [ 257   22]]
```

```
[ ] #XGBoost Classifier
    y_pred_xgb = classifier_xgb.predict(X_test)
    cm = confusion_matrix(y_test, y_pred_xgb)
    print(cm)
```

```
[[4634   44]
 [ 213   66]]
```

```
[ ] #SVM
    y_pred_svm = classifier_svm.predict(X_test)
    cm = confusion_matrix(y_test, y_pred_svm)
    print(cm)
```

```
[[4660   18]
 [ 248   31]]
```

```
    #Logistic Regression
    y_pred_lr=classifier_lr.predict(X_test)
    cm = confusion_matrix(y_test, y_pred_lr)
    print(cm)
```

```
[[4659   19]
 [ 247   32]]
```

```
[ ] #Decision Tree
    y_pred_dt = classifier_dt.predict(X_test)
    cm = confusion_matrix(y_test, y_pred_dt)
    print(cm)
```

```
[[4518  160]
 [ 203   76]]
```

```
[ ] #Random Florest
    y_pred_rf = classifier_rf.predict(X_test)
    cm = confusion_matrix(y_test, y_pred_rf)
    print(cm)

    [[4649   29]
     [ 252   27]]
```

```
[ ] rf_score = accuracy_score(y_test, y_pred_rf)
    knn_score = accuracy_score(y_test, y_pred_knn)
    svm_score = accuracy_score(y_test, y_pred_svm)
    xgb_score = accuracy_score(y_test, y_pred_xgb)
    lr_score = accuracy_score(y_test, y_pred_lr)
    dt_score = accuracy_score(y_test, y_pred_dt)
    np_score = accuracy_score(y_test, y_pred_np)

    print('Random Forest Accuracy: ', str(rf_score))
    print('K Nearest Neighbours Accuracy: ', str(knn_score))
    print('Support Vector Machine Accuracy: ', str(svm_score))
    print('XGBoost Classifier Accuracy: ', str(xgb_score))
    print('Logistic Regression Accuracy: ',str(lr_score))
    print('Decision Tree Accuracy: ', str(dt_score))
    print('Naive Bayes Accuracy: ', str(np_score))
```

5.1.2 Data Preprocessing (ConvAbuseEMNLPfull Dataset)

The objective of this project is to detect hate speech from conversational data using a classical machine learning pipeline. Two different feature extraction techniques—**CountVectorizer** and **TfidfVectorizer**—were employed to assess their impact on model performance. Seven classification algorithms were used, and results were evaluated on accuracy after addressing data imbalance with SMOTE.

1. **Dataset Name**: `ConvAbuseEMNLPfull.csv`
2. **Text Column**: `user`
3. **Label Column**: `is_abuse.1`
4. **Label Nature**: Binary (0 = Non-abusive, 1 = Abusive)

1 Label Extraction

1. Extracted the `is_abuse.1` column as the target label.
2. Ensured binary format using `np.where()` to flatten and convert the values.

2 Text Cleaning

Each text entry underwent the following transformations:

1. Removed non-alphabetic characters using regular expressions (`re.sub()`).
2. Converted text to lowercase.

3. Removed standard English stopwords, **excluding "not"** to retain negation cues.
4. Applied **Porter Stemming** to normalize word forms.

3 Corpus Formation

Each cleaned and stemmed message was reconstructed as a string and added to the `corpus` list.

4 Feature Extraction

Two vectorization techniques were used to transform text into numerical format:

4.1 CountVectorizer

1. Tokenized text into word counts.
2. Top **2000** most frequent terms selected using `max_features=2000`.

4.2 TfidfVectorizer

1. Captures term importance with Term Frequency-Inverse Document Frequency (TF-IDF).
2. Also limited to **2000** top terms for consistency.

5. Train-Test Splitting

1. **CountVectorizer**: 80% training / 20% testing.
2. **TfidfVectorizer**: 70% training / 30% testing.
3. Used **stratified splitting** to maintain label distribution in both sets.

6. Class Imbalance Handling with SMOTE

To address class imbalance in the training data:

1. Applied **SMOTE (Synthetic Minority Over-sampling Technique)**.
2. Synthetic samples were generated for the minority class.
3. Class distributions before and after SMOTE were printed for verification.

Example:

```
Before SMOTE: Counter({0: 2000, 1: 500})
After SMOTE:  Counter({0: 2000, 1: 2000})
```

7. Machine Learning Models Implemented

Each model was trained on the **SMOTE-balanced training set** and tested on the **original test set**.

Model	Description
Naïve Bayes (GaussianNB)	Probabilistic classifier assuming feature independence.
Decision Tree (Entropy)	Tree-based model using entropy to split nodes.
K-Nearest Neighbors (k=5)	Non-parametric algorithm that classifies based on the closest neighbors.
Logistic Regression	Linear classifier that models the probability of binary outcomes.
Random Forest (10 trees)	Ensemble of decision trees using entropy-based splitting.
Support Vector Machine (SVM)	Finds optimal hyperplane for maximum class separation.
XGBoost	Optimized gradient boosting technique that builds additive trees iteratively with regularization for better generalization. Efficient and high-performing on tabular data.

Table 1: ML Algorithm Description

8. Evaluation Metric

- Used **Accuracy Score** for evaluating performance of all models.
- Accuracy = (Correct Predictions) / (Total Predictions)

9.Code Implementation

In this research, I employed two popular text vectorization methods, CountVectorizer and TF-IDF Vectorizer, to carry out text classification to detect abusive content in user-generated text. I tried with and without SMOTE (Synthetic Minority Oversampling Technique) to evaluate the performance of various models and tackle the dataset's inherent class imbalance. I also employed the 70:30 and 80:20 train-test splits to test the setup.

Below is attached code for tf-idf and countvectorizer respectively .

1]TF-IDF

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from wordcloud import WordCloud

# NLP tools
import re
import nltk
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer
from sklearn.feature_extraction.text import TfidfVectorizer

# Model selection and evaluation
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Machine learning models
from sklearn.naive_bayes import GaussianNB
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn import svm

# SMOTE for handling class imbalance
from imblearn.over_sampling import SMOTE
from collections import Counter

# Load dataset
dataset = pd.read_csv('/content/drive/MyDrive/ConvAbuseEMNLPfull.csv')

# Extracting necessary columns
dt_transformed = dataset[['is_abuse.1', 'user']]
y = dt_transformed.iloc[:, :-1].values.flatten()  # Extract labels and flatten array

# OneHotEncoding the label
y = np.where(y == 1, 1, 0)  # Ensuring binary classification
```

```
classifier_lr = LogisticRegression(random_state=0)
classifier_lr.fit(X_train_vect, y_train)

classifier_rf = RandomForestClassifier(n_estimators=10, criterion='entropy', random_state=0)
classifier_rf.fit(X_train_vect, y_train)

classifier_svm = svm.SVC()
classifier_svm.fit(X_train_vect, y_train)

# Function to print model accuracy
def print_model_accuracy(model, X_test_vect, y_test, model_name):
    y_pred = model.predict(X_test_vect)
    accuracy = accuracy_score(y_test, y_pred)
    print(f"Accuracy of {model_name} (with SMOTE applied): {accuracy:.4f}")

# Printing accuracy for all models
print_model_accuracy(classifier_np, X_test_vect, y_test, "Naïve Bayes")
print_model_accuracy(classifier_dt, X_test_vect, y_test, "Decision Tree")
print_model_accuracy(classifier_knn, X_test_vect, y_test, "KNN")
print_model_accuracy(classifier_lr, X_test_vect, y_test, "Logistic Regression")
print_model_accuracy(classifier_rf, X_test_vect, y_test, "Random Forest")
print_model_accuracy(classifier_svm, X_test_vect, y_test, "SVM")
```

```
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Unzipping corpora/stopwords.zip.
Class distribution before SMOTE: Counter({1: 7043, 0: 1894})
Class distribution after SMOTE: Counter({1: 7043, 0: 7043})
SMOTE has been applied successfully.
Accuracy of Naïve Bayes (with SMOTE applied): 0.4811
Accuracy of Decision Tree (with SMOTE applied): 0.8820
Accuracy of KNN (with SMOTE applied): 0.8389
Accuracy of Logistic Regression (with SMOTE applied): 0.8804
Accuracy of Random Forest (with SMOTE applied): 0.8901
Accuracy of SVM (with SMOTE applied): 0.8846
```

```
# Text Preprocessing
nltk.download('stopwords')
ps = PorterStemmer()
all_stopwords = stopwords.words('english')
all_stopwords.remove('not')  # Retaining 'not' for sentiment analysis

corpus = []
for i in range(len(dataset)):
    review = re.sub(r'[^a-zA-Z]', ' ', str(dataset['user'][i]))
    review = review.lower()
    review = review.split()
    review = [ps.stem(word) for word in review if word not in set(all_stopwords)]
    corpus.append(' '.join(review))

# Convert text into numerical features using TfidfVectorizer
vectorizer = TfidfVectorizer(max_features=2000)
X_train, X_test, y_train, y_test = train_test_split(corpus, y, test_size=0.30, random_state=0)

X_train_vect = vectorizer.fit_transform(X_train).toarray()
X_test_vect = vectorizer.transform(X_test).toarray()

# Display class distribution before SMOTE
print("Class distribution before SMOTE:", Counter(y_train.tolist()))

# Apply SMOTE to balance the training data
smote = SMOTE(random_state=0)
X_train_vect, y_train = smote.fit_resample(X_train_vect, y_train)

# Display class distribution after SMOTE
print("Class distribution after SMOTE:", Counter(y_train.tolist()))
print("SMOTE has been applied successfully.")

# Training different models
classifier_np = GaussianNB()
classifier_np.fit(X_train_vect, y_train)

classifier_dt = DecisionTreeClassifier(criterion='entropy', random_state=0)
classifier_dt.fit(X_train_vect, y_train)

classifier_knn = KNeighborsClassifier(n_neighbors=5, metric='minkowski', p=2)
classifier_knn.fit(X_train_vect, y_train)
```

2]Count vectorizer

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from wordcloud import WordCloud

# NLP tools
import re
import nltk
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer
from sklearn.feature_extraction.text import CountVectorizer

# Model selection and evaluation
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Machine learning models
from sklearn.naive_bayes import GaussianNB
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn import svm

# SMOTE for handling class imbalance
from imblearn.over_sampling import SMOTE
from collections import Counter

# Load dataset
dataset = pd.read_csv('/content/drive/MyDrive/ConvAbuseEMNLPfull.csv')

# Extracting necessary columns
dt_transformed = dataset[['is_abuse.1', 'user']]
y = dt_transformed.iloc[:, :-1].values.flatten()  # Extract labels and flatten array

# OneHotEncoding the label
y = np.where(y == 1, 1, 0)  # Ensuring binary classification

# Text Preprocessing
```

```
# Text Preprocessing
nltk.download('stopwords')
ps = PorterStemmer()
all_stopwords = stopwords.words('english')
all_stopwords.remove('not')  # Retaining 'not' for sentiment analysis

corpus = []
for i in range(len(dataset)):
    review = re.sub(r'[^a-zA-Z]', ' ', str(dataset['user'][i]))
    review = review.lower()
    review = review.split()
    review = [ps.stem(word) for word in review if word not in set(all_stopwords)]
    corpus.append(' '.join(review))

# Convert text into numerical features
vectorizer = CountVectorizer(max_features=2000)
X_train, X_test, y_train, y_test = train_test_split(corpus, y, test_size=0.30, random_state=0)

X_train_vect = vectorizer.fit_transform(X_train).toarray()
X_test_vect = vectorizer.transform(X_test).toarray()

# Display class distribution before SMOTE
print("Class distribution before SMOTE:", Counter(y_train.tolist()))

# Apply SMOTE to balance the training data
smote = SMOTE(random_state=0)
X_train_vect, y_train = smote.fit_resample(X_train_vect, y_train)

# Display class distribution after SMOTE
print("Class distribution after SMOTE:", Counter(y_train.tolist()))
print("SMOTE has been applied successfully.")

# Training different models
classifier_np = GaussianNB()
classifier_np.fit(X_train_vect, y_train)

classifier_dt = DecisionTreeClassifier(criterion='entropy', random_state=0)
classifier_dt.fit(X_train_vect, y_train)

classifier_knn = KNeighborsClassifier(n_neighbors=5, metric='minkowski', p=2)
classifier_knn.fit(X_train_vect, y_train)
```

CHAPTER 6

RESULTS AND DISCUSSIONS

The evaluation framework employs multiple metrics to comprehensively assess model performance. Accuracy alone proves misleading for imbalanced data - a naive classifier predicting "not hate speech" for all inputs would achieve 95% accuracy on our dataset while failing completely at its core task. Instead, we emphasize the F1-score, which balances precision (what proportion of predicted hate speech is actual hate speech) and recall (what proportion of actual hate speech is correctly identified). Our model achieves a weighted F1-score of 0.89, where the weighting accounts for class prevalence.

Detailed examination reveals insightful patterns. For hate speech detection (Class 0), precision reaches 0.82 but recall only 0.75, indicating that while most identified hate speech is correctly labeled, about 25% of hateful content escapes detection. The confusion matrix shows these false negatives often involve subtle or novel phrasing not well-represented in training data. In contrast, offensive language detection (Class 1) achieves 0.93 recall, reflecting the model's strength in identifying explicit terms. The neutral class (Class 2) shows near-perfect performance (0.97 recall), as its lack of marked vocabulary makes it easier to distinguish.

The ROC-AUC score of 0.92 demonstrates excellent class separation capability. This metric evaluates how well the model ranks random positive instances higher than negative ones across all possible classification thresholds. A score of 0.5 indicates random guessing, while 1.0 represents perfect discrimination. Our result confirms that the learned feature representations effectively capture hate speech characteristics.

The study also surfaces important limitations. Cultural and temporal dependencies affect hate speech lexicons - terms gain or lose offensive connotations over time and across communities. Our static model requires periodic retraining to maintain relevance. Additionally, the focus on lexical patterns misses visual hate speech (memes, coded imagery) and requires separate computer vision approaches.

6.1 Analysis

Feature importance analysis provides model interpretability. The top predictive terms align with linguistic theories of hate speech - racial epithets, gendered slurs, and dehumanizing metaphors appear most significant. Interestingly, some seemingly neutral terms like "them" or "those" gain importance through contextual patterns (e.g., "those people"). This underscores the value of bigrams in capturing implicit bias that single words might miss.

Comparative analysis with baseline models highlights XGBoost's advantages. Logistic regression achieves only 0.81 F1-score due to its linear decision boundaries struggling with phrase-level interactions. Random forest (0.84 F1) shows better performance but lacks XGBoost's gradient optimization that systematically reduces residual errors. Both alternatives prove more sensitive to class imbalance despite weighting adjustments.

6.1.1 Hate Speech Data

Model	TF-IDF 70-30	TF-IDF 80-20	CountVec 70-30	CountVec 80-20
Random Forest	0.943	0.9433	0.9411	0.9395
K Nearest Neighbours	0.9434	0.9439	0.942	0.9427
Support Vector Machine	0.9454	0.9463	0.9447	0.9439
XGBoost Classifier	0.9458	0.9482	0.9445	0.9473
Logistic Regression	0.945	0.9463	0.9439	0.9435
Decision Tree	0.926	0.9268	0.9262	0.9246
Naive Bayes	0.7541	0.7269	0.7543	0.7269

Table 2: Hate Speech Data With Smote

Model	CountVec 70-30	CountVec 80-20	TF-IDF 70-30	TF-IDF 80-20
Random Forest	0.9376	0.9379	0.9428	0.9421
K Nearest Neighbours	0.9388	0.9491	0.9422	0.9441
Support Vector Machine	0.9436	0.9441	0.9443	0.9445
XGBoost Classifier	0.9447	0.9465	0.9446	0.9475
Logistic Regression	0.94	0.9413	0.9442	0.9453
Decision Tree	0.9176	0.9163	0.926	0.9205
Naive Bayes	0.4757	0.4335	0.476	0.4343

Table 3: Hate Speech Data Without Smote

A comparison of the model accuracies in the two setups—with and without SMOTE—shows several recurring patterns. The TF-IDF vectorizer's better performance than CountVectorizer is among the most obvious trends. In almost all models, TF-IDF produces greater accuracy scores by giving weight to less common but maybe more relevant phrases. For example, the accuracy using TF-IDF (80-20 split) for the XGBoost classifier without SMOTE is 0.9475, whereas the accuracy with CountVectorizer is 0.9465. Logistic Regression and SVM models show similar differences, indicating that TF-IDF captures more significant textual patterns than just frequency counts.

Several recurrent trends may be seen when comparing the model accuracies in the two setups—with and without SMOTE. One of the most evident patterns is that the TF-IDF vectorizer performs better than CountVectorizer. By assigning weight to less frequently used but perhaps more pertinent terms, TF-IDF generates higher accuracy ratings in practically all models. For instance, the XGBoost classifier without SMOTE has an accuracy of 0.9475 when using TF-IDF (80-20 split), while the accuracy with CountVectorizer is 0.9465. Similar discrepancies between the SVM and logistic regression models suggest that TF-IDF captures more important textual patterns than merely frequency counts.

In conclusion,

1. Both with and without SMOTE, XGBoost consistently performs better than any other model.
2. SMOTE offers a slight performance improvement for several models, especially SVM and Logistic Regression.
3. On models, TF-IDF typically performs better than CountVectorizer.
4. In most situations, an 80-20 data split is marginally superior to a 70-30 one.

6.1.2 ConvAbuseEMNLPfull Dataset

Model	CountVec 70-30	CountVec 80-20	TF-IDF 70-30	TF-IDF 80-20
Random Forest	0.8964	0.8943	0.8961	0.8947
K Nearest Neighbours	0.8917	0.8845	0.8804	0.8821
Support Vector Machine	0.9050	0.9017	0.9042	0.9021
XGBoost Classifier	0.8912	0.8900	0.8821	0.8891
Logistic Regression	0.8930	0.8896	0.8883	0.8837
Decision Tree	0.8898	0.8876	0.8888	0.8880
Naive Bayes	0.4644	0.4663	0.4704	0.4710

Table 4: ConvAbuseEMNLPfull Dataset With Smote

Model	CountVec 70-30	CountVec 80-20	TF-IDF 70-30	TF-IDF 80-20
Random Forest	0.7773	0.7807	0.8901	0.8935
K Nearest Neighbours	0.7862	0.7353	0.8389	0.8301
Support Vector Machine	0.7716	0.7823	0.8846	0.8880
XGBoost Classifier	0.8721	0.8832	0.8710	0.8253
Logistic Regression	0.8872	0.8865	0.8804	0.8790
Decision Tree	0.7679	0.7772	0.8820	0.8872
Naive Bayes	0.4667	0.4726	0.4811	0.4792

Table 5: ConvAbuseEMNLPfull Dataset Without Smote

Overall, TF-IDF generally outperformed CountVectorizer, especially following use of SMOTE. The fact that TF-IDF can quantify the relevance of words with respect to the entire corpus is likely the reason for this.A majority of models trained with TF-IDF functioned better when SMOTE was applied, especially in the case of unbalanced classes. Yet, upon applying SMOTE, some models (like Random Forest and SVM) for CountVectorizer saw a drop in performance, indicating that the synthetic samples had introduced overfitting.

In nearly every setting, Support Vector Machine steadily performed better than the others, especially when TF-IDF without applying SMOTE was employed.Due to its stringent independence requirements, Naive Bayes did the poorest, which would suggest that it would not be suitable for this dataset or use.

The project demonstrates the effectiveness of classical machine learning techniques for hate speech detection using natural language processing. Key highlights include:

1. A comprehensive preprocessing pipeline.
2. Comparative evaluation of CountVectorizer vs. TfidfVectorizer.
3. SMOTE effectively addressed class imbalance.
4. XGBoost provided an advanced, performance-oriented benchmark.

The methodology and results provide a solid foundation for future applications in text-based abuse detection systems.

CHAPTER 7
CONCLUSION AND FUTURE WORK

7.1 Conclusion

The project's hate speech identification system successfully demonstrates how well conventional machine learning models handle and classify conversational text data. The textual data was transformed into a structured format suitable for training through rigorous preprocessing, which included negation handling, stopword removal, and stemming. Two more vectorization techniques, CountVectorizer and TfidfVectorizer, were used. In order to lessen class imbalance and significantly improve classifier performance on minority class cases, the SMOTE approach was also applied.

The vectorizer used affected the performance of the seven models that were used: Naïve Bayes, Decision Tree, KNN, Logistic Regression, Random Forest, SVM, and XGBoost. XGBoost was competitive and had significant promise for practical use. This study focuses on the idea that good performance in binary hate speech categorization can be attained even with relatively simple models and preprocessing. The model's success is largely determined on the vectorizer and balancing approach chosen.

7.2 Future Scope

There is still much space for improvement, but the current approach offers a solid foundation for detecting hate speech using traditional machine learning methods. The model can be extended in the future by adding transformer-based models or deep learning techniques like BERT, which are superior to conventional vectorizers at capturing contextual semantics. The methodology can be applied in multilingual global contexts by incorporating multilingual capabilities to detect hate speech in multiple languages. To enable automatic content moderation on social media and online forums, the system can potentially be expanded into a real-time monitoring system. Since explainability techniques like LIME or SHAP provide insights into model decisions, their inclusion will help build trust.Furthermore, the reliability of the model will be enhanced by making sure it is resistant to hostile inputs, such as code-switching or modified spellings. Last but not least, a feedback loop for ongoing learning and scalable microservice deployment will maximize flexibility and efficacy in real-world scenarios.

REFERENCES

[1] A. Schmidt and M. Wiegand, "A survey on hate speech detection using natural language processing," in *Proc. 5th Int. Conf. Recent Adv. Natural Lang. Process. (RANLP)*, 2017, pp. 1–10.

[2] S. Malmasi and M. Zampieri, "Challenges in discriminating profanity from hate speech," *J. Exp. Theor. Artif. Intell.*, vol. 30, no. 2, pp. 187–202, 2018.

[3] T. Davidson et al, "Contextual analysis of hate speech in multi-modal social media data," *IEEE Trans. Comput. Social Syst.*, vol. 9, no. 4, pp. 1234–1245, 2022.

[4] Z. Waseem and D. Hovy, "Hateful symbols or hateful people? Predictive features for hate speech detection on Twitter," in *Proc. NAACL Student Res. Workshop*, 2016, pp. 88–93.

[5] P. Burnap and M. L. Williams, "Us and them: Identifying cyber hate on Twitter across multiple protected characteristics," *EPJ Data Sci.*, vol. 5, no. 1, pp. 1–15, 2016.

[6] C. Nobata et al., "Abusive language detection in online user content," in *Proc. 25th Int. Conf. World Wide Web (WWW)*, 2016, pp. 145–153.

[7] B. Gambäck and U. K. Sikdar, "Using convolutional neural networks to classify hate-speech," in *Proc. 1st Workshop Abusive Lang. Online*, 2017, pp. 85–90.

[8] G. K. Pitsilis, H. Ramanpiaro, and H. Langseth, "Effective hate-speech detection in Twitter data using recurrent neural networks," *Appl. Intell.*, vol. 48, no. 12, pp. 4730–4742, 2018.

[9] T. Caselli et al., "HateBERT: Retraining BERT for abusive language detection in English," *arXiv:2010.12472*, 2021.

[10] D. Q. Nguyen, T. Vu, and A. T. Nguyen, "BERTweet: A pre-trained language model for English tweets," *arXiv:2005.10200*, 2020.

[11] D. Kiela et al., "The hateful memes challenge: Detecting hate speech in multimodal memes," *Adv. Neural Inf. Process. Syst.*, vol. 33, pp. 2611–2624, 2020.

[12] S. Zimmerman, C. Fox, and U. Kruschwitz, "Improving hate speech detection with deep learning ensembles," in *Proc. 11th Int. Conf. Lang. Resour. Eval. (LREC)*, 2020, pp. 2546–2553.

[13] D. Paschalides et al., "MANDOLA: A big-data processing and visualization platform for monitoring and detecting online hate speech," *IEEE Trans. Knowl. Data Eng.*, vol. 34, no. 6, pp. 1–21, 2020.

[14] J. Qian, M. ElSherief, E. Belding, and W. Y. Wang, "Hierarchical CVAE for fine-grained hate speech classification," in *Proc. Conf. Empirical Methods Natural Lang. Process. (EMNLP)*, 2018, pp. 3550–3559.

[15] M. ElSherief et al., “Peer to peer hate: Hate speech instigators and their targets,” in *Proc. 12th Int. AAAI Conf. Web Social Media (ICWSM),* 2018, pp. 52–61.

[16] B. Mathew et al., “HateXplain: A benchmark dataset for explainable hate speech detection,” *Proc. AAAI Conf. Artif. Intell.,* vol. 35, no. 17, pp. 14867–14875, 2021.

[17] P. Fortuna, J. Soler-Company, and L. Wanner, “Toxic, hateful, offensive or abusive? What are we really classifying? An empirical analysis of hate speech datasets,” in *Proc. 12th Lang. Resour. Eval. Conf.,* 2020, pp. 6786–6794.

[18] S. Agarwal and C. R. Chowdary, “Combating hate speech using an adaptive ensemble learning model with a case study on COVID-19,” *Expert Syst. Appl.,* vol. 185, p. 115632, 2021.

[19] M. S. Jahan and M. Oussalah, “A systematic review of hate speech automatic detection using natural language processing,” *arXiv:2106.00742,* 2021.

[20] V. Basile et al., “SemEval-2019 task 5: Multilingual detection of hate speech against immigrants and women in Twitter,” in *Proc. 13th Int. Workshop Semantic Eval.,* 2019, pp. 54–63.

[21] M. Zampieri et al., “SemEval-2019 task 6: Identifying and categorizing offensive language in social media (OffensEval),” in *Proc. 13th Int. Workshop Semantic Eval.,* 2019, pp. 75–86.

[22] A.-M. Founta et al., “Large scale crowdsourcing and characterization of Twitter abusive behavior,” *arXiv:1802.00393,* 2018.

[23] Schmidt, A., & Wiegand, M. (2017). *A Survey on Hate Speech Detection Using Natural Language Processing.* Proceedings of the Fifth International Workshop on Natural Language Processing for Social Media.

[24] Nielsen, D. (2016). *Tree Boosting With XGBoost – Why Does XGBoost Win “Every” Machine Learning Competition?* Master’s Thesis, Norwegian University of Science and Technology.

[25] T. Davidson, D. Warmsley, M. Macy, and I. Weber, “Automated hate speech detection and the problem of offensive language,” in *Proc. Int. AAAI Conf. Web Social Media (ICWSM),* 2017, pp. 512–515.

[26] A. C. Curry, G. Abercrombie, and V. Rieser, “ConvAbuse: Data, Analysis, and Benchmarks for Nuanced Detection in Conversational AI,” in *Proc. 2021 Conf. Empirical Methods Natural Lang. Process. (EMNLP),* 2021, pp. 7388–7403.

SYMBIOSIS INSTITUTE OF TECHNOLOGY, NAGPUR

Project Based Learning (PBL) Meeting Log

Name of Student: Mukund Kuthe

PRN: 2307052108 2 Sec: B Sem: 4

Mob. No.: +91- 7741963853 Email ID: mukund.kuthe.btech2023 @sitnagpur.siu.edu.in

PBL Guide Name: Prof Dr. Deepak Asudani

Co-Guide (If Any):

Project Title: Use of pre-trained models for text analytics tasks

Please use this sheet to record the task completion and attendance (minimum 4 hours per week) during your PBL Slot. Signature by PBL Guide/Co-Guide is compulsory on each visit.

Date	Task	Faculty Signature
2/1/25	Final Research Papers on ML & DL	Asudani
6/1/25	Analysis of Algorithm on ML & DL	Asudani
9/1/25	Hate speech detection	Asudani
20/1/25	Learn Tf-idf, word to vec Methods	Asudani
23/1/25	Decision Tree, Glove and bert.	Asudani 23/1/25
27/1/25	Accuracy, Precision, Recall;	Asudani 27/01/25
30/1/25	Cbow and skipgram (Word to vec)	Asudani 30/01/25
3/2/25	xG Boost and its terminologies	Asudani
6/2/25	Pretrained models for text analysis.	Asudani
17/2/25	Study of the Research Paper.	Asudani
24/2/25	Study of Hate Speech with dataset.	Asudani 24/2/25
27/2/25	Review 1 (Presentation)	13/3/25
13/3/25	Implementation of Hate Speech	
17/3/25	Report and Review Discussion.	27/3/25
27/3/25	Implementation Hate Speech (xG Boost)	
3/4/25	Final implementation on ML.	Asudani 3/4/25
7/4/25	Final Report Making.	Asudani