Toxic Comments Classification

Abstract

The proliferation of online platforms has facilitated global communication, but it has also led to the rise of toxic comments, which can disrupt healthy discourse and lead to harassment and abuse. The Toxic Comment Classification Challenge, organized by the Conversation AI team at Jigsaw and Google, aims to improve the detection of various types of toxicity in online comments. This project involves building a multi-headed model to classify comments into six categories: toxic, severe toxic, obscene, threat, insult, and identity hate.

Using a dataset from Wikipedia's talk page edits, we explore different preprocessing techniques and machine learning models to enhance the accuracy of toxicity detection. The best-performing model, evaluated using the mean column-wise ROC AUC metric, contributes to the development of tools that foster more respectful and productive online conversations.

1. Introduction

1.1 Background

The internet has become a significant platform for public discourse, but it has also become a breeding ground for toxic behavior. Identifying and mitigating toxic comments is crucial for maintaining healthy online communities. This project aims to build a robust machine learning model that can classify comments into multiple toxicity categories to assist in moderating online platforms.

1.2 Objective

The goal of this project is to develop models capable of detecting various types of toxic comments such as:

- Toxic
- Severe toxic
- Obscene
- Threat
- Insult
- Identity hate

Using a dataset from Wikipedia's talk page edits, we explore different preprocessing techniques and machine learning models to enhance the accuracy of toxicity detection.

2. Data Collection and Preprocessing

2.1 Data Loading

The dataset was sourced from the Toxic Comment Classification Challenge hosted on Kaggle. It consists of comments from Wikipedia's talk page edits labeled with six types of toxicity. The dataset was loaded as follows:

```
import pandas as pd
import numpy as np

# Load the training and test data
train = pd.read_csv('train.csv')
test = pd.read_csv('test.csv')

# Display the first few rows of the train dataset
train.head()
```

```
                 id                                       comment_text
toxic  \
0  0000997932d777bf  Explanation\nWhy the edits made under my usern...
0
1  000103f0d9cfb60f  D'aww! He matches this background colour I'm s...
0
2  000113f07ec002fd  Hey man, I'm really not trying to edit war. It...
0
3  0001b41b1c6bb37e  "\nMore\nI can't make any real suggestions on ...
0
4  0001d958c54c6e35  You, sir, are my hero. Any chance you remember...
0

   severe_toxic  obscene  threat  insult  identity_hate
0             0        0       0       0              0
1             0        0       0       0              0
2             0        0       0       0              0
3             0        0       0       0              0
4             0        0       0       0              0
```

```
test_labels = pd.read_csv('test_labels.csv')

# Display the first few rows of the test dataset
test.head()
```

```
                 id                                       comment_text
0  00001cee341fdb12  Yo bitch Ja Rule is more succesful then you'll...
1  0000247867823ef7  == From RfC == \n\n The title is fine as it is...
2  00013b17ad220c46  " \n\n == Sources == \n\n * Zawe Ashton on Lap...
3  00017563c3f7919a  :If you have a look back at the source, the in...
4  00017695ad8997eb          I don't anonymously edit articles at all.
```

2.2 Data Preprocessing

To clean and preprocess the text data, we used the following steps:

- **HTML Tag Removal:** Remove HTML tags from comments.
- **Lowercasing:** Convert text to lowercase.
- **Special Character Removal:** Remove special characters and digits.
- **Tokenization and Lemmatization:** Tokenize the text into words and apply lemmatization to standardize word forms.
- **Stopword Removal:** Remove common English stopwords

```
import re
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
import string
from nltk.stem import WordNetLemmatizer
import nltk

# Global variables
stoplist = set(stopwords.words('english')) | set(string.punctuation)
lemmatizer = WordNetLemmatizer()

# Optimized preprocessing function
def preprocess_text(text):
    text = re.sub(r'<.*?>', '', text)
    text = text.lower()
    text = re.sub(r'[^a-z0-9\s]', '', text)
    tokens = word_tokenize(text)
    tokens = [lemmatizer.lemmatize(word) for word in tokens if word
not in stoplist and not word.isdigit()]
    return ' '.join(tokens)

# Apply preprocessing
train['comment_text'] =
train['comment_text'].astype(str).apply(preprocess_text)
test['comment_text'] =
test['comment_text'].astype(str).apply(preprocess_text)

----------------------------------------------------------------------
-----
NameError                                 Traceback (most recent call
last)
Cell In[2], line 22
     19     return ' '.join(tokens)
     21 # Apply preprocessing
---> 22 train['comment_text'] =
train['comment_text'].astype(str).apply(preprocess_text)
     23 test['comment_text'] =
test['comment_text'].astype(str).apply(preprocess_text)
```

```
NameError: name 'train' is not defined
```

3. Exploratory Data Analysis (EDA)

3.1 Word Clouds

Word clouds provide a visual representation of the most frequent words in each class. To make the word clouds more distinct, we excluded common words that appear across multiple classes.

```
from collections import Counter

from wordcloud import WordCloud
import matplotlib.pyplot as plt

# Function to generate and display word clouds
def generate_wordcloud(text, title):
    wordcloud = WordCloud(width=800, height=400,
background_color='white').generate(' '.join(text))
    plt.figure(figsize=(10, 5))
    plt.imshow(wordcloud, interpolation='bilinear')
    plt.title(title, fontsize=20)
    plt.axis('off')
    plt.show()

# Generate word clouds for each class
classes = ['toxic', 'severe_toxic', 'obscene', 'threat', 'insult',
'identity_hate']
for class_name in classes:
    text = train[train[class_name] == 1]['comment_text']
    generate_wordcloud(text, f"Word Cloud for {class_name}")
```

Word Cloud for toxic

article people hi moron fuck fuck moron hi nigger nigger wikipedia shit shit suck suck page one jew fat fuck go ball ball bark bark pig pig go fuck faggot faggot bullshit bullshit dickhead dickhead want fat jew hate hate think wanker wanker nipple nipple youre say suck dick die die fag die idiot stupid life time fucking edit even im see die fag dont way make

Word Cloud for severe_toxic

shit shit suck suck fuck fuck go fuck fuck go faggot faggot die die cocksucker mothjer mothjer fucker fuck yourselfgo fucksex fucksex offfuck offfuck suck cock nigger nigger fucker cocksucker suck mexican shut fuck huge faggot mexican suck suck dick fuck dog fuck shut faggot huge dog fuck

Word Cloud for obscene

Word Cloud for threat

Word Cloud for insult

Word Cloud for identity_hate

3.1.1 Identify Common Words

We identified common words that appear across multiple classes:

```
# Function to get the most common words in each class
def get_common_words(texts, num_common=50):
    all_words = ' '.join(texts).split()
    common_words = [word for word, count in
```

```
Counter(all_words).most_common(num_common)]
    return set(common_words)

# Get common words for each class
common_words = set()
for class_name in classes:
    texts = train[train[class_name] == 1]['comment_text']
    common_words.update(get_common_words(texts))

print(f"Common words: {common_words}")
```

```
Common words: {'fool', 'like', 'right', 'supertr0ll', 'wikipedia',
'pathetic', 'stupid', 'kill', 'talk', 'mothjer', 'bullshit', 'suck',
'asshole', 'fucksex', 'drink', 'licker', 'hate', 'bastard', 'u',
'centraliststupid', 'life', 'bitchfuck', 'forever', 'continue', 'utc',
'respect', 'die', 'fucking', 'fat', 'cunt', 'piece', 'youre', 'jew',
'as', 'pig', 'dickhead', 'must', 'moron', 'nipple', 'page', 'shit',
'ill', 'rvv', 'na', 'pussy', 'fucker', 'people', 'stop', 'wiki',
'huge', 'bunksteve', 'ancestryfuckoffjewish', 'yourselfgo', 'im',
'wale', 'live', 'fuck', 'article', 'take', 'one', 'time', 'dick',
'hope', 'edie', 'spanish', 'dont', 'eat', 'bitchesfuck', 'tommy2010',
'even', 'cant', 'cock', 'shut', 'dog', 'faggot', 'fag', 'ban',
'wanker', 'offfuck', 'homo', 'cocksucker', 'murder', 'know', 'block',
'lifetime', 'filter', 'bleachanhero', 'blank', 'gon',
'proassadhanibal911youre', 'hi', 'think', 'want', 'make', 'damn',
'dust', 'keep', 'criminalwar', 'nigger', 'going', 'steal', 'youfuck',
'anal', 'fuckin', 'would', 'ball', 'rape', 'go', 'penis', 'di',
'bitch', 'get', 'jim', 'fan1967', 'bush', 'password', 'gay',
'mexican', 'nigga', 'idiot'}
```

3.1.2 Generate Distinct Word Clouds

We generated word clouds for each class excluding the common words:

```
# Function to generate and display word clouds excluding common words
def generate_distinct_wordcloud(texts, title, common_words):
    filtered_texts = [' '.join([word for word in text.split() if word
not in common_words]) for text in texts]
    wordcloud = WordCloud(width=800, height=400,
background_color='white').generate(' '.join(filtered_texts))
    plt.figure(figsize=(10, 5))
    plt.imshow(wordcloud, interpolation='bilinear')
    plt.title(title, fontsize=20)
    plt.axis('off')
    plt.show()

# Generate distinct word clouds for each class
for class_name in classes:
    texts = train[train[class_name] == 1]['comment_text']
```

```
    generate_distinct_wordcloud(texts, f"Distinct Word Cloud for {class_name}", common_words)
```

Distinct Word Cloud for toxic

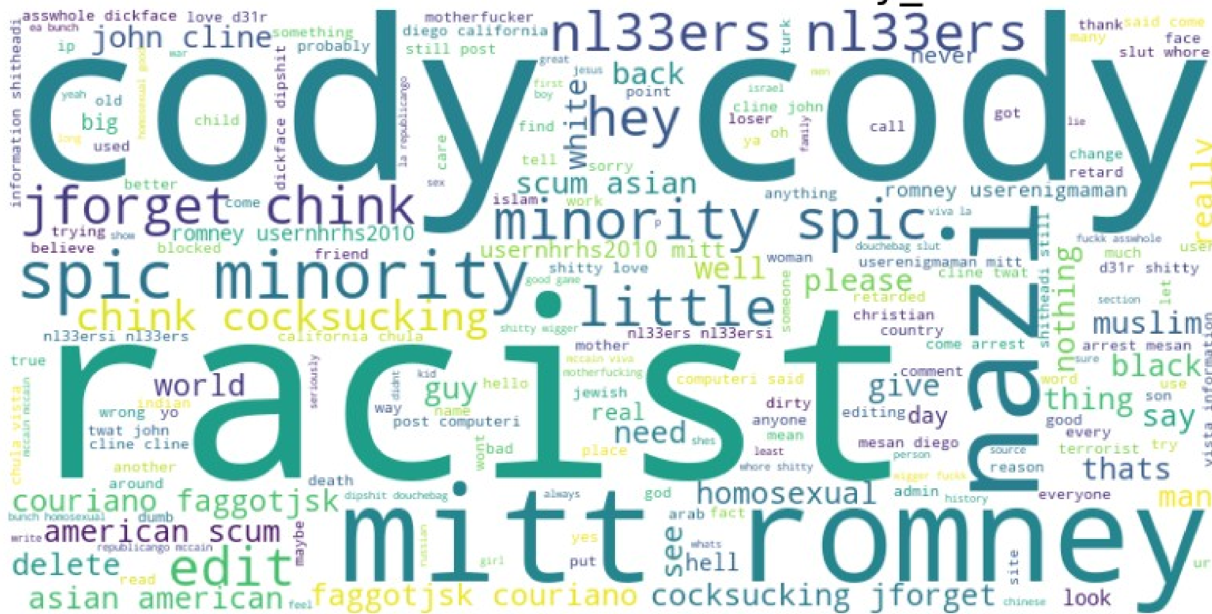
Distinct Word Cloud for severe_toxic

Distinct Word Cloud for obscene

boob boob
poop poop
please
edit
say
see
really
way
little
thats
hey
thing
buttsecks buttsecks
bollock bollock
assfuck assfuck
need
good
hell
user
well
notrhbysouthbanof notrhbysouthbanof

Distinct Word Cloud for threat

moonshine fuckunblocklifetime
youcaltlas youcaltlas
beavis butthead
niggaz moonshine
bail beavis
jessica simpson
wanta bail
death
see
edit
butthead jessica
simpson niggaz
find
fuckunblocklifetime wanta



3.2 Sentiment Analysis

We used the TextBlob library to perform elementary sentiment analysis, and visualized the sentiment distribution for each class using boxplots.

3.2.1 Compute Sentiment Polarity

1. Toxic vs. Non-Toxic Comments By comparing the sentiment distributions of toxic and non-toxic comments, we can observe distinct patterns. Toxic comments

generally have lower sentiment polarity scores, indicating a more negative tone. Non-toxic comments, on the other hand, are likely to have higher, more positive sentiment scores.

2. Detailed Class Analysis The boxplots provide a visual summary of the sentiment polarity distribution for each class:

Toxic: This class may show a wide range of negative sentiment scores, indicating various degrees of toxicity.

Severe Toxic: Comments in this class are expected to have even lower sentiment scores compared to the general toxic class.

Obscene, Threat, Insult, Identity Hate: Each of these classes may exhibit distinct negative sentiment patterns, reflecting their specific type of toxicity.

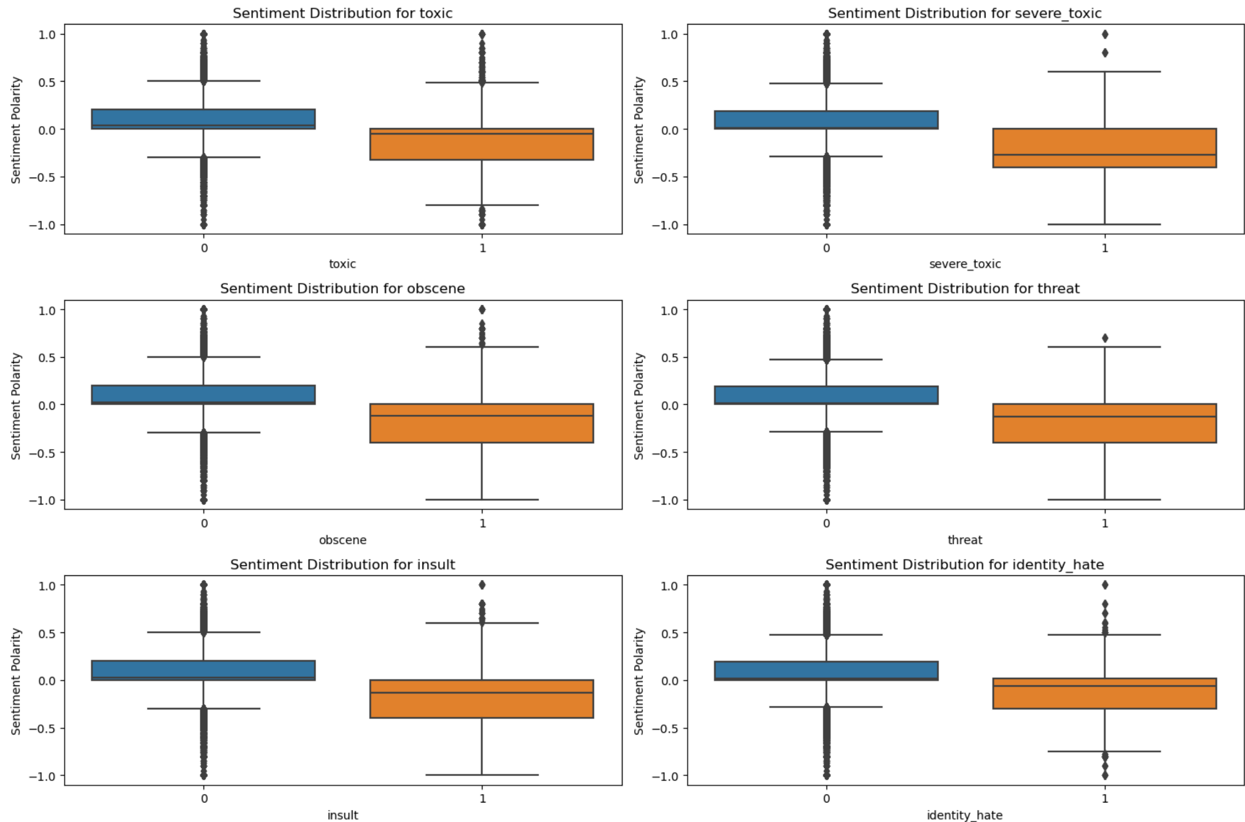
3.2.2 Sentiment Polarity: A Reference

Positive Sentiment (0 to 1): Texts with a positive sentiment score indicate favorable, happy, or approving emotions.

Negative Sentiment (-1 to 0): Texts with a negative sentiment score indicate unfavorable, unhappy, or disapproving emotions.

Neutral Sentiment (around 0): Texts with a sentiment score close to zero indicate neutral emotions, without strong positive or negative tones.

```python
from textblob import TextBlob
import seaborn as sns
import matplotlib.pyplot as plt

# Function to compute sentiment polarity
def get_sentiment(text):
    return TextBlob(text).sentiment.polarity

# Apply sentiment analysis
train['sentiment'] = train['comment_text'].apply(get_sentiment)

# Visualize sentiment distribution for each class
plt.figure(figsize=(15, 10))
for i, class_name in enumerate(classes):
    plt.subplot(3, 2, i + 1)
    sns.boxplot(x=class_name, y='sentiment', data=train)
    plt.title(f"Sentiment Distribution for {class_name}")
    plt.xlabel(class_name)
    plt.ylabel("Sentiment Polarity")

plt.tight_layout()
plt.show()
```



3.3 Detailed Metrics

We computed word count, character count, and average word length for each comment and visualized these metrics using histograms.

3.3.1 Compute Metrics

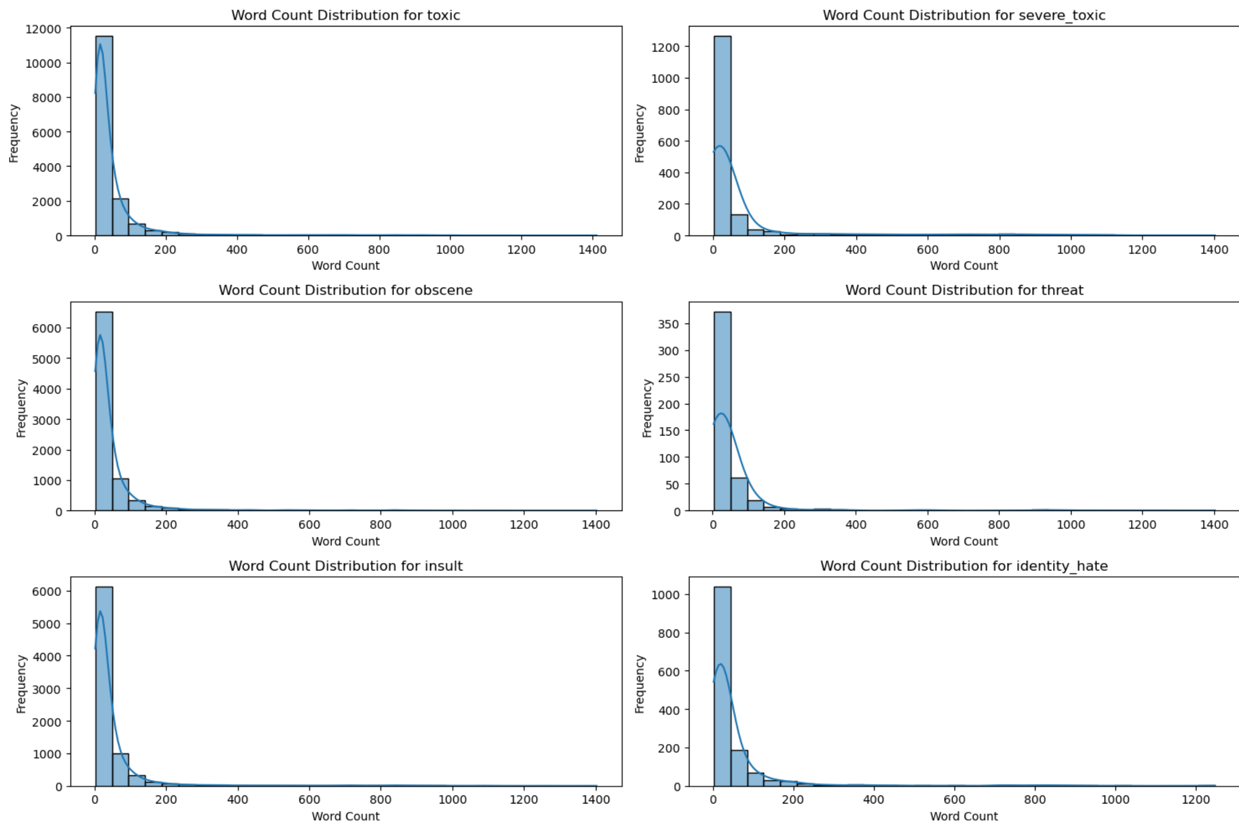
Metrics were computed for word count, character count, and average word length:

```
# Function to compute detailed metrics
def compute_metrics(df):
    df['word_count'] = df['comment_text'].apply(lambda x:
len(x.split()))
    df['char_count'] = df['comment_text'].apply(lambda x: len(x))
    df['avg_word_length'] = df['comment_text'].apply(lambda x:
np.mean([len(word) for word in x.split()]) if x.split() else 0)
    return df

# Compute metrics for train and test data
train = compute_metrics(train)
test = compute_metrics(test)

# Visualize word count distribution for each class
plt.figure(figsize=(15, 10))
for i, class_name in enumerate(classes):
```

```
    plt.subplot(3, 2, i + 1)
    sns.histplot(train[train[class_name] == 1]['word_count'], bins=30,
kde=True)
    plt.title(f"Word Count Distribution for {class_name}")
    plt.xlabel("Word Count")
    plt.ylabel("Frequency")

plt.tight_layout()
plt.show()

# Visualize character count distribution for each class
plt.figure(figsize=(15, 10))
for i, class_name in enumerate(classes):
    plt.subplot(3, 2, i + 1)
    sns.histplot(train[train[class_name] == 1]['char_count'], bins=30,
kde=True)
    plt.title(f"Character Count Distribution for {class_name}")
    plt.xlabel("Character Count")
    plt.ylabel("Frequency")

plt.tight_layout()
plt.show()

# Visualize average word length distribution for each class
plt.figure(figsize=(15, 10))
for i, class_name in enumerate(classes):
    plt.subplot(3, 2, i + 1)
    sns.histplot(train[train[class_name] == 1]['avg_word_length'],
bins=30, kde=True)
    plt.title(f"Average Word Length Distribution for {class_name}")
    plt.xlabel("Average Word Length")
    plt.ylabel("Frequency")

plt.tight_layout()
plt.show()
```

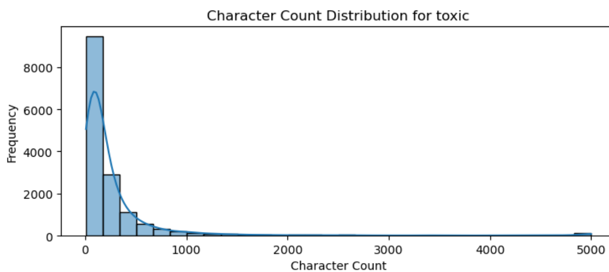
```
/opt/anaconda3/lib/python3.11/site-packages/seaborn/_oldcore.py:1119:
FutureWarning: use_inf_as_na option is deprecated and will be removed
in a future version. Convert inf values to NaN before operating
instead.
  with pd.option_context('mode.use_inf_as_na', True):
/opt/anaconda3/lib/python3.11/site-packages/seaborn/_oldcore.py:1119:
FutureWarning: use_inf_as_na option is deprecated and will be removed
in a future version. Convert inf values to NaN before operating
instead.
  with pd.option_context('mode.use_inf_as_na', True):
/opt/anaconda3/lib/python3.11/site-packages/seaborn/_oldcore.py:1119:
FutureWarning: use_inf_as_na option is deprecated and will be removed
in a future version. Convert inf values to NaN before operating
instead.
```

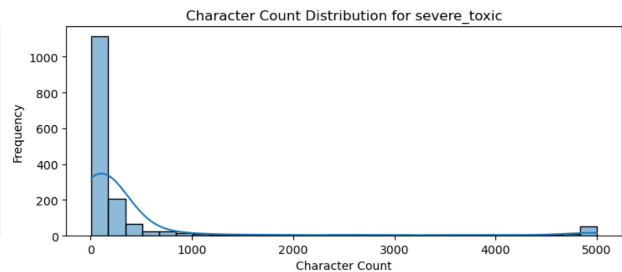
```
  with pd.option_context('mode.use_inf_as_na', True):
/opt/anaconda3/lib/python3.11/site-packages/seaborn/_oldcore.py:1119:
FutureWarning: use_inf_as_na option is deprecated and will be removed
in a future version. Convert inf values to NaN before operating
instead.
  with pd.option_context('mode.use_inf_as_na', True):
/opt/anaconda3/lib/python3.11/site-packages/seaborn/_oldcore.py:1119:
FutureWarning: use_inf_as_na option is deprecated and will be removed
in a future version. Convert inf values to NaN before operating
instead.
  with pd.option_context('mode.use_inf_as_na', True):
/opt/anaconda3/lib/python3.11/site-packages/seaborn/_oldcore.py:1119:
FutureWarning: use_inf_as_na option is deprecated and will be removed
in a future version. Convert inf values to NaN before operating
instead.
  with pd.option_context('mode.use_inf_as_na', True):
```

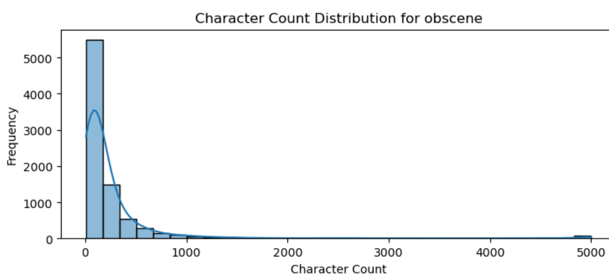


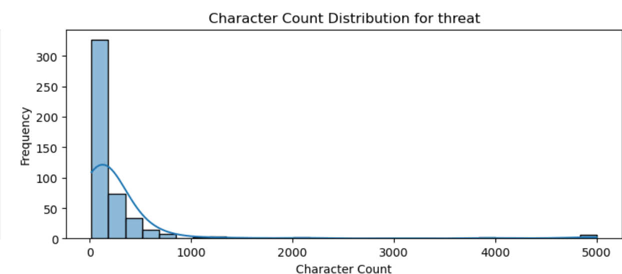

```
/opt/anaconda3/lib/python3.11/site-packages/seaborn/_oldcore.py:1119:
FutureWarning: use_inf_as_na option is deprecated and will be removed
in a future version. Convert inf values to NaN before operating
instead.
  with pd.option_context('mode.use_inf_as_na', True):
/opt/anaconda3/lib/python3.11/site-packages/seaborn/_oldcore.py:1119:
FutureWarning: use_inf_as_na option is deprecated and will be removed
```

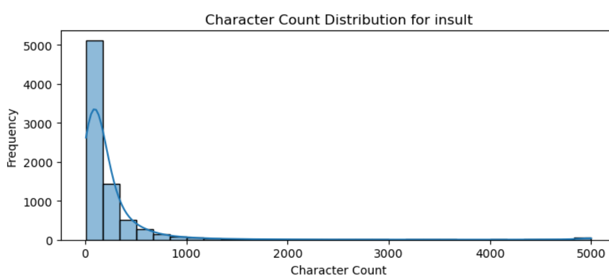
```
in a future version. Convert inf values to NaN before operating
instead.
  with pd.option_context('mode.use_inf_as_na', True):
/opt/anaconda3/lib/python3.11/site-packages/seaborn/_oldcore.py:1119:
FutureWarning: use_inf_as_na option is deprecated and will be removed
in a future version. Convert inf values to NaN before operating
instead.
  with pd.option_context('mode.use_inf_as_na', True):
/opt/anaconda3/lib/python3.11/site-packages/seaborn/_oldcore.py:1119:
FutureWarning: use_inf_as_na option is deprecated and will be removed
in a future version. Convert inf values to NaN before operating
instead.
  with pd.option_context('mode.use_inf_as_na', True):
/opt/anaconda3/lib/python3.11/site-packages/seaborn/_oldcore.py:1119:
FutureWarning: use_inf_as_na option is deprecated and will be removed
in a future version. Convert inf values to NaN before operating
instead.
  with pd.option_context('mode.use_inf_as_na', True):
/opt/anaconda3/lib/python3.11/site-packages/seaborn/_oldcore.py:1119:
FutureWarning: use_inf_as_na option is deprecated and will be removed
in a future version. Convert inf values to NaN before operating
instead.
  with pd.option_context('mode.use_inf_as_na', True):
```

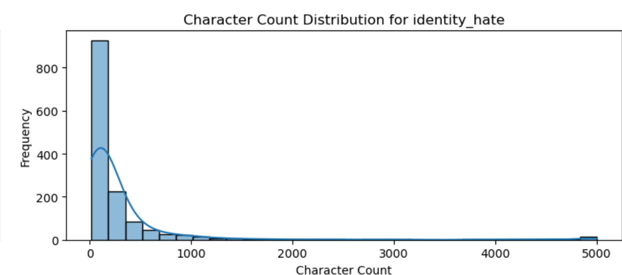




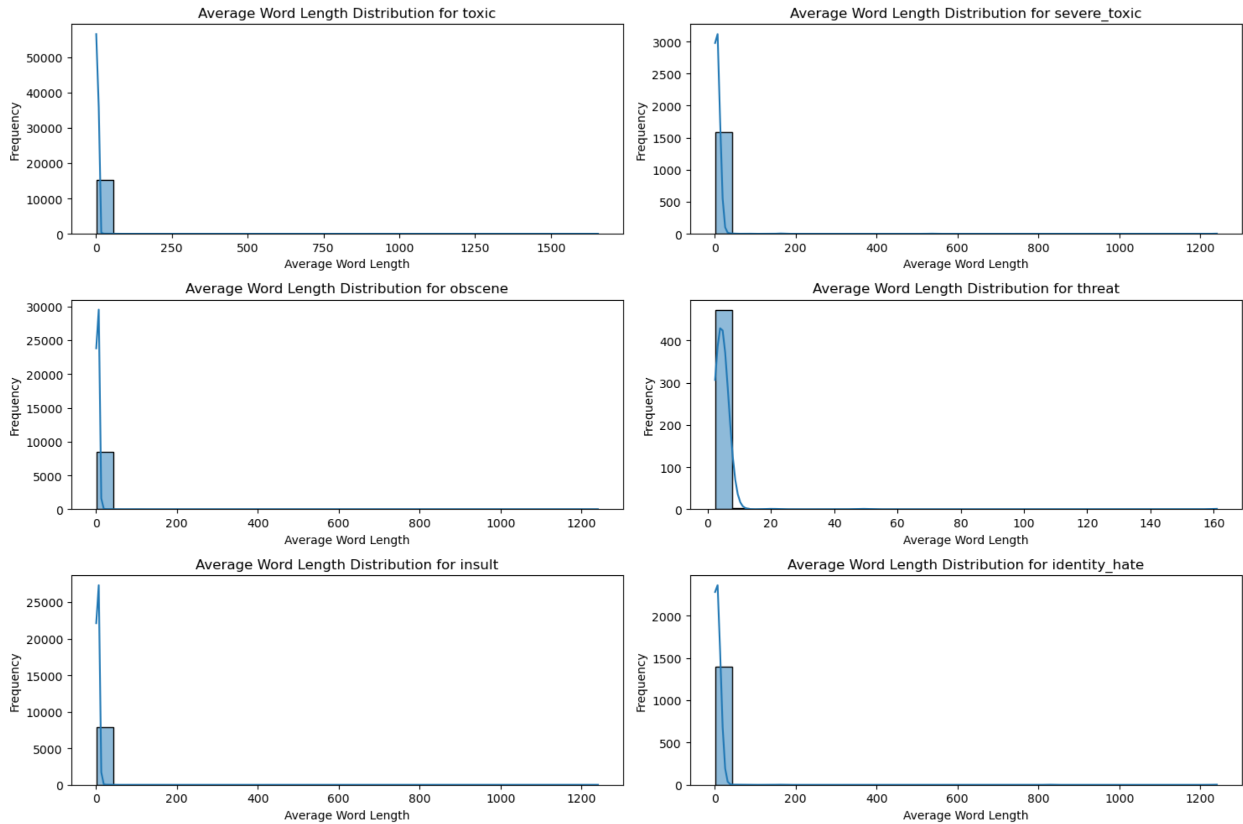








```
/opt/anaconda3/lib/python3.11/site-packages/seaborn/_oldcore.py:1119:
FutureWarning: use_inf_as_na option is deprecated and will be removed
in a future version. Convert inf values to NaN before operating
instead.
  with pd.option_context('mode.use_inf_as_na', True):
/opt/anaconda3/lib/python3.11/site-packages/seaborn/_oldcore.py:1119:
FutureWarning: use_inf_as_na option is deprecated and will be removed
in a future version. Convert inf values to NaN before operating
instead.
  with pd.option_context('mode.use_inf_as_na', True):
/opt/anaconda3/lib/python3.11/site-packages/seaborn/_oldcore.py:1119:
FutureWarning: use_inf_as_na option is deprecated and will be removed
in a future version. Convert inf values to NaN before operating
instead.
  with pd.option_context('mode.use_inf_as_na', True):
/opt/anaconda3/lib/python3.11/site-packages/seaborn/_oldcore.py:1119:
FutureWarning: use_inf_as_na option is deprecated and will be removed
in a future version. Convert inf values to NaN before operating
instead.
  with pd.option_context('mode.use_inf_as_na', True):
/opt/anaconda3/lib/python3.11/site-packages/seaborn/_oldcore.py:1119:
FutureWarning: use_inf_as_na option is deprecated and will be removed
in a future version. Convert inf values to NaN before operating
instead.
  with pd.option_context('mode.use_inf_as_na', True):
/opt/anaconda3/lib/python3.11/site-packages/seaborn/_oldcore.py:1119:
FutureWarning: use_inf_as_na option is deprecated and will be removed
in a future version. Convert inf values to NaN before operating
instead.
  with pd.option_context('mode.use_inf_as_na', True):
```



4. Feature Extraction

We used TF-IDF (Term Frequency-Inverse Document Frequency) for feature extraction.

```
from sklearn.feature_extraction.text import TfidfVectorizer

# Define the TfidfVectorizer
vectorizer = TfidfVectorizer(ngram_range=(1, 2), min_df=3, max_df=0.9,
strip_accents='unicode', use_idf=1, smooth_idf=1, sublinear_tf=1,
stop_words='english')

# Transform the data
X_train = vectorizer.fit_transform(train['comment_text'])
X_test = vectorizer.transform(test['comment_text'])

/opt/anaconda3/lib/python3.11/site-packages/sklearn/utils/
_param_validation.py:558: FutureWarning: Passing an int for a boolean
parameter is deprecated in version 1.2 and won't be supported anymore
in version 1.4.
  warnings.warn(
```

5. Model Training

5.1 Logistic Regression

Why Logistic Regression?

Logistic Regression is a simple and efficient model for binary classification, making it suitable for our task of predicting whether a comment is toxic or not. It serves as a strong baseline due to its interpretability and low computational cost.

What is Logistic Regression Doing?

- **Modeling Log-Odds**: We train a separate logistic regression model for each toxic category.
- **Probability Estimation**: It transforms log-odds into probabilities using the logistic function.
- **Binary Classification**: For each toxic category, it fits a separate binary classifier using the TF-IDF features and log-odds ratios of word occurrences.

```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import roc_auc_score
import numpy as np

# Function to calculate the probability of each word given a specific
class
def probability(y_i, y):
    occurences = X_train[y == y_i].sum(0)
    return (occurences + 1) / ((y == y_i).sum() + 1)

# Function to train the model
def get_model(y):
    y = y.values
    loga = np.log((probability(1, y)) / (probability(0, y)) )
    X_loga = X_train.multiply(loga)
    model = LogisticRegression(C=1.0, solver='liblinear',
max_iter=100, random_state=42)
    return model.fit(X_loga, y), loga

# Dictionary to store models and log-ratio values
lr_models = {}
logas = {}

# Train models for each class
classes = ['toxic', 'severe_toxic', 'obscene', 'threat', 'insult',
'identity_hate']
for class_name in classes:
    model, loga = get_model(train[class_name])
    lr_models[class_name] = model
    logas[class_name] = loga
```

5.2 XGBoost

Why XGBoost?

XGBoost is a powerful gradient boosting model known for its high performance and efficiency. It is well-suited for handling structured data and provides excellent predictive accuracy for our multi-class toxic comment classification task.

What is XGBoost Doing?

- **Ensemble Learning**: XGBoost builds an ensemble of decision trees, each correcting the errors of the previous ones.
- **Gradient Boosting**: It minimizes the loss function by sequentially adding trees that focus on the residuals.
- **Regularization**: XGBoost includes L1 and L2 regularization to prevent overfitting, enhancing the model's robustness.

```
import xgboost as xgb

# Function to train XGBoost model
def train_xgb(X_train, y_train):
    xgb_model = xgb.XGBClassifier(objective='binary:logistic',
n_estimators=200, max_depth=5, learning_rate=0.1, subsample=0.8,
colsample_bytree=0.8, random_state=42)
    xgb_model.fit(X_train, y_train)
    return xgb_model

# Train XGBoost model for each class
xgb_models = {}
for class_name in classes:
    xgb_models[class_name] = train_xgb(X_train, train[class_name])
```

5.3 Random Forest

Why Random Forest?

Random Forest is an ensemble learning method that combines multiple decision trees to improve predictive performance and control overfitting. It is robust and provides feature importance scores, making it valuable for understanding which words contribute most to toxicity.

What is Random Forest Doing?

- **Bootstrap Aggregation (Bagging)**: Random Forest trains each tree on a random subset of the data to reduce variance.
- **Random Feature Selection**: It considers a random subset of features for splitting at each node, introducing diversity among trees.
- **Ensemble Averaging**: The final prediction is the average of the probabilities predicted by all trees, enhancing stability and accuracy.

```
from sklearn.ensemble import RandomForestClassifier

# Function to train Random Forest model
def train_rf(X_train, y_train):
    rf_model = RandomForestClassifier(n_estimators=200, max_depth=10,
min_samples_split=5, random_state=42)
    rf_model.fit(X_train, y_train)
    return rf_model

# Train Random Forest model for each class
rf_models = {}
for class_name in classes:
    rf_models[class_name] = train_rf(X_train, train[class_name])
```

5.4 Model Evaluation

```
from sklearn.model_selection import train_test_split

# Split the training data for validation
X_train_split, X_valid_split, y_train_split, y_valid_split =
train_test_split(X_train, train[classes], test_size=0.2,
random_state=42)

# Function to evaluate models
def evaluate_model(model, X_valid, y_valid):
    preds = model.predict_proba(X_valid)[:, 1]
    return roc_auc_score(y_valid, preds)

# Evaluate Logistic Regression models
lr_scores = {}
for class_name in classes:
    lr_scores[class_name] = evaluate_model(lr_models[class_name],
X_valid_split.multiply(logas[class_name]), y_valid_split[class_name])
    print(f"Logistic Regression ROC AUC for {class_name}:
{lr_scores[class_name]}")

# Evaluate XGBoost models
xgb_scores = {}
for class_name in classes:
    xgb_scores[class_name] = evaluate_model(xgb_models[class_name],
X_valid_split, y_valid_split[class_name])
    print(f"XGBoost ROC AUC for {class_name}:
{xgb_scores[class_name]}")

# Evaluate Random Forest models
rf_scores = {}
for class_name in classes:
    rf_scores[class_name] = evaluate_model(rf_models[class_name],
X_valid_split, y_valid_split[class_name])
```

```
    print(f"Random Forest ROC AUC for {class_name}:
{rf_scores[class_name]}")
```

```
Logistic Regression ROC AUC for toxic: 0.9948494158908389
Logistic Regression ROC AUC for severe_toxic: 0.9992372067964322
Logistic Regression ROC AUC for obscene: 0.9981113470932366
Logistic Regression ROC AUC for threat: 0.9999465248358185
Logistic Regression ROC AUC for insult: 0.996665447588706
Logistic Regression ROC AUC for identity_hate: 0.9997341493758883
XGBoost ROC AUC for toxic: 0.955862682869173
XGBoost ROC AUC for severe_toxic: 0.9839084750702891
XGBoost ROC AUC for obscene: 0.9835427470893751
XGBoost ROC AUC for threat: 0.9912118236134442
XGBoost ROC AUC for insult: 0.9738033805142268
XGBoost ROC AUC for identity_hate: 0.9801459118165465
Random Forest ROC AUC for toxic: 0.9567758778509484
Random Forest ROC AUC for severe_toxic: 0.9870612583287534
Random Forest ROC AUC for obscene: 0.983283503562257
Random Forest ROC AUC for threat: 0.9757019039704884
Random Forest ROC AUC for insult: 0.9734849623400605
Random Forest ROC AUC for identity_hate: 0.9852094438230686
```

```
# Generate predictions for Logistic Regression
lr_preds = np.zeros((len(test), len(classes)))
for i, class_name in enumerate(classes):
    lr_preds[:, i] =
lr_models[class_name].predict_proba(X_test.multiply(logas[class_name])
)[:, 1]

# Create submission file for Logistic Regression
submission_lr = pd.DataFrame(lr_preds, columns=classes)
submission_lr['id'] = test_labels.id.astype(str)
submission_lr.to_csv('submission_lr.csv', index=False)

# Generate predictions for XGBoost
xgb_preds = np.zeros((len(test), len(classes)))
for i, class_name in enumerate(classes):
    xgb_preds[:, i] = xgb_models[class_name].predict_proba(X_test)[:,
1]

# Create submission file for XGBoost
submission_xgb = pd.DataFrame(xgb_preds, columns=classes)
submission_xgb['id'] = test_labels.id.astype(str)
submission_xgb.to_csv('submission_xgb.csv', index=False)

# Generate predictions for Random Forest
rf_preds = np.zeros((len(test), len(classes)))
for i, class_name in enumerate(classes):
    rf_preds[:, i] = rf_models[class_name].predict_proba(X_test)[:, 1]
```

```
# Create submission file for Random Forest
submission_rf = pd.DataFrame(rf_preds, columns=classes)
submission_rf['id'] = test_labels.id.astype(str)
submission_rf.to_csv('submission_rf.csv', index=False)
```

6. Deep Learning Approach

LSTM (Long Short-Term Memory)

Why LSTM?

LSTM networks are a type of recurrent neural network (RNN) that excel at capturing long-term dependencies in sequential data. They are particularly well-suited for text data, where the context of words can span many tokens.

What is LSTM Doing?

- **Memory Cells**: LSTM units have memory cells that store information over long periods, controlled by input, forget, and output gates.
- **Sequential Learning**: LSTM networks process input sequences one token at a time, maintaining a hidden state that captures context.
- **Long-Term Dependencies**: LSTMs can capture the context of words across long sequences, making them ideal for understanding the nuanced patterns in toxic comments.

6.1 Preprocessing for Deep Learning

```
import tensorflow as tf
from tensorflow.keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences

# Tokenize the text data
tokenizer = Tokenizer(num_words=20000)
tokenizer.fit_on_texts(train['comment_text'])
X_train_seq = tokenizer.texts_to_sequences(train['comment_text'])
X_test_seq = tokenizer.texts_to_sequences(test['comment_text'])

# Pad the sequences
max_length = 200
X_train_pad = pad_sequences(X_train_seq, maxlen=max_length)
X_test_pad = pad_sequences(X_test_seq, maxlen=max_length)
```

```
2024-06-22 05:06:48.232895: I
tensorflow/core/platform/cpu_feature_guard.cc:210] This TensorFlow
binary is optimized to use available CPU instructions in performance-
critical operations.
To enable the following instructions: AVX2 FMA, in other operations,
rebuild TensorFlow with the appropriate compiler flags.
```

6.2 Build and Train LSTM

```
from keras.models import Sequential
from keras.layers import Embedding, LSTM, Dense, Dropout
from keras.callbacks import EarlyStopping

# Build the LSTM model
def build_lstm_model(input_length, vocab_size):
    model = Sequential()
    model.add(Embedding(input_dim=vocab_size, output_dim=128,
input_length=input_length))
    model.add(LSTM(128, return_sequences=True))
    model.add(Dropout(0.2))
    model.add(LSTM(64))
    model.add(Dropout(0.2))
    model.add(Dense(6, activation='sigmoid'))  # 6 output units for
multi-class classification
    model.compile(loss='binary_crossentropy', optimizer='adam',
metrics=['accuracy'])
    return model

# Parameters
vocab_size = 20000
input_length = max_length

# Build the model
lstm_model = build_lstm_model(input_length, vocab_size)

# Train the model
early_stopping = EarlyStopping(monitor='val_loss', patience=3,
restore_best_weights=True)
history = lstm_model.fit(X_train_pad, train[classes].values,
validation_split=0.2, epochs=10, batch_size=64,
callbacks=[early_stopping])
```

```
/opt/anaconda3/lib/python3.11/site-packages/keras/src/layers/core/
embedding.py:90: UserWarning: Argument `input_length` is deprecated.
Just remove it.
  warnings.warn(
```

```
Epoch 1/10
1995/1995 ━━━━━━━━━━━━━━━━━━━━ 2139s 1s/step - accuracy: 0.7887 -
loss: 0.1199 - val_accuracy: 0.9941 - val_loss: 0.0527
Epoch 2/10
1995/1995 ━━━━━━━━━━━━━━━━━━━━ 2581s 1s/step - accuracy: 0.9882 -
loss: 0.0483 - val_accuracy: 0.9941 - val_loss: 0.0492
Epoch 3/10
1995/1995 ━━━━━━━━━━━━━━━━━━━━ 8536s 4s/step - accuracy: 0.9879 -
loss: 0.0420 - val_accuracy: 0.9941 - val_loss: 0.0499
Epoch 4/10
1995/1995 ━━━━━━━━━━━━━━━━━━━━ 639s 320ms/step - accuracy: 0.9876 -
```

```
loss: 0.0379 - val_accuracy: 0.9941 - val_loss: 0.0509
Epoch 5/10
1995/1995 ━━━━━━━━━━━━━━━━━━━━ 8318s 4s/step - accuracy: 0.9811 -
loss: 0.0341 - val_accuracy: 0.9941 - val_loss: 0.0533
```

6.3 Evaluate LSTM

```
# Evaluate the model
loss, accuracy = lstm_model.evaluate(X_train_pad,
train[classes].values)
print(f"LSTM Model Loss: {loss}")
print(f"LSTM Model Accuracy: {accuracy}")

# Generate predictions for the test set
preds = lstm_model.predict(X_test_pad)
submission = pd.DataFrame(preds, columns=classes)
submission['id'] = test_labels.index = test_labels.index.astype(str)
submission.to_csv('submission_lstm.csv', index=False)
```

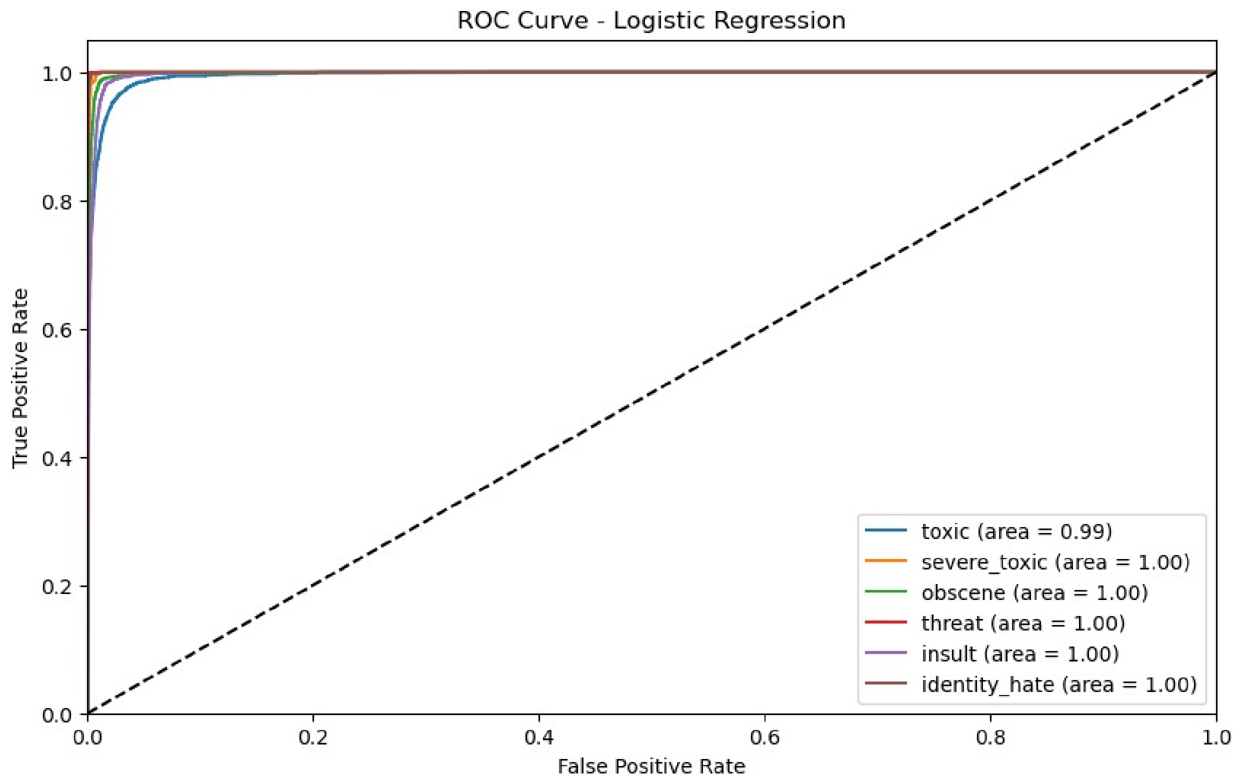
```
4987/4987 ━━━━━━━━━━━━━━━━━━━━ 309s 62ms/step - accuracy: 0.9943 -
loss: 0.0412
LSTM Model Loss: 0.04231854900717735
LSTM Model Accuracy: 0.9941655993461609
4787/4787 ━━━━━━━━━━━━━━━━━━━━ 313s 65ms/step
```

```
sdf = pd.read_csv('submission_lstm.csv',index_col=False)
# sdf.drop('Unnamed: 0', axis=1, inplace=True)
sdf['id'] = test_labels.id.astype(str)
sdf.to_csv('submission_lstm.csv', index=False)
```

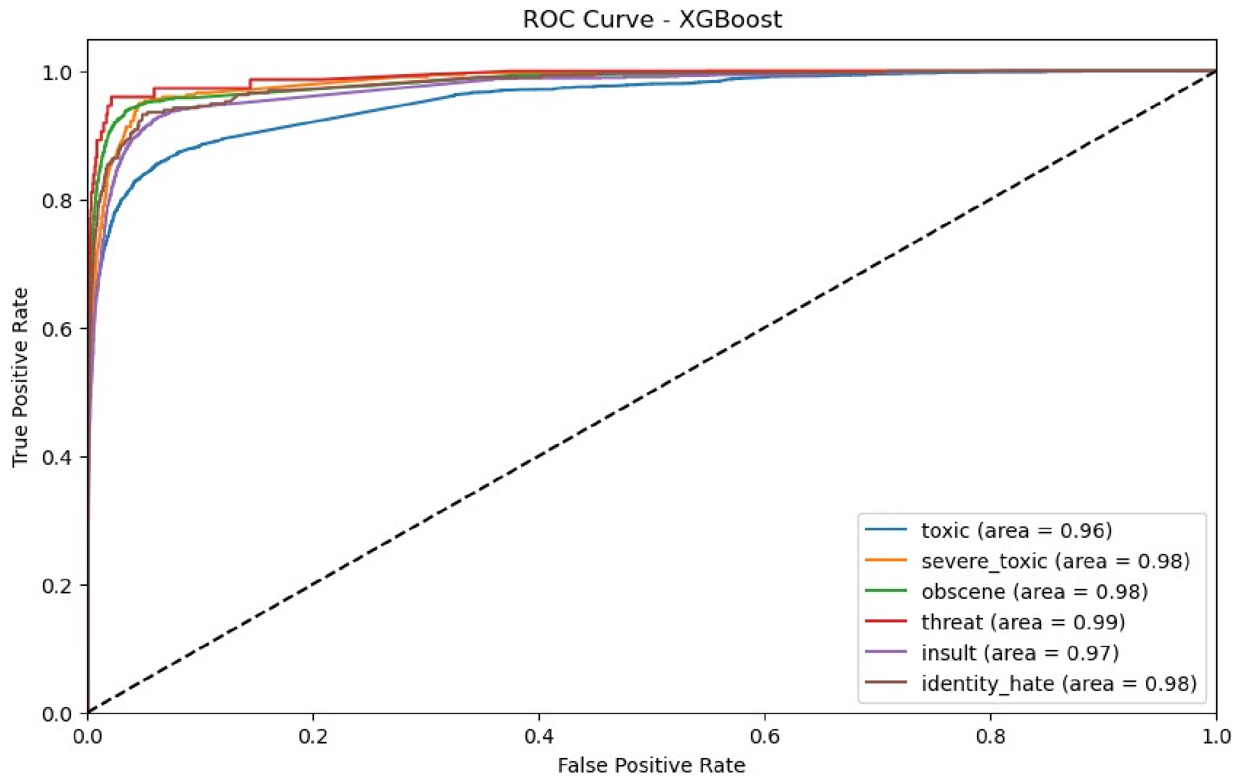
7. Model Training Visualizations

```
import matplotlib.pyplot as plt
from sklearn.metrics import roc_curve, auc

# Plot ROC Curve for Logistic Regression
plt.figure(figsize=(10, 6))

for class_name in classes:
    fpr, tpr, _ = roc_curve(y_valid_split[class_name],
lr_models[class_name].predict_proba(X_valid_split.multiply(logas[class
_name]))[:, 1])
    roc_auc = auc(fpr, tpr)
    plt.plot(fpr, tpr, label=f'{class_name} (area = {roc_auc:.2f})')

plt.plot([0, 1], [0, 1], 'k--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
```

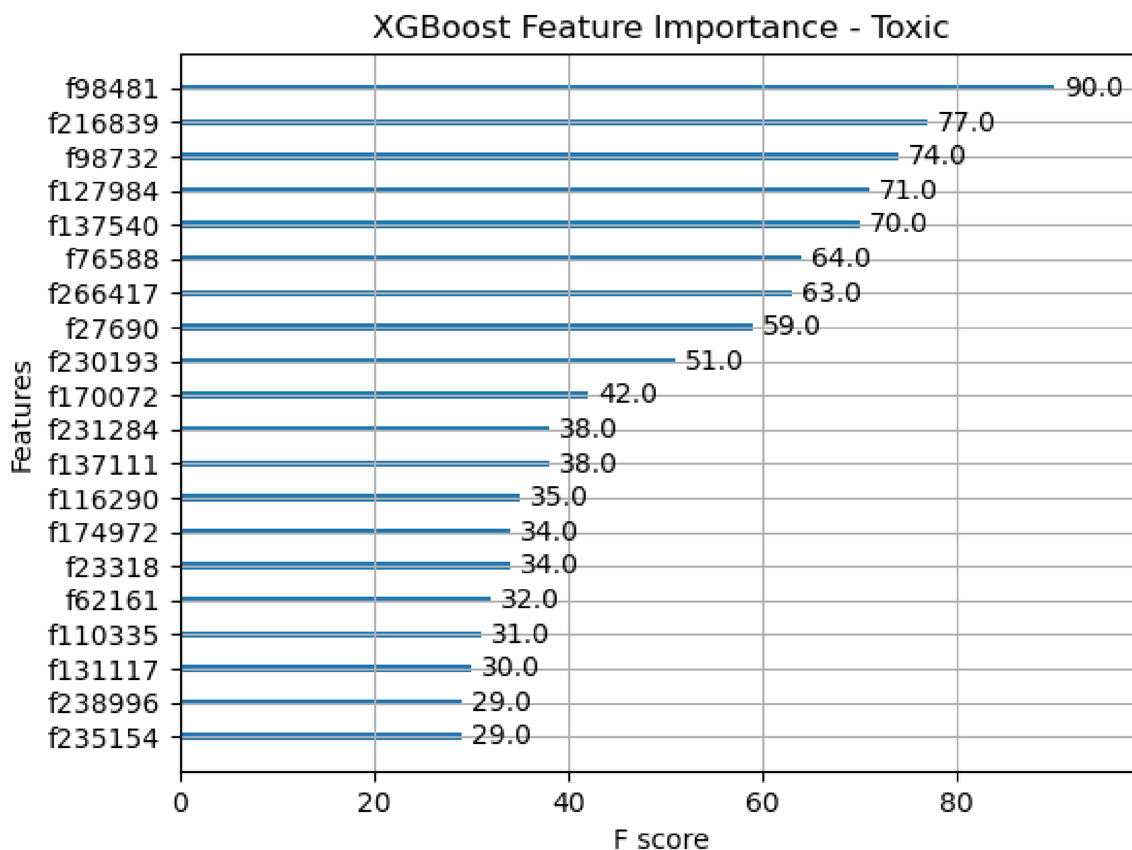
```
plt.title('ROC Curve - Logistic Regression')
plt.legend(loc='lower right')
plt.show()a
```



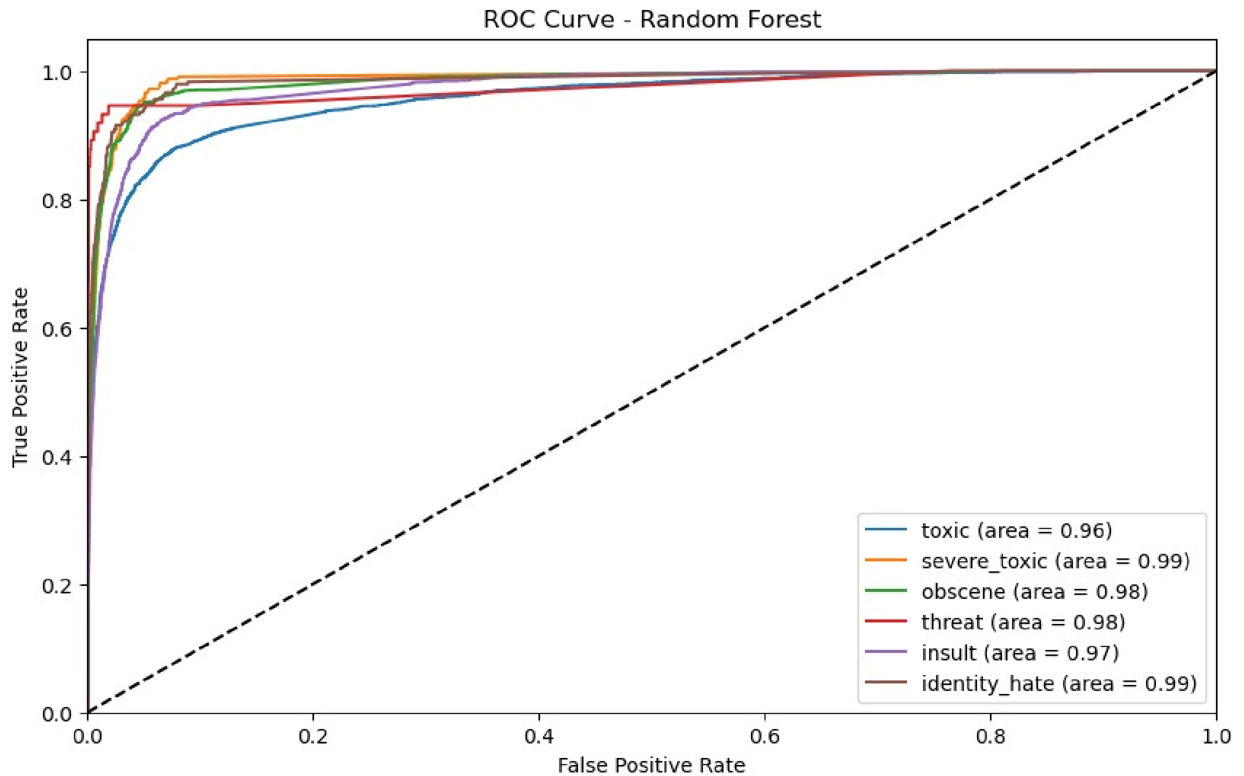

```
# Plot ROC Curve for XGBoost
plt.figure(figsize=(10, 6))

for class_name in classes:
    fpr, tpr, _ = roc_curve(y_valid_split[class_name],
xgb_models[class_name].predict_proba(X_valid_split)[:, 1])
    roc_auc = auc(fpr, tpr)
    plt.plot(fpr, tpr, label=f'{class_name} (area = {roc_auc:.2f})')

plt.plot([0, 1], [0, 1], 'k--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve - XGBoost')
plt.legend(loc='lower right')
plt.show()
```



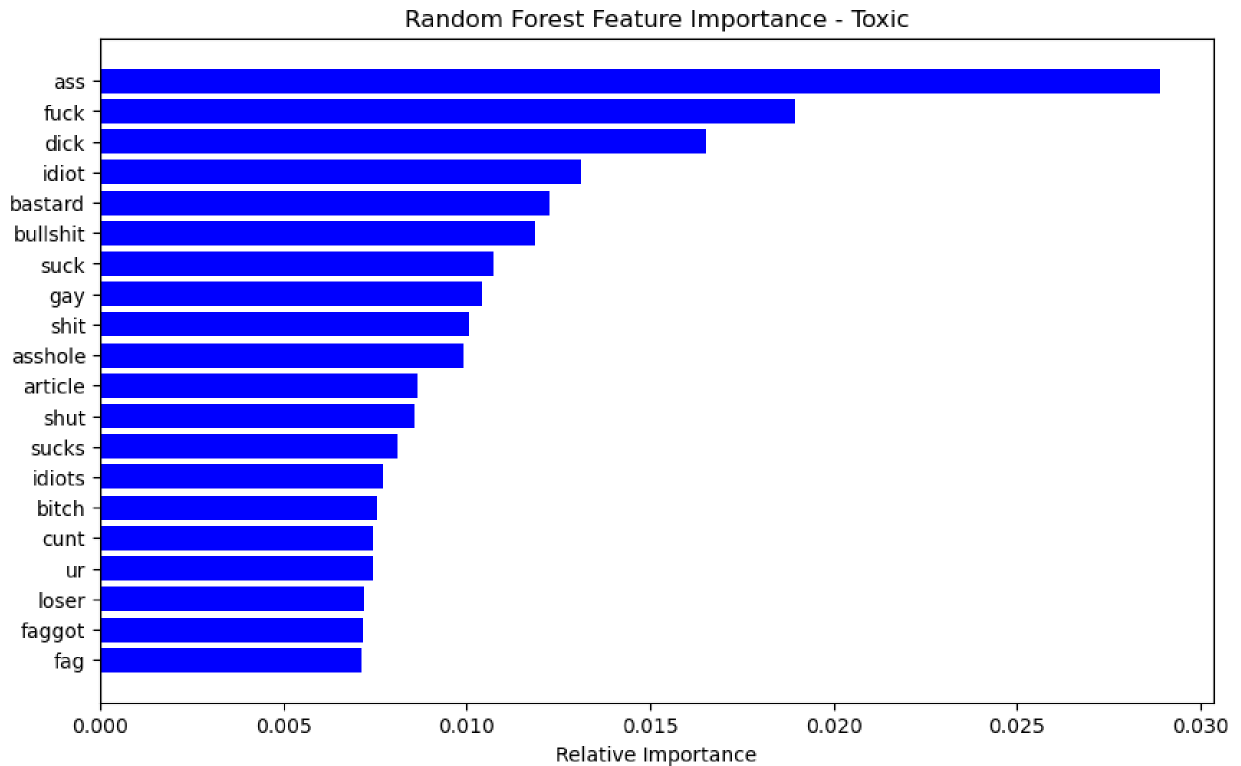

```
# Plot Feature Importance for one class (e.g., 'toxic')
xgb.plot_importance(xgb_models['toxic'], max_num_features=20,
importance_type='weight')
plt.title('XGBoost Feature Importance - Toxic')
plt.show()
```



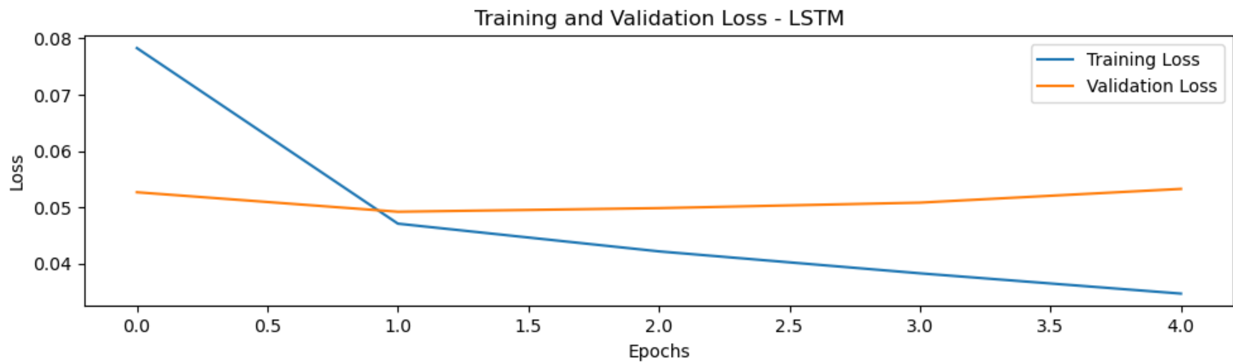

```
# Plot ROC Curve for Random Forest
plt.figure(figsize=(10, 6))

for class_name in classes:
    fpr, tpr, _ = roc_curve(y_valid_split[class_name],
rf_models[class_name].predict_proba(X_valid_split)[:, 1])
    roc_auc = auc(fpr, tpr)
    plt.plot(fpr, tpr, label=f'{class_name} (area = {roc_auc:.2f})')

plt.plot([0, 1], [0, 1], 'k--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve - Random Forest')
plt.legend(loc='lower right')
plt.show()
```

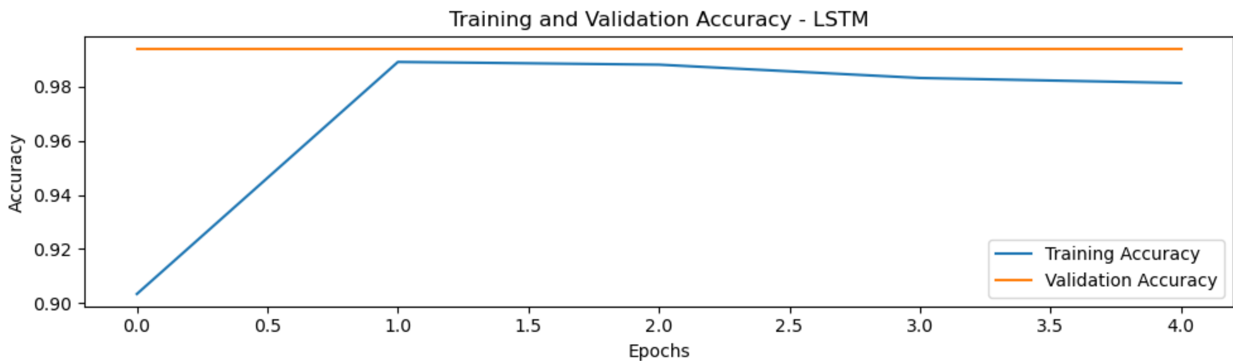



```
# Plot Feature Importance for one class (e.g., 'toxic')
importances = rf_models['toxic'].feature_importances_
indices = np.argsort(importances)[-20:]  # top 20 features

plt.figure(figsize=(10, 6))
plt.title('Random Forest Feature Importance - Toxic')
plt.barh(range(len(indices)), importances[indices], color='b',
align='center')
plt.yticks(range(len(indices)), [vectorizer.get_feature_names_out()[i]
for i in indices])
plt.xlabel('Relative Importance')
plt.show()
```




```python
# Plot training history for LSTM
plt.figure(figsize=(10, 6))

plt.subplot(2, 1, 1)
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Training and Validation Loss - LSTM')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.subplot(2, 1, 2)
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Training and Validation Accuracy - LSTM')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

plt.tight_layout()
plt.show()
```






```python
def classify_comment(input_text):
    print("Logistic Regression Predictions:")
    lr_predictions = classify_input_lr(input_text)
    for class_name, prob in lr_predictions.items():
        print(f"{class_name}: {prob:.4f}")

    print("\nXGBoost Predictions:")
    xgb_predictions = classify_input_xgb(input_text)
    for class_name, prob in xgb_predictions.items():
        print(f"{class_name}: {prob:.4f}")

    print("\nRandom Forest Predictions:")
    rf_predictions = classify_input_rf(input_text)
    for class_name, prob in rf_predictions.items():
        print(f"{class_name}: {prob:.4f}")

    print("\nLSTM Predictions:")
    lstm_predictions = classify_input_lstm(input_text)
    for class_name, prob in lstm_predictions.items():
        print(f"{class_name}: {prob:.4f}")

# Example usage
input_comment = "I can't believe you did this! You're so dumb!"
classify_comment(input_comment)
```