Toxicity Detection in Online Comments: A Multilabel NLP Approach Using TF-IDF and Word Embeddings

By

RICARDO ANDRÉ CEREZA MIGLIORINI

Submitted to

The University of Liverpool

MASTER OF SCIENCE - DATA SCIENCE AND ARTIFICIAL INTELLIGENCE

*CSCK507 – Natural Language Processing and Understanding April 2025*

Word Count: 443

27/04/2025

Toxicity Detection in Online Comments: A Multilabel NLP Approach Using TF-IDF and Word Embeddings

Submitted to

The University of Liverpool

Word Count: 443

27/03/2025

TABLE OF CONTENTS

Page

[LIST OF TABLES 2](#_Toc196650794)

[LIST OF FIGURES 3](#_Toc196650795)

[Chapter 1. Introduction 4](#_Toc196650796)

[Chapter 2. Dataset, Preprocessing, and Feature Extraction 5](#_Toc196650797)

[Chapter 3. Model Training and Evaluation 8](#_Toc196650798)

[Chapter 4. Hyperparameter Tuning and Final Model 12](#_Toc196650799)

[Chapter 5. Conclusion and Future Work 14](#_Toc196650800)

[REFERENCES 15](#_Toc196650801)

[APPENDICES 16](#_Toc196650802)

LIST OF TABLES

Page

[Table 1 – Average Word Count by Toxicity Type 6](#_Toc196650356)

LIST OF FIGURES

Page

[Figure 1. Exploratory Analysis – Distribution of Toxic Comment Types 5](#_Toc196650361)

[Figure 2. Exploratory Analysis – Correlation Between Toxicity Labels 6](#_Toc196650362)

[Figure 3. Exploratory Analysis – Distribution of Comment Length (in words) 7](#_Toc196650363)

[Figure 4. Data Processing – Word Count Distribution per Comment (Before vs After Cleaning) 7](#_Toc196650364)

[Figure 5. Model TF-IDF – Training Results: Logistic Regression 8](#_Toc196650365)

[Figure 6. Model TF-IDF – Training Results: Linear SVM 9](#_Toc196650366)

[Figure 7. Model TF-IDF – Training Results: Naïve Bayes 9](#_Toc196650367)

[Figure 8. Model TF-IDF – Overall Metrics Comparison 9](#_Toc196650368)

[Figure 9. Model Spacy Embeddings – Training Results: Logistic Regression 10](#_Toc196650369)

[Figure 10. Model Spacy Embeddings – Training Results: Linear SVM 10](#_Toc196650370)

[Figure 11. Model Spacy Embeddings – Training Results: Naïve Bayes 10](#_Toc196650371)

[Figure 12. Model Spacy Embeddings – Overall Metrics Comparison 11](#_Toc196650372)

[Figure 13. Model Spacy Embeddings – F1 Comparison 11](#_Toc196650373)

[Figure 14. Tuning RandomizedSearchCV – Optimized Logistic Regression (TF-IDF) 12](#_Toc196650374)

[Figure 15. Tuning RandomizedSearchCV – Confusion Matrix (toxic and severe\_toxic) 13](#_Toc196650375)

[Figure 16. Tuning RandomizedSearchCV – Confusion Matrix (threat and insult) 13](#_Toc196650376)

[Figure 17. Tuning RandomizedSearchCV – Confusion Matrix (obscene and identify\_hate) 13](#_Toc196650377)

[Figure 18. Kaggle – Toxic Comment Classification Challenge: Submission Result 14](#_Toc196650378)

# Introduction

Toxic comments in online discussions represent a growing challenge for digital communities. The Wikimedia Foundation, recognizing this problem, made available a dataset of Wikipedia talk page comments annotated for various types of toxicity.

In line with the evolution of Natural Language Processing (NLP) techniques described by Nadkarni, Ohno-Machado, and Chapman (2011), the goal of this project is to build an effective multilabel classification pipeline capable of detecting six types of toxicity: *toxic, severe\_toxic, obscene, threat, insult,* and *identity\_hate.*

# Dataset, Preprocessing, and Feature Extraction

The dataset, sourced from the Jigsaw Toxic Comment Classification Challenge, comprises over 150,000 Wikipedia comments annotated across six multilabel toxicity categories. An imbalance among classes like threat and identity\_hate was observed, aligning with Sharma and Kaushik (2017) who noted class skewness in real-world NLP datasets.

Following Duque (2020) and Jurafsky and Martin (2021), preprocessing involved removing URLs, punctuation, numbers, and offensive terms. Text was normalized through lowercasing, tokenization, stopword removal, and stemming, limiting each comment to 150 tokens for efficiency.

Two feature extraction strategies were adopted:

* TF-IDF Vectorization: Capturing the relative importance of unigrams and bigrams.
* Word Embeddings: Leveraging pre-trained en\_core\_web\_md vectors via spaCy to encode semantic relationships.

Both representations were standardized for compatibility with supervised machine learning models.

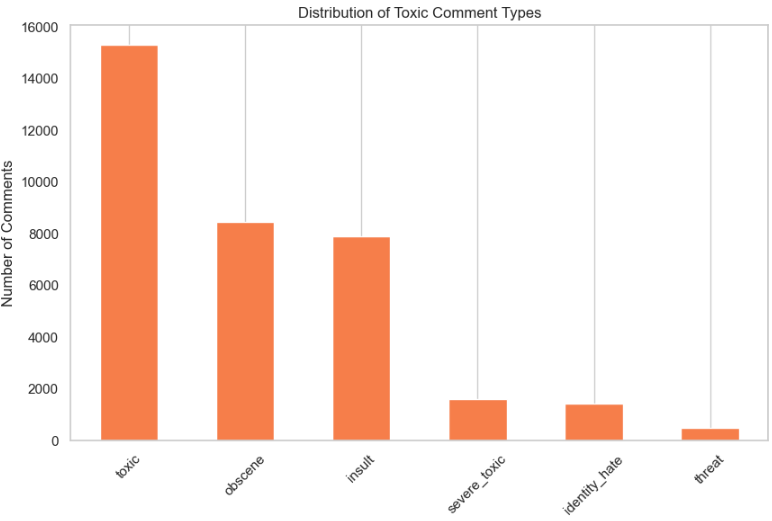


Figure 1. Exploratory Analysis – Distribution of Toxic Comment Types

Table 1 – Average Word Count by Toxicity Type

|  |  |
| --- | --- |
| **Toxicity type** | **Average word count** |
| toxic | 51.29 |
| severe\_toxic | 75.62 |
| obscene | 49.56 |
| threat | 55.17 |
| insult | 48.27 |
| identity\_hate | 52.01 |

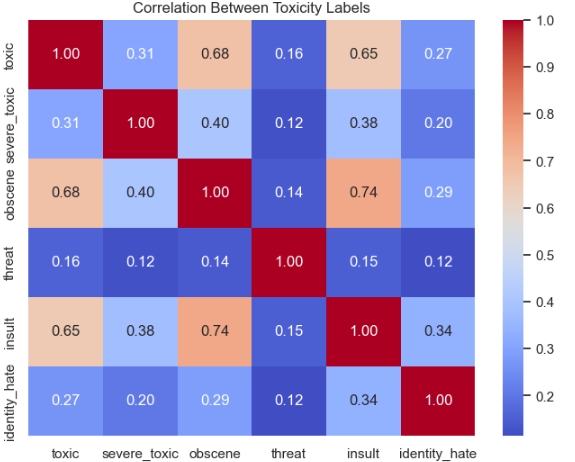


Figure 2. Exploratory Analysis – Correlation Between Toxicity Labels

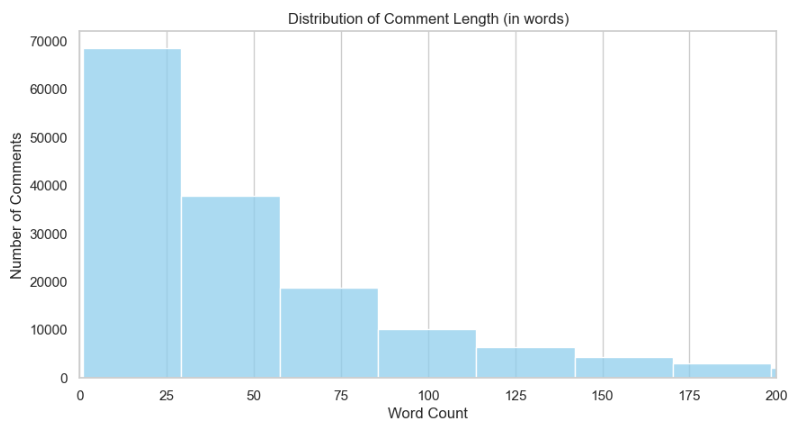


Figure 3. Exploratory Analysis – Distribution of Comment Length (in words)

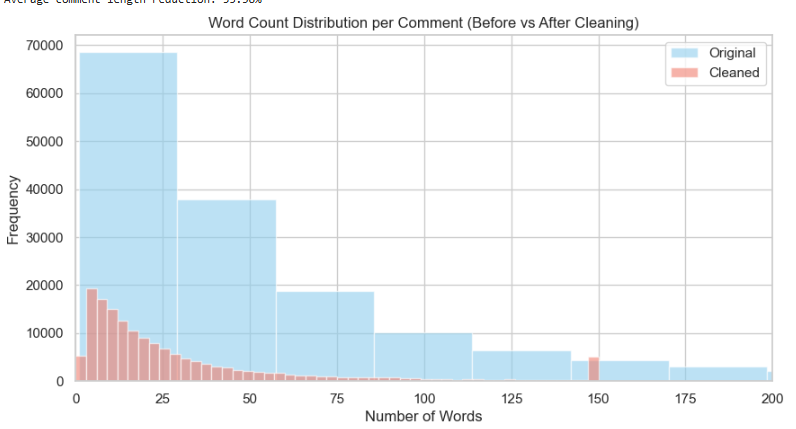


Figure 4. Data Processing – Word Count Distribution per Comment (Before vs After Cleaning)

# Model Training and Evaluation

The classification task was approached using supervised models capable of multilabel prediction, following Eisenstein (2019) and Jurafsky and Martin (2021).

Three algorithms were trained:

* Logistic Regression: A linear baseline effective for sparse, high-dimensional data.
* Linear SVM (SGDClassifier): Suitable for large feature spaces, applying hinge loss and L2 regularization.
* Naive Bayes: Using MultinomialNB for TF-IDF and GaussianNB for embeddings, following classic text classification practices.

Each model was wrapped in a MultiOutputClassifier to address multilabel outputs. Evaluation used macro-averaged F1-score, accuracy, precision, recall, and AUC when available. F1-score was prioritized due to label imbalance (Sharma and Kaushik, 2017).

Initial results showed Logistic Regression performed best on TF-IDF, while embedding models struggled with rare labels like threat and identity\_hate.

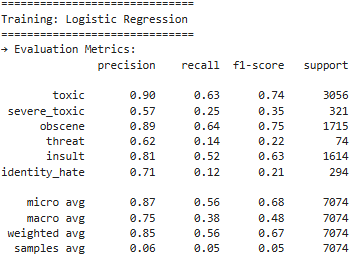


Figure 5. Model TF-IDF – Training Results: Logistic Regression

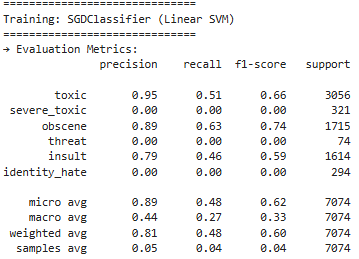


Figure 6. Model TF-IDF – Training Results: Linear SVM

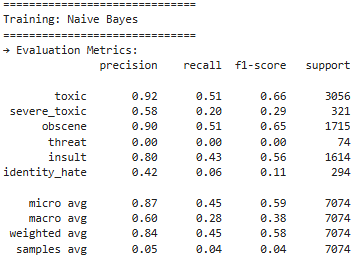


Figure 7. Model TF-IDF – Training Results: Naïve Bayes

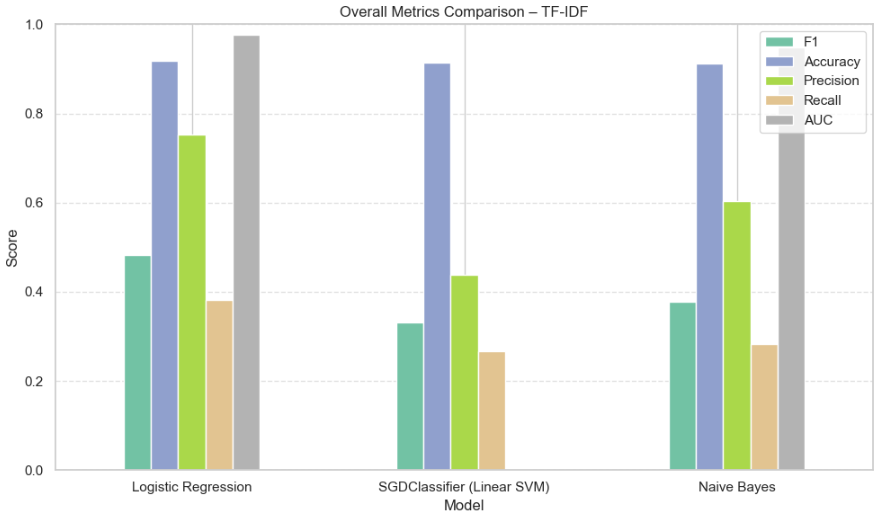


Figure 8. Model TF-IDF – Overall Metrics Comparison

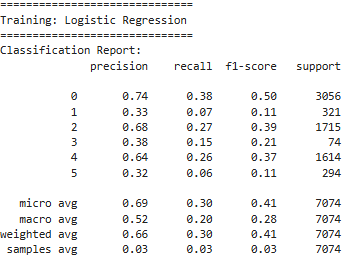


Figure 9. Model Spacy Embeddings – Training Results: Logistic Regression

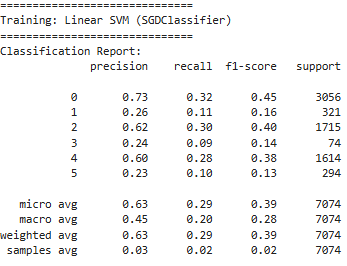


Figure 10. Model Spacy Embeddings – Training Results: Linear SVM

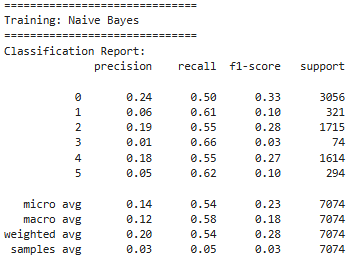


Figure 11. Model Spacy Embeddings – Training Results: Naïve Bayes

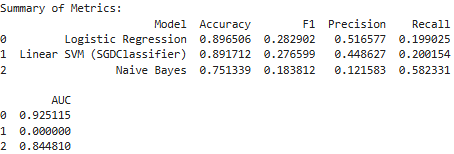


Figure 12. Model Spacy Embeddings – Overall Metrics Comparison

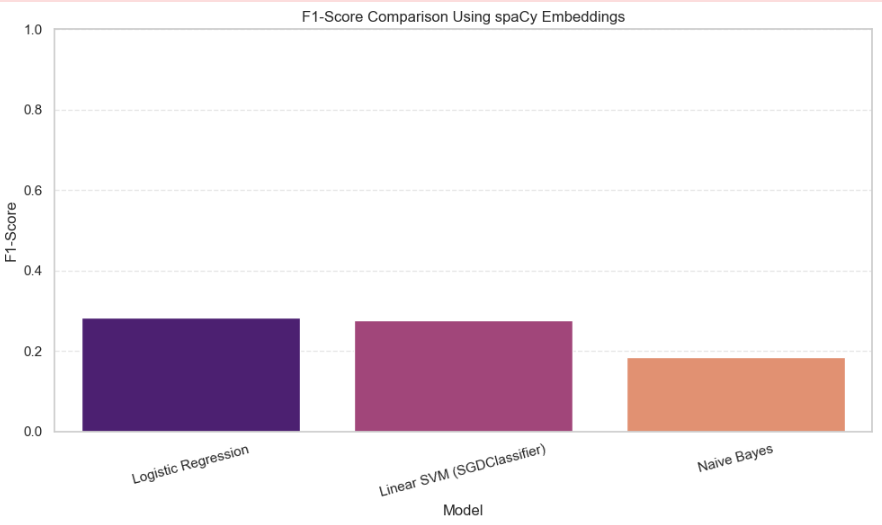


Figure 13. Model Spacy Embeddings – F1 Comparison

# Hyperparameter Tuning and Final Model

Logistic Regression was further tuned using RandomizedSearchCV, optimizing the regularization strength (Jurafsky and Martin, 2021). Following Lauriola, Lavelli, and Aiolli (2022), tuning balanced complexity and generalization.

The tuned model improved macro-F1 and AUC scores, particularly enhancing performance on rare labels. Confusion matrix analysis showed strong predictions for toxic, obscene, and insult, but difficulties remained for rarer classes. (See Figures 5 and 6.)

The final model, based on tuned TF-IDF features, was selected for Kaggle submission.

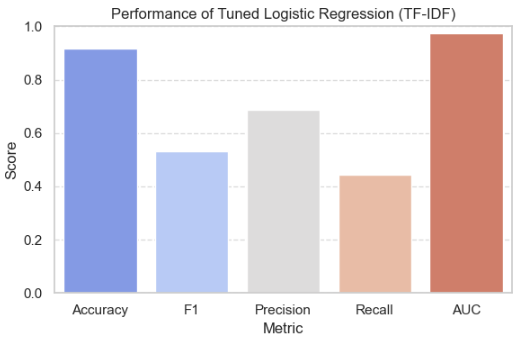


Figure 14. Tuning RandomizedSearchCV – Optimized Logistic Regression (TF-IDF)

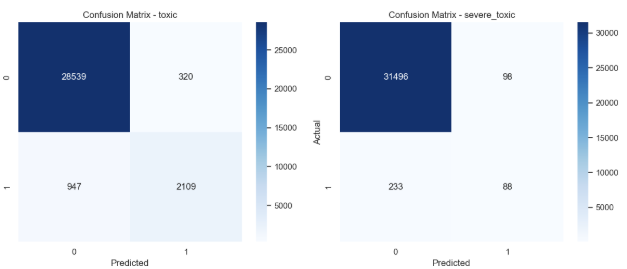


Figure 15. Tuning RandomizedSearchCV – Confusion Matrix (toxic and severe\_toxic)

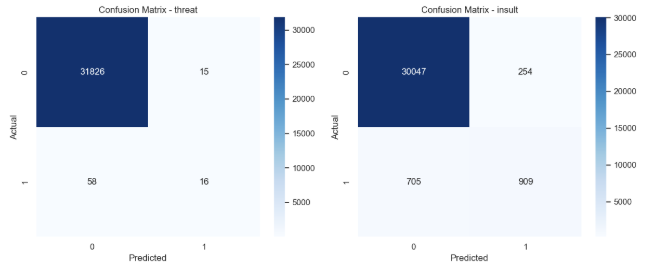


Figure 16. Tuning RandomizedSearchCV – Confusion Matrix (threat and insult)

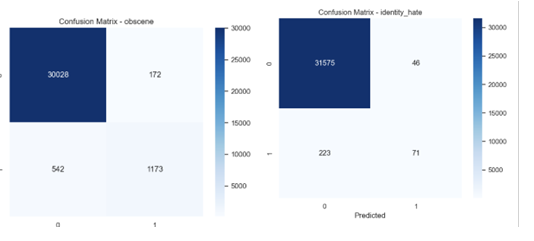


Figure 17. Tuning RandomizedSearchCV – Confusion Matrix (obscene and identify\_hate)

# Conclusion and Future Work

This project built a machine learning pipeline for detecting toxic comments across six categories. Using TF-IDF and Logistic Regression, we achieved a macro F1-score of 0.53 and AUC of 0.96, with a Private Kaggle Score of 0.965.  
Classical models proved highly effective when combined with robust preprocessing and tuning.

Word embeddings delivered lower results, especially on rare labels like threat, aligning with Jurafsky and Martin’s (2021) observations on data sparsity challenges.

For future work, improvements could include integrating Named Entity Recognition (NER), applying transformer-based models like BERT (Wolf et al., 2020), and using data augmentation to balance label distributions.

A white rectangular object with a black border

AI-generated content may be incorrect.

Figure 18. Kaggle – Toxic Comment Classification Challenge: Submission Result

REFERENCES

Costa, C.D. (2020) *Best Python Libraries for Machine Learning and Deep Learning*. Medium. Available at: <https://medium.com/> (Accessed: 25 April 2025).

Duque, T. (2020) ‘NLP Preprocessing Pipeline — what, when, why?’, *Analytics Vidhya*, 5 February. Available at: <https://www.analyticsvidhya.com/> (Accessed: 25 April 2025).

Eisenstein, J. (2019) *Introduction to Natural Language Processing*. Cambridge, Massachusetts: The MIT Press.

Jurafsky, D. and Martin, J.H. (2008) ‘Chapter 1: Introduction’, in *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. 2nd edn. Upper Saddle River, N.J: Pearson, pp. 1–16.

Jurafsky, D. and Martin, J.H. (2021) ‘Chapter 2: Regular Expressions, Text Normalization, Edit Distance’, in *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. 3rd edn. Upper Saddle River, N.J: Pearson.

Jurafsky, D. and Martin, J.H. (2021) ‘Chapter 3: N-gram Language Models’, in *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. 3rd edn. Upper Saddle River, N.J: Pearson.

Jurafsky, D. and Martin, J.H. (2021) ‘Chapter 4: Naive Bayes and Sentiment Classification’, in *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. 3rd edn. Upper Saddle River, N.J: Pearson.

Jurafsky, D. and Martin, J.H. (2021) ‘Chapter 5: Logistic Regression’, in *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. 3rd edn. Upper Saddle River, N.J: Pearson.

Lauriola, I., Lavelli, A. and Aiolli, F. (2022) ‘An introduction to Deep Learning in Natural Language Processing: Models, techniques, and tools’, *Neurocomputing*, 470, pp. 443–456. doi: 10.1016/j.neucom.2021.09.082.

Sharma, A.R. and Kaushik, P. (2017) ‘Literature survey of statistical, deep and reinforcement learning in natural language processing’, in *2017 International Conference on Computing, Communication and Automation (ICCCA)*, pp. 350–354. doi: 10.1109/CCAA.2017.8229850.

APPENDICES

###### Code Summary

This project followed a modular structure to ensure clarity and replicability of the results.

**1. Data Loading and Overview**  
We loaded and inspected training, test, and sample submission files. The dataset was verified for missing values and multilabel distribution.

**2. Exploratory Data Analysis (EDA)**  
Label frequencies and multilabel distributions were analyzed. Word counts across labels were compared, and visualizations such as WordClouds and boxplots were generated.

**3. Data Preprocessing**  
Text cleaning included the removal of URLs, punctuation, emojis, and slurs. Tokenization, stopword removal, and stemming were applied, limiting each comment to 150 tokens.

**4. Feature Extraction – TF-IDF**  
TF-IDF vectorization with unigrams and bigrams was applied. Both training and test sets were transformed, and the fitted vectorizer was exported.

**5. Model Training and Evaluation – TF-IDF**  
Logistic Regression, Linear SVM, and Naive Bayes classifiers were trained. Models were evaluated using Accuracy, F1-score, Precision, Recall, and AUC. Visual comparisons were generated, and models were saved.

**6. Feature Extraction – Word Embeddings**  
Sentence embeddings were generated using spaCy (en\_core\_web\_md), standardized, and saved.

**7. Model Training and Evaluation – Embeddings**  
The same classifiers were trained on embeddings. Evaluation focused on multilabel performance and the best model was saved.

**8. Hyperparameter Tuning – Logistic Regression (TF-IDF)**  
RandomizedSearchCV was used to optimize Logistic Regression. Tuning time, best hyperparameters, and final performance metrics were tracked and visualized.

**9. Final Prediction and Submission**  
The cleaned test set was transformed using TF-IDF. Predictions were made using the tuned Logistic Regression model, and the Kaggle submission file was created.

###### Data Loading and Preprocessing

# Chapter 1 – Data Loading and Overview

# Import necessary libraries

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

# Set visual style for plots

sns.set(style="whitegrid")

%matplotlib inline

# Load main datasets

train = pd.read\_csv("train.csv")

test = pd.read\_csv("test.csv")

test\_labels = pd.read\_csv("test\_labels.csv")

sample\_submission = pd.read\_csv("sample\_submission.csv")

# Preview the first few rows of the training set

print("Sample comments from the training dataset:")

display(train.head())

# General information about the training dataset

print("\nGeneral dataset information:")

display(train.info())

# Check for missing values in each column

print("\nMissing values per column:")

display(train.isnull().sum())

# Check distribution of toxicity labels

label\_cols = ['toxic', 'severe\_toxic', 'obscene', 'threat', 'insult', 'identity\_hate']

print("\nNumber of comments per toxicity category:")

display(train[label\_cols].sum())

# Chapter 2 – Exploratory Data Analysis (EDA)

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

import nltk

from collections import Counter

from nltk.corpus import stopwords

from nltk.tokenize import TreebankWordTokenizer

from wordcloud import WordCloud

from sklearn.feature\_extraction.text import TfidfVectorizer

# Download required NLTK stopwords

nltk.download('stopwords')

# Use a tokenizer that doesn't require punkt (avoids download issues)

tokenizer = TreebankWordTokenizer()

# Set global plot style

sns.set(style="whitegrid")

# Define multilabel columns

label\_cols = ['toxic', 'severe\_toxic', 'obscene', 'threat', 'insult', 'identity\_hate']

stop\_words = set(stopwords.words("english"))

# 2.1 – Distribution of Toxic Comment Types

plt.figure(figsize=(10, 6))

train[label\_cols].sum().sort\_values(ascending=False).plot(kind='bar', color='coral')

plt.title("Distribution of Toxic Comment Types")

plt.ylabel("Number of Comments")

plt.xticks(rotation=45)

plt.grid(axis='y')

plt.show()

# 2.2 – Number of Toxic Labels per Comment

train['n\_labels'] = train[label\_cols].sum(axis=1)

plt.figure(figsize=(8, 5))

sns.countplot(x='n\_labels', data=train, palette="Set2")

plt.title("Number of Toxic Labels per Comment")

plt.xlabel("Number of Labels")

plt.ylabel("Number of Comments")

plt.show()

multi = (train['n\_labels'] > 1).sum()

print(f"{multi} comments ({multi / len(train) \* 100:.2f}%) have multiple toxicity labels.")

# 2.3 – Word Count per Comment and by Label

train['word\_count'] = train['comment\_text'].apply(lambda x: len(str(x).split()))

print("\nAverage word count by toxicity type:")

for label in label\_cols:

avg = train[train[label] == 1]['word\_count'].mean()

print(f" - {label:<15}: {avg:.2f} words")

# 2.4 – Boxplot of Word Count by Toxicity Type

melted = pd.melt(

train,

id\_vars=['word\_count'],

value\_vars=label\_cols,

var\_name='toxic\_type',

value\_name='is\_present'

)

melted = melted[melted['is\_present'] == 1]

plt.figure(figsize=(10, 6))

sns.boxplot(data=melted, x="toxic\_type", y="word\_count", palette="pastel")

plt.title("Word Count Distribution by Toxicity Type")

plt.xlabel("Toxicity Type")

plt.ylabel("Number of Words")

plt.ylim(0, 200)

plt.grid(axis='y')

plt.show()

# 2.5 – WordCloud for Each Toxicity Class

def generate\_wordcloud(label):

text = " ".join(train[train[label] == 1]['comment\_text'].astype(str))

wordcloud = WordCloud(

stopwords=stop\_words,

background\_color='white',

max\_words=100,

width=800,

height=400

).generate(text)

plt.figure(figsize=(10, 5))

plt.imshow(wordcloud, interpolation="bilinear")

plt.axis("off")

plt.title(f"WordCloud - Class: {label}")

plt.show()

# Example WordCloud for 'toxic' class

generate\_wordcloud("toxic")

# 2.6 – Correlation Matrix Between Labels

plt.figure(figsize=(8, 6))

corr = train[label\_cols].corr()

sns.heatmap(corr, annot=True, cmap='coolwarm', fmt=".2f")

plt.title("Correlation Between Toxicity Labels")

plt.show()

# 2.7 – Histogram of Comment Length (in Words)

plt.figure(figsize=(10, 5))

sns.histplot(train['word\_count'], bins=50, kde=False, color='skyblue')

plt.title("Distribution of Comment Length (in words)")

plt.xlabel("Word Count")

plt.ylabel("Number of Comments")

plt.xlim(0, 200)

plt.grid(axis='y')

plt.show()

# 2.8 – Most Frequent Words for a Given Label (Term Frequency)

def top\_words\_by\_label(label, n=20):

texts = train[train[label] == 1]["comment\_text"].astype(str)

all\_words = " ".join(texts).lower()

tokens = [

w for w in tokenizer.tokenize(all\_words)

if w.isalpha() and w not in stop\_words

]

counter = Counter(tokens).most\_common(n)

words, freqs = zip(\*counter)

plt.figure(figsize=(10, 5))

sns.barplot(x=list(words), y=list(freqs), palette="viridis")

plt.title(f"Top {n} Frequent Words - {label}")

plt.xticks(rotation=45)

plt.ylabel("Frequency")

plt.show()

# Example: frequent words for the 'insult' label

top\_words\_by\_label("insult")

# 2.9 – Top TF-IDF Terms per Label

def top\_tfidf\_terms\_per\_label(label, n=20):

texts = train[['comment\_text', label]].copy()

texts = texts[texts[label] == 1]['comment\_text'].astype(str)

tfidf = TfidfVectorizer(stop\_words='english', max\_features=5000)

X\_tfidf = tfidf.fit\_transform(texts)

means = X\_tfidf.mean(axis=0).A1

top\_indices = means.argsort()[-n:][::-1]

terms = tfidf.get\_feature\_names\_out()[top\_indices]

scores = means[top\_indices]

plt.figure(figsize=(10, 5))

sns.barplot(x=terms, y=scores, palette="magma")

plt.title(f"Top {n} TF-IDF Terms - {label}")

plt.xticks(rotation=45)

plt.ylabel("TF-IDF Score")

plt.show()

# Example: TF-IDF for the 'obscene' label

top\_tfidf\_terms\_per\_label("obscene")

# Chapter 3 – Data Preprocessing

import re

import nltk

import pandas as pd

import matplotlib.pyplot as plt

from nltk.corpus import stopwords

from nltk.stem import PorterStemmer

from nltk.tokenize import TreebankWordTokenizer

# Download stopwords if not already present

nltk.download('stopwords')

# Initialize NLP tools

stop\_words = set(stopwords.words("english"))

stemmer = PorterStemmer()

tokenizer = TreebankWordTokenizer()

# Custom list of informal expressions, interjections, and slang

custom\_stopwords = set([

'gee', 'omg', 'wtf', 'lol', 'rofl', 'lmao', 'idk', 'idc', 'hahaha', 'haha',

'yeah', 'uhh', 'umm', 'damn', 'bro', 'yo', 'dude', 'nah', 'ayy', 'tsk', 'awww'

])

# Offensive word roots for regex detection (creative variants)

offensive\_roots = [

'fuck', 'shit', 'bitch', 'nigger', 'fag', 'asshole', 'cunt', 'dick', 'bastard'

]

# Create flexible regex patterns to match variations of toxic words

def create\_offensive\_pattern(word):

chars = list(word)

pattern = r'\W\*'.join(chars) # Example: f\W\*u\W\*c\W\*k

return rf'\b{pattern}\b'

# Compile all regex patterns into one

offensive\_patterns = [create\_offensive\_pattern(word) for word in offensive\_roots]

compiled\_offensive\_regex = re.compile('|'.join(offensive\_patterns), re.IGNORECASE)

# Robust text cleaning function

def clean\_text(text, max\_words=150):

if not isinstance(text, str) or text.strip() == "":

return ""

# 1. Convert to lowercase

text = text.lower()

# 2. Remove excessive spaces and line breaks

text = text.strip()

text = re.sub(r'\s+', ' ', text)

# 3. Remove URLs

text = re.sub(r'http\S+|www\S+|https\S+', '', text)

# 3b. Replace creative swear word variants with placeholder

text = compiled\_offensive\_regex.sub("toxicword", text)

# 4. Remove emojis and non-ASCII characters

text = re.sub(r'[^\x00-\x7F]+', '', text)

# 5. Remove numbers and punctuation

text = re.sub(r'\d+', '', text)

text = re.sub(r'[^\w\s]', '', text)

# 6. Tokenization

tokens = tokenizer.tokenize(text)

# 7. Remove stopwords and apply stemming

clean\_tokens = []

for w in tokens:

if w not in stop\_words and w not in custom\_stopwords and len(w) > 2 and w.isalpha():

try:

clean\_tokens.append(stemmer.stem(w))

except RecursionError:

continue

tokens = clean\_tokens

# 8. Limit to maximum word count

tokens = tokens[:max\_words]

return ' '.join(tokens)

# Apply text cleaning to a DataFrame

def apply\_cleaning\_to\_dataframe(df, text\_column='comment\_text', new\_column='clean\_text'):

df[new\_column] = df[text\_column].apply(clean\_text)

return df

# Apply to training and testing datasets

train = apply\_cleaning\_to\_dataframe(train)

test = apply\_cleaning\_to\_dataframe(test)

# Show an example of original and cleaned comment

print("Original comment:")

print(train['comment\_text'].iloc[0])

print("\nCleaned comment:")

print(train['clean\_text'].iloc[0])

# Save cleaned versions to CSV

train.to\_csv("train\_clean.csv", index=False)

test.to\_csv("test\_clean.csv", index=False)

# Word count comparison before and after cleaning

train['original\_word\_count'] = train['comment\_text'].apply(lambda x: len(str(x).split()))

train['clean\_word\_count'] = train['clean\_text'].apply(lambda x: len(str(x).split()))

print("\nAverage word count – Original:", train['original\_word\_count'].mean())

print("Average word count – Cleaned:", train['clean\_word\_count'].mean())

reduction = 100 \* (1 - (train['clean\_word\_count'].mean() / train['original\_word\_count'].mean()))

print(f"Average comment length reduction: {reduction:.2f}%")

# Visualization: word count distribution

plt.figure(figsize=(10, 5))

plt.hist(train['original\_word\_count'], bins=50, alpha=0.6, label='Original', color='skyblue')

plt.hist(train['clean\_word\_count'], bins=50, alpha=0.6, label='Cleaned', color='salmon')

plt.legend()

plt.title("Word Count Distribution per Comment (Before vs After Cleaning)")

plt.xlabel("Number of Words")

plt.ylabel("Frequency")

plt.xlim(0, 200)

plt.show()

# Display 5 random cleaned examples

samples = train[['comment\_text', 'clean\_text']].sample(5, random\_state=42)

for \_, row in samples.iterrows():

print("=" \* 80)

print("🔹 Original:")

print(row['comment\_text'])

print("\n🔧 Cleaned:")

print(row['clean\_text'])

print("=" \* 80)

###### Feature Extraction

# Chapter 4 – Feature Extraction with TF-IDF

import pandas as pd

import numpy as np

from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.model\_selection import train\_test\_split

import joblib

# 1. Load preprocessed training and test datasets

train = pd.read\_csv("train\_clean.csv")

test = pd.read\_csv("test\_clean.csv")

# 2. Define input texts and multilabel targets

X\_text = train["clean\_text"].fillna("")

y = train[["toxic", "severe\_toxic", "obscene", "threat", "insult", "identity\_hate"]]

# 3. Configure TF-IDF vectorizer

vectorizer = TfidfVectorizer(

max\_features=10000, # Limit to 10,000 most important features

ngram\_range=(1, 2), # Use unigrams and bigrams

stop\_words="english", # Remove built-in English stopwords

sublinear\_tf=True # Apply sublinear TF scaling (log(1 + tf))

)

# 4. Fit the vectorizer on training text and transform both train/test sets

X\_tfidf = vectorizer.fit\_transform(X\_text)

X\_test\_tfidf = vectorizer.transform(test["clean\_text"].fillna(""))

# 5. Split TF-IDF features and labels into training and validation sets

X\_train, X\_val, y\_train, y\_val = train\_test\_split(

X\_tfidf, y,

test\_size=0.2,

random\_state=42

)

# 6. Display shape and sample features for sanity check

print("TF-IDF shape (train):", X\_train.shape)

print("TF-IDF shape (val):", X\_val.shape)

print("TF-IDF shape (test):", X\_test\_tfidf.shape)

print("Example feature names:", vectorizer.get\_feature\_names\_out()[:20])

# 7. Save vectorized data and TF-IDF model

joblib.dump((X\_train, y\_train), "tfidf\_train.pkl")

joblib.dump((X\_val, y\_val), "tfidf\_val.pkl")

joblib.dump((X\_test\_tfidf, test[["id"]]), "tfidf\_test.pkl")

joblib.dump(vectorizer, "tfidf\_vectorizer.pkl")

print("✅ TF-IDF feature extraction complete and saved.")

###### Model Training and Evaluation

# Chapter 5 – Model Training and Evaluation with TF-IDF

import joblib

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from sklearn.linear\_model import LogisticRegression, SGDClassifier

from sklearn.naive\_bayes import MultinomialNB

from sklearn.multioutput import MultiOutputClassifier

from sklearn.metrics import (

classification\_report,

f1\_score,

accuracy\_score,

precision\_score,

recall\_score,

roc\_auc\_score

)

from sklearn.preprocessing import MinMaxScaler

# 1. Load TF-IDF features and multilabel targets

X\_train, y\_train = joblib.load("tfidf\_train.pkl")

X\_val, y\_val = joblib.load("tfidf\_val.pkl")

# 2. Load TF-IDF vectorizer to extract interpretable feature names

vectorizer = joblib.load("tfidf\_vectorizer.pkl")

feature\_names = vectorizer.get\_feature\_names\_out()

# 3. Define models to evaluate

models = {

"Logistic Regression": LogisticRegression(max\_iter=1000, solver="lbfgs", n\_jobs=-1),

"SGDClassifier (Linear SVM)": SGDClassifier(loss="hinge", penalty="l2", max\_iter=1000, random\_state=42),

"Naive Bayes": MultinomialNB()

}

summary = [] # Store global performance metrics

# 4. Train and evaluate each model

for name, base\_model in models.items():

print(f"\n{'=' \* 30}\nTraining: {name}\n{'=' \* 30}")

model = MultiOutputClassifier(base\_model)

model.fit(X\_train, y\_train)

y\_pred = model.predict(X\_val)

# Normalize predicted probabilities for AUC (only for models that support predict\_proba)

if hasattr(model, "predict\_proba"):

try:

y\_proba = model.predict\_proba(X\_val)

if isinstance(y\_proba, list):

y\_proba = np.array([p[:, 1] for p in y\_proba]).T

scaler = MinMaxScaler()

y\_proba = scaler.fit\_transform(y\_proba)

auc = roc\_auc\_score(y\_val, y\_proba, average="macro")

except Exception:

auc = 0.0

else:

auc = 0.0

# Print classification report

print("→ Evaluation Metrics:")

print(classification\_report(y\_val, y\_pred, target\_names=y\_train.columns))

# Compute per-label F1-score and Accuracy

f1\_scores = f1\_score(y\_val, y\_pred, average=None)

acc\_scores = [

accuracy\_score(y\_val.iloc[:, i], y\_pred[:, i])

for i in range(y\_val.shape[1])

]

macro\_f1 = f1\_score(y\_val, y\_pred, average="macro")

macro\_acc = accuracy\_score(y\_val, y\_pred)

macro\_precision = precision\_score(y\_val, y\_pred, average="macro", zero\_division=0)

macro\_recall = recall\_score(y\_val, y\_pred, average="macro", zero\_division=0)

summary.append({

"Model": name,

"F1": macro\_f1,

"Accuracy": macro\_acc,

"Precision": macro\_precision,

"Recall": macro\_recall,

"AUC": auc

})

# Bar chart per class

plt.figure(figsize=(10, 4))

plt.bar(y\_train.columns, f1\_scores, color="dodgerblue", alpha=0.7, label="F1-score")

plt.bar(y\_train.columns, acc\_scores, color="orange", alpha=0.5, label="Accuracy")

plt.title(f"{name} – Per Class Metrics")

plt.ylabel("Score")

plt.ylim(0, 1)

plt.legend()

plt.grid(axis="y", linestyle="--", alpha=0.5)

plt.xticks(rotation=30)

plt.tight\_layout()

plt.show()

# Interpretability for linear models

if hasattr(base\_model, "coef\_"):

for i, label in enumerate(y\_train.columns):

print(f"\nClass: {label}")

top\_positive = np.argsort(base\_model.coef\_[i])[-10:][::-1]

top\_negative = np.argsort(base\_model.coef\_[i])[:10]

print("Top Positive Words:", feature\_names[top\_positive])

print("Top Negative Words:", feature\_names[top\_negative])

# Save model

filename = f"model\_{name.replace(' ', '\_').lower()}.pkl"

joblib.dump(model, filename)

print(f"✅ Saved: {filename}")

# 5. Summary: Compare model performance

summary\_df = pd.DataFrame(summary)

# Plot global comparison

plt.figure(figsize=(10, 6))

summary\_df.set\_index("Model")[["F1", "Accuracy", "Precision", "Recall", "AUC"]].plot(

kind="bar", figsize=(10, 6), colormap="Set2"

)

plt.title("Overall Metrics Comparison – TF-IDF")

plt.ylabel("Score")

plt.ylim(0, 1)

plt.grid(axis="y", linestyle="--", alpha=0.6)

plt.xticks(rotation=0)

plt.tight\_layout()

plt.show()

# Chapter 6 – Feature Extraction with Word Embeddings (spaCy)

import numpy as np

import pandas as pd

import spacy

from tqdm import tqdm

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import StandardScaler

# 1. Load spaCy model with pre-trained word vectors

print("🔄 Loading spaCy model with pre-trained vectors...")

nlp = spacy.load("en\_core\_web\_md")

# 2. Load cleaned dataset

train\_df = pd.read\_csv("train\_clean.csv")

X\_text = train\_df["clean\_text"].fillna("").tolist()

y = train\_df[["toxic", "severe\_toxic", "obscene", "threat", "insult", "identity\_hate"]].values

# 3. Function to extract sentence-level embeddings using spaCy

def get\_spacy\_embeddings(texts, nlp\_model):

"""

Generate sentence embeddings by averaging token vectors from spaCy.

If no vector is available, return a zero vector of the same dimension.

"""

print("🧠 Generating spaCy embeddings...")

embeddings = []

for doc in tqdm(nlp\_model.pipe(texts, batch\_size=500), total=len(texts), desc="spaCy Embedding"):

if doc.has\_vector:

embeddings.append(doc.vector)

else:

embeddings.append(np.zeros(nlp\_model.vocab.vectors\_length))

return np.array(embeddings)

# Apply the embedding extraction

X\_embeddings = get\_spacy\_embeddings(X\_text, nlp)

# 4. Split data into training and validation sets

X\_train, X\_val, y\_train, y\_val = train\_test\_split(

X\_embeddings, y, test\_size=0.2, random\_state=42

)

# 5. Standardize the embeddings using z-score normalization

scaler = StandardScaler()

X\_train\_scaled = scaler.fit\_transform(X\_train)

X\_val\_scaled = scaler.transform(X\_val)

# 6. Save all generated arrays to disk

np.save("X\_emb\_train.npy", X\_train)

np.save("X\_emb\_val.npy", X\_val)

np.save("y\_emb\_train.npy", y\_train)

np.save("y\_emb\_val.npy", y\_val)

np.save("X\_emb\_train\_scaled.npy", X\_train\_scaled)

np.save("X\_emb\_val\_scaled.npy", X\_val\_scaled)

# 7. Final summary

print("\n✅ Embedding extraction complete.")

print(f"Training set shape: {X\_train.shape} | Validation set shape: {X\_val.shape}")

# Chapter 7 – Modeling with spaCy Embeddings

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

import joblib

from sklearn.linear\_model import LogisticRegression, SGDClassifier

from sklearn.naive\_bayes import GaussianNB

from sklearn.tree import DecisionTreeClassifier

from sklearn.multioutput import MultiOutputClassifier

from sklearn.metrics import (

classification\_report,

accuracy\_score,

f1\_score,

precision\_score,

recall\_score,

roc\_auc\_score

)

from sklearn.preprocessing import MinMaxScaler

# 1. Load scaled word embeddings and target labels

X\_train = np.load("X\_emb\_train\_scaled.npy")

X\_val = np.load("X\_emb\_val\_scaled.npy")

y\_train = np.load("y\_emb\_train.npy")

y\_val = np.load("y\_emb\_val.npy")

# 2. Define models to evaluate with multilabel support

models = {

"Logistic Regression": MultiOutputClassifier(

LogisticRegression(max\_iter=1000, solver="lbfgs", n\_jobs=-1, random\_state=42)

),

"Linear SVM (SGDClassifier)": MultiOutputClassifier(

SGDClassifier(loss="hinge", penalty="l2", max\_iter=1000, random\_state=42)

),

"Naive Bayes": MultiOutputClassifier(

GaussianNB()

)

}

results = [] # Store overall model performance

# 3. Train and evaluate each model

for name, model in models.items():

print(f"\n{'=' \* 30}\nTraining: {name}\n{'=' \* 30}")

model.fit(X\_train, y\_train)

y\_pred = model.predict(X\_val)

# Compute metrics

acc = accuracy\_score(y\_val, y\_pred)

f1 = f1\_score(y\_val, y\_pred, average="macro", zero\_division=0)

precision = precision\_score(y\_val, y\_pred, average="macro", zero\_division=0)

recall = recall\_score(y\_val, y\_pred, average="macro", zero\_division=0)

# Compute AUC if model supports predict\_proba

try:

y\_proba = model.predict\_proba(X\_val)

if isinstance(y\_proba, list):

y\_proba = np.array([p[:, 1] for p in y\_proba]).T

scaler = MinMaxScaler()

y\_proba = scaler.fit\_transform(y\_proba)

auc = roc\_auc\_score(y\_val, y\_proba, average="macro")

except Exception:

auc = 0.0

# Print report

print("Classification Report:")

print(classification\_report(y\_val, y\_pred, zero\_division=0))

results.append({

"Model": name,

"Accuracy": acc,

"F1": f1,

"Precision": precision,

"Recall": recall,

"AUC": auc

})

# 4. Save and display summary metrics

results\_df = pd.DataFrame(results)

print("\nSummary of Metrics:")

print(results\_df)

results\_df.to\_csv("spaCy\_model\_metrics.csv", index=False)

# 5. Plot F1-score comparison

plt.figure(figsize=(10, 6))

sns.barplot(x="Model", y="F1", data=results\_df, palette="magma")

plt.title("F1-Score Comparison Using spaCy Embeddings")

plt.ylabel("F1-Score")

plt.ylim(0, 1)

plt.xticks(rotation=15)

plt.grid(axis="y", linestyle="--", alpha=0.5)

plt.tight\_layout()

plt.show()

# 6. Save the best-performing model

best\_model\_name = results\_df.sort\_values(by="F1", ascending=False).iloc[0]["Model"]

best\_model = models[best\_model\_name]

best\_model\_path = f"best\_model\_embeddings\_{best\_model\_name.replace(' ', '\_').lower()}.pkl"

joblib.dump(best\_model, best\_model\_path)

print(f"\n✅ Best spaCy-based model: {best\_model\_name}")

print(f"📦 Model saved to: {best\_model\_path}")

###### Hyperparameter Tuning

# Chapter 8 – Hyperparameter Tuning with RandomizedSearchCV (Optimized TF-IDF)

import time

import joblib

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.linear\_model import LogisticRegression

from sklearn.model\_selection import RandomizedSearchCV

from sklearn.multioutput import MultiOutputClassifier

from sklearn.metrics import (

classification\_report,

accuracy\_score,

f1\_score,

precision\_score,

recall\_score,

roc\_auc\_score,

confusion\_matrix

)

# 1. Load TF-IDF features and labels

X\_train, y\_train = joblib.load("tfidf\_train.pkl")

X\_val, y\_val = joblib.load("tfidf\_val.pkl")

# 2. Define base model

base\_model = LogisticRegression(

max\_iter=1000,

solver="lbfgs",

penalty="l2",

n\_jobs=-1

)

# 3. Define parameter grid

param\_dist = {

"estimator\_\_C": np.logspace(-3, 2, 10)

}

# 4. Wrap for multilabel

multi\_model = MultiOutputClassifier(base\_model)

# 5. Setup search

random\_search = RandomizedSearchCV(

estimator=multi\_model,

param\_distributions=param\_dist,

n\_iter=6,

scoring="f1\_macro",

cv=2,

verbose=2,

random\_state=42,

n\_jobs=-1

)

# 6. Run search with timing

print("\n🚀 Starting RandomizedSearchCV...")

start = time.time()

random\_search.fit(X\_train, y\_train)

print(f"\n⏱️ Tuning took {(time.time() - start) / 60:.2f} minutes")

# 7. Evaluation

best\_model = random\_search.best\_estimator\_

y\_pred = best\_model.predict(X\_val)

accuracy = accuracy\_score(y\_val, y\_pred)

f1 = f1\_score(y\_val, y\_pred, average="macro")

precision = precision\_score(y\_val, y\_pred, average="macro")

recall = recall\_score(y\_val, y\_pred, average="macro")

# 8. AUC Score

try:

y\_proba = best\_model.predict\_proba(X\_val)

if isinstance(y\_proba, list):

y\_proba = np.array([p[:, 1] for p in y\_proba]).T

auc = roc\_auc\_score(y\_val, y\_proba, average="macro")

except:

auc = 0.0

print("\n✅ Best Hyperparameters Found:")

print(random\_search.best\_params\_)

print("\n📊 Final Classification Report:")

print(classification\_report(y\_val, y\_pred, target\_names=y\_train.columns))

# 9. Save model

joblib.dump(best\_model, "best\_model\_tuned\_logistic\_regression.pkl")

# 10. Plot metrics

metrics\_df = pd.DataFrame({

"Metric": ["Accuracy", "F1", "Precision", "Recall", "AUC"],

"Score": [accuracy, f1, precision, recall, auc]

})

plt.figure(figsize=(6, 4))

sns.barplot(data=metrics\_df, x="Metric", y="Score", palette="coolwarm")

plt.title("Performance of Tuned Logistic Regression (TF-IDF)")

plt.ylim(0, 1)

plt.grid(axis="y", linestyle="--", alpha=0.7)

plt.tight\_layout()

plt.show()

# 11. Confusion Matrices (per label)

labels = list(y\_train.columns)

y\_val\_df = pd.DataFrame(y\_val, columns=labels)

y\_pred\_df = pd.DataFrame(y\_pred, columns=labels)

fig, axes = plt.subplots(2, 3, figsize=(18, 10))

axes = axes.ravel()

for i, label in enumerate(labels):

cm = confusion\_matrix(y\_val\_df[label], y\_pred\_df[label])

sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", ax=axes[i])

axes[i].set\_title(f"Confusion Matrix - {label}")

axes[i].set\_xlabel("Predicted")

axes[i].set\_ylabel("Actual")

plt.tight\_layout()

plt.show()

###### Final Submission

## Chapter 9 – Kaggle Submission File Generation

import pandas as pd

import numpy as np

import joblib

# 1. Load original test dataset with comment IDs

test\_df = pd.read\_csv("test.csv")

X\_test\_text = test\_df["comment\_text"].fillna("")

# 2. Load pre-cleaned version of the test set

test\_clean = pd.read\_csv("test\_clean.csv")

X\_clean\_text = test\_clean["clean\_text"].fillna("")

# 3. Load the saved TF-IDF vectorizer and transform the cleaned test data

vectorizer = joblib.load("tfidf\_vectorizer.pkl")

X\_test\_tfidf = vectorizer.transform(X\_clean\_text)

# 4. Load the tuned Logistic Regression model

best\_model = joblib.load("best\_model\_tuned\_logistic\_regression.pkl")

# 5. Predict class probabilities for each label

y\_pred\_proba = best\_model.predict\_proba(X\_test\_tfidf)

# 6. Handle MultiOutputClassifier's output format

# If it's a list, convert to a 2D numpy array with one column per label

if isinstance(y\_pred\_proba, list):

y\_pred\_proba = np.array([p[:, 1] for p in y\_pred\_proba]).T

# 7. Build submission DataFrame with correct column names

submission = pd.DataFrame(

y\_pred\_proba,

columns=["toxic", "severe\_toxic", "obscene", "threat", "insult", "identity\_hate"]

)

# 8. Add comment ID column at the beginning of the DataFrame

submission.insert(0, "id", test\_df["id"])

# 9. Export the submission file in the required format

submission.to\_csv("submission.csv", index=False)

print("✅ Submission file 'submission.csv' generated successfully.")