# Toxic Comments Classification

A major project report submitted in partial fulfillment of the requirement for the award of degree of

**Bachelor of Technology**

in

**Computer Science & Engineering / Information Technology**

*Submitted by*

**Garv Mehta (201370)**

**Sahil Thakur (201525)**

**Arnav Verma (201523)**

*Under the guidance & supervision of*

***Mr. Arvind Kumar***


**Department of Computer Science & Engineering and Information Technology**

**Jaypee University of Information Technology,**

**Waknaghat, Solan - 173234 (India)**

# Candidate's Declaration

I hereby declare that the work presented in this report entitled **'Toxic Comments Classification'** in partial fulfillment of the requirements for the award of the degree of **Bachelor of Technology** in **Computer Science & Engineering / Information Technology** submitted in the Department of Computer Science & Engineering and Information Technology**,** Jaypee University of Information Technology, Waknaghat is an authentic record of my own work carried out over a period from August 2023 to December 2023 under the supervision of **Mr. Arvind Kumar** (Assistant Professor, Department of Computer Science & Engineering and Information Technology).

The matter embodied in the report has not been submitted for the award of any other degree or diploma.

(Student Signature with Date)

Student Name: Garv Mehta(201370)

Sahil Thakur (201525)

Arnav Verma (201523)

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

(Supervisor Signature with Date) Supervisor Name: Mr. Arvind Kumar Designation: Assistant Professor Department: Computer Science and Engineering Dated:

# CERTIFICATE

This is to certify that the work which is being presented in the project report titled **"Toxic Comments Classification"** in partial fulfillment of the requirements for the award of the degree of B.Tech in Computer Science And Engineering and submitted to the Department of Computer Science And Engineering, Jaypee University of Information Technology, Waknaghat is an authentic record of work carried out by Garv Mehta 201370, Sahil Thakur 201525 and Arnav Verma 201523 during the period from July 2023 December 2023 under the supervision of Mr. Arvind Kumar, Department of Computer Science and Engineering, Jaypee University of Information Technology, Waknaghat.

Garv Mehta (201370)
Sahil Thakur (201525)
Arnav Verma (201523)

The above statement made is correct to the best of my knowledge.

Mr. Arvind
Kumar
Assistant
Professor,
Computer Science & Engineering and Information Technology Jaypee University of Information Technology, Solan.

# ACKNOWLEDGEMENT

Firstly, I express my heartiest thanks and gratefulness to almighty God for His divine blessing makes it possible for us to complete the project work successfully.

I am really grateful and wish my profound indebtedness to Supervisor Mr Arvind Kumar Assistant Professor, Department of Computer Science and Engineering, Jaypee University of Information Technology, Waknaghat. Deep knowledge & keen interest of my supervisor in the field of Machine Learning to carry out this project. His endless patience, scholarly guidance, continual encouragement, constant supervision, constructive criticism, valuable advice, reading many inferior drafts and correcting them at all stages have made it possible to complete this project.

I would like to express my heartiest gratitude to Mr Arvind Kumar **,** Department of CSE, for his kind help to finish my project.

I would also generously welcome each one of those individuals who have helped me straightforwardly or in a roundabout way in making this project a win. In this unique situation, I might want to thank the various staff individuals, both educating and non-instructing, which have developed their convenient help and facilitated my undertaking.

Finally, I must acknowledge with due respect the constant support and patients of my parents.

Garv Mehta (201370)

Sahil Thakur (201525)

Arnav Verma (20152)

# Table of Content

# List of Abbreviations

Machine Learning (ML)

Natural Language Processing (NLP)

Convolutional Neural Network (CNN)

Recurrent Neural Network (RNN)

Long Short-Term Memory Networks (LSTM)

Bidirectional Encoder Representations from Transformers (BERT)

Global Vectors for Word Representation (GloVe)

# List of Figures

**List of Tables**

# ABSTRACT

In the field of online communication, discussion toxicity has become a major problem, preventing people from expressing their true selves and participating in diverse ideas due to the threat of attack or harassment. This research aims to solve this problem by using natural language processing (NLP) technology, which is primarily a technique of machine learning, to detect the use of toxic comments.

The aim is to detect and minimize toxic comments, to construct a space where authentic self-expression interchange of numerous ideas occurs. The research focuses on NLP which constitutes a significant element in machine learning and is instrumental in providing computers with speech comprehension, analysis, management and control powers. This is more than a simple act of reading in the sense that it reveals the minute details about a language.

Artificial intelligence is dependent on natural language processing. This technology goes way beyond understanding simple language, encompassing such functions as sentiment analysis, text categorization, speech recognition, and automatic text summarization. NLP helps with text management, enabling detection and control of harmful texts.

To conclude, this is an extensive study on whether nlp can tackle toxic arguments. The research aims to help create safer, better and more effective online discussions by understanding, identifying and carefully addressing toxic content.

# CHAPTER-1 PROJECT INTRODUCTION

## 1.1. Introduction

In the early days of the web, communication was mostly via email, and the platform suffered from widespread spam problems. The challenge today is to classify emails as wanted (good) or unwanted (bad), with an emphasis on identifying and filtering spam. However, over time, especially with the emergence and spread of social media, the field of online communication and information flow has also changed.

The changes brought about by social media platforms mark a significant change in the nature of online interaction. Unlike email management, social media demonstrates a strong communication and communication capability that increases the scale of content creation and use. In the changing digital ecosystem, the need to classify content as “good” or “bad” has become more evident.

The importance of sharing content stems from the need to reduce social harm and prevent behavior change in communities around the world. As people engage in different forms of expression and interaction, the potential for exposure to harmful, offensive, or toxic content increases. The project therefore aims to solve today's challenges by using machine learning and classification algorithms, to clarify the difference between construction involvement (meaning "good") and Content that may propagate harmful behavior (known as “bad”).

The project works in the context of the digital communications revolution, demonstrating the importance of content distribution in managing the online environment. The project creates a better and safer experience for users by preventing the spread of negative content and harmful behavior. The introduction lays the groundwork for understanding the evolution of online communication, the issues arising from the transition, and the overall goal of the project, which is to divide good content to increase safety in society.

The Figure 1 below shows the distribution of toxic comments in different categories based

on how severe these comments are.

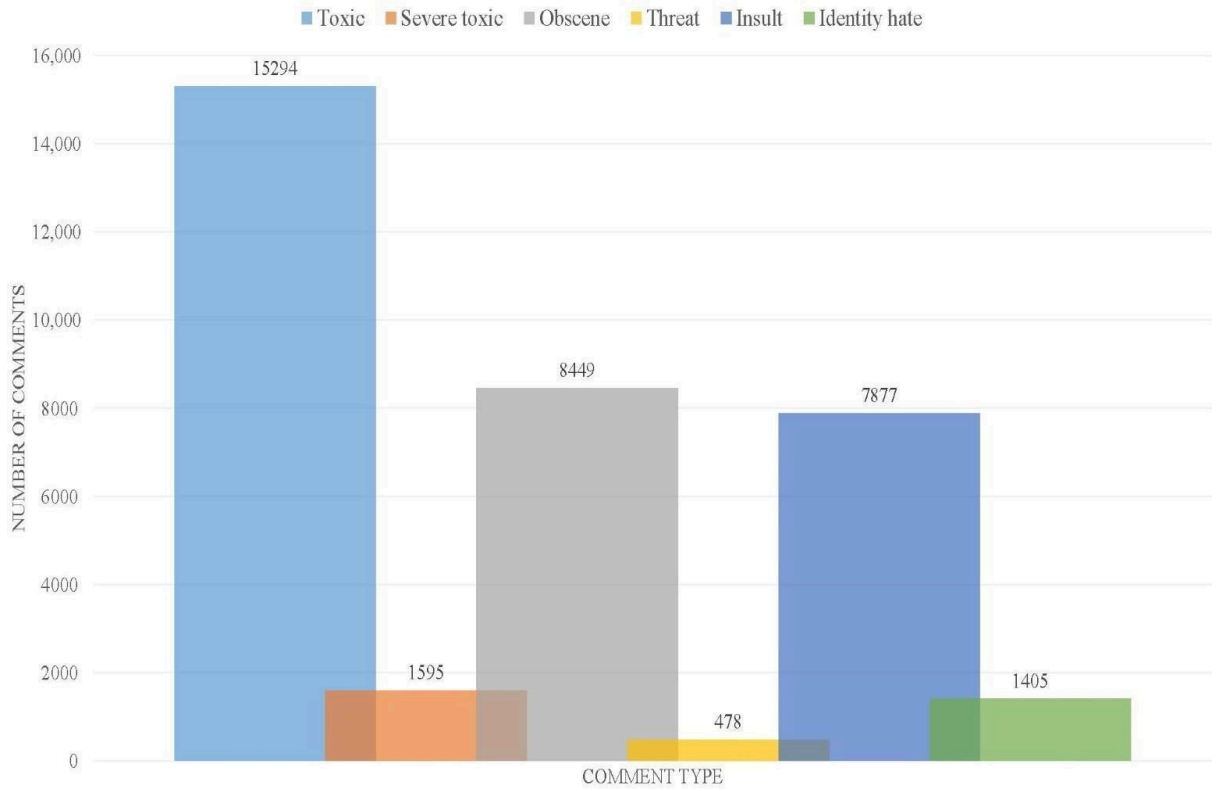


Figure 1: Distribution of toxic comments

Figure 1 shows the distribution of the toxicity of comments in six different categories and the number of comments counted in each category with general toxic comments being the most occurring and serious threats having the least occurrence.

## 1.2. Problem Statement

In the online communications landscape dominated by social media platforms, a significant and immediate challenge emerges: accurately identifying and disseminating digital content to distinguish its contribution to fostering a positive online environment from the potential harm or toxicity it may propagate. The creation of content that fosters interaction and sustains viability in the digital realm is crucial. The proliferation of objectionable content or issues across numerous online platforms presents a pressing concern for societal well-being, necessitating the establishment of a secure and safe online environment. Addressing this challenge requires the utilization of advanced Natural Language Processing (NLP) techniques such as Convolutional Neural Networks (CNN), Long

Short-Term Memory networks (LSTM), and Bidirectional Encoder Representations from Transformers (BERT). The integration of these technologies is vital for the precise interpretation and dissemination of digital content. Leveraging NLP methodologies alongside intricate neural network techniques like CNN, LSTM and BERT holds immense promise in facilitating analysis, classification, and strategic management of digital content. This integration plays a pivotal role in regulating the online environment, effectively managing networks, and mitigating social risks associated with harmful online content.

## 1.3. Objectives

**We have the following objectives when working on this project:**

**1) Data Collection and Preparation:**

Gather a diverse dataset of online comments, including both toxic and non-toxic examples. Preprocess the data by cleaning, tokenizing, and performing feature extraction. Split the dataset into training, validation, and testing sets.

**2) Model Architecture And Development:**

Utilizing a set of deep learning algorithms such as a special type of Recurrent Neural Network (RNN) called Long Short-Term Memory networks (LSTM), along with word embedding techniques like BERT to explore diverse approaches for text categorization.

**3) Performance Evaluation:**

Conducting a thorough evaluation of the chosen algorithms by employing a dataset specifically curated for the purpose. The focus is on assessing the accuracy of each algorithm, considering their individual capabilities in accurately classifying toxic comments.

**4)Comparison of Results:**

Comparing the accuracy achieved by the different algorithms and to compare them to baseline machine learning models like Logistic Regression, Naive Bayes, SVM, etc. to identify the model that exhibits the highest performance in classifying toxic comments.

**5)Model Selection:**

Selecting the model with the highest accuracy as the final classifier for predicting the toxicity of comments. This step involves making informed decisions based on the evidence derived from the performance evaluations.

**6)Unseen Data Prediction:**

Using the selected model to forecast the toxicity levels of unseen data. Creating a Gradio interface to check the model's accuracy in identifying real world toxic comments.

## 1.4. Significance and Motivation of the Project Work

**Significance of Toxic Comment Classification:**

**Online Safety:**

The trend of online communication has led to an increase in toxic comments, posing a danger to the safety and health of users. Creating an effective classification system for toxic comments is crucial to create a safe online environment.

**User Experience:**

Toxic comments can downgrade the entire user experience on social media platforms and forums. Using the techniques to filter this content can improve the quality of user interaction and support the online community.

**Motivation:**

**Social impact:**

The motivation behind this project is wanting to help improve the health of the online community. By detecting toxic comments, we aim to reduce the negative effects of negative comments and improve online health.

**Technical Challenges:**

Classifying toxic comments poses significant problems in natural language processing. The motivation comes from the contradictions involved in creating models that can understand and distinguish toxic from non-toxic elements.

**Applications:**

**Content Moderation:**

Social media forums and other online communities can use the template to use content management strategy for effective and inclusive online space.

**User Empowerment:**

Provide users with tools to filter out harmful content to give users more control over their online experience. The purpose of this project is to provide users with a safer and more useful experience.

**Advancements In Deep Learning:**

**Searching for modeling resources:**

The motivation of this study is the opportunity to think about the potential of deep learning in the context of natural language, which techniques like LSTM, CNN address.

Understanding its strengths and limitations will contribute to broader and more in-depth research.

**Importance:**

**Importance in online platforms:**

As online platforms continue to evolve, the need to identify toxic comments becomes even more important. The motivation for this project is the potential use of design in the development of content moderation.

**Educational Impact:**

**Learning and Skills Development:**

Doing this project provides an opportunity for hands-on learning in natural language processing, deep learning, and model evaluation. Considerations include the educational impact on individuals participating in the project.

In summary, the purpose and motivation of the **Toxic Comments Classification** project is to make a safe environment and contribute to the betterment of online users. The project is dedicated to benefiting the digital community and exploring the potential of deep learning for language understanding.

## 1.5. Organization of Project Report

### Chapter 2: Literature Survey

In this chapter, we studied the existing works on toxic comments classification. Existing literature includes various methods and frameworks. Previous research findings stress on the difficulties related to handling unbalanced data sets as well as the significance of

designing relevant features and their connection with the applicability of model interpretability. Additionally, looking at different metrics of evaluation used in similar studies gives more meaning to measuring a model's performance, taking into consideration F1-score, AUC, etc.

**Chapter 3: System Development**

In this chapter, we go into the project design, data preparation and implementation of our project. This involves capturing, cleaning up, and balancing to achieve the desired size and composition for the sample. The use of different algorithms is discussed. Furthermore, the section identifies a few evaluation metrics including accuracy, precision, recall, and F1-score for measuring the performance of the classification mechanisms employed.

**Chapter 4: Testing**

In this chapter, we discuss the testing strategies used to evaluate the performance of our model.

**Chapter 5: Results and Evaluation**

This chapter provides the results of the above algorithms as indicators of their effectiveness in the categorization of toxic comments. The findings obtained through the analysis help to understand whether the models are effective to solve it.

**Chapter 6: Conclusions and Future Scope:**

The final chapter of the research is the conclusion that sums up the main conclusions based on the results obtained during the study. The paper presents the significance of reliably identically the toxicities, outlining some improvements or recommendations. The final part discusses suggestions of further improvements.

# CHAPTER 2 LITERATURE SURVEY

## 2.1 Overview of Relevant Literature

Table 1: Literature Overview

| S. No. | Paper Title [Cite] | Journal/ Conference (Year) | Tools/ Techniques/ Dataset | Results | Limitations |
|---|---|---|---|---|---|
| 1. | An Automated Toxicity Classification on Social Media Using LSTM and Word Embedding | 2023 | LSTM + Glove<br>LSTM + BERT | 0.93<br>0.94 | It doesn't compare the LSTM model's performance with other commonly used approaches, such as CNNs or traditional methods. |
| 2. | Toxic Comments Classification | 2022 | Naïve Bayes<br>SVM<br>Logistic Regression<br>Random Forrest<br>XgBoost | 0.791330<br>0.789219<br>0.800149<br>0.583778<br>0.755186 | Deep Learning methods are not used that can increase the end result. |
| 3. | BERT base model for toxic comment analysis on Indonesian social media | 2022 | Multilingual BERT<br>IndoBERT<br>IndoRoBERTa Small | 0.887853<br>0.889781<br>0.885604 | The study does not explore a broader range of pre-trained models or architectures. |
| 4. | Multilabel Toxic Comment Detection and Classification | 2021 | Logistic Regression<br>SVM Binary Relevance<br>SVM Classifier Chains<br>LSTM Word2Vec<br>LSTM Glove | 0.7462<br>0.6886<br>0.7005<br>0.9664<br>0.9666 | Doesn't tell the result in accuracy or F-1 score. |
| 5. | Toxic Comment Classification | 2020 | Naïve Bayes<br>LSTM | 0.64<br>0.73 | NLP classifiers can be improved. |
| 6. | Toxic Comment Classification and Unintended Bias | 2019 | Naïve Bayes<br>Logistic Regression<br>CNN Glove 50d<br>CNN Glove 100d<br>CNN fastText 300d | 0.9202<br>0.9474<br>0.9390<br>0.9450<br>0.9484 | Hyperparameters can be improved, RNN architecture can be used to improve. |
| 7. | Convolutional Neural Networks for Toxic Comment Classification | 2018 | CNN fix<br>CNN rand<br>KNN<br>LDA<br>NB<br>SVM | 0.912<br>0.895<br>0.697<br>0.808<br>0.719<br>0.811 | It doesn't explore hyperparameters for the traditional methods. |

**An Automated Toxicity Classification on Social Media Using LSTM and Word Embedding (2023):**

The research employed two distinct training datasets: the first one categorized the comments as toxic or not while the second grouped them in the six types of toxicity. Standard procedures of text analysis like punctuation removal, lemmatization, and stop-word elimination were used for data preprocessing.

The LSTM model had been described consisting of the word-embedding layer, dropout layer, LSTM layer, and finally the output layer which formed the heart of the classification process. This paper investigated the use of two kinds of word embeddings, GloVe and BERT, to pretrain classifiers. As a contrast, the GloVe model was described as a static and context independent approach, whereas the BERT framework was depicted as a context sensitive word-embedding technique able to capture fine and meaningful nuance and relatedness between words within their specific contexts.

**Results**

Accuracy and F1 score are common evaluation metrics, which are used to assess how effective a classification model is. The experiment showed that LSTM model used together with BERT word embeddings reached up to 94% accuracy and 0.89 F1-score for toxicity detection. The fact that BERT's contextualized embeddings trumped static-embeddings like GloVe was demonstrated in this.

**Conclusions and Future Work**

Accordingly, the conclusion emphasized that bias in the data should be addressed and underlined the importance of word embeddings as an approach towards enhancing classification accuracy. Moreover, it repeated that even though GloVE word embeddings trained with huge corpus showed high score for accuracy, they were outdone by the performance of BERT.

**Toxic Comments Classification (2022):**

This study involved making use of different machine learning techniques in identifying the toxicity present in textual data. Models used are logistic regression, random forest, SVM, NB, and XG boost classifier techniques. The effectiveness of each method was assessed by how successfully it identified inappropriate items within the data. With this, we were in a position of measuring these algorithms so as to determine which was best suited for classifying such toxic messages.

**Results**

After evaluation, we considered the different criteria for judging the models such as accuracy, precision, and also recall. Of all the models that we tried, logistic regression proved the highest accuracy of 80%. Logistic regression demonstrated accuracy in separating the toxic and non-toxic comments as the optimal technique towards the present classify work undertaking. Among all the used models, logistic regression has the highest performance of correctly detecting toxic substance.

**Conclusions and Future Work**

The results of the study establish that the logistic model is, by far, the best approach to toxicity identification in text data and social media commentary. The next step would involve testing different machine learning models such as RNN, BERT, multilayer perceptron, and GRU to improve on the precision and speed with which harmful comments are detected. This will lead to better classification of harmful online content and improved moderation and management of online discussions hence improved performance.

**BERT base model for toxic comment analysis on Indonesian social media (2022):**

The study employed three pre-trained models such as multilingual BERT (MBERT), IndoBERT and IndoROBERTa Small to identify toxic comments in Indonesian social media. In particular, these models were tailored to cater for multilabel classification that involves distinguishable types of poisonous articles including hate speech, extremism,

pornography and vilification.

**Results:**

The most effective among them was IndoBERT with F1 Score of 0.8897. This result was far better than that of other models. The model was good at detecting subtle details regarding different types of obscene material including hate speech, extremism, pornography, libel and more. The strong performance of IndoBERT validates its utility in fighting cyber-toxacity and enhancing healthy social media practices in Indonesia.

**Conclusions and Future Work**:

The results show that IndoBERT is able to detect toxic posts from Indonesian social networking sites. One possible direction for future work is on improving the precision of toxic comment classification with the help of these Pre-Trained Models. Exploring how to revise to suit real time moderation and creating safety environments across all online platforms could also help to minimize spreading hazardous materials.

**Multilabel Toxic Comment Detection and Classification (2021):**

The paper uses several machine learning and deep learning models for identifying and classifying toxic remarks. The models comprise of Logistic Regression, SVM, and LSTM with various embodiments such as Glove and Word2Vec. Such algorithms are applied to a Kaggle dataset consisting of labeled comments in different categories of toxicity.

**Results**

The best performance among all LSTM models is achieved using Glove embedding after thorough assessment. The model performs excellently as it generates a high score of about 96.67% for the ROC AUC metric in multi-label classification. The hamming loss and log loss scores for the model are the best as they prove its high accuracy and efficiency on toxicity classification in comparison with other models.

**Conclusions and Future Work**

The results of this research demonstrate the superiority of the LSTM for categorizing and detecting toxicness in comments. Other future research directions would involve the use of more sophisticated models such as BERT and GRUs, trying out ensemble methods to enhance accuracy of classification and fine-tuning the available models for high levels of efficiency in classifying toxic comments. Their purpose is to provide more elaborate information on categorization for enhancing the security of internet users.

**Toxic Comments Classificaton (2020):**

This study used two different models for categorizing comments into toxic and non-toxic ones. Models included Naive Bayes because of its simplicity and effectiveness in text classification tasks, as well as LSTM, a RNN suitable for sequential data such as language. The performance of these models was measured as a way of determining the accuracy of differentiating hostile and non-hostile posts.

**Results**

LSTM was shown to be more precise and accurate and hence its F1 score(0.73) far surpassed that of Naïve Bayes(0.64). LSTM outperformed the Naïve Bayes in overall balanced classification demonstrated by the F1 score. Therefore, LSTM proved to be the best model for identifying toxic comments, with high accuracy, in the dataset used.

**Conclusions and Future Work**

The results of this study lay strong foundation for future research areas and more accurate approaches to categorise toxic comments. Extending the range of classifiers to deal with multi-label classification problems related to the seven comment classes described by the Kaggle competition can add more depth to the perception and precision in identifying toxic comments.# Exploring other advanced algorithms and techniques like SVC and CNN

provide opportunities to enhancing model's precision, but more work is needed.

**Toxic Comment Classification and Unintended Bias (2019):**

The initial modelling involved use of Naïve Bayes model with n-gram bag-of-words and logistic regression using tf-idf. Speed-wise, the Logistic Regression model exceeded some CNN models in its performance. Using Keras and tensorFlow, CNNs constructed from pretrained word embeddings from GlOve and FastText. A three layered CNN architecture involved maxim pooling, RELUs, and drop-outs for regularization was put into operation. Categorical cross-entropy loss with binary output indicating whether a comment is toxic or not.

**Results**

For seven iterations, each CNN model was trained using a batch size of 128. Importantly, FastText CNN presented the best accuracy of 94.84%. On the other hand, the use of AUC methodology revealed problems related to score conflation, especially in connection with some ethnic sub-groups such as 'black', 'white' and even gay or lesbian homosexuals. It could be argued that there are biased forecasts for

**Conclusions and Future Work**

The categorization of toxic comments was very accurate, and it revealed some implicit prejudices that were present in the model's prognoses. Future works should include using bias data to penalize negative identity biases, optimized hyperparameter tuning, incorporating pre-processing with translation, and adopting alternative neural network architectures such as a BLSTM model.

**Convolutional Neural Networks for Toxic Comment Classification (2018):**

This work focuses on a comparison of the CNN to the BoW approach in classifying toxic

comments. It uses the Kaggle set of talk-page edits in Wikipedia. Text classification procedures like SVM, NB, kNN, and LDA are used for the evaluation.
The CNN models including the ones with random initialization or word2vec-based word representations always beat conventional techniques in classifying toxic remarks into two categories.

**Results**

Experimentation reveals the supremacy of CNN models with more than 90% accuracy, precision and recalls being higher than those for SVM, kNN, NB, and LDA.

**Conclusions and Future Work**

The randomly initialized and word2vec-based CNN variants have a clear advantage over traditional methods for identifying toxic comments on multiple metrics. Finally the study argues that CNNs have potential in providing a solution to toxic comments in the future by increasing online communication safety. These results provide a basis for further work based on CNN-based schemes for adaptive training in the toxic comment classification task. Such analysis may be supplemented by incorporation of more advanced CNN models, utilizing the principles of neural attentions and a more objective comparison between n gram approaches. Further, exploring the usefulness of transfer learning and fine-tuned pre-trained models may also help improve the performance of CNNs towards classifying toxic comments in real-time on the internet.

## 2.2 Key Gaps in the Literature

**Some potential key gaps in the previous literatures were:**

**Paper 1: Automated Toxicity classification on social media using LSTM and word embedding (2023).**

**Limitations:**

Data Bias Concern: It emphasised that bias in the data had to be addressed. Despite that, it failed to address the ways of addressing this problem thus affecting the models' generalizability.
Limited Scope: It is important to point out the fact that this study primarily concentrated on comparing GloVe vs. BERT, while the other factors that may have also impacted the model's performance might have been overshadowed.

**Paper 2: Toxic Comments Classification (2022) .**

**Limitations:**

Lack of Advanced Models: The study evaluated different machine learning methods; however, it did not consider utilizing complex deep learning architectures such as transformer-based models such as BERT and GPT that may have improved its performance. Future Work Needs Clarity: It included a discussion on future works such as trying other models but no specific plans or procedures was provided on the improvement procedure.

**Paper 3: Toxic Comment Classification Using BERT Base Model for Indonesian Social Media in 2022.**

**Limitations:**

Limited Comparative Analysis: However, the study was not as broad concerning a comparative analysis with other current models of high performance.
Scalability Issues: However, the scalability of the model and limitations on real-time moderation across platforms were not widely discussed in the paper
.

**Paper 4: Towards Multilabel Toxic Comment Detection and Classification (2021).**

**Limitations:**

Limited Diversity in Model Comparison: However, this study majorly concentrated on the LSTM models with GloVe embeddings and may have left out other models and embedding techniques.
Insufficient Exploration of Ensemble Methods: The article did not elaborate on some of the ensemble methods they tried in an attempt to improve accuracy.

**Paper 5: Toxic Comment Classification (2020).**

**Limitations:**

Scope Limited to Binary Classification: The study focused only on a limited two-class toxicity prediction system, which does not cater for other categories of toxicity.
Limited Exploration of Models: It identified some suggestions for improvement and also said something about trying more sophisticated models without saying, however, in what way would such models improve the accuracy?

**Paper 6: Toxic Comment Classification and Unintentional Bias.**

**Limitations:**

Bias Mitigation Strategies: It gave biased-ness issues on the model, and yet it was not comprehensive enough in strategies necessary for reducing the biases whose reliance on predictive ability may have been compromised.
Inadequate Neural Network Exploration: The paper also indicated the need for the researchers to explore other architectures such as BLSTM for reduction of bias and improvement in classification accuracy but did not outline it clearly.

**Paper 7: Toxic Comment Classification using Deep Convolutional Neural Networks (2018).**

**Limitations:**

Lack of Comprehensive Comparison: The study has demonstrated the superiority of the CNN models but does not provide a complete comparison of the latest improved advanced models that may have adopted some new techniques that were not present during the research time.
Limited Scope: The article compared CNNs to conventional approaches; however, broader insights into alternative methods that can enhance performance in toxic comment classification were not discussed.

The contribution of each study is significant, but it comes with a set of issues which can be improved during future studies aiming at enhancing the accuracy of toxic comment classification models and their applicability.

# Chapter 3 SYSTEM DEVELOPMENT

## 3.1. Requirements and Analysis

**Language :-**

**Python:**

Python offers several benefits for toxic comment classification:
Rich Ecosystem: Python has libraries such as NLTK, Spacy, and scikit-learn for NLP operations that facilitate text preprocessing and machine learning model implementation.
Ease of Use: With an easy-to-use syntax, it makes coding and understanding easier even for novice programmers and machine learners.
NLP Libraries: Various Python libraries such as tokenization, stemming, lemmatization, and sentiment analysis are readily available to make text preprocessing easier to use in toxic comment analysis.
ML & DL Frameworks: Classification models are easily implemented in Python using frameworks such as TensorFlow, Keras, PyTorch and scikit-learn.
Community Support: The python community has an enormous range of documentation, tutorials and forums where you can get help or updated information.
Flexibility: With python, it is not difficult to combine different third parties' libraries for end to end data pipelines creating.
Open-Source: Python and associated libraries are free and open source allowing for collaborative solutions development regarding the toxic comment classifications in this research.
Cross-Platform Compatibility: Because python's a platform independent language a code can run on different OS without much change, increasing its accessibility.

**SoftwareTechnologies :-**

**TensorFlow:** An open source distributed training system designed by Google for creating and teaching machine learning algorithms, including neural networks.

**Pandas:** An open source python module with high level data structures and powerful APIs of various data manipulation and analytical methods on structured data and time.

**NumPy:** Fundamental Python package for scientific computing, including support for large, multi-dimensional arrays and matrices as well as a set of mathematical operations.

**Jupyter Notebook:** A web application where one can write, edit and share documents that include live code, equations, visualizations, and story text. It is commonly applied for data cleansing, transformering, modelization, and visualizing.

## 3.2. Project Design and Architecture

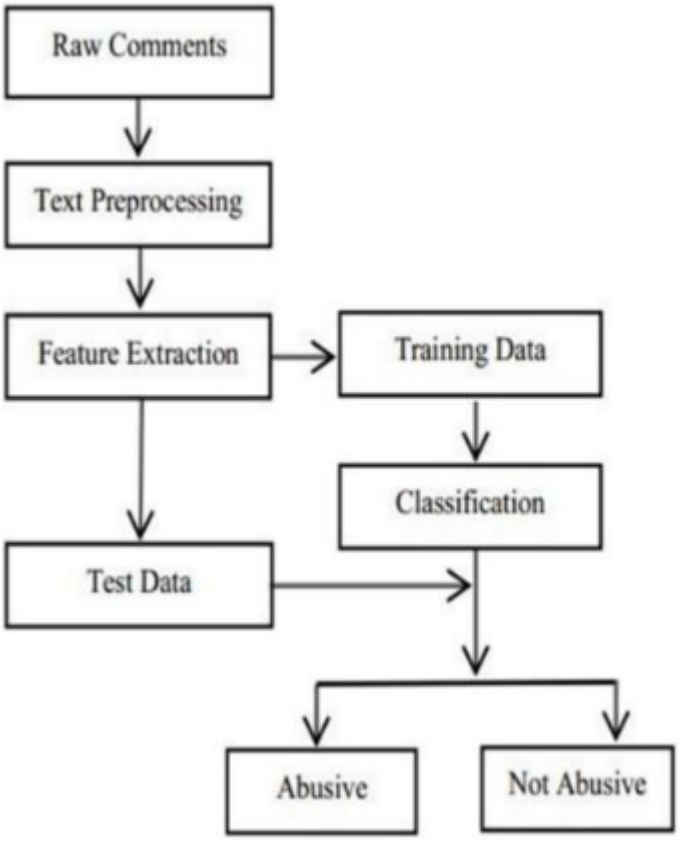


Figure 2: Flow chart for implementation

The above figure 2 shows the flow chart for the implementation of our 'Toxic Comments Classification' project.

**1) Dataset:** The main goal is to start with an initial dataset containing descriptions of comments classified as toxic or non-toxic.

**2) PreProcessing:** The aim is to prepare the data for the training model through various processes. This includes removing spaces, numbers, URLs, and stopped words, as well as lemmatization functions.

**3) Classification design:** The aim here is to create a special model for classifying toxic messages. This algorithm comprises of different algorithms and Word Embedding. The models should be trained as well as adjusting the parameters.

**4) Model Evaluation:** This is a phase that seeks to measure the quality of the model as well as identify the best one of all. The evaluation metrics include accuracy, precision, recall, F1 score, among others. Further, techniques like verification are employed to check on the accuracy of the assessment.

## 3.3 Data Preparation

**Jigsaw Multilingual Toxic Comment Classification** dataset from Kaggle is used in this project.

## 3.4 Implementation

**Step 1: Setting Up The Environment:**

As it is written in the code in figure 3, this is the foundation of the environment for training a deep learning model, especially for NLP tasks, with the BERT model, using PyTorch and PyTorch Lightning, seaborn and matplotlib are the two platforms used for visualization.
Imports: pandas and numpy for data manipulation, tqdm for progress bars, torch for deep learning. Sklearn provides machine learning tools advantageous to the Sklearn for machine learning.
Setting up the Environment: Arranging plotting parameters (sns. set, rcParams) for enhanced visualization. The process of deciding on a certain number RANDOM_SEED has been set.
Model Training Setup: Starting the PyTorch Lightning environment (pl) is the first step to solving complex problems with data. set the seed to a random value RANDOM_SEED.
The modules that are imported from PyTorch Lightning for logging, callbacks, and metrics (ModelCheckpoint, EarlyStopping, TensorBoardLogger) are the necessities to perform the task.
Model Definition and Training: BERT application in sequence classification is a great option. BERT is a transformer-based model that is pre-trained for the NLP tasks and is widely used.
Outlining a PyTorch Lightning module for the BERT-based model, for example, its forward

pass, training step, validation step, and optimizer setup.

The process of establishing the training loop, data loading with the PyTorch DataLoader, and the

optimizer is the first step in creating a successful deep learning model. g. for Coming up with the next set of words is considered , AdamW), and learning rate scheduler (get_linear_schedule_with_warmup).

The action of setting up the callbacks such as model checkpointing and early stopping is rephrased.

Evaluation: The application of Sklearn in the estimation of metrics like classification report and confusion matrix in order to assess the performance of the trained model.

Visualization: The employment of seaborn and matplotlib for visualization, e.g., creating confusion matrices is a good example.

```
import numpy as np

from tqdm.auto import tqdm

import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader

from transformers import BertTokenizerFast as BertTokenizer, BertModel, AdamW, get_linear_schedule_with_warmup

import pytorch_lightning as pl
from torchmetrics.functional import accuracy, auroc
from pytorch_lightning.callbacks import ModelCheckpoint, EarlyStopping
from pytorch_lightning.loggers import TensorBoardLogger

from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, multilabel_confusion_matrix

import seaborn as sns
from pylab import rcParams
import matplotlib.pyplot as plt
from matplotlib import rc

%matplotlib inline
%config InlineBackend.figure_format='retina'

RANDOM_SEED = 42

sns.set(style='whitegrid', palette='muted', font_scale=1.2)
HAPPY_COLORS_PALETTE = ["#01BEFE", "#FFDD00", "#FF7D00", "#FF006D", "#ADFF02", "#8F00FF"]
sns.set_palette(sns.color_palette(HAPPY_COLORS_PALETTE))
rcParams['figure.figsize'] = 12, 8

pl.seed_everything(RANDOM_SEED)
```

Figure 3: Environment Setup

**Step 2:Dividing The Dataset:**

Train-Validation Split: The dataset df is divided into the training and validation sets using the

train_test_split from scikit-learn. The validation set size is 5% of the original dataset. train_df and val_df are the training and the validation sets, respectively.

Shapes Check: The shapes of train_df, val_df, and test_df are printed to check the sizes of each split. On the contrary, test_df is mentioned but not defined in the given code.

Label Columns: The labels names are gotten from the DataFrame df. It seems that the first two columns are not label columns and the rest of the columns are considered label columns. This is represented by the action of df being sliced. columns. tolist()[2:]. The labels are then summarized to show the distribution of positive samples for each label, and the result is displayed as a horizontal bar chart by using plot(kind="barh").

This part of the code is about the way of dividing the dataset into the training and validation sets, checking their sizes, and illustrating the distribution of positive samples for each label.

```
train_df, val_df = train_test_split(df, test_size=0.05)
train_df.shape, val_df.shape, test_df.shape #test hs only ids and comment_text

LABEL_COLUMNS = df.columns.tolist()[2:]
df[LABEL_COLUMNS].sum().sort_values().plot(kind="barh");
```

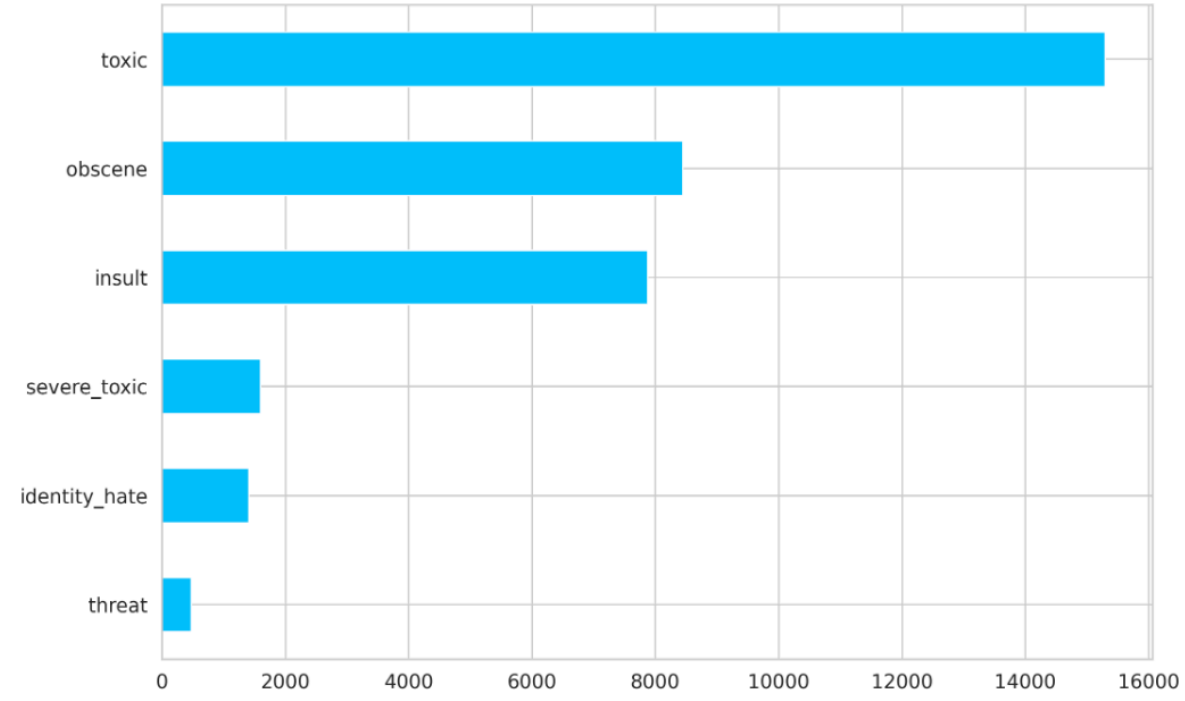


Figure 4: Dividing The Dataset

In the above figure 4, we have displayed the dataset into six toxicity levels.

**Step 3:Visualizing Toxic And Clean Comments:**

Class Imbalance Handling: This technique splits the training set into two parts according to the existence of labels. The train_toxic has rows in which one of the rows has at least one label, which is identified by train_df[LABEL_COLUMNS]. sum(axis=1) > 0. train_clean has rows where none of the labels are there, identified as the condition train_df[LABEL_COLUMNS]. sum(axis=1) == 0.

Subsampling for Balanced Training Set:Subsampling for Balanced Training Set:This synthetically creates a set of clean comments by selecting a part of the whole to make the dataset fair. It finally picks 15,000 samples from the "clean" comments usingtrain_clean. sample(15_000).

Visualization of Class Distribution: It generates a DataFrame with the amount of toxic and clean comments in the training set. It then presents a horizontal bar chart which shows the number of toxic and clean comments.

Concatenation: It is the combination of train_toxic and the sampled subset of train_clean which generates the balanced training dataset. The dataset which is well balanced is kept in train_df.

Shapes Check: It prints the training and validation sets of the balanced dataset (train_df) and the validation set (val_df) to verify their sizes. This code is designed to resolve the problem of class imbalance by subsampling the majority class (clean comments) and generating a balanced training dataset. Besides, it allows the visualization that shows the balancing achieved after subsampling. The balanced training set must be in the right size according to the requirements for the model training.

```
train_toxic = train_df[train_df[LABEL_COLUMNS].sum(axis=1) > 0]
train_clean = train_df[train_df[LABEL_COLUMNS].sum(axis=1) == 0]

pd.DataFrame(dict(
  toxic=[len(train_toxic)],
  clean=[len(train_clean.sample(15_000))]
)).plot(kind='barh');

#Balanced dataset of toxic and non_toxic comments
train_df = pd.concat([
  train_toxic,
  train_clean.sample(15_000)
])

train_df.shape, val_df.shape
```

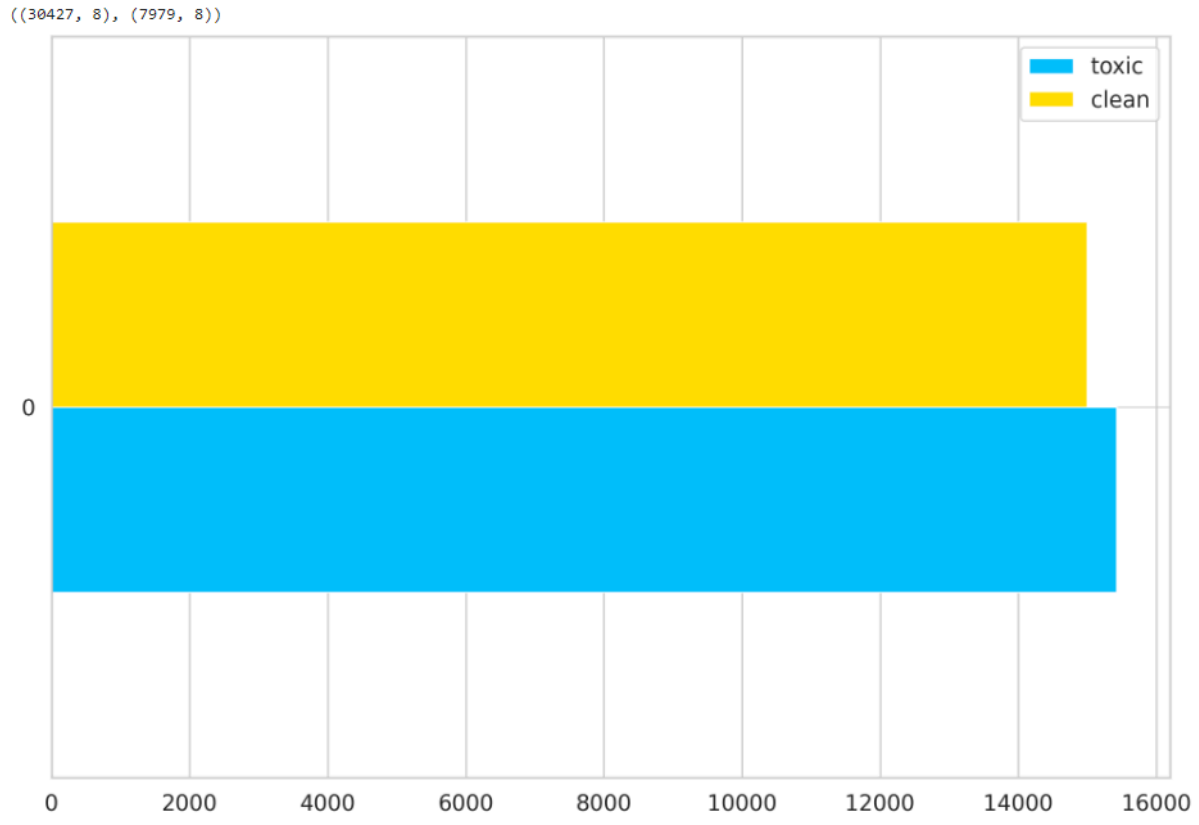


Figure 5:Visualizing Toxic And Clean Comments

In the above figure 5, we have split the dataset in two, that are clean and toxic comments.

**Step 4: Preprocessing The Dataset:**

This code shown in figure 6 represents a dataset class name ToxicCommentsDataset which is used for loading and preprocessing the dataset for the training or inference with BERT-based model. Here's a breakdown of the class:

Initialization (__init__):

It takes several parameters:

data: The Pandas DataFrame that holds the dataset is the given dataset.

tokenizer: BERT tokenizer (BertTokenizer) is used for the process of tokenizing the text data.

max_token_len: The longest input tokens limit (default is 128).

test: Boolean marker that shows whether the dataset is for testing or not the default is False.

Length Method (__len__): It gives the length of the DataFrame, that is the total of the samples in the DataFrame.

Get Item Method (__getitem__): It gets a data sample from the given dataset by taking the data of the index.

The comment text is tokenized using the BERT tokenizer, that is, the comment text is cut up into words which BERT tokenizer can divide, and the words are filled into the max_token_len, if the cutting of the text is less the max_token_len, then the text is padded until it reaches the max_token_len. Then the BERT tokenizer converts the text into input IDs.

If not in the testing mode: From the data frame, the labels for the sample are separated. It gives a dictionary that includes the sample's ID, comment text, input IDs, attention mask, and labels as PyTorch tensors.

If in testing mode: The method does not implement labels, but testing data is hardly in the possession of labels. A dictionary is given which includes the sample's ID, comment text, input IDs, and attention mask as PyTorch tensors.

This dataset type is responsible for the tokenization, padding, and the transformation of the text data to the tensor that can be used as the input for a BERT-based model. It also manages the situations of training and testing, hence, being able to load and preprocess data at its own pace.

```
def __getitem__(self, index: int):
    data_row = self.data.iloc[index]
    _id = data_row['id']
    comment_text = data_row.comment_text

    if not self.test:
        labels = data_row[LABEL_COLUMNS]

    encoding = self.tokenizer.encode_plus(
        comment_text,
        max_length=self.max_token_len,
        padding="max_length",
        truncation=True,
        add_special_tokens=True,
        return_token_type_ids=False,
        return_attention_mask=True,
        return_tensors='pt',
    )

    if not self.test:
        return dict(
        _id = _id,
        comment_text=comment_text,
        input_ids = encoding["input_ids"].flatten(),
        attention_mask=encoding["attention_mask"].flatten(),
        labels=torch.FloatTensor(labels)
    )
    else:
        return dict(
            _id = _id,
            comment_text=comment_text,
            input_ids = encoding["input_ids"].flatten(),
            attention_mask=encoding["attention_mask"].flatten()
        )
```

Figure 6: Preprocessing

**Step 5: BERT Tokenization:**

This code shown in figure 7 launches a BERT tokenizer using the bert-base-cased pre-trained model. Here's what each line does:

BERT_MODEL_NAME = 'bert-base-cased': Sets the variable BERT_MODEL_NAME to 'bert-base-cased', which means the BERT model that will be used will be the pre-trained one. This particular model has the lowercase and uppercase letters in it which means that it is able to differentiate between them.

tokenizer = BertTokenizer. from_pretrained(BERT_MODEL_NAME): This posts the tokenizer in a given BertTokenizer. the from_pretrained() method, that uses the pre-trained tokenizer associated with the BERT model indicated by BERT_MODEL_NAME to load the BERT model.

This tokenizer divides the text into tokens and then converts those tokens into numerical IDs which are linked to the embeddings present in the BERT model's vocabulary.

The from_pretrained() method loads the tokenizer from the Hugging Face model hub, based on the model name, which is provided.

Through these, we have prepared a BERT tokenizer which can tokenize the text data according to the vocabulary and tokenization scheme used by the bert-base-cased model. This tokenizer will be employed in the future for the input of the text data into the input IDs and the attention masks (for BERT model).

```
BERT_MODEL_NAME = 'bert-base-cased'
tokenizer = BertTokenizer.from_pretrained(BERT_MODEL_NAME)
```

Figure 7: BERT Tokenizer

**Step 6:Making Class Instances:**

In this code shown in figure 8, these class instances are made for the training and validation datasets. It creates a ToxicCommentsDataset object for the train_df data (training dataset).

The class ToxicCommentsDataset contains the method which takes the training DataFrame (train_df), the BERT tokenizer (tokenizer), and the maximum token length (MAX_TOKEN_COUNT) and modifies them at the constructor.

MAX_TOKEN_COUNT is considered to be a constant defined at the beginning which stands for the maximum number of tokens that can be entered in one input sequence.

Then the sentence is rephrased: The ToxicCommentsDataset object is created for the validation dataset (val_df).

The constructor of ToxicCommentsDataset receives the validation DataFrame (val_df), the BERT tokenizer (tokenizer) and the maximum token length (MAX_TOKEN_COUNT) as arguments.

These datasets will be employed for the training and the validation since the beginning of the training of the model. The ToxicCommentsDataset class is going to process the tokenization, padding, and transformation of the text data into tensors that can be fed into the BERT-based model. You should confirm that the MAX_TOKEN_COUNT is well-defined.

```
train_dataset = ToxicCommentsDataset(
  train_df,
  tokenizer,
  max_token_len=MAX_TOKEN_COUNT
)

val_dataset = ToxicCommentsDataset(
  val_df,
  tokenizer,
  max_token_len=MAX_TOKEN_COUNT
)
```

Figure 8: Class Instance

**Step 7: Creating Object For Test Data:**

This code shown in figure 9 creates a ToxicCommentsDataset object for the test dataset (test_df). The constructor of ToxicCommentsDataset receives the input test_df, the tokenizer and the maximum token length (MAX_TOKEN_COUNT) from the official test. the test=True means that these datasets are for testing, which means they do not have labels. The dataset class will deal with this by not labeling classes when test=True.

The given dataset will be utilized for testing the trained model and it doesn't have the labels since usually the test data doesn't have the ground truth labels. The ToxicCommentsDataset class will tokenize the text data, padding sequences, and prepare the input tensors for the inference with the BERT-based model. DO NOT let the MAX_TOKEN_COUNT be too high or too low for the BERT model that is being used.

```
test_dataset = ToxicCommentsDataset(
  test_df,
  tokenizer,
  max_token_len=MAX_TOKEN_COUNT,
  test=True
)
```

Figure 9: Test Data Object

**Step 8: Sample Item From The Training Data:**

This code shown in figure 10 prints information about an item sample from the training dataset. Here's what each line does:

sample_item = train_dataset[0]: Gets the first sample from the training dataset (train_dataset) by using Python's indexing ( [0] ). This gives a dictionary which has data about the sample, like its ID, comment text, input IDs, attention mask, and labels (if they are applicable).

print(sample_item. keys()): The method prints the keys of the dictionary sample_item, that represent the different data stored for the sample_item.

print(sample_item["_id"]): It prints the ID of the sample that was stored under the key "_id". print(sample_item["comment_text"]): Type of the comment text of the sample stored under the key "comment_text" is printed.

print(sample_item["input_ids"]): The input IDs are printed out under the key "input_ids". These are the tokenized and numericalized text representations of the comment, which are of the type of BERT input.

print(sample_item["attention_mask"]): It shows the attention mask of the sample collected and stored under the key "attention_mask". This mask puts forward the tokens which should be attended to and those which should be ignored during processing of the BERT model.

print(sample_item["labels"]): Prints the labels of the sample which is saved under the key "labels". This is the case only if the dataset is used for training, since the testing and validation datasets can't contain labels.

```
sample_item = train_dataset[0]
print(sample_item.keys())
print(sample_item["_id"])
print(sample_item["comment_text"])
print(sample_item["input_ids"])
print(sample_item["attention_mask"])
print(sample_item["labels"])
```

Figure 10: Sample Item

**Step 9: Creating The Dataloader:**

This code shown in figure 11 creates the PyTorch DataLoader objects for training, validation and test datasets.

The DataLoader for the training dataset (train_dataset) is initiated. The batch_size parameter is the one which depicts the number of samples per batch to be loaded during the training.

the shuffle=True means that the data will be shuffled at the start of each epoch during the training process so that the model will see different samples in each batch.

Then, the beginning of the process of creating a DataLoader for the validation dataset (val_dataset) is made.

The batch_size parameter defines the number of samples for each batch to be loaded during the validation.

Shuffle= False means that the data will not be shuffled in validation, because the order of samples has no influence on the validation performance.

unshuffle=False signifies that the data will not be shuffled during testing because the order of samples doesn't have any effect on testing.

The DataLoader objects are the key to the efficient iteration of datasets in mini-batches during the training, validation, and testing of the model in the phases of the model. The shuffle parameter makes sure that the training data is shown to the model in a different way in each epoch, but the absence of shuffling in validation and testing guarantees the same evaluation and the same results that can be reproduced.

```
train_dataloader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True)
val_dataloader = DataLoader(val_dataset, batch_size=BATCH_SIZE, shuffle=False)
test_dataloader = DataLoader(test_dataset, batch_size=BATCH_SIZE, shuffle=False)
```

Figure 11: Data Loader

**Step 10: Garbage Collection:**

In this code shown in figure 13, the gc. Python's collect() function is utilized to manually collect garbage. This approach tries to free the memory that is currently assigned to the objects which are not used or needed by the program anymore. In this way, the gc. will be called.

collect(): The Python garbage collector tries to find and delete the memory that is not necessary any more. It travels through the objects in memory and recognizes those that are not any more used by any aspect of the program. As these unreferenced objects are no longer in memory they are then available for future use. Calling gc. automatically the garbage collector runs when the memory is low or the objects are deleted, hence, collect() is not always necessary.

```
import gc
gc.collect()
```

Figure 12: Garbage Collector

**Step 11: Structure Of The Model:**

This code is written to define a PyTorch model class, ToxicCommentTagger, which is supposedly for tagging toxic comments. Here's what each part does:

Initialization (__init__ method): The constructor is the one that initializes the model.

n_classes: Number of classes is the way in which a system can handle multiple different types or characteristics. Thus, the size of the output layer is determined in this way.

n_training_steps: Training steps, which are the variables for adjusting the learning rate during training, usually used in learning rate schedules, are the number of because of this, the learner can decrease the learning rate after the initial phase, and increase it after half the training, and after the training because of this the learner can increase the learning rate.

n_warmup_steps: The number of warm-up steps, which is also used for the adjustment of the learning rate during the training, is the one that I will rephrase.

super(). __init__(): The phrase "calls the constructor of the parent class (nn." is equivalent to "calls the constructor of the parent class (nn. Module).

self. bert = BertModel. from_pretrained(BERT_MODEL_NAME, return_dict=True): The initial loads the pre-loaded BERT model using BertModel. the first component is from_pretrained() and the second component is the self. bert. The return_dict=True argument is a notation that the model should return outputs as a dictionary.

self. classifier = nn. Linear(self. bert. config. hidden_size, n_classes): It is a layer with a linear connection (nn.Linear) added on top of BERT. The input size of this layer is the hidden size of BERT, and the output size is n_classes, referring to the number of classes.

self. n_training_steps and self. the n_warmup_steps is the presenter of the number of training steps.

self. criterion = nn. BCELoss(): This is the step when the binary cross-entropy loss function (BCELoss), which is usually used for binary classification tasks, is initialized.

Forward Pass (forward method):

def forward(self, input_ids, attention_mask, labels=None):def forward(self, input_ids,

attention_mask, labels=None): Puts into action the forward pass of the model.

input_ids and attention_mask are the tensors showing tokenized text and attention masks.

output = self. bert(input_ids, attention_mask=attention_mask): It takes the input and sends it through the BERT model which in turn gives the output, that is a dictionary containing various outputs like the hidden states and the pooler output.

output = self. classifier(output. pooler_output): The pooler output (usually the output of the [CLS] token) is passed through the linear classifier layer to obtain the logits.

output = torch. sigmoid(output): It smoothen the function of the logits to make them probabilities.

The system returns both the loss value and the model output (probabilities) if labels are given, otherwise it just returns the output.

This class describes the structure and the forward pass logic for the model, which can be applied in both the training and the inference phase of the toxic comment classification.

```
class ToxicCommentTagger(nn.Module):

    def __init__(self, n_classes: int, n_training_steps=None, n_warmup_steps=None):
        super().__init__()
        self.bert = BertModel.from_pretrained(BERT_MODEL_NAME, return_dict=True) #load the pretrained bert model
        self.classifier = nn.Linear(self.bert.config.hidden_size, n_classes) # add a linear layer to the bert
        self.n_training_steps = n_training_steps
        self.n_warmup_steps = n_warmup_steps
        self.criterion = nn.BCELoss()

    def forward(self, input_ids, attention_mask, labels=None):
        output = self.bert(input_ids, attention_mask=attention_mask)
        output = self.classifier(output.pooler_output)
        output = torch.sigmoid(output)
        loss = 0
        if labels is not None:
            loss = self.criterion(output, labels)
        return loss, output
```

Figure 13: Forward Pass Logic

**Step 12: Accessing The GPU:**

This code shown in figure 14 is the guide for PyTorch to choose the platform, either CPU or GPU available. It also creates an instance of the ToxicCommentTagger model and puts it on the device that is mentioned. Here's what each line does:

device = torch. device("cuda" if torch. cuda. is_available() else "cpu"): Multiple checks CUDA CUDA (GPU) is available using torch. cuda. is_available().

Under the condition of CUDA's existence, the device is turned on to CUDA ("cuda"),

otherwise, it is turned on to CPU ("cpu").

This line guarantees that the model will be transferred to the GPU for computation if CUDA is on but if it is not, the CPU will be used.

bert_model = ToxicCommentTagger(len(LABEL_COLUMNS)). to(device): This Begins the creation of the ToxicCommentTagger model with the number of classes being the same as the length of LABEL_COLUMNS.

After running these lines, you have a ToxicCommentTagger model that is ready for training or inference or can be used for inference, depending on whether GPU is available or not, and thus you can place it on the appropriate device. You will then be able to go on with the training process of the model on your dataset.

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
bert_model = ToxicCommentTagger(len(LABEL_COLUMNS)).to(device)
```

Figure 14: Accessing GPU

**Step 13: Running The Epochs:**

In this code shown in figure 15, the number of warm-up steps and the total training steps are defined by the number of epochs (N_EPOCHS), the size of the training dataset, and the batch size. Here's what each part does:Here's what each part does:
N_EPOCHS = EPOCHS: In this way, the value of the EPOCHS variable is set to N_EPOCHS. Thus, the names of the variables are always the same throughout the entire project.

steps_per_epoch=len(train_df) // BATCH_SIZE: It gets the number of steps (batches) per epoch by dividing the total number of samples in the training dataset (len(train_df)) by the batch size (BATCH_SIZE). This is the value that shows how many batches are processed in each epoch.
total_training_steps = steps_per_epoch * N_EPOCHS:total_training_steps = steps_per_epoch * N_EPOCHS: The train steps are obtained by multiplying the steps per epoch (steps_per_epoch) by the number of epochs (N_EPOCHS). This value is the sum of the

numbers of optimization steps the model will execute during training.

warmup_steps = total_training_steps // 5:warmup_steps = total_training_steps // 5: The calculated number of warm-up steps are usually an estimated fraction of the total training steps. In this instance, it is set to one-fifth (of the total training steps) of the total training steps.

The warm-up steps that are used in some learning rate schedules are designed to increase the learning rate at the start of the training in a gradual way, thus, the learning process becomes more stable and the possibility of diverging is prevented.

The warmup_steps and total_training_steps are the two values which are significant in setting up the learning rate schedules, especially in training the big deep learning models such as BERT. They are used to the control of the learning rate during training to the improvement of convergence and performance.

```
N_EPOCHS = EPOCHS


steps_per_epoch=len(train_df) // BATCH_SIZE
total_training_steps = steps_per_epoch * N_EPOCHS
warmup_steps = total_training_steps // 5
warmup_steps, total_training_steps
```

Figure 15: Running Epochs

**Step 14: Optimizer And Scheduler:**

In this code shown in figure 16, we are initializing the optimizer for the training of the model. AdamW is a version of the Adam optimizer that applies weight decay (L2 regularization) directly to the parameters during the optimization, which hinders the overfitting.

optimizer: The optimizer object is the one that changes the learning rate.

num_warmup_steps: The number of warm-up steps before the usage of the learning rate is attained.

num_training_steps: The cumulative number of training steps (batches) for every epoch.

Thus, the scheduler boosts the learning rate linearly from 0 to the suggested learning rate (lr=2e-5) during the warm-up period and then it gets decreased linearly to 0 over the remaining steps.

The optimizer and scheduler are the ones who are the partners in adjusting the model

parameters during the training phase. The optimizer replaces the parameters based on the gradients that were computed during the backpropagation, on the other hand, the scheduler modifies the learning rate according to the given schedule, thus the training process is under control and it is improved.

```
optimizer = AdamW(model.parameters(), lr=2e-5)

scheduler = get_linear_schedule_with_warmup(
    optimizer,
    num_warmup_steps=warmup_steps,
    num_training_steps=total_training_steps
)
```

Figure 16: Optimizer and Scheduler

**Step 15:Creating A Training Function:**

This code shown in figure 17 creates the training function for the model. Here's a breakdown of what each part of the function does:

def train(): Creates a function called train() for the training of the model.

model. train(): The format of the model is set to the training mode, which allows the gradients to be computed and the parameters to be updated during the training.

It starts by creating three variables total_loss, total_accuracy, and avg_loss that will be used to store the loss and accuracy data during the training process.

The step starts with creating an empty total_preds list to contain the model predictions.

Batch Iteration: Goes over the batches of the train_dataloader one by one.

For each batch: The mentioned batch tensors are moved to the specified device.

The sentence is explained using the tools of math which makes it easier to understand. zero_grad().

Steps forward through the model to calculate the loss and the model outputs.

Calculates the total_loss by summing the loss item to the total_loss.

Adjust model parameters using the optimizer for the updates. It does the step() and schedules the learning rate with a scheduler. step().

The method separates the model outputs from the computation graph, moves them to CPU, and converts them to NumPy arrays before attaching them to the total_preds.

Loss Calculation: Calculates the mean of the training loss.

Model Predictions: Then, the predictions of all the batches are linked along the first axis to get total_preds that are in the shape (number of samples, number of classes).

Returns: Returns the average training loss for the epoch (avg_loss) and the total predictions (total_preds) which are concatenated together.

```
def train():

  model.train()

  total_loss, total_accuracy = 0, 0
  avg_loss = 0

  total_preds=[]

  for step,batch in enumerate(train_dataloader):


    if step % 50 == 0 and not step == 0:
      print('  Batch {:>5,}  of  {:>5,}.'.format(step, len(train_dataloader)))

    input_ids = batch["input_ids"].to(device)
    attention_mask = batch["attention_mask"].to(device)
    labels = batch["labels"].to(device)

    model.zero_grad()
    loss, outputs = model(input_ids, attention_mask, labels)

    total_loss = total_loss + loss.item()

    loss.backward()

    torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)

    optimizer.step()
    scheduler.step()

    outputs=outputs.detach().cpu().numpy()

    total_preds.append(outputs)

  avg_loss = total_loss / len(train_dataloader)
  print(f"{step}: {avg_loss}")

  total_preds  = np.concatenate(total_preds, axis=0)

  return avg_loss, total_preds
```

Figure 17: Training Function

**Step 16: Creating An Evaluation Function**

This code shown in figure 18 contains an evaluation function which evaluates the performance of the model on the validation dataset. Here's a breakdown of what each part of the function does:Here's a breakdown of what each part of the function does:

def evaluate(): The function evaluate() lets the model be evaluated.

Model Evaluation:

model. eval(): Transforms the model into evaluation mode, turning off dropout layers and forcing the batch normalization layers to use the correct population statistics.

The introduction of the variables total_loss, total_accuracy and avg_loss enables the tracking of the loss and accuracy during evaluation.

The method first creates two empty lists, total_preds and total_labels, which will store model predictions and true labels.

Batch Iteration: Computes over batches from the val_dataloader.

For each batch: The batch tensors are transferred to the device (GPU) of the user's choice. Unmasks the autograd feature to put it on the brakes.

Takes a backward pass through the model to compute the loss and model outputs.

It adds the loss item to the total_loss in order to compute the total loss.

Runs the detach function to free the model outputs from the computation graph, transfers them to CPU and converts them to NumPy arrays before appending to total_preds. Likewise, correct labels are cut and connected to total_labels.

Loss Calculation: The function calculates the mean validation loss for the epoch by dividing total_loss by the number of batches.

Model Predictions: By concatenating the predictions from all batches on the first axis to get total_preds in the form (number of samples, number of classes).

Links true labels from all batches along the first axis to get total_labels which are in the same shape as the labels of all batches.

Metrics Calculation: Computes the ROC AUC score for each class by utilizing roc_auc_score from scikit-learn. The meta prints the ROC AUC scores for each class.

Returns: It outputs the average validation loss of the epoch (avg_loss), the total predictions (total_preds), and the total true labels (total_labels).

This function is assessing the model on the validation data and returning the validation loss, model predictions and true labels. Besides, in addition to the ROC AUC score for each class, it also prints the result of the classification for each class, which gives us an idea of the model performance across different classes.

```
total_preds = []
total_labels = []

for step,batch in enumerate(val_dataloader):

  if step % 50 == 0 and not step == 0:

    print('  Batch {:>5,}  of  {:>5,}.'.format(step, len(val_dataloader)))

  input_ids = batch["input_ids"].to(device)
  attention_mask = batch["attention_mask"].to(device)
  labels = batch["labels"].to(device)

  with torch.no_grad():

    loss, outputs = model(input_ids, attention_mask, labels)

    total_loss = total_loss + loss.item()

    outputs = outputs.detach().cpu().numpy()
    labels = labels.detach().cpu().numpy()
    total_preds.append(outputs)
    total_labels.append(labels)


avg_loss = total_loss / len(val_dataloader)
print(f"{step}: {avg_loss}")

total_preds  = np.concatenate(total_preds, axis=0)
total_labels = np.concatenate(total_labels, axis=0)
true = np.array(total_labels)
pred = np.array(total_preds>0.5)

for i, name in enumerate(LABEL_COLUMNS):
    try:
        print(f"{name} roc_auc {roc_auc_score(true[:, i], pred[:, i])}")
    except Exception as e:
      print(e)
      pass
print(f"Evaluate loss {total_loss / len(val_dataloader)}")
return avg_loss, total_preds, total_labels
```

Figure 18: Evaluation Function

**Step 17:Training The Model:**

This code shown in figure 19 keeps training the model for a certain number of epochs and after each epoch it tests its performance on the validation dataset. Here's a breakdown of what the code does:

%%time: This is the Jupyter Notebook command that measures the execution time of the cell.

Initialization: best_valid_loss = float('inf'): The early validation loss is set to infinity.

Epoch Loop: Reiterates for each epoch from 1 to the particular number of epochs (EPOCHS). The print number of the current epoch.

Training: It runs the train() command for training for the current epoch and gathers the loss.

Evaluation: The evaluate() function is then used to evaluate the model on the validation dataset. If the current validation loss is less than the best validation loss so far, it updates the best validation loss and saves the model's weights to a file named 'saved_weights'. pt'.

Loss Logging: The earlier mentions the appending of the training and validation losses of the current epoch to their respective lists (train_losses and valid_losses).

Prints: The outputs the training and validation losses for the current epoch. This loop continues for the specified number of epochs, during training and evaluation of the model and at the same time, keeping the track of the best validation loss.

```
%%time

best_valid_loss = float('inf')

train_losses=[]
valid_losses=[]
EPOCHS = 2

for epoch in range(EPOCHS):

    print('\n Epoch {:} / {:}'.format(epoch + 1, EPOCHS))

    train_loss, _ = train()

    valid_loss, _, _ = evaluate()

    if valid_loss < best_valid_loss:
        best_valid_loss = valid_loss
        torch.save(model.state_dict(), 'saved_weights.pt')

    train_losses.append(train_loss)
    valid_losses.append(valid_loss)

    print(f'\nTraining Loss: {train_loss:.3f}')
    print(f'Validation Loss: {valid_loss:.3f}')
```

Figure 19: Model Training

**Step 18: Creating The Testing Function**

This code shown in figure 20 has a testing function that tests the trained model on the test dataset. Below is a breakdown of what each part of the function does:

def test(): It is the code that specifies the function named test() for testing the model.

Model Testing:

model. eval(): Sets the model to evaluation mode, which is a mode without the dropout layers and makes sure that the batch normalization layers are using the right statistics of the population. The variable total_loss is set to be the memory element for the loss during testing.

Begins by making empty lists total_preds and _ids which will be used as a storage for model predictions and the IDs.

Batch Iteration: Cycles through the batches from the test_dataloader.

For each batch: Doubles the batch tensors and moves them to the specified device (GPU). Automatically turns off gradients thus disabling the gradient computation. The forward pass through the model is executed to obtain the loss and model outputs. Computes the total loss by adding the loss item to the existing total_loss. The model outputs and the IDs that come with them are added to total_preds and _ids, respectively.

Loss Calculation: It calculates the mean test loss for the whole test dataset by dividing the total_loss by the number of batches.

Model Predictions: Allows the predictions from all batches to be concatenated along the first axis to get the total_preds in the shape of (number of samples, number of classes).

Joins the related IDs from all batches on the first axis and gets _ids as a NumPy array.

Results: Constructs a dictionary containing the terms _ids and total_preds which gives the IDs and the predictions respectively.

Returns: Returns the average test loss for the complete test set (avg_loss), concatenated predictions (total_preds), and the dictionary of results (results). This function is the one that evaluates the model that has been trained on the test dataset and gives the test loss, predictions of the test data and then it returns the test IDs. It is a good tool to evaluate the model's performance on the unseen data once the training is done.

```
def test():

  print("\nTesting...")
  model.eval()

  total_loss, total_accuracy = 0, 0

  total_preds = []
  _ids = []

  for step,batch in enumerate(test_dataloader):

    if step % 50 == 0 and not step == 0:

      print('  Batch {:>5,}  of  {:>5,}.'.format(step, len(test_dataloader)))

    _id = batch["_id"]
    input_ids = batch["input_ids"].to(device)
    attention_mask = batch["attention_mask"].to(device)

    with torch.no_grad():

      loss, outputs = model(input_ids, attention_mask)

      total_loss = total_loss + loss

      outputs = outputs
      _ids.append(_id)
      total_preds.append(outputs)

  avg_loss = total_loss / len(test_dataloader)
  _ids = np.concatenate(_ids, axis=0)
  total_preds  = torch.cat(total_preds, axis=0)
  results = dict(id=_ids,
      predictions = total_preds
      )

  return avg_loss, total_preds, results
```

Figure 20: Testing Function

**Step 19: Making The ROC Curve:**

In this code shown in figure 21, the given evaluate_roc function is aimed at the assessment of the model's performance by the Receiver Operating Characteristic (ROC) curve and the metrics that are related to it. Below is a breakdown of what each part of the function does:

Function Definition:

def evaluate_roc(probs, y_true): The function evaluate_roc is defined that gets predicted probabilities(probs) and true labels(y_true) as its parameters.

ROC Curve and Metrics:

ROC Curve: Computes the False Positive Rate (FPR), True Positive Rate (TPR) and the thresholds using the roc_curve from scikit-learn. The function computes the Area Under the

Curve (AUC) of the ROC curve by using the function from scikit-learn. It plots the ROC curve with the AUC value indicated on the plot.

Accuracy: Calculates the accuracy of the model by classifying the predicted probabilities into 0 when the model is wrong and 1 when the model is right.

Printing Metrics: Outprints the ACC and the AUC on the test set.

Plotting: The issue is introduced then the ROC curve is set up the plot for it. The graph shows the ROC curve, adding the AUC on the graph. Includes a legend and the labels of what is in the plot.

Returns: No explicit return value. The procedure prints the AUC and the accuracy and shows the ROC curve plot. This function gives a thorough review of the model's ability to predict using the ROC curve and the related metrics. It is a good tool for checking the quality of the model and for showing its performance in different classification thresholds and the discrimination between positive and negative samples.

```
from sklearn.metrics import accuracy_score, roc_curve, auc

def evaluate_roc(probs, y_true):
    """
    - Print AUC and accuracy on the test set
    - Plot ROC
    @params    probs (np.array): an array of predicted probabilities with shape (len(y_true), 2)
    @params    y_true (np.array): an array of the true values with shape (len(y_true),)
    """
    preds = probs[:, 1]
    fpr, tpr, threshold = roc_curve(y_true, preds)
    roc_auc = auc(fpr, tpr)
    print(f'AUC: {roc_auc:.4f}')

    y_pred = np.where(preds >= 0.5, 1, 0)
    accuracy = accuracy_score(y_true, y_pred)
    print(f'Accuracy: {accuracy*100:.2f}%')

    plt.title('Receiver Operating Characteristic')
    plt.plot(fpr, tpr, 'b', label = 'AUC = %0.2f' % roc_auc)
    plt.legend(loc = 'lower right')
    plt.plot([0, 1], [0, 1],'r--')
    plt.xlim([0, 1])
    plt.ylim([0, 1])
    plt.ylabel('True Positive Rate')
    plt.xlabel('False Positive Rate')
    plt.show()
```

Figure 21: ROC Curve

**Step 20: Calculating The Loss:**

In this code shown in figure 22, we have made the evaluate() function call, which checks the model on the validation dataset and gives the average validation loss, model predictions

(total_preds), and true labels (total_labels). We shall now start breaking the model and getting the values.

This indicates that we have tested the model on the validation data and got the average validation loss, model predictions (total_preds) and true labels (total_labels). With this we can calculate performance metrics and measure the model's performance.

```
avg_loss, total_preds, total_labels = evaluate()
```

```
Evaluating...
  Batch    50  of    250.
  Batch   100  of    250.
  Batch   150  of    250.
  Batch   200  of    250.
249: 0.0570118059553206
toxic roc_auc 0.9372597890691963
severe_toxic roc_auc 0.7172941195087986
obscene roc_auc 0.9272100978089316
threat roc_auc 0.6534695532938064
insult roc_auc 0.8900963171343961
identity_hate roc_auc 0.7246566144464155
Evaluate loss 0.0570118059553206
```

Figure 22: Average Loss

**Step 5: Calculation THe Model Accuracy:**

This code shown in figure 23 goes over every column of labels and checks the model's performance with the evaluate_roc function for each label. Here's a breakdown of what each part of the code does:

Iteration over Label Columns: for i, name in enumerate(LABEL_COLUMNS):The sentence iterates over the label columns one by one where i is the index number and name is the name of the label column.

Evaluation for each Label: print(f"label: {name}"): Shows the name of the label.

Total_labels[:,i]): The evaluate_roc function is called to assess the model's performance for the current label.

total_preds[:, i] > 0. 5 is used to set the limit of threshold probabilities for the current label at 0. 5 which will assist in determining the binary outcome.

total_labels[:, it] supplies the accurate labels for the restricted label column.

evaluate_roc Function: This function calculates the AUC and the accuracy for each label.

Through the process of checking each label column and reviewing the model's performance for every label, you can determine the model's success for each individual label in the multi-label classification problem. This enables us to find the model's aptitude and inadequacy for each category.

```
for i, name in enumerate(LABEL_COLUMNS):
    print(f"label: {name}")
    evaluate_roc(total_preds[:,i]>0.5, total_labels[:,i])
```

Figure 23: Model Accuracy

## 3.5 Key Challenges

**Complexity of Toxicity:**

There are plenty of forms for toxic remarks such as hate speech, threatening, sarcasm and others. Developing a model which is capable of understanding and classifying these subtleties correctly is difficult.

**Model Overfitting:**

They experience a problem of overfitting, especially in the case where data are restricted or when the model is too complicated for the objective. These include the regularization methods of dropout and early stopping to avoid overfitting.

**Hyperparameter Tuning:**

For LSTM or BERT models, a very careful selection of the architecture, number of layers, learning rate, batch size etc. is needed during the trial and error process.

**Interpretability:**

Simpler models are preferred because they are not as complex or less interpretable than the LSTM or hybrid models. Analyzing how these models decide whether substances are harmful may be challenging.

# CHAPTER 4 TESTING

## 4.1 Testing Strategy

In this code shown in figure 24 ,we've run the test() function, which evaluates the trained model on the test dataset and generates the average test loss, model predictions (total_test_preds), and a dictionary containing IDs and predictions (sub). Now we should start with the model evaluation on the test dataset and their corresponding values will be listed below.

We have evaluated the model on the test dataset and achieved the average test loss, model predictions (total_test_preds), and a dictionary that contains the IDs and predictions (sub). With this, you can look at the model's result on the test data or, even use its predictions for any other tasks.

```
avg_test_loss, total_test_preds, sub = test()
```

```
Testing...
  Batch    50  of  4,787.
  Batch   100  of  4,787.
  Batch   150  of  4,787.
  Batch   200  of  4,787.
  Batch   250  of  4,787.
  Batch   300  of  4,787.
  Batch   350  of  4,787.
  Batch   400  of  4,787.
  Batch   450  of  4,787.
  Batch   500  of  4,787.
  Batch   550  of  4,787.
  Batch   600  of  4,787.
  Batch   650  of  4,787.
  Batch   700  of  4,787.
  Batch   750  of  4,787.
  Batch   800  of  4,787.
  Batch   850  of  4,787.
  Batch   900  of  4,787.
  Batch   950  of  4,787.
  Batch 1,000  of  4,787.
  Batch 1,050  of  4,787.
```

Figure 24: Test Loss

Then as explained in the code shown in figure 25, we made a DataFrame called D and added a column 'id' to it. The values in this column are from the dictionary sub, which probably has the IDs and accordingly the predictions generated by the model on the test dataset.

Here's a breakdown of the steps:

DataFrame Creation: We've generated a DataFrame named D that is not filled. The

DataFrame that will be created will be used to store the IDs obtained from the sub dictionary.

Column Addition: We've placed a column named 'id' in the DataFrame D with the help of D['id'] = sub['id']. This column contains the IDs which are the result of the sub dictionary extracted.

Through the process of making this DataFrame and adding the IDs to it, you will be able to easily manipulate and analyze the test data predictions as well as, if necessary, further manipulate and analyze the test data predictions.

```
D = pd.DataFrame()
D['id'] = sub['id']
D
```

| | id |
|---|---|
| 0 | 00001cee341fdb12 |
| 1 | 0000247867823ef7 |
| 2 | 00013b17ad220c46 |
| 3 | 00017563c3f7919a |
| 4 | 00017695ad8997eb |
| ... | ... |
| 153159 | fffcd0960ee309b5 |
| 153160 | fffd7a9a6eb32c16 |
| 153161 | fffda9e8d6fafa9e |
| 153162 | fffe8f1340a79fc2 |
| 153163 | ffffce3fb183ee80 |

153164 rows × 1 columns

Figure 25: Dataframe

## 4.2 Test Cases and Outcomes

As explained in the code shown in figure 26, we are making the predictions from the sub dictionary to the DataFrame D through colonization of the fields in the DataFrame D.

Assigning Predictions:

sub['predictions']. cpu(). numpy(): This extracts the predictions from the 'predictions' of the sub dictionary. The prediction seems to be stored as tensors on the CPU, and . cpu(). numpy() is the function that is used to get the arrays into NumPy arrays.

D[LABEL_COLUMNS]: This assigns the predictions to the column names specified in LABEL_COLUMNS in the DataFrame D. LABEL_COLUMNS most probably is the list of labels in the dataset and thus, by doing D[LABEL_COLUMNS], we are assigning the

predictions to the specific labels specified in this list in the DataFrame.

Through the placement of the predictions in the DataFrame columns that correspond to the label columns, we are on the way to organizing the predictions in a logical order, hence it will be easier to analyze and compare them with the ground truth labels or perform further processing if needed.

```
D[LABEL_COLUMNS] = (sub['predictions'].cpu().numpy())
D
```

| | id | toxic | severe_toxic | obscene | threat | insult | identity_hate |
|---|---|---|---|---|---|---|---|
| 0 | 00001cee341fdb12 | 0.986540 | 0.434799 | 0.967243 | 0.130517 | 0.950799 | 0.688200 |
| 1 | 0000247867823ef7 | 0.005331 | 0.004695 | 0.006866 | 0.005086 | 0.005364 | 0.004337 |
| 2 | 00013b17ad220c46 | 0.007648 | 0.003909 | 0.006691 | 0.004546 | 0.004666 | 0.004366 |
| 3 | 00017563c3f7919a | 0.004259 | 0.005359 | 0.005865 | 0.006806 | 0.005096 | 0.005096 |
| 4 | 00017695ad8997eb | 0.008092 | 0.003817 | 0.005989 | 0.004750 | 0.005252 | 0.003937 |
| ... | ... | ... | ... | ... | ... | ... | ... |
| 153159 | fffcd0960ee309b5 | 0.867226 | 0.005281 | 0.478026 | 0.002010 | 0.086353 | 0.004920 |
| 153160 | fffd7a9a6eb32c16 | 0.021681 | 0.002440 | 0.007316 | 0.003922 | 0.006635 | 0.003654 |
| 153161 | fffda9e8d6fafa9e | 0.006966 | 0.003731 | 0.007416 | 0.004339 | 0.005165 | 0.004004 |
| 153162 | fffe8f1340a79fc2 | 0.018777 | 0.002714 | 0.006707 | 0.003892 | 0.007516 | 0.009365 |
| 153163 | ffffce3fb183ee80 | 0.977858 | 0.021897 | 0.813599 | 0.007217 | 0.522236 | 0.003769 |

153164 rows × 7 columns

Figure 26: Predictions

# CHAPTER 5 RESULTS AND EVALUATION

## 4.1. Results

We have made the ROC curve for each of the six levels of comments toxicity and calculated AUC and accuracy for each level.

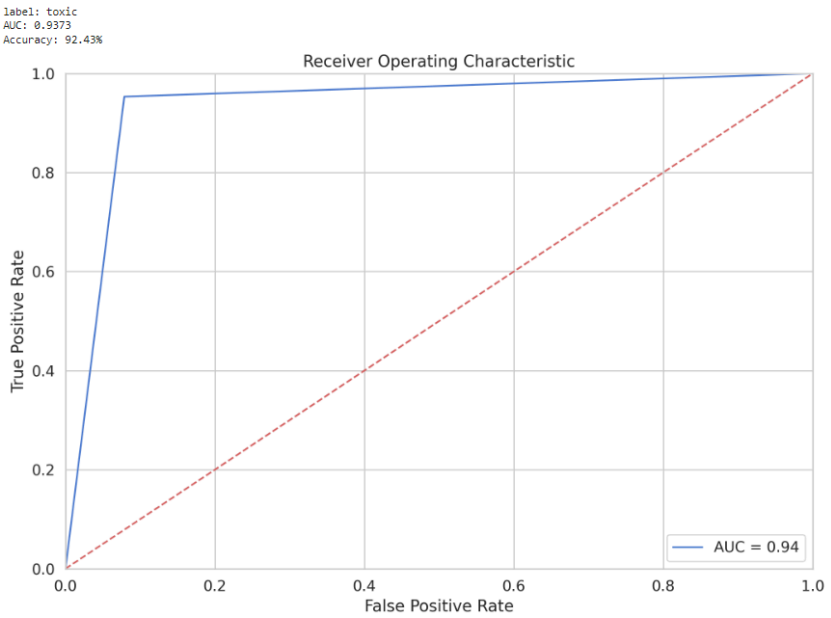


Figure 27: ROC Curve For label ‘toxic’

In the above figure 27, we have plotted a ROC curve between True Positive Rate and False Positive Rate for the label ‘toxic’ and displayed the AUC and Accuracy of this label.

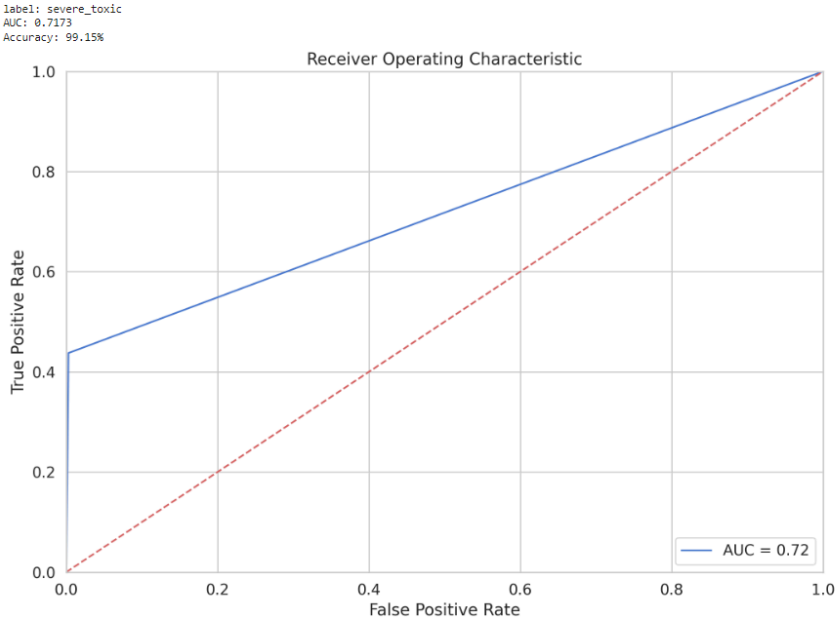


Figure 28: ROC Curve For label severe_toxic

In the above figure 28, we have plotted a ROC curve between True Positive Rate and False Positive Rate for the label 'severe_toxic' and displayed the AUC and Accuracy of this label.

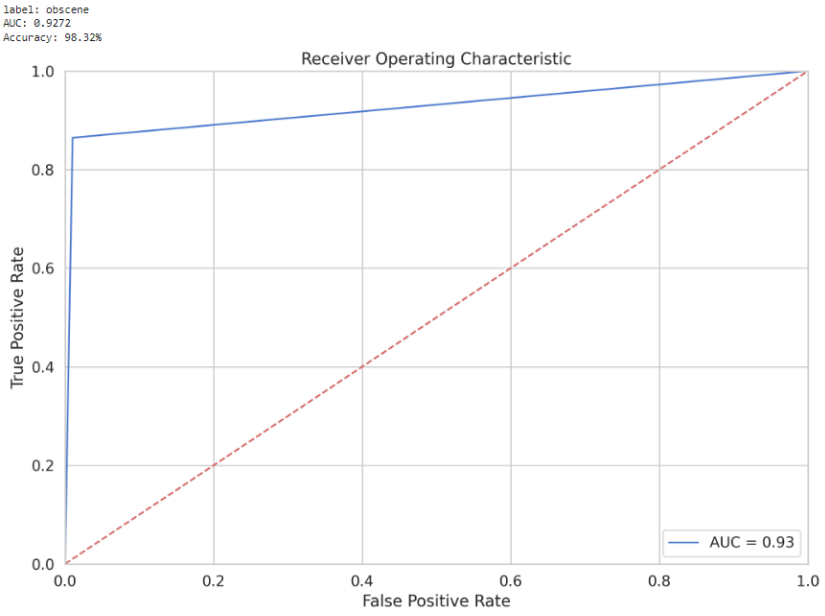


Figure 29: ROC Curve For label 'obscene'

In the above figure 29, we have plotted a ROC curve between True Positive Rate and False Positive Rate for the label 'obscene' and displayed the AUC and Accuracy of this label.

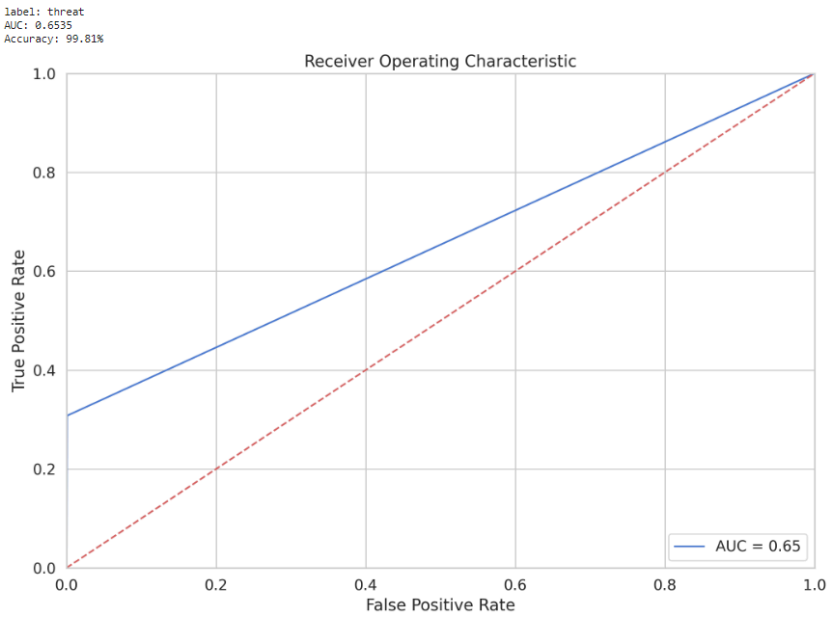


Figure30: ROC Curve For label ‘threat’

In the above figure 30, we have plotted a ROC curve between True Positive Rate and False Positive Rate for the label ‘threat’ and displayed the AUC and Accuracy of this label.

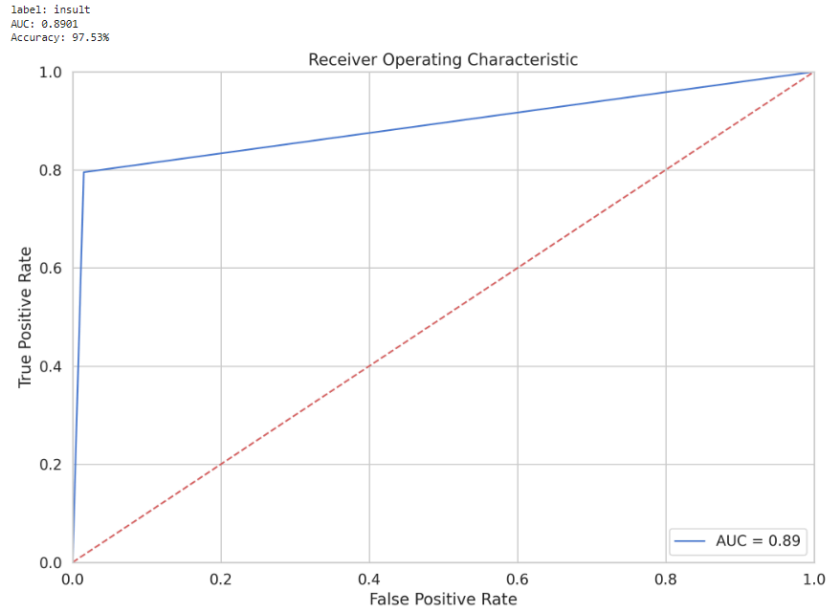


Figure 31: ROC Curve For label ‘insult’

In the above figure 31, we have plotted a ROC curve between True Positive Rate and False Positive Rate for the label ‘insult’ and displayed the AUC and Accuracy of this label.

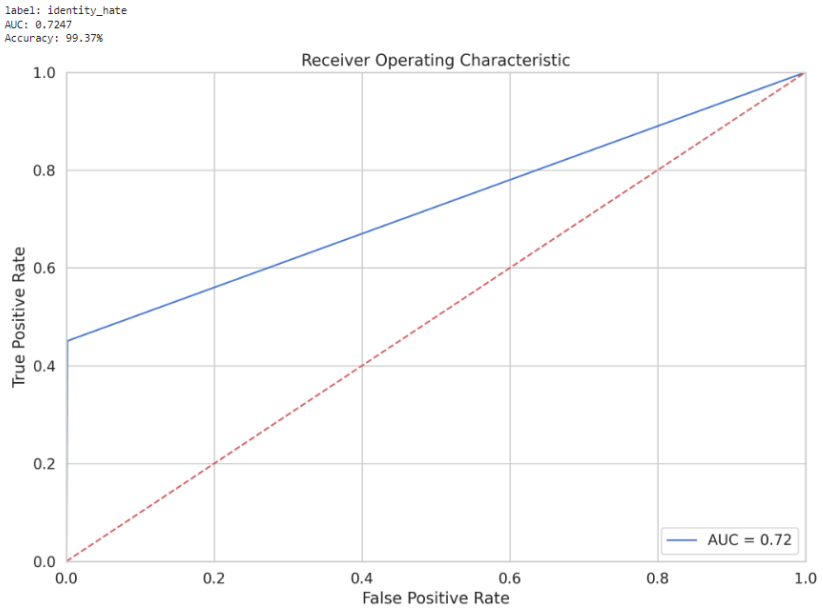


Figure 32: ROC Curve For label 'identity_hate'

In the above figure 32, we have plotted a ROC curve between True Positive Rate and False Positive Rate for the label 'identity_hate' and displayed the AUC and Accuracy of this label.

## 4.2. Comparison with Existing Solutions

The best existing solution in the literature review was in **Multilabel Toxic Comment Detection and Classification (2021)** where LSTM with GloVe showed an accuracy of 96.66%. Our models prove to be better in detecting toxic comments than the previous studies.

# CHAPTER 6 CONCLUSION AND FUTURE SCOPE

## 6.1. Conclusion

We worked with a complex deep learning model in this project. This implementation took place using a Natural Language Processing use-case. We also came to understand some efficient ways of cleaning messy textual data during the different projects preprocessing activities as well as in feature engineering.

We learned how different deep-learning models operate including LSTM and BERT models. We got the understanding of word embedding and their merits if compared to untrained word embedding. We also learned about different libraries that enabled us to tune perfectly the hyper- parameters for both models as well as optimize the results.

## 6.2. Future Scope

**1. Enhanced Model Architectures:**

Transformer-based Models: Use advanced architectures such as BERT, GPT or RoBERTa to better understand toxic discourse contexts.
Ensemble Techniques: Use various models in combination in order to optimize the performance in a better way by exploiting the advantages of each model.

**2. Multilingual Adaptability:**

Language Expansion: Expand model to take into account larger number of languages, accents and toxicity detection for increased inclusivity.

**3. Contextual Understanding and Nuanced Detection:**

Fine-grained Classification: Develop tools which can recognize slight manifestations and various types of poisonousness like sarcasm, subtle hate speech, and others.
Contextual Analysis: Consider including data about the current situation in the room or any other element, which may help explain the true intention of the comment rather than its surface meaning.

**4. Continuous Learning and Adaptation:**

Active Learning Strategies: Use methods of incremental improvement on man-related models.
Dynamic Dataset Expansion: Update/enrich/refresh datasets on a regular basis as they become obsolete in light of altering patterns of online languages and behavior.

**5. Real-time Monitoring and Intervention:**

Live Moderation Systems: Integrate models for real-time platforms, filtering and removal of toxic text/content as it appears.
Prompt Interventions: Respond with severity design prompt/response & intervention systems that detect severe toxicity.

**6. Ethical Considerations and Bias Mitigation:**

Bias Detection and Mitigation: Apply techniques for finding and eradicating any bias, which may be embedded in data, in order to generate unbiased and reliable forecasts.
Ethical Guidelines: Develop guidelines for use of the AI to monitor online contents with a view of not censoring too much nor biasing the same.

**7. Domain-specific Applications:**

Industry-specific Solutions: Overcome industry-specific challenges through building niche customized models for niches such as social media, game, education, etc.

**8. Regulatory Compliance and Standards:**

Compliance Measures: Ensure that you don't violate any of the statutes as stated by matching the models with statutes on privacy and content censorship. More elaborate, flexible and culturally sensible model should be created to cope with the complex nature of internet-based dialect without infringement and transparency. Development of more effective toxicity detection systems, incorporating advanced AI/NLP technologies will also persist.

# REFERENCE

**[1]** Ahmad Alsharef, Karan Aggarwal, Sonia, Deepika Koundal, Hashem Alyami, and Darine Ameyed, "An Automated Toxicity Classification on Social Media Using LSTM and Word Embedding," *Hindawi Computational Intelligence and Neuroscience*, vol. 2023, article ID 9850820.

**[2]** Neha Reddy, Neha Ram, Pallavi, Dr. K. Kranthi Kumar, and Dr. K Sree Rama Murthy, "Toxic Comments Classification," *International Journal for Research in Applied Science & Engineering Technology (IJRASET)*, ISSN: 2321-9653, vol. 10, no. VI, June 2022.

**[3]** Ghinaa Zain Nabiilaha, Simeon Yuda Prasetyoa, Zahra Nabila Izdihara, and Abba Suganda Girsanga, "BERT base model for toxic comment analysis on Indonesian social media," in *7th International Conference on Computer Science and Computational Intelligence*, 2022.

**[4]** Sharayu Lokhande, Ajay Kumar, Kumar Shivam, Naresh Kumar, and Rahul Malhan, "Multilabel Toxic Comment Detection and Classification," *International Journal of Engineering Research & Technology (IJERT)*, ISSN: 2278-0181, vol. 10, issue 05, May 2021.

**[5]** Sara Zaheri, Jeff Leath, and David Stroud, "Toxic Comment Classification," *Southern Methodist University*, 6425 Boaz Lane, Dallas, Texas 75205, 2020.

**[6]** 59Chady Ben Hamida, Victoria Ge, and Nolan Miranda, "Toxic Comment Classification and Unintended Bias," June 12, 2019.

**[7]** Spiros V. Georgakopoulos, Thessally Lamia, Sotiris K. Tasoulis, Aristidis G. Vrahatis, and Vassilis P. Plagianakos, "Convolutional Neural Networks for Toxic Comment Classification," *arXiv:1802.09957v1 [cs.CL]*, 27 Feb 2018.

**[8]** J. Zhang, Y. Li, J. Tian, and T. Li, "LSTM-CNN Hybrid Model for Text Classification," in *2018 IEEE 3rd Advanced Information Technology, Electronic and Automation Control Conference (IAEAC)*, Chongqing, 2018, pp. 1675–1680, doi: 10.1109/IAEAC.2018.8577620.

**[9]** cjadams, Jeffrey Sorensen, Julia Elliott, Lucas Dixon, Mark McDonald, nithum, and Will Cukierski, "Toxic Comment Classification Challenge," *Kaggle*, 2017.

**[10]** Jang, Beakcheol, Myeonghwi Kim, Gaspard Harerimana, Sang-ug Kang, and Jong

Wook Kim, "Bi-LSTM Model to Increase Accuracy in Text Classification: Combining Word2vec CNN and Attention Mechanism," *Applied Sciences*, vol. 10, no. 17, p. 5841, 2020.

**[11]** J. F. T. P. A. R. A. J. K. Marilyn Walker, "A Corpus for Research on Deliberation and Debate," in *Proceedings of the Eighth International Conference on Language Resources and Evaluation (LREC'12)*, Istanbul, 2012.

**[12]** P. S. H. T. S. R. P. S. S. K. M. P. G. A. M. Binny Mathew, "Thou shalt not hate: Countering Online Hate Speech," in *ICWSM 2019*, 2019.

**[13]** J. T. A. T. Y. M. Y. C. Chikashi Nobata, "Abusive Language Detection in Online User Content," in *WWW '16: Proceedings of the 25th International Conference on World Wide Web*, 2016.

**[14]** S. K. V. T. M. IKONOMAKIS, "Text Classification Using Machine Learning Techniques," *WSEAS TRANSACTIONS on COMPUTERS*, vol. 4, no. 8, 2005.

**[15]** M.-L. N. Huy Nguyen, "A Deep Neural Architecture for Sentence-level Sentiment Classification in Twitter Social Networking," in *PACLING Conference*, 2017.

**[16]** J. W. K. R. L. X. Z. Liang-Chih Yu, "Refining Word Embeddings for Sentiment Analysis," in *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, 2017.

**[17]** N. T. L. D. Ellery Wulczyn, "Ex Machina: Personal Attacks Seen at Scale," in *WWW '17: Proceedings of the 26th International Conference on World Wide Web*, 2017.

**[18]** S. K. B. Z. R. P. Hossein Hosseini, "Deceiving Google's Perspective API Built for Detecting Toxic Comments," in *Computers and Society (cs.CY); Social and Information Networks (cs.SI)*, 2017.

**[19 ]**Y. Kim, "Convolutional Neural Networks for Sentence Classification," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2014.

**[20]** Y. Chen, Y. Zhou, S. Zhu, and H. Xu, "Detecting Offensive Language in Social Media to Protect Adolescent Online Safety," in *2012 International Conference on Privacy, Security, Risk and Trust and 2012 International Conference on Social Computing*, Amsterdam, Netherlands, 2012.

**[21]** N. Chakrabarty, "A Machine Learning Approach to Comment," in *INTERNATIONAL*

*CONFERENCE ON COMPUTATIONAL INTELLIGENCE IN PATTERN RECOGNITION (CIPR 2019)*, 2019.

**[22]** H. K. J. H. G. S. Rahul, "Classification of Online Toxic Comments Using Machine Learning Algorithms," in *Proceedings of the International Conference on Intelligent Computing and Control Systems*, 2020.

**[23]** E. K. Ikonomakis, S. Kotsiantis, and V. Tampakas, "Text Classification Using Machine Learning Techniques," no. August, 2005.

**[24]** cjadams, Jeffrey Sorensen, Julia Elliott, Lucas Dixon, Mark McDonald, nithum, and Will Cukierski, "Toxic Comment Classification Challenge," *Kaggle*, 2017.

**[25]** Ian Kivlichan, Jeffrey Sorensen, Julia Elliott, Lucy Vasserman, Martin Görner, and Phil Culliton, "Jigsaw Multilingual Toxic Comment Classification," *Kaggle*,2017.

201370

ORIGINALITY REPORT

| 13% | 10% | 8% | 6% |
|---|---|---|---|
| SIMILARITY INDEX | INTERNET SOURCES | PUBLICATIONS | STUDENT PAPERS |

PRIMARY SOURCES

| | | |
|---|---|---|
| 1 | www.ijert.org<br>Internet Source | 3% |
| 2 | curiousily.com<br>Internet Source | 1% |
| 3 | www.ijraset.com<br>Internet Source | 1% |
| 4 | www.publicatie-online.nl<br>Internet Source | 1% |
| 5 | www.hindawi.com<br>Internet Source | <1% |
| 6 | Eklas Hossain. "Machine Learning Crash Course for Engineers", Springer Science and Business Media LLC, 2024<br>Publication | <1% |
| 7 | Minhui Liang, Tiansen Niu. "Research on Text Classification Techniques Based on Improved TF-IDF Algorithm and LSTM Inputs", Procedia Computer Science, 2022<br>Publication | <1% |

# JAYPEE UNIVERSITY OF INFORMATION TECHNOLOGY, WAKNAGHAT

## PLAGIARISM VERIFICATION REPORT

**Date: …….**

**Type of Document (Tick):** **PhD Thesis** | **M.Tech Dissertation/ Report** | **B.Tech Project Report** | **Paper**

**Name:** _____ **Department:** _____ **Enrolment No** _____

**Contact No.** _____ **E-mail.** _____

**Name of the Supervisor:** _____

**Title of the Thesis/Dissertation/Project Report/Paper (In Capital letters):** _____

_____

_____

## UNDERTAKING

I undertake that I am aware of the plagiarism related norms/ regulations, if I found guilty of any plagiarism and copyright violations in the above thesis/report even after award of degree, the University reserves the rights to withdraw/revoke my degree/report. Kindly allow me to avail Plagiarism verification report for the document mentioned above.

**Complete Thesis/Report Pages Detail:**

- Total No. of Pages =
- Total No. of Preliminary pages =
- Total No. of pages accommodate bibliography/references =

**(Signature of Student)**

## FOR DEPARTMENT USE

We have checked the thesis/report as per norms and found **Similarity Index** at..... (%). Therefore, we
are forwarding the complete thesis/report for final plagiarism check. The plagiarism verification report may be handed over to the candidate.

**(Signature of Guide/Supervisor)** **Signature of HOD**

## FOR LRC USE

The above document was scanned for plagiarism check. The outcome of the same is reported below:

<table>
<tr><td>Copy Received on</td><td rowspan="4">• All Preliminary Pages<br>• Bibliography/Images/Quotes<br>• 14 Words String</td><td rowspan="2"></td><td>Word Counts</td><td></td></tr>
<tr><td>Report Generated on</td><td>Character Counts</td><td></td></tr>
<tr><td rowspan="2"></td><td>Submission ID</td><td>Total Pages Scanned</td><td></td></tr>
<tr><td></td><td>File Size</td><td></td></tr>
</table>

**Checked by**
**Name & Signature** **Librarian**

……

**Please send your complete thesis/report in (PDF) with Title Page, Abstract and Chapters in (Word File) through the supervisor at plagcheck.juit@gmail.com**