Optimisation and Deep Learning

CT118-3-3-ODL

Individual Assignment 1 and Assignment 2

APU3F2408CS(DA)

Student Name : Tang Yu Tong

TP Number : TP069018

Hand In Date : 18th May 2025

Table of Contents

Introduction

In today's digital age, with the increasing use of online platforms such as social media, forums, and comment sections, the problem of online trolls has become increasingly serious. Detecting trolls is not only crucial for maintaining healthy online discourse, but also for preventing cyberbullying, hate speech, and harassment. Different with the traditional text classification tasks, toxic comments detection faces unique challenges, including the subjectivity of the definition of “toxic”, the use of slang, abbreviations, or intentional misspellings, and the prevalence of implicit or sarcastic offensive language. This assignment develops and compares two deep learning models that can classify toxic comments. This project focuses on building adn training two models, Bi-LSTM and CNN-BiLSTM with using the dataset from Kaggle Competition dataset. Key techniques explored include using FastText vectors to handle noisy and informal language, using Focal Loss to address data imbalance, and using Bayesian optimization for effective hyperparameter tuning.

1.0 Dataset & Algorithm Selection

1.1 Literature Review

1.1.1 Challenge in Toxic Comment Classification/Text Classification

According to (Arhin et al., 2021) and (Vinod et al., 2024), there are some detection difficulties in the field of NLP classification. In the toxic comment classification problem discussed in this report, the difficulty is that there are subjective differences in the definition of “toxic”. Everyone defines toxic comments differently. Some people think it is a toxic comment, while others think that this comment is just a sharp statement and does not constitute malicious intent. Similar problems also appear in toxic-related classification tasks, such as “hate speech” or “offensive comment” detection. The boundaries between these categories are already blurred, which increases the complexity of classification and the difficulty of consistent labeling.

With the explosive growth of Internet data, social platforms, comment areas, and online forums are flooded with a large amount of semantically complex text information, and the malicious, offensive, or harmful speech contained in them is having an increasingly serious impact on the social environment. However, traditional detection methods, such as keyword-based filtering systems or early shallow machine learning models, do not perform well when faced with toxic statements that are more obscure or sarcastic (Bonetti et al., 2023). The experimental in (Vinod et al., 2024) relies solely on a traditional Multinomial Naive Bayes machine learning model,

with a bag-of-words (BoW) representation. While this approach can handle basic frequency-based text patterns, it fundamentally lacks the capacity to capture semantic relationships or contextual nuances between words. As a result, it is ineffective in detecting complex toxic expressions such as sarcasm and subtle toxic sentences.

According to the research of (Khan, 2023), in practical applications, toxic comments datasets generally have the problem of class imbalance, and the number of normal comments is much larger than that of toxic comments. This imbalance will cause a series of problems in the model training process. The most direct impact is that the model tends to favour the majority class, resulting in a high false negative rate during detection. If the model does not fully learn the characteristics of toxic comments during training, the system will frequently misclassify in actual operation and identify toxic comments as non-toxic comments. At the same time, if the model is trained too conservatively, non-toxic comments may be misclassified as toxic (false positives), affecting the user experience. Therefore, reducing the false positive rate while maintaining high accuracy is also a key challenge facing the current NLP classification detection task.

1.1.2 Method to Handle Data Imbalance Problem

According to the (Yu & Zhou, 2021), a variety of methods for dealing with data imbalance were evaluated, including sampling techniques, ensemble learning methods, and cost-sensitive methods. Oversampling methods increase the number of minority classes by generating synthetic samples of minority classes, thereby alleviating the class imbalance problem. Common oversampling techniques such as SMOTE synthesize new sample points to expand the number based on the feature relationship between existing minority class samples. Undersampling method is reducing the number of majority class samples. After comparison, the best methods are using ensemble models, such as XGBoost, random forest and Easy Ensemble. These models can handle imbalanced data and usually do not require additional sampling processing. The article states that linear models are more suitable for using SMOTE methods. Ensemble learning methods and SMOTE methods are the best choices for dealing with imbalanced data (Yu & Zhou, 2021). The AUC score of the logistic regression model improved from 85.08% to 87% and the linear SVC model improved from 83.33% to 86%.

According to the research of (Yesi Novaria Kunang et al., 2021), the focus is on the problem of class imbalance in deep learning proposes a solution by using focal loss function. Focal loss is better than the traditional cross-entropy loss function (CE). This is because cross-entropy does not consider the weight of each category and treats samples of each category equally, which is suitable for data with balanced category distribution. Focal loss introduces two adjustment parameters gamma and alpha to reduce the focus on easy-to-classify samples and improve the model's learning ability for minority class samples.

Experimental results show that even in the case of extreme imbalance, the minority class accounts for only 0.05% of the total samples, focal loss can still maintain a macro recall rate of up to 98.27%. In contrast, traditional imbalance processing methods have certain limitations. The SMOTE oversampling method is prone to overfitting, and the random undersampling method will cause the loss of important information of the majority class. Although the weighted cross entropy (WCE) alleviates the class imbalance problem to a certain extent, it performs worse than cross-entropy when facing extremely imbalanced data. After using focal loss on extremely imbalanced data sets, the model's minority class recall rate increased from 67.82% of traditional cross entropy to 74.14%, and the F1 score also increased from 74.21% to 83.77%. Experiments have shown that focal loss is superior to other traditional methods under extremely imbalanced data.

Another research paper (Song et al., 2021) discussed the role of Focal Loss in solving the problem of class imbalance by using the Jigsaw Multilingual Toxic Comment dataset. This research paper compared between using binary cross-entropy (BCE) and focal loss. (Song et al., 2021) evaluated two pre-trained models (XLM-RoBERTa and MBERT), each model trained separately by using BCE and focal loss. The outputs of these models were combined into a hybrid model call RoBERTa-MBERT-BCE-Focal Model. Result show that only using focal loss single model (MBERT-Focal) achieved a high precision of 0.9035 but a relatively low recall of 0.7381. Therefore, using focal loss can reduce false positives and improve precision. However, this improvement would let the model have lower recall, as some positive samples were misclassified. In contrast, the hybrid model combined with different loss functions can achieve a more balanced result, with the precision of 0.8360 and recall of 0.8539 in the problem of class imbalance.

1.1.3 Word Embedding Techniques

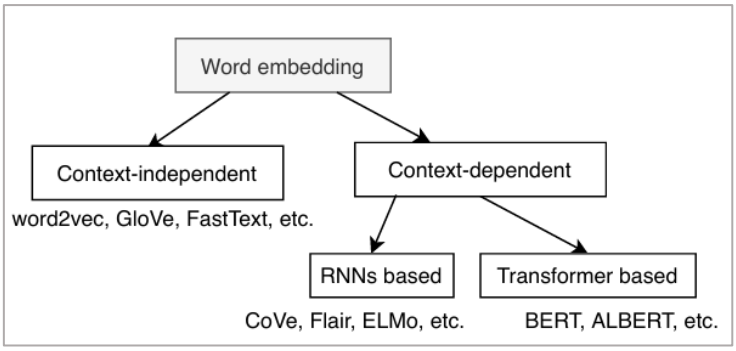


Figure 1: Types of Word Embedding (Wang et al., 2020)

Word vectors can be divided into two types: context-independent word vectors and context-dependent word vectors (Wang et al., 2020). FastText, Glove and Word2Vec are more common context-independent word vectors. Each word in a context-independent word vector has only one fixed representation, regardless of the context in which it appears. Therefore, context-independent word vectors are pre-trained on large-scale general texts and released in the form of files that users can download directly and use as the initial word vector of the model in NLP tasks.

Word2Vec captures word meanings by combining Skip-gram and Continuous Bag of Words (CBOW) (Allen & Hospedales, 2019). Word2Vec captures the relationship between words by placing similar words close together in vector space. A classic example given by (Allen & Hospedales, 2019) is “woman is to queen as man is to king”. Linear relationships in word vectors are used to reveal semantic similarities and relationships between words. GloVe trains word vectors to capture global semantic relationships by counting the number of times words appear together in the entire corpus (Wang et al., 2020). Different with TF-IDF, which treats each word as an independent unit and does not account for relationships between words, GloVe considers how frequently words appear together, enabling it to learn meaningful vector representations reflecting semantic similarity. FastText word embedding was developed by Facebook, FastText that not only learns the embedding of the entire word but also splits the word into character n-grams to learn the representation of sub-words. This makes it effective in handling rare words and out-of-vocabulary (OOV) words (Pritom Mojumder et al., 2020).

According to the study by (Dessì et al., 2021), different word embedding techniques are compared. These include Word2Vec, BERT, and Mimicking Word Embeddings to detect toxic comments. Word2Vec has a 300-dimensional vector, and BERT vectors have 1024 dimensions. Mimicking word embeddings are generated using character-level BiLSTM, which is

specialized for handling out-of-vocabulary (OOV) words. The experiment uses a 10-fold cross-validation setting. The training batch size is 128 for each model, the training cycle is 20 epochs, and the RMSprop optimizer with a learning rate is 0.001.

Table 1: Word Embedding Comparison ROC Result Provided by (Dessì et al., 2021)

Deep Learning Model	Word2Vec	Bert	Mimicking
DNN	0.921	0.898	**0.922**
CNN	0.905	0.881	**0.906**
LSTM	0.970	0.930	**0.971**
Bi-LSTM	**0.969**	0.930	**0.969**

The comparison of different word embedding techniques shows that Mimicking Word Embeddings achieve slightly better performance overall, due to the ability to handle out-of-vocabulary (OOV) words. However, the results of Google's Word2Vec embeddings are very close. BERT word vectors are currently the most advanced word vectors, but they perform poorly in the toxic detection task. The reason is that the length of the input text comments and the frequent spelling and grammatical errors in them cause the BERT word vectors to fail to capture the context of the words, thus passing the wrong information to the model (Dessì et al., 2021). Spelling errors are also a common problem and challenge in malicious comments, because users will deliberately misspell English words to evade malicious detection, resulting in a lot of noise in the dataset (Garg et al., 2023).

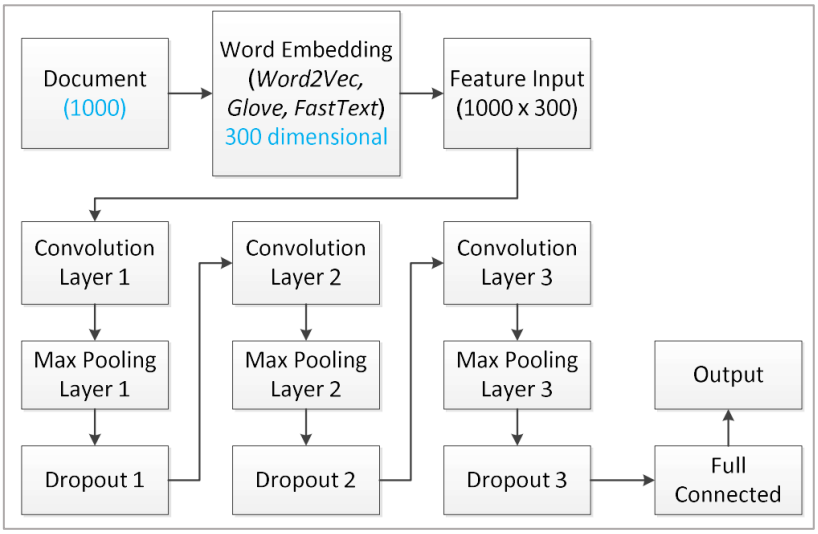


Figure 2: (Dharma et al., 2022) CNN Architecture for Text Classification

Another comparison conducted by (Dharma et al., 2022) evaluated different word embedding techniques in a CNN-based text classification model. The model was trained using the Adam

optimizer with a dropout rate of 0.5, a batch size of 128 and 20 epochs. The results showed that using FastText achieved the highest accuracy of 97.2%, followed by GloVe 95.8% and Word2Vec 92.5%. These findings suggest that FastText's ability to capture subword information makes it more effective in handling rare or out-of-vocabulary words, leading to improved classification performance.

1.1.4 Recurrent Neural Networks (RNN) Model

According to (Sharkawy, 2020), traditional recurrent neural networks (RNN) are challenging to capture long sequences of text. RNN models essentially rely on the sequential nature of data and process inputs step by step through a chain structure. However, this sequential dependency is susceptible to problems such as gradient vanishing and gradient exploding, resulting in the RNN model's weak to capture long-range dependencies and difficulty retaining early input information, leading to poor performance of the model in complex language modelling tasks.

To address these limitations, long short-term memory networks (LSTM) and gated recurrent units (GRU), two enhanced RNN variations, have been released. According to (Shinde et al., 2024), LSTM and GRU were originally designed to alleviate the gradient vanishing problem and are widely used in natural language processing (NLP) classification tasks. LSTM consists of three main gates, including the forget gate, input gate, and output gate. The amount of the prior memory that is deleted is decided by the forget gate. While the output gate chooses which portion of the memory is sent to the following layer, the input gate regulates how much of the current input should be saved in the memory. These gating mechanisms regulate the flow of information, enabling the model to selectively retain or discard information in long sequences, that is why LSTM is able to learn long-range dependencies more effectively than ordinary RNN models.

GRU is a simplified version of LSTM. GRU combines the functions of the forget gate and the input gate into an update gate and introduces a reset gate to control the amount of past information forgotten. The amount of data from the prior time step that should be kept in the present state is decided by the update gate. When the reset gate is close to zero, the model tends to discard previous information and rely more on the current input. GRU has fewer parameters than LSTM, so it is faster and more computationally efficient when training. In contrast, LSTM has high computational complexity per time step because LSTM has three gate structures and

an explicit memory unit (cell state). In each time step, LSTM needs to calculate the activation values of each gate separately, update the cell state, and then calculate the current hidden state through the output gate. This complex structure improves the expressiveness of the model but requires greater computational overhead. In comparison, GRU has a simpler structure, contains only two gates, and uses a unified hidden state to simultaneously assume the role of short-term and long-term memory, so the overall calculation process is simpler and the training speed is faster (Wang & Zhang, 2021).

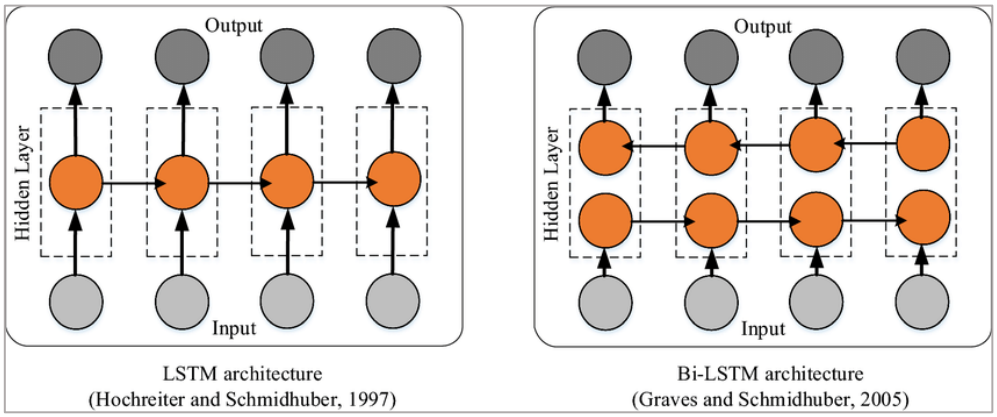


Figure 3: LSTM and Bi-LSTM Concept

Both GRU and LSTM can be extended to bidirectional models (Bi-GRU and Bi-LSTM). When processing sequences, the network not only reads data from front to back in order but also reads from back to front, which can help more fully understand the context of each time step. In unidirectional models, predictions are made only based on contextual information before the current time step, which limits the model's ability to use future information.

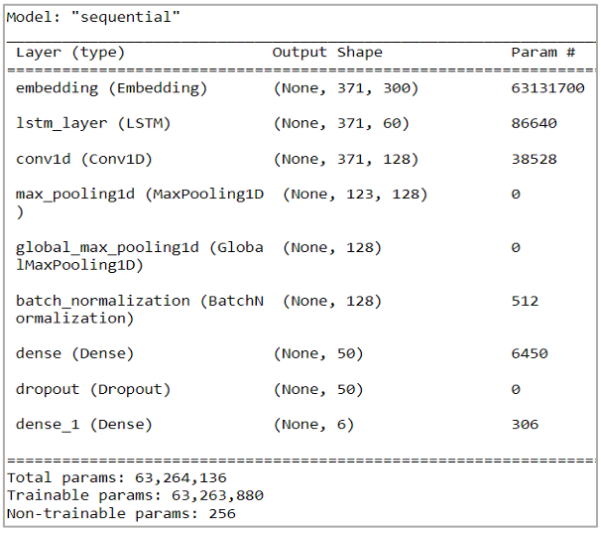
The Bi-LSTM model's accuracy was 99.13%, which was much greater than the unidirectional LSTM model's 97.3%, according to the results of model training trials conducted on the toxic comments dataset (Shinde et al., 2024). The GRU model outperforms the LSTM model in the unidirectional model with an accuracy of 98.42%. When the Bi-GRU and LSTM models were used for text categorization in (Asrawi et al., 2023), the Bi-GRU model outperformed the LSTM model, with an accuracy of 99.87% as opposed to 97.3%. This demonstrates that the bidirectional model outperforms the unidirectional model in contextually dependent processing tasks and is better captures the text's context, which enhances the model's performance.

Furthermore, (Song et al., 2021) found that Bi-GRU outperformed Bi-LSTM across multiple evaluation metrics. The Bi-GRU model achieved an accuracy of 0.8912, recall of 0.8586,

precision of 0.8015, and an F1-score of 0.8249. Although Bi-LSTM is theoretically more suited for capturing complex long-term dependencies due to the explicit memory cell and gating mechanisms, its higher computational complexity and larger number of parameters can reduce training efficiency or increase the risk of overfitting on specific datasets.

1.1.5 Hybrid Model

According to (Naidu et al., 2023), compare the performance of single models and hybrid models in toxic comment classification by using Jigsaw toxic comments dataset. This paper more focus on research LSTM combined with CNN model architecture. CNN are particularly effective at extracting local features, such as word-level or character-level n-gram patterns. CNN can detect words that users deliberately enter incorrectly or spelling variants to evade the toxic detection systems. This paper notes that character-level CNN can have better handling on variants words like “bl@ck” or “b1ack”. While, LSTM is good at capturing long-term dependencies in text, including sentence context and semantic associations, LSTM is suitable for processing sequence data. By combining both, the hybrid model first processes the global context with LSTM and then enhances local feature extraction with CNN, enabling a more comprehensive and robust analysis of toxic text.



```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding (Embedding)       (None, 371, 300)          63131700

 lstm_layer (LSTM)           (None, 371, 60)           86640

 conv1d (Conv1D)             (None, 371, 128)          38528

 max_pooling1d (MaxPooling1D  (None, 123, 128)         0
 )

 global_max_pooling1d (Globa  (None, 128)              0
 lMaxPooling1D)

 batch_normalization (BatchN  (None, 128)              512
 ormalization)

 dense (Dense)               (None, 50)                6450

 dropout (Dropout)           (None, 50)                0

 dense_1 (Dense)             (None, 6)                 306

=================================================================
Total params: 63,264,136
Trainable params: 63,263,880
Non-trainable params: 256
```

Figure 4: LSTM-CNN Hybrid Model Architecture by (Naidu et al., 2023)

(Naidu et al., 2023) hybrid model architecture first uses an embedding layer to convert text into a 300-dimensional word vector and then processes the sequence through a 60 units LSTM layer. A one-dimensional convolutional layer with 128 filters is then used to extract local features, and the sequence length is compressed through maximum pooling, and then key features are

obtained through global maximum pooling. After the batch normalization layer accelerates training, the 50-dimensional fully connected layer is used to reduce the dimension, and dropout is used to prevent overfitting, and finally 6 categories of results are output.

Model	LSTM	Character-level CNN	Word-level CNN	Hybrid model (LSTM+CNN)
Accuracy	88.75%	83.54%	90.34%	92.63%

Figure 5: Model Comparison Result by (Naidu et al., 2023)

According to (Naidu et al., 2023), this research compares of four deep learning models, LSTM, character-level CNN, word-level CNN and a hybrid CNN-LSTM model. Among these models, the CNN-LSTM hybrid model achieved the highest performance, surpassing others in both classification accuracy 92.63% and AUC-ROC score of 0.9842. According to a different study by Shinde et al. (2024), the CNN-LSTM hybrid model outperformed other models including GRU, LSTM, Bi-GRU, and Bi-LSTM, achieving the greatest accuracy of 99.68%. The advantage of the hybrid model is that it can capture the order of words in a sentence, as well as some key features. The LSTM is like "understanding" the overall context of the sentence, while the CNN layer enhances the feature extraction process by highlighting the most significant parts of the LSTM output.

According to (Wang & Zhang, 2021), conducted on several deep learning architectures for toxic comment classification, including LSTM, Bi-GRU, Bi-GRU+CNN, and the BG-GCNN model. Traditional CNN architectures rely on local pooling followed by fully connected layers for classification. In the Bi-GRU+CNN model, the Bi-GRU layer captures sequential dependencies, and its output is passed to a CNN for local convolution and pooling. The pooled features are then concatenated and fed into fully connected layers for final classification. In contrast, the BG-GCNN model introduces a more efficient structure by applying global average pooling and global max pooling directly to the CNN output. BG-GCNN model generates fixed-length feature vectors without the need for additional fully connected layers, thereby reducing computational complexity. The results showed that while LSTM and Bi-GRU models achieved higher F1 scores than the BiGR-CNN model. And the best model is BG-GCNN, achieving the best F1 score of 81%.

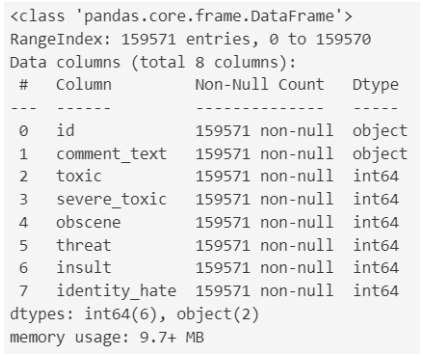
1.2 Summary of Literature Review

No	Research Topic	Author(s)	Description	Model Evaluation	Findings/Challenges	Limitations/Gap
1	A Study of Multilingual Toxic Text Detection Approaches under Imbalanced Sample Distribution	Guizhe Song, Zhifeng Xiao, Degen Huang	This research focuses on comparing different loss functions and evaluating the performance of pre-trained versus non-pretrained models.	**Non-pretrained Model F1 score:** Logistic Regression: 75.94% CNN+FastText: 78.22% Bi-LSTM: 78.28% Bi-GRU: 82.49% **Pre-trained Model F1 score:** XLM-R (BCE): 83.81% XLM-R (focal loss): 83.72% XLM-Mix: 83.89% MBERT-Mix: 83.73% XLM-Mix-MBERT-Mix: 84.46%	**Findings:** Experimental results show that the fusion method based on different loss functions effectively solves the problem of imbalanced accuracy and recall caused by imbalanced sample distribution.	**Limitation:** The number of multilingual corpora other than English is small, which reduces the model performance to a certain extent. **Gap:** Failure to use the sample language reconstruction technique to improve the text's linguistic diversity and richness in order to lessen the effects of data imbalance.
2	Toxic Comment Classification Based on Bidirectional Gated Recurrent Unit and Convolutional Neural Network	Zhongguo Wang, Bao Zhang	This research focuses on comparing hybrid model with includes more efficient model structure (BG-GCNN).	**F1-Score:** LSTM: 80% Bi-GRU: 80% Bi-GRU+CNN: 78% **BG-GCNN:81%**	**Challenges:** Data imbalance problem. **Finding:** Experimental results show that the bidirectional gated recurrent unit (BG-GCNN) model combined with a global pooling optimized convolutional neural network (CNN) achieves good performance.	**Limitation:** Using GloVe word vectors may not cover words in some specific fields, such as Internet slang. Moreover, it is a fixed word vector and cannot flexibly adjust the meaning of words according to different contexts. Its understanding of polysemous words will be limited. **Gap:** Optimize the algorithm further to solve performance problems caused by less data or data imbalance.

No	Research Topic	Author(s)	Description	Model Evaluation	Findings/Challenges	Limitations/Gap
3	A Comparative Study on Word Embeddings in Deep Learning for Text Classification	Congcong Wang,Paul Nult, David Lillis	This research paper compares different word embeddings (word2vec, GloVes, Fast-Text, ELMo, BERT) performed across various models.	**20NewsGroup Dataset macro-f1:** CNN (Word2Vec): 82.14% CNN (ELMo): 78.33% CNN (Bert): 82.88% CNN (FastText-300): 81.57% CNN (Glove-300): 81.85% **CNN (Glove300+FastText300): 83.65%** Bi-LSTM (Word2Vec): 62.43% Bi-LSTM ((ELMo): 71.13% **Bi-LSTM ((Bert): 80.54%** Bi-LSTM ((FastText-300): 62.40% Bi-LSTM ((Glove-300): 73.97% Bi-LSTM ((Glove300+FastText300): 75.77%	**Findings:** 1.GloVe/FastText cannot handle polysemous words. 2. Pre-trained embeddings outperform randomly initialized baseline models.	**Limitations:** 1. Limited to four benchmark datasets and does not cover other text classification tasks such as multi-label and multi-language. 2. No optimization strategy is designed for data imbalance problems. **Gaps:** 1. Lack of extended exploration of more pre-trained models.
4	An Assessment of Deep Learning Models and WordEmbeddings for Toxicity Detection within OnlineTextual Comments	Danilo Dessì, Diego Reforgiato Recupero, Harald Sack	This research paper explores the most effective deep learning layers and different word embeddings (Word2Vec, Bert, Mimicking) for detecting toxic comments.	**F1 Score:** CNN (word2Vec): 84.80% CNN (BERT): 80.50% **CNN (Mimicking): 85%** LSTM (word2Vec): 91.40% LSTM (BERT): 85.80% **LSTM (Mimicking): 91.60%** Bi-LSTM (word2Vec): 91.40% Bi-LSTM (BERT): 85.60% **Bi-LSTM (Mimicking): 91.50%**	**Findings:** 1. Mimicked word embeddings work best in detection tasks because it inherit the knowledge of large-scale pre-trained embeddings and ability to handle out-of-vocabulary problems. 2. BERT performs poorly on this task, due to the failure of context learning due to input text length, spelling errors or grammatical issues. 3. LSTM-based model have a better performance than CNN and DNN model.	**Limitatios:** 1. Data Imbalance problem cause poor generalization ability in real data. 2. BERT's characteristic of assigning different vectors to the same word may interfere with deep model training. **Gap:** Evaluate more contextual embedding methods, such as ELMo.

No	Research Topic	Author(s)	Description	Model Evaluation	Findings/Challenges	Limitations/Gap
5	Bidirectional LSTM with Convolution for Toxic Comment Classification	Ashish Shinde, Pranav Shankar, Atul, , Srikari Rallabandi	This research paper compares different models for toxic comment classification, including unidirectional models, bidirectional models, and a hybrid LSTM+CNN model.	**Accuracy:** LSTM: 97.3% GRU: 98.42% Bi-LSTM: 99.13% **Bi-GRU: 99.87%** LSTM+CNN: 99.68%	**Finding:** The bidirectional model performs better than the unidirectional model. The hybrid model performs better than bidirectional model.	**Limitatons:** 1. Lack of robustness against adversarial inputs and evolving language patterns affects model generalization. 2. Models are not optimized for real-time processing or large-scale deployment. **Gap:** 1. Most current approaches focus only on text and ignore multimodal information like images or user context.
6	Toxic Comment Classification using Deep Learning	B. Ramesh Naidu, Naresh Tangudu, Ch. Chandra Sekhar, K. Kavitha, B. V. Ramana, P. Venkateswarlu Reddy, Jayavardhanarao Sahukaru, Raj Ganesh Lopinti	This study uses LSTM, CNN and hybrid models to classify and detect toxic comments from online platforms.	**Accuracy:** LSTM: 88.75% Character-Level CNN: 83.54% Word-Level CNN: 90.34% **LSTM+CNN: 92.63%**	**Findings:** 1. Character-level CNN can have better handle variant words. 2. The hybrid model (LSTM + CNN) outperformed the other single models in terms of AUC ROC score and accuracy.	**Limitation:** 1. Research focused on English-language datasets and lacks evaluation across other languages or multilingual contexts. **Gap:** 1. The current model structure does not explore deeper or more complex architectures, such as bidirectional LSTM, which could improve long-text classification.

1.3 Justify Data Selection

Data Set Name	Dataset Explanation	Dataset Size	The Proportion of Target Column	Observation	Data Source	Data Source Link
Jigsaw-Toxic-Comment-Classification	This dataset is for toxic comment classification project. This dataset is author's get from Kaggle, and those comments are from Wikipedia talk pages.	159,571 comments and 8 columns	1: toxic (144277 rows) 0: non-toxic (15294 rows)	 The primary target label for classification is toxic, which is highly imbalanced with 144,277 toxic comments (90.4%) and 15,294 non-toxic comments (9.6%). Showing that this dataset has an imbalance problem. Without proper handling dataset imbalance problem, the model is likely to become biased toward the majority class, leading to poor performance in detecting toxic comments.	GitHub	https://github.com/praj2408/Jigsaw-Toxic-Comment-Classification/tree/main

Jigsaw's toxic classification dataset was selected since this dataset is well-known and is also utilized by Kaggle for competition standards. This dataset provides a comprehensive collection of real online comments from Wikipedia containing 159,571 entries with six binary labels, including toxic, severely toxic, obscene, threat, insult and hate speech. For this study, only the toxic label is used as the target variable, other categories are not considered. The ratio of the number of non-malicious comments to the number of malicious comments in this dataset is 9:1, which simulates the distribution of toxic comments on the real platform and tests the performance of the model under extremely unbalanced data.

2.0 Data Preparation

2.1 Exploratory Data Analysis (EDA) – Before Text Preprocessing

2.1.1 Distribution of Toxic and Non-Toxic Comments

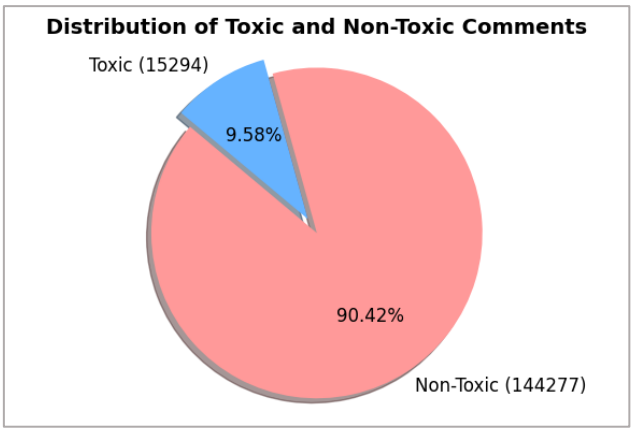


Figure 6: Distribution of Toxic and Non-Toxic Comments

Before text preprocessing, this dataset consisted of 159,571 rows, with 144,277 (90.42%) labelled as non-toxic and 15,294 (9.58%) labelled as toxic. This indicates that the dataset is imbalanced, which may affect model performance.

2.1.2 Check Empty and Duplicated Rows

```
file_path = r".\Ori_Dataset\ToxicData.csv"
data = pd.read_csv(file_path, encoding='ISO-8859-1')

print("\nMissing values per column before drop:")
print(data.isnull().sum())
```

✓ 0.7s

```
Missing values per column before drop:
text     0
label    0
dtype: int64
```

Figure 7: Check Empty Rows

```
file_path = r".\Ori_Dataset\ToxicData.csv"
data = pd.read_csv(file_path, encoding='ISO-8859-1')

duplicate_rows = data[data.duplicated()]
print(f"Number of duplicated rows: {duplicate_rows.shape[0]}")
```

```
Number of duplicated rows: 3
```

Figure 8: Check Duplicated Rows

After dropping the unused columns, this dataset has no empty rows, but three duplicated entries may need to be removed.

2.1.3 Word Frequency

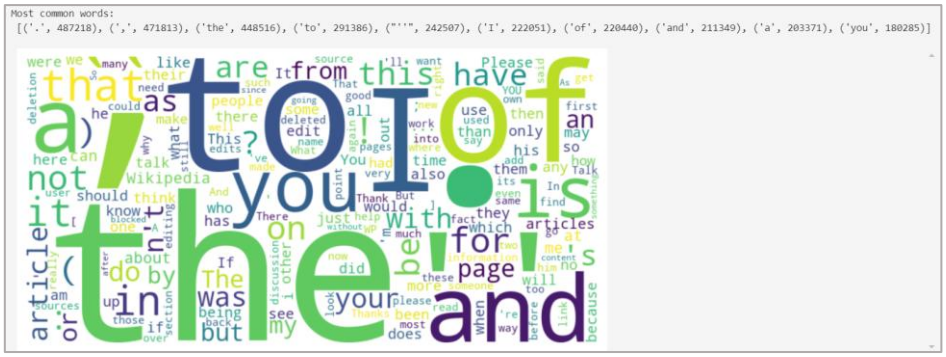


Figure 9: Word Frequency Before Text Preprocessing

From the figure, before text preprocessing, the most frequent tokens are punctuation and common stop words. This is expected in raw text data, but these tokens typically do not contribute meaningful information for classification tasks. Therefore, removing stop words and punctuation is necessary to improve model performance by focusing on more informative words.

2.1.4 Unknown Word

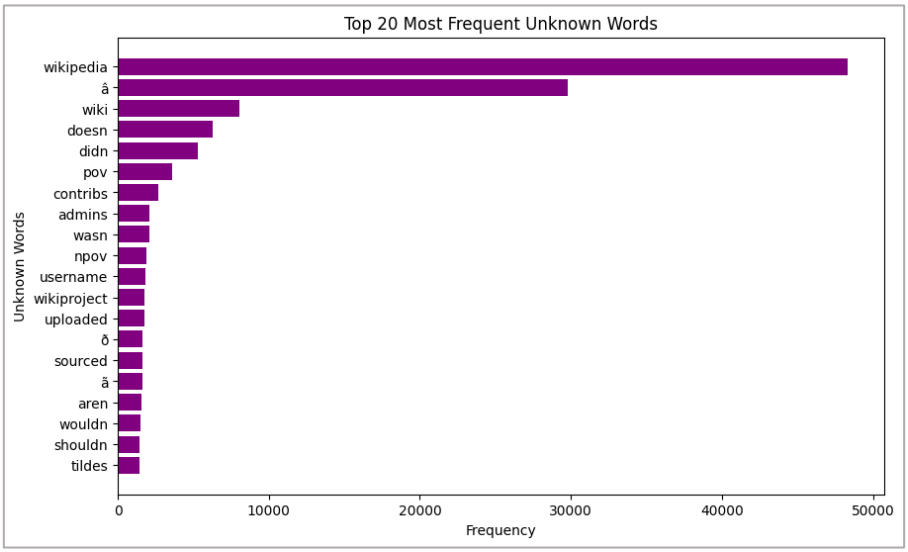


Figure 10: Top 20 Unknown Words Before Text Preprocessing

This unknown word bar chart is used to identify potential data noise by comparing the dataset's words against the NLTK WordNet and NLTK word dictionaries. This step helps highlight abbreviations or unknown words that need to be standardized or replaced during text preprocessing. The most frequent unknown terms include "Wikipedia", "â" and various abbreviation. If these data noises are not cleaned up, the model may incorrectly learn associations between noisy words and the target class, leading to inaccurate predictions.

2.1.5 N-Gram of Toxic Comments

Table 2: N-Grams of Toxic Comments Before Text Preprocessing

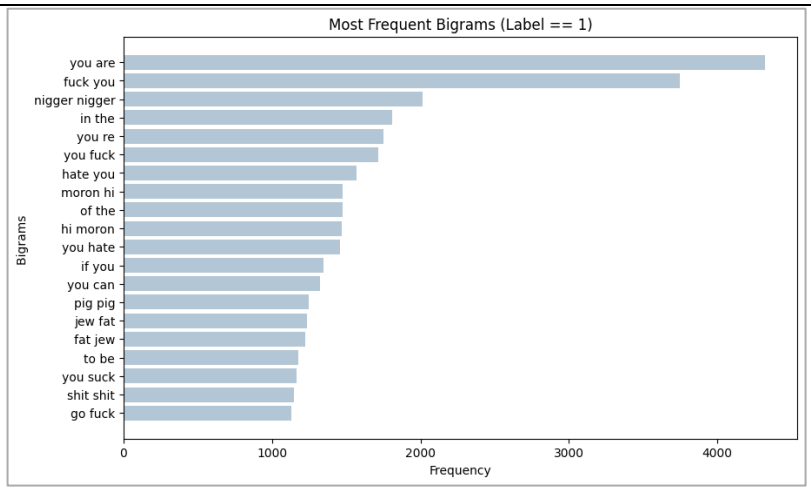
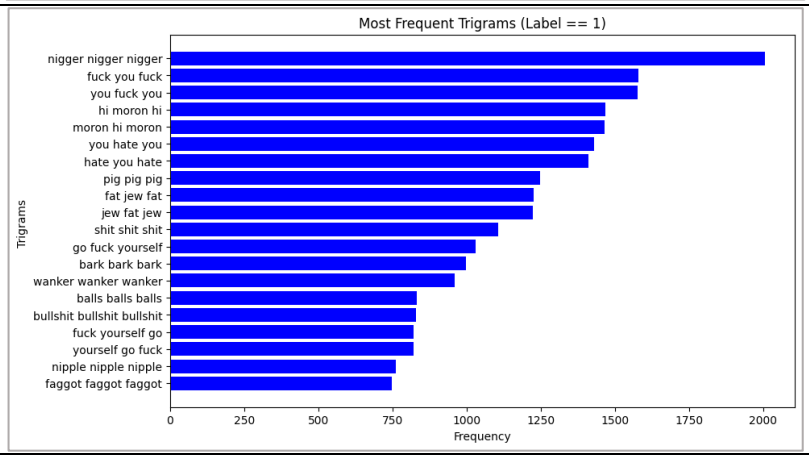
Bigrams	
Trigrams	

Figure 11: Repeated Words in Dataset

As observed from the trigram frequency table and the dataset screenshot, the dataset contains many instances of repeated words such as “nigger nigger nigger”, “pig pig pig” and “shit shit shit”. For example, in row 13354, the word “nigger” appears repeatedly. A large number of repeated words must be removed during text preprocessing, those words will mislead the model to focus too much on certain words rather than the meaning of the sentence, which may lead to model misclassification.

2.2 Text Preprocessing

2.2.1 Drop Unused Column & Rename Header

Figure 12: Drop Unused Column & Rename Header

Drop severe toxic, obscene, threat, insult, and hate column, only remain “comment text” and “toxic” columns used for training the model. And rename the header to text and label.

2.2.1 Lowercase

```
file_path = r".\Ori_Dataset\ToxicData.csv"
df = pd.read_csv(file_path, encoding='ISO-8859-1')

df['text'] = df['text'].str.lower()
print(df['text'].head())

file = r".\Pre_Dataset\1_Lowercasing.csv"
df.to_csv(file, index=False, encoding='ISO-8859-1')
```

```
0    explanation\nwhy the edits made under my usern...
1    d'aww! he matches this background colour i'm s...
2    hey man, i'm really not trying to edit war. it...
3    "\nmore\ni can't make any real suggestions on ...
4    you, sir, are my hero. any chance you remember...
Name: text, dtype: object
```

Figure 13: Lowercase

Lowercase all text data to ensure consistency and uniformity of the dataset. This step helps avoid treating the same word, such as “Bad” and “bad”, with different capitalization as two words during text analysis and model training.

2.2.2 Remove Duplicated Rows

```
file_path = r".\Pre_Dataset\1_Lowercasing.csv"
data = pd.read_csv(file_path, encoding='ISO-8859-1')

duplicate_rows = data[data.duplicated()]
print(f"Number of duplicated rows: {duplicate_rows.shape[0]}")

data = data.drop_duplicates()

output_file_path = r".\Pre_Dataset\1_Lowercasing.csv"
data.to_csv(output_file_path, index=False, encoding="utf-8")

duplicate_rows = data[data.duplicated()]
print(f"Number of duplicated rows (after Drop) : {duplicate_rows.shape[0]}")
```

```
Number of duplicated rows: 43
Number of duplicated rows (after Drop) : 0
```

Figure 14: Remove Duplicated Rows

After converting all text to lowercase, the number of duplicated rows increased to 34 and was removed. If these duplicates were not removed can cause data leakage, as identical rows may appear in the training, validation and test sets. This would lead to inaccurate model evaluation by inflating performance metrics and compromising the model's generalization ability.

2.2.3 Remove URL, Emojis, Punctuation, Excessive Whitespace, Only Number Rows, Elongation

```
# Remove URLs
df['text'] = df['text'].apply(lambda x: re.sub(r'\b(?:http|https|www)\S+|\b\S+\.(com|net|org|io|gov|edu|cn)\b', '', str(x)))

# Remove HTML
html_tags_pattern = r'<.*?>'
# sub(pattern,replace,text)
df['text'] = df['text'].apply(lambda x: re.sub(html_tags_pattern, ' ', str(x)))
df['text'] = df['text'].apply(lambda x: re.sub(r'\S+html\b', ' ', str(x)))

# Remove emojis
def remove_emojis(text):
    text = emoji.demojize(text)
    text = re.sub(r':\w+:', ' ', text)
    return text
df['text'] = df['text'].apply(remove_emojis)

# Remove ASCII characters & delete unused punctuation
df['text'] = df['text'].apply(lambda x: x.replace("_", " "))
df['text'] = df['text'].apply(lambda x: re.sub(r"[^A-Za-z0-9\s']", ' ', str(x)))
df['text'] = df['text'].apply(lambda x: x.replace("'", ""))

# Remove excessive whitespace
# .strip() removes any leading or trailing whitespace from the text
df['text'] = df['text'].apply(lambda x: re.sub(r'\s+', ' ', x).strip())

# Remove only number rows
only_numbers_df = df[df['text'].astype(str).str.strip().str.isdigit()]
# print(only_numbers_df)
df = df[~df['text'].astype(str).str.strip().str.isdigit()] #turn all data into string (astype(str), remove space (strip()), check is whole string is digit)

# Removing elongation (example: goodddddddddd)
df['text'] = df['text'].apply(lambda x: re.sub(r'(.)\1{2,}', r'\1\1', x))
```

Figure 15: Remove Data Noise

This step is to reduce noise in the dataset by cleaning elements such as URLs/HTTP links, HTML tags, punctuation, excessive whitespace, and rows containing only numbers. It also includes the removal of character elongation.

Users often repeat characters to emphasize emotions, such as “happyyyy” or missss.” In this preprocessing step, repeated characters are reduced to a maximum of two consecutive letters

("happyyyy" becomes "happyy", "missss" becomes "miss"). Without this normalization, the classifier may treat elongated words as entirely different from their base forms, leading to misinterpretation and reduced frequency counts (Siino et al., 2024).

2.2.4 Replace Abbreviations and Unknown Words

```
abbreviations = {
    "werent":"were not","arent": "are not",
    "isnt": "is not",
    "cant": "can not",
    "hasnt": "has not",
    "hadnt": "had not",
    "shes": "she is","hes": "he is",
    "youre": "you are",
    "youll": "you will",
    "youve": "you have",
    "weve": "we have",
    "yall":"you all",
    "theyre": "they are",
    "theyve": "they have",
    "doesnt": "does not",
    "dont":"do not",
    "didnt": "did not",
    "wont": "will not",
    "wouldnt": "would not",
    "shouldnt": "should not",
    "couldnt": "could not",
    "im": "i am",
    "iam": "i am",
    "ive": "i have",
    "id": "i would",
    "cuz":"because", "bcuz":"because","becuz":"because",
    "fucksex": "fuck sex",
    "yourselfgo": "yourself go",
    "bitchesfuck": "bitches fuck",
    "youfuck": "you fuck",
    "diff": "different",
    "backgroundcolor": "background color",
}
def replace_abbreviations(text):
    text = str(text)
    for abbr, full_form in abbreviations.items():
        text = re.sub(r'\b' + re.escape(abbr) + r'\b', full_form, text)
    return str(text)
df['text'] = df['text'].apply(replace_abbreviations)
```

Figure 16: Replace Abbreviations and Unknown Words

Based on the visualization of unknown words diagram create separate abbreviation lists for toxic comment dataset.

Nowadays, online communication often includes the use of abbreviations "cuz" and "youve". This normalization helps align informal language with a standard vocabulary, improving the model's ability to understand the words and process the text accurately. Replace abbreviations in the dataset using \b around abbreviations to ensure the pattern matches only when abbreviations appear as a standalone word, not as part of another word. For example, if the abbreviation is "im," \bim\b would match "im" but would not match "important" or "improve."

2.2.4 Tokenization

```
filteredTokens = []
for token in df['text']:
    token = str(token)
    wordtokens = nltk.tokenize.word_tokenize(token)
    filteredTokens.append(wordtokens)
df['text']=filteredTokens
```

Figure 17: Tokenization

Tokenizing each word streamlines the processes of removing stop words and performing lemmatization. In this step, use the NLTK "word_tokenize" function to split sentences into individual words for more effective preprocessing.

2.2.5 Remove Stop Words

```
df['text'] = df['text'].apply(lambda x: ast.literal_eval(x) if isinstance(x, str) else x) #(Yadav, 2023)

stopTokens = nltk.corpus.stopwords.words("english")

def removeStopWord(words):
    return [word for word in words if word.lower() not in stopTokens]
df['text'] = df['text'].apply(removeStopWord)
```

Figure 18: Remove Stop Words

Removing stop words is crucial because if not removed, common words like "a", "and" and "the" will dominate the frequency analysis, as these are frequently used in English sentences. This can negatively impact the model's ability to capture meaningful toxic features, as the model might incorrectly associate the presence of these common words with specific toxic or non-toxic features.

2.2.6 Lemmatization

```
# Lemmatization
import nltk
from nltk.stem import WordNetLemmatizer
from nltk.corpus import wordnet
import pandas as pd
nltk.download('wordnet')
nltk.download('averaged_perceptron_tagger') #used for Part-of-Speech (POS) tagging

lemmatizer = WordNetLemmatizer()

def get_pos_tagging(word):
    #[0][1]:('running', 'VBG')
    #[0][1][0]:('V')
    tag = nltk.pos_tag([word])[0][1][0].upper()
    tag_dict = {"J": wordnet.ADJ, "N": wordnet.NOUN, "V": wordnet.VERB, "R": wordnet.ADV} #need this for wordnet cuz wordnet only have 4 postag
    return tag_dict.get(tag, wordnet.NOUN)  # Default to noun if no match

def lemmatize_text(text):
    lemmatized_words = [lemmatizer.lemmatize(word, get_pos_tagging(word)) for word in text]
    return ' '.join(lemmatized_words)
```

Figure 19: Lemmatization

For this project, lemmatization was selected instead of stemming because it preserves semantic accuracy and contextual meaning, which are important for analysing text data. Stemming usually reduces words to a rough root form by removing word suffixes without regard for grammar or meaning. For example, stemming may reduce “better” and “be” to the same root word “be”, the meanings of the two words are different, after stemming the sentences will lose their original meaning. While lemmatization reduces words to their correct base form based on the relationship of the context without losing the original meaning of the sentence. Therefore, lemmatization is a more appropriate choice for this project.

2.2.7 Remove Excessive Repetitions within a Sentence

```
def remove_excessive_repetition(text, phrase_len=2, repeat=5):
    words = text.split()
    cleaned = []
    i = 0

    while i < len(words):
        phrase = ' '.join(words[i:i + phrase_len])
        count = 1
        j = i + phrase_len

        while j + phrase_len <= len(words) and ' '.join(words[j:j + phrase_len]) == phrase:
            count += 1
            j += phrase_len

        if count >= repeat:
            cleaned.extend(phrase.split())
            i = j
        else:
            cleaned.append(words[i])
            i += 1

    return ' '.join(cleaned)

df['text'] = df['text'].apply(lambda x: remove_excessive_repetition(x, phrase_len=2, repeat=5))
```

Figure 20: Remove Excessive Repetitions of the Same Phrase within a Sentence

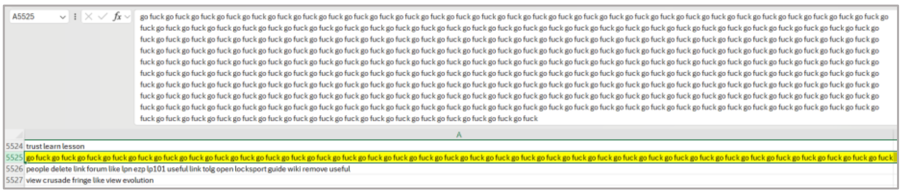


Figure 21: Examples of Rows Containing Excessive Bigram Repetitions

Through the bigram and trigram chart show there are many repetitions of the same phrase in a sentence. So, this step is designed to remove those excessive repetitions. This code identifies consecutive repeated bigrams phrases that appear more than a specified 5 times and eliminates those repetitions, leaving only one occurrence of the phrase. To avoid affecting model classification.

```
file_path = r".\Pre_Dataset\7_Lemmatization.csv"
df = pd.read_csv(file_path, encoding='utf-8')

def remove_repeated_words(text):
    if isinstance(text, str):
        return re.sub(r'\b(\w+)( \1\b)+', r'\1', text)
    return text

df['text'] = df['text'].apply(remove_repeated_words)

df = df.dropna()
df = df.drop_duplicates()

output_file_path = r".\Pre_Dataset\Final.csv"
df.to_csv(output_file_path, index=False, encoding="utf-8")
```

Figure 22: Remove Duplicated, Null and High Frequency Repeated Words

Figure 23: Examples of Rows Containing Excessive Words Repetitions

After remove repetitions of the same phrase, but dataset still contain excessive repetition of the same word in a sentence, such as "gay gay gay". Last, is remove any duplicate or empty entries. The resulting dataset is clean and ready to be used for training the toxic comment classification model.

2.3 After Text Preprocessing

2.3.1 Dataset Description

```
file_path = r".\Pre_Dataset\Final.csv"
data = pd.read_csv(file_path, encoding='ISO-8859-1')
data.info()
```

✓ 0.5s

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 158167 entries, 0 to 158166
Data columns (total 2 columns):
 #   Column  Non-Null Count   Dtype
---  ------  --------------   -----
 0   text    158167 non-null  object
 1   label   158167 non-null  int64
dtypes: int64(1), object(1)
memory usage: 2.4+ MB
```

Figure 24: Data Information After Text Preprocessing

After text preprocessing, the dataset contained a total of 158,167 entries.

2.3.2 Distribution of Toxic and Non-Toxic Comments

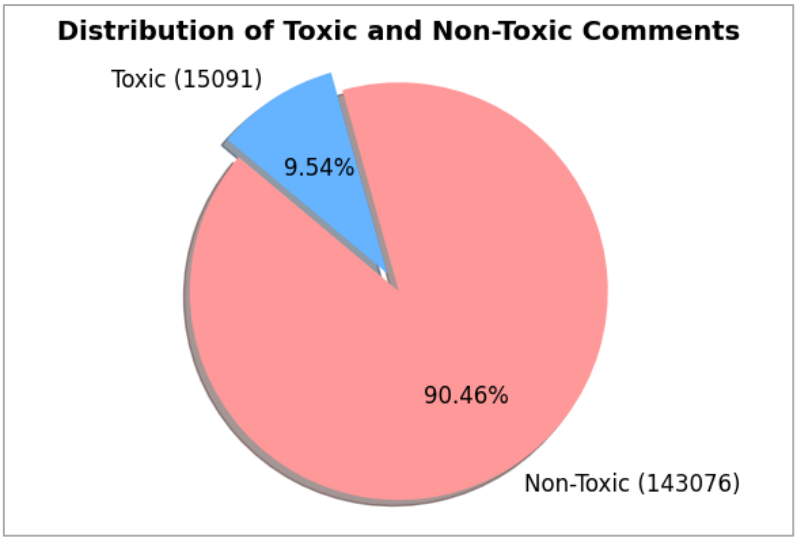


Figure 25: Distribution of Labels

Among the 158,167 entries, 143,076 (90.46%) are labelled as non-toxic, while 15,091 (9.54%) are labelled as toxic.

2.3.3 Check Empty Rows & Duplicated Rows

```
file_path =  r".\Pre_Dataset\Final.csv"
data = pd.read_csv(file_path, encoding='ISO-8859-1')

print("\nMissing values per column before drop:")
print(data.isnull().sum())

duplicate_rows = data[data.duplicated()]
print(f"Number of duplicated rows: {duplicate_rows.shape[0]}")
```

✓ 0.7s

```
Missing values per column before drop:
text     0
label    0
dtype: int64
Number of duplicated rows: 0
```

Figure 26: No Empty Rows and Duplicated Rows Before Text Preprocessing

After text cleaning, do not have empty and duplicated rows.

2.3.4 Word Frequency

Non-Toxic Comments Word Frequency

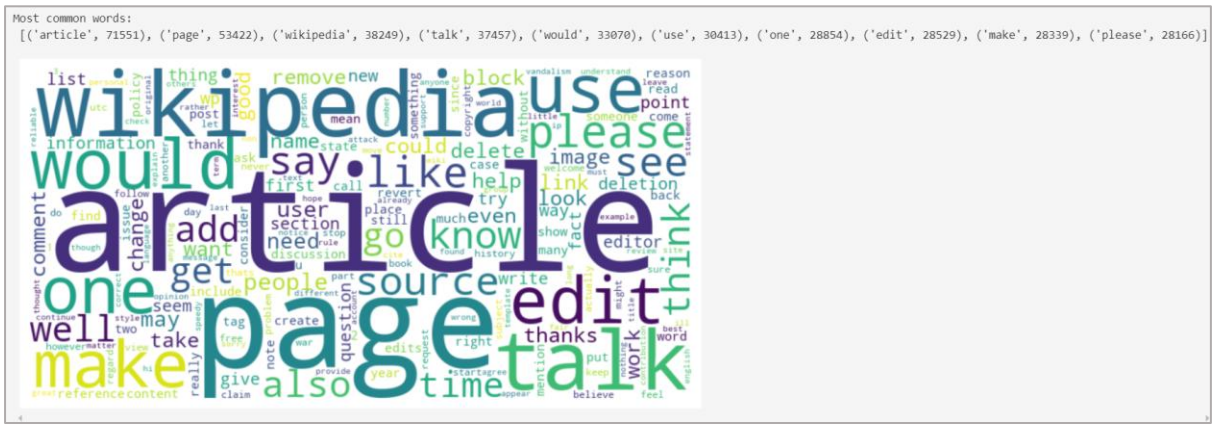


Figure 27: Non-Toxic Comments Word Frequency

The top five most frequent words in non-toxic comments are “article”, “page”, “Wikipedia”, “talk”, and “would”.

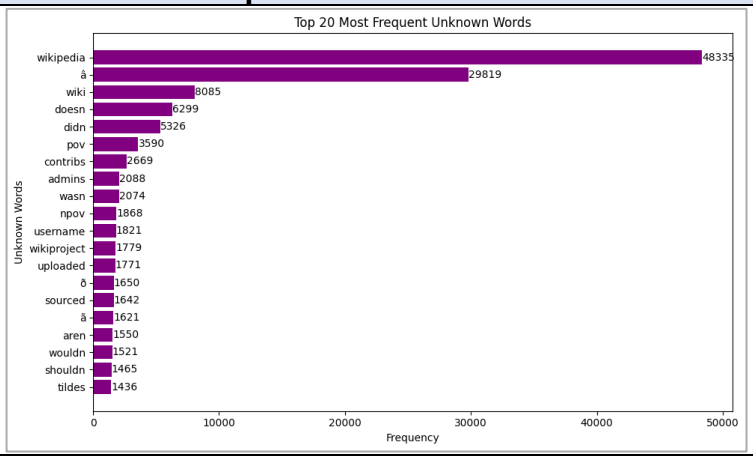
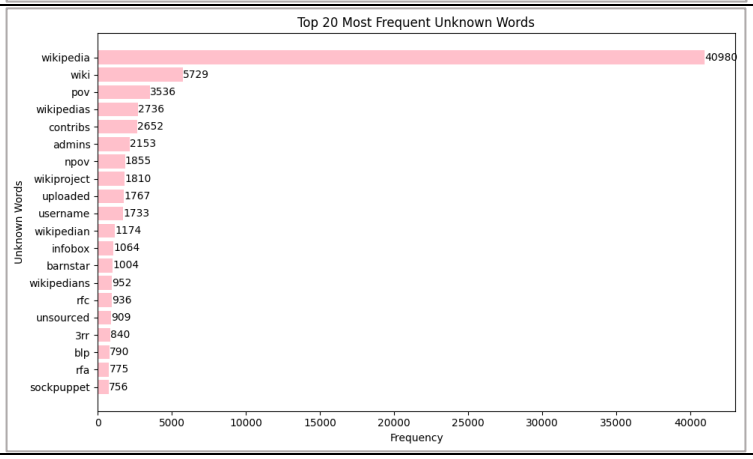
Toxic Comments Word Frequency

Figure 28: Toxic Comment Word Frequency

In toxic comments, the most frequent words are clearly offensive terms such as “fuck”, “suck” and “shit”, reflecting the toxic of the content.

2.3.5 Unknown Word

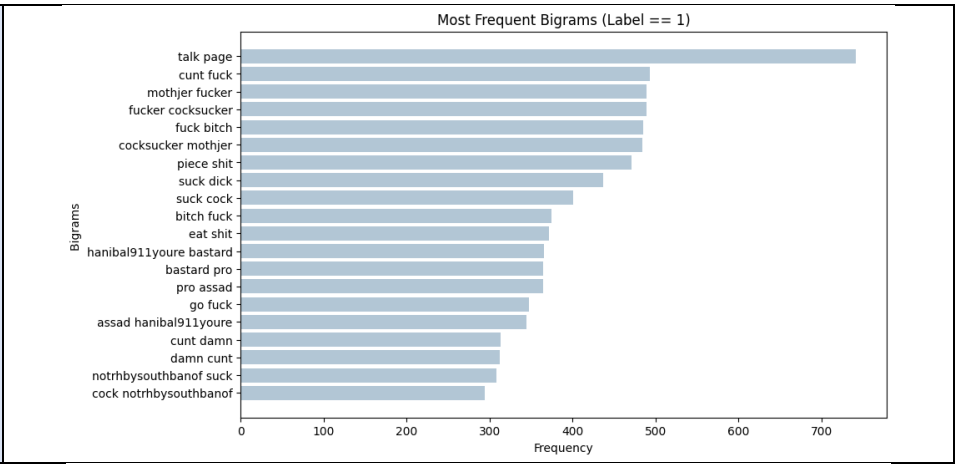
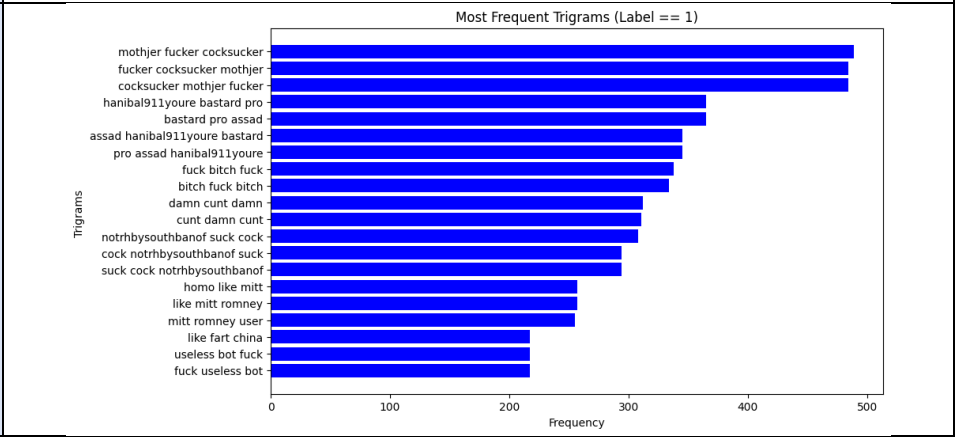
Table 3: Top 20 Unknown Word Before and After Text Preprocessing

	Top 20 Unknown Words
Before Text Preprocessing	
After Text Preprocessing	

By replacing unknown words and abbreviations, a portion of the noise in the data was reduced. For example, tokens like “â”, “'ð”, “doesn” and “wasn” were cleaned or corrected during text preprocessing.

2.3.6 N-Gram of Toxic Comments

Table 4: N-Grams of Toxic Comments

Bigrams	
Trigrams	

After removing the consecutive repeated words in the sentence, the data set becomes cleaner and more meaningful, reducing noise interference. This can prevent the model from being misled by high-frequency repeated words and make it more focused on learning the true relationship between words. As shown in the table, the most common bigram in malicious comments is “talk page”, while the most common triple is “mothjer fucker cocksucker”.

2.3.7 POS Tags

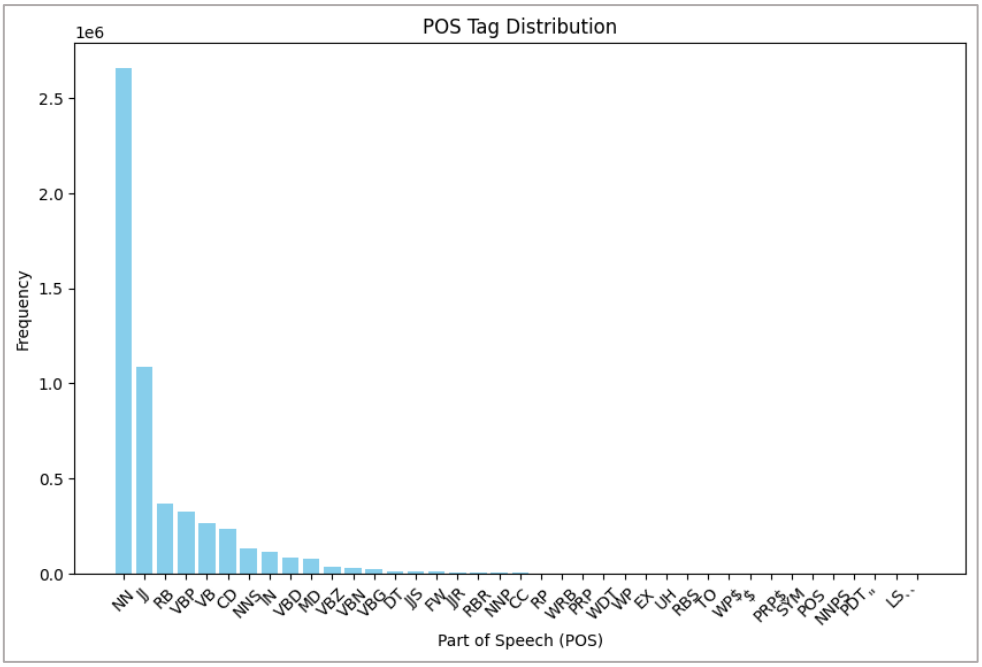


Figure 29: POS Tags Distribution

This bar chart represents the Part of Speech (POS) Tag Distribution within the toxic comment dataset, showing the frequency of various POS tags. The most frequent tag is "NN" (noun), followed by “JJ” (adjective) and “RB” (Adverb) (Penn Treebank P.O.S. Tags, n.d.). Less frequent tags include “SYM” (Symbol) and “NNPS" (Proper noun, plural) (Penn Treebank P.O.S. Tags, n.d.).

3.0 Import Library and Data Preparation

Import Library

```
# ========== Data Preparation ==========
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from imblearn.under_sampling import RandomUnderSampler
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from collections import Counter
import joblib

# ========== Model Building ==========
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, LSTM, Conv1D, MaxPooling1D, Dense, Dropout, Bidirectional
from tensorflow.keras.optimizers import Adam, SGD, Adagrad
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau
from tensorflow.keras.saving import register_keras_serializable
from tensorflow.keras import backend as K

# ========== Word Embedding ==========
from gensim.models import FastText

# ========== Model Tuning ==========
from keras_tuner.tuners import BayesianOptimization

# ========== Evaluation ==========
from sklearn.metrics import classification_report, confusion_matrix, f1_score, precision_score, recall_score
from sklearn.metrics import roc_curve, auc, accuracy_score, precision_recall_curve, average_precision_score

# ========== Visualization ==========
import matplotlib.pyplot as plt
import seaborn as sns
import matplotlib.ticker as mtick
```

Figure 30: Import Library

Some libraries such as Pandas, NumPy, and Sklearn used for data preparation, label encoding, and dataset splitting, while RandomUnderSampler is applied for handling class imbalance through undersampling.

In **model building**, TensorFlow and Keras provided core functionalities for constructing neural network layers such as Conv1D, Bidirectional LSTM, and Dense, along with optimizers like Adam, SGD, and Adagrad. For **word embeddings**, the FastText model from the gensim.models package was employed to capture subword-level semantics. During **model training**, EarlyStopping was used to monitor the validation loss and prevent overfitting by halting training once performance stopped improving. When validation loss does not decrease, the learning rate is auto lowered by ReduceLROnPlateau.

For **evaluation**, including accuracy, precision, recall, F1-score, and ROC and PR curves were computed using sklearn.metrics.

Finally, for **visualization**, matplotlib and seaborn were used to generate informative plots, such as learning curves, confusion matrices, precision-recall curves and correlation between threshold and precision-recall.

Data Preparation

```
import pandas as pd

file_path = r".\Pre_Dataset\Final.csv"
data = pd.read_csv(file_path)

text_column = 'text'

data['text_length'] = data[text_column].astype(str).apply(lambda x: len(x.split()))

text_length_stats = data['text_length'].describe(percentiles=[0.95])
print(text_length_stats)
```

```
count    158176.000000
mean         35.208894
std          53.909820
min           1.000000
50%          19.000000
95%         118.000000
max        1250.000000
Name: text_length, dtype: float64
```

Figure 31: Text Length in Whole Dataset

This analysis helps us understand the sentence length across the dataset. As shown in the figure, 95% of the sentences contain fewer than or equal to 118 words. Based on this observation, setting the maximum sequence length to 200 words during the data preparation step. This value strikes a balance between covering most sentences and avoiding excessively long input sequences that would increase training time and computational cost.

```
file_path =  r".\Pre_Dataset\Final.csv"
data = pd.read_csv(file_path)

X = data['text'].astype(str).values
Y = data['label'].values
label_encoder = LabelEncoder()
Y = label_encoder.fit_transform(Y)

max_words = 20000  # vocab size
max_len = 200  # the max length of the each sentences

tokenizer = Tokenizer(num_words=max_words)
tokenizer.fit_on_texts(X)
X_seq = tokenizer.texts_to_sequences(X)
X_padding = pad_sequences(X_seq, maxlen=max_len)

# Train-Valid-Test Split
# First split into train 70%, valid 20% + test 10%
X_train, X_valid_test, Y_train, Y_valid_test = train_test_split(X_padding, Y,  test_size=0.3, stratify=Y, random_state=42)
# get 10% of valid data as test
X_valid, X_test, Y_valid, Y_test = train_test_split( X_valid_test, Y_valid_test, test_size=0.1, stratify=Y_valid_test, random_state=42)

# Undersampling
print("\nClass distribution before resampling (total dataset):", Counter(Y))
undersampler = RandomUnderSampler(random_state=42)
x_train_resampled, y_train_resampled = undersampler.fit_resample(X_train, Y_train)
```

Figure 32: Data Preparation

The text data was tokenized and padded to the maximum length during the data preparation stage. The dataset was then split into 70% for training, 20% for validation, and 10% for testing and undersampling the train set due to the dataset is seriously unbalanced.

```
Class distribution before resampling (total dataset): Counter({0: 143076, 1: 15091})

X_train shape (21128, 200)
Y_train shape (21128,)
X_valid shape (42705, 200)
Y_valid shape (42705,)
X_test shape (4746, 200)
Y_test shape (4746,)

Class distribution after resampling (80% Train Set): Counter({0: 10564, 1: 10564})
Class distribution (10% Validation Set): Counter({0: 38631, 1: 4074})
Class distribution (10% Test Set): Counter({0: 4293, 1: 453})

Total number of training samples after resampling: 21128
```

Figure 33: Output of Data Preparation

After undersampling, total have 21128 rows for training, 42705 rows for validation, 4746 rows for testing.

4.0 Evaluation Metrics Selection and Explanation

```
train_y_pred_prob = model.predict(x_train_resampled)
y_pred_prob = model.predict(X_valid)

train_y_pred = (train_y_pred_prob > 0.5).astype(int)
valid_accuracy = accuracy_score(y_train_resampled, train_y_pred)
print(f"📌 Train Accuracy: {valid_accuracy:.4f}")

y_pred = (y_pred_prob > 0.5).astype(int)
valid_accuracy = accuracy_score(Y_valid, y_pred)
print(f"📌 Validation Accuracy: {valid_accuracy:.4f}")

f1_weighted = f1_score(Y_valid, y_pred, average='weighted')
print(f"\n📌 Weighted F1 Score: {f1_weighted:.4f}")

precision = precision_score(Y_valid, y_pred, average='macro')
recall = recall_score(Y_valid, y_pred, average='macro')

print(f"📌 Precision: {precision:.4f}")
print(f"📌 Recall: {recall:.4f}")

fpr, tpr, thresholds = roc_curve(Y_valid, y_pred_prob)
roc_auc = auc(fpr, tpr)
print(f"\n📌 ROC AUC: {roc_auc:.4f}")

precision, recall, _ = precision_recall_curve(Y_valid, y_pred_prob)
pr_auc = average_precision_score(Y_valid, y_pred_prob)
print(f"\n📌 Precision-Recall AUC: {pr_auc:.4f}")

print("\n📌 Classification Report:")
print(classification_report(Y_valid, y_pred))

print("\n📌 Confusion Matrix:")
print(confusion_matrix(Y_valid, y_pred))
```

Figure 34: Model Evaluation Code

Model evaluation metrics using F1-score, binary-precision, binary-recall, ROC-AUC score, PR-AUC score, train and validation accuracy. The F1 Score, which is the mean of precision and recall, serves as a balanced measure of model performance. High F1 Score indicates that the model achieves a good trade-off between avoiding false positives and minimizing false

negatives. The ROC-AUC score is to show the model's overall performance. The classification report gives detailed performance results for each class, helping to understand whether the model is making fair and accurate predictions across all categories. The evaluation threshold was set to 0.5, predictions probability above 0.5 were classified as positive (toxic) and those below as negative (non-toxic).

Confusion Matrix

```
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix

cm = confusion_matrix(Y_valid, y_pred)

sns.heatmap(cm, annot=True, fmt="d",cmap = sns.color_palette("ch:start=1,rot=-.5"),
            xticklabels=['0: Non-Toxic', '1: Toxic'],
            yticklabels=['0: Non-Toxic', '1: Toxic'])


plt.xlabel("Predicted Label")
plt.ylabel("Actual Label")
plt.title("Confusion Matrix")
plt.show()
```

Figure 35: Confusion Matrix Code

Confusion matrix is to evaluate the model performs by showing the number of correct and incorrect in predictions for each class.

- True Positives (TP) is correctly predicted in positive cases.
- True Negatives (TN) is correctly predicted in negative cases.
- False Positives (FP) is incorrectly predicted in positive cases.
- False Negatives (FN) is incorrectly predicted in negative cases.

When precision is low, usually false positive rate will be high, when recall is high, suggests that false negative rate is low. Precision and recall are closely related. A good model typically has high values for both true positives and true negatives in the confusion matrix, indicating strong overall performance.

ROC Curve

```
fpr, tpr, thresholds = roc_curve(Y_valid, y_pred_prob)
roc_auc = auc(fpr, tpr)

print(f"\n📌 ROC AUC: {roc_auc:.4f}")

plt.figure()

plt.plot(fpr, tpr, color='blue', lw=2, label=f'ROC Curve (AUC = {roc_auc:.4f})')
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--',label='Random Guess')

plt.xlim([-0.01, 1.0])
plt.ylim([0.0, 1.02])

plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')

plt.title('Receiver Operating Characteristic (ROC)')
plt.legend(loc="lower right")
plt.show()
```

Figure 36: ROC-AUC Curve Code

The ROC curve tests the TP rate and FP rate at different thresholds to see the model's ability to distinguish categories. The model performs better overall at differentiating between classes if the curve is closer to the top left corner and the area under the ROC curve (AUC) is higher. Conversely, the closer the model is to the baseline, the weaker the overall prediction ability of the model.

Precision-Recall Curve

```
import matplotlib.pyplot as plt
from sklearn.metrics import precision_recall_curve, average_precision_score

precision, recall, _ = precision_recall_curve(Y_valid, y_pred_prob)
pr_auc = average_precision_score(Y_valid, y_pred_prob)

print(f"\n📌 Precision-Recall AUC: {pr_auc:.4f}")

plt.figure()
plt.plot(recall, precision, color='blue', lw=2, label=f'PR Curve (AUC = {pr_auc:.4f})')

baseline = sum(Y_valid) / len(Y_valid)
plt.plot([0, 1], [baseline, baseline], color='navy', lw=2, linestyle='--', label='Random Guess')

plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.02])

plt.xlabel('Recall')
plt.ylabel('Precision')

plt.title('Precision-Recall Curve')
plt.legend(loc="upper right")
plt.show()
```

Figure 37: Precision-Recall Curve Code

It is not suitable to use ROC-AUC score to judge the overall performance of the model in data imbalance because the ROC-AUC score may be affected by the majority class (non-toxic comments). The model with good performance has very weak detection ability for the minority class (toxic comments). In data imbalance, more attention should be paid to PR-AUC score, because PR score only focuses on the accuracy and recall of the positive class (toxic comment),

which can more accurately reflect the detection ability of the model on the minority class (Richardson et al., 2024).

For example, suppose there are 20 examples of toxic comments and 10,000 examples of non-toxic comments in a dataset. Initially, if 50 examples of non-toxic comments are misclassified, the false positive rate is only $\frac{50}{(50+9950)} = 0.005$, which is a small change on the ROC curve. However, there will be a significant drop on the PR curve's $precision = \frac{20}{(20+50)} = 0.286$, so The PR curve provides a better evaluation of model performance when dealing with imbalanced data, as it focuses on the precision and recall of the minority class.

Precision and Recall Scores at Different Thresholds for Toxic Comments

```
precision, recall, thresholds = precision_recall_curve(Y_valid, y_pred_prob)

thresholds_full = np.append(thresholds, 1.0)

# Create interpolation points for thresholds from 0 to 1 with a step of 0.1
interp_thresholds = np.arange(0.0, 1.01, 0.1)
# Interpolate the precision and recall based on the new thresholds
precision_interp = np.interp(interp_thresholds, thresholds_full, precision)
recall_interp = np.interp(interp_thresholds, thresholds_full, recall)

plt.figure(figsize=(8, 6), dpi=100)
plt.plot(interp_thresholds, precision_interp, label="Precision", color='dodgerblue', linewidth=2, marker='o')
plt.plot(interp_thresholds, recall_interp, label="Recall", color='darkorange', linewidth=2, marker='s')

# Set the y-axis to percentage format
plt.gca().yaxis.set_major_formatter(mtick.PercentFormatter(1.0))
plt.gca().yaxis.set_major_locator(mtick.MultipleLocator(0.1))
# Set the x-axis scale value to the threshold
plt.xticks(interp_thresholds)

# Add the percentage text of precision and recall at each threshold position
for t, p, r in zip(interp_thresholds, precision_interp, recall_interp):
    plt.text(t, p, f'{p*100:.0f}%', fontsize=9, color='black', ha='center', va='bottom')
    plt.text(t, r, f'{r*100:.0f}%', fontsize=9, color='black', ha='center', va='top')

plt.xlabel("Threshold", fontsize=14, fontweight='bold', color='darkslategray')
plt.ylabel("Score (%)", fontsize=14, fontweight='bold', color='darkslategray')
plt.title("Positvie Class (Toxic Comments) Precision-Recall & Thresholds", fontsize=16, fontweight='bold', color='midnightblue')
plt.legend(loc='best', fontsize=12, frameon=False)
plt.grid(True, linestyle='--', alpha=0.7)
plt.tight_layout()
plt.show()
```

Figure 38: Precision and Recall Scores at Different Thresholds for Toxic Comments

By showing precision and recall at different thresholds, to evaluate the performance of the model under different decision conditions and know the threshold to adjust. For example, a higher threshold has higher precision (reduced false positives) but may reduce recall (missing negatives). A lower threshold may increase recall but may also increase false positives. This chart helps find the best threshold to balance precision and recall to optimize the overall performance of the model.

Learning Curve

```
accuracy = history.history['accuracy']
val_acc = history.history['val_accuracy']
loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(1, len(accuracy) + 1 )
sns.set_style("darkgrid")

fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(10, 4))

# Plotting the Accuracy Curve
axes[0].plot(epochs, accuracy, 'bo-', label='Train Acc', linewidth=2, markersize=5)
axes[0].plot(epochs, val_acc, 'rs-', label='Valid Acc', linewidth=2, markersize=5)
axes[0].set_xlabel('Epoch', fontsize=12)
axes[0].set_ylabel('Accuracy', fontsize=12)
axes[0].set_title('Model Accuracy', fontsize=14, fontweight='bold')
axes[0].legend(loc='upper left', fontsize=10)
axes[0].grid(True, linestyle='--', alpha=0.6)

for i in range(len(accuracy)):
    axes[0].text(epochs[i], accuracy[i], f"{accuracy[i]:.4f}", ha='right', va='bottom', fontsize=8, color='black')
    axes[0].text(epochs[i], val_acc[i], f"{val_acc[i]:.4f}", ha='right', va='top', fontsize=8, color='black')

# Plotting the Loss Curve
axes[1].plot(epochs, loss, 'bo-', label='Train Loss', linewidth=2, markersize=5)
axes[1].plot(epochs, val_loss, 'ro-', label='Valid Loss', linewidth=2, markersize=5)
axes[1].set_xlabel('Epoch', fontsize=12)
axes[1].set_ylabel('Loss', fontsize=12)
axes[1].set_title('Model Loss', fontsize=14, fontweight='bold')
axes[1].legend(loc='lower left', fontsize=10)
axes[1].grid(True, linestyle='--', alpha=0.6)

for i in range(len(loss)):
    axes[1].text(epochs[i], loss[i], f"{loss[i]:.4f}", ha='right', va='bottom', fontsize=8, color='black')
    axes[1].text(epochs[i], val_loss[i], f"{val_loss[i]:.4f}", ha='right', va='top', fontsize=8, color='black')

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

Figure 39: Learning Curve Code

The learning curve includes training and validation accuracy and loss during model training. These metrics help identify whether the model is underfitting or overfitting, and reveal how well the model is converging over each epoch.

5.0 Model Building 1: Bi-LSTM Binary Classification Model

Justification for Selecting Bi-LSTM Model:

The reason for choosing Bi-LSTM is that it is mentioned in many research papers that the Bi-The LSTM model does well in text classification tasks and is capable of capturing contextual information and long-range relationships in sequence data. Unlike traditional RNNs, Bi-LSTM solves the gradient vanishing problem through its gating mechanism (Shinde et al., 2024). This makes it very suitable for processing complex language patterns, different sentence lengths, and contextual nuances commonly seen in natural language.

5.1 Bi-LSTM Hyperparameter Explanation

Table 5: Bi-LSTM Hyperparameter Explanation

Hyperparameter	Explanation
Units	Increasing the number of units in a model can help capture more complex patterns, but it also raises the risk of overfitting. While there is no default value, but have several commonly value such as 16, 32, 64, 128, 256, or 512 units are often selected based on the task complexity and the size of the dataset (Tian et al., 2020).
Optimizer	Determines how the model updates weights during training. The choice of optimizer affects the convergence speed and the quality of the final model. According to (Choi et al., 2020), optimizers like NAdam, RMSProp and Adam achieve faster and better results across most tasks. In fact, Adam and NAdam were shown to train models about 60% faster than Momentum. While Momentum and SGD optimizers can be more stable during hyperparameter tuning but slow convergence speed, typically suitable for simpler tasks or when computational resources are limited.
Activation	Activation functions are used to introduce non-linearity into neural networks, allowing them to learn complex patterns and relationships in data. This paper (Rodríguez & Buitrago, 2022) compares the three most used hidden layer activation functions in deep learning: ReLU, Sigmoid, and Tanh. ReLU is currently the most recommended choice, especially in MLP and convolutional neural networks (CNN). It not

	only has fast calculation speed, but also effectively alleviates the gradient vanishing problem, making model training more stable. Sigmoid and Tanh are prone to gradient vanishing problems in deep networks.
L2 regularization	By introducing the square sum penalty term of the weight in the loss function, the growth of model parameters can be effectively constrained, preventing the weights from learning too large values and thus reducing overfitting. Make the model simpler, more stable, and easier to adapt to new data that has not been seen, especially in large dataset.
Dropout Rate	During training, certain neurons are randomly dropped to avoid overfitting. This allows the model to not rely on certain specific neuron features to improve the model's ability to generalize on unseen data.
Learning Rate	The learning rate determines how big the "step" of each weight update of the model is during the training process. Although a step that is too large (high learning rate) allows the model to converge quickly, it may also miss the best results. A step that is too small (low learning rate) allows for more stable convergence, but it may take longer time.
Batch Size	Establishes how many samples are processed prior to weight updates. Larger batch sizes provide stable gradients but require more memory. According to (Akpatsa et al., 2022), commonly used batch sizes include 16, 32, 64, 128, 256, and 512.
Epochs	Epochs decide how many times the model looks at the whole training set. If there are too less, the model might not learn enough. But if there are too many, it could overfit, due to model basically memorizing the training data and have generalization issue on new inputs. Early stopping function are useful for model stop training when model starting overfitting.

5.2 Bi-LSTM Basic Model

Model Building Using Default or Common Value

```
MAX_SEQUENCE_LENGTH = 200
EMBEDDING_DIM = 300
MAX_VOCAB_SIZE = 20000

# https://www.tensorflow.org/api_docs/python/tf/keras/layers/LSTM
model = Sequential([
    Embedding(input_dim=MAX_VOCAB_SIZE, output_dim=EMBEDDING_DIM, input_length=MAX_SEQUENCE_LENGTH),
    Bidirectional(LSTM(32, return_sequences=True)),
    Bidirectional(LSTM(64, return_sequences=False)),  # by default: activation='tanh', recurrent_activation='sigmoid',return_sequences=False)
    Dense(128, activation='tanh') ,
    Dense(1, activation='sigmoid')
])

model.compile(
    loss="binary_crossentropy",
    optimizer=Adam(), # Default LR: 0.001
    metrics=["accuracy"]
)

history = model.fit(
    x_train_resampled,
    y_train_resampled,
    epochs=10,
    batch_size=32,
    validation_data=(X_valid, Y_valid),
    verbose=1
)
model.summary()
```

Figure 40: Bi-LSTM Basic Model

```
Epoch 1/10
c:\Users\Asus\AppData\Local\Programs\Python\Python312\Lib\site-packages\keras\src\layers\core\embedding.py:90: UserWarning: Argument `input_length` is deprecated. Just remove it.
  warnings.warn(
661/661 ━━━━━━━━━━━━━━━━━━━━ 159s 233ms/step - accuracy: 0.8154 - loss: 0.3894 - val_accuracy: 0.9125 - val_loss: 0.2327
Epoch 2/10
661/661 ━━━━━━━━━━━━━━━━━━━━ 149s 226ms/step - accuracy: 0.9424 - loss: 0.1534 - val_accuracy: 0.9148 - val_loss: 0.2377
Epoch 3/10
661/661 ━━━━━━━━━━━━━━━━━━━━ 141s 213ms/step - accuracy: 0.9675 - loss: 0.0945 - val_accuracy: 0.9084 - val_loss: 0.2507
Epoch 4/10
661/661 ━━━━━━━━━━━━━━━━━━━━ 133s 202ms/step - accuracy: 0.9808 - loss: 0.0548 - val_accuracy: 0.8961 - val_loss: 0.3038
Epoch 5/10
661/661 ━━━━━━━━━━━━━━━━━━━━ 139s 210ms/step - accuracy: 0.9875 - loss: 0.0356 - val_accuracy: 0.8833 - val_loss: 0.4510
Epoch 6/10
661/661 ━━━━━━━━━━━━━━━━━━━━ 147s 222ms/step - accuracy: 0.9905 - loss: 0.0264 - val_accuracy: 0.8786 - val_loss: 0.4914
Epoch 7/10
661/661 ━━━━━━━━━━━━━━━━━━━━ 151s 229ms/step - accuracy: 0.9908 - loss: 0.0244 - val_accuracy: 0.8820 - val_loss: 0.4985
Epoch 8/10
661/661 ━━━━━━━━━━━━━━━━━━━━ 147s 222ms/step - accuracy: 0.9950 - loss: 0.0151 - val_accuracy: 0.8595 - val_loss: 0.6548
Epoch 9/10
661/661 ━━━━━━━━━━━━━━━━━━━━ 149s 225ms/step - accuracy: 0.9931 - loss: 0.0171 - val_accuracy: 0.8649 - val_loss: 0.6543
Epoch 10/10
661/661 ━━━━━━━━━━━━━━━━━━━━ 152s 229ms/step - accuracy: 0.9960 - loss: 0.0117 - val_accuracy: 0.8919 - val_loss: 0.5834
```

Figure 41: Bi-LSTM Basic Model Training Epochs

Layer (type)	Output Shape	Param #
embedding (Embedding)	(None, 200, 300)	6,000,000
bidirectional (Bidirectional)	(None, 200, 64)	85,248
bidirectional_1 (Bidirectional)	(None, 128)	66,048
dense (Dense)	(None, 128)	16,512
dense_1 (Dense)	(None, 1)	129

Total params: 18,503,813 (70.59 MB)

Trainable params: 6,167,937 (23.53 MB)

Non-trainable params: 0 (0.00 B)

Optimizer params: 12,335,876 (47.06 MB)

Figure 42: Output of Bi-LSTM Basic Model Summary

The model is built with one input embedding layer, followed by three hidden layers including two Bi-LSTM layers and one fully connected dense layer. Each hidden layer uses the

commonly used number of units suggested by (Tian et al., 2020) and uses the default activation function “tanh”. For binary classification tasks, the sigmoid is used as the activation function in the final output layer. During training, the model runs for 10 epochs with a batch size of 32. Adam is the optimizer, and 0.001 is the learning rate by default.

5.2.1 Bi-LSTM Basic Model Evaluation

Evaluation Metrics

```
📌 Train Accuracy: 0.9973
📌 Validation Accuracy: 0.8919

📌 Weighted F1 Score: 0.9060
📌 Precision: 0.7257
📌 Recall: 0.8882

📌 ROC AUC: 0.9477

📌 Classification Report:
              precision    recall  f1-score   support

           0       0.99      0.89      0.94     38631
           1       0.46      0.88      0.61      4074

    accuracy                           0.89     42705
   macro avg       0.73      0.89      0.77     42705
weighted avg       0.94      0.89      0.91     42705
```

Figure 43: Bi-LSTM Model Evaluation

The model achieved a high training accuracy of 99.73% and validation accuracy of 89.19%, weighted F1 score of 90.60% in threshold 0.5. Overall, the performance is good with a ROC-AUC score of 94.77%. However, for class 1 (toxic comments) precision is low just 46%. Even after adjusting the decision threshold to 0.6 or 0.7, the precision only improved slightly by about 0.1, which suggests the model struggles to correctly identify toxic comments without also misclassifying many non-toxic.

Confusion Matrix

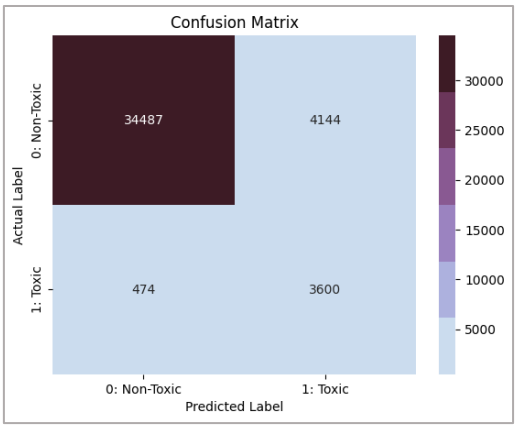


Figure 44: Bi-LSTM Confusion Matrix

The confusion matrix shows the model correctly classified 34487 rows of data are non-toxic comments and 3600 rows of data are toxic comment. The model misclassified 4144 rows of non-toxic as toxic comments (false positive) and 474 rows of toxic comments as non-toxic (false negative). This model has a false positive rate is ($\frac{4144}{4144+34487} * 100$) equal to 10.73%. The false negative rate is ($\frac{474}{474+3600} * 100$) equal to 11.63%.

ROC-AUC Curve and Precision-Recall Curve

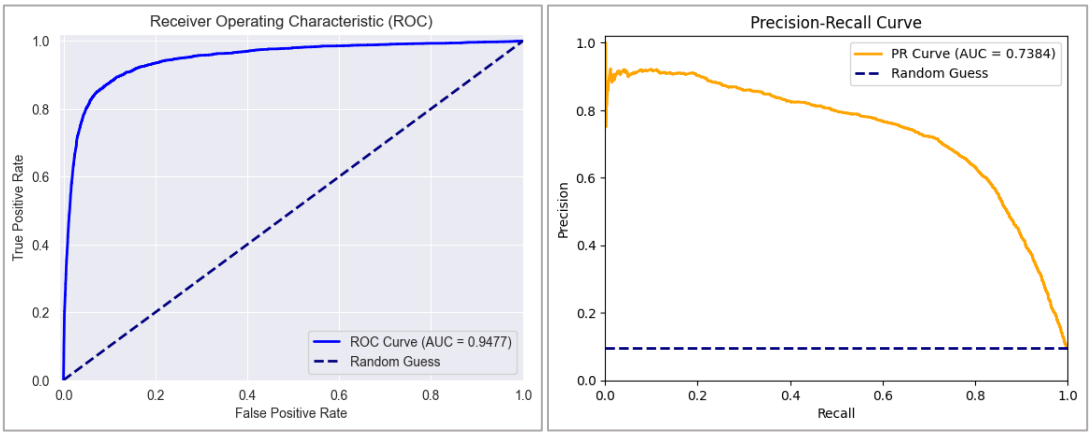


Figure 45: Bi-LSTM ROC Curve (left) and PR Curve (right)

The model's ROC-AUC score is 94.77% has a good overall performance, but the PR-AUC score is 73.84%, which is lower than 21% of the ROC-AUC score, indicating ROC-AUC score is biased towards non-toxic comments due to class imbalance. The ROC curve remains high even with some false positives, while the PR curve shows precision drop when recall increases.

Learning Rate

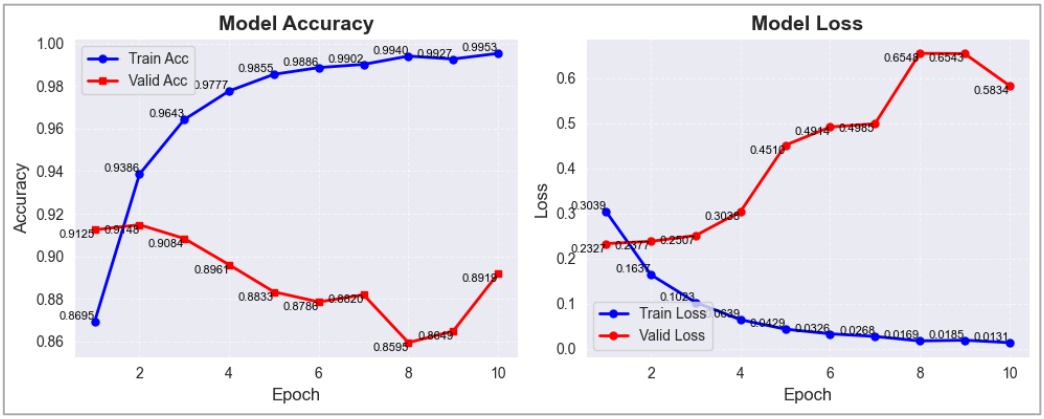


Figure 46: Bi-LSTM Learning Rate

This model shows clear signs of overfitting. Starting from the second epoch, the training accuracy keeps increasing and the training loss continues to decrease, while at the same time

the validation accuracy starts to decrease, and the validation loss begins to rise. This is a classic indication of overfitting. Using the default settings, without using techniques such as L2 regularization, dropout or early stops, is more prone to overfitting, especially in imbalanced datasets.

5.3 Bi-LSTM Hyperparameter Tuning

```
MAX_SEQUENCE_LENGTH = 200
EMBEDDING_DIM = 100
MAX_VOCAB_SIZE = 20000

def build_model(hp):
    model = Sequential()
    model.add(Embedding(input_dim=MAX_VOCAB_SIZE, output_dim=EMBEDDING_DIM, input_length=MAX_SEQUENCE_LENGTH))

    model.add(Bidirectional(LSTM(units=hp.Int("LSTM_layer_1", min_value=64, max_value=256, step=64),return_sequences=True)))
    model.add(Dropout(hp.Float("dropout_1", min_value=0.2, max_value=0.4, step=0.1)))

    model.add(Bidirectional(LSTM(units=hp.Int("LSTM_layer_2", min_value=32, max_value=128, step=32), return_sequences=True )))
    model.add(Dropout(hp.Float("dropout_2", min_value=0.2, max_value=0.4, step=0.1)))

    model.add(Bidirectional(LSTM(units=hp.Int("LSTM_layer_3", min_value=16, max_value=64, step=16), return_sequences=False )))
    model.add(Dropout(hp.Float("dropout_3", min_value=0.2, max_value=0.4, step=0.1)))

    model.add(Dense(units=hp.Int("dense_units", min_value=16, max_value=64, step=16),
                    activation=hp.Choice("activation", ["relu"]),
                    kernel_regularizer=tf.keras.regularizers.l2(hp.Float("l2_reg", min_value=0.001, max_value=0.1, step=0.01))))
    model.add(Dense(1, activation="sigmoid"))

    optimizer_choice = hp.Choice('optimizer', ['adam'])
    learning_rate = hp.Choice("learning_rate", [0.0001, 0.003, 0.01, 0.3])

    if optimizer_choice == 'adam':
        optimizer = Adam(learning_rate=learning_rate)

    model.compile(
        optimizer = optimizer,
        loss="binary_crossentropy",
        metrics=["accuracy"]
    )
    return model
```

```
tuner = BayesianOptimization(
    build_model,
    objective='val_accuracy',
    max_trials=5,
    executions_per_trial=1,
    directory='my_dir_Bi_LSTM',
    project_name='2_HP_ReleAdam_NoFastText'
)

tuner.search(x_train_resampled,y_train_resampled,
             epochs=5,
             batch_size=32,
             validation_data=(X_valid,Y_valid))
```

Figure 47: Bi-LSTM Hyperparameter Tuning Model Building

```
Trial 5 Complete [01h 14m 52s]
val_accuracy: 0.9046013355255127

Best val_accuracy So Far: 0.9372204542160034
Total elapsed time: 05h 55m 32s
```

Figure 48: Output of Bi-LSTM Hyperparameter Tuning

Table 6: Bi-LSTM Justification of Hyperparameter Setting

Hyperparameter	Range/Value	Justification of Hyperparameter Setting
Bi-LSTM Units	16-256	The flexible range provided by the number of units in the Bi-LSTM layer (Tian et al., 2020) enables the model to learn different levels of sequence feature complexity according to task requirements, thereby improving the ability to understand text structure.
Dropout	0.2-0.4	Dropout common range is 0.2-0.4, setting too large will lead to model underfitting. The dropout layer is commonly used to balance regularization avoid overfitting.
Dense Units	16-64	The role of the Dense layer is to integrate step information output all the time by the Bi-LSTM through a fully connected structure and make the final prediction in the connected output layer. If the Dense layer is omitted and the output of the last time step of the LSTM is directly used for prediction, the model may be underfitted because it lacks sufficient nonlinear transformation capabilities and model cannot fully learn the relationship between complex features.
Activation	ReLU	The main reason for choosing the ReLU activation function is that according to the research paper (Szandała, 2020), ReLU can effectively alleviate the gradient vanishing problem. Secondly, compared with other activation functions such as Sigmoid, Tanh, Swish and Leaky ReLU, ReLu has faster training speed and convergence efficiency. The RNN model needs to capture long-distance dependencies cause training time is longer. Choosing the more computationally efficient ReLU activation function can improve the overall training efficiency.
Optimizer	Adam	Adam is regarded as the most efficient optimizer out of all the choices that were examined, providing the

		optimum trade-off between model performance and convergence speed across a variety of deep learning tasks (Jena et al., 2021).
l2_reg (L2 regularization)	0.001-0.1	Adding L2 regularization helps prevent overfitting by discouraging large weight values.
Learning Rate (LR)	0.0001, 0.003, 0.01, 0.3	Multiple learning rates are tested to identify the optimal value. Standard learning rate value is between 0.001 to 0.1 (B.G.M. Vandeginste et al., 1998, pp. 649–699). According to the (Jena et al., 2021), the most used learning rate in deep learning studies is 0.001, adopted in 28% of the study, followed by 0.0001 (used in 13%) and 0.01 (used in 7% of the studies).
Epochs	5	Epochs are limited to 5 based on experimental observations, after around 4 to 5 epochs, the model starts to show signs of overfitting. A smaller number of epochs helps speed up training while still allowing the model to learn effectively.
Max Trials	5	Max Trials is set to 5 because the optimizer and activation function have already been fixed, which reduces the number of parameters to tune and allows for a more focused hyperparameter search.
Batch Size	32	Batch Size is set to 32, as the training dataset contains 21128 rows, which is not very huge dataset. For larger datasets, batch sizes can be set as 64 or 128, more appropriate to optimize training efficiency.

Justification of Hyperparameter Tuning Technique Selected "BayesianOptimization"

In this study, hyperparameter tuning using Bayesian Optimization method is because according to (Rimal et al., 2024) Bayesian Optimization is particularly suitable for dealing with hyperparameter optimization problems with high dimensions, continuous search space and large range. Taking the Bi-LSTM model as an example, the model contains 9 hyperparameters to be tuned, if using Grid Search method, each hyperparameter has 3 values need to tune, the model will try 3^9=19683 times of combinations, which require a lot of training time.

Bayesian optimization is different from random search. When searching for hyperparameters, Bayesian optimization adjusts the next search direction according to the results of previous attempts. Random search uses a probability model to select possible better parameter combinations. It blindly tries new combinations without referring to the results of previous attempts. This method waste resources in multiple attempts. For example, when performing 10 trials, half of the parameter combinations may work poorly, resulting in inefficient model training and unstable performance, while also wasting a lot of time and computing resources.

Best Hyperparameter

```
best_model = tuner.get_best_hyperparameters(num_trials=1)[0]
print(f"""Best Hyperparameter:
        Bi-LSTM  Layer 1: {best_model.get('LSTM_layer_1')}
        Bi-LSTM  Layer 2: {best_model.get('LSTM_layer_2')}
        Bi-LSTM  Layer 3: {best_model.get('LSTM_layer_3')}
        Dropout 1: {best_model.get('dropout_1')}
        Dropout 2: {best_model.get('dropout_2')}
        Dropout 3: {best_model.get('dropout_3')}
        Dense : {best_model.get('dense_units')}
        L2 Regularizers: {best_model.get('l2_reg')}
        Activitation : {best_model.get('activation')}
        Learning Rate : {best_model.get('learning_rate')}
        Optimizer Choice: {best_model.get('optimizer')}
        """)
```

Figure 49: Bi-LSTM Get Best Hyperparameter Code

```
Best Hyperparameter:
        Bi-LSTM  Layer 1: 256
        Bi-LSTM  Layer 2: 64
        Bi-LSTM  Layer 3: 16
        Dropout 1: 0.2
        Dropout 2: 0.30000000000000004
        Dropout 3: 0.30000000000000004
        Dense : 32
        L2 Regularizers: 0.031
        Activitation : relu
        Learning Rate : 0.0001
        Optimizer Choice: adam
```

Figure 50: Bi-LSTM Best Hyperparameter

Through the hyperparameter tuning, the first Bi-LSTM layer consists of 256 units with a dropout rate of 0.2, followed by a second Bi-LSTM layer with 64 units and a dropout rate of 0.3. The third Bi-LSTM layer has 16 units with a dropout rate of 0.3. The fully connected dense layer consists of 32 units, utilizing the ReLu activation and an L2 regularization of 0.031. The model uses the Adam optimizer with a tuned learning rate of 0.0001.

Model Building with the Best Hyperparameter

```
early_stopping = EarlyStopping(monitor="val_loss", patience=3, restore_best_weights=True)

best_model = tuner.get_best_hyperparameters(num_trials=1)[0]
model = build_model(best_model)
history = model.fit(x_train_resampled, y_train_resampled,
                    epochs=15, batch_size=32,
                    validation_data=(X_valid, Y_valid),
                    callbacks=[early_stopping]
                   )
model.summary()
```

Figure 51: Build Model Using Best Hyperparameter

```
Epoch 1/15
661/661 ━━━━━━━━━━━━━━━━━━━━ 1203s 2s/step - accuracy: 0.6529 - loss: 1.4541 - val_accuracy: 0.9000 - val_loss: 0.8451
Epoch 2/15
661/661 ━━━━━━━━━━━━━━━━━━━━ 1191s 2s/step - accuracy: 0.8423 - loss: 0.8587 - val_accuracy: 0.8635 - val_loss: 0.6770
Epoch 3/15
661/661 ━━━━━━━━━━━━━━━━━━━━ 1245s 2s/step - accuracy: 0.9282 - loss: 0.4869 - val_accuracy: 0.8677 - val_loss: 0.5131
Epoch 4/15
661/661 ━━━━━━━━━━━━━━━━━━━━ 1244s 2s/step - accuracy: 0.9475 - loss: 0.3135 - val_accuracy: 0.8835 - val_loss: 0.3967
Epoch 5/15
661/661 ━━━━━━━━━━━━━━━━━━━━ 1226s 2s/step - accuracy: 0.9647 - loss: 0.2055 - val_accuracy: 0.8932 - val_loss: 0.3582
Epoch 6/15
661/661 ━━━━━━━━━━━━━━━━━━━━ 1132s 2s/step - accuracy: 0.9732 - loss: 0.1487 - val_accuracy: 0.8663 - val_loss: 0.4014
Epoch 7/15
661/661 ━━━━━━━━━━━━━━━━━━━━ 1221s 2s/step - accuracy: 0.9758 - loss: 0.1267 - val_accuracy: 0.8827 - val_loss: 0.3777
Epoch 8/15
661/661 ━━━━━━━━━━━━━━━━━━━━ 1674s 3s/step - accuracy: 0.9793 - loss: 0.1045 - val_accuracy: 0.8724 - val_loss: 0.4263
```

Figure 52: Bi-LSTM Best Hyperparameter Model Training Epochs

Layer (type)	Output Shape	Param #
embedding_1 (Embedding)	(None, 200, 100)	2,000,000
bidirectional_3 (Bidirectional)	(None, 200, 512)	731,136
dropout_3 (Dropout)	(None, 200, 512)	0
bidirectional_4 (Bidirectional)	(None, 200, 128)	295,424
dropout_4 (Dropout)	(None, 200, 128)	0
bidirectional_5 (Bidirectional)	(None, 32)	18,560
dropout_5 (Dropout)	(None, 32)	0
dense_2 (Dense)	(None, 32)	1,056
dense_3 (Dense)	(None, 1)	33

Total params: 9,138,629 (34.86 MB)

Trainable params: 3,046,209 (11.62 MB)

Non-trainable params: 0 (0.00 B)

Optimizer params: 6,092,420 (23.24 MB)

Figure 53: Output of Bi-LSTM Best Hyperparameter Model Summary

With a batch size of 32, the model was trained across 15 epochs. To monitor validation loss and avoid overfitting, the early stopping function was used with patience 3.

5.3.1 Bi-LSTM Hyperparameter Tuning Model Evaluation

```
📌 Train Accuracy: 0.9764
📌 Validation Accuracy: 0.8932

📌 Weighted F1 Score: 0.9072
📌 Precision: 0.7281
📌 Recall: 0.8937

📌 ROC AUC: 0.9554

📌 Classification Report:
              precision    recall  f1-score   support

           0       0.99      0.89      0.94     38631
           1       0.47      0.89      0.61      4074

    accuracy                           0.89     42705
   macro avg       0.73      0.89      0.78     42705
weighted avg       0.94      0.89      0.91     42705
```

Figure 54: Bi-LSTM HP Model Evaluation

Threshold 0.5, hyperparameter tuning model training accuracy of 97.64% and validation accuracy of 89.32%. Weighted F1 score is 90.72% and ROC-AUC score of 95.54% shows that overall performance is good.

However, similar to earlier Bi-LSTM basic model observations, precision for class 1 (toxic comments) remains relatively low at 47%, despite a high recall of 89%, the model still struggles with precision-specific optimization for class 1.

Confusion Matrix

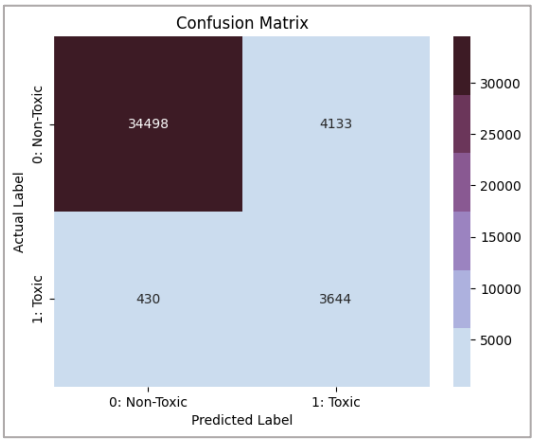


Figure 55: Bi-LSTM HP Confusion Matrix

Bi-LSTM hyperparameter tuning model correctly classified 34498 rows of data are non-toxic comments and 3644 rows of toxic comments. False positive rate is 10.70% and false negative rate is 10.55%.

ROC Curve and PR Curve

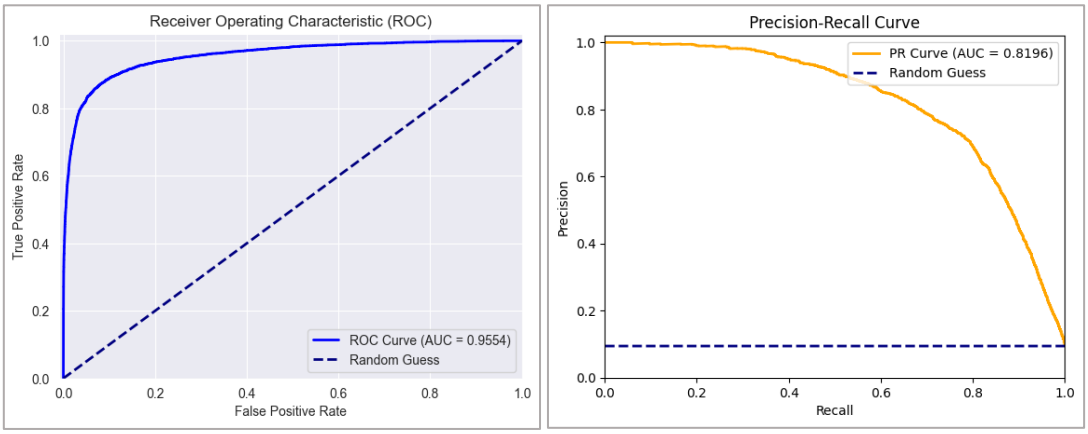
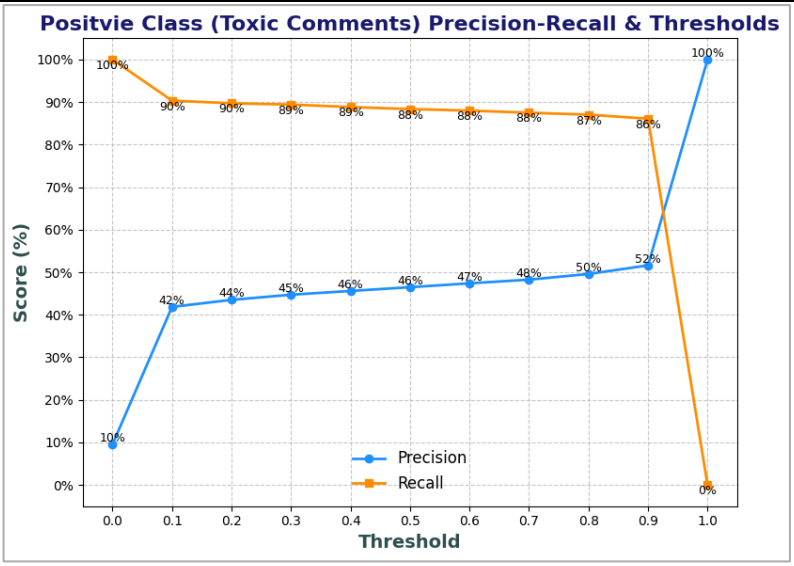


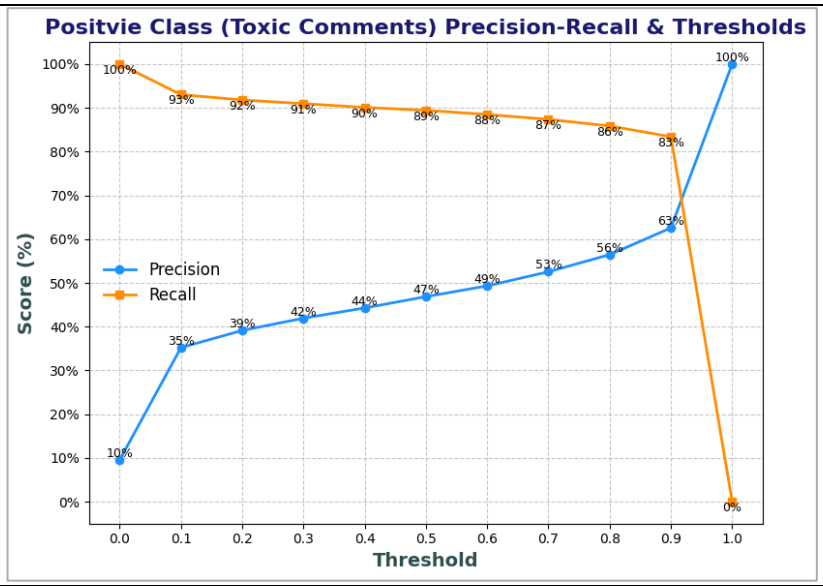
Figure 56: Bi-LSTM HP Model ROC Curve (left) and PR Curve (right)

After hyperparameter tuning, the ROC-AUC score reached 0.9554 and the PR-AUC score reached 0.8196, which is 8% higher than the basic model. This shows that the overall performance of the model is better than the basic model, with ROC-AUC and PR-AUC scores exceeding 80%. However, the precision of class 1 is relatively low (47%), which indicates that the default threshold of 0.5 may not be the best choice. Can adjust to the higher threshold to reduce the false positive rate.

The table below show the result of Bi-LSTM model and Bi-LSTM Hyperparameter Tuning Model, compares how different threshold values affect the precision and recall for class 1 (toxic comments).

Table 7:Correlation Between Precision-Recall Scores and Threshold Values

	Precision-Recall and Threshold
Basic Model	

Hyperparameter Tuning	

For Bi-LSTM Basic Model:

Class 1 **Precision 46%** (threshold 0.5), PR-AUC (73.84%): Both precision and PR-AUC are low, it means that the model does not perform well in class 1. Even after adjusting the threshold to 0.9, the precision only **reaches 52%.** The precision for class 1 remains low, and the false positive rate is still high, indicating that need to improve the model architecture.

For Bi-LSTM Hyperparameter Tuning Model:

Class 1 **precision 47%** (threshold 0.5), PR-AUC (81.96%): Low precision but high PR-AUC does indicate that the default threshold of 0.5 may not be optimal, and the threshold needs to be adjusted to improve the precision of class 1. From the threshold diagram, adjust threshold to 0.9 can improve the **precision of class 1 to 63%**.

Table 8: Bi-LSTM HP Model Compare with Different Threshold

Hyperparameter Tuning Model Evaluation Metrics	Thresholds: 0.5	Thresholds: 0.9
Train Accuracy	0.9764	0.9580
Validation Accuracy	0.8932	0.9365
Weighted F1 Score	0.9072	0.9405
Precision	0.7281	0.8037

Recall	0.8937	0.8906
ROC-AUC Score	0.9554	0.9554
PR-AUC Score	0.8196	0.8196
False Positive Rate	10.70%	5.26%
False Negative Rate	10.55%	16.62%

By adjusting the decision threshold of the hyperparameter-tuned model from 0.5 to 0.9, the validation set accuracy increased by 4%, the weighted F1 score increased by 3%, and the precision increased from 72.81% to 80.37%, reducing the false positive rate by 5%. However, this also brought a trade-off, with the recall rate decreasing by 0.31% and the false negative rate increasing by 6.07%.

Learning Curve

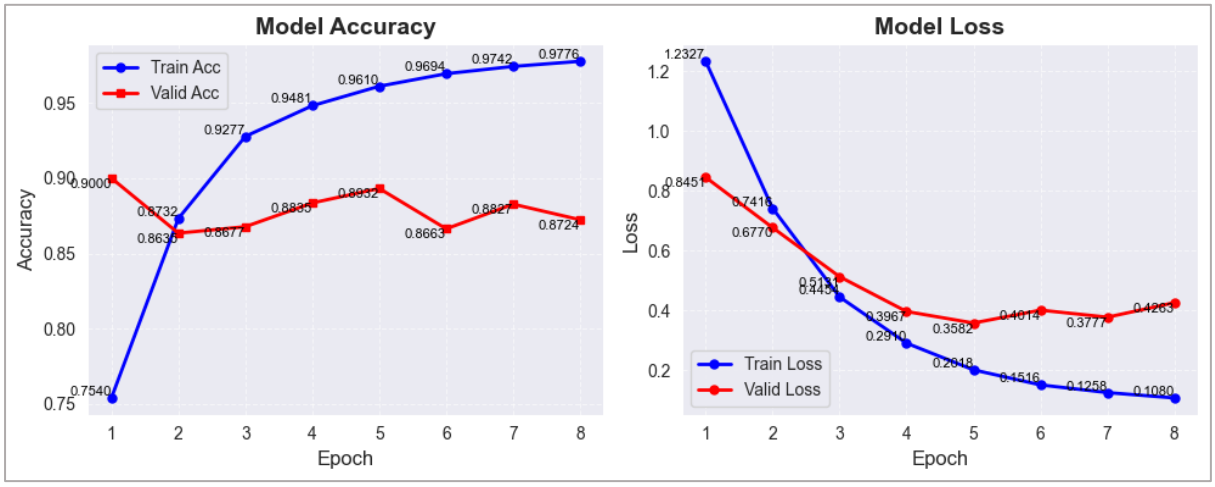


Figure 57: Bi-LSTM HP Model Learning Curve

Although early stopping was used in the hyperparameter tuning model and training was stopped early at the 5th epoch, the model still suffered from overfitting. In the 5th epoch, the accuracy of the training set differed from that of the validation set by 6.78%, which was a large difference. This indicated that the model overlearned on the training set and had insufficient generalization ability on the validation set.

5.4 Bi-LSTM Fine-Tuning

```
MAX_SEQUENCE_LENGTH = 200
EMBEDDING_DIM = 300
MAX_VOCAB_SIZE = 20000

# https://fasttext.cc/docs/en/english-vectors.html
fasttext_path = "crawl-300d-2M-subword.bin"
word_vectors = FastText.load_fasttext_format(fasttext_path)

# declare matrix
embedding_matrix = np.zeros((MAX_VOCAB_SIZE, EMBEDDING_DIM))

for word, i in tokenizer.word_index.items():
    if i < MAX_VOCAB_SIZE:
        if word in word_vectors.wv:
            embedding_matrix[i] = word_vectors.wv[word]
        else:
            embedding_matrix[i] = np.random.normal(size=(EMBEDDING_DIM,))

# embedding_matrix = np.load("embedding_matrix_20k_200.npy")

embedding_layer = Embedding(
    input_dim=MAX_VOCAB_SIZE,
    output_dim=EMBEDDING_DIM,
    weights=[embedding_matrix],
    input_length=MAX_SEQUENCE_LENGTH,
    trainable=False
)
# np.save("embedding_matrix_20k.npy", embedding_matrix)
```

Figure 58: FastText Word Embedding

```
# ------------------------------------------------------------------------------------------------------------------
# When saving the model, Keras will automatically recognize my_custom_loss and load it correctly
@register_keras_serializable(package="CustomLoss")
# Function focal_loss code et from GitHub (aldi-dimara, 2018)
# Gamma, Alpha default values recommended by the paper (Lin et al., 2018) # https://arxiv.org/pdf/1708.02002v2
def focal_loss(y_true, y_pred, gamma=2.0, alpha=0.45):
    epsilon = K.epsilon()
    y_pred = K.clip(y_pred, epsilon, 1.0-epsilon)
    # Predicted probability of the correct class
    pt = tf.where(K.equal(y_true, 1), y_pred, 1-y_pred)
    # Alpha factor for balancing class imbalance.
    alpha_factor = K.ones_like(y_true)*alpha
    # Dynamically assigns different alpha values to different classes
    alpha_t = tf.where(K.equal(y_true, 1), alpha_factor, 1-alpha_factor)
    # Cross-entropy loss
    cross_entropy = -K.log(pt)
    # Alpha > 0.5, then class 1 weigth > class 0 weigth. Alpha < 0.5, then class 0 weigth > class 1 weigth
    weight = alpha_t * K.pow((1-pt), gamma) #alpha_t*(1 - pt) ^ gamma
    loss = weight * cross_entropy
    # loss = K.sum(loss, axis=1)
    loss = K.mean(loss, axis=1)
    return loss
```

Figure 59: Focal Loss Function

```python
model = Sequential([
    embedding_layer,
    Bidirectional(LSTM(64, return_sequences=True)),
    Dropout(0.2),
    Bidirectional(LSTM(64, return_sequences=True)),
    Dropout(0.3),
    Bidirectional(LSTM(32, return_sequences=False)),
    Dropout(0.4),
    Dense(48, activation='relu', kernel_regularizer=tf.keras.regularizers.l2(0.001)),
    Dense(1, activation='sigmoid')
])
optimizer = Adam(learning_rate=0.003)

model.compile(
    loss=focal_loss,
    optimizer=optimizer,
    metrics=["accuracy"]
)

# Reduce learning rate when loss has stopped reducing
reduce_lr = ReduceLROnPlateau(
    monitor='val_loss',
    factor=0.5,         # Decrease LR (current LR * 0.5)
    patience=1,           # If val_loss does not improve for 1 epoch, reduce LR
    min_lr=0.0001,        # Setting minimum learning rate to prevent it from being too low
    verbose=1
)
early_stopping = EarlyStopping(monitor='val_loss', patience=3, restore_best_weights=True)

history = model.fit(
    x_train_resampled,
    y_train_resampled,
    epochs=15,
    batch_size=32,
    validation_data=(X_valid, Y_valid),
    callbacks=[early_stopping, reduce_lr],
    verbose=1
)
model.summary()
```

Figure 60: Bi-LSTM Fine-Tuning Model

```
Epoch 1/15
661/661 ━━━━━━━━━━━━━━━━━━━━ 182s 265ms/step - accuracy: 0.6699 - loss: 0.0860 - val_accuracy: 0.8304 - val_loss: 0.0653 - learning_rate: 0.0030
Epoch 2/15
661/661 ━━━━━━━━━━━━━━━━━━━━ 180s 273ms/step - accuracy: 0.7475 - loss: 0.0628 - val_accuracy: 0.8810 - val_loss: 0.0519 - learning_rate: 0.0030
Epoch 3/15
661/661 ━━━━━━━━━━━━━━━━━━━━ 180s 273ms/step - accuracy: 0.7845 - loss: 0.0562 - val_accuracy: 0.9158 - val_loss: 0.0411 - learning_rate: 0.0030
Epoch 4/15
661/661 ━━━━━━━━━━━━━━━━━━━━ 0s 156ms/step - accuracy: 0.8108 - loss: 0.0507
Epoch 4: ReduceLROnPlateau reducing learning rate to 0.0015000000013038516.
661/661 ━━━━━━━━━━━━━━━━━━━━ 181s 274ms/step - accuracy: 0.8108 - loss: 0.0507 - val_accuracy: 0.8373 - val_loss: 0.0523 - learning_rate: 0.0030
Epoch 5/15
661/661 ━━━━━━━━━━━━━━━━━━━━ 0s 159ms/step - accuracy: 0.8312 - loss: 0.0458
Epoch 5: ReduceLROnPlateau reducing learning rate to 0.000750000006519258.
661/661 ━━━━━━━━━━━━━━━━━━━━ 183s 277ms/step - accuracy: 0.8312 - loss: 0.0458 - val_accuracy: 0.7885 - val_loss: 0.0601 - learning_rate: 0.0015
Epoch 6/15
661/661 ━━━━━━━━━━━━━━━━━━━━ 186s 282ms/step - accuracy: 0.8462 - loss: 0.0431 - val_accuracy: 0.8985 - val_loss: 0.0409 - learning_rate: 7.5000e-04
Epoch 7/15
661/661 ━━━━━━━━━━━━━━━━━━━━ 0s 166ms/step - accuracy: 0.8538 - loss: 0.0413
Epoch 7: ReduceLROnPlateau reducing learning rate to 0.000375000003259629.
661/661 ━━━━━━━━━━━━━━━━━━━━ 189s 286ms/step - accuracy: 0.8538 - loss: 0.0413 - val_accuracy: 0.8915 - val_loss: 0.0421 - learning_rate: 7.5000e-04
Epoch 8/15
661/661 ━━━━━━━━━━━━━━━━━━━━ 186s 282ms/step - accuracy: 0.8607 - loss: 0.0402 - val_accuracy: 0.8950 - val_loss: 0.0404 - learning_rate: 3.7500e-04
Epoch 9/15
661/661 ━━━━━━━━━━━━━━━━━━━━ 185s 280ms/step - accuracy: 0.8693 - loss: 0.0382 - val_accuracy: 0.9041 - val_loss: 0.0391 - learning_rate: 3.7500e-04
Epoch 10/15
661/661 ━━━━━━━━━━━━━━━━━━━━ 187s 283ms/step - accuracy: 0.8677 - loss: 0.0383 - val_accuracy: 0.9146 - val_loss: 0.0356 - learning_rate: 3.7500e-04
Epoch 11/15
661/661 ━━━━━━━━━━━━━━━━━━━━ 0s 168ms/step - accuracy: 0.8689 - loss: 0.0380
Epoch 11: ReduceLROnPlateau reducing learning rate to 0.0001875000016298145.
661/661 ━━━━━━━━━━━━━━━━━━━━ 189s 285ms/step - accuracy: 0.8689 - loss: 0.0380 - val_accuracy: 0.8993 - val_loss: 0.0413 - learning_rate: 3.7500e-04
Epoch 12/15
661/661 ━━━━━━━━━━━━━━━━━━━━ 0s 168ms/step - accuracy: 0.8709 - loss: 0.0369
Epoch 12: ReduceLROnPlateau reducing learning rate to 0.0001.
661/661 ━━━━━━━━━━━━━━━━━━━━ 188s 285ms/step - accuracy: 0.8709 - loss: 0.0369 - val_accuracy: 0.8842 - val_loss: 0.0437 - learning_rate: 1.8750e-04
Epoch 13/15
661/661 ━━━━━━━━━━━━━━━━━━━━ 179s 271ms/step - accuracy: 0.8754 - loss: 0.0360 - val_accuracy: 0.8931 - val_loss: 0.0428 - learning_rate: 1.0000e-04
```

Figure 61: Bi-LSTM Fine-Tuning Model Training Epochs

Layer (type)	Output Shape	Param #
embedding (Embedding)	(None, 200, 300)	6,000,000
bidirectional (Bidirectional)	(None, 200, 128)	186,880
dropout (Dropout)	(None, 200, 128)	0
bidirectional_1 (Bidirectional)	(None, 200, 128)	98,816
dropout_1 (Dropout)	(None, 200, 128)	0
bidirectional_2 (Bidirectional)	(None, 64)	41,216
dropout_2 (Dropout)	(None, 64)	0
dense (Dense)	(None, 48)	3,120
dense_1 (Dense)	(None, 1)	49

Total params: 6,990,245 (26.67 MB)

Trainable params: 330,081 (1.26 MB)

Non-trainable params: 6,000,000 (22.89 MB)

Optimizer params: 660,164 (2.52 MB)

Figure 62: Bi-LSTM Fine-Tuning Model Summary

The model uses pre-trained word vectors from FastText, which are set as non-trainable in the embedding layer (trainable=False). This means the vectors retain their original semantic knowledge and are not updated during training. As a result, the 6,000,000 non-trainable parameters refer to these fixed FastText word embeddings.

Table 9: Compare Model Architecture Between Hyperparameter Tuning and Fine-Tuning

	Hyperparameter Tuning (Best Hyperparameter)	**Fine Tuning**
Word Embedding Layer	Randomly Initialized Trainable Embedding	**FastText**
Bi-LSTM Layer 1 Units	256	**64**
Dropout Layer 1	0.2	0.2
Bi-LSTM Layer 2 Units	64	64
Dropout Layer 2	0.3	0.3
Bi-LSTM Layer 3 Units	16	**32**
Dropout Layer 3	0.3	**0.4**
Dense Units	32	**48**
Activation	ReLu	ReLu
L2 regularization	0.031	**0.001**
Loss Function	Binary Cross Entropy	**Focal Loss (gamma=2.0, alpha=0.45)**
Optimizer	Adam	Adam

Learning Rate	0.0001	**0.003**
Epochs	15	15
Batch Size	32	32
Callback	Early Stopping (patience=3)	Early Stopping (patience=3), **ReduceLROnPlateau (factor=0.5, patience=1)**

The Bi-LSTM model exhibited signs of overfitting during the hyperparameter tuning the purpose of fine-tuning is to mitigate this issue. During hyperparameter tuning, Bi-LSTM model using FastText word embedding method that proposed by Facebook. Different with traditional embeddings, FastText takes advantage of subword information, which helps the model handle out-of-vocabulary (OOV) words more effectively (Pritom Mojumder et al., 2020). This is especially useful for toxic comments, which often contain slang, misspellings, or informal language. It able let the model have a better learn toxic comments that often include slang, misspellings or informal language. Loads the crawl-300d-2M-subword.bin file that download from FastText official website, which provides 300-dimensional pre-trained embeddings that enrich each token's representation.

To address the overfitting issue observed during hyperparameter tuning, several architectural adjustments were made to reduce model complexity and enhance generalization. Specifically, the number of units in the first Bi-LSTM hidden layer was reduced from 256 to 64. Third Bi-LSTM layer's units were increased from 16 to 32. The dense layer size was adjusted from 32 to 48 and combined with an L2 regularization factor of 0.001 to penalize over-complex weight magnitudes. 15 epochs and a batch size of 32 were kept unchanged. During model training, in addition to using EarlyStopping, the ReduceLROnPlateau callback was applied to monitor validation loss. Factor setting to 0.5, the learning rate was reduced to 50% of the current value if the validation loss did not improve for 1 epoch.

In addition, to improve the performance of the model on the minority class (class 1), the loss function uses Focal Loss rather than using traditional binary cross-entropy. This loss function refers to the formula proposed by (aldi-dimara, 2018). By adjusting gamma and alpha, the weight of simple-to-categorize samples is dynamically reduced and hard-to-categorize samples are emphasized. In this experiment, gamma is set to 0.2, and alpha is adjusted from the original recommended value of 0.25 to 0.45. According to the comparison (Focal Loss Function

Calculation with Different Alpha Values (0.25 and 0.75)), alpha > 0.5 helps to improve the model's attention to minority classes and alpha < 0.5 more focus on majority classes. Although values above 0.5 give more focus to the minority class, experiments showed that Alpha was set to 0.45 (slightly below 0.5) to have a better balance between precision and recall.

```python
@register_keras_serializable(package="CustomLoss")
# Function focal_loss code et from GitHub (aldi-dimara, 2018)
# Gamma, Alpha default values recommended by the paper (Lin et al., 2018) # https://arxiv.org/pdf/1708.02002v2
def focal_loss(y_true, y_pred, gamma=2.0, alpha=0.55):
    epsilon = K.epsilon()
    y_pred = K.clip(y_pred, epsilon, 1.0-epsilon)
    # Predicted probability of the correct class
    pt = tf.where(K.equal(y_true, 1), y_pred, 1-y_pred)
    # Alpha factor for balancing class imbalance.
    alpha_factor = K.ones_like(y_true)*alpha
    # Dynamically assigns different alpha values to different classes
    alpha_t = tf.where(K.equal(y_true, 1), alpha_factor, 1-alpha_factor)
    # Cross-entropy loss
    cross_entropy = -K.log(pt)
    # Alpha > 0.5, then class 1 weigth > class 0 weigth. Alpha < 0.5, then class 0 weigth > class 1 weigth
    weight = alpha_t * K.pow((1-pt), gamma) #alpha_t*(1 - pt) ^ gamma
    loss = weight * cross_entropy
    loss = K.mean(loss, axis=1)
    return loss
```

Figure 63: Focal Loss Function

Table 10: Focal Loss Function Calculation with Different Alpha Values (0.25 and 0.75)

Category 1: Toxic 0: Non-toxic	**Y Pred**	**PT Value**	**"alpha_t" Value (Alpha=0.25)**	**"alpha_t" Value (Alpha=0.75)**	**Class Weight (Alpha=0.25)**	**Class Weight (Alpha=0.75)**
True Predict Class 1	0.7	0.7	0.25	0.75	0.0225	0.0675
True Predict Class 0	0.2	0.8	0.75	0.25	0.03	0.01
False Predict Class 1	0.4	0.4	0.25	0.75	0.09	**0.27**
False Predict Class 0	0.7	0.3	0.75	0.25	**0.3675**	0.1225

Category	**Cross Entropy**	**Loss (Alpha=0.25)**	**Loss (Alpha=0.75)**
True Predict Class 1	0.3567	0.0080	0.0241
True Predict Class 0	0.2231	0.0067	0.0022
False Predict Class 1	0.9163	0.0825	**0.2474**
False Predict Class 0	1.2040	**0.4425**	0.1475

Higher alpha values (above 0.5) make the model focus more on class 1 (the minority class) by increasing its loss weight, while values below 0.5 model more attention to class 0 (the majority class).

5.4.1 Bi-LSTM Fine-Tuning Model Evaluation

```
📌 Train Accuracy: 0.8535
📌 Validation Accuracy: 0.9286

📌 Weighted F1 Score: 0.9316
📌 Precision: 0.7874
📌 Recall: 0.8405

📌 ROC AUC: 0.9294

📌 Classification Report:
              precision    recall  f1-score   support

           0       0.97      0.95      0.96     38631
           1       0.60      0.73      0.66      4074

    accuracy                           0.93     42705
   macro avg       0.79      0.84      0.81     42705
weighted avg       0.94      0.93      0.93     42705
```

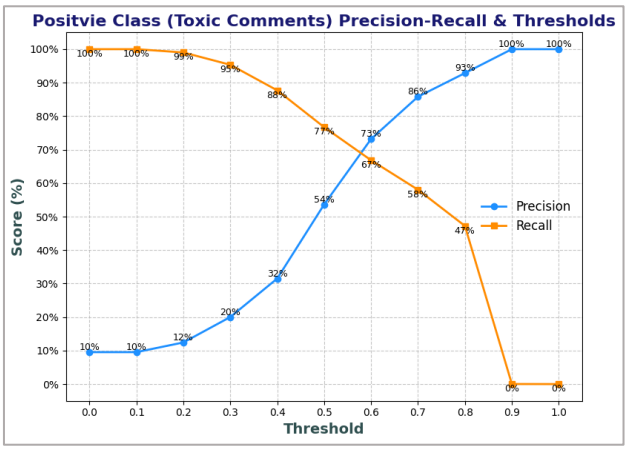


Figure 64: Bi-LSTM Fine Tuning Evaluation

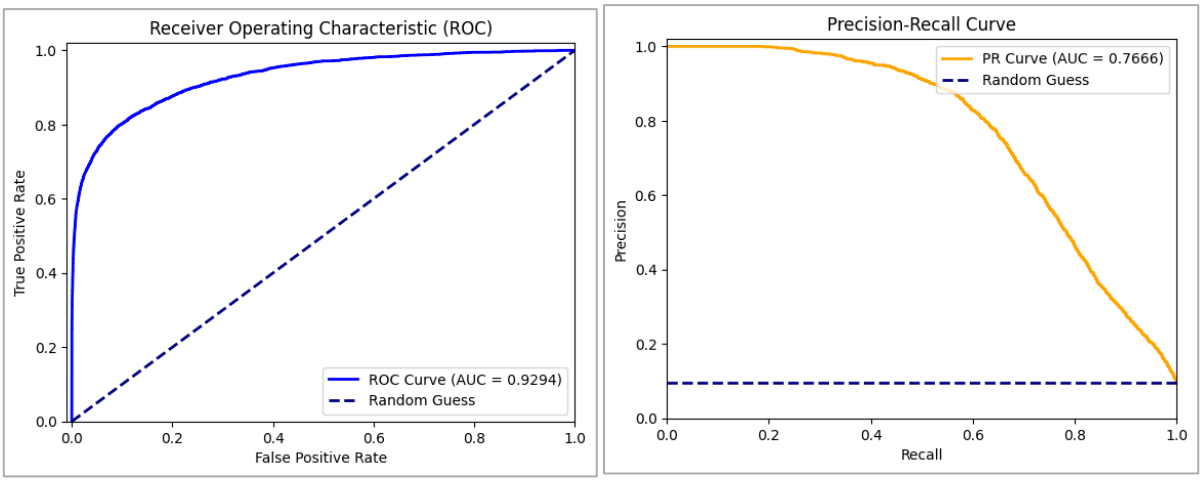
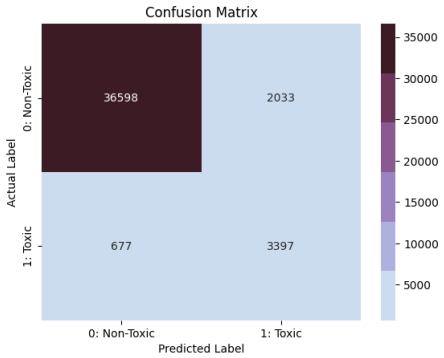
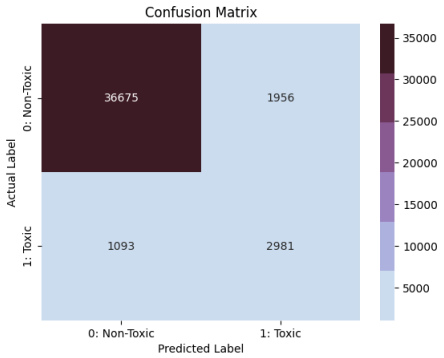
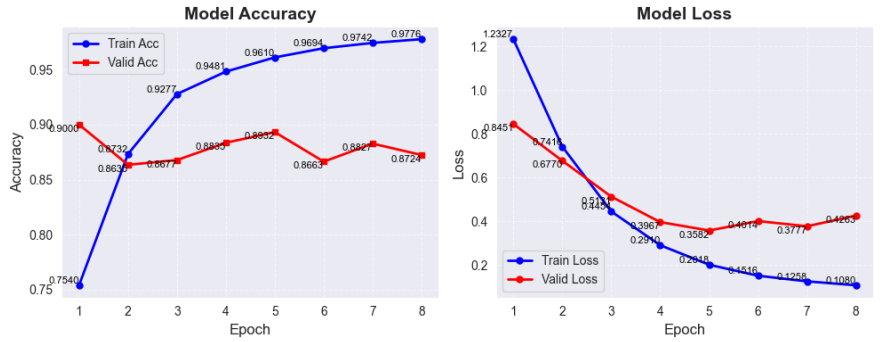
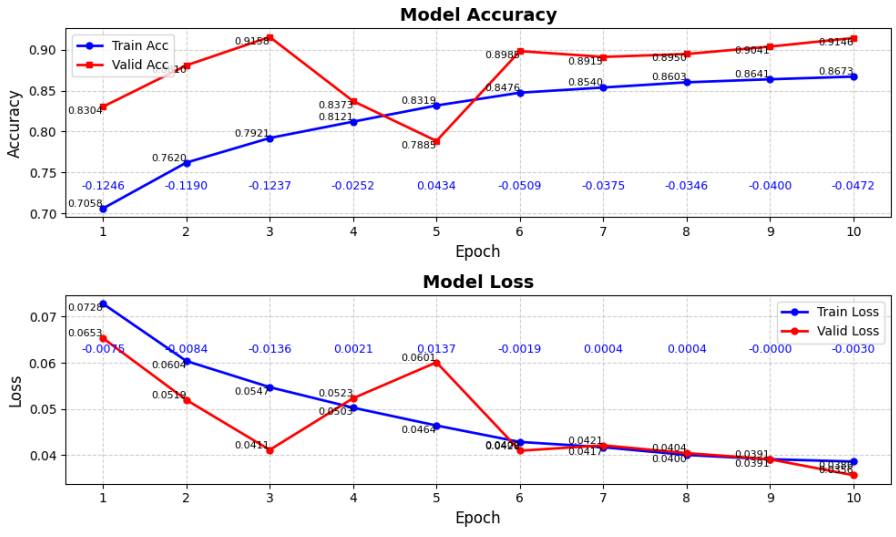


Figure 65: Bi-LSTM Fine Tuning ROC Curve and PR Curve

Table 11: Comparison Between Bi-LSTM Hyperparameter Tuning and Fine Tuning

	Hyperparameter Tuning (Threshold 0.9)	Fine Tuning (Threshold 0.53)
Train Accuracy	0.9580	0.8535
Validation Accuracy	0.9365	0.9286
Weighted F1 Score	0.9405	0.9316
Macro Precision	0.8037	0.7874
Macro Recall	0.8906	0.8405
Binary Precision	0.63	0.60
Binary Recall	0.83	0.73

ROC-AUC Score	0.9554	0.9294
PR-AUC Score	0.8196	0.7666
Confusion Matrix		
False Positive Rate	5.26%	5.1%
False Negative Rate	16.62%	26.83%
	Learning Curve	
Hyperparameter Tuning		
Fine Tuning		

In order to alleviate the overfitting problem, the overall performance of the model in the fine-tuning stage is slightly lower than the result of hyperparameter tuning. In fine-tuning, the

threshold is set to 0.53 to have a balance between precision (60%) and recall (73%) of class 1 (toxic comments). Here choose to keep a higher recall rate because recall rate is more important than precision rate in the toxic comment detection task.

The validation set accuracy of the fine-tuning model is 92.86%, and the training set accuracy is 85.35%, which is 7.51% lower than the validation accuracy. The weighted F1 score is 93.16%, the ROC-AUC score is 92.94%, and the PR-AUC score is 76.66%. In terms of the false positive rate, the false positive rate remains at 5.10%, which is 0.16% lower than hyperparameter tuning but the false negative rate increases to 26.83%. This result reflects that while the model reduces the misjudgement of non-toxic comments, it sacrifices some of its ability to identify toxic comments, which is also a common trade-off problem in class imbalanced data.

Compared with the hyperparameter adjustment stage, fine-tuning effectively solves the overfitting problem. The validation learning and loss curve gradually stabilizes from the 6th epoch. Through the EarlyStopping function, the model automatically stops training at the 10th epoch to prevent the model overfitting problem. Although the validation set accuracy curve has a slight fluctuation in the 7th epoch (decreases by 0.7% but this fluctuation is still within an acceptable range. It is worth noting that the validation set accuracy is higher than the training set accuracy and the training accuracy reaches 86.73%. This shows that the model has neither overfitting nor underfitting and has a good generalization fitting effect on the data. The overall performance is better than hyperparameter tuning, although other evaluation metrics are slightly lower.

6.0 Model 2: CNN-Bi LSTM Hybrid Model

Justification for Selecting CNN-Bi LSTM Model:

The CNN-BiLSTM model was selected due to its ability to capture long-term relationships and local patterns in text data. The CNN (Convolutional Neural Network) component excels at identifying key n-gram features, such as specific word combinations or local patterns. On the other hand, the BiLSTM layer can capture the sequential nature of language in both the forward and reverse directions and can handle complex sentence structures. By combining CNN with BiLSTM, the model can improve text classification performance by leveraging CNN to extract local contextual patterns while leveraging BiLSTM's expertise in capturing long-term dependencies and semantic information.

6.1 CNN Hyperparameter Explanation

Table 12: CNN Hyperparameter Explanation

Hyperparameter	Explanation
Convolutional Filter (Conv1D Layer)	In text processing, the number of convolution filters determines how many different local patterns the model can learn from the input sequence. Each filter is responsible for capturing a specific n-gram feature, such as a phrase or word group. The number of output channels in the generated feature map corresponds to the number of filters used. According to (Azam et al., 2024), the number of convolution filters in CNN models is usually selected as a power of 2, such as 32, 64, 128, 256.
Kernel Size (Conv1D Layer)	Kernel size determines the number of words processed in each convolution operation. For example, kernel size equal to 3 means processing 3 words at a time. Larger kernels capture longer-range dependencies but increase computational cost. Recommend size is 3, 5, or 7 (Azam et al., 2024).
Padding (Conv1D Layer)	The main function of padding is to prevent the convolution operation from shortening the sequence length and reduce the loss of edge information. **Valid** padding means that the input is not padded with zeros, so each convolution will shorten the output sequence length. **Same** pads zeros at both ends of the

	input sequence, so that the length of the output sequence after convolution is consistent with the input, which helps to preserve the structural information at the beginning and end of the sentence and avoid information loss during feature extraction.
Strides (Conv1D Layer)	The stride of the kernel. The stride of the kernel determines the density of feature extraction in the convolution operation. The larger the stride, the smaller the output, but may lose fine-grained information. The default stride is 1 for dense features and 2 for downsampling (Azam et al., 2024).
Activation (Conv1D Layer)	The activation function adds non-linearity to help the model learn complex patterns. ReLU is the most commonly used activation in Conv1D layers because it is fast and helps prevent vanishing gradients (Azam et al., 2024).
Pool Size (MaxPooling1D Layer)	Pooling size is used to reduce the dimension of feature map and reduce the amount of calculation.

6.2 CNN-BiLSTM Basic Model

```
MAX_SEQUENCE_LENGTH = 200
EMBEDDING_DIM = 300
MAX_VOCAB_SIZE = 20000

model = Sequential([
    Embedding(input_dim=MAX_VOCAB_SIZE, output_dim=EMBEDDING_DIM, input_length=MAX_SEQUENCE_LENGTH),
    Conv1D(filters=128, kernel_size=3, activation="relu", padding="same"),
    MaxPooling1D(pool_size=3),
    Bidirectional(LSTM(64, return_sequences=True)),
    Dropout(0.2),
    Bidirectional(LSTM(64, return_sequences=False)),
    Dropout(0.3),
    Dense(48, activation='relu', kernel_regularizer=tf.keras.regularizers.l2(0.001)),
    Dense(1, activation='sigmoid')
])

model.compile(
    loss="binary_crossentropy",
    optimizer=Adam(learning_rate=0.001),
    metrics=["accuracy"]
)

history = model.fit(x_train_resampled, y_train_resampled,
                    epochs=10, batch_size=32,
                    validation_data=(X_valid, Y_valid))
```

Figure 66: CNN-Bi LSTM Basic Model

```
Epoch 1/10
661/661 ━━━━━━━━━━━━━━━━━━━━ 123s 172ms/step - accuracy: 0.7691 - loss: 0.4837 - val_accuracy: 0.8640 - val_loss: 0.3031
Epoch 2/10
661/661 ━━━━━━━━━━━━━━━━━━━━ 131s 198ms/step - accuracy: 0.9216 - loss: 0.2038 - val_accuracy: 0.8423 - val_loss: 0.3463
Epoch 3/10
661/661 ━━━━━━━━━━━━━━━━━━━━ 141s 214ms/step - accuracy: 0.9512 - loss: 0.1281 - val_accuracy: 0.8788 - val_loss: 0.3204
Epoch 4/10
661/661 ━━━━━━━━━━━━━━━━━━━━ 148s 224ms/step - accuracy: 0.9764 - loss: 0.0647 - val_accuracy: 0.8512 - val_loss: 0.5347
Epoch 5/10
661/661 ━━━━━━━━━━━━━━━━━━━━ 162s 245ms/step - accuracy: 0.9830 - loss: 0.0457 - val_accuracy: 0.8065 - val_loss: 0.7085
Epoch 6/10
661/661 ━━━━━━━━━━━━━━━━━━━━ 155s 235ms/step - accuracy: 0.9854 - loss: 0.0406 - val_accuracy: 0.8427 - val_loss: 0.5803
Epoch 7/10
661/661 ━━━━━━━━━━━━━━━━━━━━ 161s 244ms/step - accuracy: 0.9889 - loss: 0.0316 - val_accuracy: 0.8704 - val_loss: 0.5837
Epoch 8/10
661/661 ━━━━━━━━━━━━━━━━━━━━ 127s 192ms/step - accuracy: 0.9894 - loss: 0.0272 - val_accuracy: 0.7562 - val_loss: 0.8608
Epoch 9/10
661/661 ━━━━━━━━━━━━━━━━━━━━ 111s 168ms/step - accuracy: 0.9888 - loss: 0.0292 - val_accuracy: 0.8535 - val_loss: 0.6683
Epoch 10/10
661/661 ━━━━━━━━━━━━━━━━━━━━ 112s 169ms/step - accuracy: 0.9923 - loss: 0.0202 - val_accuracy: 0.8517 - val_loss: 0.7591
```

Figure 67: CNN-Bi LSTM Basic Model Training Epochs

Layer (type)	Output Shape	Param #
embedding (Embedding)	(None, 200, 300)	6,000,000
conv1d (Conv1D)	(None, 200, 128)	115,328
max_pooling1d (MaxPooling1D)	(None, 66, 128)	0
bidirectional (Bidirectional)	(None, 66, 128)	98,816
dropout (Dropout)	(None, 66, 128)	0
bidirectional_1 (Bidirectional)	(None, 128)	98,816
dropout_1 (Dropout)	(None, 128)	0
dense (Dense)	(None, 48)	6,192
dense_1 (Dense)	(None, 1)	49

Total params: 6,319,203 (24.11 MB)

Trainable params: 6,319,201 (24.11 MB)

Non-trainable params: 0 (0.00 B)

Optimizer params: 2 (12.00 B)

Figure 68: CNN_Bi-lSTM Basic Model Summary

The CNN-BiLSTM basic model is built using a randomly initialized, trainable embedding layer, followed by a 1D convolutional (Conv1D) layer and a MaxPooling layer with pool size of 3. The convolutional layer extracts local n-gram features, which are then down sampled by max pooling. This is followed by two Bi-LSTM layers, each with 64 units. To prevent overfitting, dropout layers with rates of 0.2 and 0.3 are applied after the Bi-LSTM layers. A fully connected Dense layer with 48 units and ReLU activation is used before the output layer, incorporating L2 regularization of 0.001 to reduce overfitting by penalizing large weights. Dense output layer with a sigmoid activation function to produce the final binary classification result. The Adam optimizer is used to train the model across 10 epochs with a batch size of 32 and a learning rate of 0.001.

6.2.1 CNN-BiLSTM Basic Model Evaluation

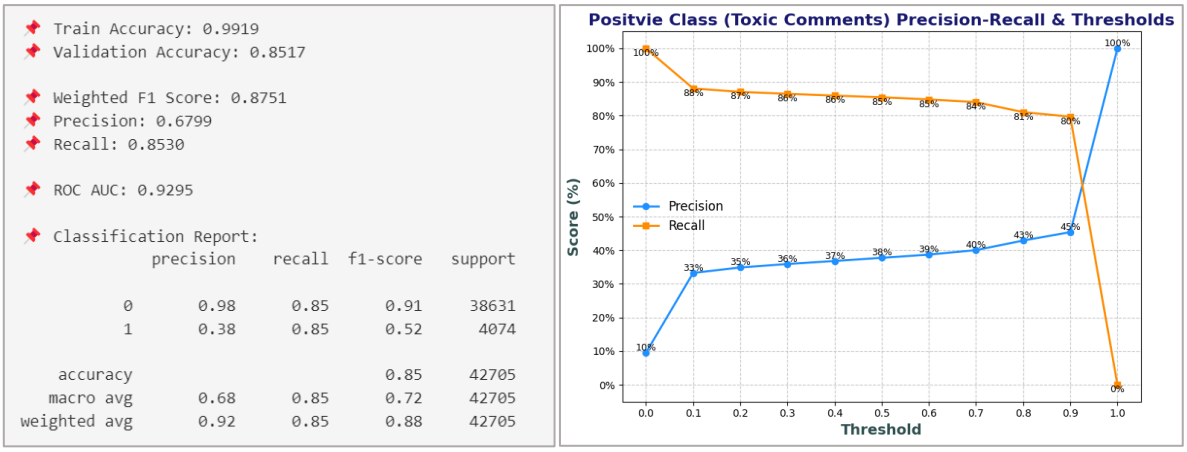


Figure 69: CNN-BiLSTM Evaluation Metrics

In threshold 0.5, model achieves train accuracy 99.19% but validation accuracy 85.17%, weighted F1 score of 87.51%, precision of 67.99% and recall of 85.30%. The ROC-AUC score of 92.95%.

Although the model performed well overall, its accuracy on category 1 (toxic comments) was still low at only 38%. Even when the decision threshold was increase to 0.9, the class 1 precision only increased to 45%, indicating that the model still had significant deficiencies in reducing false positives and was prone to misclassifying non-toxic content as toxic when identifying toxic comments.

Confusion Matrix

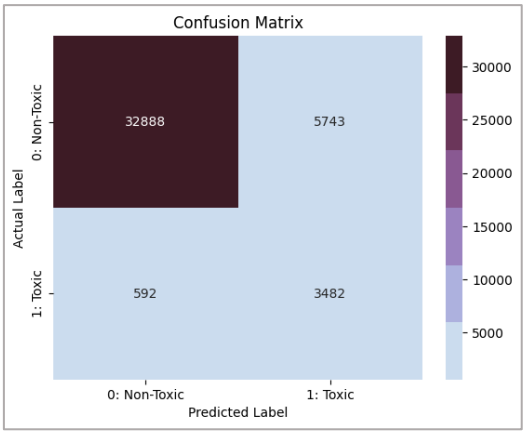


Figure 70: CNN-BiLSTM Basic Model Confusion Matrix

CNN Bi-LSTM hybrid model correctly classified 32888 rows of data are non-toxic comments and 3482 rows of toxic comments. The false positive rate is 14.87% and the false negative rate is 14.53%.

ROC-AUC Curve and Precision-Recall Curve

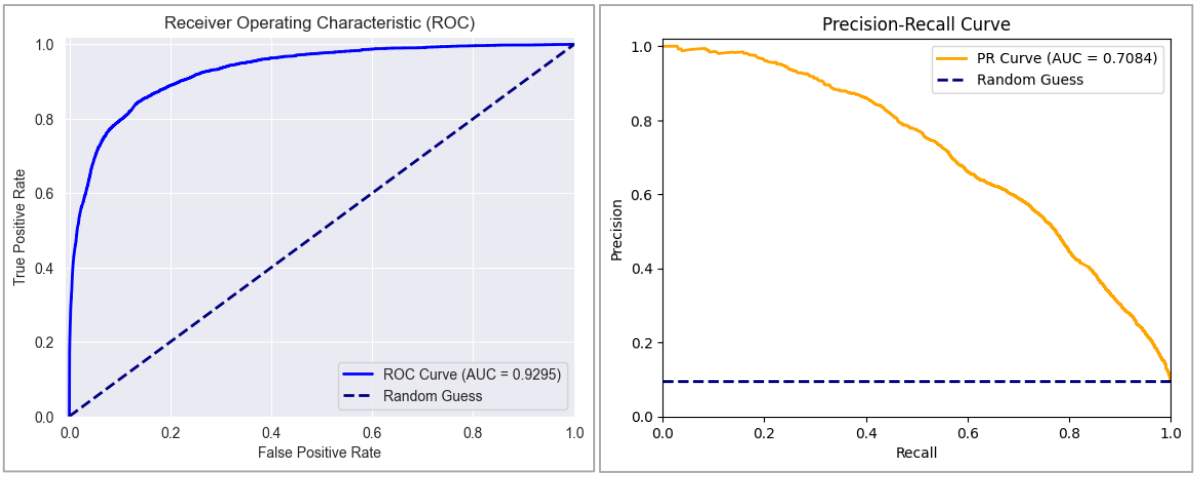


Figure 71: CNN-BiLSTM Basic Model ROC Curve (left) and PR Curve (right)

Although the model performs well overall, with a ROC-AUC score of 92.95%, this result is affected by the class imbalance of the dataset. The model tends to predict the majority class (no toxic comments). However, PR-AUC score only 70.84% for class 1, show that the model's performance in identifying toxic comments is still limited.

Learning Curve

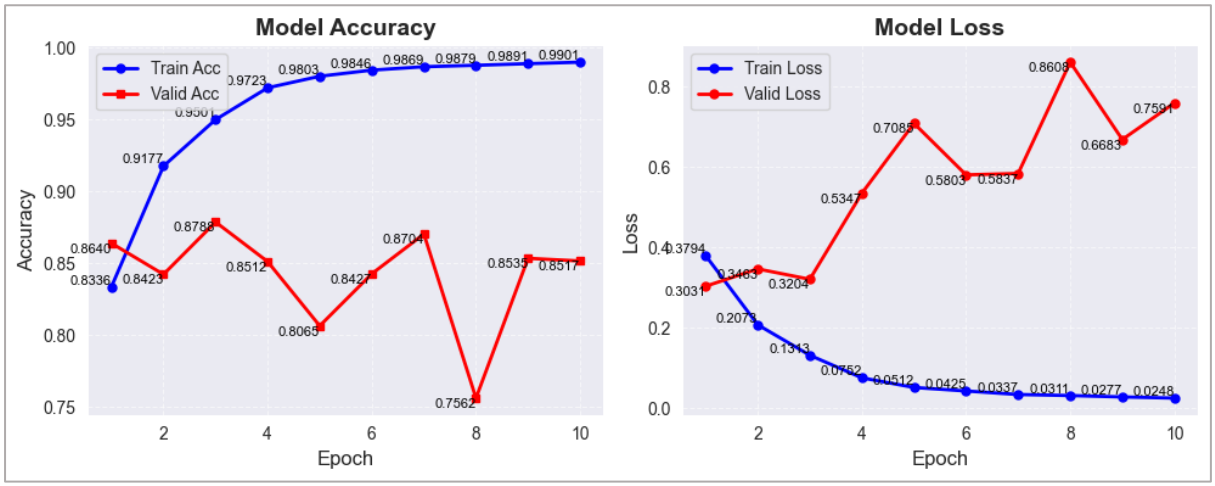


Figure 72: CNN-BiLSTM Basic Model Learning Curve

The CNN-BiLSTM model's performance fluctuations and overfitting starting from epoch 4. This issue may be attributed to several factors, most notably the class imbalance in the validation set, where non-toxic comments have 38631 samples, while toxic comments account for only 4074 samples. Class imbalance can lead the model to overly favor the non-toxic class, reducing its generalization ability for the minority class (toxic comments).

6.3 CNN-BiLSTM Hyperparameter Tuning Model

```
MAX_SEQUENCE_LENGTH = 200
EMBEDDING_DIM = 300
MAX_VOCAB_SIZE = 20000

# fastText dowload from: https://fasttext.cc/docs/en/english-vectors.html
fasttext_path = "crawl-300d-2M-subword.bin"
word_vectors = FastText.load_fasttext_format(fasttext_path)

# declare matrix
embedding_matrix = np.zeros((MAX_VOCAB_SIZE, EMBEDDING_DIM))

for word, i in tokenizer.word_index.items():
    if i < MAX_VOCAB_SIZE:
        if word in word_vectors.wv: # If the word exists in the pretrained word vectors
            embedding_matrix[i] = word_vectors.wv[word] # Assign the pretrained embedding to the corresponding index
        else:
            embedding_matrix[i] = np.random.normal(size=(EMBEDDING_DIM,)) # For out-of-vocab,initialize randomly

# embedding_matrix = np.load("embedding_matrix_20k_200.npy")

embedding_layer = Embedding(
    input_dim=MAX_VOCAB_SIZE,
    output_dim=EMBEDDING_DIM,
    weights=[embedding_matrix],
    input_length=MAX_SEQUENCE_LENGTH,
    trainable=False
)
# np.save("embedding_matrix_20k.npy", embedding_matrix)
```

Figure 73: FastText Word Embedding

```
def build_model(hp):
    model = Sequential()
    model.add(embedding_layer)
    # The kernel size determines the number of words processed in each convolution operation,  for example, kernel size=3 means processing 3 words at a time
    model.add(Conv1D(filters=hp.Int("conv_filters", min_value=32, max_value=256, step=32),
                     kernel_size=hp.Int("kernel_size", min_value=3, max_value=5, step=1), activation='relu',padding='same'))
    model.add(MaxPooling1D(pool_size=hp.Int("pool_size", min_value=2, max_value=5, step=1))) # Apply 1D max pooling to downsample the feature maps

    model.add(Bidirectional(LSTM(units=hp.Int("LSTM_layer_1", min_value=64, max_value=256, step=64),return_sequences=True)))
    model.add(Dropout(hp.Float("dropout_1", min_value=0.2, max_value=0.4, step=0.1)))

    model.add(Bidirectional(LSTM(units=hp.Int("LSTM_layer_2", min_value=32, max_value=128, step=32),return_sequences=True)))
    model.add(Dropout(hp.Float("dropout_2", min_value=0.2, max_value=0.4, step=0.1)))

    model.add(Bidirectional(LSTM(units=hp.Int("LSTM_layer_3", min_value=16, max_value=64, step=16),return_sequences=False)))
    model.add(Dropout(hp.Float("dropout_3", min_value=0.2, max_value=0.4, step=0.1)))

    model.add(Dense(units=hp.Int("dense_units", min_value=16, max_value=64, step=16),
                    activation=hp.Choice("activation", ["relu", "tanh","sigmoid"]),
                    kernel_regularizer=tf.keras.regularizers.l2(hp.Float("l2_reg", min_value=0.001, max_value=0.07, step=0.005))))
    model.add(Dense(1, activation="sigmoid"))

    optimizer_choice = hp.Choice('optimizer', ['adam', 'sgd', 'adagrad'])
    learning_rate = hp.Choice("learning_rate", [0.0001, 0.003, 0.01, 0.03])
    if optimizer_choice == 'sgd':
        optimizer = SGD(learning_rate=learning_rate, momentum=0.9)
    elif optimizer_choice == 'adam':
        optimizer = Adam(learning_rate=learning_rate)
    elif optimizer_choice == 'adagrad':
        optimizer = Adagrad(learning_rate=learning_rate)

    model.compile(
        optimizer = optimizer,loss="binary_crossentropy",metrics=["accuracy"])
    return model

tuner = BayesianOptimization(
    build_model,
    objective='val_accuracy',
    max_trials=10,
    executions_per_trial=1,
    directory='my_dir_CNN_BiLSTM',
    project_name='1_HP_CNN_BiLSTM'
)

tuner.search(x_train_resampled,y_train_resampled,
             epochs=10,
             batch_size=32,
             validation_data=(X_valid,Y_valid))
```

Figure 74: CNN-BiLSTM Model Hyperparameter Tuning

```
Trial 10 Complete [00h 13m 54s]
val_accuracy: 0.9046013355255127

Best val_accuracy So Far: 0.916239321231842
Total elapsed time: 02h 20m 03s
```

Figure 75: Output of Hyperparameter Tuning

During hyperparameter tuning, CNN-BiLSTM hybrid model using FastText word embedding to let the model have a better handle out-of-vocabulary (OOV) word. The hybrid model architecture consists of one 1D convolutional layer followed by a max pooling layer to capture local n-gram features and reduce dimensionality. It then stacks three Bidirectional LSTM layers to learn contextual and sequential dependencies in the text data. To prevent overfitting, each LSTM layer is followed by a dropout layer. Model includes one dense (fully connected) layer with an activation function (ReLU, tanh, or sigmoid) and applies L2 regularization to control complexity. Finally, a single-node output layer with sigmoid activation is used to perform binary classification.

The model architecture consists of a total of five hidden layers, including one convolutional layer, three bidirectional LSTM layers and one dense layer. Hyperparameter tuning was performed using the Bayesian Optimization technique. Justification can refer to Hyperparameter Tuning Technique Selected.

Tables below show each layer CNN and Bi-LSTM hyperparameter tuning setting.

Table 13: CNN Hyperparameter Settings

CNN Hyperparameters	Range/Value
Convolutional Filter (Conv1D Layer)	32-256
Kernel Size (Conv1D Layer)	3-5
Padding (Conv1D Layer)	same
Activation (Conv1D Layer)	ReLu
Pool Size (MaxPooling1D Layer)	2-5

Those CNN hyperparameters setting are references to research paper (Azam et al., 2024) more detail list in part CNN Hyperparameter Explanation.

Table 14: Bi-LSTM Hyperparameters Setting

Bi-LSTM Hyperparameters	Range/Value
Bi-LSTM Units	16-256
Dropout	0.2-0.4
Dense Units	16-64
Activation	ReLU, tanh, sigmoid
l2_reg (L2 regularization)	0.001-0.07

The Bi-LSTM layer hyperparameter settings are almost the same with single Bi-LSTM model (first model: Bi-LSTM Hyperparameter Setting). However, in the hybrid architecture, the Bi-LSTM layer incorporates both tanh and sigmoid activation functions for tuning. Additionally, the L2 regularization range was adjusted from 0.001–0.1 to 0.001–0.07, as higher regularization values were found to cause underfitting, restricting the model's capacity to identify significant patterns in the data.

Table 15: Model Building Hyperparameter Settings

Model Building Hyperparameters	Range/Value
Optimizer	Adam, SGD, Adagrad
Learning Rate (LR)	0.0001, 0.003, 0.01, 0.03
Epochs	10
Max Trials	10
Batch Size	32

The model tuning process involved experimenting with different optimizers, including Adam, SGD and Adagrad, across a range of learning rates from 0.0001 to 0.03. A total of 10 trials were conducted, each using 10 epochs and a batch size of 32, to evaluate which optimizer and learning rate combination have a best performance.

Best Hyperparameter

```
best_model = tuner.get_best_hyperparameters(num_trials=1)[0]
print(f"""Best Hyperparameter:
        CNN  conv_filters: {best_model.get('conv_filters')}
        CNN  kernel_size: {best_model.get('kernel_size')}
        CNN  pool_size: {best_model.get('pool_size')}
        Bi-LSTM  Layer 1: {best_model.get('LSTM_layer_1')}
        Bi-LSTM  Layer 2: {best_model.get('LSTM_layer_2')}
        Bi-LSTM  Layer 3: {best_model.get('LSTM_layer_3')}
        Dropout 1: {best_model.get('dropout_1')}
        Dropout 2: {best_model.get('dropout_2')}
        Dropout 3: {best_model.get('dropout_3')}
        Dense : {best_model.get('dense_units')}
        Activation : {best_model.get('activation')}
        Learning Rate : {best_model.get('learning_rate')}
        Optimizer Choice: {best_model.get('optimizer')}
        L2: {best_model.get('l2_reg')}
        """)
```

Figure 76: Code of Get Best Hyperparameter

```
Best Hyperparameter:
        CNN  conv_filters: 160
        CNN  kernel_size: 4
        CNN  pool_size: 4
        Bi-LSTM  Layer 1: 128
        Bi-LSTM  Layer 2: 32
        Bi-LSTM  Layer 3: 32
        Dropout 1: 0.30000000000000004
        Dropout 2: 0.4
        Dropout 3: 0.2
        Dense : 48
        Activation : tanh
        Learning Rate : 0.003
        Optimizer Choice: sgd
        L2: 0.016
```

Figure 77: Output of CNN-BiLSTM Best Hyperparameter

Following hyperparameter adjustment, 160 convolution filters with a kernel size of 4 are the optimal hyperparameter for the CNN-BiLSTM model to capture local n-gram information. The pooling layer with a size of 4 reduces dimensionality and helps prevent overfitting by minimizing the number of trainable parameters. The Bi-LSTM component is composed of three stacked layers with 128, 32, and 32 units, each layer follow by a dropout layer 0.3, 0.4, 0.2 to mitigate overfitting. A fully connected dense layer with 48 units and tanh activation, L2 regularization 0.016 was used before the output layer. The model was trained using the SGD optimizer with a learning rate of 0.003.

Model Building with the Best Hyperparameter

```
model = build_model(best_model)
history = model.fit(x_train_resampled, y_train_resampled,
                    epochs=10,
                    batch_size=32,
                    validation_data=(X_valid, Y_valid))

model.summary()

joblib.dump(tuner, "1_HP_CNN_BiLSTM_Tuner.pkl")
model.save("1_HP_CNN_BiLSTM_Model.h5")
joblib.dump(history.history, "1_HP_CNN_BiLSTM_History.pkl")
print("Saved successfully!")
```

Figure 78: Build Model Using Best Hyperparameter

```
Epoch 1/10
661/661 ━━━━━━━━━━━━━━━━━━━━ 79s 110ms/step - accuracy: 0.5288 - loss: 1.3785 - val_accuracy: 0.4810 - val_loss: 0.9314
Epoch 2/10
661/661 ━━━━━━━━━━━━━━━━━━━━ 63s 95ms/step - accuracy: 0.5852 - loss: 0.8702 - val_accuracy: 0.4378 - val_loss: 0.7751
Epoch 3/10
661/661 ━━━━━━━━━━━━━━━━━━━━ 64s 96ms/step - accuracy: 0.5925 - loss: 0.7254 - val_accuracy: 0.5594 - val_loss: 0.6666
Epoch 4/10
661/661 ━━━━━━━━━━━━━━━━━━━━ 64s 97ms/step - accuracy: 0.5965 - loss: 0.6869 - val_accuracy: 0.6090 - val_loss: 0.6333
Epoch 5/10
661/661 ━━━━━━━━━━━━━━━━━━━━ 64s 96ms/step - accuracy: 0.6084 - loss: 0.6668 - val_accuracy: 0.8690 - val_loss: 0.5240
Epoch 6/10
661/661 ━━━━━━━━━━━━━━━━━━━━ 66s 100ms/step - accuracy: 0.6796 - loss: 0.5978 - val_accuracy: 0.8286 - val_loss: 0.4628
Epoch 7/10
661/661 ━━━━━━━━━━━━━━━━━━━━ 71s 107ms/step - accuracy: 0.7201 - loss: 0.5443 - val_accuracy: 0.8310 - val_loss: 0.4561
Epoch 8/10
661/661 ━━━━━━━━━━━━━━━━━━━━ 70s 106ms/step - accuracy: 0.7469 - loss: 0.5131 - val_accuracy: 0.8231 - val_loss: 0.4627
Epoch 9/10
661/661 ━━━━━━━━━━━━━━━━━━━━ 71s 108ms/step - accuracy: 0.7486 - loss: 0.5139 - val_accuracy: 0.7597 - val_loss: 0.6400
Epoch 10/10
661/661 ━━━━━━━━━━━━━━━━━━━━ 65s 98ms/step - accuracy: 0.7184 - loss: 0.5364 - val_accuracy: 0.8637 - val_loss: 0.3760
```

Figure 79: CNN_Bi-LSTM Best Hyperparameter Model Training Epochs

Layer (type)	Output Shape	Param #
embedding (Embedding)	(None, 200, 300)	6,000,000
conv1d_1 (Conv1D)	(None, 200, 160)	192,160
max_pooling1d_1 (MaxPooling1D)	(None, 50, 160)	0
bidirectional_3 (Bidirectional)	(None, 50, 256)	295,936
dropout_3 (Dropout)	(None, 50, 256)	0
bidirectional_4 (Bidirectional)	(None, 50, 64)	73,984
dropout_4 (Dropout)	(None, 50, 64)	0
bidirectional_5 (Bidirectional)	(None, 64)	24,832
dropout_5 (Dropout)	(None, 64)	0
dense_2 (Dense)	(None, 48)	3,120
dense_3 (Dense)	(None, 1)	49

Total params: 7,180,164 (27.39 MB)

Trainable params: 590,081 (2.25 MB)

Non-trainable params: 6,000,000 (22.89 MB)

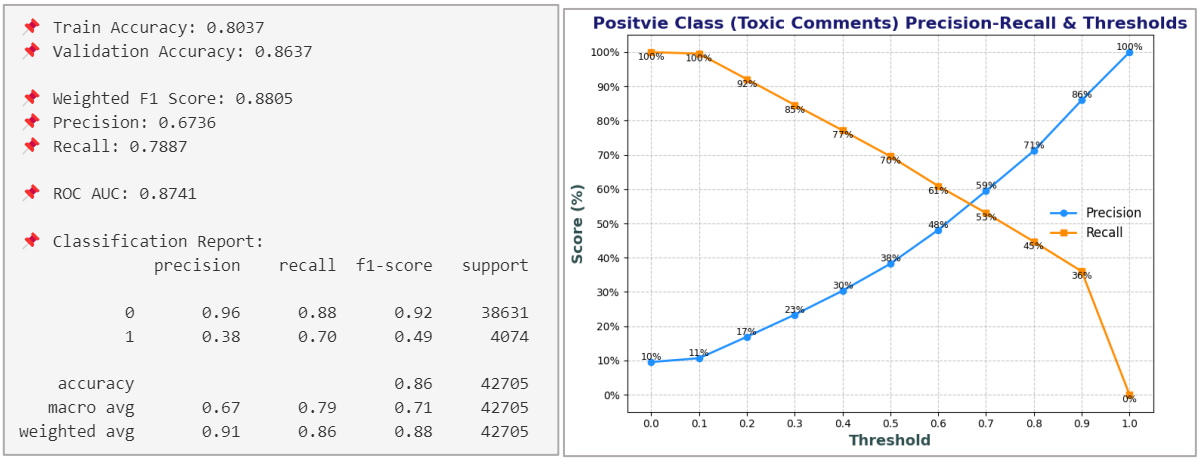
Optimizer params: 590,083 (2.25 MB)

Figure 80: CNN_Bi-LSTM Best Hyperparameter Model Summary

Batch size of 32 was used to train the model across 10 epochs. The model is a hybrid neural network structure based on CNN and multi-layer bidirectional LSTM, with a total parameter

count of approximately 7.18 million, of which 6 million parameters come from pre-trained word embedding layers (FastText), which are set to be non-trainable to retain semantic information.

6.3.1 CNN-BiLSTM Hyperparameter Tuning Model Evaluation



📌 Train Accuracy: 0.8037
📌 Validation Accuracy: 0.8637

📌 Weighted F1 Score: 0.8805
📌 Precision: 0.6736
📌 Recall: 0.7887

📌 ROC AUC: 0.8741

📌 Classification Report:

	precision	recall	f1-score	support
0	0.96	0.88	0.92	38631
1	0.38	0.70	0.49	4074
accuracy			0.86	42705
macro avg	0.67	0.79	0.71	42705
weighted avg	0.91	0.86	0.88	42705

Figure 81: CNN-BiLSTM Evaluation Metrics

Threshold 0.5, hyperparameter tuning model achieve train accuracy 80.37%, validation accuracy 86.37%, weighted F1 score of 88.05%, precision of 67.36% and recall of 78.87%. The ROC-AUC score of 87.41%.

The model's class 1 (toxic comments) precision is 38% and recall is 70%. The precision-recall-threshold curve further illustrates that the model has difficult to accurately identifying toxic comments. As precision increases, recall drops sharply, forming an X-shaped trade-off pattern in the graph. This shows that the model cannot simultaneously achieve high precision and high recall for class 1, mainly due to the data imbalance issue. Although the false positive rate is lower than the false negative rate, the number of false positives remains high because of the large volume of class 0 samples.

Confusion Matrix

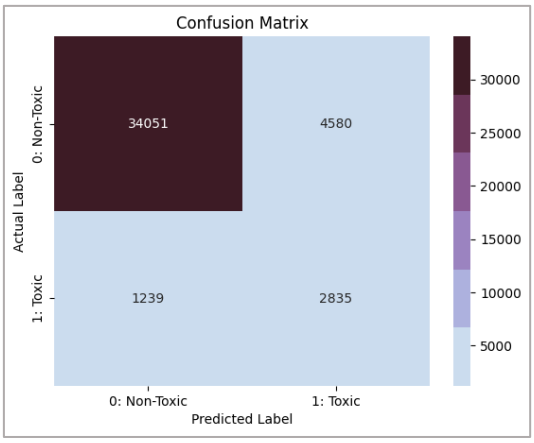


Figure 82: CNN-BiLSTM Hyperparameter Tuning Confusion Matrix

CNN Bi-LSTM hyperparameter tuning model correctly classified 34051 rows of data are non-toxic comments and 2835 rows of toxic comments. The false positive rate is 11.86% and the false negative rate is 30.41%. Compared to the basic model, the false negative rate increased by 15.88%, while the false positive rate decreased by 3.01%

ROC-AUC Curve and Precision-Recall Curve

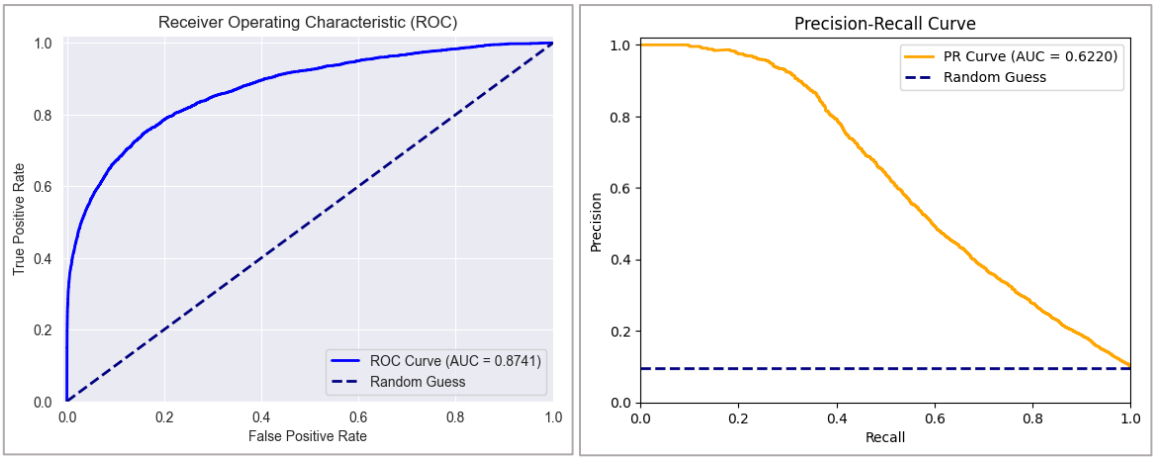


Figure 83: CNN-BiLSTM Hyperparameter Tuning Model ROC Curve (left) and PR Curve (right)

Although the model achieves a relatively high ROC-AUC score of 87.41%, the PR-AUC score is only 62.20%, which clearly indicates that the model performs poorly on class 1 (toxic comments).

Learning Curve

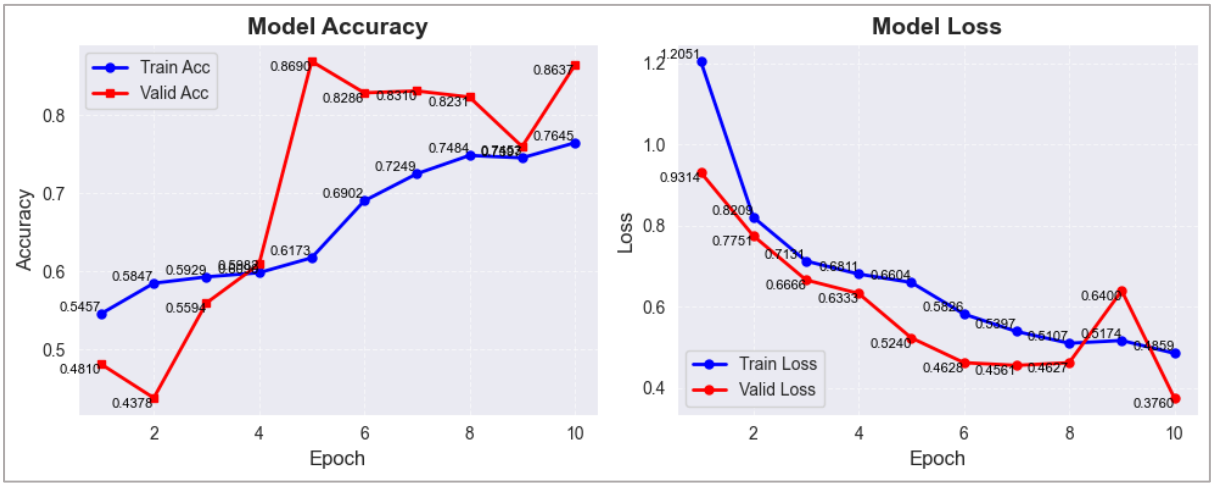


Figure 84: CNN-BiLSTM Hyperparameter Tuning Model Learning Curve

The learning curve becomes unstable starting from epoch 5, where the training accuracy and loss remain consistent, but the validation accuracy and loss fluctuate. This indicates that the model sensitive to noise in the validation set due to class imbalance.

6.4 CNN-BiLSTM Fine Tuning

```
MAX_SEQUENCE_LENGTH = 200
EMBEDDING_DIM = 300
MAX_VOCAB_SIZE = 20000

# https://fasttext.cc/docs/en/english-vectors.html
fasttext_path = "crawl-300d-2M-subword.bin"
word_vectors = FastText.load_fasttext_format(fasttext_path)

# declare matrix
embedding_matrix = np.zeros((MAX_VOCAB_SIZE, EMBEDDING_DIM))

for word, i in tokenizer.word_index.items():
    if i < MAX_VOCAB_SIZE:
        if word in word_vectors.wv:
            embedding_matrix[i] = word_vectors.wv[word]
        else:
            embedding_matrix[i] = np.random.normal(size=(EMBEDDING_DIM,))

# embedding_matrix = np.load("embedding_matrix_20k_200.npy")

embedding_layer = Embedding(
    input_dim=MAX_VOCAB_SIZE,
    output_dim=EMBEDDING_DIM,
    weights=[embedding_matrix],
    input_length=MAX_SEQUENCE_LENGTH,
    trainable=False
)
# np.save("embedding_matrix_20k.npy", embedding_matrix)
```

Figure 85: FastText Embedding Layer

```python
# When saving the model, Keras will automatically recognize my_custom_loss and load it correctly
@register_keras_serializable(package="CustomLoss")
# Function focal_loss code et from GitHub (aldi-dimara, 2018)
# Gamma, Alpha default values recommended by the paper (Lin et al., 2018) # https://arxiv.org/pdf/1708.02002v2
def focal_loss(y_true, y_pred, gamma=2.0, alpha=0.55):
    epsilon = K.epsilon()
    y_pred = K.clip(y_pred, epsilon, 1.0-epsilon)
    # Predicted probability of the correct class
    pt = tf.where(K.equal(y_true, 1), y_pred, 1-y_pred)
    # Alpha factor for balancing class imbalance.
    alpha_factor = K.ones_like(y_true)*alpha
    # Dynamically assigns different alpha values to different classes
    alpha_t = tf.where(K.equal(y_true, 1), alpha_factor, 1-alpha_factor)
    # Cross-entropy loss
    cross_entropy = -K.log(pt)
    # Alpha > 0.5, then class 1 weigth > class 0 weigth. Alpha < 0.5, then class 0 weigth > class 1 weigth
    weight = alpha_t * K.pow((1-pt), gamma) #alpha_t*(1 - pt) ^ gamma
    loss = weight * cross_entropy
    loss = K.mean(loss, axis=1)
    return loss
```

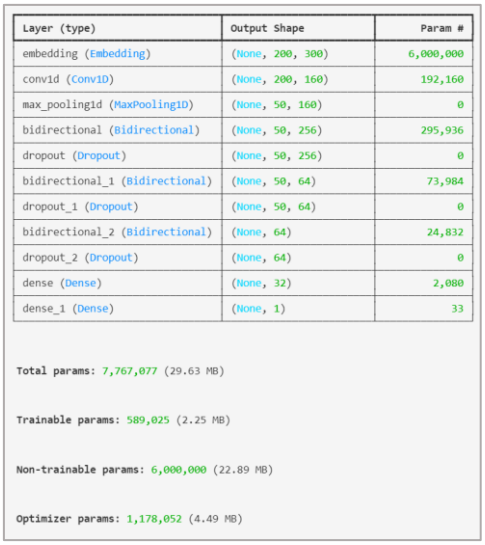
Figure 86: Focal Loss Function

```python
model = Sequential([
    embedding_layer,
    Conv1D(filters=160, kernel_size=4, activation="relu", padding="same"),
    MaxPooling1D(pool_size=4),

    Bidirectional(LSTM(128, return_sequences=True)),
    Dropout(0.3),
    Bidirectional(LSTM(32, return_sequences=True)),
    Dropout(0.4),
    Bidirectional(LSTM(32, return_sequences=False)),
    Dropout(0.3),
    Dense(32, activation='relu', kernel_regularizer=tf.keras.regularizers.l2(0.07)),
    Dense(1, activation='sigmoid')
])

model.compile(
    loss=focal_loss,
    optimizer=Adam(learning_rate=0.0001),
    metrics=["accuracy"]
)

early_stopping = EarlyStopping(monitor='val_loss', patience=3, restore_best_weights=True)

history = model.fit(x_train_resampled, y_train_resampled,
                    epochs=15, batch_size=32,
                    validation_data=(X_valid, Y_valid),
                    callbacks=[early_stopping],
                    )
```

Figure 87: CNN-BiLSTM Fine Tuning Hybrid Model

```
Epoch 1/15
661/661 ━━━━━━━━━━━━━━━━━━━━ 80s 111ms/step - accuracy: 0.6111 - loss: 2.5529 - val_accuracy: 0.7582 - val_loss: 1.3654
Epoch 2/15
661/661 ━━━━━━━━━━━━━━━━━━━━ 72s 109ms/step - accuracy: 0.7977 - loss: 1.1220 - val_accuracy: 0.8373 - val_loss: 0.5671
Epoch 3/15
661/661 ━━━━━━━━━━━━━━━━━━━━ 79s 120ms/step - accuracy: 0.8361 - loss: 0.4605 - val_accuracy: 0.7926 - val_loss: 0.2345
Epoch 4/15
661/661 ━━━━━━━━━━━━━━━━━━━━ 82s 123ms/step - accuracy: 0.8701 - loss: 0.1764 - val_accuracy: 0.8779 - val_loss: 0.0904
Epoch 5/15
661/661 ━━━━━━━━━━━━━━━━━━━━ 82s 124ms/step - accuracy: 0.9083 - loss: 0.0688 - val_accuracy: 0.8486 - val_loss: 0.0590
Epoch 6/15
661/661 ━━━━━━━━━━━━━━━━━━━━ 81s 122ms/step - accuracy: 0.9351 - loss: 0.0348 - val_accuracy: 0.6049 - val_loss: 0.1206
Epoch 7/15
661/661 ━━━━━━━━━━━━━━━━━━━━ 84s 127ms/step - accuracy: 0.9483 - loss: 0.0252 - val_accuracy: 0.8870 - val_loss: 0.0423
Epoch 8/15
661/661 ━━━━━━━━━━━━━━━━━━━━ 87s 132ms/step - accuracy: 0.9627 - loss: 0.0204 - val_accuracy: 0.7917 - val_loss: 0.0881
Epoch 9/15
661/661 ━━━━━━━━━━━━━━━━━━━━ 78s 119ms/step - accuracy: 0.9654 - loss: 0.0188 - val_accuracy: 0.8119 - val_loss: 0.0895
Epoch 10/15
661/661 ━━━━━━━━━━━━━━━━━━━━ 77s 116ms/step - accuracy: 0.9635 - loss: 0.0185 - val_accuracy: 0.9097 - val_loss: 0.0439
```

Figure 88: CNN_Bi-LSTM Fine-Tuning Training Epochs



Layer (type)	Output Shape	Param #
embedding (Embedding)	(None, 200, 300)	6,000,000
conv1d (Conv1D)	(None, 200, 160)	192,160
max_pooling1d (MaxPooling1D)	(None, 50, 160)	0
bidirectional (Bidirectional)	(None, 50, 256)	295,936
dropout (Dropout)	(None, 50, 256)	0
bidirectional_1 (Bidirectional)	(None, 50, 64)	73,984
dropout_1 (Dropout)	(None, 50, 64)	0
bidirectional_2 (Bidirectional)	(None, 64)	24,832
dropout_2 (Dropout)	(None, 64)	0
dense (Dense)	(None, 32)	2,080
dense_1 (Dense)	(None, 1)	33

Total params: 7,767,077 (29.63 MB)

Trainable params: 589,025 (2.25 MB)

Non-trainable params: 6,000,000 (22.89 MB)

Optimizer params: 1,178,052 (4.49 MB)

Figure 89: CNN_Bi-LSTM Fine-Tuning Model Summary

Table 16: Compare Model Architecture Between Hyperparameter Tuning and Fine-Tuning

	Hyperparameter Tuning (Best Hyperparameter)	**Fine Tuning**
Word Embedding Layer	FastText	FastText
Convolutional Filter	160	160
Kernel Size	4	4
Padding	Same	Same
Activation (Conv1D Layer)	ReLu	ReLu
Pool Size	4	4
Bi-LSTM Layer 1 Units	128	128
Dropout Layer 1	0.3	0.3
Bi-LSTM Layer 2 Units	32	32
Dropout Layer 2	0.4	0.4
Bi-LSTM Layer 3 Units	32	32
Dropout Layer 3	0.2	**0.3**
Dense Units	48	**32**
Activation	Tanh	**ReLu**
L2 regularization	0.016	**0.07**
Loss Function	Binary Cross Entropy	**Focal Loss (gamma=2.0, alpha=0.55)**
Optimizer	SGD	**Adam**

Learning Rate	0.003	**0.0001**
Epochs	10	**15**
Batch Size	32	32
Callback	-	**Early Stopping (patience=3)**

The model constructed with the best parameters obtained through hyperparameter tuning has extremely low precision in class 1 (toxic comments) and a false negative rate of up to 30.41%. To solve these problems, the model structure was improved in the fine-tuning stage.

First, dropout layer 3 was increased to 0.3 and the number of units in the dense layer was adjusted from 48 to 32 to reduce the complexity of the model. The activation function uses ReLU because ReLU with Adam optimizer has a better performance in multiple parameter adjustments, but this combination is easy to overfit. Therefore, the L2 regularization term is increased from 0.016 to 0.07 to prevent the model overfitting.

In addition, in order to improve the performance of the model on the minority class (class 1), the loss function is using Focal Loss. By adjusting alpha from 0.25 to 0.55, the weight of easy-to-classify samples is dynamically reduced and difficult-to-classify samples are emphasized. A comparison is provided at Focal Loss Function Calculation with Different Alpha Values (0.25 and 0.75). This comparison shows that alpha > 0.5 helps to improve the model's attention to minority classes and improve the ability to identify toxic comments.

Optimizer is using Adam with the learning rate is 0.0001, which helps to mitigate the fluctuation problem. The batch size 32 not change, and the number of epochs was set to 15 to give the model adequate time to converge and stabilize its performance. Finally, the EarlyStopping function is used to monitor the validation loss and patience 0.3 to avoid over-training of the model.

6.4.1 CNN-BiLSTM Fine Tune Model Evaluation

```
📌 Classification Report:
              precision    recall  f1-score   support

           0       0.97      0.90      0.94     38631
           1       0.45      0.77      0.57      4074

    accuracy                           0.89     42705
   macro avg       0.71      0.84      0.75     42705
weighted avg       0.92      0.89      0.90     42705
```

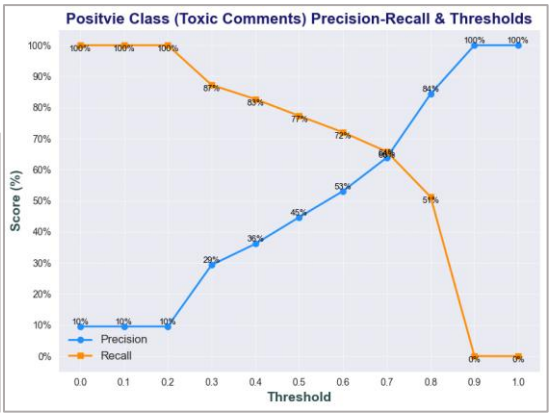


Figure 90: CNN-BiLSTM Fine-Tuning Classification Report and Precision-Recall Threshold Graph

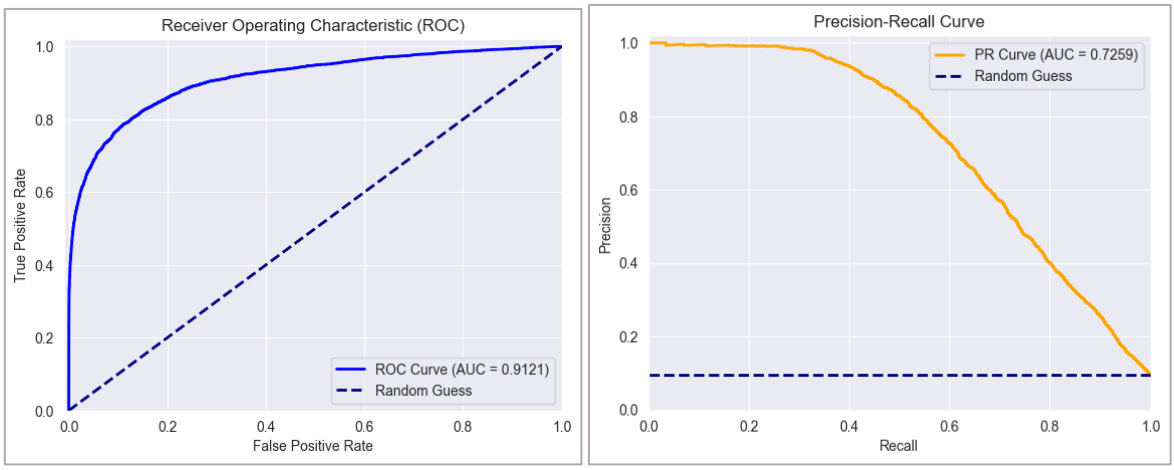


Figure 91: CNN-BiLSTM Fine Tuning ROC Curve (left) and PR Curve (right)

Table 17: Comparison Between Hyperparameter Tuning and Fine-Tuning

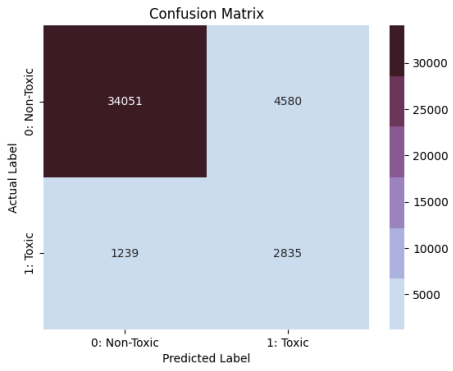
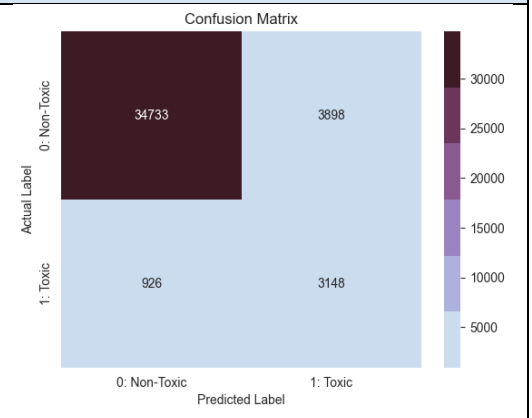
	Hyperparameter Tuning (Threshold 0.5)	Fine Tuning (Threshold 0.5)
Train Accuracy	0.8037	**0.9460**
Validation Accuracy	0.8637	**0.8870**
Weighted F1 Score	0.8805	**0.8999**
Macro Precision	0.6736	**0.7104**
Macro Recall	0.7887	**0.8359**
Binary Precision	0.38	**0.45**
Binary Recall	0.70	**0.77**

ROC-AUC Score	0.8741	**0.9121**
PR-AUC Score	0.6220	**0.7259**

After fine-tuning accuracy of 88.70% and a weighted F1 score of 89.99% on the validation set, the model's overall performance has improved. The macro recall rate has increased to 71.04% and the precision rate has increased to 83.59%. For minority class 1 (toxic comments), the recall rate has increased from 38% to 45% and the precision rate has increased to 77%. In terms of overall evaluation indicators, the ROC-AUC score of 91.21% has increased by 3.8%, and the precision-recall AUC score has also increased from 62% to 72.59%, enhancing the model's ability to detect toxic comments.

Through fine-tuning, it is observed that using the ReLU activation function in combination with the Adam optimizer and focal loss function helps the model converge more efficiently and improves the model's overall performance.

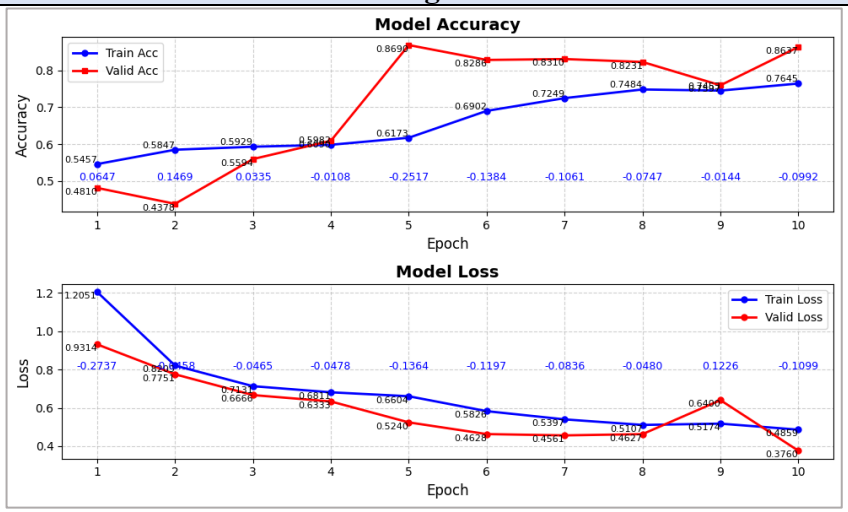
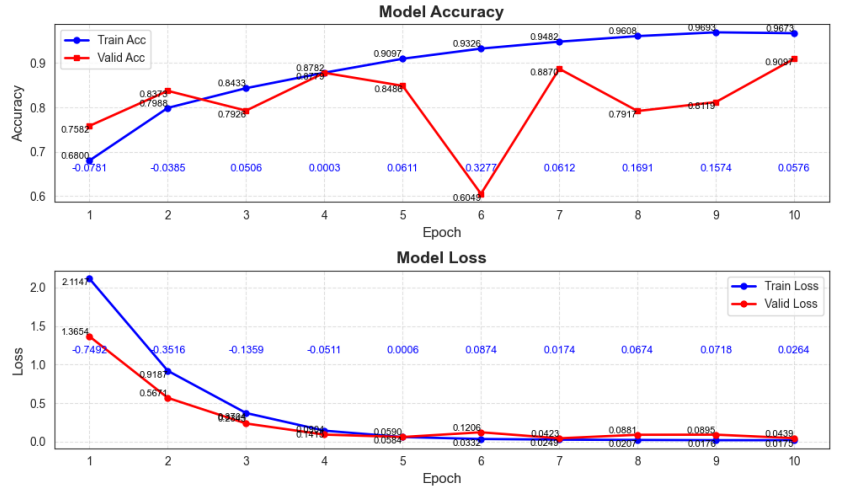
Table 18: Hyperparameter Tuning and Fine-Tuning Confusion Matrices

	Hyperparameter Tuning (Threshold 0.5)	**Fine-Tuning (Threshold 0.5)**
		
False Positive Rate	11.86%	**10.10%**
False Negative Rate	30.41%	**22.73%**

The fine-tuning model's false positive rate (FPR) decreased from 11.86% to 10.10%, with some improvement in reducing misclassified non-hate speech. The false negative rate (FNR) decreased by 7.68% from 30.41% to 22.73%. Since the number of non-toxic comments in the

dataset is far greater than that of toxic comments, this class imbalance means that even with a relatively low false positive rate, the absolute number of misclassified non-toxic comments remains large, leading to low precision for class 1.

Table 19: Hyperparameter Tuning and Fine-Tuning Learning Curve

	Learning Curve
Hyperparameter Tuning	
Fine Tuning	

During the hyperparameter tuning and fine-tuning phase, the training accuracy and loss curves were generally stable, showing that the model did a good job of fitting the training set. However, after multiple rounds of optimization, the validation accuracy still fluctuated significantly. The main reason for this phenomenon is the class imbalance problem, non-toxic comments (class 0) have 38631 rows and toxic comments (class 1) only 4074 rows. This distribution difference makes the model more inclined to learn the features of the majority class, resulting in unstable

performance when identifying the minority class (toxic comments) and it is difficult to maintain good generalization ability.

7.0 Both Models Comparison & Discussion

Table 20: Both Model Comparison

Validation Set Evaluation	Bi-LSTM Fine-Tuning	CNN-BiLSTM Fine-Tuning	Test Set Evaluation	Bi-LSTM	CNN-BiLSTM
Threshold	**0.53**	**0.5**	**Threshold**	**0.5**	**0.5**
Train Accuracy	0.8535	**0.9460**	-		
Validation Accuracy	**0.9286**	0.8870	Test Accuracy	**0.9096**	0.8803
Weighted F1 Score	**0.9316**	0.8999	Weighted F1 Score	**0.9171**	0.8942
Macro Precision	**0.7874**	0.7104	Macro Precision	**0.7459**	0.6993
Macro Recall	**0.8405**	0.8359	Macro Recall	**0.8444**	0.8213
Binary Precision	**0.60**	0.45	Binary Precision	0.52	0.43
Binary Recall	0.73	**0.77**	Binary Recall	**0.76**	0.75
ROC-AUC Score	**0.9294**	0.9121	ROC-AUC Score	**0.9165**	0.9080
PR-AUC Score	**0.7666**	0.7259	PR-AUC Score	**0.7329**	0.6878
False Positive Rate	**5.1%**	10.10%	False Positive Rate	**7.5%**	10.58%
False Negative Rate	26.83%	**22.73%**	False Negative Rate	**23.62%**	25.17%

This table presents the evaluation results of Bi-LSTM and CNN-BiLSTM models on both the validation and test sets. The Bi-LSTM model demonstrates a better balance between binary precision (60%) and recall (73%) on the validation set when using a lower threshold of 0.53, whereas the CNN-BiLSTM model only achieves 45% precision and 77% recall at threshold of

0.5. This difference highlights Bi-LSTM's better ability to distinguish toxic comments under the data imbalance problem.

All assessment measures show that the Bi-LSTM model performs better overall on the test set than the CNN-BiLSTM hybrid model. Bi-LSTM model achieves a test accuracy of 90.96%, a weighted F1 score of 91.71%, binary precision of 52%, binary recall of 76%, a ROC-AUC score of 91.65%, and a PR-AUC score of 73.29%. In addition, the Bi-LSTM model yields a lower false positive rate of 7.5% and a false negative rate of 23.62%. This model has better control over both types of errors. Although the binary precision remains relatively low, this is the outcome of the dataset's severe class imbalance, where there are around 9.5 times as many non-toxic (class 0) comments as toxic (class 1) comments. Due to the large volume of class 0 samples, even a low false positive rate results in a significant drop in precision for the minority class (toxic).

In conclusion, the Bi-LSTM model is the better choice in overall performance and model stability. It achieves a validation accuracy of 92.86% and an F1 score of 93.16%, with both binary precision and recall above 60%, at 60% precision and 73% recall. Compared to the CNN-BiLSTM model, it maintains a lower false positive rate (5.1%), but the false negative rate (26.83%) is a bit higher than CNN-BiLSTM model. Even though both models adopted the focal loss function to mitigate class imbalance, the Bi-LSTM model consistently outperforms the CNN-BiLSTM in terms of PR-AUC scores across both the validation and test sets.

Compare with Other Research Paper Results

Below here provides a comparison of the results with two research paper.

Table 21: Compare Model Result with (Naidu et al., 2023)

	This Report Model	**(Naidu et al., 2023) Model**
Accuracy	Bi-LSTM: 92.86% CNN-BiLSTM: 88.70%	(Naidu et al., 2023) LSTM: 88.75% LSTM+CNN: 92.63%

Compared to the (Naidu et al., 2023) using the same Jigsaw Kaggle competition dataset, the Bi-LSTM model of this report achieves higher validation accuracy of 92.86% compared to 88.75% of Naidu's LSTM model. Moreover, Naidu's LSTM+CNN model (92.63%)

outperformed the CNN-BiLSTM model (88.70%), indicating that applying the LSTM layer before the CNN layer yields better performance than the reverse order.

Table 22: Compare Model Result with (Dessì et al., 2021)

	This Report Model	**(Dessì et al., 2021) Model**
F1 Score	Bi-LSTM (FastText): 93.16% CNN-BiLSTM: 89.99%	Bi-LSTM (Mimicking): 91.50% CNN (Mimicking): 85%

Compared to the (Dessì et al., 2021) which used the same Jigsaw Kaggle competition dataset, but using different word embedding techniques. The comparison between the CNN single model and the CNN-BiLSTM hybrid model shows that the hybrid model has a better performance. However, the CNN-BiLSTM hybrid model performance not better than the Bi-LSTM single model. This suggests that Bi-LSTM can capture contextual information more effectively than CNN or CNN hybrid model in toxic comment classification tasks.

8.0 Conclusion

This study explored two deep learning models for toxic comment classification. First model is Bi-LSTM model andthe second is CNN-BiLSTM hybrid model, both model utilizing FastText word embeddings. Hyperparameter tuning was used Bayesian Optimization. To improve data quality, enhanced the text preprocessing step by replacing abbreviations and removing excessive repetitions within sentences. In addition, to handle the class imbalance problem both model are using focal loss function and prevent model overfitting problem by using early stopping and ReduceLROnPlateau to monitor validation loss during the model training. Among them, the Bi-LSTM model achieved the best performance, demonstrating a strong ability to capture contextual dependencies and better handle data is extremely unbalanced in toxic comments. For model can have a better in subtle and implicit toxic comments, in future model can incorporate emotion or sentiment features into the model, can use multi-task model or using concept-level fusion model. In addition, merging the current dataset with external datasets may help increase the proportion of minority class samples helps to improve the minority class's precision and recall. This could also result in a more stable validation curve during training. Enhancements such as attention mechanisms may also be considered to refine the model’s focus on key toxic cues within complex comments.

References

Akpatsa, S. K., Lei, H., Li, X., & Obeng, S. (2022). Sentiment Classification of Covid-19 Online Articles: The Impact of Changing Parameter Values on Bidirectional LSTM. *International Journal of Engineering Research & Technology*, *11*(1). https://doi.org/10.17577/IJERTV11IS010033

aldi-dimara. (2018). *keras-focal-loss/focal_loss.py at master · aldi-dimara/keras-focal-loss*. GitHub. https://github.com/aldi-dimara/keras-focal-loss/blob/master/focal_loss.py

Allen, C., & Hospedales, T. (2019). *Analogies Explained: Towards Understanding Word Embeddings*. https://proceedings.mlr.press/v97/allen19a/allen19a.pdf

Arhin, K., Baldini, I., Wei, D., Ramamurthy, K. N., & Singh, M. (2021, December 7). *Ground-Truth, Whose Truth? -- Examining the Challenges with Annotating Toxic Text Datasets*. ArXiv.org. https://arxiv.org/abs/2112.03529

Asrawi, H., Utami, E., & Yaqin, A. (2023). LSTM and Bidirectional GRU Comparison for Text Classification. *Sinkron*, *8*(4), 2264–2274. https://doi.org/10.33395/sinkron.v8i4.12899

Azam, M., Sadman Sakib, Nur Mohammad Fahad, Abdullah Al Mamun, Md. Anisur Rahman, Swakkhar Shatabda, & Hossain, S. (2024). A systematic review of hyperparameter optimization techniques in Convolutional Neural Networks. *Decision Analytics Journal*, *11*, 100470–100470. https://doi.org/10.1016/j.dajour.2024.100470

B.G.M. Vandeginste, Massart, D. L., L.M.C. Buydens, S. De Jong, Lewi, P. J., & J. Smeyers-Verbeke. (1998). Artificial Neural Networks. In *Data handling in science and technology* (pp. 649–699). Elsevier BV. https://doi.org/10.1016/s0922-3487(98)80054-3

Bonetti, A., Martínez-Sober, M., Torres, J. C., Vega, J. M., Pellerin, S., & Vila-Francés, J. (2023). Comparison between Machine Learning and Deep Learning Approaches for

the Detection of Toxic Comments on Social Networks. *Applied Sciences*, *13*(10), 6038. https://doi.org/10.3390/app13106038

Choi, D., Shallue, C., Nado, Z., Lee, J., Maddison, C., & Dahl, G. (2020). *On Empirical Comparisons of Optimizers for Deep Learning*. https://arxiv.org/pdf/1910.05446

Dessì, D., Recupero, D. R., & Sack, H. (2021). An Assessment of Deep Learning Models and Word Embeddings for Toxicity Detection within Online Textual Comments. *Electronics*, *10*(7), 779. https://doi.org/10.3390/electronics10070779

Dharma, E., Lumban Gaol, F., Leslie, H., Warnars, H., & Soewito, B. (2022). THE ACCURACY COMPARISON AMONG WORD2VEC, GLOVE, AND FASTTEXT TOWARDS CONVOLUTION NEURAL NETWORK (CNN) TEXT CLASSIFICATION. *Journal of Theoretical and Applied Information Technology*, *31*(2). http://www.jatit.org/volumes/Vol100No2/5Vol100No2.pdf

Garg, T., Masud, S., Suresh, T., & Chakraborty, T. (2023). Handling Bias in Toxic Speech Detection: A Survey. *ACM Computing Surveys*, 1–32. https://doi.org/10.1145/3580494

Jena, B., Saxena, S., Nayak, G. K., Saba, L., Sharma, N., & Suri, J. S. (2021, August 27). *Artificial intelligence-based hybrid deep learning models for image classification: The first narrative review* . Science Direct. https://www.sciencedirect.com/science/article/pii/S0010482521005977

Khan, M. A. (2023, November 8). *Determination of toxic comments and unintended model bias minimization using Deep learning approach*. ArXiv.org. https://arxiv.org/abs/2311.04789

Naidu, B. R., Tangudu, N., Sekhar, C., Kavitha, K., Ramana, B. V., Reddy, P. Venkateswarlu., Sahukaru, J., & Lopinti, R. G. (2023). Toxic Comment Classification using Deep Learning. *International Journal on Recent and Innovation Trends in*

Computing and Communication, *11*(7), 93–104. https://doi.org/10.17762/ijritcc.v11i7.7834

Pritom Mojumder, Hasan, M., Hossain, M. F., & Hasan, A. (2020). A Study of fastText Word Embedding Effects in Document Classification in Bangla Language. *Springer EBooks*, 441–453. https://doi.org/10.1007/978-3-030-52856-0_35

Richardson, E., Trevizani, R., Greenbaum, J. A., Carter, H., Nielsen, M., & Peters, B. (2024). The receiver operating characteristic curve accurately assesses imbalanced datasets. *Patterns*, *5*(6), 100994–100994. https://doi.org/10.1016/j.patter.2024.100994

Rimal, Y., Sharma, N., & Abeer Alsadoon. (2024). The accuracy of machine learning models relies on hyperparameter tuning: student result classification using random forest, randomized search, grid search, bayesian, genetic, and optuna algorithms. *Multimedia Tools and Applications*, *1-16*. https://doi.org/10.1007/s11042-024-18426-2

Rodríguez, A., & Buitrago, X. (2022). How to choose an activation function for deep learning Cómo elegir una función de activación para el aprendizaje profundo. *Howto Choose an Activation Function for Deep Learning*, *19*(1), 23–32. https://revistas.udistrital.edu.co/index.php/tekhne/article/download/20337/18805/123656#:~:text=In%20summary%2C%20there%20are%20a,and%20the%20vanishing%20gradient%20problem.

Sharkawy, A.-N. (2020). Principle of Neural Network and Its Main Types: Review. *Journal of Advances in Applied & Computational Mathematics*, *7*(1), 8–19. https://doi.org/10.15377/2409-5761.2020.07.2

Shinde, A., Shankar, P., Atul Atul, & Srikari Rallabandi. (2024). Bidirectional LSTM with convolution for toxic comment classification. *Bidirectional LSTM with Convolution for Toxic Comment Classification*, *1-13*. https://doi.org/10.4108/eai.23-11-2023.2343140

Song , G., Huang, D., & Xiao, Z. (2021). *A Study of Multilingual Toxic Text Detection Approaches under Imbalanced Sample Distribution*. 1–16. https://www.mdpi.com/2078-2489/12/5/205

Szandała, T. (2020). Review and Comparison of Commonly Used Activation Functions for Deep Neural Networks. *Bio-Inspired Neurocomputing*, *1-26*(19), 203–224. https://doi.org/10.1007/978-981-15-5495-7_11

Tian, Y., Wu, Q., & Zhang, Y. (2020). A Convolutional Long Short-Term Memory Neural Network Based Prediction Model. *INTERNATIONAL JOURNAL of COMPUTERS COMMUNICATIONS & CONTROL*, *15*(5). https://doi.org/10.15837/ijccc.2020.5.3906

Vinod, A., V, A. K., Manoranjan M, Ramsha Riyaz, & Mr. Arul N. (2024). Toxic Comment Detection and Classifier. *IJARCCE*, *13*(4). https://doi.org/10.17148/ijarcce.2024.134174

Wang, C., Nulty, P., & Lillis, D. (2020). A Comparative Study on Word Embeddings in Deep Learning for Text Classification. *Proceedings of the 4th International Conference on Natural Language Processing and Information Retrieval*, *1-10*. https://doi.org/10.1145/3443279.3443304

Wang, Z., & Zhang, B. (2021). Toxic Comment Classification Based on Bidirectional Gated Recurrent Unit and Convolutional Neural Network. *Toxic Comment Classification Based on Bidirectional Gated Recurrent Unit and Convolutional Neural Network*, *21*(3), 1–12. https://doi.org/10.1145/3488366

Yesi Novaria Kunang, Siti Nurmaini, Deris Stiawan, Yudho Bhakti, & Suprapto. (2021). Deep learning with focal loss approach for attacks classification. *TELKOMNIKA (Telecommunication Computing Electronics and Control)*, *19*(4), 1407–1418. https://doi.org/10.12928/TELKOMNIKA.v19i4.18772

Yu, L., & Zhou, N. (2021, April 6). *Survey of Imbalanced Data Methodologies*. ArXiv.org. https://arxiv.org/abs/2104.02240