INSY 5377 – Web and Social Analytics

Section 002

Project Report

Toxic Comments Classification – Social media Analytics

Team 1

Devi Chhatre
Dharshini Vasudevan
Sabarish Gopinath
Naga Sai Raghuveer Madireddy
Kritika Mishra

Executive Summary

- **Overview** – The toxic comments classification a project based on social media analysis aims to classify offensive and abusive language on a social media post and report the accuracy of these comments or words, thereby providing leading social media firms to create a safe environment for their users.

- **Problem Statement Summary** - Social media companies faced a lot of backlashes for explicit videos, images, and comments towards users and nowadays, social media companies are leaning towards providing a safer environment for their users. We build a model capable of detecting and classifying toxic comments using machine learning and neural networks. To predict the occurrence and accuracy of toxic comments using machine learning models.

- **Summary of Solution** – We make use of machine learning model with artificial neural network layers to train the model and obtain the best accuracy. Four different configurations were used to construct and train this model, each capable of reporting accuracy. We determined that the percentage of toxic comments/words existing in our dataset was 99.45% which was an increase from initial accuracy of 95.7%. Our aim with this was to create a safe environment for users.

Introduction

Online forums and web based social media provides individuals with the ability to put forward their thoughts and express their opinions on various incidents and in some cases, comments on social media contain explicit comments which may offend some users.

Social media companies faced a lot of backlashes for explicit videos, images, and comments towards users and nowadays, social media companies are leaning towards providing a safer environment for their users. According to a social media article today, social media applications like Instagram is rolling out moderation tools that uses machine learning to detect offensive language. This could mean the comments may be discouraging, ill-intentional or hateful towards users.

With the help of machine learning, we implement a toxic comments classifier using a dataset which contains both toxic and nontoxic words with the aim of creating a safer environment for users.

Problem Statement

To build a model capable of detecting and classifying toxic comments using machine learning and neural networks. To predict the occurrence and accuracy of toxic comments using machine learning models.

Dataset Description and Source

The dataset, which is a collection of toxic and non-toxic comments to implement this project was taken from a Kaggle challenge and can be found using the following link

https://www.kaggle.com/code/watermasterz/toxic-comments-classification/notebook

Contents of the dataset

a. Comment_text – The comments in English
b. Id – Acts as a comment id (not used in our implementation)
c. Identity_hate, insult, obscene, severe toxic, threat – The type of toxicity in the comment (descriptive field)
d. Set – Tells if the comment belongs to the training or the test set (not used in our implementation)
e. Toxicity – Measure of toxicity. Is a 0 or 1 variable field where:
    - 0 – means the comment is not toxic
    - 1- means the comment contains toxic content

Since our dataset is descriptive, no data cleaning tasks were performed. The dataset is divided into training and test set based on an encoding which assigns a padded sequence of numbers to each letter. This methodology is explained in detail in the upcoming section of our report. A screenshot of the first few rows of the dataset using the head command in python is given in the below screenshot.

Classification Code Starts Here :

In [19]:
```
import tensorflow as tf
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
dir = "train_preprocessed.csv"
```

In [20]:
```
data = pd.read_csv(dir)
data.head()
```

Out[20]:

	comment_text	id	identity_hate	insult	obscene	set	severe_toxic	threat	toxic	toxicity
0	explanation why the edits made under my userna...	0000997932d777bf	0.0	0.0	0.0	train	0.0	0.0	0.0	0.0
1	d aww he matches this background colour i m s...	000103f0d9cfb60f	0.0	0.0	0.0	train	0.0	0.0	0.0	0.0
2	hey man i m really not trying to edit war it...	000113f07ec002fd	0.0	0.0	0.0	train	0.0	0.0	0.0	0.0
3	more i can t make any real suggestions on im...	0001b41b1c6bb37e	0.0	0.0	0.0	train	0.0	0.0	0.0	0.0
4	you sir are my hero any chance you remember...	0001d958c54c6e35	0.0	0.0	0.0	train	0.0	0.0	0.0	0.0

Fig 1: Dataset Summary using 'head' command

Methodology

Since this is a toxic comments classifier, we use python programming libraries and packages. The project constructs machine learning models and makes use of neural networks to compare the models and predict the accuracy of the comments. A step-by-step approach as to how we solved the problem is described below.

Libraries and packages

A few libraries that were used for the purpose of the project and their description is given below:

- Tensorflow – Tensorflow is a free and open-source software library for machine learning and artificial intelligence. It can be used across a range of tasks but has a particular focus on training and inference of deep neural networks.

- Tensorflow.keras – Keras is an open-source software library that provides a Python interface for artificial neural networks. Keras acts as an interface for the TensorFlow library.

- NumPy – NumPy is a python library which mainly deals with operations that can be performed on arrays and the operations related to linear algebra, fourier transform, and matrices.

- Pandas - Pandas is a library in Python that deals with operations for data analysis. Pandas allows importing data from various file formats such as comma-separated values, JSON, SQL, Microsoft Excel. Pandas allows various data manipulation operations such as merging, reshaping, selecting, as well as data cleaning, and data wrangling features.

Instead of using NLP, to process the dataset, we used an encoding function to process the dataset. The collection of comments in the dataset is converted to a padded sequence of numbers to help train our model. We use a tokenizer function to carry out this operation. A brief explanation of the tokenizer function and the code used to execute the tokenizer function below.

Tokenizer Function

Using the function tokenizer, text to sequences and padded sequences, we were able to obtain an array of padded sequence of numbers which pertain to each token. Each number represents a unique value and hence using them to tarin the model will result in a better accuracy. Using numbers requires less work, whereas processing words takes more time. Hence these padded sequences of outputs are used to infer the accuracy of the model.

In [23]:
```
tokenizer = Tokenizer(num_words=NUM_WORDS)

# Fit the tokenizer on the comments
tokenizer.fit_on_texts(Features)

# Get the word index of the top 20000 words from the dataset
word_idx = tokenizer.word_index

# Convert the string sentence to a sequence of their numerical values
Feature_sequences = tokenizer.texts_to_sequences(Features)

# Pad the sequences to make them of uniform length
padded_sequences = pad_sequences(Feature_sequences, maxlen = MAXLEN, padding = PADDING)
```

In [24]:
```
print("The Transformation of sentence:")
print("\n\nThe normal Sentencen:\n")
print(Features[2])
print("\n\nThe tokenized sequence:\n")
print(Feature_sequences[2])
print("\n\nThe padded sequence:\n")
print(padded_sequences[2])

# Convert to array for passing through the model
X = np.array(padded_sequences)
```

Fig 2: Tokenizer Function

The above screenshot is the code of the tokenizer function which was used to convert texts to a padded sequence of numbers. The resultant output is a padded sequence of unique numbers assigned to each word in a matrix format.

```
The Transformation of sentence::


The normal Sentencen:

hey man  i m really not trying to edit war  it s just that this guy is constantly removing relevant information and talking to me through e
dits instead of my talk page  he seems to care more about the formatting than the actual info


The tokenized sequence:

[406, 415, 3, 69, 138, 14, 252, 2, 77, 317, 10, 17, 54, 9, 13, 591, 8, 2183, 493, 503, 107, 5, 598, 2, 39, 332, 131, 365, 4, 31, 38, 30, 51
, 214, 2, 429, 61, 40, 1, 2276, 97, 1, 717, 466]


The padded sequence:

[ 406  415    3   69  138   14  252    2   77  317   10   17   54    9
   13  591    8 2183  493  503  107    5  598    2   39  332  131  365
    4   31   38   30   51  214    2  429   61   40    1 2276   97    1
  717  466    0    0    0    0    0    0]
```

Fig 3: Output of the tokenizer function

The above screenshot describes the output of the tokenizer function. A sentence is given as an input and every word in the sentence is assigned a unique value/padded sequences which are stored in the form of a matrix. If a word occurs twice in a sentence at different positions, each of the word is assigned the number in the position of where the word occurs. These padded sequences are used to train the model.

The next step after tokenizing is to construct a matrix using this padded sequence of obtained numbers which are used to train the model. The code/function used for this is given in the screenshot below.

In [27]:
```
# initialize the word to index dictionary
word_2_vec = {}
with open("glove.6B.50d.txt", encoding='utf-8') as f:
    for line in f:

        # spilt the elements by space
        elements = line.split()
        word = elements[0]
        # convert to np array
        vecs = np.asarray(elements[1:], dtype='float32')
        word_2_vec[word] = vecs

print("Done....\n")
```

```
Done....
```

In [28]:
```
word_2_vec
```

Out[28]:
```
{'the': array([ 4.1800e-01,  2.4968e-01, -4.1242e-01,  1.2170e-01,  3.4527e-01,
        -4.4457e-02, -4.9688e-01, -1.7862e-01, -6.6023e-04, -6.5660e-01,
         2.7843e-01, -1.4767e-01, -5.5677e-01,  1.4658e-01, -9.5095e-03,
         1.1658e-02,  1.0204e-01, -1.2792e-01, -8.4430e-01, -1.2181e-01,
        -1.6801e-02, -3.3279e-01, -1.5520e-01, -2.3131e-01, -1.9181e-01,
        -1.8823e+00, -7.6746e-01,  9.9051e-02, -4.2125e-01, -1.9526e-01,
         4.0071e+00, -1.8594e-01, -5.2287e-01, -3.1681e-01,  5.9213e-04,
         7.4449e-03,  1.7778e-01, -1.5897e-01,  1.2041e-02, -5.4223e-02,
        -2.9871e-01, -1.5749e-01, -3.4758e-01, -4.5637e-02, -4.4251e-01,
         1.8785e-01,  2.7849e-03, -1.8411e-01, -1.1514e-01, -7.8581e-01],
       dtype=float32),
 ',': array([ 0.013441,  0.23682 , -0.16899 ,  0.40951 ,  0.63812 ,  0.47709 ,
        -0.42852 , -0.55641 , -0.364   , -0.23938 ,  0.13001 , -0.063734,
        -0.39575 , -0.48162 ,  0.23291 ,  0.090201, -0.13324 ,  0.078639,
        -0.41634 , -0.15428 ,  0.10068 ,  0.48891 ,  0.31226 , -0.1252  ,
        -0.037512, -1.5179  ,  0.12612 , -0.02442 , -0.042961, -0.28351 ,
         3.5416  , -0.11956 , -0.014533, -0.1499  ,  0.21864 , -0.33412 ,
        -0.13872 ,  0.31806 ,  0.70358 ,  0.44858 , -0.080262,  0.63003 ,
         0.32111 , -0.46765 ,  0.22786 ,  0.36034 , -0.37818 , -0.56657 ,
         0.044691,  0.30392 ], dtype=float32),
 '.': array([ 1.5164e-01,  3.0177e-01, -1.6763e-01,  1.7684e-01,  3.1719e-01,
         3.3973e-01, -4.3478e-01, -3.1086e-01, -4.4999e-01, -2.9486e-01,
         1.6608e-01,  1.1963e-01, -4.1328e-01, -4.2353e-01,  5.9868e-01,
         2.8825e-01, -1.1547e-01, -4.1848e-02, -6.7989e-01, -2.5063e-01,
         1.8472e-01,  8.6876e-02,  4.6582e-01,  1.5035e-02,  4.3474e-02,
        -1.4671e+00, -3.0384e-01, -2.3441e-02,  3.0589e-01, -2.1785e-01,
         3.7460e+00,  4.2284e-03, -1.8436e-01, -4.6209e-01,  9.8329e-02,
```

Fig 3: Function to convert the padded sequence into matrix

We then use neural network layers and machine learning to construct models and predict accuracy of the dataset. We make use of four different neural network layers to construct four models. The different layers and their functions are explained below.

- Densely Connected neural network layer – A densely connected layer is also known as a fully connected layer which performs linear operations on the layer's input vector. In this layer, a linear operation with an input is connected to every output by a weight. It is usually followed by a non-linear activation function.
- One dimensional convolution layer – This layer uses a subset of the weights as a linear operation from a dense layer nearby inputs are

connected to the nearby outputs and a weight for that convolution layer at each location is shared. Due to this weight sharing and the use of subset of the weights of the dense layer there are far less weights in the dense layer which leads to better accuracy when we use one dimensional convolution layers.

- LSTM layer – Stands for long short-term memory and it is a type of recurrent neural network that is capable of learning order dependence in sequence prediction problems. This means that even the order in which the training data is being input into the training function will also matter well training this model. These are usually used in complex problem domains like machine translation speech recognition etc.
- Embedding layer - is a part of the LSTM layer or you can say that it is used with the LSTM layer where it will learn an embedding for all of the words all data in the training data set. The embedding layer is initialized with random weights before this.
- Relu function – Is known as the rectifier activation function defined as the positive part of its argument.
- Sigmoid function - Sigmoid Function acts as an activation function in machine learning which is used to add non-linearity in a machine learning model, in simple words it decides which value to pass as output and what not to pass

The training code and the function which is used to train the model using the above-mentioned layers is given in the below screenshot. The function uses a sigmoid function to construct a graph which gives the visualization.

In [32]:

```
model = tf.keras.models.Sequential([

    # Embedding layers that takes in the embedding matrix. Be sure to set trainable to false or else it will mess up your
    # nicely pre trained vectors
    tf.keras.layers.Embedding(num, EMBEDDING_DIM, weights=[embedding_matrix], input_length=MAXLEN,trainable=False),
    tf.keras.layers.LSTM(20, return_sequences=True),
    tf.keras.layers.GlobalAveragePooling1D(),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
model.summary()
```

Fig 4: Code to train the model using the neural network layers

The next step in our process is to train our model. We constructed four models using 30 epoch values. 23 experiments were done with 30 epoch values and a fit model was constructed. A screenshot containing the code which was used to train the model is presented in the screenshot below. The matplot library is used to construct a visualization which makes use of the sigmoid function. The accuracy initially began at 95% and we were able to attain an accuracy of 99.5% by carrying out different experiments and trials.

The next section of the code is the training function which builds the model and trains the same using the neural network layers. We make use of 30 epochs and run 23 experiments/trials on each epoch for each model which determines the accuracy for every epoch value. A graphical plot of epoch V.S validation accuracy is plotted to determine and compare the accuracy of each of the model. The code snippet is presented below.

In [31]:

```
def train(models, epochs, graph=True, verbose=2):
    n = 1
    plt.figure(figsize=(10, 7))

    histories = []
    for model in models:
        print(f"model number : {n} is training")
        model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['acc'])

        history = model.fit(
            X,
            Labels,
            batch_size=128,
            epochs=epochs,
            validation_split=0.1, # 10 percent data reserved for validation to avoid or monitor overfitting/ underfitting
            verbose=verbose,
        )
        histories.append(history)

        if graph:
            plt.plot(history.history['val_acc'], label=f"Model {n}")
        n+=1

    plt.xlabel('Epochs')
    plt.ylabel('Validation Accuracy')
    plt.legend()
```

Fig 5: Building and training the classifier model

As mentioned above, the code written builds and trains the model and plots a graph between epochs and validation accuracy. We have used bar charts and a line graph to compare the models.

After the train function, we focused on training each model individually. Four separate models were created, and each model was trained individually using different configuration of the neural network layers. The final model that we created used 30 epochs and a 10% validation set was used from the training set. The four neural network models used in building and training the model are densely connected layer, LSTM layer, 1D convolution layer and embedding layer. The screenshot of the function below contains the code which was used to train the first model.

The four densely connected use a relu function as an activation function and the densely connected layer uses the sigmoid function as an activation function.

In [32]:

```
model = tf.keras.models.Sequential([

    # Embedding layers that takes in the embedding matrix. Be sure to set trainable to false or else it will mess up your
    # nicely pre trained vectors
    tf.keras.layers.Embedding(num, EMBEDDING_DIM, weights=[embedding_matrix], input_length=MAXLEN,trainable=False),
    tf.keras.layers.LSTM(20, return_sequences=True),
    tf.keras.layers.GlobalAveragePooling1D(),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
model.summary()
```

```
Model: "sequential_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_4 (Embedding)     (None, 50, 50)            2000000

 lstm_4 (LSTM)               (None, 50, 20)            5680

 global_average_pooling1d_4   (None, 20)               0
 (GlobalAveragePooling1D)

 dense_12 (Dense)            (None, 1)                 21

=================================================================
Total params: 2,005,701
Trainable params: 5,701
Non-trainable params: 2,000,000
_________________________________________________________________
```

Fig 6: Training function of the first model

In the above screenshot we have used the four different neural network layers and the activation function to train the first model. The number 20 inside the LSTM layer represents the number of units in that layer and the number 20 specifically was obtained after multiple trials.

In [33]:

```
model2= tf.keras.models.Sequential([
    tf.keras.layers.Embedding(num, EMBEDDING_DIM, weights=[embedding_matrix], input_length=MAXLEN,trainable=False),
    tf.keras.layers.LSTM(100, return_sequences=True),
    tf.keras.layers.GlobalAveragePooling1D(),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(80, activation='relu'),
    tf.keras.layers.Dense(20, activation='relu'),
    tf.keras.layers.Dense(5, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
model2.summary()
```

```
Model: "sequential_5"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_5 (Embedding)     (None, 50, 50)            2000000

 lstm_5 (LSTM)               (None, 50, 100)           60400

 global_average_pooling1d_5   (None, 100)              0
 (GlobalAveragePooling1D)

 dense_13 (Dense)            (None, 128)               12928

 dense_14 (Dense)            (None, 80)                10320

 dense_15 (Dense)            (None, 20)                1620

 dense_16 (Dense)            (None, 5)                 105

 dense_17 (Dense)            (None, 1)                 6

=================================================================
Total params: 2,085,379
Trainable params: 85,379
Non-trainable params: 2,000,000
_________________________________________________________________
```

Fig 7: Training function of the second model

The training function of the second model in the screenshot consists of 8 layers one of them being embedding layer, The second being a LSTM layer, Thought being a 1D-Pooling layer and the rest of the five layers being densely connected Neural network layers. The units inside the densely connected neural network layers and the units inside the LSTM were obtained after many experiments were conducted.

The third model includes an embedding layer, an LSTM layer, a 1D-convolutional layer, a 1D-Pooling layer and five densely connected neural network layers. The units inside the densely connected neural network layers and the units inside the LSTM were obtained after many experiments were conducted.

The fourth model uses the same configuration as the first mode but contains an embedding layer instead of the LSTM layer.

In [34]:

```
model3= tf.keras.models.Sequential([
    tf.keras.layers.Embedding(num, EMBEDDING_DIM, weights=[embedding_matrix], input_length=MAXLEN,trainable=False),

    tf.keras.layers.LSTM(100, return_sequences=True),
    tf.keras.layers.Conv1D(10,15, activation='relu'),

    tf.keras.layers.GlobalAveragePooling1D(),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(80, activation='relu'),
    tf.keras.layers.Dense(20, activation='relu'),
    tf.keras.layers.Dense(5, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])

model3.summary()
```

```
Model: "sequential_6"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_6 (Embedding)     (None, 50, 50)            2000000

 lstm_6 (LSTM)               (None, 50, 100)           60400

 conv1d_1 (Conv1D)           (None, 36, 10)            15010

 global_average_pooling1d_6   (None, 10)               0
 (GlobalAveragePooling1D)

 dense_18 (Dense)            (None, 128)               1408

 dense_19 (Dense)            (None, 80)                10320

 dense_20 (Dense)            (None, 20)                1620

 dense_21 (Dense)            (None, 5)                 105

 dense_22 (Dense)            (None, 1)                 6

=================================================================
Total params: 2,088,869
Trainable params: 88,869
Non-trainable params: 2,000,000
_________________________________________________________________
```

Fig 8: Training function of model 3

In [35]:

```
model4= tf.keras.models.Sequential([

    # Embedding layers that takes in the embedding matrix. Be sure to set trainable to false or else it will mess up your
    # nicely pre trained vectors
    tf.keras.layers.Embedding(num, EMBEDDING_DIM, weights=[embedding_matrix], input_length=MAXLEN,trainable=False),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(20, return_sequences=True)),
    tf.keras.layers.GlobalAveragePooling1D(),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
model4.summary()
```

```
Model: "sequential_7"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_7 (Embedding)      (None, 50, 50)            2000000

bidirectional_1 (Bidirectio  (None, 50, 40)            11360
nal)

global_average_pooling1d_7   (None, 40)              0
(GlobalAveragePooling1D)

dense_23 (Dense)             (None, 1)                 41

=================================================================
Total params: 2,011,401
Trainable params: 11,401
Non-trainable params: 2,000,000
_________________________________________________________________
```

Fig 9: Training function of model 4

After training all the four models individually, we then initialize the function to train all the models with 30 epochs and obtain the accuracy of each model. We have attained an accuracy of 99.5% after conducting 23 experiments using 30 epochs. The function is given in the screenshot below.

In [36]:

```
models = [model1,model2,model3,model4]
train(models, epochs=30)
```

Fig 10: Execution of the training models

Results and Discussion

The accuracy of the models using the LSTM layer is obtained. The first section is the output obtained using the matplot library and a sigmoid function which is plot between validation accuracy and epoch values to determine the accuracy of each model. Bar charts are the second section and an individual chart for each model is presented below.

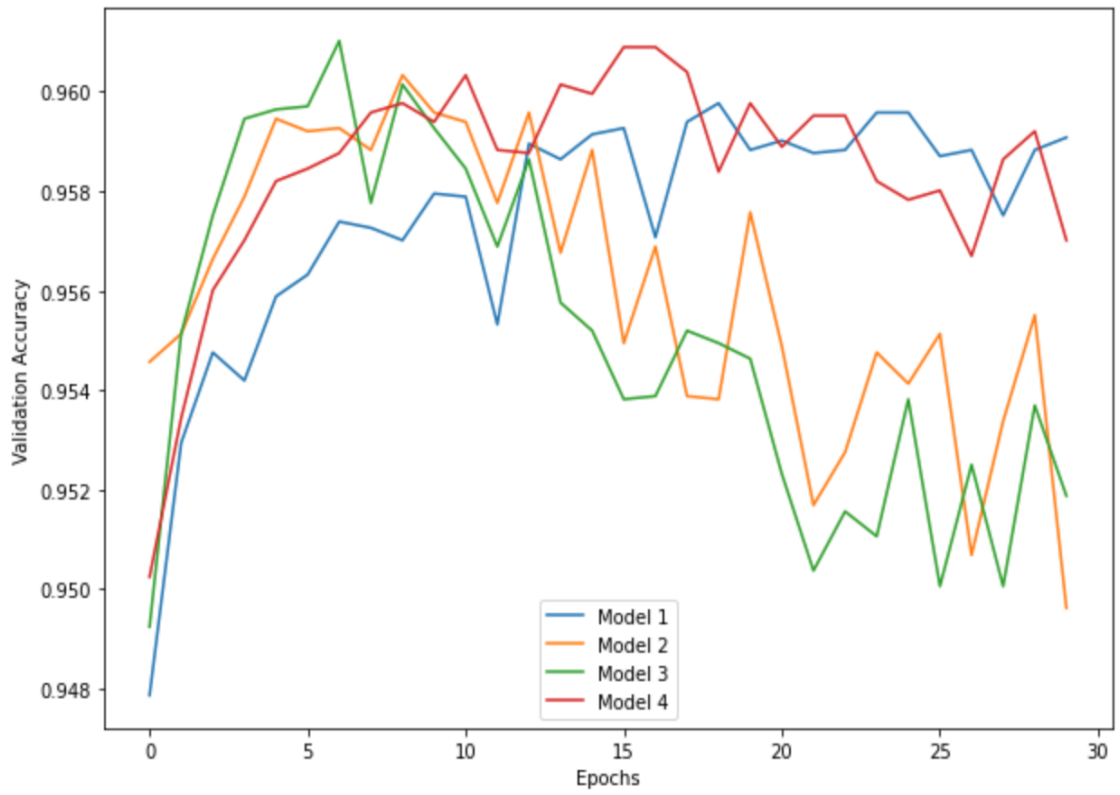

Fig 11: Results of Validation Accuracy vs Epoch

The above screenshot gives the accuracy of the model. The graph plot is between validation accuracy and epoch. A validation set of 7.5% was used with 30 epochs.

From the above graph, it can be inferred that the third model gives the best accuracy from the dataset with an accuracy of 99.45% and we use this model to argue that maximum accuracy is attained.

The four bar charts of the models constructed are presented below. The bar chart can be used for simple interpretation/explanation of the results of each of the four models.

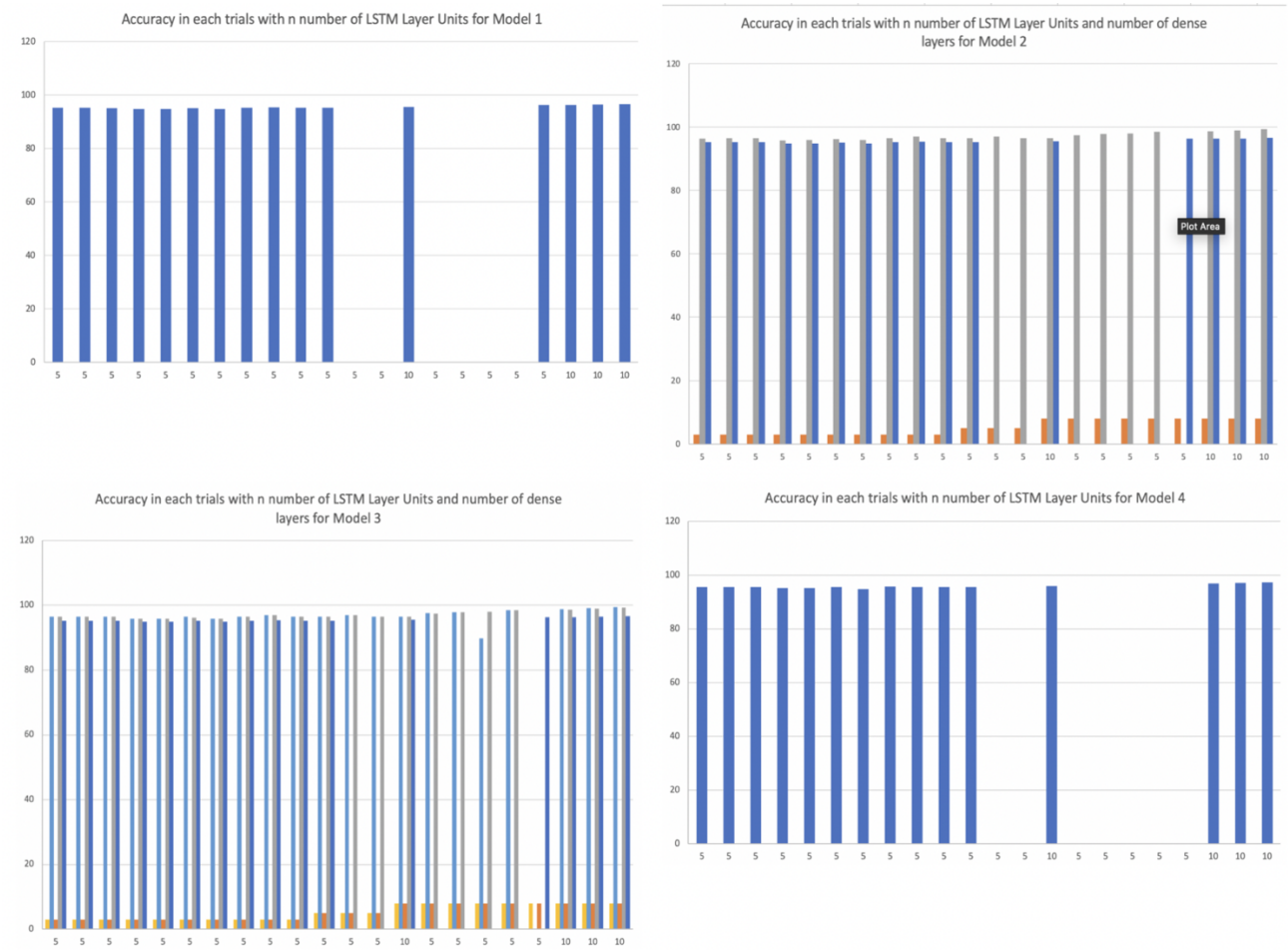

Fig 12: Bar charts of epochs VS validation accuracy (4 models)

The above bar graphs show the accuracy reported by each model after training them. The first and the fourth bar chart presented, are similar as we used similar configuration of the neural network layers to train the model. Model 1 and model 4 give us the highest accuracy which is 99.45%. Hence, we can derive that the third model gives an accurate prediction. The summary of this graph and the sigmoid function are compiled into a table which contains the configuration used for each model and the size of the training and the validation set. The sigmoid function and the tabular summarization are presented below.

Model	Trial Number	Epochs	Training+Testing Set Size	Validation Set Size	LSTM Layer Units (For Model 2 and 3)	Number of Dense Layers (For Model 2 and 3)	Accuracy
1	Final Trial	30	92.5%	7.5%	10	NA	**96.58**
2	Final Trial	30	92.5%	7.5%	100	8	**99.33**
3	Final Trial	30	92.5%	7.5%	100	8	**99.45**
4	Final Trial	30	92.5%	7.5%	10	NA	**97.23**

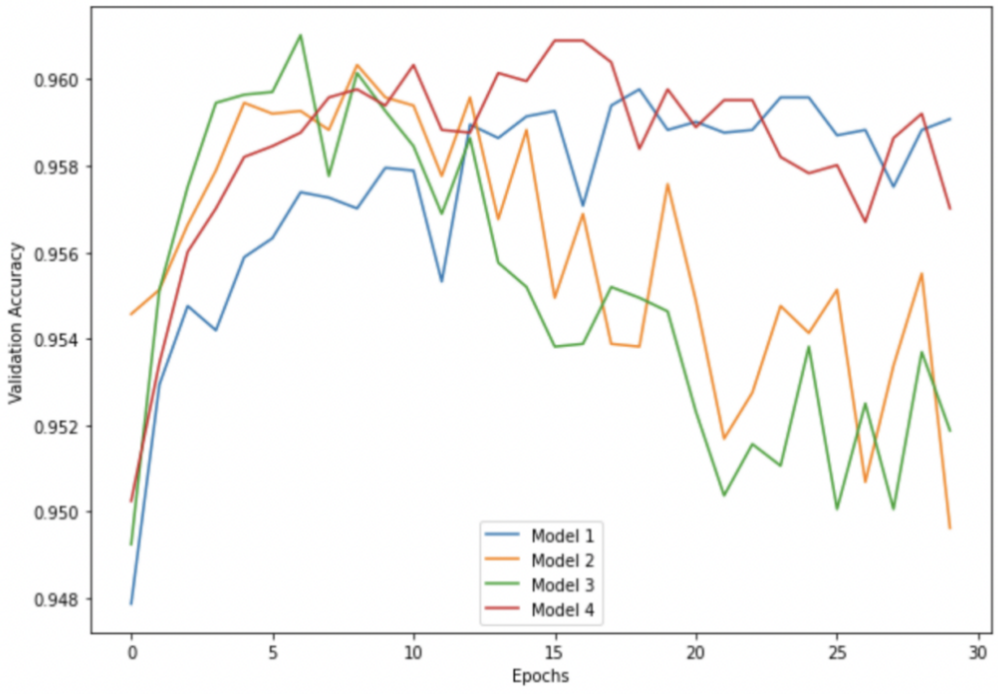

Fig 13: Summary of tabular output and sigmoid function

From the above obtained results, we can compare the accuracy of four models and choose the best model for deployment. From our observation, model 3 gives us the best accuracy from the dataset. A configuration similar to the third model can be deployed in a real time application with the aim of providing a safer environment for the user.

Hence with the reported accuracy of 99.45% we can conclude that this model can be implemented on social media platforms to classify toxic contents and hence making social media platforms a safe place for users.

Conclusion

The toxic comments classifier, an application which analyzes social media can be used to address the occurrences of toxic comments in a particular user comment on social media. With the help of machine learning and neural networks, a classification model was constructed that gives the accuracy of toxic words. The accuracy predicted can be used by organizations to provide a much safer platform to their users.

The future scope for the project can be implementing an extension on media platforms which can be used to filter out such comments there by providing a better experience for user.

The filter can aim to remove or censor out such content. Another extension can be explicit image content classification where a user can be warned that the image, they are about to view might have sensitive content or words in them.

References

The following are the references that were used to execute the project.

- https://www.kaggle.com/watermasterz/toxic-comments-classification/notebook
- https://whatis.techtarget.com/definition/validation-set
- https://machinelearningmastery.com/gentle-introduction-long-short-term-memory-networks-experts/

- https://machinelearningmastery.com/use-word-embedding-layers-deep-learning-keras/
- https://en.wikipedia.org/wiki/Keras
- https://en.wikipedia.org/wiki/Pandas_(software)
- https://en.wikipedia.org/wiki/TensorFlow