Data Science Intern at Data Glacier

Project: Hate Speech Detection using Transformers (Deep Learning)

Team Member: Manhui Zhu

Email: zmanhui09@outlook.com

Country: China

College: University of Southern California

Specialization: Data Science

Table of Contents

1. Project Plan

Weeks	Date	Deliverables
Week 7	June 19, 2024	Problem Statement, Data Intake Report, Project Plan
Week 8	June 26, 2024	Data Preprocessing
Week 9	July 2, 2024	EDA (Exploratory Data Analysis)
Week 10	July 9, 2024	Feature Extraction
Week 11	July 16, 2024	Model Building and Training
Week 12	July 23, 2024	Model Performance Evaluation
Week 13	July 30, 2024	Final Submission (Slides + Report + Code)

2. Problem Statement

The term hate speech is understood as any type of verbal, written or behavioral communication that attacks or uses derogatory or discriminatory language against a person or group based on based on their religion, ethnicity, nationality, race, color, ancestry, sex or another identity factor. To detect hate speech from various free speech, we will take you through a hate speech detection model with Machine Learning and Python. Then we will deploy our ML model on Flask to create a web application.

Hate Speech Detection is generally a task of sentiment classification. A model that can classify hate speech from a certain piece of text can be achieved by training it on a dataset that is generally used to classify sentiments. For the task of hate speech detection model, we will use the Twitter tweets to identify tweets containing Hate speech.

3. Data Intake Report

Name: Twitter Hate Speech
Report date: 06/19/2024
Internship Batch: LISUM33
Version: 1.0
Data intake by: Manhui Zhu
Data Intake reviewer: Data Glacier
Data Storage location: https://github.com/Manhui-z/Data-Glacier-Internship/tree/0083a551094656a2b96e6b2b64fd353394d34756/Week%207

Tabular data details:

Name of data	hate_speech.csv
Total number of observations	31962
Total number of features	3
Base format of the file	.csv
Size of the data	2.95 MB

Proposed Approach:

- The full dataset is consisting of 3 features: `id` with data type int64, `label` with data type int 64, and `tweet` with data type object.
- There is no missing value in the dataset.

4. Data Preprocessing

There are mainly 4 approaches to transform raw text into a structured format, making it easier for models to analyze and learn from the data. They are text cleaning, removing stop words, tokenization, and lemmatization.

4.1 Text Cleaning

The usual tweets have many causal colloquial expressions, special characters, and emojis. These messy texts make it difficult for the model to learn the underlying pattern and classify the hate speech and non-hate speech. Therefore, we need to clean the text first before we fit data into the model for training. Here are detailed steps.

- **Lowercasing:** For the same words apple and Apple, the computer will recognize them as different words. To avoid this from happening, we need to all words in lowercase.

- **Removing User Mentions:** @Users is used when we mentioned someone in our tweets. It usually doesn't have any special meanings, so we remove @Users by using `re` (regular expression) package.
- **Removing URLs:** Since the model cannot directly interpret whether the content represented by the URLs is problematic, it is not helpful for the model training, so we remove it.
- **Removing Special Characters:** For better text understanding, we remove special characters in tweets by keep only letters and whitespace. Number doesn't have special meaning, so we remove all digits.
- **Removing leading and trailing whitespace:** we remove meaningless whitespace before and after each tweet.

4.2 Removing Stop Words

Stop Words are some high-frequency common words in English language expression like 'and', 'the', 'is', but they may not contribute to the meaning and context of the sentence. We use `nltk` library to help remove the stop words. This reduces number of words the model needs to handle so that models can focus on more meaningful words, which improves efficiency and reduce noise.

4.3 Tokenization

Tokenization splits text into smaller units called token, it is usually in units of words. It is the foundation for other steps in NLP task, like stemming, lemmatization, and vectorization, which take tokens as their input.

4.4 Lemmatization

Lemmatization reduces words to their base or root form by considering the context and meaning of the word. It leads to better consistency in features, reduces redundancy, and helps in normalizing text, which is useful for classification task that requiring semantic understanding.

5. EDA (Exploratory Data Analysis)

In this part, we do the simple analysis to understand the data's structure and patterns. We check the number of total words in tweets and find the most frequents words appeared in hate and non-hate tweets. Then, we check the class distribution and use resampling technique to solve the imbalance issue.

5.1 Number of Total Words and Stop Words

We check total number of words and stop words in each tweet. From figure 1, we can find that more than half of tweets contain between 10 and 20 words.

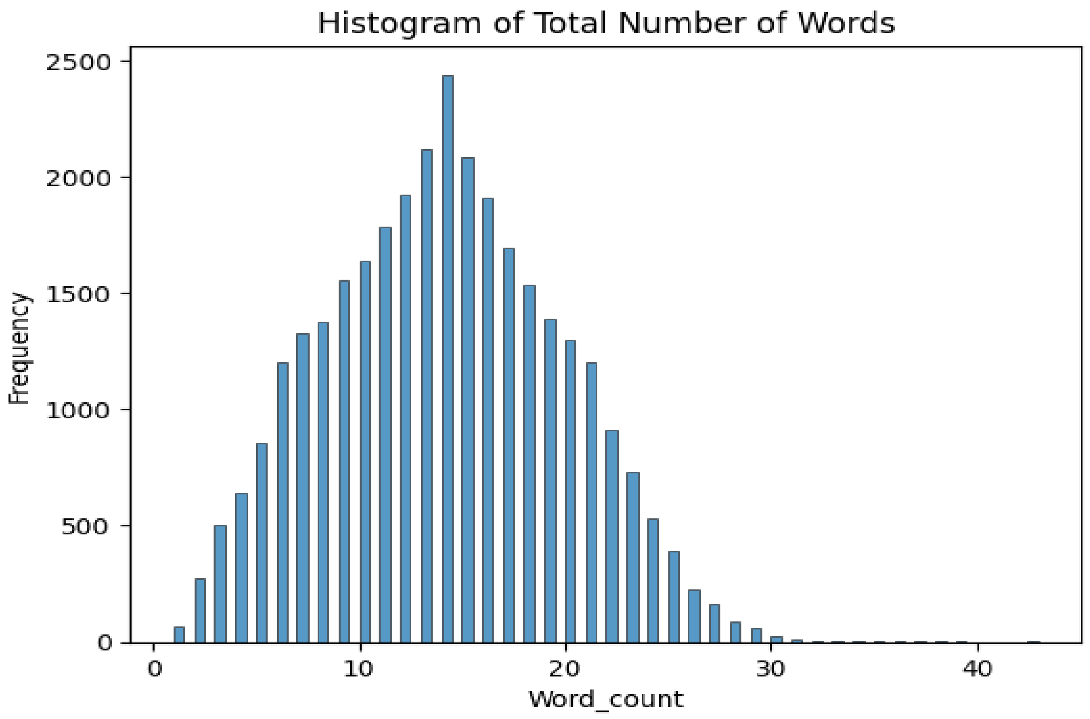


Figure 1. Histogram of Total Number of Words in Each tweet

5.2 – 5.3 The Most Frequent Words in the Whole Dataset, Hate Tweets, and Non-hate Tweets

After removing the stop words, the top 10 frequent words in the whole dataset are ‘love’, ‘day’, ‘happy’, ‘amp’, ‘im’, ‘u’, ‘life’, ‘time’, ‘like’, ‘today’. The figure 2 shows the top 10 frequent words in hate and non-hate speech are in the following graph. The word ‘amp’ appeared frequently in both classes. But it only means the symbol & (an ampersand), which usually appeared in HTML codes. Since our dataset may collected by directly web scraping from Twitter, it may convert symbol ‘&’ into string ‘amp’. Therefore, this word does not have significant impact for the model’s understanding of the context and meaning of tweet. We can either keep it or remove it.

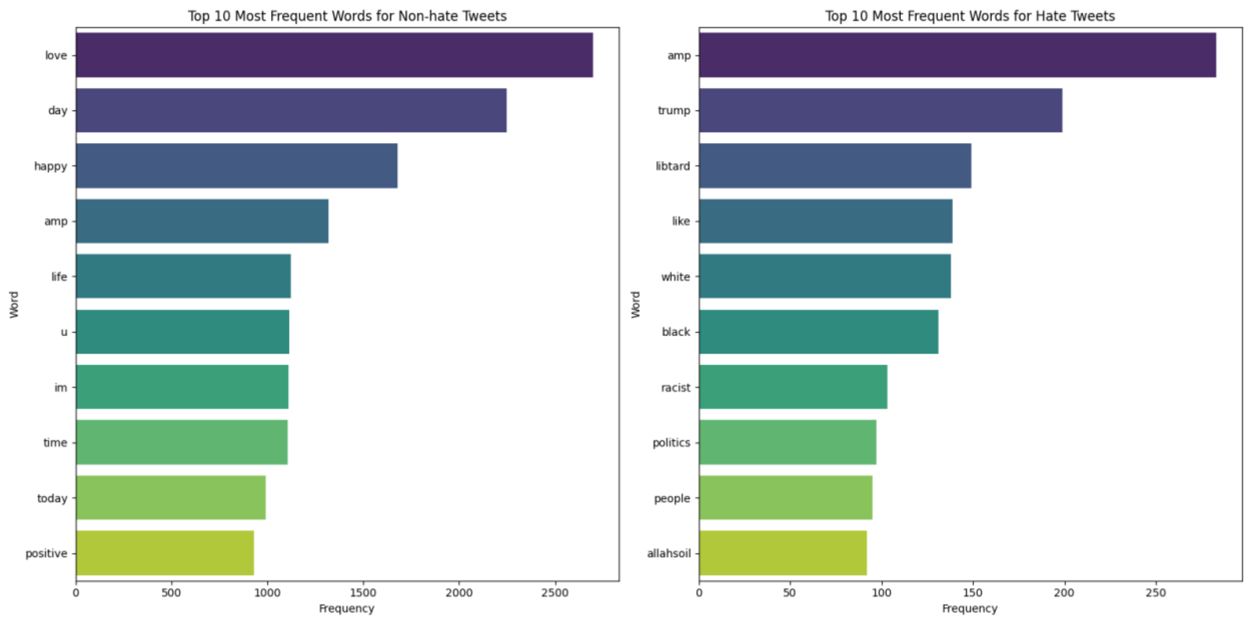


Figure 2. Top 10 Most Frequent Words for Non-hate Tweets/Hate Tweets

5.4 Word Clouds for Each Label

To better visualize the high-frequent words in each class, we plot the word cloud for each class and display in figure 3. The larger the size of word, the more frequently it occurs. The graph below shows that *'love', 'time', 'day', 'happy'*, 'life' are high frequent words in non-hate tweets. These words are either positive or at least neutral. While in hate tweets, the high-frequent words are *'trump', 'white', 'politics', 'black', and 'racist'*. These words are associated with politics and race, which is usually controversial and prone to disputes.

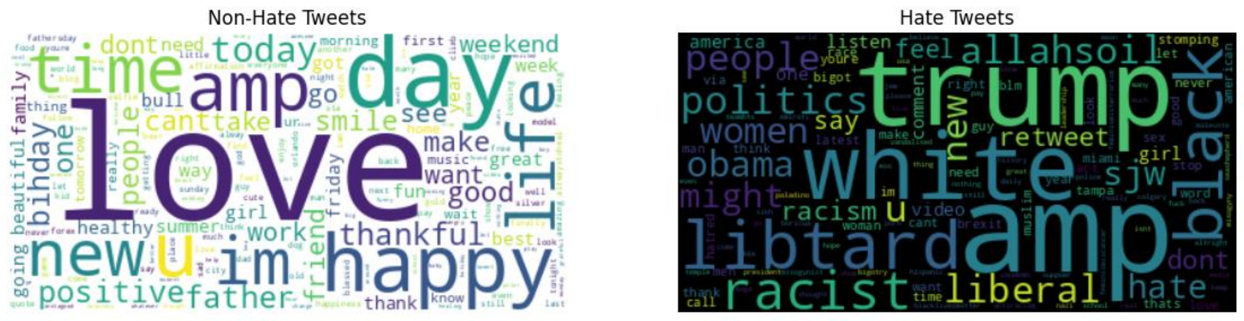


Figure 3. Word Cloud of Non-hate Tweets/Hate Tweets

5.5 – 5.6 Data Distribution & Solving Imbalance Problem

There are 29720 non-hate tweets and 2242 hate tweets. It indicates the imbalance in our dataset. The models trained on imbalanced data often become biased towards the major class, resulting in incorrectly classifying minority class instances. It will also cause underfitting of the minority class, leading to poor model performance on minority instances. To avoid this problem, we resample the minority samples (label = 1) with replacement so that it reaches the same number of samples of majority class (label = 1). After resampling, we have 29720 samples in each class.

6. Feature Extraction

In this section, we transform raw text data into structured numerical representations that machine learning models can process. By extracting meaningful features, we can reduce the data dimension, making the model more efficient and faster to train. Here are techniques we used to do the feature extraction.

6.1 Bag of Words (BoW)

This method converts text into a matrix of token counts (numerical representations), ignoring grammar and word order but keeping multiplicity. The intuition behind this method is that similar text fields will contain similar kind of words, therefore having similar bag of words. It has three steps:

- **Tokenization**: Splitting the text into individual words and tokens
- **Vocabulary**: Creating a set of unique words (tokens) from the entire text corpus
- **Vectorization**: Converting each document into a vector where each dimension corresponds to a word from the vocabulary and the value represents the word's frequency in the document.

6.2 TF-IDF (Term Frequency – Inverse Document Frequency)

This method evaluates the importance of a word in a document relative to a collection of documents (corpus), reducing the weight of common words.

- **Term Frequency (TF):** Measures how frequently a term occurs in a document. The assumption is that terms that appear more frequently within a document are more important.

$$TF(t,d) = \frac{\text{Number of term } t \text{ appears in document } d}{\text{Total number of terms in document } d}$$

- **Inverse Document Frequency (IDF):** Measures how important a term is within the entire corpus. It helps reduce the weight of terms that occur very frequently in many documents and increase the weight of terms that occur rarely.

$$IDF(t, D) = \log \left(\frac{\text{Total Number of documents}}{\text{Number of documents containing term } t}\right)$$

- The TF-IDF score is the product of TF and IDF:

$$TF - IDF(t, d, D) = TF(t, d) \times IDF(t, D)$$

For both BoW and TF-IDF feature extraction method, we set the parameter *`max_features = 4000`*, it limits the feature space to the most 4000 important terms, reducing the computational complexity and memory usage of the model.

6.3 Word Embeddings (*Word2Vec*)

This method represents words in a dense vector space, capturing semantic relationships. Words with similar meanings are located close to each other in this vector space.

Word embeddings are typically trained on large text corpora using neural networks. Here, we use the pre-trained `Word2Vec` models developed by Google (`GoogleNews-vectors-negative300.bin`), which contains vectors for a large vocabulary of around 3 million words and phrases. Each word vector is represented in a 300-dimensional space.

Twitter language is full of slang, abbreviations, and emojis. Word2Vec can learn these informal and evolving expressions, leading to accurate detection of hate tweets.

6.4 BERT (Bidirectional Encoder Representations from Transformers)

BERT is a pre-trained transformer-based model designed for a wide range of NLP tasks. It is known for its ability to understand the context of a word in a sentence, reading text in both direction (left-to-right and right-to-left). Unlike traditional models that only look at a word's surrounding context in a single direction, BERT looks at the words that come before and after it.

- **Transformer:** it applies self-attention mechanism, allowing the model to weight the importance of different words in a sentence when encoding a particular word. It enables the model to capture long-range dependencies.
- **Positional Encoding:** Since transformers do not have any built-in notion of word order, positional encodings are added to the input embeddings to provide information about the position of each word in the sequence.
- **Encoder:** it processes the input text and generates a contextualized representation. The original transformer model consists of an encoder and a decoder, BERT only use the encoder, its architecture involves stacking multiple transformer encoders, which allows it to build rich contextual representations of text. Each encoder layer consists of:

- Multi-head self-attention mechanism: it enables the model to focus on different parts of the sentence simultaneously, capturing various aspects of the relationships between words.
- Feed-Forward Neural Networks (FFNN): it applies to each position separately and identically. FFNN layers introduces non-linearities and transforms the feature extracted by the self-attention mechanism, increasing the model's capacity to capture intricate relationships in the data.
- Add and normalize operations for layer normalization and residual connections: it stabilizes training and allows gradients to flow through the network more effectively.

If receiving a batch of string that are too large, the BERT tokenizer cannot process at once. So we first break the large dataset into small batches with size 32 samples. The final embedding vector for each tweet has 768 dimensions, which aligns with the expectation of BERT base model.

7. Model Building and Training

We build both traditional machine learning models (Random Forest, XGBClassifier) and deep learning models (LSTM, Transformer (Fine-tune BERT)) to do the classification task. Based on the capacity of the models to utilize features effectively, we will use different feature extraction techniques.

For traditional machine learning models:

- It is suitable to use Bag of Words (BoW) and TF-IDF techniques. They produce sparse, high-dimensional vectors, focus on word frequency but do not capture semantic meaning or context. Traditional machine learning models can handle high-dimensional, sparse data effectively.
- Traditional machine learning models work well with linear relationships between features, which BoW and TF-IDF often provide.
- These models are usually computationally less intensive, making them suitable for simpler algorithms that can handle large feature spaces.

For deep learning models:

- It is suitable to use Word Embeddings and BERT techniques. They generate dense, low-dimension vectors where each word is represented by a vector that captures semantic relationships and context.
- Deep learning models can capture and learn from the semantic and contextual information encoded in the embeddings.

- Deep learning architectures can handle the non-linear, intricate relationships and interactions between features that embeddings provide. The transformer-based models are designed to handle and benefit from the rich, contextual information that BERT provides.

Based on the different nature and complexity of features, we decide to use TF-IDF features for traditional machine learning models, and Word Embeddings features for LSTM models and BERT features for Transformer (fine-tune BERT).

7.1 Random Forest & XGBClassifier

These two models are traditional machine learning models. Both are tree-based method, the Random Forest uses bagging, while XGBClassifier uses boosting.

Random Forest model creates multiple decision trees using different subsets of the data (bootstrapping technique: sampling with replacement) and takes a majority vote (for classification task). At each split in a tree, a random subset of features is selected to find the best split, which introduces randomness and reduces overfitting. We add parameter *`n-estimators = 100`* when constructing the model. It means that the random forest will consist of 100 individual decision trees.

The XGBClassifier is part of the XGBoost (Extreme Gradient Boosting) library, which is gradient-boosted decision trees. This model sequentially builds trees where each tree tries to correct the errors of the previous trees. It optimizes an objective function (often a combination of a loss function and a regularization term) using gradient descent. Different from random forest, all features are considered at each split, but the algorithm focuses on features that reduce the loss the most. We also add parameter *`n-estimators = 100`* in model construction. Each tree is trained on a random subset of the training data, it contributes to the overall robustness and performance of the model.

We split the dataset into training (80%) and test (20%) sets. The test accuracy is 98.95% for random forest model and 90.16% for XGBClassifier.

7.2 LSTM (Long Short-Term Memory)

LSTM models is a neural network that can capture and understand the context and dependencies within text sequences like tweets. It is introduced to address the vanishing gradient problem of RNNs (Recurrent Neural Network) by incorporating a more complex memory cell structure.

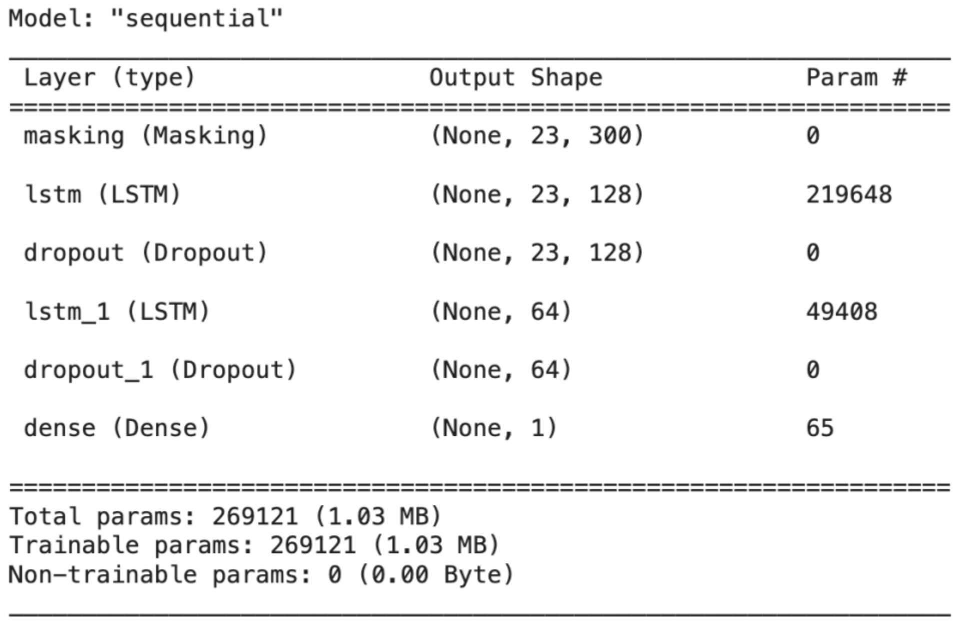
LSTM networks use a memory cell that can maintain information over long periods of time, allowing them to learn and remember long-term dependencies in data. The gating mechanisms (input gate: add/update new information, forget gate: forget irrelevant information, output gate: pass updated information) that control the flow of information.

Different tweets can have varying lengths. However, LSTM model requires input sequences (sequence length: number of words/tokens in each tweet) to be the same length for batch processing. So we use *`pad_sequences`* method to ensures all tweets are of uniform length.

`MAX_SEQUENCE_LENGTH` is defined to set a fixed length for all sequences. After removing the stop words, the maximum sequence length of tweets is 23 (tokens/words). So we set *`max_sequence_len = 23`*. We also add *`padding='post'`* parameters, if a sequence is shorter than 23, it will be padded with zeros at the end of the sequence.

We split the dataset into the training (80%) and test (20%) sets. But in each epoch, we randomly picked 20% samples from the training data as the validation set during training process.

Figure 4 is the LSTM model architecture. We have two hidden layers (LSTM layer), the first one has 128 hidden neurons, and the second on has 64 hidden neurons. After each hidden layers, we add a dropout layers. A randomly picked 20% of the hidden neurons in the previous layers will be ignored. Training stops at the 10th epochs, and the highest validation accuracy is 98.55%. The test accuracy is 98.47%.



```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 masking (Masking)           (None, 23, 300)           0

 lstm (LSTM)                 (None, 23, 128)           219648

 dropout (Dropout)           (None, 23, 128)           0

 lstm_1 (LSTM)               (None, 64)                49408

 dropout_1 (Dropout)         (None, 64)                0

 dense (Dense)               (None, 1)                 65

=================================================================
Total params: 269121 (1.03 MB)
Trainable params: 269121 (1.03 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Figure 4. Architecture of LSTM Model

7.3 Transformer-based Model (BERT)

Transformer (BERT) model uses self-attention mechanisms to understand the context of a word based on its surrounding words, making them highly effective for text classification. Here we fine-tune the pre-trained BERT model on our specific dataset and task. Fine tuning involves updating weights of the pre-trained BERT model based on my dataset, allowing it to adapt to our hate speech detection task.

First, we do the data split. Since the `*validation_split*` parameter in `*model.fit*` function is not supported for TensorFlow datasets created using the `*tf.data.Dataset*` API. We need to manually split your dataset into training and validation sets before creating the `*tf.data.Dataset*` objects. In this approach, the validation set is independent of the training set, providing a more rigorous evaluation. The final data split is 60% for training set, 20% for validation set, and the last 20% for test set.

When preparing data for a BERT model, the text undergoes several transformations to convert it into a format suitable for model input. These transformations ensure that the text is tokenized correctly, padded to the required length, and accompanied by attention masks and segment tokens where necessary. This process helps BERT understand and process the input text effectively, leading to better model performance. Input IDs and attention masks are two important components for model training.

- **Input IDs**: they are the token IDs obtained from the tokenizer, which represent the input text in a format that the BERT model can process.
- **Attention Masks**: they are binary masks that help the model to differentiate between real tokens and padding tokens (Padding tokens are added to ensure that all input sequences in a batch are of the same length). Attention masks allow the model to focus on the real input tokens and ignore the padding tokens during processing, enabling efficient attention mechanism and training process.

To fine-tune the pre-trained BERT, we use `*TFBertForSequenceClassification*` class when loading the model to adapt the BERT model for our specific hate speech detection tasks by adding a classification head (usually a linear layer) on top of the BERT encoder. The training stops at the 5th epoch, and the highest validation accuracy is 98.71%, the test accuracy is 98.70%.

8. Model Performance Evaluation

We evaluate our model performance from confusion matrix, precision, recall (sensitivity), F1-Score, ROC-AUC score, and ROC curve.

8.1 Confusion Matrix and Classification Report

In **confusion matrix**, the number of correct and incorrect predictions is summarized with count values and broken down by each class.

- TN (**True Negative**): the upper left corner is the number of correct predictions that an instance is not in a specific class (correctly predicted "Non-hate" tweets)
- FP (**False Positive**): the upper right corner is the number of incorrect predictions that an instance is in a specific class (incorrectly predicted "Hate" tweets that are actually "Non-hate").

- FN (**False Negative**): the lower left corner is the number of incorrect predictions that an instance is not in a specific class (incorrectly predicted "Non-hate" tweets that are actually "Hate").
- TP (**True Positive**): the lower right corner is the number of correct predictions that an instance is in a specific class (correctly predicted "Hate" tweets).

After resampling we did before, we have the same number hate tweets and non-hate tweets. The probability in the confusion matrix visualization represents the percentage of samples in this section (TN, FP, FN, TP) out of all samples.

Therefore, the proportion of TN and FP add up to 50%, which is the proportion of non-hate tweets out of all samples. The proportion of FN and TP add up to 50%, which is the proportion of hate tweets out of all samples. And they add up to 100%, which represents all samples.

The **classification report** provides a detailed breakdown of the performance metrics.

- **Precision**: how often the model is correct when it predicts 'hate tweet'/'non-hate tweet'. It equals to number of all correctly predicted hate/non-hate tweets divide by number of all samples that are classified as hate/non-hate. For example, precision of 'hate' class:

$$\mathrm{P}recision\ (label_1) = \frac{TP}{TP + FP}$$

- **Recall (Sensitivity):** how often the model predicts 'hate tweet'/'non-hate tweet'. It equals to number of all correctly predicted hate/non-hate tweets divide by number of all samples that is actually hate/non-hate. For example, sensitivity of 'hate' class:

$$\mathrm{R}ecall\ (label_1) = \frac{TP}{TP + FN}$$

- **F1-score**: the harmonic mean of precision and recall. For example, F1-score of 'hate' class:

$$\mathrm{F1}_{Score}(label_1) = 2 \cdot (\frac{Precision \times Recall}{Precision + Recall})$$

- **Support**: the number of true instances of the hate/non-hate tweets in my test dataset.
- **Accuracy**: The overall accuracy of the model. It is the ratio of correctly predicted instances (both classes) to the total instances in the dataset.

From the confusion matrices in figure 5 and classification reports, all four models perform very well. The Random Forest has the highest accuracy (98.95%). Despite similar model performance of Transformer (fine-tune BERT), it needs more computing time and resources than traditional machine learning models. The Random Forest performs best on predicting non-hate tweets, while the Transformer (fine-tune BERT) performs best on predicting hate tweets.

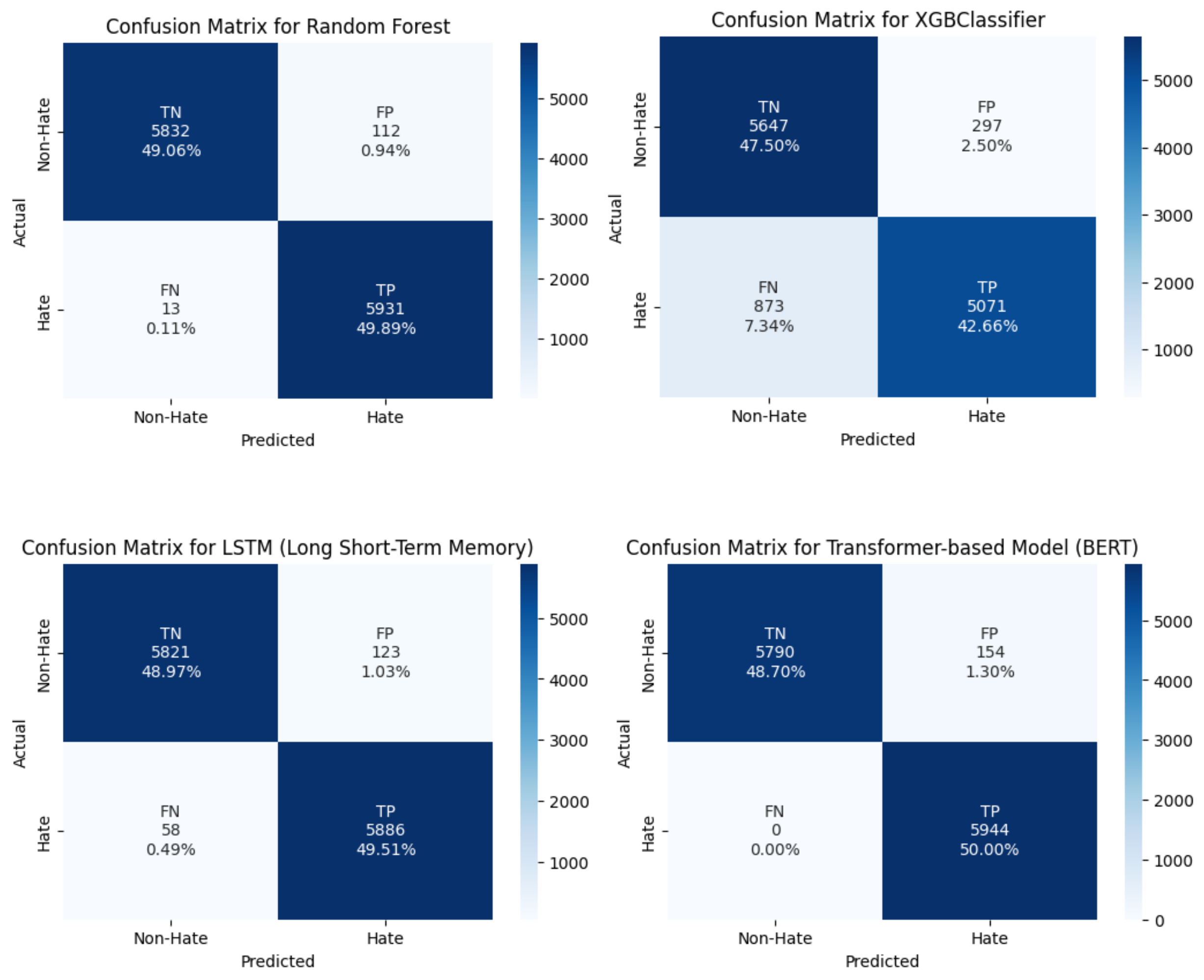


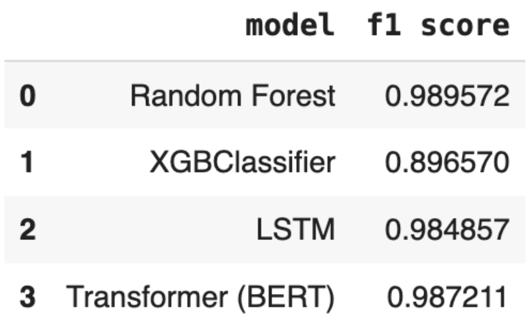
Figure 5. Confusion Matrices of 4 Models

8.2 F1-Score

F1-Score is the harmonic mean of precision and recall. It balances precision and recall.

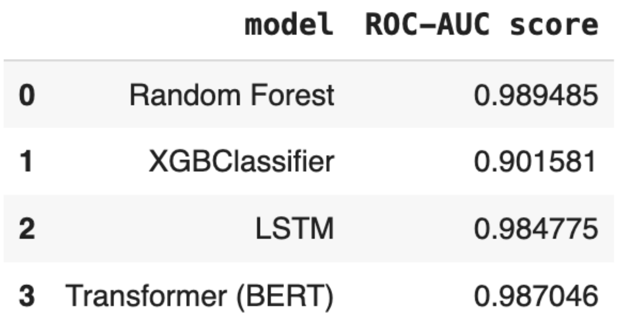
- **Precision**: the ratio of true hate tweet predictions to the total number of hate tweet predictions (true positives + false positives). High precision means that when the model predicts a positive instance, it is usually correct.
- **Recall (Sensitivity):** means the ratio of true hate tweet predictions to the total number of actual hate tweets (true positives + false negatives). High recall means that the model captures most of the actual positive instances.
- The F1 score is useful when we want to balance precision and recall, especially in cases where the classes are imbalanced. (In our case, we resample data to have balanced dataset.)

From the F1-Score table in figure 6, the Random Forest model has the best trade-off between precision and recall, which means that it performs well in identifying hate tweets and correctly predicting non-hate tweets.



	model	f1 score
0	Random Forest	0.989572
1	XGBClassifier	0.896570
2	LSTM	0.984857
3	Transformer (BERT)	0.987211

Figure 6. F1-Score of 4 Models



	model	ROC-AUC score
0	Random Forest	0.989485
1	XGBClassifier	0.901581
2	LSTM	0.984775
3	Transformer (BERT)	0.987046

Figure 7. ROC-AUC Score of 4 Models

8.3 ROC-AUC Score

The ROC-AUC score is a performance metric for evaluating the quality of a binary classification model. ROC stands for **Receiver Operating Characteristic**, and AUC stands for **Area Under the Curve**. It provides insight into the model's ability to distinguish between positive and negative classes across different threshold values. A higher ROC-AUC score indicates better overall performance and discrimination capability of the model.

From the ROC-AUC score table in figure 7, the Random Forest model has the highest ROC-AUC score, indicating that it has the best overall performance and discrimination capability of all four models we used.

8.4 ROC Curve

The ROC curve is a graphical plot that shows the diagnostic ability of a binary classifier system as its discrimination threshold is varied (each point in the curve means a different classification threshold). It plots the **true positive rate** (Recall: the proportion of actual hate tweets that are correctly identified by the model) against the **false positive rate** (1 - specificity: the proportion of actual non-hate tweet that are incorrectly identified as hate tweet by the model) at various threshold settings.

The AUC score represents the area under this curve and provides an aggregate measure of performance across all possible classification thresholds. The AUC score ranges from 0 to 1, where a score of 1 indicates perfect classification and a score of 0.5 indicates performance no better than random guessing.

From the figure 8, the Random Forest model is represented by the blue line, which is overlapped with red line and green line. It has the largest AUC, proving its ability to distinguish between hate and non-hate tweets across different threshold settings.

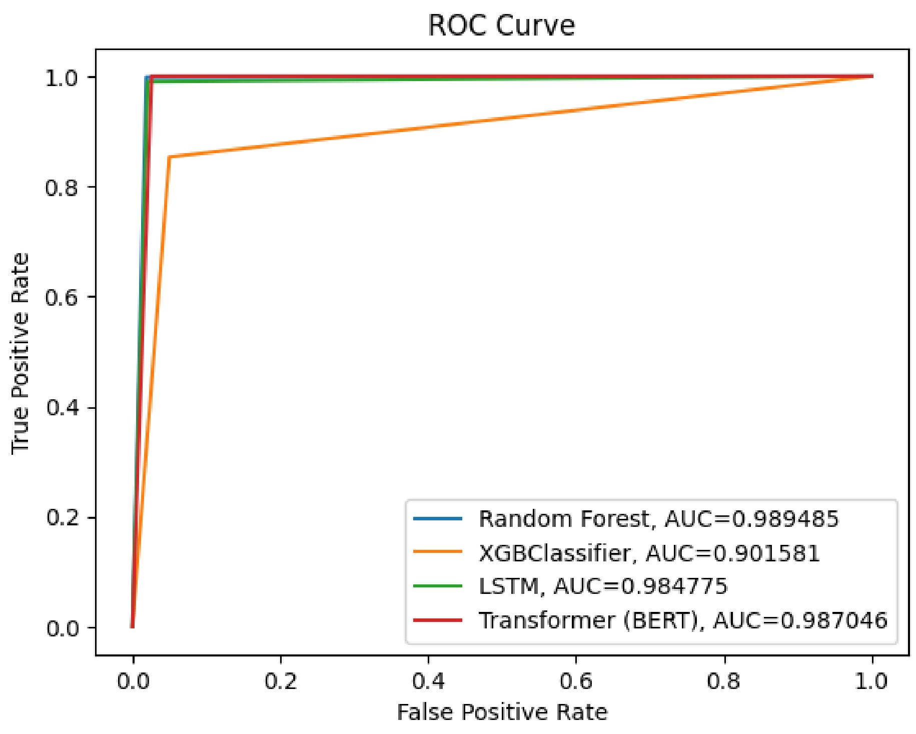


Figure 8. ROC Curve of 4 Models

9. Application Design

9.1 Deploying ML Model on Flask to Create a Web Application

In this part, we deploy our ML model on Flask to create a web application. The figure 9 displays the workflow of the web application. It consists of a single web page that let us to enter input text. Then the input is sent to the Flask server via an *HTTP POST* request. The Flask server processes it (vectorization) and feeds it into the ML model. The model generates a prediction, and the Flask server sends the prediction back to the web application, which will display the prediction result to the user.

The figure 10 shows the structure of the web application folder and files in it: *static* folder includes the *image.jpg* file that is image shown on the website and the *style.css* file which *is* style of our web design; the *templates* folder has *index.html* file that illustrates all the components on the front-end website; the *app.py* file is the Flask application code; the *requirements.txt* file lists all the package needed to run the Flask web application; *vectorizer.pkl* is TF-IDF vectorizer; *rf_model.pkl* is pre-trained Random Forest model.

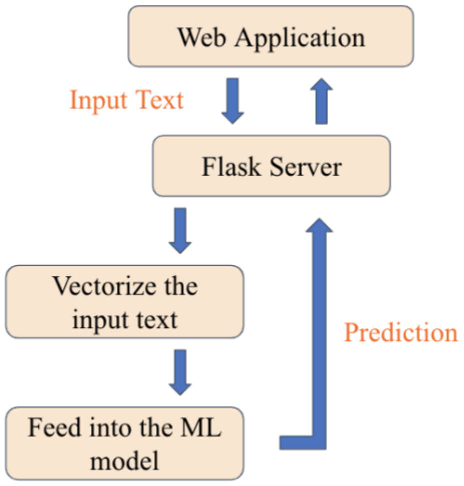
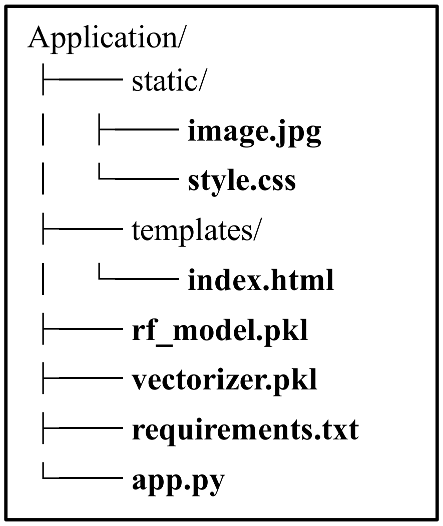


Figure 9. Web Application Workflow

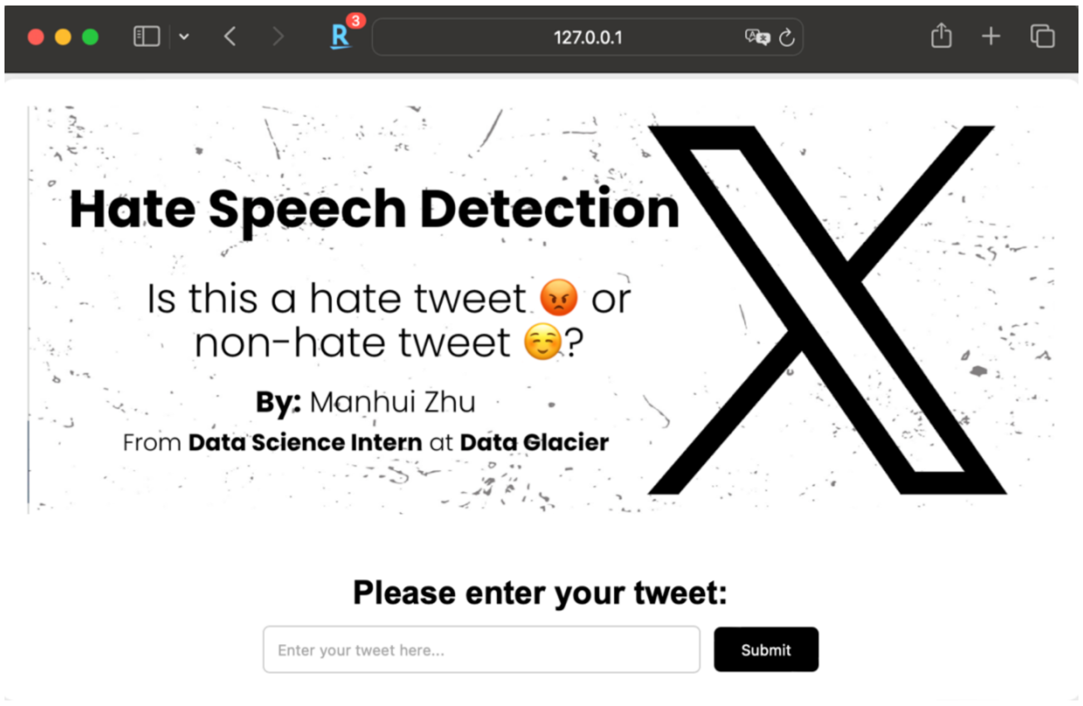


```
Application/
├──── static/
|     ├──── image.jpg
|     └──── style.css
├──── templates/
|     └──── index.html
├──── rf_model.pkl
├──── vectorizer.pkl
├──── requirements.txt
└──── app.py
```

Figure 10. Application Folder Structure and Files

9.2 Running Procedure

After having all the files needed, we can run the *app.py* file on our terminal, the output will give us the link of the website: http://127.0.0.1:5000/ . Thank we can click this link to navigate the web application. Figure 10 is an overview of the website:



Hate Speech Detection

Is this a hate tweet 😡 or non-hate tweet ☺️?

By: Manhui Zhu

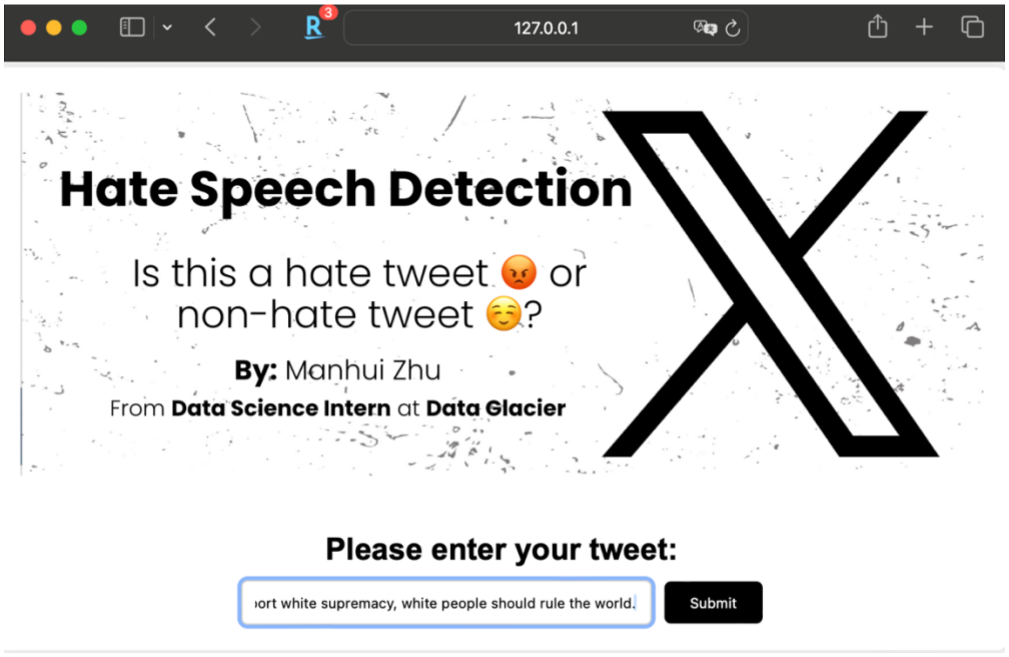
From **Data Science Intern** at **Data Glacier**

Please enter your tweet:

Enter your tweet here...

Figure 10. Website Overview

Then we can enter a speech/tweet.



Hate Speech Detection

Is this a hate tweet 😡 or non-hate tweet ☺️?

By: Manhui Zhu

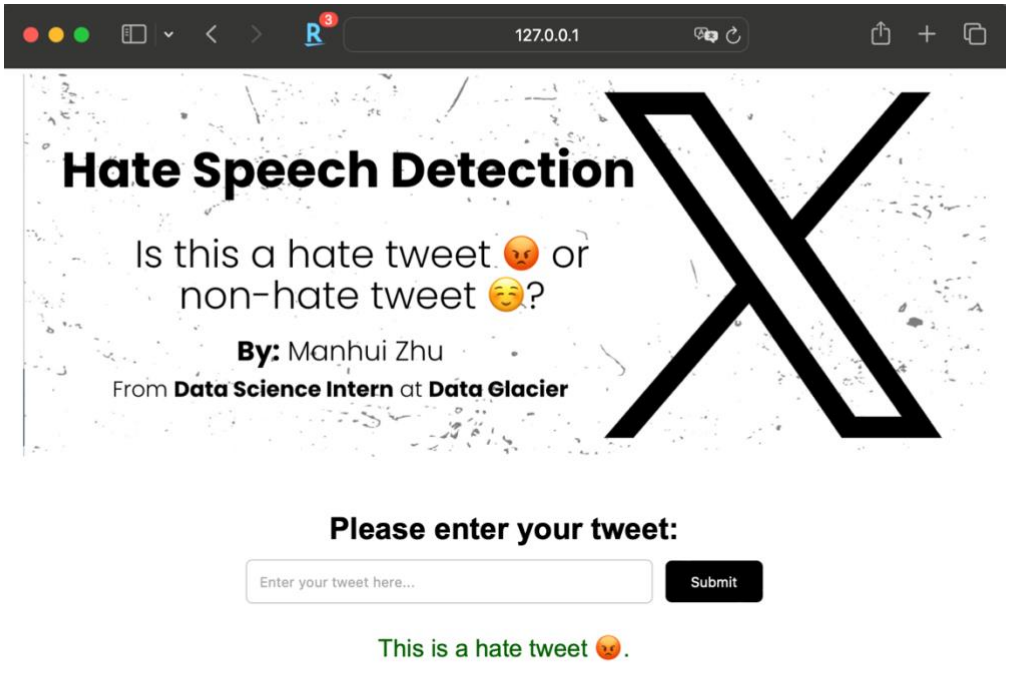
From **Data Science Intern** at **Data Glacier**

Please enter your tweet:

ıort white supremacy, white people should rule the world.

Figure 11. Enter the Speech/Tweet in the Web Application

After clicking the 'Submit' button, the speech/tweet if first be passed through the TF-IDF vectorizer, then fitted into our ML model, and the classifier will output a result and display on the website.



Hate Speech Detection

Is this a hate tweet 😡 or non-hate tweet ☺️?

By: Manhui Zhu

From **Data Science Intern** at **Data Glacier**

Please enter your tweet:

This is a hate tweet 😡.

Figure 10. Detect the Tweet in the Web Application

10. Conclusion

In this project, we try to build a model to detect the hate speech and non-hate speech and use the model to create the web application that can do the text classification on tweets. We try 4 feature extraction methods: Bag of Words (BoW), TF-IDF, Word Embeddings (Word2Vec) and BERT. We construct 4 models: Random Forest, XGBClassifier, LSTM, Transformer (Fine-tune BERT). The best performing model is Random Forest model, the model accuracy reaches 98.95%. After save the model and TF-IDF vectorizer, we deploy our model on Flask to create a Web Application. Given the error rate in the model and the limited text data, we will enhance our model performance by using more diverse data to train the model and customize the feature extraction method.

Reference

[1] Dataset: https://www.kaggle.com/datasets/vkrahul/twitter-hate-speech/code

[2] Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. Neural Computation, 9(8), 1735-1780. DOI: 10.1162/neco.1997.9.8.1735

[3] Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2019). BERT: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805.