University of New South Wales

TOXIC COMMENT CLASSIFICATION CHALLENGE

Working Group: Rui Liang z5174445, Zhicheng Tao z5148738,

Chuguan Tian z5145006, Lingxiang Zhou z5143224

A report submitted for COMP9417/19T2 Assignment

August 11, 2019

Abstract

In this project, we have applied and experimented with multiple machine learning models to a real world machine learning challenge from Kaggle competition: **Toxic Comment Classification Challenge**, which requires us to identify toxicity level from a piece of comment text. We mainly tried to use traditional machine learning model to solve this problem, including Naive Bayes, Logistic Regression, Decision Tree, Random Forest, Boosting, Ensemble Learning and Neural Networks. In the meaning time, we compared the performance of different machine learning algorithms and drew a conclusion of pros and cons of applying these algorithms to the text classification problem.

Contents

Chapter 1

Introduction

Over the past twenty years, our daily life has changed dramatically due the burst of the internet. With the rapid growth of internet speed, the volume of contents on the internet also grows exponentially. Nowadays, we constantly expose to the internet and absorbing information from vary kinds of materials. People can express their opinions more freely through these medium without any responsibility. For these reasons, toxic comments can spread out rapidly and influence a large amount of people including teenagers. Therefore, the censorship of comments on the internet becomes more and more important. Many community begins to take responsibility for identifying toxic comments and protect people from reading those offensive contents.

1.1 Aims and Objectives

In the toxic comment classification challenge project, our task is to develop and train several machine learning models which can automatically identify six types of toxic comments (toxic, severe toxic, obscene, threat, insult, and identity hate) given the comment text. For example, we are given a comment text string "This message is not toxic!", then our goal is to label each class type with "0" or "1":

comment_text	toxic	severe_toxic	obscene	threat	insult	identity_hate
"This message is not toxic!"	0/1?	0/1?	0/1?	0/1?	0/1?	0/1?

Table 1.1: *Toxic comment classification example*

In the past, we usually use keyword filtering to achieve this goal, however, it can be easily overkill which may block some normal conversation and reduces customer satisfaction. For example, if a non-toxic sentence that needs to be classified is: The rape of the forest forces many animals migrate to the near cities. This sentence contains a word "rape", if we our classifier was built on filtering, then it will be classified as toxic. Therefore, in this project, we mainly focus on developing a more robust machine learning model using **supervised learning** method to identify true toxic comments and which also achieves a great performance.

1.2 Overview of the Report

The report consists of three main parts, the first part briefly introduce the necessary background of the project. Then we start to dive into the training dataset and perform data analysis. After that, we apply several machine models to the dataset and evaluate their performance separately. In the end, we compare all the models together and draw a conclusion.

Chapter 2

Data Exploration and Data Cleaning

2.1 Overview of the dataset

The Kaggle dataset (dataset link is provided in Conclusion, Acknowledgement part) consists of training dataset and test dataset, in this project, we use training dataset to build our machine learning model and then use test dataset to evaluate model performance. Then we submit our prediction to Kaggle in order to see the final score. The training dataset contains the following columns:

	id	comment_text	toxic	severe_toxic	obscene	threat	insult	identity_hate
0	0000997932d777bf	Explanation\nWhy the edits made under my usern...	0	0	0	0	0	0
1	000103f0d9cfb60f	D'aww! He matches this background colour I'm s...	0	0	0	0	0	0
2	000113f07ec002fd	Hey man, I'm really not trying to edit war. It...	0	0	0	0	0	0
3	0001b41b1c6bb37e	"\nMore\nI can't make any real suggestions on ...	0	0	0	0	0	0
4	0001d958c54c6e35	You, sir, are my hero. Any chance you remember...	0	0	0	0	0	0
5	00025465d4725e87	"\n\nCongratulations from me as well, use the ...	0	0	0	0	0	0
6	0002bcb3da6cb337	COCKSUCKER BEFORE YOU PISS AROUND ON MY WORK	1	1	1	0	1	0
7	00031b1e95af7921	Your vandalism to the Matt Shirvington article...	0	0	0	0	0	0
8	00037261f536c51d	Sorry if the word 'nonsense' was offensive to ...	0	0	0	0	0	0
9	00040093b2687caa	alignment on this subject and which are contra...	0	0	0	0	0	0

Figure 2.1: *Training dataset*

Column 1 is the "id" of a training record, column 2 is the comment text that needs to be labeled (it has already been labeled in training data), the rest of the columns are the corresponding toxic level categories. The positive class is "1" and the negative class is "0". In addition, the training data contains 159571 rows in total, so it can be viewed as a 159571×8 matrix. After plotting the basic statistics and counting the null value in the table, we find that there is no null value, so that we are ready to start data analysis.

The test dataset in Kaggle is split into two parts, one part contains test comment id and comment text, which requires us to classify. Another part contains the ground truth labels for each category. Note that many of the label are marked as "-1", which means this comment is not used on scoring in the Kaggle competition.

2.2 Data analysis

We first compare the number of comments for each category with different (positive/negative) values:

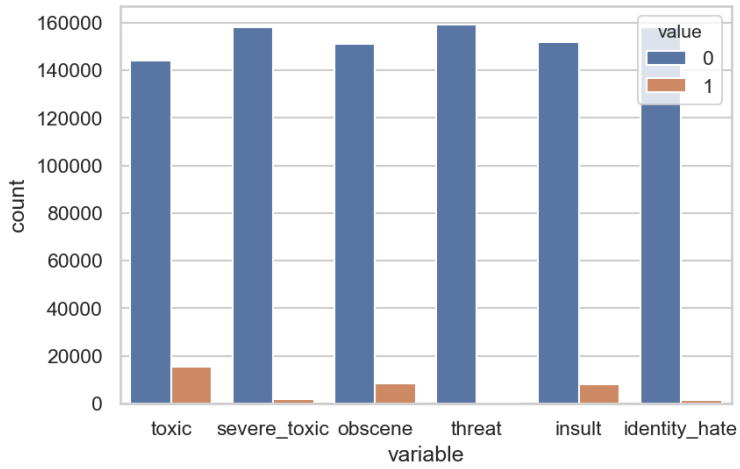

Figure 2.2: *comment count with different labels and values*

We can see from the figure that most of the comments belongs to negative class and only a few samples are labeled as positive. So that this is a highly **unbalanced** training dataset which requires us to develop a more robust machine learning model and evaluation model to tackle this problem.

If we focus on the number of comments labeled as positive in each category, we can get the following graph:

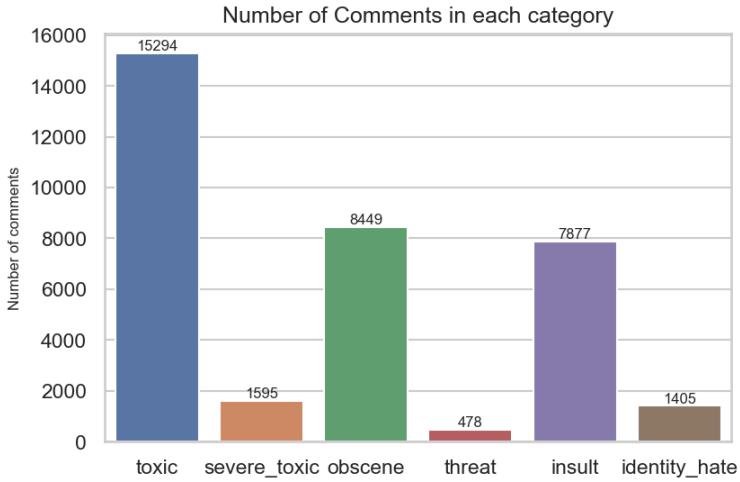

Figure 2.3: *number of comments in each category*

The category of "severe_toxic", "threat" and "identify_hate" only contains a relative small amount of samples, which may make our machine learning model hard to generalise for these

categories.

Then we found out that most of comments only contains less than 150 words. This may help us to fine tune model parameter for some machine learning models.

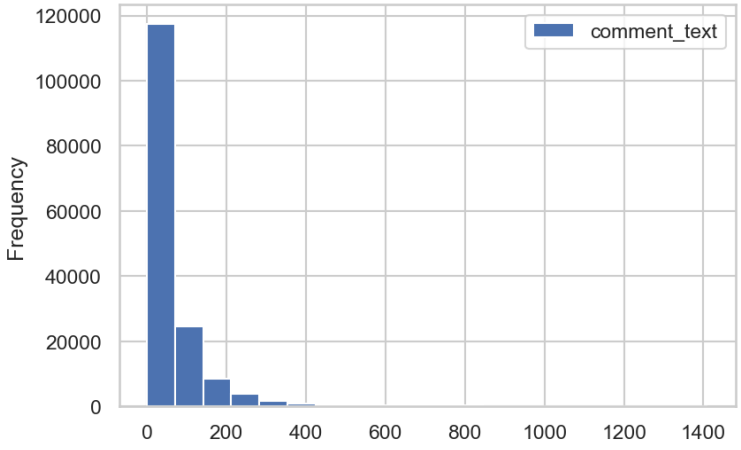

Figure 2.4: *comment length distribution*

After computing the pairwise correlation between each category, we found that there are relatively strong correlation between "obscene" and "toxic", "insult" and "toxic", "insult" and "obscene". This may help us to build more sophisticated models which depends on previous predicted labels in order to achieve more higher performance.

Figure 2.5: *correlation heatmap between each category*

2.3 Data cleaning

In the data cleaning process, we first try to manually inspect what are the possible terms that would appear in the comments. We find out that a comment may include the following terms:

- Ordinary English words
- Unicode characters
- Http(s) links
- Emoji ideograms
- Files and pictures
- CSS tags
- Meaningless numbers
- Meaningless number and character combinations
- Words separated by numbers or non-word characters (e.g. d*rty)

After that, we need to decide which term should we remove and it still maintains the original semantic meaning. Since there are only a small amount of Unicode characters (Chinese, Japanese and Korean characters, etc.) and all of them are labelled as non-toxic. Besides, we mainly focus on identify toxic comments in English, so that we will try to remove Unicode characters in the cleaning process. Http(s) links, emoji ideogram and file names can contain words among them, so that we can try to remove non-word characters and keep the original words. CSS tags and meaningless numbers or character combinations can be seen as noise for this case, therefore we will use regex to remove all the CSS tags as well as numbers. For the last case, we decide to use a word correction package to correct word spelling and retrieve the original word. The data cleaning procedure can be divided into the following steps:

1. Convert all the character into lower case.
2. Remove CSS tages.
3. Remove non-word and Unicode characters.
4. Remove multiple spaces or line breaks.
5. Correct word spelling using “symspellpy” package.

The cleaning process needs a significant amount of time due to word correction. Usually, one comment needs one to two seconds to be processed, so that we need to split the dataset into multiple chunks and then we can process each part of the dataset in parallel.

The cleaned data achieves a better performance on the Naive Bayes baseline model (see performance comparison graph in the Naive Bayes model section), so that we keep using the cleaned data for other models.

Chapter 3

Methodology

In this chapter, we are going to introduce the general training and testing methodology for applying a machine learning model in our project.

3.1 Training and Testing Split

Usually, in a real world problem, we only have a training dataset but without a test set. In our project, Kaggle provides us both the training and test dataset, however, we still need to train our model only on training dataset and after fine tune every parameter, then we can finally test the model performance on the Kaggle test set.

In order to train and tune the parameter for our models, we can split the training dataset into two parts, one for training and another for testing/validation. In this project, we the ratio of training and testing/validation split is 8:2, which means 80% of the training data is used for training and the rest 20% is used for testing/validation.

Figure 3.1: *number of comments labeled as positive in train and test/validation split*

In Naive Bayes baseline model, we split the training dataset into training and testing set and use K-fold cross validation to compare Naive Bayes model with different hyper-parameters. After selecting the final parameter, we compare the model performance on split test set and Kaggle test set.

For some other machine learning models, we use the test split as validation set rather than test set to fine tune the hyper-parameters to avoid overfitting. Then we compare the performance on split validation set and Kaggle test set. Note that in this process, a better way to fine tune the hyper-parameters is to use "GridSearchCV" in "sklearn", which can perform exhaustive search on different parameters together with using cross validation. However, this kind of search would need a significant amount of time to train a model especially when the dataset and the number of features are large. For the sake of time limits, we try to compare different machine learning models with "good" parameter rather than "best" parameter.

In neural network, we split the training dataset into three parts with 70% training data, 20% validation data and 10% test data. We train the model on training data and turn the number of neurones as well as layers according to validation score. Then test the model on split test set and finally compare the performance with Kaggle test dataset.

3.2 Model Evaluation Method

Since our training dataset is highly unbalanced and the majority of the comments are non-toxic, so that the most common "accuracy" score is not suitable to evaluate the performance of a machine learning model. For example, in the case of "threat" comments, there are only 478 comments are considered as positive, the other $159571 - 478 = 159093$ comments are labeled negative. If our model predict all the "threat" comments as negative, we can still achieve $\frac{159093}{159571} = 0.997$ accuracy! In order to solve this problem, we can use ROC-AUC (Area Under the Receiver Operating Characteristic Curve) score to evaluate the model performance. The following graph shows an example of ROC curve and the AUC score calculated:

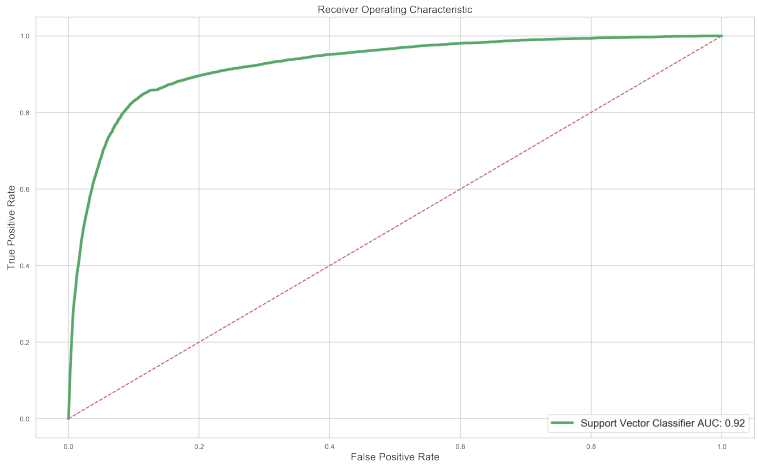

Figure 3.2: *ROC curve with calculated AUC*

ROC curve is a graph plot of TPR (true positive rate) as y-axis and FPR (false positive rate) as x-axis of a model with different discrimination probability thresholds (from 0.0-1.0) to predict a sample to be positive or negative. TPR is calculated as $TPR = \frac{TP}{TP+FN}$ and $FPR = \frac{FP}{FP+TN}$. AUC is the area under the ROC curve (integration from x-axis 0-1) which measures the performance over all different thresholds. Another reason why we choose to use ROC-AUC score is that we want to evaluate the model in general, which means we do

not want to optimise our model to predict positive with higher probability (this will lower customers satisfaction) or predict negative with higher probability (this may miss many toxic comments).

3.3 Text Vectorisation

In order to apply machine learning model to text data, we can use "Bag-of-words" model to represent the data. Firstly, we need to extract words from all the raw comment text and build a dictionary, then transform each raw "comment_text" records into a very large feature vector. Each index of the vector represents a word and each value of the entry represents the word count in a specific comment record. Then each comment is represented as a very large vector, however, most of the entry is 0 because a comment only contains a small number of words in the dictionary. So that we can represent such data structure in a "dense" way using sparse matrices in python in order to reduce memory usage. Python provides us with a handy tool called "CountVectorizer" which can vectorise each comment easily.

Another way to vectorise the text is to use TFIDF (term frequency-inverse document frequency). We observe that the length of each "comment_text" varies a lot, when we apply the "word count vector" to the dataset, longer text will tend to have more counts, which will result in less informative for capturing the true "meaning" of the document. We can tackle this problem using TFIDF, intuitively, we can interpret TFIDF as the importance of a word in the documents, the higher the value of TFIDF, the more important of the term is.

In Naive Bayes model, according to the "sklearn" documentation, it is better to use "CountVectorizer" rather than "TfidfVectorizer" due to the fact that multinomial distribution requires integer feature counts. We have experimented with these two methods in Naive Bayes model as well as other models. We found out that "CountVectorizer" performs better in Naive Bayes but performs the same as "TfidfVectorizer" in other models, so that we keep using "TfidfVectorizer" in the rest of the models other than Naive Bayes.

3.4 Multi-label Classification

Since our problem is multi-label classification problem, which means given a comment, it can be labeled as any combination of the six categories. So that we need a strategy to train our model using a reasonable method to output multiple labels. The most common method is to use "One-vs-the-rest" classifier. This strategy trains individual model for each label and use all the models to predict the output. For example, if we are training the Naive Bayes model, we have to build six such model independently for each category (each model training process can be viewed as binary classification problem, e.g. "toxic" category can only be classified as "0" or "1"). Then we use all six models to predict the given test comments.

Chapter 4

Apply Machine Learning Models

In this chapter, we are going to introduce Naive Bayes classifier, Decision Tree classifier, Random Forest, Adaptive Boosting, Logistic Regression Classifier, Support Vector Machine and Neural Networks, then apply them to the dataset. Finally, we analysis the performance of each algorithm regarding the toxic comment classification problem.

4.1 Naive Bayes Classifier as Baseline Model

Naive Bayes classifier is often used as a baseline model for machine learning problems, since it is easy to understand and the training process is very fast, it also generates relatively good performance for many problems. The most important part of the Naive Bayes classifier is that it is based on the Naive Bayes assumption, which assumes every attribute (feature) of a sample is conditionally independent. So that we can calculate the probability of a sample X which has n features belongs to a specific class C_i as

$$\begin{aligned} P(C_i|X) &= \frac{P(X|C_i) \cdot P(C_i)}{P(X)} \\ &\propto P(X|C_i) \cdot P(C_i) = P(C_i) \prod_{k=1}^{n} P(x_k|C_i) \end{aligned} \tag{4.1}$$

Then we can use MAP (maximum a posteriori estimation) to predict the most likely class:

$$\begin{aligned} \hat{c}_{map} &= \arg\max_{C_i \in C} \hat{P}(C_i|X) \\ &= \arg\max_{c_i \in C} [\log \hat{P}(C) + \sum_{1 \le k \le n} \log \hat{P}(x_k|C_i)] \end{aligned} \tag{4.2}$$

For text classification problem, there are some different methods to apply Naive Bayes, such as Multinomial Naive Bayes and Multivariate Bernoulli Naive Bayes model. These two algorithms use the same core idea of Naive Bayes but encode the text differently. Multinomial Naive Bayes keeps track of multiple occurrence of a term, however, Multivariate Bernoulli only uses binary occurrence information. Since our project needs to identify the level of toxic,

and multiple occurrence of a term is related to the severity of a toxic comment, so that we choose to use Multinomial Naive Bayes classifier rather than Multivariate Bernoulli model.

The following box plot shows the result of applying Multinomial Naive Bayes classifier on the 80% training data using 10-fold cross validation with different hyper-parameters evaluated using scoring method "roc_auc":

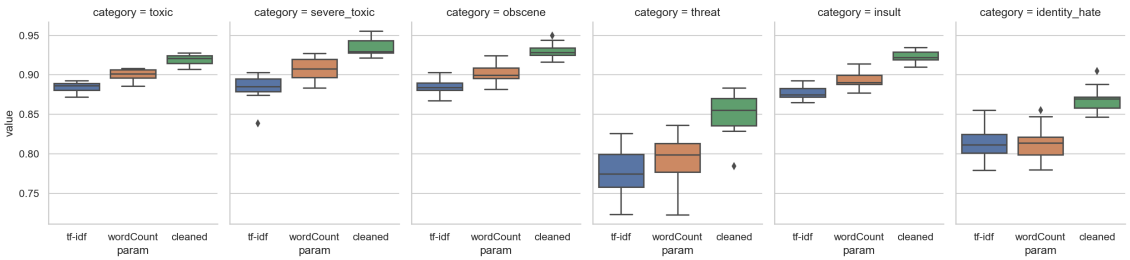

Figure 4.1: *Multinomial Naive Bayes with different hyper-parameters*

We can see that using "CountVectorizer" to vectorise text performs better than "TfidfVectorizer" for most of the categories. In this comparison, we also try to apply the model on cleaned data using "CountVectorizer" method, which achieve the best performance for all the categories. In the following models, we will keep using the cleaned data rather than the raw dataset.

Then we compared the model performance by applying it on the 20% test split and the Kaggle test set (result ROC curve is in appendixB):

Data set	toxic	severe_toxic	obscene	threat	insult	identity_hate
Split validation AUC score	0.9203	0.9306	0.9302	0.8611	0.9226	0.8606
Kaggle test AUC score	0.9248	0.9251	0.9305	0.8745	0.9201	0.9022

Table 4.1: *Apply Multinomial Naive Bayes on different test datasets*

We can see that Multinomial Naive Bayes performs well and stable on most of the categories. However, since the lack of samples labeled as positive in category "threat", it has a higher variance than other categories.

4.2 Decision Tree Classifier

The core idea behind Decision Tree classifier is to construct multiple "if...else" structure based on some splitting criteria and select the best split. There are two popular splitting criteria, namely Gini Index and Information Gain.

Information Gain can be intuitively interpreted as the expected reduction in entropy (measures how "chaos" the data is) due to splitting on some attribute A. In Decision tree learning, we want to maximise Information Gain after splitting. This can be calculated as:

$$Gain(S, A) = Entropy(S) - \sum_{v \in Values(A)} \frac{|S_v|}{|S|} Entropy(S_v) \tag{4.3}$$

where

$$Entropy(S) = -\sum_{i=1}^{m} \frac{|S_i|}{|S|} \log_2 \frac{|S_i|}{|S|} \tag{4.4}$$

S is training samples

S_v is training samples agree on value v

m is the number of classes (e.g. for binary classification $m = 2$)

S_i is training samples belongs to class i

Gini Index is similar to Information Gain, which is another way to measure the impurity of a dataset, the higher the Gini Index, the more impurity the dataset is. In decision tree training, we want to minimise Gini Index after splitting on some attribute A. Gini Index of training samples S can be calculated by:

$$Gini(S) = 1 - \sum_{i=1}^{m} p_i^2 \tag{4.5}$$

and

$$Gini(S, A) = \sum_{i}^{m} \frac{|S_i|}{|S|} Gini(S_i) \tag{4.6}$$

There are many variations regarding Decision Tree algorithms, the most popular ones are ID3 (Iterative Dichotomiser 3) and CART (Classification and Regression Trees). The significant difference between these two algorithms is that CART generates a binary tree whereas ID3 constructs a multiway tree. In our project, we use "DecisionTreeClassifier" from "sklearn", which is implemented using an optimised CART algorithm. Since Decision Tree classifier can easily overfit the data, so that we need to fine tune the maximum depth parameter of the tree (pruning is not currently supported in sklearn). The graph in appendixA (Figure A.1) shows training score and validation score for six categories with different maximum depth of the tree when applying Decision Tree classifier on the 80% training data and 20% validation data using "entropy" splitting criterion.

Then we select "max_depth" with the following optimised values for the Decision Tree classifier using "entropy" splitting criterion.

Parameter	toxic	severe_toxic	obscene	threat	insult	identity_hate
max_depth	63	6	20	7	15	6

Table 4.2: *Optimum values of max_depth parameter for "entropy" method*

We also use the same method for another Decision Tree classifier using "gini" criterion and select the following "max_depth" values:

Parameter	toxic	severe_toxic	obscene	threat	insult	identity_hate
max_depth	55	6	25	6	15	11

Table 4.3: *Optimum values of max_depth parameter for "gini" method*

Then we use these optimised values to compare Decision Tree classifier performance with different splitting criteria using 10-fold cross validation on the 80% training data and get the following results:

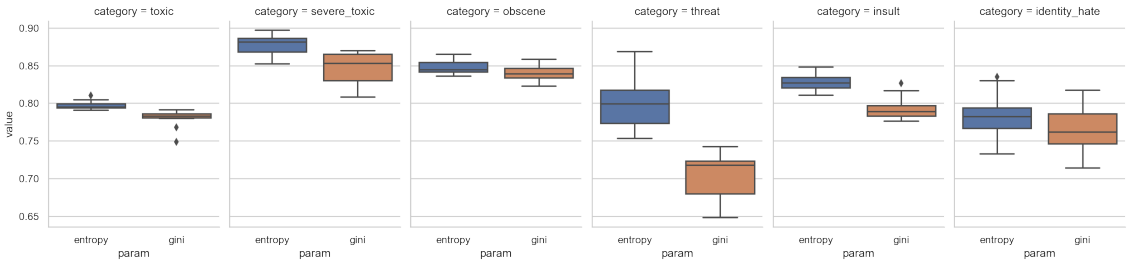

Figure 4.2: *Decision Tree Classifier with different splitting criteria*

We can see that in our project, Decision Tree classifier using "entropy" splitting method is better than "gini" and more stable. Finally, we test the model on validation set and Kaggle test set using "entropy" splitting method, which gives the following results (result ROC curve is in appendixB):

Data set	toxic	severe_toxic	obscene	threat	insult	identity_hate
Split validation AUC score	0.7898	0.8555	0.8427	0.8305	0.8138	0.7612
Kaggle test AUC score	0.8170	0.8738	0.8383	0.8454	0.8216	0.8016

Table 4.4: *Apply Decision Tree model on different test datasets*

We can see that Decision Tree classifier performs not that well on both the test set in this classification problem. One possible reason is that the feature space for text classification is very large, Decision Tree classifier can easily overfit the training data and performs poorly on the test data. In order to tackle this problem, next we are going to introduce a more robust tree learning model which utilise ensemble learning method for classification.

4.3 Random Forest

The core idea behind Random Forest is that it constructs multiple different "subtrees" from adapted versions of the training data, then combine these "subtrees" together to perform prediction (e.g. majority vote). For each iteration in the Random Forest algorithm, we sample S_t training examples with replacement using Bootstrap, then we randomly select n features from all the features, finally we train the subtree on these S_t examples using n features. In the process of training Random Forest model, we need to find out the number of subtrees to train, so that we try to build Random Forest model on different subtree numbers with 80% data training and 20% data validating. Parameter tuning result is in appendixA

(Figure A.2). We can see that the changing of subtree number does not help much on the validation score, so that we can choose the following maximum subtree number parameter for training the Random Forest model:

Parameter	toxic	severe_toxic	obscene	threat	insult	identity_hate
max_estimators	35	10	15	10	35	15

Table 4.5: *Optimum values of max_estimators parameter for Random Forest*

Then we use 10-fold cross validation to compare Random Forest with Decision Tree classifier and get the following result:

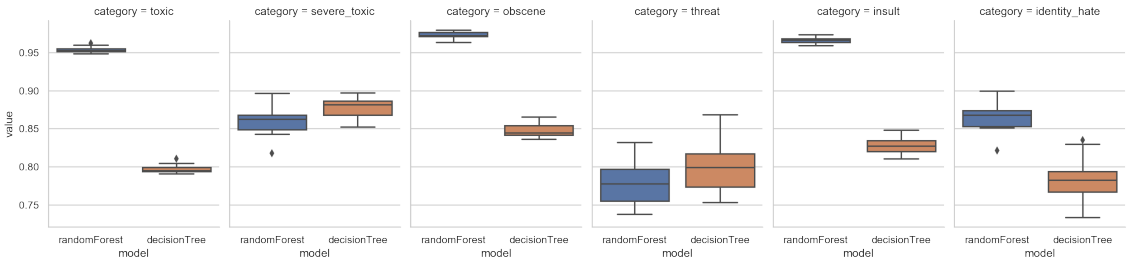

Figure 4.3: *Compare Random Forest and Decision Tree using cross validation*

We can see that Random Forest can achieve a much better performance than Decision Tree classifier on some categories. However, for those lack of positive examples categories like "severe_toxic" and "threat", it performs even worse than Decision Tree classifier. One reason for this is that in the ensemble learning process, each subtree only learns on a small subset of the training samples, which further reduces the minority class samples and causes each subtree to be a "bad" learner. By combining these "bad" learners together, we still get a "bad" model. Hence, in order to achieve better performance for Random Forest classifier, we need sufficient amount of both negative and positive samples.

Finally we test the model on validation set and Kaggle test set for evaluation (result ROC curve is in appendixB):

Data set	toxic	severe_toxic	obscene	threat	insult	identity_hate
Split validation AUC score	0.9582	0.8758	0.9702	0.7909	0.9640	0.8448
Kaggle test AUC score	0.9553	0.8800	0.9584	0.8045	0.9570	0.9124

Table 4.6: *Apply Random Forest model on different test datasets*

4.4 Adaptive Boosting

AdaBoost (Adaptive Boosting) is another Ensemble learning method. In Random Forest classifier, each subtree is independent to each other, however, in AdaBoost, we try to build a strong learner from a number of dependent weak classifiers. In the iteration process, each weak learner is influenced by the performance of previous ones and learn the "errors" caused by them. This can be done by increasing the weight of those misclassified samples, and

decrease the weight of those correctly classified samples. Then in the final boosted model, we try to combine the output of all the weak learners together with the corresponding weights and perform the prediction. Another major difference between AdaBoost and Random Forest is that each subtree in Random Forest only leans a subset of the training samples, whereas each weak learners in AdaBoost learn the whole training dataset. This can remedy the lack of minority class sample problem (e.g. there are only 478 positive samples in category "threat" but 159093 negative samples). In order to train train the AdaBoost model, we need to specify the maximum number of estimators, the graph in appendixA (Figure A.3) shows how different estimators influence the training and validation scores. Then, we select the following optimum parameter:

Parameter	toxic	severe_toxic	obscene	threat	insult	identity_hate
max_estimators	300	150	225	50	250	450

Table 4.7: *Optimum values of max_estimators parameter for AdaBoost*

We use the above parameters to further compare the performance with Random Forest and Decision Tree using k-fold cross validation on the 80% training data:

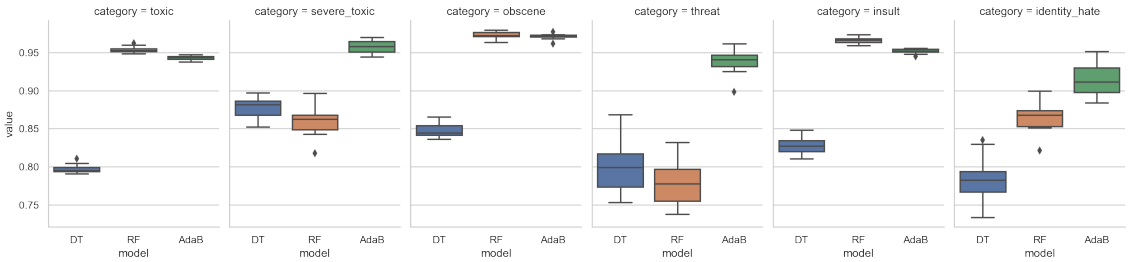

Figure 4.4: *Compare three models using cross validation*

We can see that indeed AdaBoost can remedy the lack of minority class sample problem and boost the performance for category "severe_toxic", "threat" and "identify_hate". Next, we apply the AdaBoost model on validation data and Kaggle test set and compare its performance (result ROC curve is in appendixB):

Data set	toxic	severe_toxic	obscene	threat	insult	identity_hate
Split validation AUC score	0.9447	0.9574	0.9736	0.9428	0.9539	0.9206
Kaggle test AUC score	0.9403	0.9496	0.9610	0.9595	0.9470	0.9590

Table 4.8: *Apply AdaBoost on different test datasets*

4.5 Logistic Regression

Logistic Regression is a Discriminative model which tries to model probability $P(Y|X)$ directly as a function: $P(Y|X) = f(X; w^T)$ from the training data. This is different from Generative model such as Naive Bayes classifier which learns the generative probability $P(X|Y)$ and use Bayes rule to calculate $P(Y|X) = \frac{P(X|Y)P(Y)}{P(X)}$. Intuitively, it is very hard to learn

$P(X|Y)$ accurately in the case of high dimensional feature vector, because we need a large number of samples to learn the generative probability. In Logistic Regression, we usually use Sigmoid function as the activation function to predict sample X:

$$P(Y|X) = Sigmoid(w^T x) = \frac{1}{1 + e^{-w^T x}} \tag{4.7}$$

In order to learn parameter w^T from the training data, we can estimate the likelihood (assuming every sample is independent) of seeing the whole training set as:

$$\begin{aligned} L(w) &= \prod_{i=1}^{n} p(x^{(i)})^{y^{(i)}} (1 - p(x^{(i)}))^{1-y^{(i)}} \\ &= \sum_{i=1}^{n} y^{(i)} \log p(x^{(i)}) + (1 - y^{(i)}) \log(1 - p(x^{(i)})) \end{aligned} \tag{4.8}$$

where $p = Sigmoid(w^T x)$ and $y^{(i)}$ indicate the label of sample $x^{(i)}$.

Our goal is to maximise this likelihood function, however, there is no closed form solution for this problem. So that we need to use some optimisation methods such as Gradian Ascent, Simulated Annealing, Newton's Method, etc. In order to avoid overfitting in logistic regression, we usually add a regularisation term:

$$L(w) = [\sum_{i=1}^{n} y^{(i)} \log p(x^{(i)}) + (1 - y^{(i)}) \log(1 - p(x^{(i)}))] + \lambda R(w) \tag{4.9}$$

where λ is the regularisation strength and $R(w)$ can be $l2$, $l1$, etc.

In order to apply Logistic Regression model, we have to fine tune the regularisation value, the graph in appendixA (Figure A.4) show training and validation score on different regularisation values on the 80% training data and the 20% validation data. Then we select the following regularisation value

Parameter	toxic	severe_toxic	obscene	threat	insult	identity_hate
Regularisation Value	4	0.4	4	10	4	4

Table 4.9: *Optimum values of regularisation value for Logistic Regression*

Then we apply Logistic Regression on the 80% training data using 10-fold cross validation to evaluate the model:

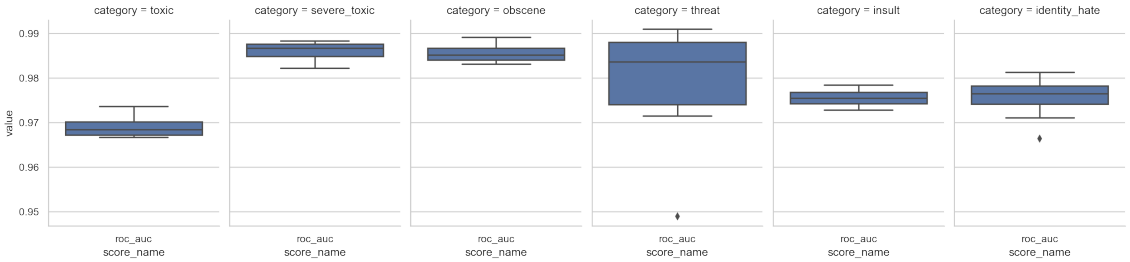

Figure 4.5: *Evaluate Logistic Regression using cross validation*

We can see that Logistic Regression overall performs very stable on most of the categories and achieves a very high score.

Finally we compare the performance of Logistic Regression on the 20% validation set and Kaggle test set (result ROC curve is in appendixB):

Data set	toxic	severe_toxic	obscene	threat	insult	identity_hate
Split validation AUC score	0.9705	0.9859	0.9872	0.9859	0.9762	0.9769
Kaggle test AUC score	0.9597	0.9852	0.9729	0.9840	0.9647	0.9818

Table 4.10: *Apply Logistic Regression on different test datasets*

4.6 Support Vector Machine

The core idea behind Support Vector Machine is that it utilises "kernel trick" to map non-linear separable data into higher dimensions implicitly and find a maximum margin hyperplane to seperate the data. The large margin can make SVM more tolerate to noise and increase the probability of correctly classifying samples which are close to the hyperplane. Since SVM can cope with very high dimension data even with a small number of samples, it is suitable for text classification problems. In our project, we use linear kernel to train the SVM model and fine tune the regularisation parameter C. The graph in appendixA (Figure A.5) shows the training and validation score on different regularisation values. Then we decide to select the regularisation value to 0.1 to train the SVM model. After that, we compare it with other machine learning models together using 10-fold cross validation on the 80% training data and get the following results:

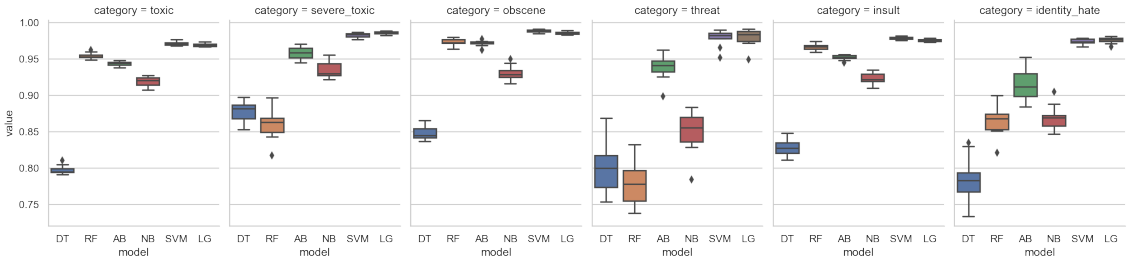

Figure 4.6: *Training and validation score on different regularisation values*

We can see from the graph that Support Vector Machine outperform most of all the other models and only a little worse than Logistic Regression in some categories. In general, SVM is a strong leaner in most of the cases.

Finally, we apply SVM classifier on validation data and Kaggle test set in order to compare test performance (result ROC curve is in appendixB):

Data set	toxic	severe_toxic	obscene	threat	insult	identity_hate
Split validation AUC score	0.9730	0.9806	0.9895	0.9815	0.9792	0.9759
Kaggle test AUC score	0.9627	0.9819	0.9750	0.9816	0.9691	0.9834

Table 4.11: *Apply SVM on different test datasets*

4.7 Neural Network

The key idea behind Neural Network is to construct multiple layers of "artificial neurones" (functions with input variables and weights), then connect neurones output in the previous layer as input for the following adjacent layer, until it reaches the last layer which computes the final output. When we feed the Neural Network with training data, it will automatically adjust weights for each neurone and find the "best" weight (it may end up with local optimum) for such Neural Network model. One of the most popular way so solve this optimisation problem is Gradient Descent, which computes gradients through Backpropagation. In our project, we use "Keras" in google "colab research" to apply Neural Network model. In regards of training the model, we need to experiment with different number of neurones in each network layer and the number of layers combinations. In this process, we split the training data into 70% for training, 20% for validation (fine tune hyper-parameters) and 10% for testing. The experiment result are attached in the appendixA of the report. Finally we conclude that in our project, multiple hidden layers tend to make the neural network more easily to overfit the training data. So that we test the Neural Network model using 96 neurones in the first layer (each has input size of 54351 features) and 6 neurones for the output layer (output probability for 6 categories). The total number of parameters we need to learn in the first layer is $54351 \times 96 + 96 = 5217792$ and the number of parameters for the output layer is $96 \times 6 + 6 = 582$. The training process we use is Batch Gradient Descent with batch size 512 and use validation loss as the early stopping callback. Then we apply our model on the split test set and Kaggle test set and get the following results (the ROC curve is attached in the appendixB):

Data set	toxic	severe_toxic	obscene	threat	insult	identity_hate
Split test AUC score	0.9547	0.9792	0.9752	0.9740	0.9680	0.9645
Kaggle test AUC score	0.9488	0.9801	0.9640	0.9707	0.9563	0.9740

Table 4.12: *Apply Neural Network on different test datasets*

Chapter 5

Results

5.1 Kaggle Submission Results

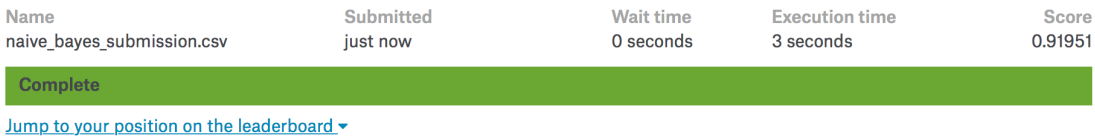

Figure 5.1: *Naive Bayes Kaggle Submission Score*

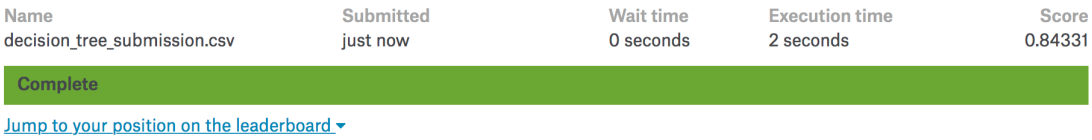

Figure 5.2: *Decision Tree Kaggle Submission Score*

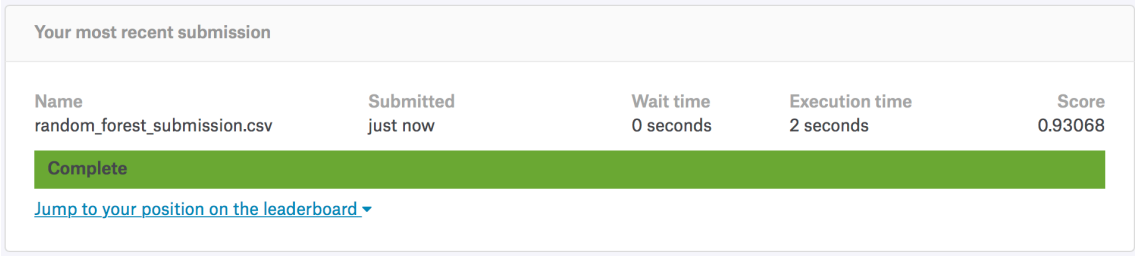

Figure 5.3: *Random Forest Kaggle Submission Score*

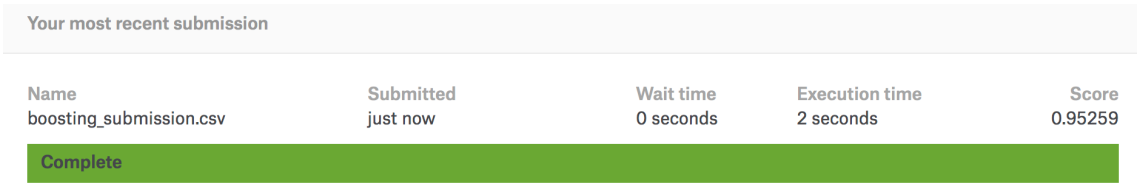

Figure 5.4: *AdaBoost Kaggle Submission Score*

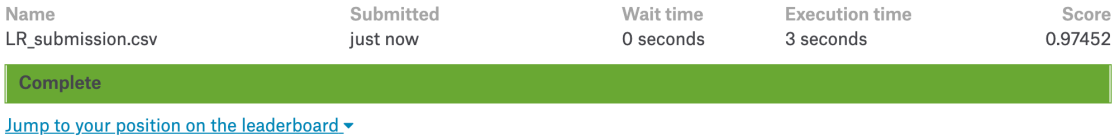

Figure 5.5: *Logistic Regression Kaggle Submission Score*

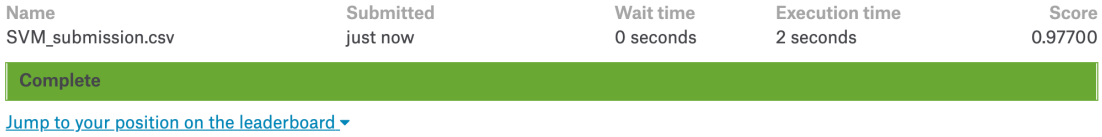

Figure 5.6: *SVM Kaggle Submission Score*

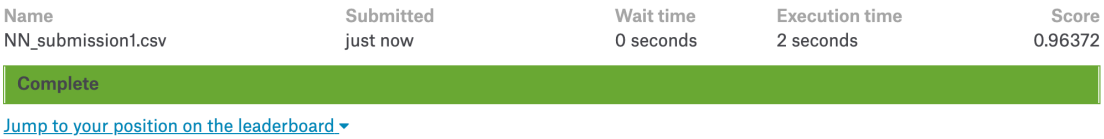

Figure 5.7: *Neural Network Kaggle Submission Score*

Chapter 6

Conclusions

The following table shows the rank of each machine learning model score after submitting the Kaggle test file:

Model	score
Support Vector Machine	0.9770
Logistic Regression	0.97452
Neural Network	0.96372
AdaBoost	0.95259
Random Forest	0.93068
Naive Bayes	0.91951
Decision Tree	0.84331

Table 6.1: *Kaggle test ranks*

In conclusion, we have apply and test seven machine models on the Kaggle Toxic Comment Classification Challenge problem, in the meantime, we have tried and designed multiple ways in order to train as well as test the model performance. According to the Kaggle test results, we can see that for high dimension text classification problems, large margin classifier such as SVM achieves a very high score, since it can cope with very high dimension data and even infinite number of dimensions. The large margin provides it extra ability to classify an unseen data with higher probability. We can also see that Logistic Regression achieves nearly the same score as SVM, this is because in essence, they both try to find a hyper-plane in high dimension space to separate the data correctly. Neural Network also obtains high score in this project, however, it tend to overfit the training data easily and we have to spend a large amount of time to fine tune the hyper-parameters. AdaBoost is an amazing machine learning model that can boost a weak learner to a strong one and we have already seen the powerful effect in this project. Another ensemble learning method Random Forest also obtains good scores in the text classification problem, however, it needs sufficient amount of data to train the model since it randomly select a subset of the data to build multiple subtrees. Multinomial Naive Bayes classifier achieves a moderate score but it is easy to understand and implement, it can be trained and test very fast and efficient, so that it is often used in Big Data analysis projects. Decision Tree classifier does not work very well in this text classification problem due to high dimensionality and unbalanced samples.

Acknowledgement

- Some materials derived for this report is from COMP9417 19T2 lecture notes
- The implementation of the code is based on python Sklearn machine learning library and python Keras deep learning library.
- The dataset we use for this project is from Kaggle competition: Toxic Comment Classification Challenge
(https://www.kaggle.com/c/jigsaw-toxic-comment-classification-challenge/overview)

Appendices

Appendix A

Hyper-Parameter Tuning

A.1 Decision Tree hyper-parameter tuning

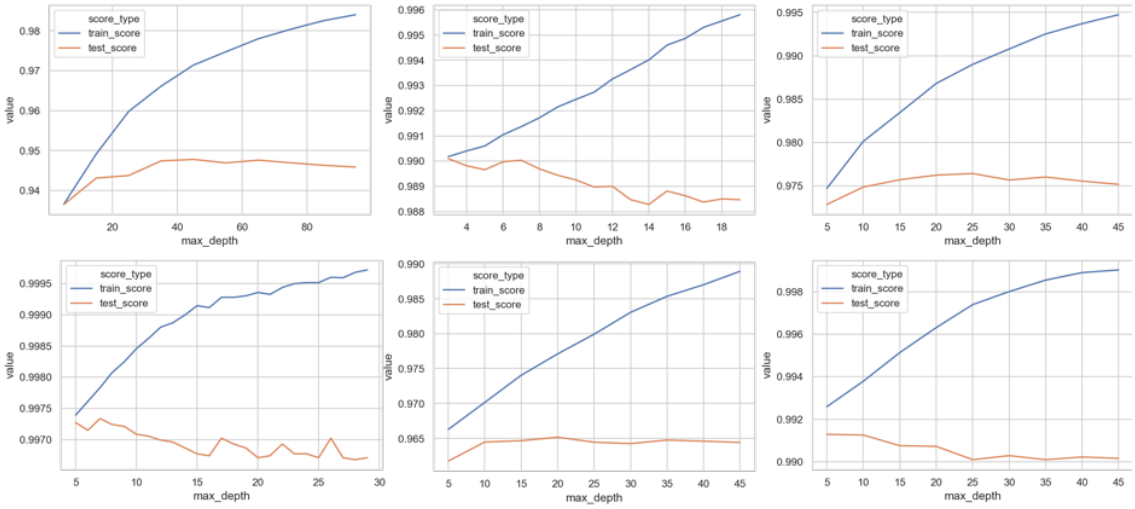

Figure A.1: *Training/validation score on different max_depth value for Decision Tree*

A.2 Random Forest hyper-parameter tuning

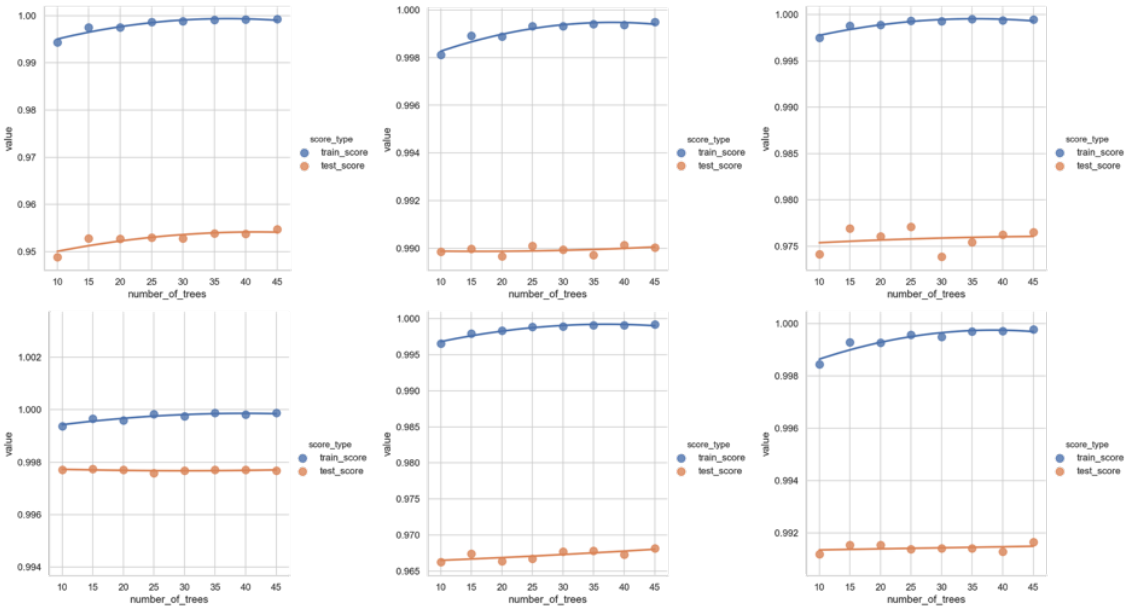

Figure A.2: *Training/testing scores on different number of subtrees for Random Forest*

A.3 AdaBoost hyper-parameter tuning

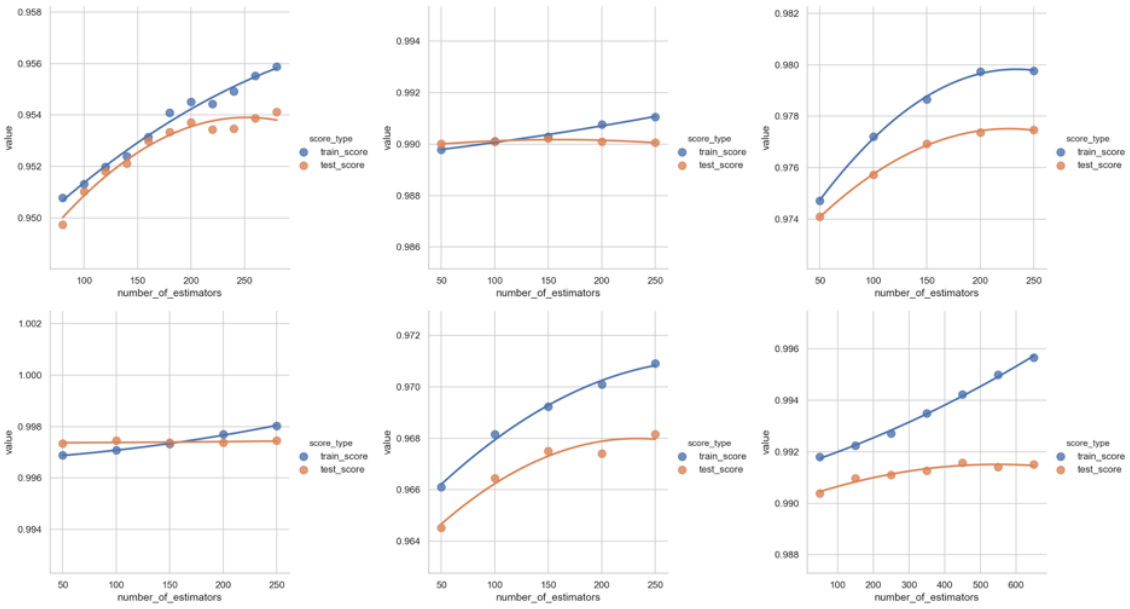

Figure A.3: *Training/testing score on different number of estimators for AdaBoost*

A.4 Logistic Regression hyper-parameter tuning

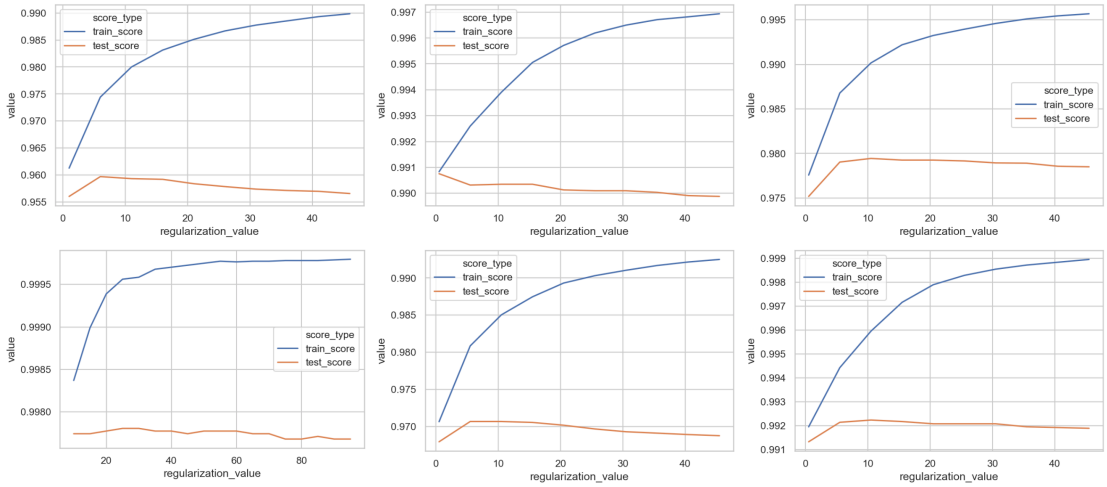

Figure A.4: *Training and test score on different regularisation values for Logistic Regression*

A.5 SVM hyper-parameter tuning

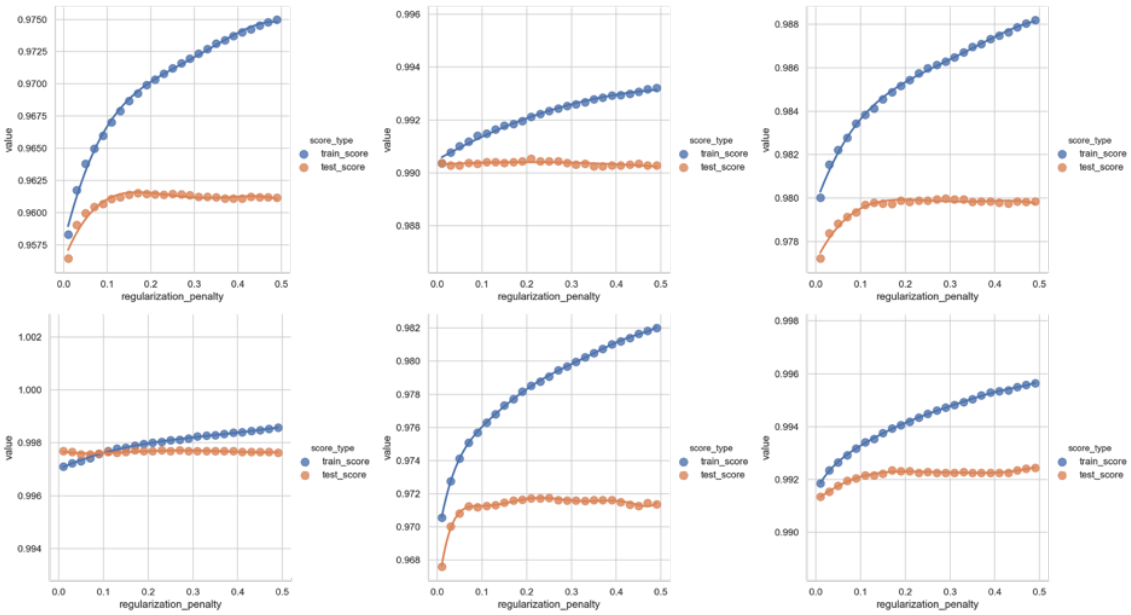

Figure A.5: *Training and validation score on different regularisation values for SVM*

A.6 Neural Network hyper-parameter tuning

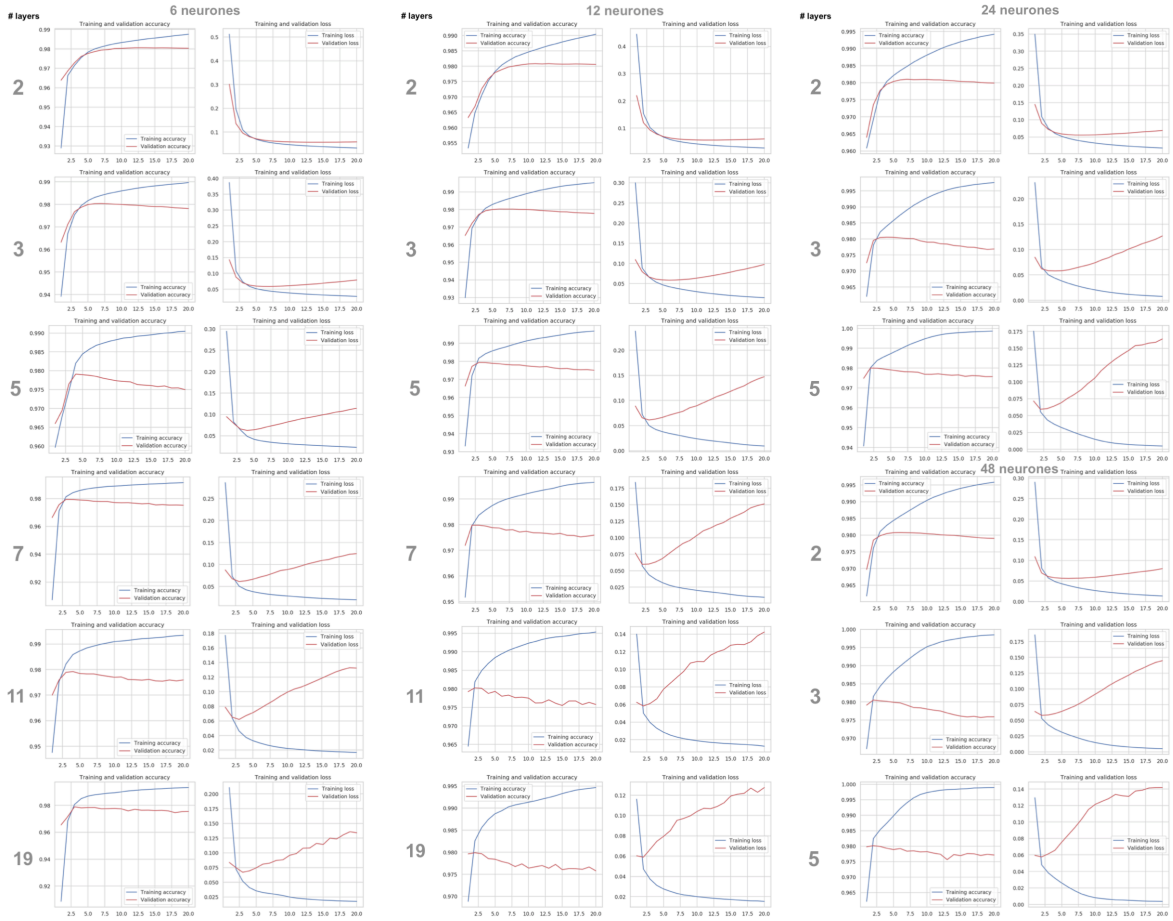

Figure A.6: *Training and validation score on different neurone number and layers*

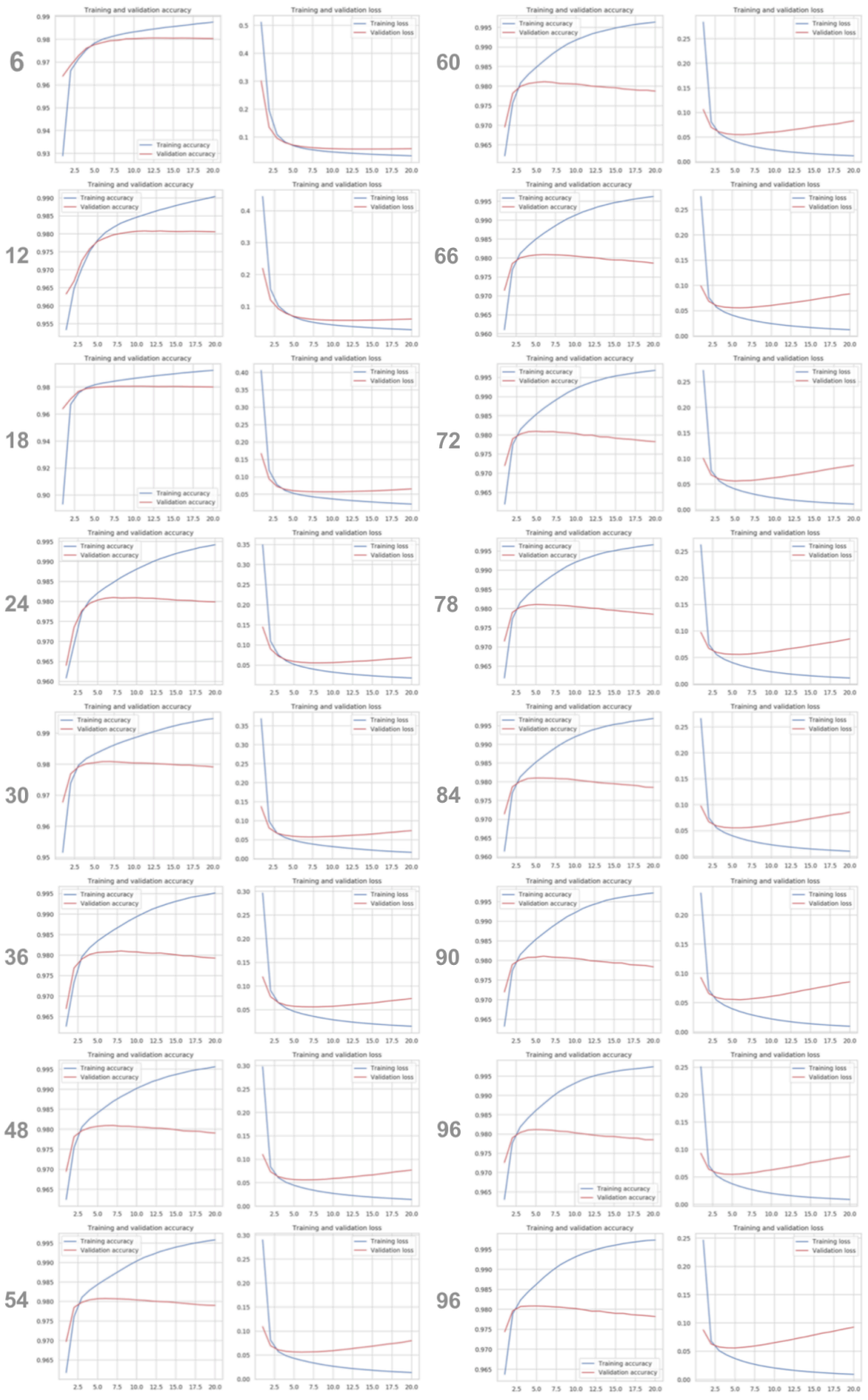

Figure A.7: *Training and validation score on different neurone numbers with 2 layers*

Appendix B

ROC plot for test results

B.1 Multinomial Naive Bayes test result

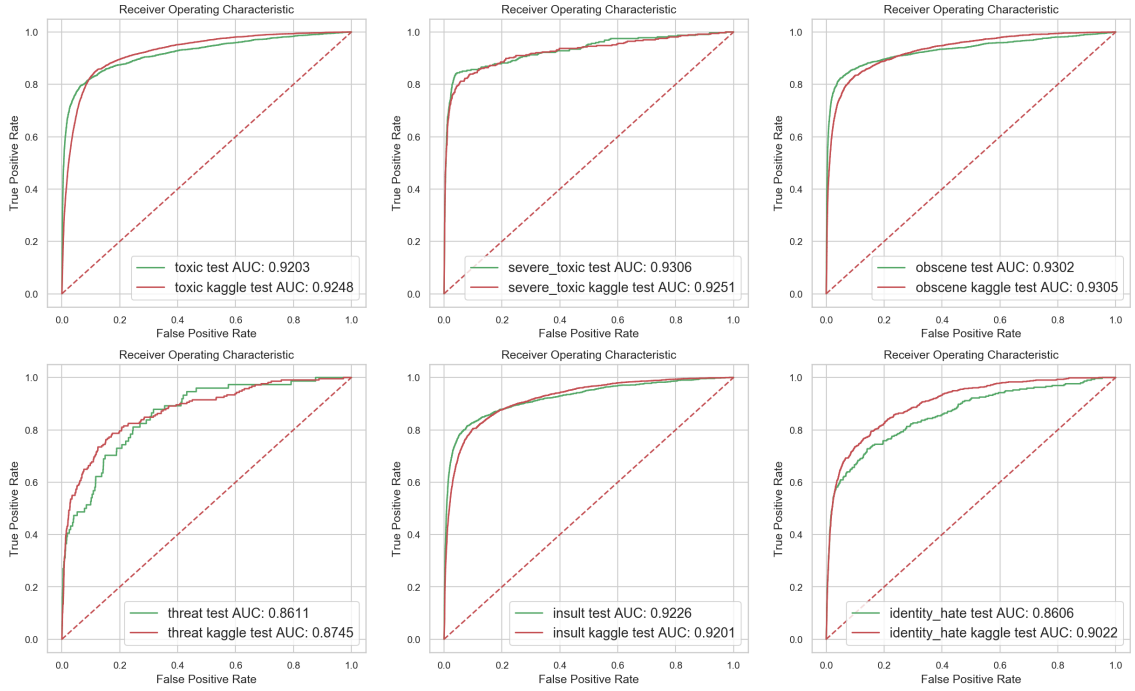

Figure B.1: *Apply Multinomial Naive Bayes on 20% split test dataset and Kaggle test dataset*

B.2 Decision Tree test result

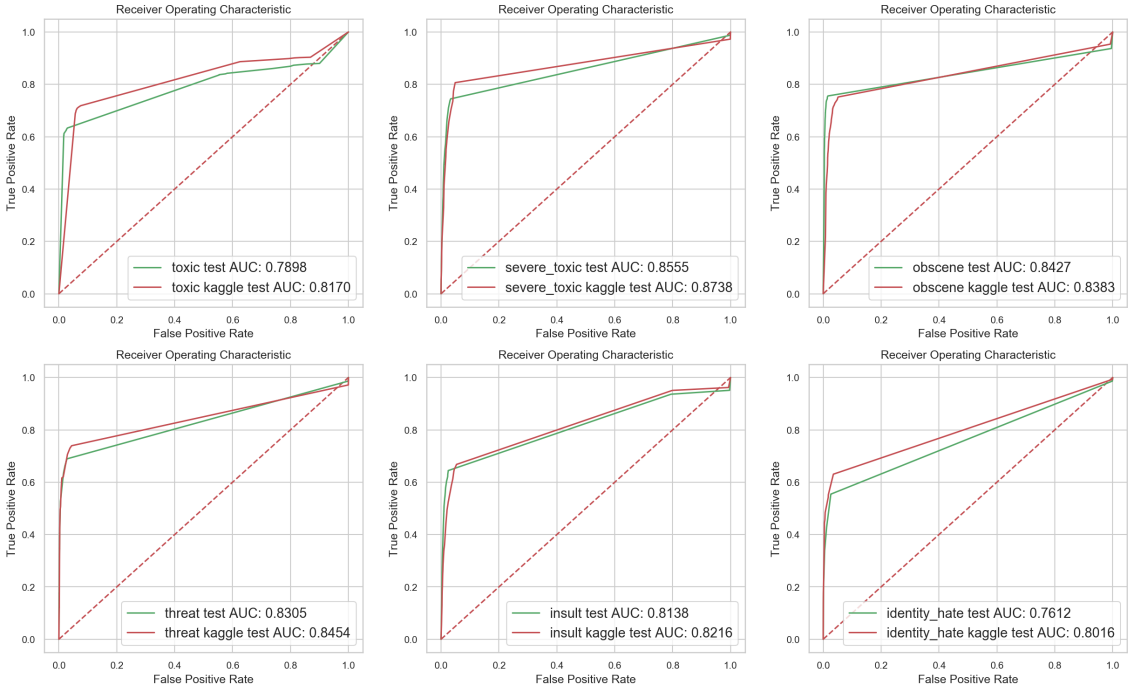

Figure B.2: *Apply Decision Tree on 20% split validation dataset and Kaggle test dataset*

B.3 Random Forest test result

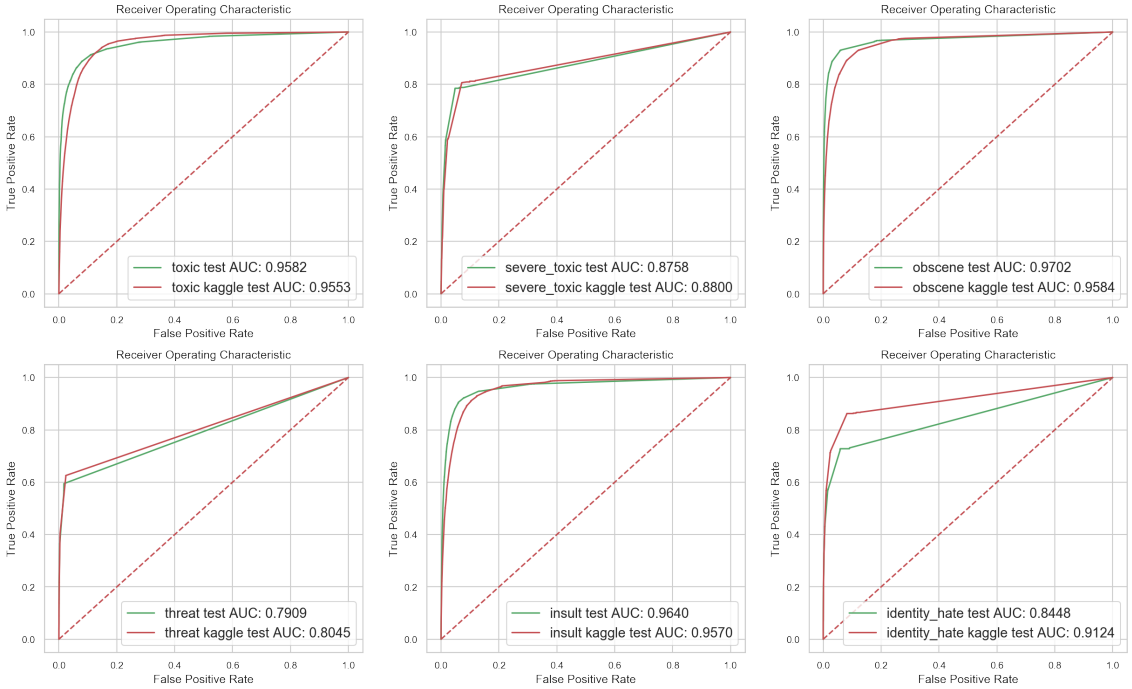

Figure B.3: *Apply Random Forest on 20% split validation dataset and Kaggle test dataset*

B.4 AdaBoost test result

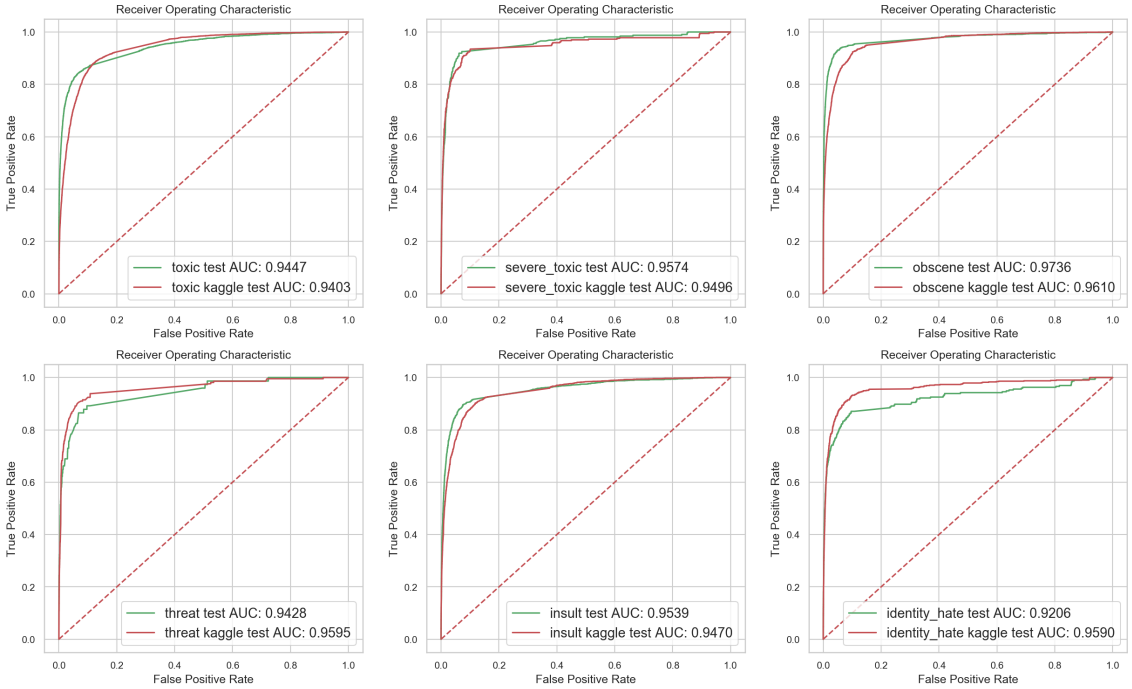

Figure B.4: *Apply AdaBoost on 20% split validation dataset and Kaggle test dataset*

B.5 Logistic Regression test result

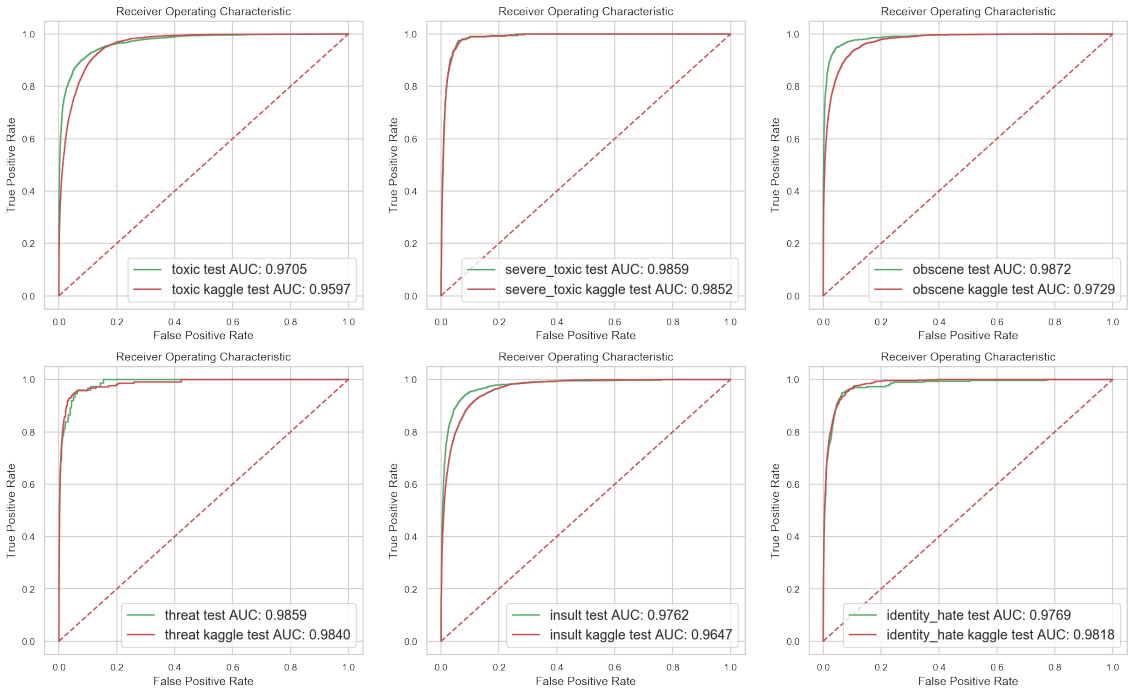

Figure B.5: *Apply Logistic Regression on 20% split validation dataset and Kaggle test dataset*

B.6 SVM test result

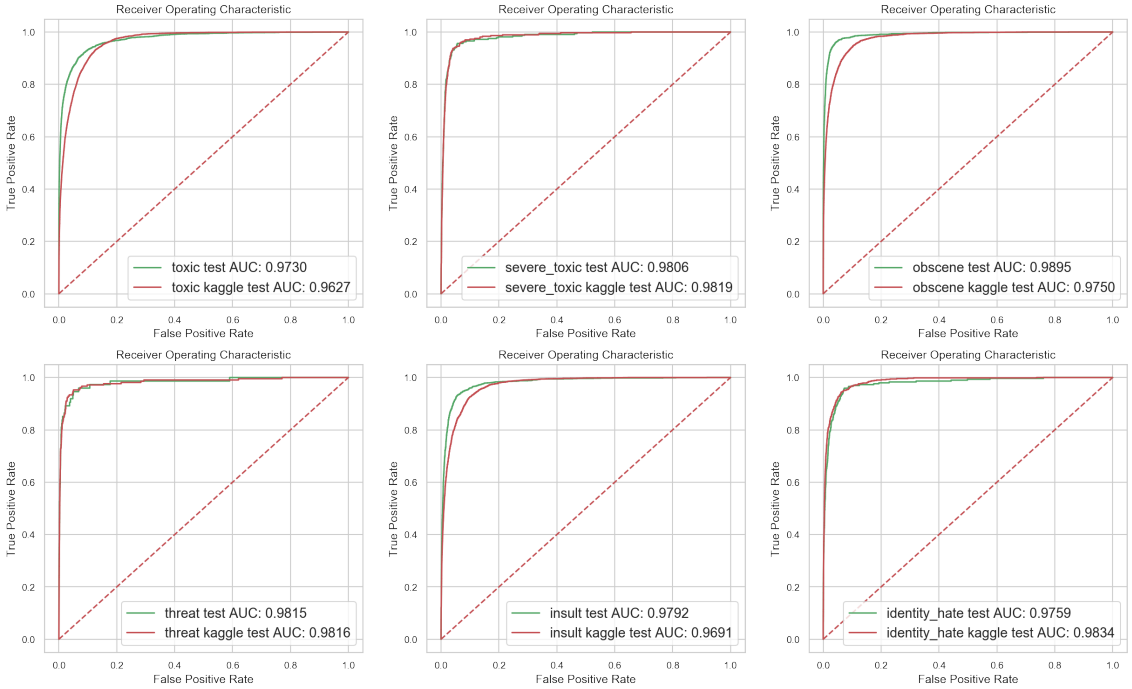

Figure B.6: *Apply SVM on 20% split validation dataset and Kaggle test dataset*

B.7 Neural Network test result

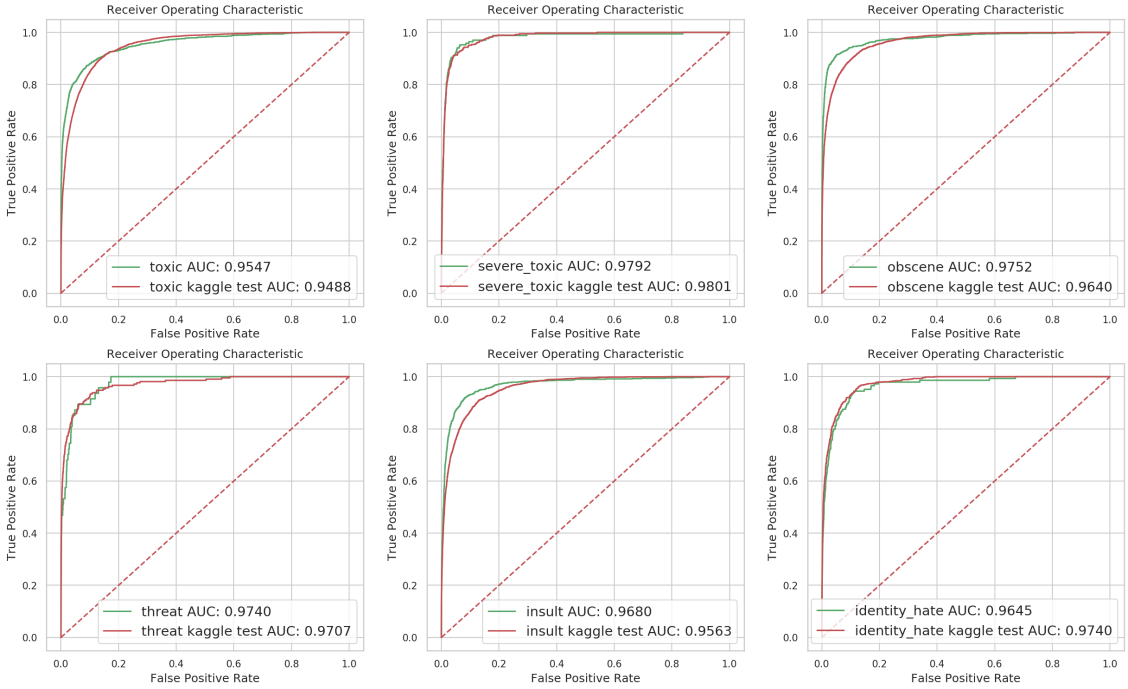

Figure B.7: *Apply Neural Network on 10% split test dataset and Kaggle test dataset*