Supervised Machine Learning Classification: Identifying Phishing Sites

Introduction

Business Problem & Solution

Phishing incidents have nearly doubled in frequency from 2019 to 2020, becoming the most common type of cybercrime with 241,324 cases in 2020 alone. Victims who fall prey to phishing attacks may suffer severe financial/security repercussions. Such attacks also negatively affect companies whose employees have been affected, as there is a risk of sensitive data being revealed. Self-validation is also tedious and prone to human error. Hence, without the use of machine learning, affected individuals, especially those who are less tech-savvy, are unable to validate fake websites quickly and accurately. Therefore, our team has decided to implement a solution through the utilization of machine learning techniques learnt in class.

An automated solution that accurately identifies fraudulent sites through building 5 different machine learning models and selecting the best at predicting fake web pages. As our problem is a binary classification problem, our model output will be the probability of a binary outcome - either phishing (1) or not phishing (0).

- X is all the attributes of the data
- Y will be the output of whether the input link is a phishing site

Dataset Description

The following table illustrates the dataset our team is using in our solution. The dataset was taken from Kaggle.

Data Attribute	Description
10,000 rows	5K from phishing webpages & 5K from legitimate webpages
50 columns (see Appendix)	48 features such as Subdomain level, Random String, etc. ● a = 19 (numerical) ● b = 29 (categorical encoded) ● 19 + 0.1(61) = 25.1 >= 10

Exploratory Data Analysis

Our team started off with exploratory data analysis to understand our data better. To determine which features had the greatest correlation to the CLASS_LABEL, the team decided to plot heatmaps (see Appendix) for easy visualisation. From the heat maps generated, we noticed that the feature FrequentDomainNameMismatch[1] was the most positively correlated with the highest positive value on the heatmaps. Followed by PctNullSelfRedirectHyperlinks[2] and InsecureForms[3]. The most negatively correlated was PctExtNullSelfRedirectHyperlinksRT[4].

Another observation made was the high correlation between several attributes such as jQueryComponents and NumAmpersand or RelativeFormAction and ExtFormAction. Moreover, though acknowledging that dropping out high correlated features would reduce overfitting and address storage and speed concerns, our team recognises the need to keep all features for model training due to the ever-changing nature of phishing sites where features that are not relevant in the past might be relevant in future. Thus, applying Principal Component Analysis (PCA) would be our best approach

[1] mismatch of popular domain names
[2] amount of self redirect hyperlinks in the website
[3] insecure forms located in the website
[4] amount of hyperlinks in HTML source code with different domain names

since it summarizes information from input variables by identifying important dimensions. Our team would investigate the impact of PCA on our models' performance by training models on data without PCA and with PCA separately and compare their performance metrics.

Data Splitting

Our team decided to split the data into a 7:3 ratio, with 70% delegated to model training and 30% for testing. 10 rows are separated and removed from the dataset to be used for demonstration purposes.

- 6,993 rows will be used for training
- 2,997 rows will be used for testing
- 10 rows will be used for the final demonstration

Performance Metrics

Metrics	**Description**
Accuracy [0, 1]	$Accuracy = \frac{True\ Prediction}{Total\ Prediction}$ Because of the balanced label distribution, an accuracy closer to 1 shows a better performance.
Precision [0, 1]	$Precision = \frac{True\ Positive}{True\ Positive + False\ Positive}$ reflects the level of false positives, with a value closer to 1 shows a better performance.
Recall [0, 1]	$Recall = \frac{True\ Positive}{True\ Positive + False\ Negative}$ reflects the level of false negatives, with a value closer to 1 shows a better performance.
F2-Score [0, 1]	$F_2 = \frac{Precision^{-1} + 2^2 \times Recall^{-1}}{1 + 2^2}$ gives a weighted harmonic means of Precision and Recall with a value closer to 1 showing a better performing model.
Log Loss [0, 1]	Log loss considers the probability, rather than the number of correct predictions and heavily penalizes models that are not very confident about a classification.
ROC-AUC [0.5, 1]	ROC curve helps to choose thresholds for discriminating positive or negative examples. AUC is the area under the ROC curve, and maximising this ensures the True Positive Rate is maximised and False Positive Rate is minimised, with AUC maximising to be near 1.

Subsequently, the team constructed a confusion matrix from our models' predictions with the values of true and false positives and negatives as well as calculated Precision and Recall. The team will focus on **maximising Recall**, thereby minimizing false negatives. In a phishing situation, it would be worse-off for a user if a website is classified as a false negative than as a false positive. Hence, the team would be using a weighted harmonic mean, ie. F-beta where beta is equal to 2 instead of the harmonic mean or F1 score.

Machine Learning Techniques

GridSearchCV for tuning

To maximize model performance, tuning our model hyperparameters was paramount. Hence, the team decided to use GridSearchCV - a simple but effective hyperparameter tuning algorithm. Essentially, every combination of parameters[5] is evaluated exhaustively and the performance metrics of the resulting models is computed. With this algorithm, a set of hyperparameters and tuning values can be easily configured as well as specifying the usage of Recall as our scoring metric. To measure and

[5] provided in a list

evaluate each machine learning algorithm carefully, four models per technique are constructed: Base model without PCA, Base model with PCA, Tuned model without PCA, Tuned model with PCA.

Logistic Regression

Logistic regression algorithm is a useful analysis method for classification problems. Given an input, it models the probability of a discrete outcome. It is usually used to classify the problem into binary outcomes (Edgar, 2017). Since the label of a phishing or non-phishing site will only result in 2 possible outcomes, it is a binary logistic regression.

GridSearchCV with RepeatedStratifiedKFold was used to tune the hyperparameters. A grid of parameters is defined and used to get the results of the tests. The parameters used are type of solver (newton-cg, lbfgs, liblinear, saga, sag), size of penalty (12) and c values (100, 10, 1.0, 0.1, 0.01). The model with the highest accuracy is the "best" model and hence the hyperparameters are used. In our case, the hyperparameters include the solver being newton-cg, size of penalty is 12 and c value is 100 for without PCA and the hyperparameters for the logistic regression with PCA, the best solver is liblinear, size of penalty is 12 and c value is 100 (Qiao, 2019). After tuning the model with the hyperparameters achieved, there was an improvement in the accuracy, precision, recall and f-score.

Model	**Accuracy Score**	**Precision**	**Recall**	**F2-Score**	**AUC Score**	**Log loss**
Logistic Regression before Tuning without PCA	0.939	0.929	0.950	0.946	0.981	0.176
Logistic Regression before Tuning With PCA	0.925	0.916	0.934	0.931	0.975	0.214
Logistic Regression after Tuning without PCA	0.940	0.930	0.951	0.946	0.982	0.175
Logistic Regression after Tuning with PCA	0.940	0.930	0.951	0.930	0.975	0.214

Decision Tree

Decision Tree belongs to the family of supervised learning algorithms but can be used to solve both regression and classification problems (Singh, n.d.). For this problem, a decision tree classifier is used to train and fit the model. Although feature scaling was performed, it did not affect the result; outliers in the dataset do not impact decision tree classification as data is split using scores calculated by the homogeneity of the resultant data points (Thenraj, 2020). Thus, feature scaling was unnecessary. The model without PCA (see Appendix) depicts the base model.

An ensemble technique - Adaboost is used to tune hyperparameters and built on top of the base model. At each iteration, the next classifier is built based on the past misclassification error improving predictive accuracy (Alto, 2020). However, just using AdaBoost can be challenging to configure as the algorithm has many hyperparameters that influence the behavior of the model. As such, it is a good practice to use a search process to discover a configuration of the hyperparameters that works best for a given predictive modeling problem (Brownlee, 2020). Therefore, GridSearchCV is performed together with Adaboost. There are four key hyperparameters and a range of popular performing values is used for each hyperparameter. Each configuration combination will be evaluated using repeated k-fold cross-validation and configurations are compared to the recall-score. Ultimately, tuning definitely boosts the performance scores as the best model turnt out to be the tuned decision tree classifier without PCA.

	Max Depth	**No. of estimators**	**Min. samples leaf**	**Learning Rate**

Options	2, 4, 6, 8, 10	10, 50, 100	5, 6, 7, 8, 9, 10	0.1, 1
Based model (With/without PCA)	3	-	1	-
Tuned model (With/without PCA)	4	100	10	0.1

Model	**Accuracy Score**	**Precision**	**Recall**	**F2-Score**	**AUC Score**	**Log loss**
Decision Tree Classifier without PCA	0.936	0.927	0.946	0.942	0.936	2.21
Decision Tree Classifier With PCA	0.839	0.854	0.818	0.825	0.839	5.55
Tuned Decision Tree Classifier without PCA	0.984	0.980	0.987	0.986	0.984	0.565
Tuned Decision Tree Classifier with PCA	0.983	0.982	0.984	0.984	0.983	0.588

Random Forest

Random Forest is an ensemble learning method that first selects a sample of the dataset at random with replacement (bootstrapping), and builds decision trees based on each sample. It then takes the average result of these decision trees (aggregating). With bootstrapping , overfitting can be avoided, since the models are only trained on a sample of the dataset, and not its entirety.

The classifier takes in several parameters. We have chosen four to focus on, ie. number of decision trees that will be built (n_estimators), the maximum depth of those trees (max_depth), the minimum number of samples at a leaf node, ie. splits will occur only when the number of training samples is greater than or equal to this number in both left and right branches (min_samples_leaf), and random state which controls the randomness of the bootstrapping process (random_state). These four were chosen as they are typically the most important hyperparameters for Random Forest. For example, in experiments studying the importance of hyperparameters across multiple datasets, the most important parameters included min_samples_leaf and n_estimators (Coelho, 2020). Max_depth may not be particularly important but was included to keep Random Forest standardised (for comparison purposes later) with Decision Trees, which also tunes this hyperparameter.

To find the best values for these hyperparameters, we used GridSearchCV to exhaustively search through all combinations of the specified hyperparameters. The ranges for each hyperparameter were chosen with both evaluation metrics and grid search time taken into consideration. We also undertook some research to find a suitable range. For example, 64 to 128 trees can be taken as a good range for n_estimators (Oshiro et al., 2012), hence we supplied a range of 50 to 140 (inclusive) to grid search. The chosen hyperparameters and the model performance metrics are indicated in the two tables below.

Type	**No. of estimators**	**Max depth**	**Min_samples_leaf**	**Random State**
Options (only for GridSearchCV)	50, 60, 70, 80, 90, 100, 110, 120, 130, 140	2, 3, 4	10, 15, 20, 25	10, 20, 30, 40, 50
No tuning, PCA / non-PCA	100	0	1	None
GridSearchCV, non-PCA	60	4	15	30
GridSearchCV, PCA	140	4	10	20

Dataset	**Training Accuracy**	**Testing Accuracy**	**Precision**	**Recall**	**F2 Score**	**Log Loss**	**AUC**
Base Classifier	1.000	0.979	0.979	0.979	0.979	0.087	0.997
Base Classifier & PCA	1.000	0.954	0.950	0.959	0.957	0.163	0.991
Tuned Classifier	0.962	0.952	0.949	0.955	0.954	0.280	0.989
Tuned Classifier & PCA	0.915	0.913	0.910	0.916	0.915	0.371	0.967

Artificial Neural Network (ANN)

A non-linear statistical data modeling tool consisting of a group of multiple neurons in three (Input, Hidden & Output) or more interconnected layers. ANNs are able to find relationships between inputs and outputs and can even detect complex nonlinear relationships. Furthermore, neural networks have high computing power and are hence very suitable for our large database. However, something to be wary of is the low interpretability of the ANNs, especially when there are many nodes/layers. ANNs are like a black box and it is difficult to figure out how they derive their outputs.

Our implemented ANN Model involves 4 Dense layers with ReLu and Sigmoid as the activation function for the input and output layer respectively. Other parameters are binary cross entropy for loss function and AdamOptimizer for optimizer. Further explanation and details on ANN implementation can be found in the Appendix. To tune the ANN model, we chose to optimize the learn_rate and the batch_size of the model via GridSearchCV. Learning rate is used to tune the parameters of the AdamOptimizer, whereas batch size is the number of samples to be processed before the model is updated.

	learn_rate	**batch_size**
Options	0.01, 0.1, 0.2, 0.3	10,20,40,60
Without PCA	0.01	20
With PCA	0.2	10

To implement GridSearch, a build_classifier function that takes in the two hyperparameters as input was created. This outputs a model with the same characteristics as the base models, with the only difference being the learning rate and batch size. Then, we pass build_classifier to the KerasClassifier wrapper, and define epochs = 10 to ensure each iteration of the models are also trained for 10 epochs, like the base models. We then pass each output model from the KerasClassifier, together with the dictionary of hyperparameters, and our model evaluation tool, which in this case is recall as we are optimizing for recall. The performance metrics are recorded in the table below.

Model	Accuracy	Precision	Recall	F2 Score	AUC Score	Log Loss
Base Model	0.952	0.942	0.963	0.959	0.988	1.660

Base Model with PCA	0.947	0.947	0.947	0.959	0.985	1.821
Tuned Model	0.955	0.939	0.973	0.966	0.990	1.556
Tuned Model with PCA	0.947	0.938	0.958	0.954	0.987	1.821

k-Nearest Neighbours (KNN)

KNN model is a classification model with the ability to perform classification while not requiring many specific assumptions. This is a simple algorithm with no explicit training phase and it is very fast to train. KNN keeps all the training data and predicts based on the majority label of the nearest known data using a set similarity measure of the new input and the dataset. This also means that the technique would require a high amount of memory, but we are still interested in pursuing this model because it is easy to train and implement and its versatility would also allow new data to be added in without impacting the predicting power of the algorithm. This is necessary since the number of phishing sites would only grow and become harder to generalize the phishing patterns. KNN avoids having to generalize and classify the sites by looking at the similarity of other phishing and non-phishing sites.

For hyperparameter tuning, we went with the more efficient hyper parameter tuning with GridSearchCV and Stratified K Fold cross-validation. The hyperparameter values were chosen around the default values since the team could not find any sources regarding hyperparameter values to choose for KNN tuning. Standard Scaler and PCA (if necessary) is fit on the pipeline since cross-validation data should be presented as an unseen dataset, so using the already preprocessed data would be considered pipeline leakage. The hyperparameters selected are indicated below.

	n_neighbors	weights	leaf_size	p	metric
grid_params	1,2,3,4,5,6,7, 8,9,10	“uniform”, “distance”	20,25,30,35,40	1,2	“minkowski”, “euclidean”
Without PCA	7	“distance”	20	1	“minkowski”
With PCA	3	“distance”	35	2	“minkowski”

Plotting the values of the metrics performance with different values of n_neighbors from 1 to 10 while keeping the other hyperparameters constant, we find the best value of n_neighbors to be 7 (see Appendix). The models performance metrics are recorded in the two tables below.

Model	**Accuracy**	**Precision**	**Recall**	**F2-Score**	**AUC Score**	**Log loss**
KNN Base	0.946	0.946	0.952	0.950	0.946	0.579
KNN Base & PCA	0.945	0.945	0.950	0.948	0.945	0.617
KNN Tuned	0.959	0.959	0.978	0.970	0.959	0.347
KNN Tuned & PCA	0.947	0.948	0.959	0.955	0.948	0.827

Results and Comparison

As shown from the previous performance metrics tables, the best performing models for all machine learning techniques are the tuned models from GridSearchCV, since GridSearchCV would find the most optimal combination of hyperparameters. Furthermore, PCA reduced the dimensionality of the dataset, which would have resulted in the model performing slightly worse. The best model for each ML technique was selected and their performance metrics are added to the table below.

Model	**Accuracy Score**	**Precision**	**Recall**	**F2-Score**	**AUC Score**	**Log loss**
Logistic Regression	0.940	0.930	0.951	0.946	0.982	0.175
Decision Tree	0.984	0.980	0.987	0.986	0.984	0.565
Random Forest	0.951952	0.948871	0.955214	0.953939	0.989396	0.279604
ANN	0.955	0.939	0.973	0.966	0.990	1.556
KNN	0.959	0.959	0.978	0.970	0.959	0.347

From the training process, an interesting observation is that Tree-based algorithms (i.e. Decision Tree, Random Forest) does not require scaling. In contrast, Linear Regression and KNN need scaling for better performance. This is because Decision trees and ensemble methods do not require feature scaling to be performed as they are not sensitive to the variance in the data (Thenraj, 2020).

In what scenarios, which technique could be better?

In our business case of phishing sites detection, any classification model that can possibly warn users of a high risk phishing site would prove to be valuable. However, in cases when it is a necessary website for the business, being blocked and labeled as a phishing site is frustrating to say the least. In cases like this, justification on why one is a suspicious phishing site is important. Tree-based algorithms with its feature importance analysis would be able to indicate the feature importance and provide more concrete answers to this than black-box models like ANN.

Since we optimise on Recall, the tuned model with the best Recall score is Decision Tree, as shown in the table above. In addition, out of all the other scores: Accuracy, Precision, F2-Score, AUC Score, Log Loss, the decision tree model had produced the best overall scores for three out of five of the scores amongst all the other models. As such, our team decided to use the Decision Tree as the best performing model.

Challenges

One of the team's greatest bottlenecks is the lack of computational power to run all untuned and tuned models efficiently in a condensed jupyter notebook environment. In the future, other collaboration tools and methods would be researched and considered. Additionally, it is difficult to standardize the hyperparameters tuning as different models react differently to different concepts. Random Forest may not be suitable for Adaboost whereas Decision Tree could use it to improve performance. To alleviate this problem and the problem of having multiple parameters to train, we have conducted our own literature review for the individual models so that the hyperparameters can be tuned most efficiently within a reasonable grid search range.

One challenge when deploying our chosen AdaBoost Decision Tree Model is that this model is very sensitive to overfitting since the tree decides the splits based on the Entropy information from the training data and AdaBoost might give higher weight for noisy data which would be difficult to classify. Thus, it is important to clean data well before feeding them as input to train this model in the future. As phishing websites evolve, there are bound to be more features to take in as inputs. This increase in dimensionality would result in an increase in dimensionality and a possible overfitting on unuseful features. To counter this problem for a growing dataset, we can consider applying PCA and feature engineering to the future models.

Insight

What message do you want to deliver at the end of the project?

While writing code to build and train the models is fairly straightforward with the help of libraries like SKLearn, it is important to understand how each model works under the hood, to understand how to better tune it according to business needs. With a clear understanding of the models, the process of building them becomes a lot smoother.

What did you learn from this project besides building models?

There are many things to consider before a model can be built, such as hyperparameters. These usually have less to do with any machine learning techniques or model-building, and are dependent on the business use case, and/or the type, size and format of data provided.

How does PCA affect our models?

From our models above, the ones with PCA required less run time. However, the ones with PCA performed slightly worse than the ones without PCA, as the features are not very highly correlated as shown in the heatmap.

What features were more important from our model?

Our chosen model AdaBoost deprives its feature importance from the average feature importance provided by its Decision Tree base classifier. The top five important features determined by the mean decrease in entropy are NumDots, NumDash, InsecureForms, ExtMetaScriptLinkRT, QueryLength. Unfortunately, the top five features identified here are not the features that show the highest correlations to the CLASS_LABEL (see Appendix).

Fortunately, this can be explained by the bias toward the high-cardinality features of decision trees. To avoid this problem, permutation feature importance can be used to overcome limitations of the impurity-based feature importance and can be computed on an unseen test set (Pedregosa et al., 2011). The top five features after applying PctExtHyperlinks, PctNullSelfRedirectHyperlinks, PctExtNullSelfRedirectHyperlinksRT, InsecureForms, PctExtResourceUrls. As mentioned in the EDA, PctNullSelfRedirectHyperlinks, InsecureForms and PctExtNullSelfRedirectHyperlinksRT are highlighted as the most correlated to the CLASS_LABEL.

Observing that FrequentDomainNameMismatch is only ranked 10th in terms of importance, our team attempted to explain this pattern as correlation indicates the linear correlation between a feature and CLASS_LABEL. Decision trees in AdaBoost rank the importance with the mean information gained from entropy changes, resulting in features that non linearly have higher importance in the binary classification task.

Should we use all the features in the dataset or select the more important features?

Undoubtedly, selecting the more important features would help reduce overfitting, improve accuracy and reduce training time for the model. However, our team also needed to take into consideration the ever-changing nature of phishing sites. If the attackers were to know which features were used to predict the classification of a phishing site, they would simply use the features which were not used in the prediction, in order to avoid detection. Hence, we felt that it is necessary to include all features as they may be important in the classification of future phishing sites.

References

Alto, V. (2020, January 11). *Understanding AdaBoost for Decision Tree*. Towards Data Science. https://towardsdatascience.com/understanding-adaboost-for-decision-tree-ff8f07d2851

Brownlee, J. (2020, May 1). *How to Develop an AdaBoost Ensemble in Python*. Machine Learning Mastery. https://machinelearningmastery.com/adaboost-ensemble-in-python/

Brownlee, J. (2020, August 27). *Tune hyperparameters for Classification Machine Learning Algorithms*. Machine Learning Mastery. https://machinelearningmastery.com/hyperparameters-for-classification-machine-learning-algorithms/.

Brownlee, J. (2020, November 8). *A gentle introduction to probability scoring methods in Python*. Machine Learning Mastery. https://machinelearningmastery.com/how-to-score-probability-predictions-in-python/.

Brownlee, J. (2021, January 12). *How to use ROC curves and precision-recall curves for classification in Python*. Machine Learning Mastery. https://machinelearningmastery.com/roc-curves-and-precision-recall-curves-for-classification-in-python/.

Coelho, A. (2020, June 17). *Narrowing the search: Which hyperparameters really matter?* Dataiku. Retrieved November 12, 2021, from https://blog.dataiku.com/narrowing-the-search-which-hyperparameters-really-matter.

Edgar, T. W. (2017). Chapter 4 - Exploratory Study. In D. O. Manz (Ed.), *Research Methods for Cyber Security* (pp. 95–130). essay, SYNGRESS.

Oshiro, T. M., Perez, P. S., & Baranauskas, J. A. (2012). How many trees in a random forest? *Machine Learning and Data Mining in Pattern Recognition*, 154–168. https://doi.org/10.1007/978-3-642-31537-4_13

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., & Duchesnay, E. (2011). Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, *12*, 2825–2830.

Qiao, F. (2019, January 9). Logistic Regression Model Tuning with scikit-learn — Part 1. Towards Data Science. https://towardsdatascience.com/logistic-regression-model-tuning-with-scikit-learn-part-1-425142e01af5

Singh, N. C. (n.d.). *Decision Tree Algorithm, Explained*. KDNuggets. https://www.kdnuggets.com/2020/01/decision-tree-algorithm-explained.html

Thenraj, P. (2020, June 22). *Do Decision Trees need Feature Scaling?* Towards Data Science. https://towardsdatascience.com/do-decision-trees-need-feature-scaling-97809eaa60c6

Appendix

Dataset Features

```
#   Column
---  ------
0   id
1   NumDots
2   SubdomainLevel
3   PathLevel
4   UrlLength
5   NumDash
6   NumDashInHostname
7   AtSymbol
8   TildeSymbol
9   NumUnderscore
10  NumPercent
11  NumQueryComponents
12  NumAmpersand
13  NumHash
14  NumNumericChars
15  NoHttps
16  RandomString
17  IpAddress
18  DomainInSubdomains
19  DomainInPaths
20  HttpsInHostname
21  HostnameLength
22  PathLength
23  QueryLength
24  DoubleSlashInPath
25  NumSensitiveWords
26  EmbeddedBrandName
27  PctExtHyperlinks
28  PctExtResourceUrls
29  ExtFavicon
30  InsecureForms
31  RelativeFormAction
32  ExtFormAction
33  AbnormalFormAction
34  PctNullSelfRedirectHyperlinks
35  FrequentDomainNameMismatch
36  FakeLinkInStatusBar
37  RightClickDisabled
38  PopUpWindow
39  SubmitInfoToEmail
40  IframeOrFrame
41  MissingTitle
42  ImagesOnlyInForm
43  SubdomainLevelRT
44  UrlLengthRT
45  PctExtResourceUrlsRT
46  AbnormalExtFormActionR
47  ExtMetaScriptLinkRT
48  PctExtNullSelfRedirectHyperlinksRT
49  CLASS_LABEL
dtypes: object(50)
```

Correlation Heatmap between features

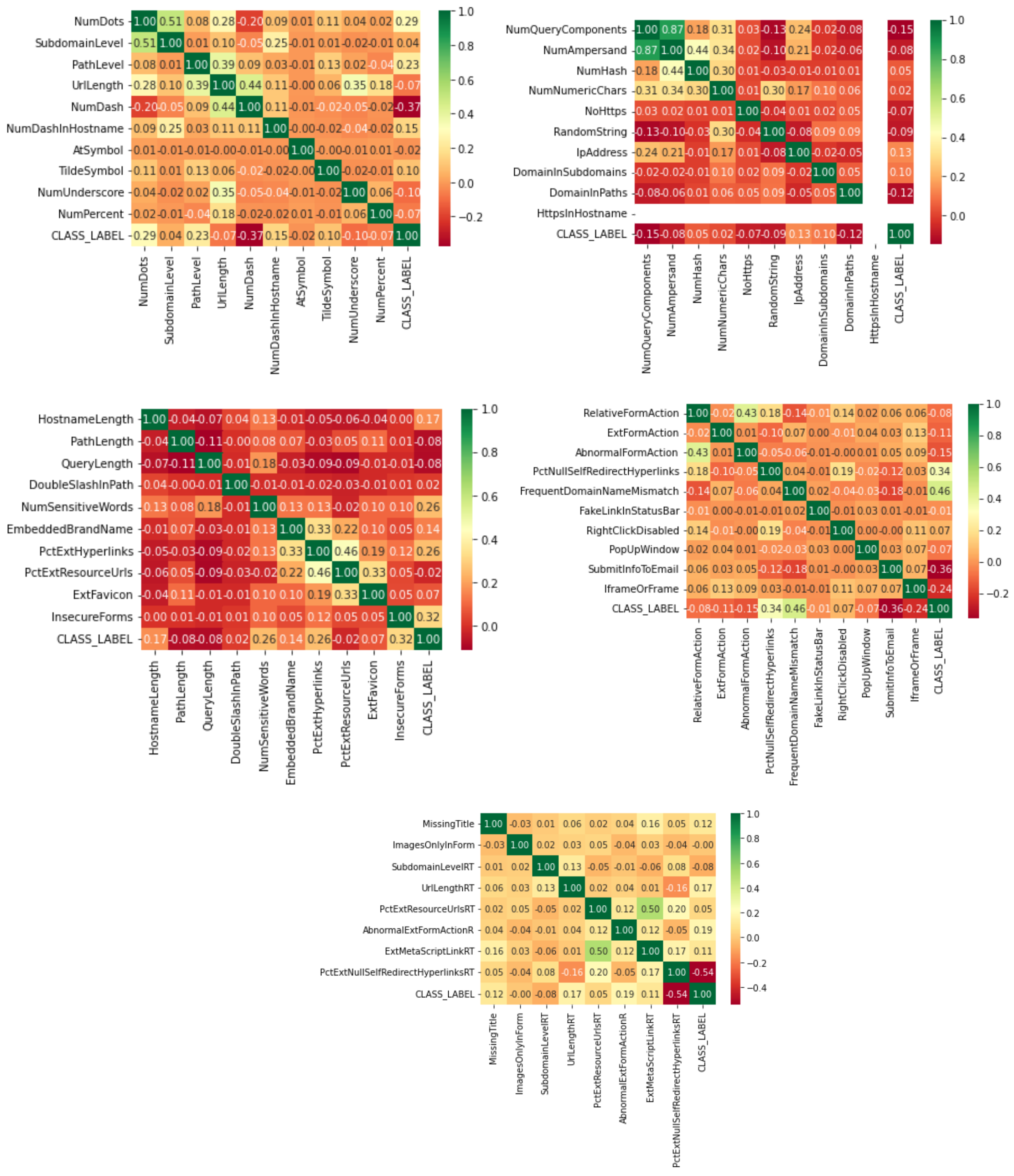
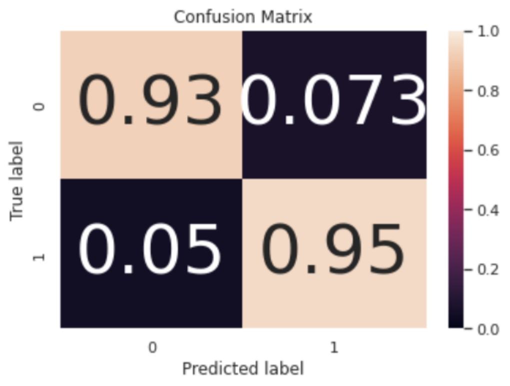
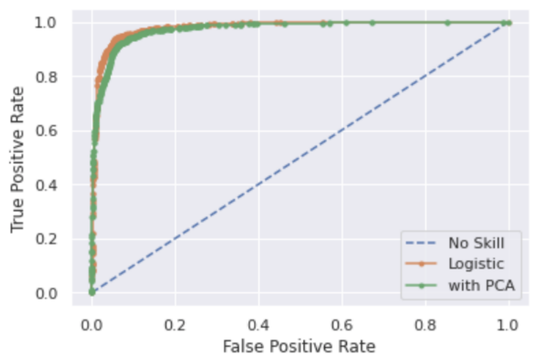


Figure 1. Correlation Heatmap of Features and Class Label

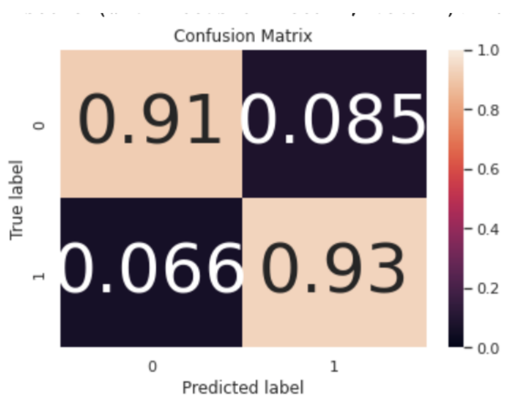
<u>Logistic Regression confusion matrices and ROC curves</u>



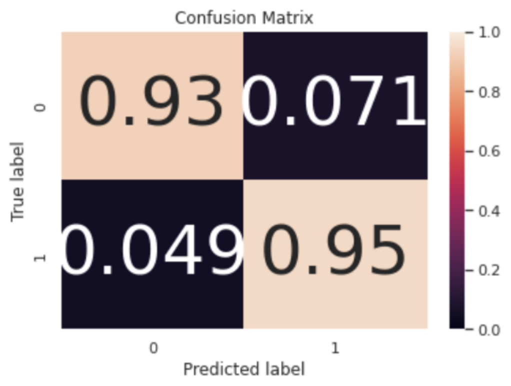
Confusion matrix of Logistic Regression without PCA



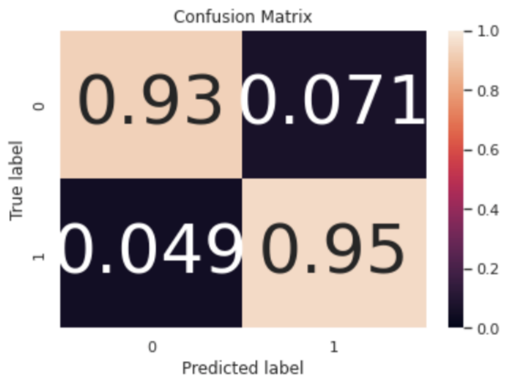
ROC Curve of Logistic Regression without PCA



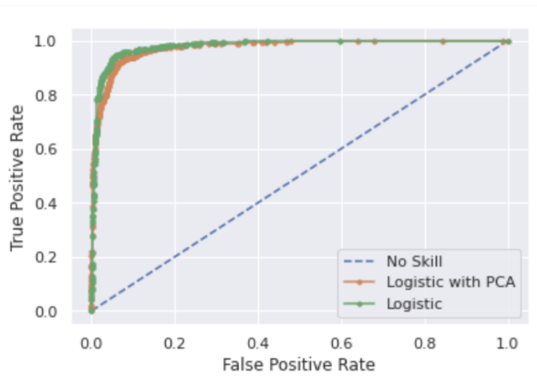
Confusion matrix of Logistic Regression with PCA



Confusion matrix of Logistic Regression GridSearchCV with PCA



Confusion matrix of Logistic Regression GridSearchCV without PCA



ROC Curve of Logistic Regression GridSearchCV without PCA

The base model of decision tree classifier

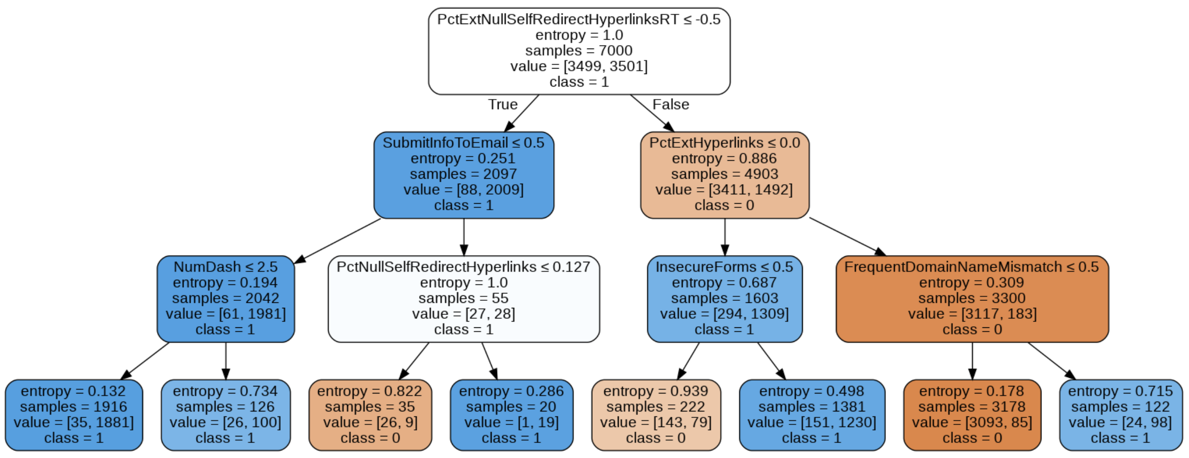
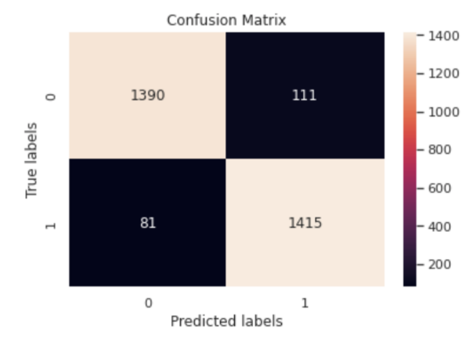
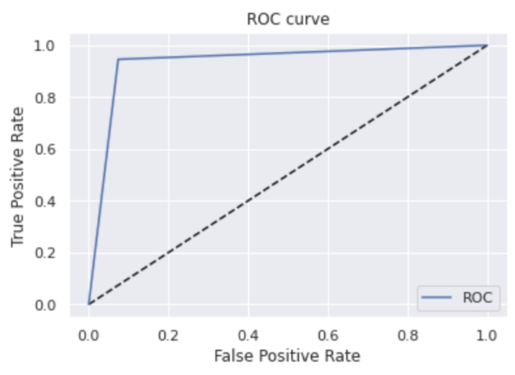


Figure 2: The base model of decision tree classifier

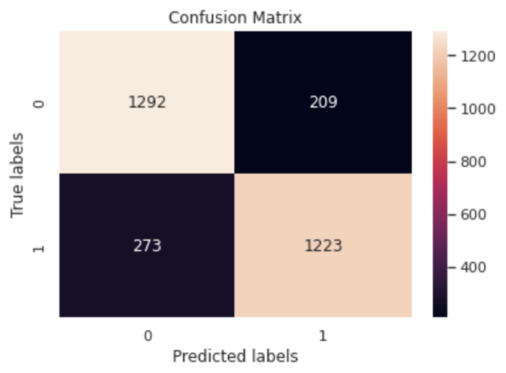
Decision Tree classifier confusion matrices and ROC curves



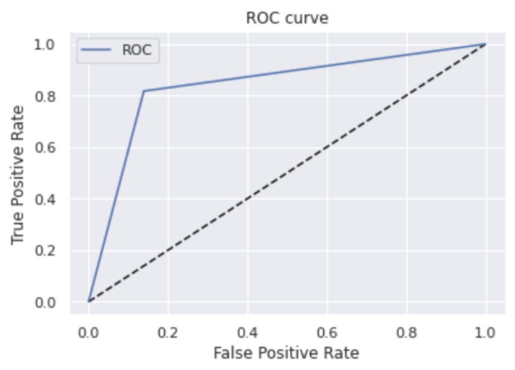
Confusion matrix of Decision Tree without PCA



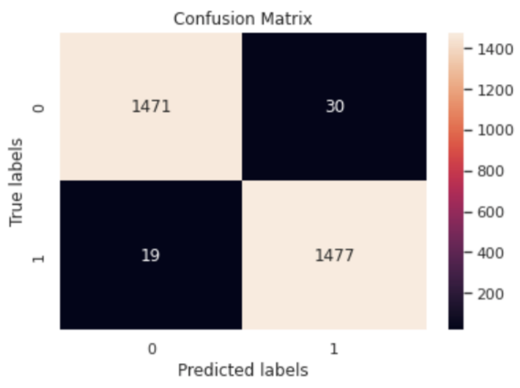
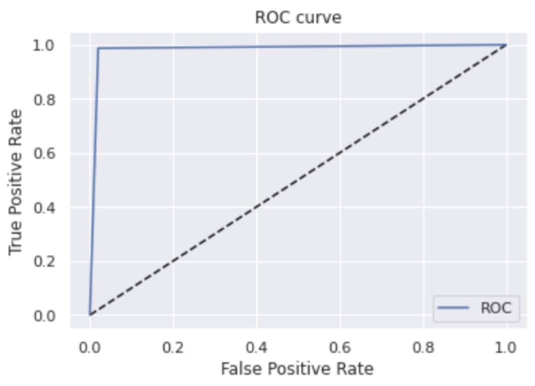
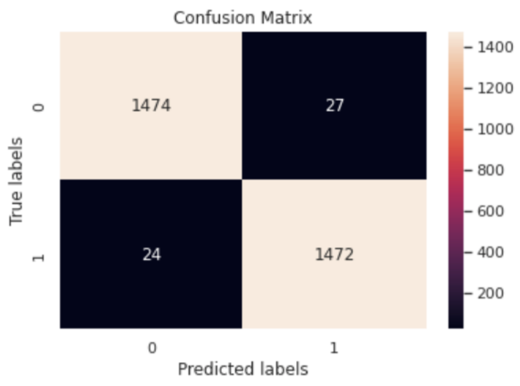
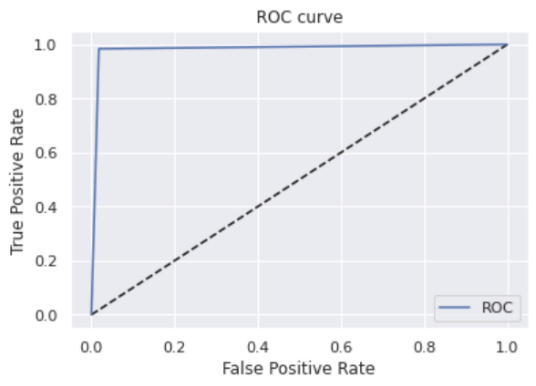
ROC Curve of Decision Tree without PCA



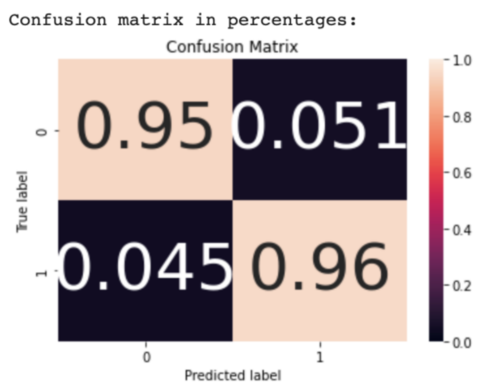
Confusion matrix of Decision Tree with PCA



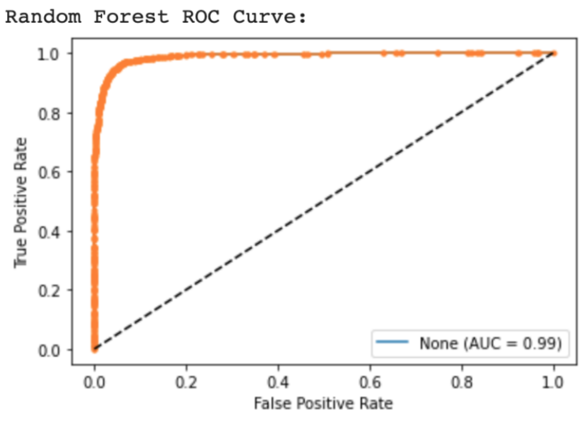
ROC Curve of Decision Tree with PCA

 *Confusion matrix of Decision Tree GridSearchCV Adaboost without PCA*	 *ROC Curve of Decision Tree GridSearchCV Adaboost without PCA*
 *Confusion matrix of Decision Tree GridSearchCV Adaboost with PCA*	 *ROC Curve of Decision Tree GridSearchCV Adaboost with PCA*

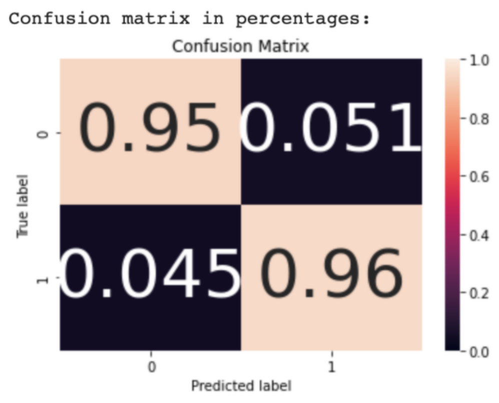
Random Forest classifier confusion matrices and ROC curves


Confusion matrix in percentages:

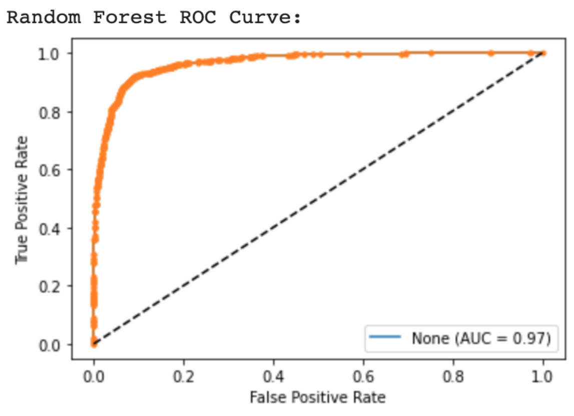
Tuned Random Forest classifier confusion matrix (non-PCA)


Random Forest ROC Curve:

Tuned Random Forest ROC Curve (non-PCA)

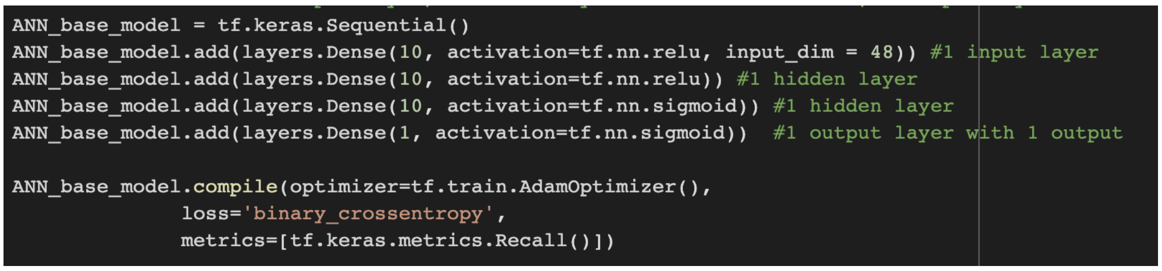

Confusion matrix in percentages:

Tuned Random Forest classifier confusion matrix (PCA)


Random Forest ROC Curve:

Tuned Random Forest ROC Curve (PCA)

Implementation of the ANN Model


```
ANN_base_model = tf.keras.Sequential()
ANN_base_model.add(layers.Dense(10, activation=tf.nn.relu, input_dim = 48)) #1 input layer
ANN_base_model.add(layers.Dense(10, activation=tf.nn.relu)) #1 hidden layer
ANN_base_model.add(layers.Dense(10, activation=tf.nn.sigmoid)) #1 hidden layer
ANN_base_model.add(layers.Dense(1, activation=tf.nn.sigmoid))  #1 output layer with 1 output

ANN_base_model.compile(optimizer=tf.train.AdamOptimizer(),
              loss='binary_crossentropy',
              metrics=[tf.keras.metrics.Recall()])
```

For each ANN model, we used 4 Dense layers. The first layer is the input layer and it is of input dimension 48 (30 if we are using PCA data). The next two layers are the hidden layers. The last layer outputs a single value, since we just want the result of a classification.

For the first two layers, we used ReLu to try and reduce the impact of the vanishing gradient problem, which would affect the learning of our model. Furthermore, training networks with ReLu is faster, which will save time when we are training the model. For the last two layers, we used Sigmoid to transform the values into some output between 0 and 1 for our binary classification.

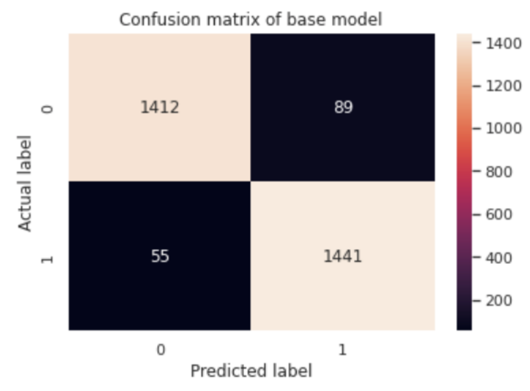
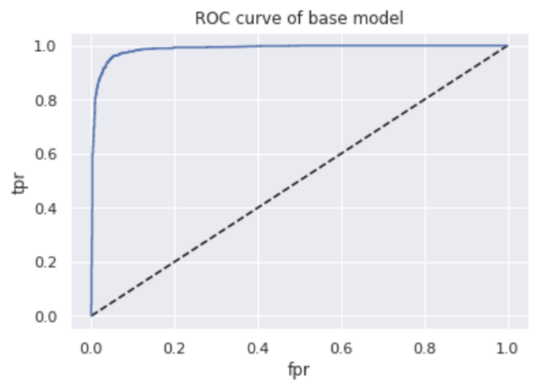
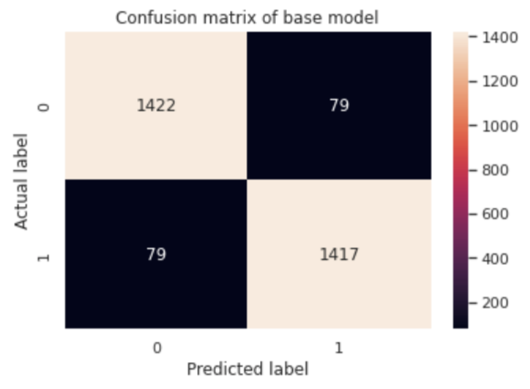
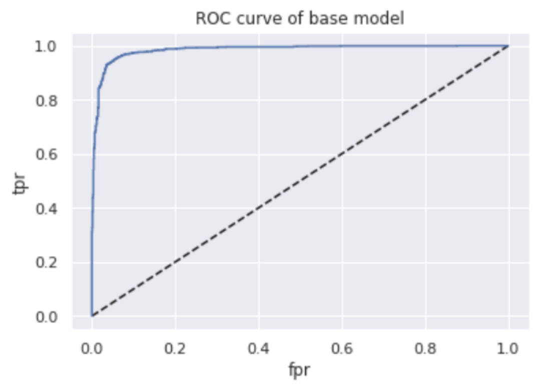
For the loss function, we used binary cross entropy: Binary since we only have 2 label classes; Cross-entropy to compare each of the predicted probabilities to actual class output. The loss function then calculates a score that penalizes the probabilities based on the distance from the expected value.

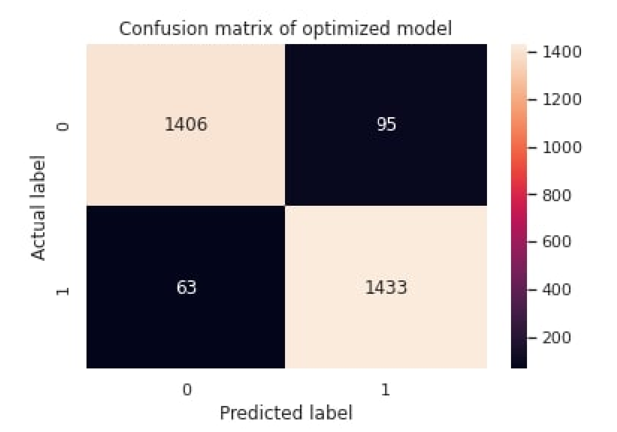
For model compilation, we used AdamOptimizer as the optimizer and Recall as the metric.

Build_classifier code for ANN

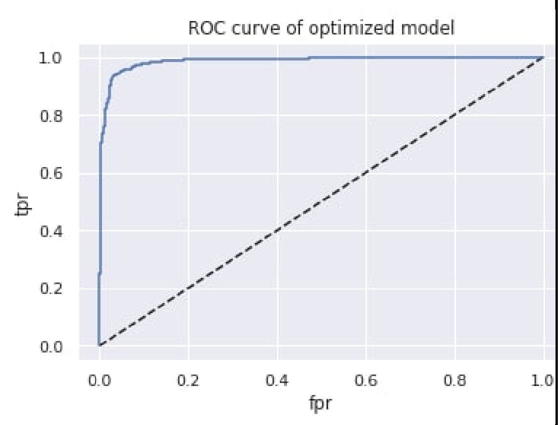
```
def build_classifier(learn_rate, batch_size):
  ANN_opt_model = Sequential()
  ANN_opt_model.add(Dense(10, input_dim= 48, activation= 'relu'))
  ANN_opt_model.add(Dense(10, activation= 'relu'))
  ANN_opt_model.add(Dense(10, activation= 'sigmoid'))
  ANN_opt_model.add(Dense(1, activation='sigmoid'))

  opt = adam(learning_rate=learn_rate)

  ANN_opt_model.compile(loss='binary_crossentropy', optimizer= opt,
metrics=[tf.keras.metrics.Recall()])
  return ANN_opt_model
```

ANN confusion matrices and ROC curves

 *Confusion matrix of ANN without PCA*	 *ROC Curve of ANN without PCA*
 *Confusion matrix of ANN with PCA*	 *ROC Curve of ANN with PCA*

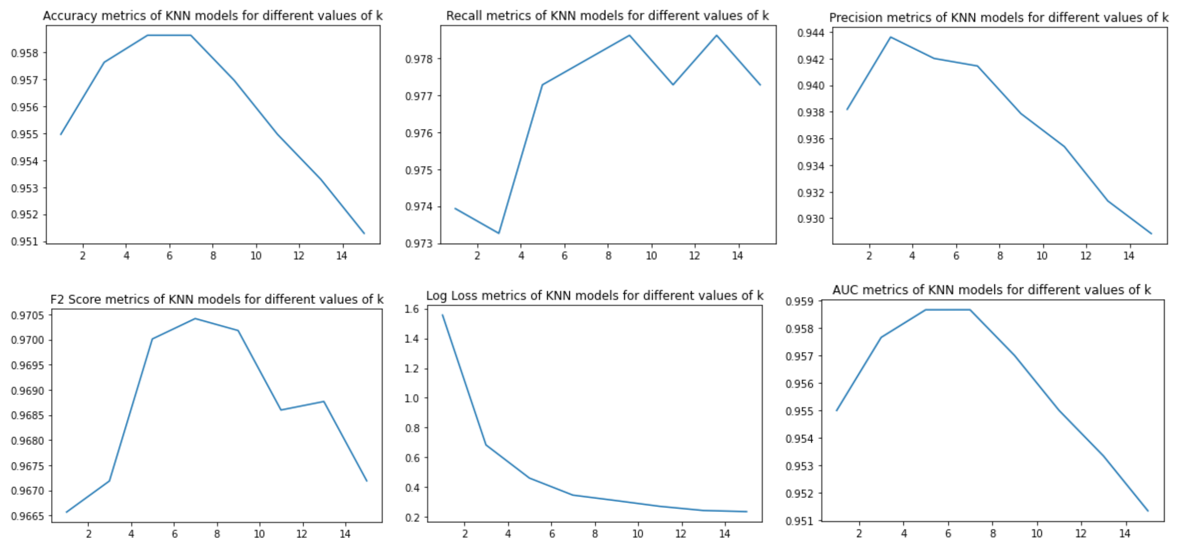


Confusion matrix of ANN GridSearchCV with PCA

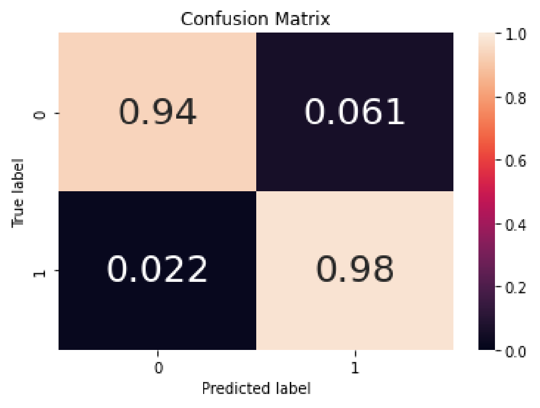


ROC Curve of ANN GridSearchCV with PCA

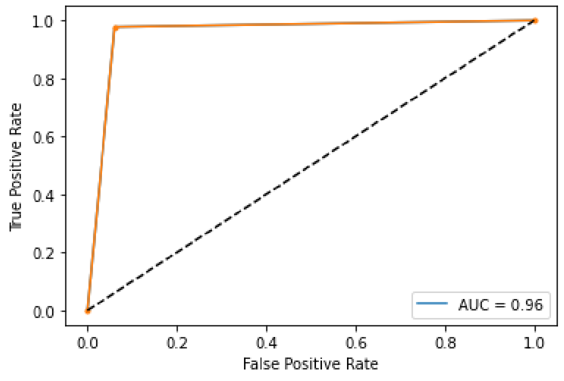
KNN metrics values for different values of k



KNN confusion matrix and ROC curve



Confusion matrix of KNN GridSearchCV without PCA



ROC Curve of KNN GridSearchCV without PCA