National College of Ireland

Configuration Manual for Using Genetic Algorithms for Optimized Feature Selection in Machine Learning and Deep Learning Models to Detect Phishing Websites

MSc Research Project
Msc Cyber Security

Akinola David Omotola
Student ID: X22133755

School of Computing
National College of Ireland

Supervisor: Raza UI Mustafa

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name: ……..Akinola David Omotola ……

Student ID: …… x22133755…….……..……

Programme: …… Msc Cyber Security …… **Year:** ……2024……..

Module: ……..……

Lecturer: ……27/5/2024…….……

Submission Due Date: ……..……

Project Title: ……Using Genetic Algorithms for Optimized Feature Selection in Machine Learning and Deep Learning Models to Detect Phishing Websites ……

Word Count: ……5357…… **Page Count:** ……30……..……

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature: ……Akinola David Omotola……

Date: ……27/5/2024……

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	□
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	

Penalty Applied (if applicable):	

Using Genetic Algorithms for Optimized Feature Selection in Machine Learning and Deep Learning Models to Detect Phishing Websites

Akinola David Omotola
X22133755

1 Introduction

This report aims to produce a manual which is a comprehensive guideline for creating the code implementation setup of the research on "Using Genetic Algorithms for Optimized Feature Selection in Machine Learning and Deep Learning Models to Detect Phishing Websites". The research was carried out to investigate how well the application of genetic algorithm (GA) is to select optimal features to improve the efficiency of deep learning and supervised learning algorithms in classifying websites as legitimate websites or phishing websites analysing historical website datasets. In carrying out this experiment, the Phishing Websites Dataset (Ariyadasa, Fernando and Fernando, 2021) was analysed using four machine learning algorithms. The machine learning algorithms that were implemented included the Graph Neural Network (GNN) which deep learning technique and three well-known supervised learning techniques: Random Forest (RF), Support Vector Machine (SVM), and Gradient Boost Classifier (GBC). This research conducted two sets of experiments namely Experiment 1 which is the baseline model implementation and Experiment 2 which combines GA feature selection on the dataset with hyperparameter tuning of the implemented models. The models were evaluated using Accuracy, Area Under the Curve (AUC) and F1 Score evaluation metrics.

This report's remaining content is shown as follows: In Section 2, system specifications for the hardware and software components used in this code implementation are covered. In Section 3, software installation, setting up the Anaconda environment, and installing the Python libraries used in this code implementation are covered. Section 4 will include code implementation, model evaluation, and how well the models worked with each dataset's analysis. Section 5 will provide closing remarks, and Section 6 will include a list of references.

2 System Specification

2.1 Hardware Specification

The configuration for the PC hardware used in the code implementation for this research work is specified in the table below

Hardware	Configuration Value
Random Access Memory (RAM)	16 GB
Processor Type	Intel core i5 with 2.50 GHz processing speed
Read Only Memory (ROM)	1 TB SSD

Table 1. System Hardware Configuration Summary

2.2 System Software Requirement

The table below briefly summarises and describes all software required to carry out this coding implementation

Software	Description
Window 10 Operating System	The fundamental software that serves as the platform that other supplementary software needed for this implementation project work will be installed, providing a reliable and compatible environment for the project's development and implementation.
Anaconda V 2.5.2	Anaconda is a package manager for the Python programming language. It allows for managing multiple Python versions and creating isolated environments, facilitating the creation and management of distinct workspaces on a single machine.
Jupyter Notebook V 7.0.8	Jupyter Notebook is an interactive programming IDE that executes code snippets directly within your web browser.

Web Browser (Google Chrome, Microsoft Edge, or Mozilla Firefox)	Jupyter Notebook uses a web browser to display code segments within cells or blocks, as well as the corresponding output when those code blocks are executed.

Table 2. Software Requirement Summary

3 Software Installation and Python Libraries

3.1 Anaconda Installation Guide

This guide provides a step-by-step walkthrough for installing Anaconda Navigator on your PC. First, visit the Anaconda download page and obtain the installation media. Follow the instructions below to set up Anaconda on your system.

Step 1:

Locate the downloaded installation media and double-click it to start the installation process. When the Anaconda setup screen appears, click the "Next" button to proceed.

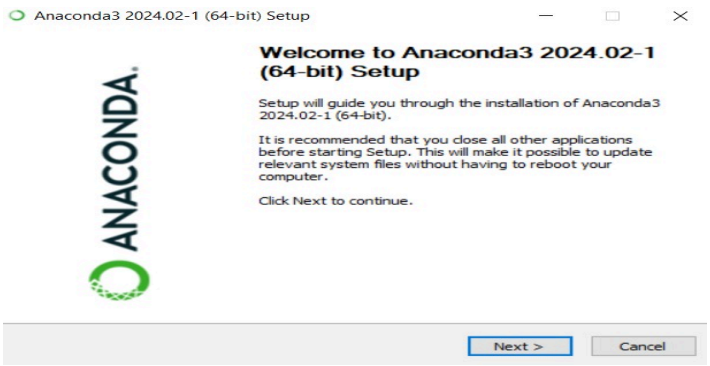

Figure 1. Anaconda installer welcome page

Step 2:

In the license agreement section, read through the terms, then click the "I Agree" button to proceed.

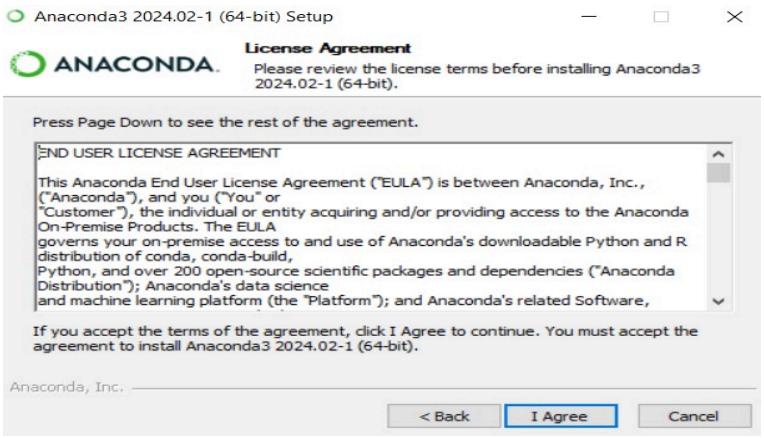

Figure 2. Anaconda installer license agreement page

Step 3:

On the "Select Installation Type" page, click the "Next" button to continue.

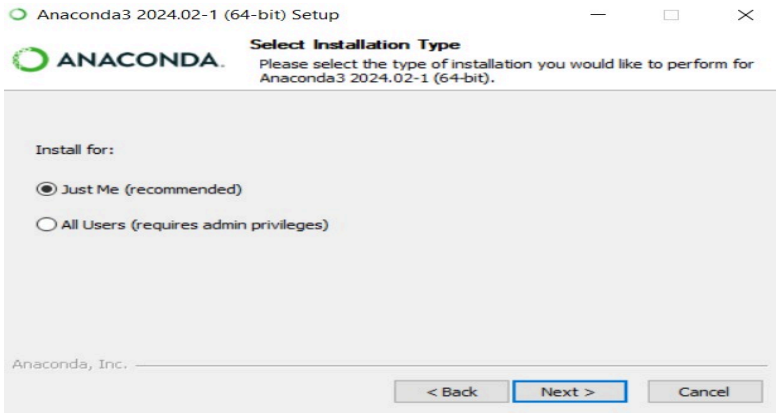

Figure 3. Anaconda installer select installation type page

Step 4:

In the "Choose Install Location" section, click the "Next" button to proceed.

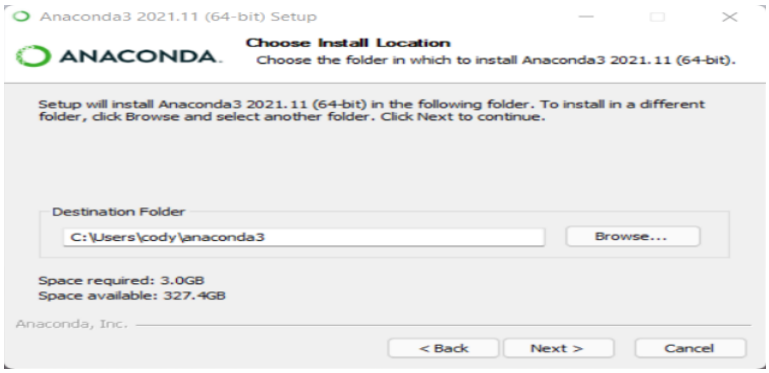

Figure 4. Anaconda installer chooses the install path page

Step 5:

On the "Advanced Installation Options" screen, click the "Install" button to begin writing the setup file to memory.

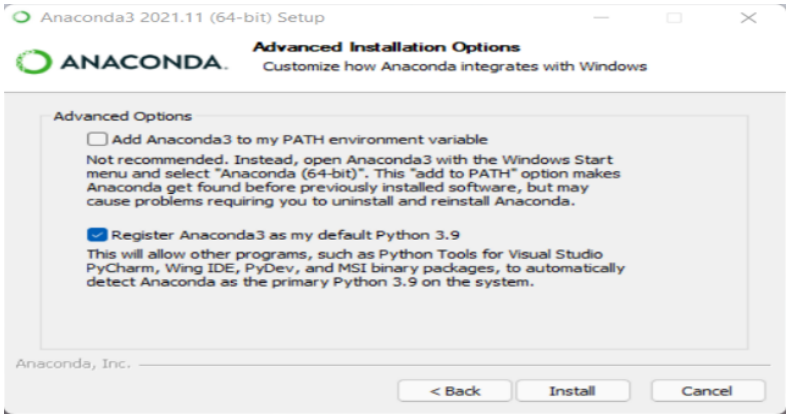

Figure 5. Anaconda installer advance installation options page

Step 6:

On the "Installation Complete" page, wait for the process to finish, then click the "Next" button.

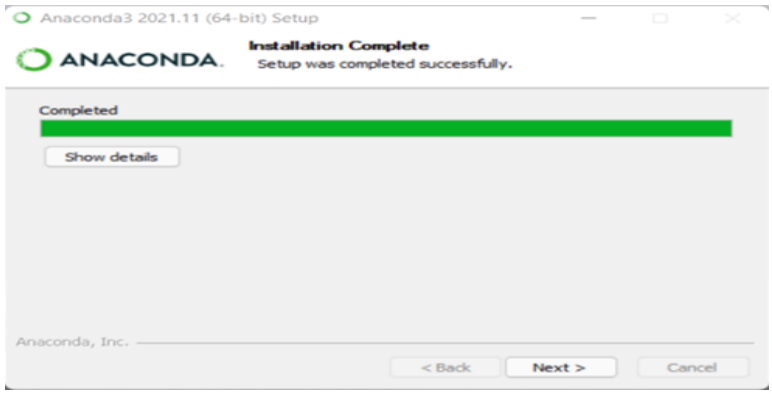

Figure 6. Anaconda installer installation complete page

Step 7:

On the "Thank You" page, click the "Finish" button to complete the Anaconda installation.

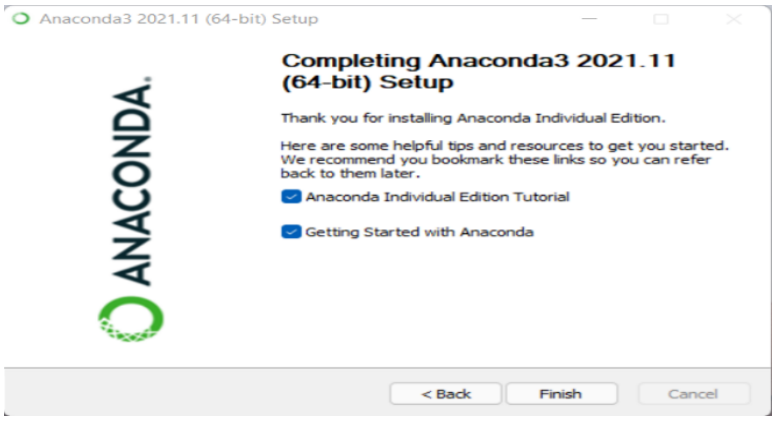

Figure 7. Anaconda installer completing setup page

3.2 Python Environment and Libraries

The Anaconda installation set up a default environment called "base" which comes with several pre-installed Python libraries essential for coding the project. Additional libraries will be installed to fully configure the environment for this coding task.

3.2.1 Python Libraries

Library Name	Installation Command
Pandas	pip install pandas
Numpy	pip install numpy
Statistics	pip install statistics
Seaborn	pip install seaborn
Scikit-learn	pip install scikit-learn
Imbalanced-learn	pip install imbalanced-learn
Tensorflow	pip install tensorflow
Scikeras	pip install scikeras
Deap	pip install deap

Table 3. Python Libraries and their installation command

3.2.2 How to install Python Libraries from Jupyter Notebook

To open Anaconda, click on the Start menu and find the Anaconda3 folder. Expand the folder, then double-click on Anaconda Navigator to launch the application. Once Anaconda Navigator is running, locate the Jupyter Notebook tile and click "Launch" to open the interactive Python IDE in your web browser, as illustrated below.

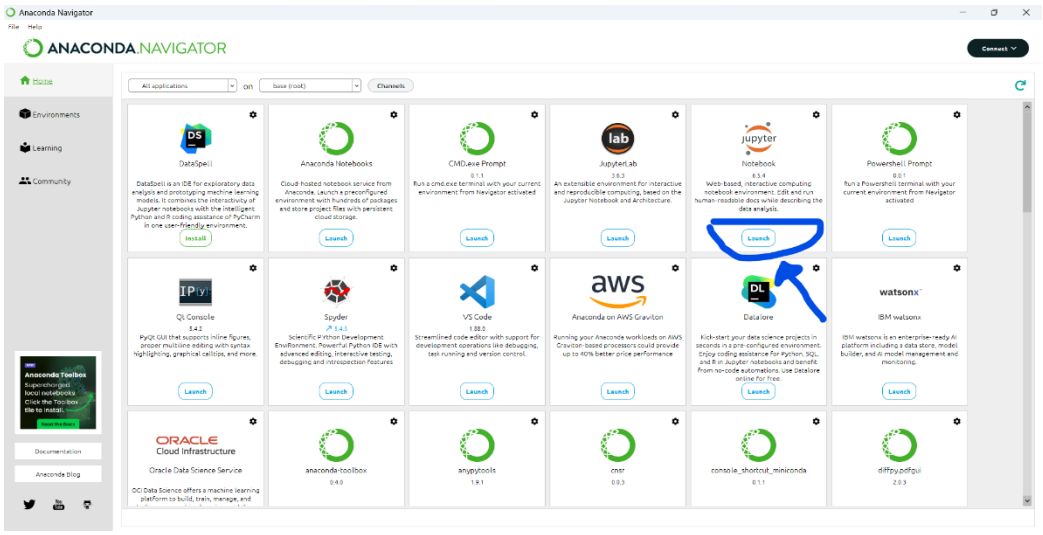

Figure 8. Anaconda navigator home page when launched

On the launch of Jupyter Notebook, all Python libraries for implementing this research work will be loaded in a cell and the code will be executed. If ModuleNotFoundError is encountered as shown in the figure x below, this suggests that the associated library has not been installed.

```
In [2]:
###
###       Python Libraries
###

import time
import pandas as pd
import numpy as np
import math
import random
import statistics
import seaborn as sns
import matplotlib.pyplot as plt
from IPython.core.display import HTML as Center

from deap import creator, base, tools, algorithms

from sklearn.model_selection import train_test_split, cross_val_score, GridSearchCV
from sklearn.preprocessing import MinMaxScaler

from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import GradientBoostingClassifier
from sklearn import svm
from sklearn.neural_network import MLPClassifier
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import classification_report, confusion_matrix, f1_score, roc_auc_score, accuracy_score

from imblearn.over_sampling import SMOTE

import tensorflow as tf
from tensorflow.keras import layers
from tensorflow.keras.optimizers import *
from scikeras.wrappers import KerasClassifier

###
### Custom Libraries for Feature Extraction
###
from features_extraction import run_features_from_datasets, data_dir, extracted_features_csv
from features_to_extract import get_column_names
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[2], line 15
     12 import matplotlib.pyplot as plt
     13 from IPython.core.display import HTML as Center
---> 15 from deap import creator, base, tools, algorithms
     17 from sklearn.model_selection import train_test_split, cross_val_score, GridSearchCV
     18 from sklearn.preprocessing import MinMaxScaler

ModuleNotFoundError: No module named 'deap'
```

Figure 9. Missing Python library exception thrown page

To install the missing library, copy the installation command for the missing library into an empty cell or code block with exclamation (!) prefixing the installation command. Run the code block with the installation and wait for the completion the the library being installed as shown in Figure 10 below.

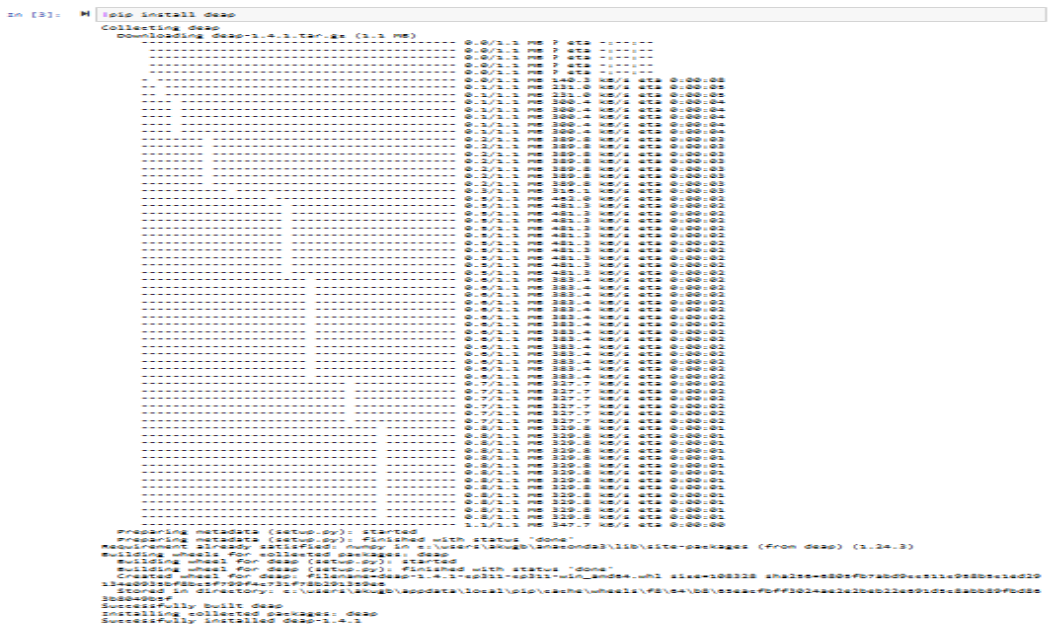

Figure 10. Successfully installed Python using pip command on Jupyter Notebook

Run the code block with the Python libraries again and check if the ModuleNotFoundError exception occurs. Repeat the process as above for the missing library if the error occurs else all Python libraries required for this model implementation are completed.

4 Project Implementation and Evaluation

After you have successfully installed Anaconda and set up all the required libraries, this section demonstrates the code implementation for this research project. It provides a step-by-step breakdown of the code, including explanations of key functions and processes. Let's dive into the code to understand how it addresses the research questions and achieves the project's objectives.

4.1 Import Python Libraries

In the initial code block, we'll include all the necessary Python libraries required to run this project. If you add or modify any library imports in this block, be sure to click the "Run" button to ensure these changes take effect within the integrated development environment (IDE). This step is crucial for making sure your project has access to all the correct packages and dependencies for successful execution.

```
[1]: import warnings
     import os

     def fxn():
         warnings.warn("deprecated", DeprecationWarning)

     warnings.simplefilter("ignore")
     fxn()

     os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'

[2]: ###
     ###      Python Libraries
     ###

     import time
     import pandas as pd
     import numpy as np
     import math
     import random
     import statistics
     import seaborn as sns
     import matplotlib.pyplot as plt
     from IPython.core.display import HTML as Center

     from sklearn.model_selection import train_test_split, cross_val_score, GridSearchCV
     from sklearn.preprocessing import MinMaxScaler

     from sklearn.ensemble import RandomForestClassifier
     from sklearn.ensemble import GradientBoostingClassifier
     from sklearn import svm
     from sklearn.neural_network import MLPClassifier
     from sklearn.preprocessing import LabelEncoder
     from sklearn.metrics import classification_report, confusion_matrix, f1_score, roc_auc_score, accuracy_score

     from imblearn.over_sampling import SMOTE

     import tensorflow as tf
     from tensorflow.keras import layers
     from tensorflow.keras.optimizers import *
     from scikeras.wrappers import KerasClassifier

     from deap import creator, base, tools, algorithms

     ###
     ### Custom Libraries for Feature Extraction
     ###
     from features_extraction import run_features_from_datasets, data_dir, extracted_features_csv
     from features_to_extract import get_column_names
```

Figure 11. This code snippet shows the importing of all Python libraries into the IDE

4.2 Global Variables Definition

The second code cell is specifically designed for declaring global variables that will play a central role in tracking and storing the output produced by various code segments as they execute. This setup allows us to maintain a consistent state across different parts of the codebase, facilitating smooth data flow and information sharing. Global variables defined in this cell will be accessible throughout the entire project, providing a centralized repository for storing key outputs, intermediate results, and other critical data. Figure y below provides a comprehensive list of all the global variables utilized throughout this research project. These variables serve as essential building blocks for integrating and coordinating the diverse elements of the codebase.

```
[3]: %matplotlib inline
plt.style.use("ggplot")

Center("""<style>
    .output_png {
        display: table-cell;
        text-align: center;
        vertical-align: middle;
    }
</style>""")

target_variable  =  "label"

train_x          =  "train_x"
train_y          =  "train_y"
test_x           =  "test_x"
test_y           =  "test_y"

gbc_model_name = {
    "name": "Gradient Boost Classifier",
    "short": "GBC",
    "analysis": "Gradient Boost Classification Analysis"
}

gnn_model_name = {
    "name": "Graph Neural Network",
    "short": "GNN",
    "analysis": "Graph Neural Network Classification Analysis"
}

svm_model_name = {
    "name": "Support Vector Machine",
    "short": "SVM",
    "analysis": "Support Vector Machine Classification Analysis"
}

rfc_model_name = {
    "name": "Random Forest Classifier",
    "short": "RFC",
    "analysis": "Random Forest Classification Analysis"
}

acc_score_label  =  "ACCURACY_SCORE"
f1_score_label   =  "F1_SCORE"
auc_score_label  =  "AUC_SCORE"
```

Figure 12. This code snippet shows the code block or cell used global variable definition code

4.3 Reuse Function Definition

In the course of implementing the code for this research project on phishing analysis, two distinct experiments were carried out. Given this, task repetition was inevitable, making code reuse a crucial approach to reduce redundant coding. To facilitate code reuse and minimize the overall volume of code, a set of functions was defined to encapsulate commonly repeated operations. This modular approach not only streamlined the coding process but also improved maintainability and readability. By leveraging these functions, the project team could focus more on analysis and experimentation while ensuring consistency and reducing potential errors due to repetitive coding.

4.3.1 Data Cleaning, Preprocessing and Visualization Function

4.3.1.1 Remove missing data from the analysis dataset function

This function was used to remove missing data from the dataset by checking for records NaN, positive or negative infinity values. If positive or negative infinity values were found in the dataset, these values were converted into NaN using the Numpy library. All records with NaN data value are removed from the dataset by calling the ***dropna*** function on the Pandas' data frame as shown in Figure 13 below.

```
[4]: def remove_missing_or_nan_from_dataset(analysis_dataset):
         analysis_dataset.dropna(inplace=True)
         analysis_dataset = analysis_dataset.replace([np.inf, -np.inf], np.nan)
         analysis_dataset.dropna(inplace=True)
         return analysis_dataset
```

Figure 13. This code snippet shows the function implementation used for handling missing data or NaN in the dataset

4.3.1.2 Get data frame column names with unique one unique value

For a variable to be useful in the classification of phishing websites, it must contain values to help distinguish between the two classes "legitimate" or "phishing" websites. Having one unique value does help in this classification hence, this function is used to retrieve columns for removal from the dataset.

```
[5]: def get_column_names_that_has_one_unique_valud(analysis_dataset):
         column_names = list()
         for col_name in analysis_dataset.columns:
             unique_values = analysis_dataset[col_name].nunique()
             if unique_values <= 1:
                 column_names.append(col_name)
         return column_names
```

Figure 14. This code snippet shows the function implementation used to remove columns with help in labelling the dataset i.e. data columns with unique data

4.3.1.3 Generate a Pie Chart Diagram

This function is used to generate a Pie chart diagram for showing the percentage distribution of the binary class in the target variable in the dataset. This function accepts 4 arguments namely ***analysis_dataset*** (data frame containing data records)***, category_label*** (a list containing a label for identifying binary)***, main_title*** (a descriptive title for the plot to be generated) and ***target_variable*** *(*the column name used to retrieve the series used to generate the pie chart*)*.

```
[6]: def plot_pie_chart_for_binary_distribution_visualization(analysis_dataset, category_label, main_title, target_variable):
         colors = {1: 'r', 0: 'gray'}
         count = analysis_dataset[target_variable].value_counts().to_frame().sort_index()

         fig = plt.figure(figsize=(5, 5), dpi=144)
         phishing_classification = [analysis_dataset[target_variable].value_counts()[0], analysis_dataset[target_variable].value_counts()[1]]
         plt.title(main_title)
         plt.pie(phishing_classification, labels=category_label, autopct='%1.1f%%', colors=[colors[c] for c in count.index])
         plt.axis('equal')
         plt.legend(loc = 0)
         plt.show()
```

Figure 15. This code snippet shows the function implementation used to plot binary class distribution plots for phishing and legitimate website count

4.3.1.4 The function is used to normalize, split the dataset into training and testing datasets, and resample the dataset to eliminate class imbalance in the dataset.

To prepare the dataset for analysis, the function defined by the code block in Figure 16 is used to scale the dataset such that data records are scaled evenly (normalized). The normalized dataset is then divided into a training dataset and a testing dataset using the ***train_test_split*** function from the sci-kit learn module. The SMOTE module from the imbalanced learn library was used to eliminate the class imbalance in the training dataset.

```
[7]: def normalize_resample_and_split_dataset(predictor_variables, target_variable):
         scaler = MinMaxScaler()
         resample_with_smote = SMOTE()

         predictor_variables = scaler.fit_transform(predictor_variables)
         pred_vars_train, pred_vars_test, tar_var_train, tar_var_test = train_test_split(predictor_variables, target_variable, test_size=0.2,

         pred_vars_train, tar_var_train = resample_with_smote.fit_resample(pred_vars_train, tar_var_train)

         return pred_vars_train, pred_vars_test, tar_var_train, tar_var_test
```

Figure 16. This code snippet shows the function implementation used to normalize the dataset, resample the dataset with the SMOTE technique and split the dataset into training and testing dataset

4.3.1.5 Process analysis dataset function

This function code snippet defined in Figure 17 invokes the ***normalize_resample_and_split_dataset*** function above to generate the processed analysis dataset and store the dataset in Python dictionary data structure which will be used to run model analysis for each of the implemented ML models.

```
[8]: def process_analysis_dataset(analysis_independent_variables, analysis_dependent_variable):
         processed_dataset = dict()

         pred_vars_train, pred_vars_test, tar_var_train, tar_var_test = normalize_resample_and_split_dataset(
             analysis_independent_variables, analysis_dependent_variable
         )

         processed_dataset[train_x] = pred_vars_train
         processed_dataset[train_y] = tar_var_train

         processed_dataset[test_x] = pred_vars_test
         processed_dataset[test_y] = tar_var_test

         return processed_dataset
```

Figure 17. This code snippet shows the function implementation used to prepare the dataset for analysis by calling the normalize_resample_split_dataset function

4.3.2 Model Building and Evaluation Summary Function

4.3.2.1 Function for building GNN and hyper-parameter tuned GNN models

Neural networks are assembled using a build function to the input layer parameters and output layer parameters for both the baseline and hyper-parameter tuned model's implementation as shown in the code snippet in figure 18. For the baseline model, 2 layers of the graph convolutional layers (gc1 and gc2) were defined to construct the input layer. 64 neurons were used to initialize the parameter gc1 layer and 32 neurons were used to initialize the parameter gc2 layer. gc2 layer applies a second layer to the output of the first layer (gc1) which transforms features after passing through the first layer. The hidden layer of the GNN had 32 neurons which is applied to the input layer along with an activation function and the output layer has one neuron and a sigmoid activation function because it is a binary classifier. Both the input layer and the hidden layer activation function ***ReLU*** (Rectified Linear Unit). The ***ReLU*** activation function outputs the input directly if it is positive; otherwise, it outputs zero. This introduces non-linearity into the model, which helps it learn more complex patterns. Once the input layer, the hidden layer and the output layer are defined, the Keras model function is used to create a GNN model with the defined input and output layer parameters. The compile method configures the model for training by setting the optimizer to "***adam***" (optimization algorithm that adjusts the learning rate based on the first and second moments of the gradients), loss function to "***binary_crossentropy***" (loss function is used for binary classification tasks. It measures the difference between the predicted probabilities and the actual binary labels), and metrics to "***accuracy***" (which calculates the proportion of correctly classified instances out of the total instances) be tracked.

For the hyperparameter tunned model, a list of integer values for the gc1, gc2, hidden layer and the reshape parameters. These parameters will be used to tune the GNN model and select the optimal parameter.

```
[8]: def build_gnn_model(input_shape):
         inputs = layers.Input(shape=input_shape)
         gc1 = layers.Dense(64, activation='relu')(inputs)
         gc2 = layers.Dense(32, activation='relu')(gc1)
         reshaped = layers.Reshape((-1, 32))(gc2)
         global_pooling = layers.GlobalAveragePooling1D()(reshaped)
         dense1 = layers.Dense(32, activation='relu')(global_pooling)
         outputs = layers.Dense(1, activation='sigmoid')(dense1)
         model = tf.keras.Model(inputs=inputs, outputs=outputs)
         model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
         return model

[9]: def build_hyperparameter_gnn_model(input_shape, units_gc1=64, units_gc2=32, units_reshape=32, units_dense1=32):
         inputs = layers.Input(shape=input_shape)
         gc1 = layers.Dense(units=units_gc1, activation='relu')(inputs)
         gc2 = layers.Dense(units=units_gc2, activation='relu')(gc1)
         reshaped = layers.Reshape((-1, units_reshape))(gc2)
         global_pooling = layers.GlobalAveragePooling1D()(reshaped)
         dense1 = layers.Dense(units=units_dense1, activation='relu')(global_pooling)
         outputs = layers.Dense(1, activation='sigmoid')(dense1)
         model = tf.keras.Model(inputs=inputs, outputs=outputs)
         model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
         return model
```

Figure 18. This code snippet shows the implementation of the function used to create the build function for the baseline GNN and the hyper-parameter tuned GNN model

4.3.2.2 Instantiate and run random classifier (RFC) forest model

The code block labelled 11 in Figure 19 is used to create an instance of the random forest classifier (RFC) model. The function makes it easy to create new instances of the RFC model repeatedly with the same set of instructions which is easily maintained and managed.

The code block labelled 12 in Figure 19 is used to run the baseline and hyperparameter-tuned RFC model depending on the value of the ***grid_params*** argument. If the ***grid_params*** value is provided and set to None, the baseline RFC model will be created, but if the ***grid_params*** contains a list of hyperparameters for tuning the RFC model, the hyperparameter-tuned model RFC will be created.

```
[11]: def instantiate_rfc_model():
          return RandomForestClassifier(random_state=123)

[12]: def run_rfc_model(processed_analysis_dataset, analysis_result, model_name, grid_params=None):
          model = instantiate_rfc_model()
          if grid_params:
              model = GridSearchCV(estimator=model, param_grid=grid_params, cv=5, scoring='accuracy', n_jobs=-1)
          model_summary = run_model_analysis(model, model_name, processed_analysis_dataset)
          analysis_result[model_name["short"]] = model_summary
```

Figure 19. This code snippet shows the implementation of the function used to build and run the baseline RFC model and the hyper-parameter tuned RFC model

4.3.2.3 Instantiate and run support vector machine (SVM) model

The code block labelled 13 in Figure 20 is used to create an instance of the Support Vector Machine (SVM) classifier model. The function makes it easy to create new instances of the SVM model repeatedly with the same set of instructions which is easily maintained and managed.

The code block labelled 14 in Figure 20 is used to run the baseline and hyperparameter-tuned SVM model depending on the value of the ***grid_params*** argument. If the ***grid_params*** value is provided and set to None, the baseline SVM model will be created, but if the ***grid_params*** contains a list of hyperparameters for tuning the SVM model, the hyperparameter-tuned model SVM will be created.

```
[13]: def instantiate_svm_model():
          return svm.SVC()

[14]: def run_svm_model(processed_analysis_dataset, analysis_result, model_name, grid_params=None):
          model = instantiate_svm_model()
          if grid_params:
              model = GridSearchCV(estimator=model, param_grid=grid_params, cv=5, scoring='accuracy', n_jobs=-1)
          model_summary = run_model_analysis(model, model_name, processed_analysis_dataset)
          analysis_result[model_name["short"]] = model_summary
```

Figure 20. This code snippet shows the implementation of the function used to build and run the baseline SVM model and the hyper-parameter tuned SVM model

4.3.2.4 Instantiate and run the Gradient Boost Classifier (GBC) model

The code block labelled 15 in Figure 21 is used to create an instance of the Gradient Boost Classifier (GBC) classifier model. The function makes it easy to create new instances of the GBC model repeatedly with the same set of instructions which is easily maintained and managed.

The code block labelled 16 in Figure 21 is used to run the baseline and hyperparameter-tuned GBC model depending on the value of the ***grid_params*** argument. If the ***grid_params*** value is provided and set to None, the baseline GBC model will be created, but if the ***grid_params*** contains a list of hyperparameters for tuning the GBC model, the hyperparameter-tuned model GBC will be created.

```
[15]: def instantiate_gbc_model():
          return GradientBoostingClassifier(random_state=123)

[16]: def run_gbc_model(processed_analysis_dataset, analysis_result, model_name, grid_params=None):
          model = instantiate_gbc_model()
          if grid_params:
              model = GridSearchCV(estimator=model, param_grid=grid_params, cv=5, scoring='accuracy', n_jobs=-1)
          model_summary = run_model_analysis(model, model_name, processed_analysis_dataset)
          analysis_result[model_name["short"]] = model_summary
```

Figure 21. This code snippet shows the implementation of the function used to build and run the baseline GBC model and the hyper-parameter tuned GBC model

4.3.2.5 Instantiate and run Graphical Neural Network (GNN) model

The code block labelled 17 in Figure 22 is used to create an instance of the Graphical Neural Network (GNN) classifier model. The function makes it easy to create new instances of the GNN model repeatedly with the same set of instructions which is easily maintained and managed.

The code block labelled 18 in Figure 22 is used to run the baseline and hyperparameter-tuned GNN model depending on the value of the ***grid_params*** argument. If the ***grid_params*** value is provided and set to None, the baseline GNN model will be created, but if the ***grid_params*** contains a list of hyperparameters for tuning the GNN model, the hyperparameter-tuned model GNN will be created.

```
[17]: def instantiate_gnn_model(input_shape, grid_params=None):
          if grid_params == None:
              return KerasClassifier(lambda: build_gnn_model(input_shape), epochs=10, batch_size=32)
          else:
              return KerasClassifier(model=build_hyperparameter_gnn_model, input_shape=grid_params["input_shape"], units_gc1=grid_params[

[18]: def run_gnn_model(processed_analysis_dataset, analysis_result, model_name, grid_params=None):
          input_shape = processed_analysis_dataset[train_x].shape[1]
          model = instantiate_gnn_model(input_shape, None)
          if grid_params:
              grid_params["input_shape"] = [input_shape]
              gnn = instantiate_gnn_model(input_shape, grid_params)
              model = GridSearchCV(estimator=gnn, param_grid=grid_params, cv=5, scoring='accuracy', n_jobs=-1)
          model_summary = run_model_analysis(model, model_name, processed_analysis_dataset)
          analysis_result[model_name["short"]] = model_summary
```

Figure 22. This code snippet shows the implementation of the function used to build and run the baseline GNN model and the hyper-parameter tuned GNN model

4.3.2.6 The function used to perform model analysis for all implemented models

The function whose code snippet is shown in Figure 23 below is used to perform model analysis for the created ML model being the baseline model or the hyperparameter-tuned model. The function accepts three arguments namely – ***model*** (the model instance to be analysed),

model_name (the name used to identify the model being analysed) and ***processed_analysis_dataset*** (contains the processed dataset used for model analysis). The function invokes four other functions to complete the analysis for all the implemented models.

```
[20]: def run_model_analysis(model, model_name, processed_analysis_dataset):

          # Train model
          model = train_model(model, processed_analysis_dataset, model_name)

          # Generate model predictions
          predictions = generate_model_prediction(model, processed_analysis_dataset, model_name)

          # Generate model evaluation summary
          result = generate_summary_result(processed_analysis_dataset[test_y], predictions)

          # Display statistical summary and visualization for the implementation
          generate_analysis_summary(processed_analysis_dataset[test_y], predictions, result, model_name)

          return result
```

Figure 23. This code snippet shows the function implementation used to assemble and run model analysis for all implemented models

4.3.2.7 Train ML model function

The function in the code block labelled 21 whose code snippet is shown in figure 24 below is used to perform model training on the training dataset and track the amount of time in seconds used in the training model using the training dataset.

The function in the code block labelled 22 whose code snippet is shown in Figure 24 below is used to perform model prediction after successfully training the model being analysed. The function also tracks the time in seconds used in making predictions using the testing dataset.

```
[21]: def train_model(model, data, model_name):
          start = time.time()
          model.fit(data[train_x], data[train_y])
          end = time.time()
          print("\n\n")
          print("{} MODEL TRAINING TIME IS {} SECONDS".format(model_name["name"].upper(), end - start))
          print("\n\n")
          return model
```

```
[22]: def generate_model_prediction(model, data, model_name):
          start = time.time()
          predictions = model.predict(data[test_x])
          end = time.time()
          print("{} MODEL GENERATED MODEL PREDICTIONS IN {} SECONDS".format(model_name["name"].upper(), end - start))
          print("\n\n")
          return predictions
```

Figure 24. This code snippet shows the functions implementation used to train and generate predictions for the model being analyzed

4.3.2.8 Generate summary analysis summary function

The function in the code block labelled 23 whose code snippet is shown in Figure 25 is used to generate summary analysis from the generated prediction of the previous function in code block labelled 22 and the actual values in test y variables. The function accepted four variables namely – ***test_y*** (the actual labels assigned to the testing dataset), ***predictions*** (the generated labels from the predictions made for the ***testing_x*** predictors), ***result*** (the dictionary data structure used in tracking evaluation metrics for the implemented models), ***model_name*** (the name used to identify the model being analysed). The function also invokes three other functions to complete the summary analysis to be completed.

The function in the code block labelled 24 whose code snippet is shown in Figure 25 is used classification report summary for the implemented model. The classification summary report provides a detailed overview of a classification model's performance, breaking it down into precision, recall, F1-score, and support for each class, as well as overall averages.

The function in the code block labelled 25 whose code snippet is shown in Figure 25 is used to generate the confusion matrix diagram. The confusion matrix plot is used to provide visual detailed insights into how predictions match the actual labels, enabling evaluation of the efficacy of the implemented model.

The function in the code block labelled 26 whose code snippet is shown in Figure 25 is used to display the estimated score for the implemented evaluation metrics namely – Accuracy score, AUC score and F1-Score.

```
[23]: def generate_analysis_summary(test_y, predictions, result, model_name):
          # Show classification summary
          show_classification_report_summary(test_y, predictions, model_name)

          # Plot confusion matrix
          plot_confusion_matrix(test_y, predictions, model_name)

          # Show model evaluation summary
          show_model_evaluation_summary(result, model_name)

[24]: def show_classification_report_summary(test_y, predictions, model_name):
          print("CLASSIFICATION SUMMARY FOR {} ({})".format(model_name["name"].upper(), model_name["short"].upper()))
          print("\n\n")
          class_labels = ["LEGITIMATE WEBSITE", "PHISHING WEBSITE"]
          print(classification_report(predictions, test_y, target_names=class_labels))
          print("\n\n")

[25]: def plot_confusion_matrix(test_y, predictions, model_name):
          print("\n\n")
          confusion_matrix_data = confusion_matrix(test_y, predictions)
          plt.figure(figsize=(5,5))
          title = "CONFUSION MATRIX PLOT FOR {} ({}) MODEL".format(model_name["name"].upper(), model_name["short"].upper())
          ax_plot = sns.heatmap(confusion_matrix_data, annot=True)
          ax_plot.set_title(title)
          ax_plot.set_xlabel("ORIGINAL RESULT")
          ax_plot.set_ylabel("PREDICTION RESULT")
          ax_plot.xaxis.set_ticklabels(["FALSE", "TRUE"])
          ax_plot.yaxis.set_ticklabels(["FALSE", "TRUE"])
          plt.show()
          print("\n\n")

[26]: def show_model_evaluation_summary(result, model_name):
          print("EVALUATION SUMMARY FOR {} ({})".format(model_name["name"].upper(), model_name["short"].upper()))
          print("\n\n")
          print(set_evaluation_result(acc_score_label, result[acc_score_label]))
          print(set_evaluation_result(f1_score_label, result[f1_score_label]))
          print(set_evaluation_result(auc_score_label, result[auc_score_label]))
          print("\n\n\n\n\n")
```

Figure 25. This code snippet shows the implementation of the function used to generate and display the model evaluation summary and confusion matrix plot.

4.3.2.9 Display summary table for implementing model's function

The function in the code block labelled 30 whose code snippet is shown in Figure 26 is used to draw a summary table for the models in the baseline and hyperparameter-tuned implementation. The function accepts three arguments namely – ***result*** (contains estimated values for Accuracy, AUC and F1-Score for the implemented models), ***record_desc*** (contains short names for the implemented models), and ***metrics*** (contains names for evaluation metrics used in evaluating the models). These arguments hold all the data used in rendering the summary table for the models in the baseline and hyperparameter-tuned implementations.

```
[30]: def show_summary_model_evaluation_table(result, record_desc, metrics):
          table = ""
          max_table_width = 100
          num_column = len(metrics) + 1
          content_width = int(max_table_width / num_column)
          if max_table_width > (content_width * num_column):
              max_table_width = (content_width * num_column)
          next_line = "\n"

          horizontal_line = ""
          for count in range(max_table_width):
              horizontal_line += "="

          horizontal_line += next_line

          table += horizontal_line
          pos = 1

          table += draw_table_content("", pos, num_column, content_width)
          for idx in range(len(metrics)):
              pos += 1
              table += draw_table_content(" ".join(metrics[idx].split("_")), pos, num_column, content_width)

          table += horizontal_line

          for label in record_desc:
              pos = 1
              table += draw_table_content(label, pos, num_column, content_width)
              for idx in range(len(metrics)):
                  pos += 1
                  table += draw_table_content(set_table_estimate(result[label][metrics[idx]]), pos, num_column, content_width)

              table += horizontal_line

          print(table)
          print()
```

Figure 26. This code snippet shows the function implementation used to generate an evaluation summary table for all implemented models for experiment 1 and experiment 2.

The function in the code block labelled 31 whose code snippet is shown in Figure 27 is used to draw a table of cell data enforcing consistent width for all cells drawn in the summary table. The function accepts four arguments namely – ***text*** (the text data to be rendered in the cell to be drawn), position (the column position of the cell to be drawn), num_of_columns (the total number of columns on the table), column_width (the size for the cell to be drawn).

The function in the code block labelled 32 whose code snippet is shown in Figure 27 is used to horizontal bar across the table being drawn or across the summary presentation area.

[31]:
```
def draw_table_content(text, position, num_of_columns, column_width):
    content = "| "
    try:
        padding = column_width - len(text)
        content += text
    except:
        padding = column_width
        content += ""

    for num in range((padding-3)):
        content += " "

    if position == num_of_columns:
        content += "|\n"
    else:
        content += " "

    return content
```

[32]:
```
def draw_line(width):
    line = ""
    for i in range(width):
        line += "="
    return line
```

Figure 27. This code snippet shows functions that are used to set table cell records and draw horizontal bars across a table respectively.

4.3.3 Genetic Algorithm Implementation Function

4.3.3.1 The function used to couple the genetic algorithm (GA) and run the analysis

The function in the code block labelled 33 whose code snippet is shown in figure 28 is used to initialize the parameters needed to run the genetic algorithm. These parameters are then fed into the defined ***genetic_algorithm*** function. The output from the genetic algorithm (GA) function is then used to retrieve the best-performing individual from the GA analysis.

The function in the code block labelled 34 whose code snippet is shown in Figure 28 is used to retrieve the best individual from GA hall of fame output. The function loops through the hall of fame variable and selects the individual with the high fitness value. The function returns the column name for the select features, the estimated fitness value and the individual genome.

```
[33]: def run_genetic_algorithm_model_base_solution(analysis_dataset, model, is_func=None):
          label_encoder = LabelEncoder()
          label_encoder.fit(analysis_dataset.iloc[:, -1])
          target = label_encoder.transform(analysis_dataset.iloc[:, -1])
          predictors = analysis_dataset.iloc[:, :-1]

          individual = [1 for i in range(len(predictors.columns))]
          print(get_model_fitness(individual, predictors, target, model, is_func))
          hall_of_fame = genetic_algorithm(predictors, target, model, is_func, predictors.shape[1], 10)
          return generate_best_individual_summary(hall_of_fame, predictors, target)
```

```
[34]: def generate_best_individual_summary(hall_of_fame, predictors, target):
          max_accuracy = 0.0
          for individual in hall_of_fame:
              if individual.fitness.values[0] > max_accuracy:
                  max_accuracy = individual.fitness.values[0]
                  _individual = individual

          _individual_header = [list(predictors)[i] for i in range(len(_individual)) if _individual[i] == 1]
          return _individual.fitness.values, _individual, _individual_header
```

Figure 28. This code snippet shows function used to assemble the implemented genetic algorithms (GA), while function 34 is used to select the best individual in the population

4.3.3.2 Genetic Algorithm (GA) implementation function

The function in the code block labelled 35 whose code snippet is shown in Figure 29 is the GA implementation function designed to optimize a predictive model by selecting the best subset of predictors (features).

First, the ***creator*** module from deap library is used to define the representation of a chromosome or genome or individual in the population. Secondly, the instance of the ***Toolbox (toolbox)*** is created from the ***base*** module and the created instance of ***toolbox*** to register and configure population parameters used for the GA analysis. Next, the toolbox registers the fitness function used to evaluate the fitness of an individual using the provided model and fitness function. Also, the toolbox instance registers other GA operations namely – ***mate*** (also known as the Crossover operation with the attribute "cxTwoPoint" is used to swap segments between two parents to create new offspring), ***mutate*** (also known as the Mutation operation with the attribute "mutFlipBit" is used to flip bit in the chromosome with a probability of 0.05) and ***select*** (also known as the Selection operation with the attribute "selTournament" is used to select the best individuals out of a randomly chosen subset of the population) operations respectively. To execute the GA, the initial population is created by calling the toolbox population method, and the evolutionary process is initiated by calling ***eaMuPlusLambda*** module in ***algorithms*** module. The ***eaMuPlusLambda*** algorithm is an evolutionary strategy

that generates ***lambda*** (the number of offspring which is equal to twice the population size) offspring from ***mu*** (the number of parents which is equal to the population size) parents.
After each evolutionary process, the best-performing individual is best individuals (those with the highest fitness scores) from the current generation are preserved and directly passed on to the next generation without undergoing crossover or mutation. This ensures that the most optimal solutions found so far are not lost due to the stochastic nature of the genetic operations. When the GA operation has been completed, the hall of fame which is used to store the best individuals found during the evolution process is returned for further processing.

```
[35]: def genetic_algorithm(predictors, target, model, is_func, num_of_population, num_of_generation):
          # Define chromosome representation
          creator.create("FitnessMax", base.Fitness, weights=(1.0,))
          creator.create("Individual", list, fitness=creator.FitnessMax)

          toolbox = base.Toolbox()
          toolbox.register("attr_bool", random.randint, 0, 1)
          toolbox.register("individual", tools.initRepeat, creator.Individual, toolbox.attr_bool, n=len(predictors.columns))
          toolbox.register("population", tools.initRepeat, list, toolbox.individual)
          toolbox.register("evaluate", get_model_fitness, predictors=predictors, target=target, model=model, is_func=is_func)
          toolbox.register("mate", tools.cxTwoPoint)
          toolbox.register("mutate", tools.mutFlipBit, indpb=0.05)
          toolbox.register("select", tools.selTournament, tournsize=3)

          population = toolbox.population(n=num_of_population)
          hall_of_fame = tools.HallOfFame(num_of_population * num_of_generation)
          stats = tools.Statistics(lambda ind: ind.fitness.values)
          stats.register("avg", np.mean)
          stats.register("min", np.min)
          stats.register("max", np.max)

          population, log = algorithms.eaMuPlusLambda(population, toolbox, mu=num_of_population, lambda_=num_of_population * 2, cxpb=

          # Apply elitism after evolution
          best_individual = tools.selBest(population, 1)[0]
          hall_of_fame.update([best_individual])

          return hall_of_fame
```

Figure 29. This code snippet shows the Genetic Algorithm (GA) implementation function

The function in the code block labelled 37 whose code snippet is shown in figure 30 defines the fitness function used by the GA to determine the fitness of the individual in the population of a given generation. The fitness function accepts five arguments namely – individual (binary string for the selected features), predictors (the independent variables in the dataset), target (the dependent variable in the dataset), model (the model used to analyse the individual), and is_func (a Boolean variation use to build neural network model should the model need to be compiled). The fitness function uses the mean score of running cross-validation scores from the fitting model and the dataset provided to calculate the fitness of the individual.

```
[37]: def get_model_fitness(individual, predictors, target, model, is_func=False):
          selected_features = [pred for pred, sel in zip(predictors.columns, individual) if sel]
          X_subset = predictors[selected_features]
          if is_func:
              model = model(X_subset.shape[1])
          scores = cross_val_score(model, X_subset, target, cv=5, scoring='accuracy')
          return (np.mean(scores),)

[38]: def avg(score_list):
          return sum(score_list)/float(len(score_list))
```

Figure 30. This code snippet shows the fitness function implementation for GA and the function used to calculate the mean score for members of a population in GA

4.4 Feature Extraction from Raw Data

Two Python modules, ***features_extractions.py*** and ***features_to_extract.py***, were created to streamline feature extraction from a raw phishing website dataset. The ***features_to_extract.py*** module has functions to extract individual attributes from the raw data, while ***features_extractions.py*** manages the overall feature extraction process. The primary function in ***features_extractions.py*** is ***run_features_from_database()***, which takes three parameters: ***data_dir*** (the directory containing the raw data), ***extracted_features_csv*** (the path to save the extracted features), and ***columns*** (a list of column names for the extracted features). This function can be imported into the project code to extract and process features from the phishing website dataset.

```
run_features_from_datasets(data_dir, extracted_features_csv, get_column_names())
```

Figure 31. This code snippet shows the function call used to perform feature extraction from the raw dataset

4.5 Load Phishing Website Dataset

To begin the analysis of phishing websites, the extracted dataset was loaded into Jupyter Notebook using Python's Pandas library, creating a Pandas DataFrame. The code snippet in Figure x illustrates how to import the datasets into the IDE.

```
[40]: analysis_df = pd.read_csv(extracted_features_csv)
[41]: analysis_df
```

[41]:

	url_length	num_dots	num_hyphens	num_slash	num_www	https_token	ratio_digits_url	ratio_digits_host	has_shortening_service
0	34	2	0	4	0	1	0.029412	0.083333	1
1	40	3	0	3	1	0	0.000000	0.000000	1
2	48	3	0	4	1	0	0.125000	0.000000	1
3	52	3	1	5	1	0	0.000000	0.000000	1
4	33	4	0	3	0	0	0.000000	0.000000	1
...	...	...	...	...	...	...	...	...	...
79984	48	1	0	5	0	0	0.000000	0.000000	1
79985	245	2	0	6	0	1	0.334694	0.000000	1
79986	49	2	0	4	1	0	0.081633	0.000000	1
79987	44	3	1	4	1	0	0.022727	0.000000	1
79988	29	2	0	3	0	0	0.068966	0.100000	1

79989 rows × 39 columns

Figure 32. Code snippets showing the loading and visualization of the first 5 records and the last first record.

4.6 Data Cleaning, Preprocessing and Exploration

These code snippets utilize predefined functions to handle missing data, remove irrelevant attributes that don't contribute to distinguishing legitimate websites from phishing ones, and examine the distribution of the target variable to explore its binary class structure.

```
[42]: ### check and remove missing values
      analysis_df = remove_missing_or_nan_from_dataset(analysis_df)
      analysis_df.isnull().sum()
```

[42]:

```
url_length                  0
num_dots                    0
num_hyphens                 0
num_slash                   0
num_www                     0
https_token                 0
ratio_digits_url            0
ratio_digits_host           0
has_shortening_service      0
num_subdomains              0
prefix_suffix               0
length_words_raw            0
char_repeat                 0
shortest_words_raw          0
shortest_word_host          0
shortest_word_path          0
longest_words_raw           0
longest_word_host           0
longest_word_path           0
avg_words_raw               0
avg_word_host               0
avg_word_path               0
num_links                   0
ratio_int_links             0
ratio_ext_links             0
num_ext_css                 0
ratio_ext_redirection       0
ratio_ext_errors            0
external_favicon            0
links_in_tags               0
ratio_int_media             0
ratio_ext_media             0
safe_anchor                 0
empty_title                 0
domain_registration_length  0
has_dns_record              0
request_url                 0
anchor_url                  0
label                       0
dtype: int64
```

Figure 33. This code snippet shows the removal of possible missing values from the dataset

```
[43]: ### Remove features that does no help to classify website type i.e feature with one unique value e.g 0
non_contributory_features = get_column_names_that_has_one_unique_valud(analysis_df)
analysis_df = analysis_df.drop(non_contributory_features, axis=1)

[44]: analysis_df
```

[44]:

	url_length	num_dots	num_hyphens	num_slash	num_www	https_token	ratio_digits_url	ratio_digits_host	has_shortening_service	num_subdomains	...
0	34	2	0	4	0	1	0.029412	0.083333	1	2	...
1	40	3	0	3	1	0	0.000000	0.000000	1	3	...
2	48	3	0	4	1	0	0.125000	0.000000	1	3	...
3	52	3	1	5	1	0	0.000000	0.000000	1	3	...
4	33	4	0	3	0	0	0.000000	0.000000	1	3	...
...	...	...	...	...	...	...	...	...	...	...	...
79984	48	1	0	5	0	0	0.000000	0.000000	1	1	...
79985	245	2	0	6	0	1	0.334694	0.000000	1	2	...
79986	49	2	0	4	1	0	0.081633	0.000000	1	2	...
79987	44	3	1	4	1	0	0.022727	0.000000	1	3	...
79988	29	2	0	3	0	0	0.068966	0.100000	1	2	...

79989 rows × 39 columns

Figure 34. This code snippet for removing useless attributes from the dataset if they existed

```
### Pie Chart showing the binary class distribution for the dependent or target variable
category_label = ['LEGITIMATE WEBSITE', 'PHISHING WEBSITE']
plot_pie_chart_for_binary_distribution_visualization(analysis_df, category_label, "LEGITIMATE WEBSITE TO PHISHING WEBSITE DISTRIBUTION RATIO IN THE DATASET", t
```

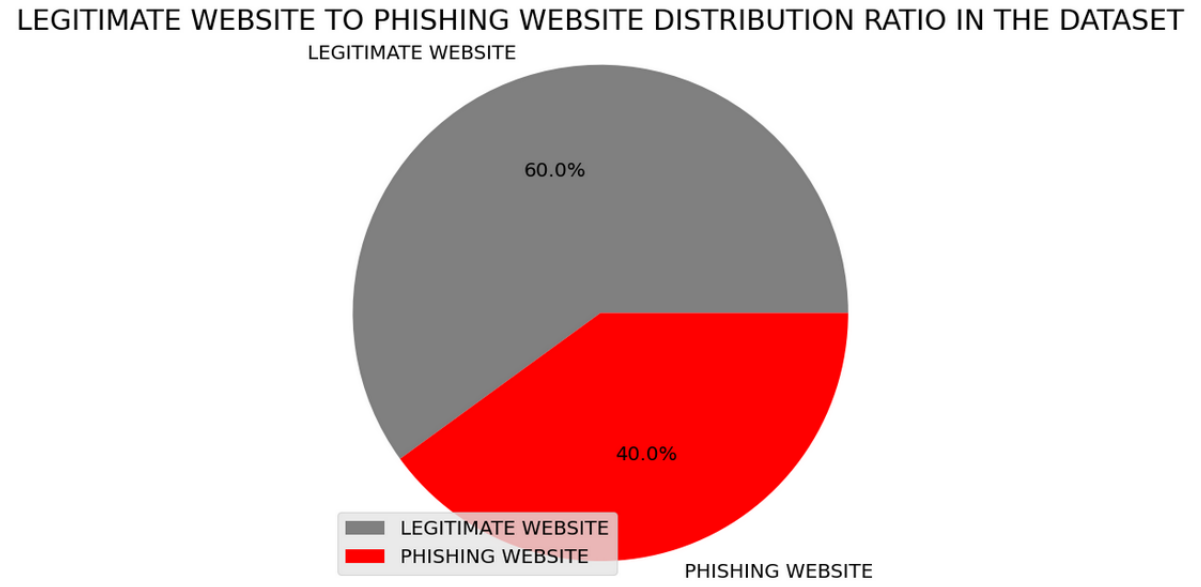

Figure 35. This code snippet for showing the class distribution of data either legitimate or phishing website

4.7 Model Implementation and Evaluation

The following code snippets demonstrate the execution of functions used to perform model analysis for Experiment 1 and Experiment 2. Additionally, this section includes a code snippet for generating a summary table, providing a comprehensive overview of the implementation

process. This summary table encapsulates key metrics and results from the experiments, serving as a reference point for evaluating the performance of different models.

The code block labelled 46 whose code snippet is shown in Figure 36 shows the final preparation of the dataset before running the baseline model analysis

```
[46]: analysis_result = dict()

      analysis_dependent_variable = analysis_df[target_variable]
      analysis_independent_variables = analysis_df.drop([target_variable], axis=1)

      input_shape = analysis_independent_variables.shape[1]

      processed_analysis_dataset = process_analysis_dataset(analysis_independent_variables, analysis_dependent_variable)
```

Figure 36. This code snippet shows the processing of the dataset into analysis-ready data to be analyzed by the model implementation

The code blocks whose code snippets are shown in Figure 37 show the invocation of the function used in running the baseline analysis for all implemented models.

```
[ ]: run_rfc_model(processed_analysis_dataset, analysis_result, rfc_model_name)

[ ]: run_svm_model(processed_analysis_dataset, analysis_result, svm_model_name)

[ ]: run_gbc_model(processed_analysis_dataset, analysis_result, gbc_model_name)

[ ]: run_gnn_model(processed_analysis_dataset, analysis_result, gnn_model_name)
```

Figure 37. This code snippet shows the code for running code blocks for implementing the baseline models or experiment 1 for the four models (RFC, SVM, GBC and GNN)

The code blocks whose code snippets are shown in Figure 38 show the invocation of ***run_genetic_algorithm_model_base_solution*** instance of the RFC given as a parameter to the function. After the GA feature selection is completed, the output from the GA operation is used to preparation of the dataset for the RFC hyperparameter-tuned model. The hyperparameters for the RFC model are stored in a dictionary data structure with a variable name ***grid_params*** and the ***run_rfc_model*** is invoked to run the RFC hyperparameter model analysis.

```
[ ]: accuracy, individual, header = run_genetic_algorithm_model_base_solution(analysis_df, instantiate_rfc_model(), False)

[ ]: analysis_dependent_variable = analysis_df[target_variable]
     analysis_independent_variables = analysis_df[header]
     processed_analysis_dataset = process_analysis_dataset(analysis_independent_variables, analysis_dependent_variable)

[ ]: rfc_model_name["name"] = "Genetic Algorithm Feature Selection with Random Forest Classifier"
     rfc_model_name["short"] = "GA-RFC"
     rfc_model_name["analysis"] = "Genetic Algorithm Feature Selection with Random Forest Classification Analysis"

     grid_params = {
         "n_estimators": [20, 50, 100],
         "criterion": ["gini", "entropy"],
         "max_depth": range(1, 10),
         "min_samples_split": range(1, 10),
         "min_samples_leaf": range(1, 5)
     }

     run_rfc_model(processed_analysis_dataset, analysis_result, rfc_model_name, grid_params)
```

Figure 38. This code snippet shows code for running the code block for implementing the GA feature selection with hyper-parameter turning for RFC model in experiment 2

The code blocks whose code snippets are shown in Figure 39 show the invocation of ***run_genetic_algorithm_model_base_solution*** instance of the SVM given as a parameter to the function. After the GA feature selection is completed, the output from the GA operation is used to preparation of the dataset for the SVM hyperparameter-tuned model. The hyperparameters for the SVM model are stored in a dictionary data structure with a variable name ***grid_params*** and the ***run_svm_model*** is invoked to run the SVM hyperparameter model analysis.

```
[ ]: accuracy, individual, header = run_genetic_algorithm_model_base_solution(analysis_df, instantiate_svm_model(), False)

[ ]: analysis_dependent_variable = analysis_df[target_variable]
     analysis_independent_variables = analysis_df[header]
     processed_analysis_dataset = process_analysis_dataset(analysis_independent_variables, analysis_dependent_variable)

[ ]: svm_model_name["name"] = "Genetic Algorithm Feature Selection with Support Vector Machine"
     svm_model_name["short"] = "GA-SVM"
     svm_model_name["analysis"] = "Genetic Algorithm Feature Selection with Support Vector Machine Classification Analysis"

     grid_params = {
         'kernel': ['linear', 'rbf'],
         'C': [0.001, 0.01, 0.1, 1, 10],
         'gamma': [1, 0.1, 0.01, 0.001, 0.0001],
     }

     run_svm_model(processed_analysis_dataset, analysis_result, svm_model_name, grid_params)

[ ]: accuracy, individual, header = run_genetic_algorithm_model_base_solution(analysis_df, instantiate_gbc_model(), False)
```

Figure 39. This code snippets shows code for running of code block for implementing the GA feature selection with hyper-parameter turning for SVM model in experiment 2

The code blocks whose code snippets are shown in Figure 40 show the invocation of ***run_genetic_algorithm_model_base_solution*** instance of the GBC given as a parameter to the function. After the GA feature selection is completed, the output from the GA operation is used to preparation of the dataset for the GBC hyperparameter-tuned model. The hyperparameters for the GBC model are stored in a dictionary data structure with a variable name ***grid_params*** and the ***run_gbc_model*** is invoked to run the GBC hyperparameter model analysis.

```
[ ]: accuracy, individual, header = run_genetic_algorithm_model_base_solution(analysis_df, instantiate_gbc_model(), False)

[ ]: analysis_dependent_variable = analysis_df[target_variable]
     analysis_independent_variables = analysis_df[header]
     processed_analysis_dataset = process_analysis_dataset(analysis_independent_variables, analysis_dependent_variable)

[ ]: gbc_model_name["name"] = "Genetic Algorithm Feature Selection with Gradient Boost Classifier"
     gbc_model_name["short"] = "GA-GBC"
     gbc_model_name["analysis"] = "Genetic Algorithm Feature Selection with Gradient Boost Classification Analysis"

     grid_params = {
         'n_estimators': [10, 20, 50],
         'learning_rate': [0.0001, 0.001, 0.01, 0.1, 0.2, 0.3],
         'max_depth': [3, 5, 7, 10]
     }

     run_gbc_model(processed_analysis_dataset, analysis_result, gbc_model_name, grid_params)

[ ]: accuracy, individual, header = run_genetic_algorithm_model_base_solution(analysis_df, get_model_for_ga(), False)
```

Figure 40. This code snippet shows code for running the code block for implementing the GA feature selection with hyper-parameter turning for the GBC model in experiment 2

The code blocks whose code snippets are shown in Figure 41 show the invocation of ***run_genetic_algorithm_model_base_solution*** instance of the GNN given as a parameter to the function. After the GA feature selection is completed, the output from the GA operation is used to preparation of the dataset for the GNN hyperparameter-tuned model. The hyperparameters for the GNN model are stored in a dictionary data structure with a variable name ***grid_params*** and the ***run_gnn_model*** is invoked to run the GNN hyperparameter model analysis.

```
[ ]: accuracy, individual, header = run_genetic_algorithm_model_base_solution(analysis_df, get_model_for_ga(), False)

[ ]: analysis_dependent_variable = analysis_df[target_variable]
     analysis_independent_variables = analysis_df[header]
     processed_analysis_dataset = process_analysis_dataset(analysis_independent_variables, analysis_dependent_variable)

[ ]: gnn_model_name["name"] = "Genetic Algorithm Feature Selection with Graph Neural Network"
     gnn_model_name["short"] = "GA-GNN"
     gnn_model_name["analysis"] = "Genetic Algorithm Feature Selection with Graph Neural Network Classification Analysis"

     grid_params = {
         'units_gc1': [32, 64, 128],
         'units_gc2': [16, 32, 64],
         'units_reshape': [16, 32, 64],
         'units_dense1': [16, 32, 64]
     }

     run_gnn_model(processed_analysis_dataset, analysis_result, gnn_model_name, grid_params)
```

Figure 41. This code snippet shows code for running the code block for implementing the GA feature selection with hyper-parameter turning for the GNN model in experiment 2

The code block whose code snippet is shown in Figure 42 shows the generation of the summary table for all models both in the baseline and hyperparameter-tuned implementation.

```
[ ]: models = list(analysis_result.keys())
     metrics = list(analysis_result[models[0]].keys())

     print("\n\n")
     title = "EVALUATION TABLE SHOWING SUMMARY FOR THE PERFORMANCE ALL MODELS (BASELINE AND GENETIC ALGORITHM FEATURE SELECTION MODELS)"
     print(title)
     print(draw_line(len(title)))
     print(draw_line(len(title)))
     print("\n\n")
     show_summary_model_evaluation_table(analysis_result, models, metrics)
     print("\n\n\n")
```

Figure 42. This code snippet shows code used for generating the summary table for all models in both experiment 1 and experiment 2

5 Conclusion

This configuration manual provides comprehensive guidelines for researchers aiming to replicate the code implementation outlined in this research report. By following these steps and using the same datasets, you can ensure consistent outcomes similar to those achieved in this study. The report includes detailed code snippets that collectively document the process used to meet the project's objectives and goals. By adhering to these guidelines, researchers can confidently replicate the experiment, validating the findings and further exploring the topics addressed in this work.

References

Ariyadasa, S., Fernando, Shantha and Fernando, Subha (2021) 'Phishing websites dataset.' Mendeley Data. https://doi.org/10.17632/n96ncsr5g4.1.