Configuration Manual

MSc Research Project
MSc in Data Analytics

Mani Maran Rajendran
Student ID: 23204711

School of Computing
National College of Ireland

Supervisor: Prof. Christian Horn

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name:	Mani Maran Rajendran		
Student ID:	23204711		
Programme:	MSc in Data Analytics	**Year:**	2024-2025
Module:	MSc Research Project		
Supervisor:	Christian Horn		
Submission Due Date:	12-12-2024		
Project Title:	Detecting Ransomware Payments in the Bitcoin Network: A Comprehensive Analysis and Classification Using Bitcoin Heist Ransomware Address Dataset		
Word Count:	944 words	**Page Count:**	17 pages

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: Mani Maran Rajendran

Date: 12-12-2024

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):	

Detecting Ransomware Payments in the Bitcoin Network: A Comprehensive Analysis and Classification Using Bitcoin Heist Ransomware Address Dataset

Mani Maran Rajendran
23204711

1 Introduction

This configuration manual outlines the steps required to replicate the research project using the provided Python code. The project aims to analyse Bitcoin ransomware data and evaluate machine learning and deep learning models to predict patterns in the dataset.

2 System Requirements

2.1 Hardware Requirements

- Processor: Intel(R) Core(TM) i5-1035G1 CPU @ 1.00GHz, 1190 Mhz, 4 Core(s), 8 Logical Processor(s)
- RAM: 8 GB
- Storage: 512 GB SSD + 512 GB HDD

2.2 Software Requirements

- Operating System: Windows 10 (64-bit)
- Development Environment: Jupyter Notebook (Anaconda)
- Programming Language: Python 3.7
- Libraries:
  - NumPy
  - Pandas
  - Matplotlib
  - Seaborn
  - Scikit-learn
  - TensorFlow
  - Keras
  - PyTorch Geometric
  - Scikeras

Figure 1 below depicts the importing of libraries. Anaconda does not include, the Pytorch, Pytorch Geometric and Scikeras Libraries by Default. These libraries are needed to be installed in order to work with the given code. To install these libraries following commands should be run in Anaconda terminal with the default environment activated:

1. pip install torch_geometric – This installs the Pytorch Geometric Library

2. pip install scikeras – This installs the Scikeras Library

```
#Import the libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.metrics import precision_score, recall_score, f1_score, accuracy_score
from scikeras.wrappers import KerasClassifier
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten, Conv1D, MaxPooling1D
from imblearn.over_sampling import SMOTE
import torch
from torch_geometric.data import Data
from torch.optim import Adam
from torch_geometric.nn import GINConv
from torch_geometric.nn import GCNConv
from sklearn.neighbors import kneighbors_graph
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv1D, MaxPooling1D, Flatten, Dense
from sklearn.base import BaseEstimator, ClassifierMixin
from torch.optim import Adam
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import StratifiedKFold
from torch_geometric.data import Data
import torch.nn.functional as F
import torch_geometric.transforms as T
from torch_geometric.nn import GCNConv
from sklearn.preprocessing import LabelEncoder, StandardScaler
import warnings
warnings.filterwarnings('ignore')
```

Figure 1: Importing the required libraries

3 Data Acquisition

Data required for the implementation is available in a CSV file downloaded from the UCI Machine Learning Repository. The data is then read into the workspace using the read_csv method in the Pandas library. Figure 2 shows the code to import the data.

```
#Read the dataset
df_bhd = pd.read_csv('BitcoinHeistData.csv')
df_bhd.head()
```

Figure 2: Importing the dataset

	address	year	day	length	weight	count	looped	neighbors	income	label
0	111K8kZAEnJg245r2cM6y9zgJGHZtJPy6	2017	11	18	0.008333	1	0	2	100050000.0	princetonCerber
1	1123pJv8jzeFQaCV4w644pzQJzVWay2zcA	2016	132	44	0.000244	1	0	1	100000000.0	princetonLocky
2	112536im7hy6wtKbpH1qYDWtTyMRAcA2p7	2016	246	0	1.000000	1	0	2	200000000.0	princetonCerber
3	1126eDRw2wqSkWosjTCre8cjjQW8sSeWH7	2016	322	72	0.003906	1	0	2	71200000.0	princetonCerber
4	1129TSjKtx65E35GiUo4AYVeyo48twbrGX	2016	238	144	0.072848	456	0	1	200000000.0	princetonLocky

Figure 3: First five rows of the dataset

4 Data Exploration

The data is then explored in depth to identify the dimensions, datatypes, and range of values present in the dataset. This is done mostly using the Pandas library. Figure 4 below shows the code to get the dataset dimensions.

```
#Dimension of dataset
df_bhd.shape

(2916697, 10)
```

Figure 4: Getting the dimensions of the dataset

Figure 5 depicts the column metadata.

```
#Checking the columns information
df_bhd.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2916697 entries, 0 to 2916696
Data columns (total 10 columns):
 #   Column     Dtype
---  ------     -----
 0   address    object
 1   year       int64
 2   day        int64
 3   length     int64
 4   weight     float64
 5   count      int64
 6   looped     int64
 7   neighbors  int64
 8   income     float64
 9   label      object
dtypes: float64(2), int64(6), object(2)
memory usage: 222.5+ MB
```

Figure 5: Column metadata information using Pandas

Figure 6 below shows the code and results for getting the presence of null values in the dataset.

```
#Checking the null value
df_bhd.isnull().sum()

address      0
year         0
day          0
length       0
weight       0
count        0
looped       0
neighbors    0
income       0
label        0
dtype: int64
```

Figure 6: Presence of Null Check

Pandas DataFrame's describe function is used to get the statistical information of the dataset as shown in Figure 7.

```
#Statistical Description of dataset
df_bhd.describe()
```

	year	day	length	weight	count	looped	neighbors	income
count	2.916697e+06	2.916697e+06	2.916697e+06	2.916697e+06	2.916697e+06	2.916697e+06	2.916697e+06	2.916697e+06
mean	2.014475e+03	1.814572e+02	4.500859e+01	5.455192e-01	7.216446e+02	2.385067e+02	2.206516e+00	4.464889e+09
std	2.257398e+00	1.040118e+02	5.898236e+01	3.674255e+00	1.689676e+03	9.663217e+02	1.791877e+01	1.626860e+11
min	2.011000e+03	1.000000e+00	0.000000e+00	3.606469e-94	1.000000e+00	0.000000e+00	1.000000e+00	3.000000e+07
25%	2.013000e+03	9.200000e+01	2.000000e+00	2.148438e-02	1.000000e+00	0.000000e+00	1.000000e+00	7.428559e+07
50%	2.014000e+03	1.810000e+02	8.000000e+00	2.500000e-01	1.000000e+00	0.000000e+00	2.000000e+00	1.999985e+08
75%	2.016000e+03	2.710000e+02	1.080000e+02	8.819482e-01	5.600000e+01	0.000000e+00	2.000000e+00	9.940000e+08
max	2.018000e+03	3.650000e+02	1.440000e+02	1.943749e+03	1.449700e+04	1.449600e+04	1.292000e+04	4.996440e+13

Figure 7: Obtaining the statistical description of the dataset

5 Exploratory Data Analysis

Exploratory Data Analysis or the EDA is done through numerous visualisations to identify any patterns pertaining to Ransomware transactions. These include, count plots, scatter plots, violin plots, bar plots, histograms, and finally a correlation check is performed on the dataset to find the presence of multicollinearity. Implementation of these things are given sequentially below.

```
# Filter out the "white" category
df_bhd_filtered = df_bhd[df_bhd['label'] != 'white']

# Create the countplot
sns.countplot(x='label', data=df_bhd_filtered, palette='hsv')
plt.xticks(rotation=90)
plt.show()
```

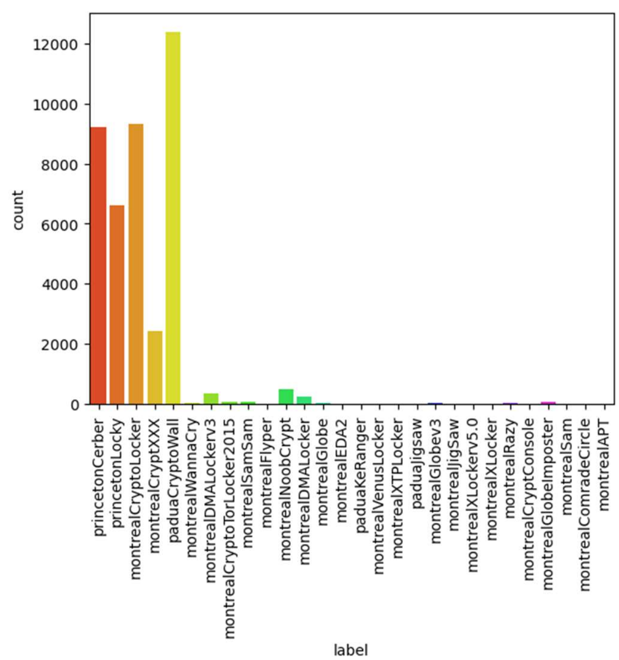

Figure 8: Count plot for Ransomware Types

```python
# Filter the dataset into two subsets
white_data = df_bhd[df_bhd['label'] == 'white']

# Scatterplot for 'White' category
sns.scatterplot(x='weight', y='income', hue='label', style='label', data=white_data, s=100)
plt.title('Income vs. Weight for White Category')
plt.xlabel('Weight')
plt.ylabel('Income')

# Adjust the legend position outside the plot
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
plt.show()
```

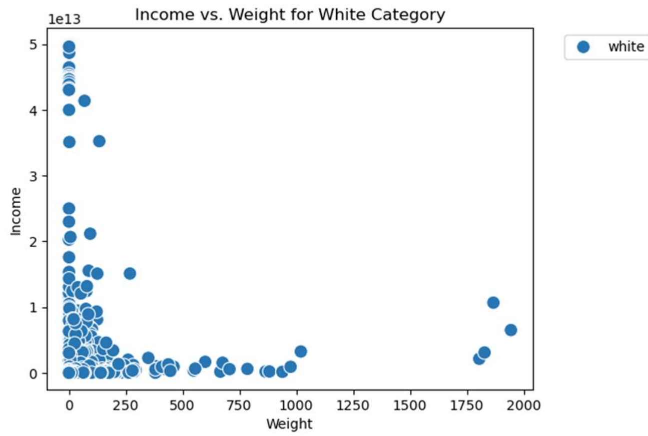

Figure 9: Scatterplot for Income vs. Weight for White Category

```python
# Filter the dataset into other categories
other_data = df_bhd[df_bhd['label'] != 'white']

# Set up the figure
plt.figure(figsize=(8, 6))

# Create scatterplot without labels
sns.scatterplot(
    x='weight',
    y='income',
    data=other_data,
    s=100,
    color='orange',
    edgecolor='black'
)

# Titles and labels
plt.title('Income vs. Weight for Other Categories', fontsize=16, pad=20)
plt.xlabel('Weight', fontsize=14, labelpad=10)
plt.ylabel('Income', fontsize=14, labelpad=10)

# Grid lines for alignment
plt.grid(visible=True, linestyle='--', linewidth=0.6, alpha=0.7, axis='both')

# Ensure all components are well-aligned
plt.tight_layout()

# Show the plot
plt.show()
```

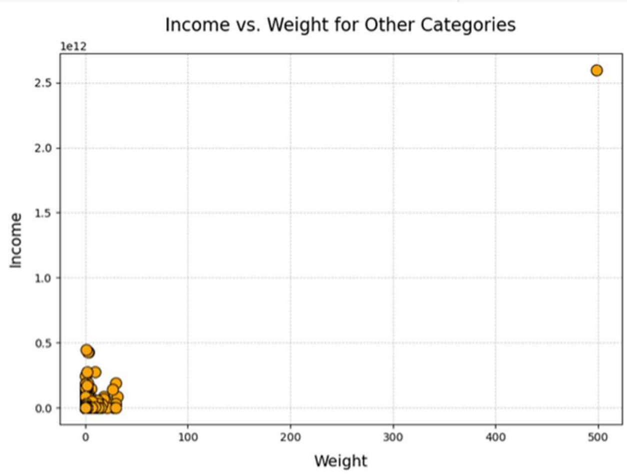

Figure 10: Scatterplot for Weigh vs Income Ransomware Category

```python
#Violinplot for label and length
plt.figure(figsize=(10, 6))
sns.violinplot(
    x='label',
    y='length',
    data=df_bhd,
    palette='muted'
)

# Add titles and axis labels
plt.title('Violin Plot of Length by Label', fontsize=16, pad=20)
plt.xlabel('Label', fontsize=14, labelpad=10)
plt.ylabel('Length', fontsize=14, labelpad=10)

# Rotate x-axis labels for clarity
plt.xticks(rotation=90, fontsize=12)
plt.yticks(fontsize=12)

# Ensure the layout is properly aligned
plt.tight_layout()

# Show the plot
plt.show()
```

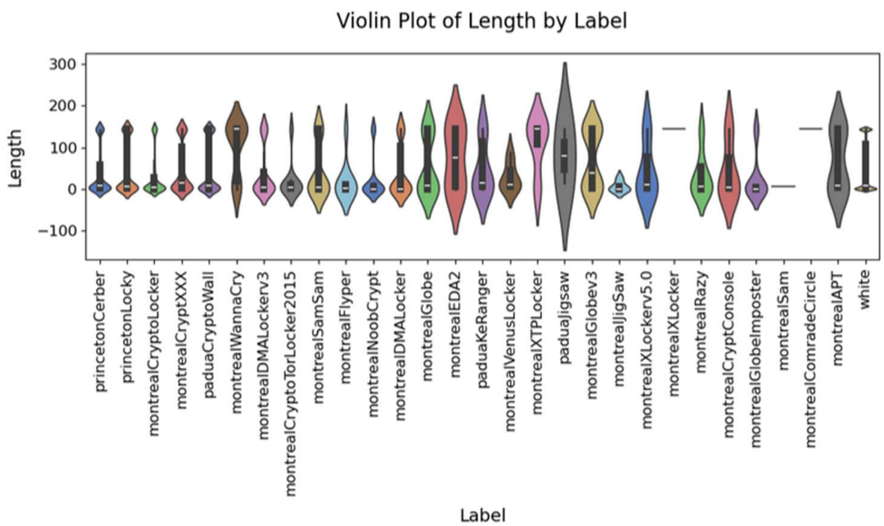

Figure 11: Violin Plot for Length by Label

```python
# Barplot year and length
sns.barplot(x='year', y='length', data=df_bhd,estimator='mean', ci=None,palette='hsv')
plt.title('Average Length by Year')
plt.xlabel('Year')
plt.ylabel('Average Length')
plt.show()
```

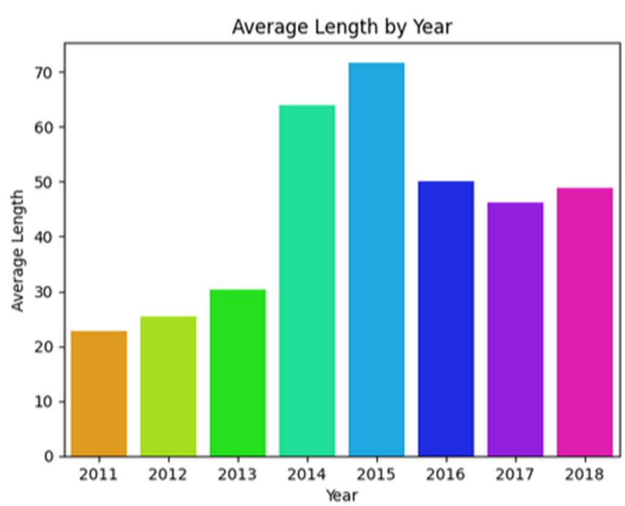

Figure 12: Average Length by Year Bar Plot

```python
# Create scatterplot without labels
sns.scatterplot(
    x='length',
    y='count',
    data=white_data,
    s=100,
    color='blue',
    edgecolor='black'
)

# Titles and labels
plt.title('Length vs. Count for White Category', fontsize=16, pad=20)
plt.xlabel('Length', fontsize=14, labelpad=10)
plt.ylabel('Count', fontsize=14, labelpad=10)

# Grid lines for alignment and readability
plt.grid(visible=True, linestyle='--', linewidth=0.6, alpha=0.7, axis='both')

# Ensure proper alignment of components
plt.tight_layout()

# Show the plot
plt.show()
```

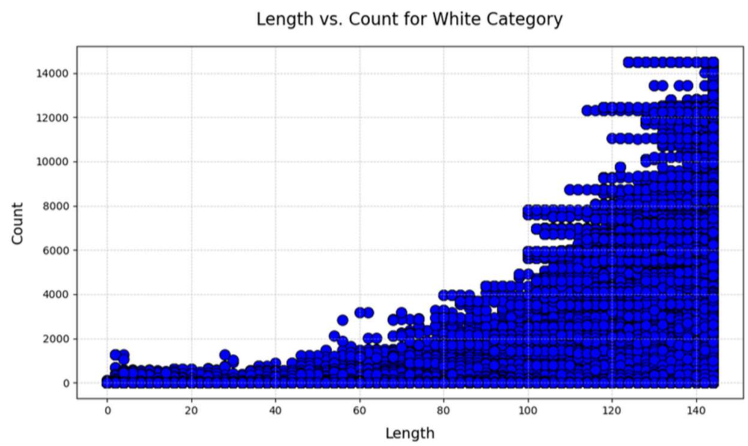

Figure 13: Scatterplot for Length vs. Count for White Category

```python
# Scatterplot for other categories
plt.figure(figsize=(10, 6))

# Create scatterplot without labels
sns.scatterplot(
    x='length',
    y='count',
    data=other_data,
    s=100,
    color='green',
    edgecolor='black'
)

# Titles and labels
plt.title('Length vs. Count for Other Categories', fontsize=16, pad=20)
plt.xlabel('Length', fontsize=14, labelpad=10)
plt.ylabel('Count', fontsize=14, labelpad=10)

# Grid lines for alignment and readability
plt.grid(visible=True, linestyle='--', linewidth=0.6, alpha=0.7, axis='both')

# Ensure proper alignment of components
plt.tight_layout()

# Show the plot
plt.show()
```

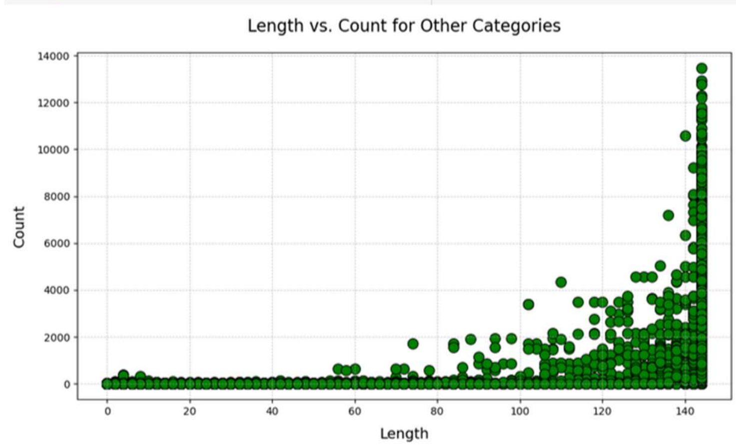

Figure 14: Scatterplot for Length vs. Count for Ransomware Categories

```python
#Barplot for year and income
sns.barplot(x='year', y='income', data=df_bhd, palette='hsv')
plt.title('Income Distribution by Year')
plt.xlabel('Year')
plt.ylabel('Income')
plt.show()
```

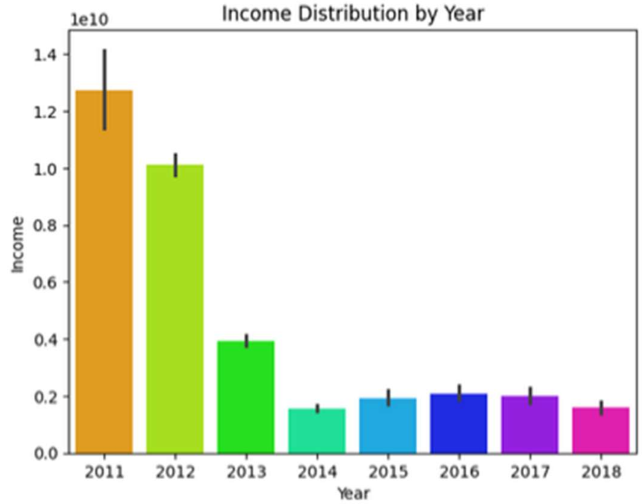

Figure 15: Bar plot for Income Distribution by Year

```python
#Histplot for year
sns.histplot(df_bhd['year'], bins=7, color='orange', edgecolor='black', kde=False)

plt.title('Histogram of Year')
plt.xlabel('Year')
plt.ylabel('Frequency')

# Ensure x-ticks align properly with the bin edges
plt.xticks(sorted(df_bhd['year'].unique()))

plt.grid(axis='y')
plt.show()
```

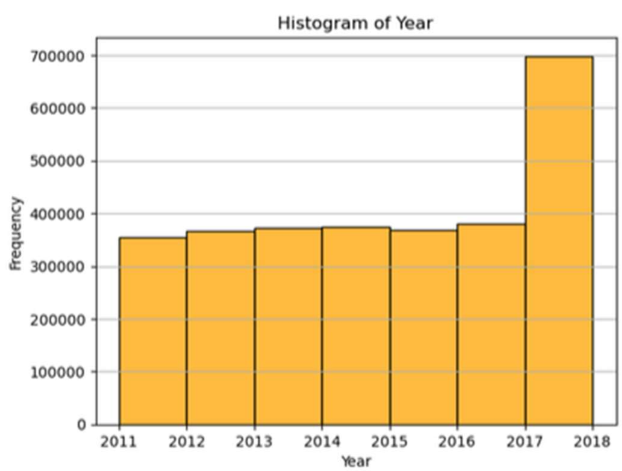

Figure 16: Histogram for Number of Transactions per Year

```python
# Histplot for Length
# Create the histplot with KDE
sns.histplot(df_bhd['length'], bins=30, kde=True, color='skyblue', edgecolor='black')

# Add titles and axis labels
plt.title('Distribution of Length', fontsize=16, pad=20)
plt.xlabel('Length', fontsize=14, labelpad=10)
plt.ylabel('Frequency', fontsize=14, labelpad=10)

# Adjust x and y axis ticks for better readability
plt.xticks(fontsize=12)
plt.yticks(fontsize=12)

# Ensure the layout is properly aligned
plt.tight_layout()

# Show the plot
plt.show()
```

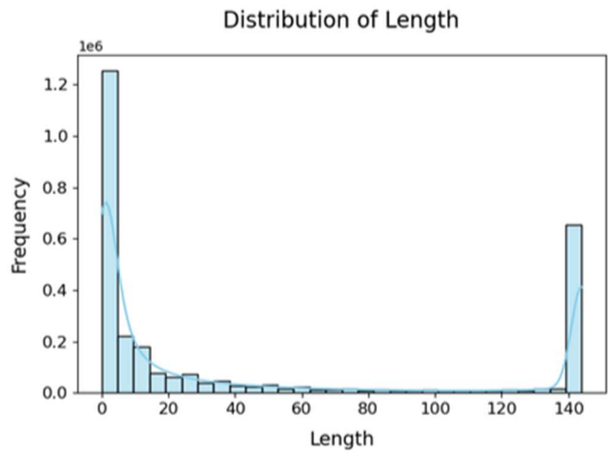

Figure 17: Histogram of the Length Column in the Dataset

```python
#Heatmap correlation
numerical_cols = df_bhd.select_dtypes(include=['float64', 'int64']).columns

# Calculate the correlation matrix
correlation_matrix = df_bhd[numerical_cols].corr()

plt.figure(figsize=(10, 6))

# Create a heatmap with the correlation matrix
sns.heatmap(correlation_matrix, annot=True, cmap='viridis', fmt='.2f', square=True, cbar_kws={"shrink": .8}, linewidths=0.5)

# Title and labels
plt.title('Correlation Heatmap of Numerical Features')
plt.show()
```

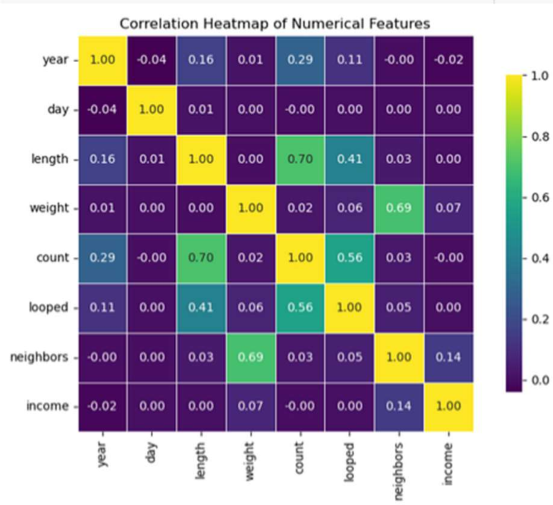

Figure 18: Correlation Matrix for The Dataset

6 Preprocessing

There are several preprocessing steps that has been taken to make the data ready for modelling starting from removing unnecessary columns, conversion of multi-label data to binary, label encoding of the label column in the dataset.

```
#Drop the address
df_bhd = df_bhd.drop(columns=["address","year","day"])
```

```
#Ransomware list
ransomware_list = ['princetonCerber','princetonLocky','montrealCryptoLocker',
                   'montrealCryptXXX','montrealWannaCry','montrealDMALockerv3', 'montrealCryptoTorLocker2015',
                   'montrealSamSam', 'montrealFlyper', 'montrealNoobCrypt','montrealDMALocker', 'montrealGlobe',
                   'montrealEDA2','montrealVenusLocker', 'montrealXTPLocker','montrealGlobev3', 'montrealJigSaw',
                   'montrealXLockerv5.0', 'montrealXLocker', 'montrealRazy','montrealCryptConsole', 'montrealGlobeImposter',
                   'montrealSam','montrealComradeCircle', 'montrealAPT',
                  'paduaCryptoWall','paduaKeRanger','paduaJigsaw']

# Replace ransomware names with a common label
df_bhd['label'] = df_bhd['label'].apply(lambda x: 'ransomware' if x in ransomware_list else x)

# Initialize the label encoder
label_encoder = LabelEncoder()

# Fit and transform the label column
df_bhd['label'] = label_encoder.fit_transform(df_bhd['label'])
```

Figure 19: Dropping unnecessary columns, conversion to binary classification problem, and label encoding

After this the data sampling has been performed for reducing computational overhead for the system.

```
# Randomly sample 40,000 rows from the dataset
df_bhd = df_bhd.sample(n=40000, random_state=42)
```

Figure 20: Data Sampling

The independent variables in the dataset are then separated from the dependent variable to create two separate DataFrames.

```
# Separating features and target variable
X = df_bhd.drop('label', axis=1)
y = df_bhd['label']
```

Figure 21: Separating Dependent and Independant Variables

The data is then divided into Training and Testing Set using train_test_split method from Sklearn.

```
# Splitting the dataset
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Figure 22: Data Splitting

Following this the variables are subjected to SMOTE to increase the number of samples for the class in minority.

```
# SMOTE for handling class imbalance
smote = SMOTE(random_state=42)
X_train, y_train = smote.fit_resample(X_train, y_train)
```

Figure 23: Implementation of the SMOTE

The features are then standardized ending the preprocessing and making data ready for modelling.

```
# Standard scaling
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

Figure 24: Data Standardisation

7 Modelling and Evaluation

Five different models namely, Random Forest, XGBoost, Convolutional Neural Network (CNN), Graph Convolutional Network (GCN), and Graph Isomorphism Network (GIN) are implemented in the study with hyperparameter tuning performed using the sklearn's GridSearchCV() method. The libraries used for the modelling are given in table below.

Model	Library	Hyperparameter Tuning
Random Forest	Sklearn	Sklearn
XGBoost	Xgboost	Sklearn
CNN	Tensorflow	Scikeras + Sklearn
GCN	Torch Geometric	Scikeras + Sklearn
GIN	Torch Geometric	Scikeras + Sklearn

Table 1: Models and Libraries

The implementations of these models are discussed hereafter.

```python
# Random Forest Hyperparameter Grid
rf_param_grid = {
    'n_estimators': [50, 100],
    'max_depth': [None, 10]
}

# Create Random Forest model
rf_model = RandomForestClassifier(random_state=42)

# Grid Search for Random Forest
rf_grid_search = GridSearchCV(estimator=rf_model, param_grid=rf_param_grid,
                              cv=StratifiedKFold(n_splits=3), scoring='f1', verbose=1)
rf_grid_search.fit(X_train, y_train)

# Best parameters for Random Forest
print("Best parameters for Random Forest:", rf_grid_search.best_params_)

# Predict on test data
y_pred_rf = rf_grid_search.predict(X_test)

# Evaluation for Random Forest
rf_accuracy = accuracy_score(y_test, y_pred_rf)
rf_precision = precision_score(y_test, y_pred_rf)
rf_recall = recall_score(y_test, y_pred_rf)
rf_f1 = f1_score(y_test, y_pred_rf)

print("Random Forest Evaluation:")
print(f"Accuracy: {rf_accuracy:.4f}")
print(f"Precision: {rf_precision:.4f}")
print(f"Recall: {rf_recall:.4f}")
print(f"F1 Score: {rf_f1:.4f}")
```

```
Fitting 3 folds for each of 4 candidates, totalling 12 fits
Best parameters for Random Forest: {'max_depth': None, 'n_estimators': 100}
Random Forest Evaluation:
Accuracy: 0.9464
Precision: 0.9885
Recall: 0.9568
F1 Score: 0.9724
```

Figure 25: RF Implementation and Evaluation

```python
# XGBoost with Grid Search
from xgboost import XGBClassifier

xgb_param_grid = {
    'n_estimators': [50, 100],
    'learning_rate': [0.01, 0.1]
}

# Create XGBoost model
xgb_model = XGBClassifier(use_label_encoder=False, eval_metric='logloss', random_state=42)

# Grid Search for XGBoost
xgb_grid_search = GridSearchCV(estimator=xgb_model, param_grid=xgb_param_grid,
                               cv=StratifiedKFold(n_splits=3), scoring='f1', verbose=1)
xgb_grid_search.fit(X_train, y_train)

# Best parameters for XGBoost
print("Best parameters for XGBoost:", xgb_grid_search.best_params_)

# Predict on test data
y_pred_xgb = xgb_grid_search.predict(X_test)

# Evaluation for XGBoost
xgb_accuracy = accuracy_score(y_test, y_pred_xgb)
xgb_precision = precision_score(y_test, y_pred_xgb)
xgb_recall = recall_score(y_test, y_pred_xgb)
xgb_f1 = f1_score(y_test, y_pred_xgb)

print("XGBoost Evaluation:")
print(f"Accuracy: {xgb_accuracy:.4f}")
print(f"Precision: {xgb_precision:.4f}")
print(f"Recall: {xgb_recall:.4f}")
print(f"F1 Score: {xgb_f1:.4f}")
```

```
Fitting 3 folds for each of 4 candidates, totalling 12 fits
Best parameters for XGBoost: {'learning_rate': 0.1, 'n_estimators': 100}
XGBoost Evaluation:
Accuracy: 0.9129
Precision: 0.9897
Recall: 0.9214
F1 Score: 0.9543
```

Figure 26: XGBoost Implementation and Evaluation

```python
# Function to create model
def create_model(filters=32, kernel_size=2, activation='relu', pool_size=2):
    model = Sequential()
    model.add(Conv1D(filters=filters, kernel_size=kernel_size, activation=activation, input_shape=(6, 1)))
    model.add(MaxPooling1D(pool_size=pool_size))
    model.add(Flatten())
    model.add(Dense(10, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))  # Binary classification
    model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
    return model

# Wrapping Keras model with KerasClassifier
cnn_model = KerasClassifier(model=create_model, verbose=0)

# Define the parameter grid for grid search
param_grid = {
    'epochs': [10],
    'batch_size': [32, 64]
}

# GridSearchCV for hyperparameter tuning
grid = GridSearchCV(estimator=cnn_model, param_grid=param_grid, n_jobs=-1, cv=3)

# Reshape X data to be compatible with Conv1D (samples, time steps, features)
X_train_reshaped = X_train.reshape((X_train.shape[0], X_train.shape[1], 1))
X_test_reshaped = X_test.reshape((X_test.shape[0], X_test.shape[1], 1))

# Fit the model using grid search
grid_result = grid.fit(X_train_reshaped, y_train)

# Best parameters from grid search
best_params = grid_result.best_params_
print(f"Best parameters: {best_params}")

# Predict on test data
y_pred = grid.predict(X_test_reshaped)

# Convert probabilities to binary predictions
y_pred_binary = (y_pred > 0.5).astype(int)
```

Figure 27: Implementation of the CNN model

```python
# Evaluation Metrics
accuracy_cnn = accuracy_score(y_test, y_pred_binary)
precision_cnn = precision_score(y_test, y_pred_binary)
recall_cnn = recall_score(y_test, y_pred_binary)
f1_cnn = f1_score(y_test, y_pred_binary)

# Display results
print(f"Accuracy: {accuracy_cnn:.4f}")
print(f"Precision: {precision_cnn:.4f}")
print(f"Recall: {recall_cnn:.4f}")
print(f"F1 Score: {f1_cnn:.4f}")
```

```
Best parameters: {'batch_size': 32, 'epochs': 10}
Accuracy: 0.7917
Precision: 0.9893
Recall: 0.7976
F1 Score: 0.8832
```

Figure 28: Evaluation of the CNN Model

```python
# Create k-NN graph for edge index
def create_edge_index(X, k=5):
    A = kneighbors_graph(X, k, mode='connectivity', include_self=True)
    edge_index = np.array(A.nonzero())
    return torch.tensor(edge_index, dtype=torch.long)

# Create edge_index using k-NN (k=5)
edge_index_train = create_edge_index(X_train)
edge_index_test = create_edge_index(X_test)

y_train_np = y_train.to_numpy()
y_test_np = y_test.to_numpy()
```

Figure 29: Creating Graph Data for Modelling

```python
# Graph Convolutional Network (GCN) model
class GCN(torch.nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super(GCN, self).__init__()
        # First GCN layer
        self.conv1 = GCNConv(input_dim, hidden_dim)
        # Second GCN layer
        self.conv2 = GCNConv(hidden_dim, output_dim)

    def forward(self, data):
        # Extract node features and edge index
        x, edge_index = data.x, data.edge_index
        # Pass through the first GCN layer
        x = self.conv1(x, edge_index)
        # Apply ReLU activation function
        x = F.relu(x)
        # Pass through the second GCN layer
        x = self.conv2(x, edge_index)
        # Apply log-softmax for output probabilities
        return F.log_softmax(x, dim=1)
```

```python
# Custom GCN classifier for GridSearch compatibility
class GCNClassifier(BaseEstimator, ClassifierMixin):
    def __init__(self, input_dim=6, hidden_dim=16, output_dim=2, epochs=10, lr=0.01):
        self.input_dim = input_dim
        self.hidden_dim = hidden_dim
        self.output_dim = output_dim
        self.epochs = epochs
        self.lr = lr
        self.model = None

    def fit(self, X, y):
        # Prepare data for PyTorch Geometric
        edge_index = create_edge_index(X)
        X_tensor = torch.tensor(X, dtype=torch.float)
        y_tensor = torch.tensor(y, dtype=torch.long)
        data = Data(x=X_tensor, edge_index=edge_index, y=y_tensor)

        # Initialize model and optimizer
        self.model = GCN(self.input_dim, self.hidden_dim, self.output_dim)
        optimizer = Adam(self.model.parameters(), lr=self.lr)
        criterion = torch.nn.CrossEntropyLoss()

        # Training loop
        for epoch in range(self.epochs):
            self.model.train()
            optimizer.zero_grad()
            out = self.model(data)
            loss = criterion(out, data.y)
            loss.backward()
            optimizer.step()

        return self

    def predict(self, X):
        # Prepare data for prediction
        edge_index = create_edge_index(X)
        X_tensor = torch.tensor(X, dtype=torch.float)
        data = Data(x=X_tensor, edge_index=edge_index)

        # Model inference
        self.model.eval()
        with torch.no_grad():
            out = self.model(data)
            pred = out.argmax(dim=1)
        return pred.cpu().numpy()

    def score(self, X, y):
        # Calculate accuracy
        y_pred = self.predict(X)
        return accuracy_score(y, y_pred)
```

```python
# Prepare the dataset in the required format
X_train_np = X_train.values if isinstance(X_train, pd.DataFrame) else X_train
y_train_np = y_train.to_numpy() if isinstance(y_train, pd.Series) else y_train

# Grid search parameters
param_grid = {
    'hidden_dim': [16, 32],
    'lr': [0.001, 0.01]
}

# Run GridSearchCV with the GCN model
grid = GridSearchCV(estimator=GCNClassifier(input_dim=6, output_dim=len(np.unique(y_train_np))),
                    param_grid=param_grid, cv=3, n_jobs=-1)
grid_result = grid.fit(X_train_np, y_train_np)

# Display the best parameters
print(f"Best parameters: {grid_result.best_params_}")
```

```
Best parameters: {'hidden_dim': 16, 'lr': 0.01}
```

Figure 30: Implementation of the GCN Model

```python
# Get predictions on test data
y_pred = grid.predict(X_test)

# Compute evaluation metrics
accuracy_gcn = accuracy_score(y_test, y_pred)
precision_gcn = precision_score(y_test, y_pred, average='weighted')
recall_gcn = recall_score(y_test, y_pred, average='weighted')
f1_gcn = f1_score(y_test, y_pred, average='weighted')

# Print metrics
print(f"Accuracy: {accuracy_gcn:.4f}")
print(f"Precision: {precision_gcn:.4f}")
print(f"Recall: {recall_gcn:.4f}")
print(f"F1 Score: {f1_gcn:.4f}")
```

```
Accuracy: 0.6555
Precision: 0.9735
Recall: 0.6555
F1 Score: 0.7809
```

Figure 31: Evaluation of the GCN Model

```python
# Define the GIN Model
class GIN(torch.nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super(GIN, self).__init__()
        self.conv1 = GINConv(torch.nn.Linear(input_dim, hidden_dim))
        self.conv2 = GINConv(torch.nn.Linear(hidden_dim, output_dim))

    def forward(self, data):
        x, edge_index = data.x, data.edge_index
        x = self.conv1(x, edge_index)
        x = F.relu(x)
        x = self.conv2(x, edge_index)
        return F.log_softmax(x, dim=1)

# Define the GINClassifier class for GridSearch
class GINClassifier(BaseEstimator, ClassifierMixin):
    def __init__(self, input_dim=6, hidden_dim=16, output_dim=2, epochs=10, lr=0.01):
        self.input_dim = input_dim
        self.hidden_dim = hidden_dim
        self.output_dim = output_dim
        self.epochs = epochs
        self.lr = lr
        self.model = None

    def fit(self, X, y):
        # Prepare the GIN model and data
        edge_index = create_edge_index(X)
        X_tensor = torch.tensor(X, dtype=torch.float)
        y_tensor = torch.tensor(y, dtype=torch.long)
        data = Data(x=X_tensor, edge_index=edge_index, y=y_tensor)

        # Initialize model and optimizer
        self.model = GIN(self.input_dim, self.hidden_dim, self.output_dim)
        optimizer = Adam(self.model.parameters(), lr=self.lr)
        criterion = torch.nn.CrossEntropyLoss()

        # Training loop
        for epoch in range(self.epochs):
            self.model.train()
            optimizer.zero_grad()
            out = self.model(data)
            loss = criterion(out, data.y)
            loss.backward()
            optimizer.step()

        return self
```

Figure 32: Implementation of the GIN model

```python
def predict(self, X):
    # Make predictions using the trained model
    self.model.eval()
    edge_index = create_edge_index(X)
    X_tensor = torch.tensor(X, dtype=torch.float)
    data = Data(x=X_tensor, edge_index=edge_index)

    with torch.no_grad():
        out = self.model(data)
        pred = out.argmax(dim=1)
    return pred.cpu().numpy()

def score(self, X, y):
    y_pred = self.predict(X)
    return accuracy_score(y, y_pred)
```

Figure 33: Functions Evaluation of the GIN Model

```python
# Prepare data in the proper format for GINClassifier
X_train_np = X_train.values if isinstance(X_train, pd.DataFrame) else X_train
y_train_np = y_train.to_numpy() if isinstance(y_train, pd.Series) else y_train

# Perform grid search
param_grid = {
    'hidden_dim': [16, 32],
    'lr': [0.001, 0.01]
}

# Initialize GINClassifier and grid search
grid = GridSearchCV(estimator=GINClassifier(input_dim=6, output_dim=len(np.unique(y_train_np))),
                    param_grid=param_grid, cv=3, n_jobs=-1)

grid_result = grid.fit(X_train_np, y_train_np)

# Display the best parameters
print(f"Best parameters: {grid_result.best_params_}")

# Get predictions on test data
y_pred = grid.predict(X_test)

# Compute evaluation metrics
accuracy_gin = accuracy_score(y_test, y_pred)
precision_gin = precision_score(y_test, y_pred, average='weighted')
recall_gin = recall_score(y_test, y_pred, average='weighted')
f1_gin = f1_score(y_test, y_pred, average='weighted')

# Print metrics
print(f"Accuracy: {accuracy_gin:.4f}")
print(f"Precision: {precision_gin:.4f}")
print(f"Recall: {recall_gin:.4f}")
print(f"F1 Score: {f1_gin:.4f}")
```

```
Best parameters: {'hidden_dim': 32, 'lr': 0.01}
Accuracy: 0.3385
Precision: 0.9739
Recall: 0.3385
F1 Score: 0.4933
```

Figure 34: Evaluation of the GIN

```python
# Results for each model
results = {
    "Model": ["Random Forest", "XGBoost", "CNN", "GCN", "GIN"],
    "Accuracy": [rf_accuracy, xgb_accuracy, accuracy_cnn, accuracy_gcn, accuracy_gin],
    "Precision": [rf_precision, xgb_precision, precision_cnn, precision_gcn, precision_gin],
    "Recall": [rf_recall, xgb_recall, recall_cnn, recall_gcn, recall_gin],
    "F1 Score": [rf_f1, xgb_f1, f1_cnn, f1_gcn, f1_gin]
}

# Create a DataFrame
evaluation_df = pd.DataFrame(results)

# Display the DataFrame
print(evaluation_df)
```

```
           Model  Accuracy  Precision    Recall  F1 Score
0  Random Forest  0.946375   0.988486  0.956814  0.972392
1        XGBoost  0.912875   0.989661  0.921353  0.954286
2            CNN  0.791750   0.989318  0.797619  0.883186
3            GCN  0.655500   0.973518  0.655500  0.780931
4            GIN  0.338500   0.973888  0.338500  0.493266
```

Figure 35: Comparative Analysis of the Models

8 Execution Guide

- Start the Anaconda Navigator
- Open Jupyter or JupyterLab – This will open a webpage
- Upload the Code File and Dataset File to the WorkSpace
- Hit Run All to Execute all the Cells or Hit Run to Run Each Cell Separately

9 References

- *Guide* (no date). https://www.tensorflow.org/guide.
- *PyG Documentation — pytorch_geometric documentation* (no date). https://pytorch-geometric.readthedocs.io/en/latest/.
- *scikit-learn: machine learning in Python — scikit-learn 0.16.1 documentation* (no date). https://scikit-learn.org/.
- *Welcome to SciKeras's documentation! — SciKeras 0.13.0 documentation* (no date). https://adriangb.com/scikeras/stable/.
- *UCI Machine Learning Repository* (no date b). https://archive.ics.uci.edu/dataset/526/bitcoinheistransomwareaddressdataset.