National College of Ireland

# Enhancing Network Security Using Machine Learning Model-Agnostic Approach on Diverse Datasets

## Configuration Manual

MSc Research Project
Data Analytics

Muhammad Zaeem
Student ID: x23108088

School of Computing
National College of Ireland

Supervisor: Arjun Chikkankod

**National College of Ireland**

**MSc Project Submission Sheet**

**School of Computing**

| | | | |
|---|---|---|---|
| **Student Name:** | Muhammad Zaeem | | |
| **Student ID:** | X23108088 | | |
| **Programme:** | MSc Data Analytics | **Year:** | 2024 |
| **Module:** | MSc Research Project | | |
| **Lecturer:** | Arjun Chikkankod | | |
| **Submission Due Date:** | 12th August, 2024 | | |
| **Project Title:** | Enhancing Network Security Using Machine Learning Model-Agnostic Approach on Diverse Datasets | | |
| **Word Count:** | 2500+ | **Page Count:** | 22 |

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:** Muhammad Zaeem

**Date:** 12th August, 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): | |

# Enhancing Network Security Using Machine Learning Model-Agnostic Approach on Diverse Datasets

Muhammad Zaeem
X23108088

# 1 Introduction

It is a mode-agnostic project for the in-depth analysis of different network traffic classifications and intrusion datasets to address challenges like class imbalance, feature importance, and hyperparameter tuning aiming to find the best-performing environment for model to improve their accuracy and efficiency. Another goal is to find an inference model that can be suggested to be implemented in real-life intrusion detection systems and firewalls to enhance network security. This configuration manual is designed to provide all the details that can help anyone recreate this research.

# 2 System Requirements

For the efficient execution of this research artefact, the following hardware and software requirements should be considered beforehand

## 2.1 Hardware requirements

**Table 1.** shows the needed hardware requirements to run the code artefact

| | |
|---|---|
| **Operating System (OS)** | Windows 10 or similar OS |
| **Processor** | Intel i5 8th Gen or above |
| **RAM** | 16 GB DDR4 |
| **Storage** | 50 GB available space |

**Table 1: Hardware Specifications**

## 2.2 Software Requirements

Microsoft Excel is used for the early exploration of datasets. For the implementation of the project, the Jupyter Notebook IDE from Anaconda Environment is used which has Python kernel preinstalled in it.

- **Python Kernel version: 3.11.4**
- **Jupyter Notebook version: 7.0.0**

# 3 Dataset Collection

Three datasets are used in this research. The first one is the network traffic classification dataset taken from the UCI Machine Learning repository[1] named as Internet Firewall Dataset. The second Dataset is the UNSW NB15 dataset taken from Kaggle[2], which was published by the Cyber Security Department of the University of New South Wales, Canberra, Australia (Moustafa and Slay, 2015).

[1] https://archive.ics.uci.edu/dataset/542/internet+firewall+data
[2] https://www.kaggle.com/datasets/dhoogla/unswnb15

The third dataset, NF-UQ-NIDS-v2 Network Intrusion Detection Dataset made by the combination of small datasets and is available on Kaggle[3] with a usability rate of 10 points.

# 4 Project Development

Before diving into the working of specific datasets, the following Python libraries and their specific versions are essential for running the code efficiently in this project. Ensure you install these versions to avoid compatibility issues:

- **Pandas** – For data manipulation and analysis.

  **Version: 1.5.3**

- **NumPy** – For numerical computations.

  **Version: 1.24.3**

- **Scikit-learn** – For machine learning model development, feature scaling, evaluation metrics, and hyperparameter tuning.

  **Version: 1.5.3**

- **XGBoost** – For implementing XGBoost models.

  **Version: 1.5.3**

- **Scikeras** – For interfacing Keras with Scikit-learn.

  **Version: 1.5.3**

- **TensorFlow** – As the backend for Keras.

  **Version: 2.16.1**

- **Matplotlib** – For visualising data and results.

  **Version: 3.7.1**

- **Seaborn** – For statistical data visualisation.

  **Version: 0.12.2**

- **Imbalanced-learn (Imblearn)** – For handling imbalanced datasets using techniques like random oversampling.

  **Version: 1.5.3**

These are needed for the development and smooth working of project functions. If any package is not available, it can be installed using the "pip install package_name" command in jupyter notebook. Import these into all three notebooks for three datasets as shown in the Figure 1.

[3] https://www.kaggle.com/datasets/aryashah2k/nfuqnidsv2-network-intrusion-detection-dataset

```
import warnings
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.preprocessing import LabelEncoder
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer

from sklearn.model_selection import train_test_split,cross_val_score,GridSearchCV, StratifiedKFold
from sklearn.ensemble import RandomForestClassifier
from sklearn.neighbors import KNeighborsClassifier
from xgboost import XGBClassifier
from sklearn.svm import SVC, LinearSCV
from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import GaussianNB

from sklearn.preprocessing import StandardScaler
from imblearn.over_sampling import RandomOverSampler
from sklearn.metrics import accuracy_score, recall_score, precision_score, f1_score, confusion_matrix

import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from scikeras.wrappers import KerasClassifier

warnings.simplefilter(action="ignore", category=FutureWarning)
```

**Figure 1: Libraries to be Imported**

## 4.1 Internet Firewall Dataset (IFD)

Import the Internet Firewall dataset into Jupyter Notebook IDE using the pandas function to read a CSV file as shown in Figure 2.

```
df = pd.read_csv("C:\\Users\\hp\\Downloads\\log2.csv")
df.shape
```

**Figure 2: Read the CSV file for IFD**

After an exploratory data analysis, such as checking shape, data types, missing values, and outliers, we found some duplicate values, we dropped them using 'df.drop_duplicates(inplace=True)' function. After that, we performed label encoding of the target variable as shown in Figure 3.

```
label_encoder = LabelEncoder()
df["Action"] = label_encoder.fit_transform(df["Action"])
```

**Figure 3: Encoding Target Variable**

Data is split into train and test sets with a split ratio of 0.3, meaning the training set has 70% data and the training set with 30% data as shown in Figure 4.

```
X = df.drop("Action",axis=1)
y = df["Action"]
X_train, X_test, y_train, y_test = train_test_split(X,y, test_size = 0.3, random_state = 88)

print("X_train shape:", X_train.shape)
print("y_train shape:", y_train.shape)
print("X_test shape:", X_test.shape)
print("y_test shape:", y_test.shape)
```

**Figure 4: Splitting Dataset into Train and Test Sets**

Upon initial implementation of models, errors were analyzed due to the presence of some infinite values in the dataset. The mean imputation method shown in Figure 5. is used to replace all such outliers, infinite and nan values in the dataset.

```
# Replaced infinite with NaN
X_train = np.where(np.isfinite(X_train), X_train, np.nan)
X_test = np.where(np.isfinite(X_test), X_test, np.nan)

# Replace too large outliers values with NaN
max_allowed_value = np.finfo(np.float32).max
X_train[X_train > max_allowed_value] = np.nan
X_test[X_test > max_allowed_value] = np.nan

# Impute NaNs with column means
imputer = SimpleImputer(strategy='mean')
X_train = imputer.fit_transform(X_train)
X_test = imputer.transform(X_test)
```

**Figure 5: Mean Imputation for Nan, Infinite values and Outliers**

After this initial data preprocessing, the model-agnostic pipeline is created following the steps shown in Figures 6, 7 and 8. All models are implemented except for ANN as it is implemented separately. Feature scaling and model evaluation are also performed within this pipeline and results are saved into a data frame.

```
# List of models
models = [
    ("Logistic Regression", LogisticRegression(random_state=88, max_iter=10000)),
    ("Naive Bayes", GaussianNB()),
    ("Random Forest", RandomForestClassifier(random_state=88)),
    ("XGBoost", XGBClassifier(random_state=88)),
    ("KNN", KNeighborsClassifier()),
    ("SVM", SVC(random_state=88))
]

# List of class names
class_names = ['allow', 'drop', 'deny', 'reset-both']

# Initialize best_model to None
best_model = None
best_accuracy = 0.0

# List to store results
results = []
```

**Figure 6: Defining Models and Class Names for Pipeline**

```python
# Iterate over models and evaluate their performance
for name, model in models:
    # pipeline for feature scaling
    pipeline = Pipeline([
        ("scaler", StandardScaler()),
        ("model", model)
    ])
    # Fitting pipeline to train
    pipeline.fit(X_train, y_train)
    # prediction on training data
    X_pred = pipeline.predict(X_train)
    # prediction on test data
    y_pred = pipeline.predict(X_test)
    # accuracy score
    train_accuracy = accuracy_score(y_train, X_pred)
    accuracy = accuracy_score(y_test, y_pred)
    # recall score
    recall = recall_score(y_test, y_pred, average='macro')
    # precision score
    precision = precision_score(y_test, y_pred, average='macro')
    # F1 score
    f1 = f1_score(y_test, y_pred, average='macro')
    # confusion matrix
    cm = confusion_matrix(y_test, y_pred)
    # Calculate class-specific accuracy
    class_specific_accuracy = cm.diagonal() / cm.sum(axis=1)
    # Printing evaluation metrics
    print(f"Model: {name}")
    print(f"Training Accuracy: {train_accuracy}")
    print(f"Test Accuracy: {accuracy}")
    print(f"Recall (Macro): {recall}")
    print(f"Precision (Macro): {precision}")
    print(f"F1 Score (Macro): {f1}")
    for i, class_name in enumerate(class_names):
        print(f"Class {class_name}: {class_specific_accuracy[i]}")
    print()

    # Appending results to the list
    results.append({
        "Model": name,
        "Training Accuracy": train_accuracy,
        "Test Accuracy": accuracy,
        "Recall (Macro)": recall,
        "Precision (Macro)": precision,
        "F1 Score (Macro)": f1,
        **{f"Class {class_name}": class_specific_accuracy[i] for i, class_name in enumerate(class_names)}
    })
```

**Figure 7: Model Agnostic Pipeline**

```python
    # checking the best model
    if accuracy > best_accuracy:
        best_accuracy = accuracy
        best_model = pipeline

    # Plot confusion matrix
    plt.figure(figsize=(12, 6))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', cbar=False, xticklabels=class_names, yticklabels=class_names)
    plt.xlabel('Predicted labels')
    plt.ylabel('True labels')
    plt.title(f'Confusion Matrix for {name}')
    plt.show()
print("Best Model:", best_model)
results_df = pd.DataFrame(results)
results_df
```

**Figure 8: Displaying best model, confusion matrices and results**

ANN is applied following the same steps as other models; it is applied separately as shown in Figure 9. to get clearer output as it runs on many epochs creating longer outputs.

```
model = Sequential()
model.add(Dense(64, input_shape=(X_train.shape[1],), activation='relu'))
model.add(Dense(64, activation='relu'))
model.add(Dense(32, activation='relu'))
model.add(Dense(len(np.unique(y_train)), activation='softmax'))

# model compiling
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])

# Training model
history = model.fit(X_train, y_train, epochs=50, batch_size=32, validation_split=0.2, verbose=1)

# Evaluate on testset
test_loss, test_accuracy = model.evaluate(X_test, y_test, verbose=0)

# predictions
y_pred = model.predict(X_test)
y_pred_classes = np.argmax(y_pred, axis=1)

# evaluation metrics
accuracy = accuracy_score(y_test, y_pred_classes)
recall = recall_score(y_test, y_pred_classes, average='macro')
precision = precision_score(y_test, y_pred_classes, average='macro')
f1 = f1_score(y_test, y_pred_classes, average='macro')
conf_matrix = confusion_matrix(y_test, y_pred_classes)
class_specific_accuracy = conf_matrix.diagonal() / conf_matrix.sum(axis=1)

# Printing metrics
print(f"Model: ANN")
print(f"Training Accuracy: {history.history['accuracy'][-1]}")
print(f"Test Accuracy: {test_accuracy}")
print(f"Recall (Macro): {recall}")
print(f"Precision (Macro): {precision}")
print(f"F1 Score (Macro): {f1}")
for i, class_name in enumerate(class_names):
    print(f"Class {class_name}: {class_specific_accuracy[i]}")
print()
```

```
# store results to df
new_row = {
    'Model': 'ANN',
    'Training Accuracy': history.history['accuracy'][-1],
    'Test Accuracy': test_accuracy,
    'Recall (Macro)': recall,
    'Precision (Macro)': precision,
    'F1 Score (Macro)': f1,
    **{f'Class {class_name}': class_specific_accuracy[i] for i, class_name in enumerate(class_names)}
}

results_df = results_df.append(new_row, ignore_index=True)

# Plot confusion matrix
plt.figure(figsize=(10, 8))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.show()

results_df
```

**Figure 9: ANN implementation in the same pipeline**

After the initial implementation of models on an imbalanced dataset with all the features in it, we will perform feature selection to check its effect on model performance. For this purpose, a random forest is used to create a feature importance plot as shown in Figure

10. to check features that are important and which features can be removed due to their lesser contribution to the prediction.

```
model = RandomForestClassifier(random_state=88)

pipeline = Pipeline([
    ("scaler", StandardScaler()),
    ("model", model)
])

pipeline.fit(X_train, y_train)

# Extracting feature importances
importances = pipeline.named_steps['model'].feature_importances_
feature_names = np.array(X.columns)
indices = np.argsort(importances)[::-1]

plt.figure(figsize=(12, 6))
plt.title("Feature Importances (Random Forest)")
sns.barplot(x=importances[indices], y=feature_names[indices], palette='colorblind')
plt.xlabel('Relative Importance')
plt.ylabel('Features')
plt.xticks(rotation=90)
plt.show()
```

**Figure 10: Feature Importance using Random Forest**

After checking feature importance, we found only one feature that was not contributing to the output, so it was removed as shown in Figure 11.

```
X = X.drop(columns=['pkts_sent'])
```

**Figure 11: Feature Elimination**

Data imbalance is handled in Figure 12. using random oversampling that oversampled the minority class instances and made the dataset balanced.

```
over_sampler = RandomOverSampler(random_state=88)
X_resampled, y_resampled = over_sampler.fit_resample(X, y)

# class distribution after oversampling
class_distribution = pd.Series(y_resampled).value_counts()
print("Class distribution after oversampling:")
print(class_distribution)
```

**Figure 12: Oversampling Minority classes using Random Oversampling**

After data balance, Figure 13 shows the balanced data set is again split into train, and test sets. The same model pipelines from Figures 6, 7, 8 and 9 are applied to the balanced dataset again. However, this time their results are saved into the new ‘results_df2’ data frame to separate balanced data results from imbalanced ones.

```
X_train, X_test, y_train, y_test = train_test_split(

    X_resampled, y_resampled, test_size = 0.3, random_state = 88
)

print("X_train shape:", X_train.shape)
print("y_train shape:", y_train.shape)
print("X_test shape:", X_test.shape)
print("y_test shape:", y_test.shape)
```

**Figure 13: Train Test Split after data balance**

For the final Analysis, hyperparameter tuning of all models is performed as shown in Figures 14 and 15 to find the best parameter for each model and their results are saved into a third data frame.

```
models = {
    'Logistic Regression': (LogisticRegression(random_state=88, max_iter=10000), {'C': [0.1, 1, 10]}),
    'Naive Bayes': (GaussianNB(), {}),
    'Random Forest': (RandomForestClassifier(random_state=88), {'n_estimators': [10, 100], 'max_depth': [None, 10, 20]}),
    'XGBoost': (XGBClassifier(random_state=88), {'n_estimators': [10, 100], 'max_depth': [3, 5, 7]}),
    'KNN': (KNeighborsClassifier(), {'n_neighbors': np.arange(3, 30, 2)}),
    'SVM': (SVC(random_state=88), {'kernel': ['rbf', 'poly'], 'C': [0.1, 1, 10]}),
}

class_names = ['allow', 'drop', 'deny', 'reset-both']

#  storing results
results_df3 = pd.DataFrame(columns=[
    'Model', 'Training Accuracy', 'Test Accuracy', 'Recall (Macro)', 'Precision (Macro)', 'F1 Score (Macro)',
    *[f'Class {class_name}' for class_name in class_names]
])

# Iterate models
for name, (model, params) in models.items():
    #  StandardScaler
    pipeline = Pipeline([
        ("scaler", StandardScaler()),
        ("model", model)
    ])

    #  find best hyperparameters
    grid_search = GridSearchCV(pipeline, param_grid={'model__' + k: v for k, v in params.items()}, cv=5, n_jobs=-1)

    # Fit on train data
    grid_search.fit(X_train, y_train)

    # prediction on test data
    y_pred = grid_search.predict(X_test)

    # evaluation metrics
    test_accuracy = accuracy_score(y_test, y_pred)
    recall = recall_score(y_test, y_pred, average='macro')
    precision = precision_score(y_test, y_pred, average='macro')
    f1 = f1_score(y_test, y_pred, average='macro')
    train_accuracy = accuracy_score(y_train, grid_search.predict(X_train))
    cm = confusion_matrix(y_test, y_pred)
    class_specific_accuracy = cm.diagonal() / cm.sum(axis=1)

    # Print
    print(f"Model: {name}")
    print(f"Best Parameters: {grid_search.best_params_}")
    print(f"Training Accuracy: {train_accuracy}")
    print(f"Test Accuracy: {test_accuracy}")
    print(f"Recall (Macro): {recall}")
    print(f"Precision (Macro): {precision}")
    print(f"F1 Score (Macro): {f1}")
    for i, class_name in enumerate(class_names):
        print(f"Class-specific accuracy for {class_name}: {class_specific_accuracy[i]}")
    print()

    # save to DataFrame
    new_row = {
        'Model': name,
        'Training Accuracy': train_accuracy,
        'Test Accuracy': test_accuracy,
        'Recall (Macro)': recall,
        'Precision (Macro)': precision,
        'F1 Score (Macro)': f1,
        **{f'Class {class_name}': class_specific_accuracy[i] for i, class_name in enumerate(class_names)}
    }
    results_df3 = results_df3.append(new_row, ignore_index=True)

    #confusion matrix
    plt.figure(figsize=(12, 6))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', cbar=False, xticklabels=class_names, yticklabels=class_names)
    plt.xlabel('Predicted labels')
    plt.ylabel('True labels')
    plt.title(f'Confusion Matrix for {name}')
    plt.show()
#results
print("Results DataFrame:")
print(results_df3)
```

**Figure 14: Hyperparameter Tuning Using Grid Search**

```python
#  ANN model
def create_ann(optimizer='adam', init='glorot_uniform', neurons=32, layers=2, **kwargs):
    model = Sequential()
    model.add(Dense(neurons, input_dim=X_train.shape[1], kernel_initializer=init, activation='relu'))
    for _ in range(layers - 1):
        model.add(Dense(neurons, kernel_initializer=init, activation='relu'))
    model.add(Dense(len(class_names), kernel_initializer=init, activation='softmax'))
    model.compile(loss='sparse_categorical_crossentropy', optimizer=optimizer, metrics=['accuracy'])
    return model

# Wrap model using KerasClassifier
ann = KerasClassifier(model=create_ann, verbose=0)

# hyperparameter grid
param_grid = {
    'model__optimizer': ['adam', 'rmsprop'],
    'model__init': ['glorot_uniform', 'normal'],
    'model__neurons': [32, 64, 128],
    'model__layers': [2, 3],
    'model__epochs': [50, 100],
    'model__batch_size': [32, 64]
}

# GridSearchCV finding hyperparameters
kfold = StratifiedKFold(n_splits=3, shuffle=True, random_state=42)
grid_search = GridSearchCV(estimator=ann, param_grid=param_grid, n_jobs=-1, cv=kfold)

# training data
grid_search.fit(X_train, y_train)

# Prediction
y_pred = grid_search.predict(X_test)

# Evaluation metrics
test_accuracy = accuracy_score(y_test, y_pred)
recall = recall_score(y_test, y_pred, average='macro')
precision = precision_score(y_test, y_pred, average='macro')
f1 = f1_score(y_test, y_pred, average='macro')
train_accuracy = accuracy_score(y_train, grid_search.predict(X_train))
cm = confusion_matrix(y_test, y_pred)
class_specific_accuracy = cm.diagonal() / cm.sum(axis=1)
```

```python
# Printing
print(f"Model: ANN")
print(f"Best Parameters: {grid_search.best_params_}")
print(f"Training Accuracy: {train_accuracy}")
print(f"Test Accuracy: {test_accuracy}")
print(f"Recall (Macro): {recall}")
print(f"Precision (Macro): {precision}")
print(f"F1 Score (Macro): {f1}")
for i, class_name in enumerate(class_names):
    print(f"Class-specific accuracy for {class_name}: {class_specific_accuracy[i]}")
print()

# Saving result
new_row = {
    'Model': 'ANN',
    'Training Accuracy': train_accuracy,
    'Test Accuracy': test_accuracy,
    'Recall (Macro)': recall,
    'Precision (Macro)': precision,
    'F1 Score (Macro)': f1,
    **{f'Class {class_name}': class_specific_accuracy[i] for i, class_name in enumerate(class_names)}
}
results_df3 = results_df3.append(new_row, ignore_index=True)

# confusion matrix
plt.figure(figsize=(12, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', cbar=False, xticklabels=class_names, yticklabels=class_names)
plt.xlabel('Predicted labels')
plt.ylabel('True labels')
plt.title('Confusion Matrix for ANN')
plt.show()

# DataFrame
print("Results DataFrame:")
print(results_df3)
```

**Figure 15: Grid Search Tuning for ANN**

For the comparative evaluation of results from all three model pipeline implementations, their saved results are combined and a bar plot of F1-scores of these models from Imbalanced, balanced and Grid Search Tuned datasets is displayed to conclude the effect of data oversampling, feature selection and hyperparameter tuning on model performances as shown in Figure 16.

```
combined_df = pd.concat([results_df, results_df2, results_df3], keys=['Imbalanced', 'Balanced', 'GridSearch Tuned'])

combined_df = combined_df.reset_index(level=0).rename(columns={'level_0': 'Dataset'})

sns.set_palette("colorblind")

plt.figure(figsize=(14, 8))
sns.barplot(x='Model', y='F1 Score (Macro)', hue='Dataset', data=combined_df, ci=None)

plt.title('Comparison of F1 Score Imbalanced, Balanced, and Tuned Datasets', fontsize=16)
plt.xlabel('Model', fontsize=14)
plt.ylabel('F1 Score (Macro)', fontsize=14)
plt.xticks(rotation=45)

plt.legend(title='Dataset')
plt.show()
```

**Figure 16: Displaying results from all three model Implementations**

## 4.2 UNSW NB15 dataset

Some steps for this dataset are the same as the previous one, so instead of putting the code snippets again, figures from the IFD section will be referred to here again. Import the UNSW NB15 into Jupyter Notebook IDE using the pandas function to read a CSV file as shown in Figure 17.

```
df = pd.read_csv("C:\\Users\\hp\\Downloads\\UNSW_NB15_testing-set.csv")
df.shape
```

**Figure 17: Read the CSV file for IFD**

One column is removed during EDA as it is a column containing just index values in it, as shown in Figure 18.

```
# droping because it is just an index
df.drop(columns=['id'],inplace=True)
```

**Figure 18: Dropping index column**

After exploratory data analysis and dropping irrelevant columns, we performed label encoding of all the categorical variables in the dataset as shown in Figure 19.

```
label_encoders = {}
for column in df.select_dtypes(include=['object']).columns:
    label_encoders[column] = LabelEncoder()
    df[column] = label_encoders[column].fit_transform(df[column])
```

**Figure 19: Encoding Categorical Variables**

Data is split into train and test sets with a split ratio of 0.3, meaning the training set has 70% data and the training set with 30% data as shown in Figure 20.

## Vertical Splitting

```
X = df.drop("attack_cat",axis=1)
y = df["attack_cat"]
```

## Horizontal Splitting

```
X_train, X_test, y_train, y_test = train_test_split(X,y, test_size = 0.3, random_state = 32)

print("X_train shape:", X_train.shape)
print("y_train shape:", y_train.shape)
print("X_test shape:", X_test.shape)
print("y_test shape:", y_test.shape)
```

**Figure 20: Splitting Dataset into Train and Test Sets**

This dataset also had some infinite values and Nan values. The mean imputation method shown in Figure 5. Subsection 4.1 is used again to replace all such outliers, infinite and nan values in the dataset.

After this initial data preprocessing, the model-agnostic pipeline is created following the steps shown in Figures 21, 22 and 23. All models are implemented except for ANN as it is implemented separately. Feature scaling and model evaluation are also performed within this pipeline and results are saved into a data frame.

```
# models
models = [
    ("Logistic Regression", LogisticRegression(random_state=32, max_iter=10000)),
    ("Naive Bayes", GaussianNB()),
    ("Random Forest", RandomForestClassifier(random_state=32)),
    ("XGBoost", XGBClassifier(random_state=32)),
    ("KNN", KNeighborsClassifier()),
    ("SVM", SVC(random_state=32))
]

# class names
class_names = ['Normal', 'Backdoor', 'Analysis', 'Fuzzers', 'Shellcode',
               'Reconnaissance', 'Exploits', 'DoS', 'Worms', 'Generic']

# Initialize best_model to None
best_model = None
best_accuracy = 0.0

# List to store results
results = []
```

**Figure 21: Defining Models and Class Names for Pipeline**

```python
# Iterate over models and evaluate their performance
for name, model in models:
    # pipeline for feature scaling
    pipeline = Pipeline([
        ("scaler", StandardScaler()),
        ("model", model)
    ])
    # Fitting pipeline to train
    pipeline.fit(X_train, y_train)
    # prediction on training data
    X_pred = pipeline.predict(X_train)
    # prediction on test data
    y_pred = pipeline.predict(X_test)
    # accuracy score
    train_accuracy = accuracy_score(y_train, X_pred)
    accuracy = accuracy_score(y_test, y_pred)
    # recall score
    recall = recall_score(y_test, y_pred, average='macro')
    # precision score
    precision = precision_score(y_test, y_pred, average='macro')
    # F1 score
    f1 = f1_score(y_test, y_pred, average='macro')
    # confusion matrix
    cm = confusion_matrix(y_test, y_pred)
    # Calculate class-specific accuracy
    class_specific_accuracy = cm.diagonal() / cm.sum(axis=1)
    # Printing evaluation metrics
    print(f"Model: {name}")
    print(f"Training Accuracy: {train_accuracy}")
    print(f"Test Accuracy: {accuracy}")
    print(f"Recall (Macro): {recall}")
    print(f"Precision (Macro): {precision}")
    print(f"F1 Score (Macro): {f1}")
    for i, class_name in enumerate(class_names):
        print(f"Class {class_name}: {class_specific_accuracy[i]}")
    print()

    # Appending results to the list
    results.append({
        "Model": name,
        "Training Accuracy": train_accuracy,
        "Test Accuracy": accuracy,
        "Recall (Macro)": recall,
        "Precision (Macro)": precision,
        "F1 Score (Macro)": f1,
        **{f"Class {class_name}": class_specific_accuracy[i] for i, class_name in enumerate(class_names)}
    })
```

**Figure 22: Model Agnostic Pipeline**

```python
    # Append results to list
    results.append({
        "Model": name,
        "Training Accuracy": train_accuracy,
        "Test Accuracy": accuracy,
        "Recall (Macro)": recall,
        "Precision (Macro)": precision,
        "F1 Score (Macro)": f1,
        **{f"Class {class_name}": class_specific_accuracy[i] for i, class_name in enumerate(class_names)}
    })

    # best model
    if accuracy > best_accuracy:
        best_accuracy = accuracy
        best_model = pipeline

    # confusion matrix
    plt.figure(figsize=(12, 6))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', cbar=False, xticklabels=class_names, yticklabels=class_names)
    plt.xlabel('Predicted labels')
    plt.ylabel('True labels')
    plt.title(f'Confusion Matrix for {name}')
    plt.show()
print("Best Model:", best_model)
results_df = pd.DataFrame(results)
results_df
```

**Figure 23: Displaying best model, confusion matrices and results**

ANN is applied following the same steps as other models and previous dataset; it is applied separately as shown in Figure 24. to get clearer output as it runs on many epochs creating longer outputs.

```
# model
model = Sequential()
model.add(Dense(128, input_shape=(X_train.shape[1],), activation='relu'))
model.add(Dense(128, activation='relu'))
model.add(Dense(64, activation='relu'))
model.add(Dense(len(np.unique(y_train)), activation='softmax'))
# compiling
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
# Training
history = model.fit(X_train, y_train, epochs=50, batch_size=32, validation_split=0.2, verbose=1)
# testset
test_loss, test_accuracy = model.evaluate(X_test, y_test, verbose=0)
# Predictions
y_pred = model.predict(X_test)
y_pred_classes = np.argmax(y_pred, axis=1)
# Evaluation metrics
accuracy = accuracy_score(y_test, y_pred_classes)
recall = recall_score(y_test, y_pred_classes, average='macro')
precision = precision_score(y_test, y_pred_classes, average='macro')
f1 = f1_score(y_test, y_pred_classes, average='macro')
conf_matrix = confusion_matrix(y_test, y_pred_classes)
class_specific_accuracy = conf_matrix.diagonal() / conf_matrix.sum(axis=1)
# Print metrics
print(f"Model: ANN")
print(f"Training Accuracy: {history.history['accuracy'][-1]}")
print(f"Test Accuracy: {test_accuracy}")
print(f"Recall (Macro): {recall}")
print(f"Precision (Macro): {precision}")
print(f"F1 Score (Macro): {f1}")
for i, class_name in enumerate(class_names):
    if i < len(class_specific_accuracy):
        print(f"Class {class_name}: {class_specific_accuracy[i]}")
    else:
        print(f"Class {class_name}: N/A (index out of bounds)")
print()
# Store results
new_row = {
    'Model': 'ANN',
    'Training Accuracy': history.history['accuracy'][-1],
    'Test Accuracy': test_accuracy,
    'Recall (Macro)': recall,
    'Precision (Macro)': precision,
    'F1 Score (Macro)': f1,
    **{f'Class {class_name}': class_specific_accuracy[i] if i < len(class_specific_accuracy) else None for i,
       class_name in enumerate(class_names)}
}
results_df = results_df.append(new_row, ignore_index=True)
# confusion matrix
plt.figure(figsize=(10, 8))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=class_names, yticklabels=class_names)
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.show()
```

**Figure 24: ANN implementation in the same pipeline**

After the initial implementation of models on an imbalanced dataset with all the features in it, we will perform feature selection to check its effect on model performance. Random forest is used to create a feature importance plot as shown in Figure 10. of Subsection 4.1 to check features that are important and which features can be removed due to their lesser contribution to the prediction.

Since it is a large dataset, in Figure 25 for the oversampling of data we have taken fewer instances by importing the dataset again with a defined number of rows and dropped irrelevant columns as well. A full dataset can also be imported as well depending on the computation resources of a system.

```
nrows = 120000
df2 = pd.read_csv("C:\\Users\\hp\\Downloads\\UNSW_NB15_testing-set.csv", nrows=nrows)
# Again dropping columns Irrelevant
df2.drop(columns=['id'],inplace=True)
df2.shape
```

**Figure 25: Importing dataset again with less rows**

Based on the feature importance plot of the random forest, the following features in Figure 26 are dropped as they do not contribute to model training for making predictions.

```
df2 = df2.drop(columns=['is_ftp_login', 'ct_ftp_cmd', 'is_sm_ips_ports', 'trans_depth', 'dwin', 'swin', 'ct_flw_http_mthd',
                        'response_body_len', 'state', 'stcpb', 'dtcpb', 'dpkts', 'djit', 'sloss', 'spkts', 'dloss' ])
```

**Figure 26: Feature Elimination**

Class imbalance is handled by the oversampling of minority classes using the Random Over sampler as shown in Figure 27.

```
X = df2.drop("attack_cat",axis=1)
y = df2["attack_cat"]
```

```
over_sampler = RandomOverSampler(random_state=32)
X_resampled, y_resampled = over_sampler.fit_resample(X, y)

# class distribution after oversampling
class_distribution = pd.Series(y_resampled).value_counts()
print("Class distribution after oversampling:")
print(class_distribution)
```

**Figure 27: oversampling of minority classes**

**Figure 12: Oversampling Minority classes using Random Oversampling**

After data balancing, Figure 28 shows the balanced data set is again split into train, and test sets. Since we had imported the data again, the mean imputation method is shown in Figure 5. Subsection 4.1 is used again to replace all such outliers, infinite and nan values in the dataset and the same model pipelines from Figures 21, 22, 23 and 24 are applied to the balanced dataset again. However, this time their results are saved into the new ‘results_df2’ data frame to separate balanced data results from imbalanced ones.

```
X_train, X_test, y_train, y_test = train_test_split(

X_resampled, y_resampled, test_size = 0.3, random_state = 32)

print("X_train shape:", X_train.shape)
print("y_train shape:", y_train.shape)
print("X_test shape:", X_test.shape)
print("y_test shape:", y_test.shape)
```

**Figure 28: Train Test Split after data balance**

For the final Analysis, hyperparameter tuning of all models is performed as shown in Figures 29 and 30 to find the best parameter for each model and their results are saved into a third data frame.

```
models = {
    'Logistic Regression': (LogisticRegression(random_state=32, max_iter=10000), {'C': [0.1, 1, 10]}),
    'Naive Bayes': (GaussianNB(), {}),
    'Random Forest': (RandomForestClassifier(random_state=32), {'n_estimators': [10, 100], 'max_depth': [None, 10, 20]}),
    'XGBoost': (XGBClassifier(random_state=32), {'n_estimators': [10, 100], 'max_depth': [3, 5, 7]}),
    'KNN': (KNeighborsClassifier(), {'n_neighbors': np.arange(3, 30, 2)}),
    'SVM': (LinearSVC(random_state=32, max_iter=10000), {'C': [0.1, 1, 10]}),
}


class_names = ['Normal', 'Backdoor', 'Analysis', 'Fuzzers', 'Shellcode',
               'Reconnaissance', 'Exploits', 'DoS', 'Worms', 'Generic']

results_df3 = pd.DataFrame(columns=[
    'Model', 'Training Accuracy', 'Test Accuracy', 'Recall (Macro)', 'Precision (Macro)', 'F1 Score (Macro)',
    *[f'Class {class_name}' for class_name in class_names]
])

for name, (model, params) in models.items():
    # StandardScaler
    pipeline = Pipeline([
        ("scaler", StandardScaler()),
        ("model", model)
    ])

    # GridSearch find best parameters
    grid_search = GridSearchCV(pipeline, param_grid={'model__' + k: v for k, v in params.items()}, cv=5, n_jobs=-1)

    # training data
    grid_search.fit(X_train, y_train)

    # test data
    y_pred = grid_search.predict(X_test)

    # evaluation metrics
    test_accuracy = accuracy_score(y_test, y_pred)
    recall = recall_score(y_test, y_pred, average='macro')
    precision = precision_score(y_test, y_pred, average='macro')
    f1 = f1_score(y_test, y_pred, average='macro')

    train_accuracy = accuracy_score(y_train, grid_search.predict(X_train))

    cm = confusion_matrix(y_test, y_pred)
    class_specific_accuracy = cm.diagonal() / cm.sum(axis=1)

    print(f"Model: {name}")
    print(f"Best Parameters: {grid_search.best_params_}")
    print(f"Training Accuracy: {train_accuracy}")
    print(f"Test Accuracy: {test_accuracy}")
    print(f"Recall (Macro): {recall}")
    print(f"Precision (Macro): {precision}")
    print(f"F1 Score (Macro): {f1}")
    for i, class_name in enumerate(class_names):
        print(f"Class-specific accuracy for {class_name}: {class_specific_accuracy[i]}")
    print()

    new_row = {
        'Model': name,
        'Training Accuracy': train_accuracy,
        'Test Accuracy': test_accuracy,
        'Recall (Macro)': recall,
        'Precision (Macro)': precision,
        'F1 Score (Macro)': f1,
        **{f'Class {class_name}': class_specific_accuracy[i] for i, class_name in enumerate(class_names)}
    }
    results_df3 = results_df3.append(new_row, ignore_index=True)

    # confusion matrix
    plt.figure(figsize=(12, 6))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', cbar=False, xticklabels=class_names, yticklabels=class_names)
    plt.xlabel('Predicted labels')
    plt.ylabel('True labels')
    plt.title(f'Confusion Matrix for {name}')
    plt.show()

print("Results DataFrame:")
print(results_df3)
```

**Figure 29: Hyperparameter Tuning Using Grid Search**

```python
# ANN model
def create_ann(optimizer='adam', init='glorot_uniform', neurons=32, layers=2, **kwargs):
    model = Sequential()
    model.add(Dense(neurons, input_dim=X_train.shape[1], kernel_initializer=init, activation='relu'))
    for _ in range(layers - 1):
        model.add(Dense(neurons, kernel_initializer=init, activation='relu'))
    model.add(Dense(len(class_names), kernel_initializer=init, activation='softmax'))
    model.compile(loss='sparse_categorical_crossentropy', optimizer=optimizer, metrics=['accuracy'])
    return model

# Wrap KerasClassifier
ann = KerasClassifier(model=create_ann, verbose=0)

# parameters
param_grid = {
    'model__optimizer': ['adam', 'rmsprop'],
    'model__init': ['glorot_uniform', 'normal'],
    'model__neurons': [32, 64, 128],
    'model__layers': [2, 3],
    'model__epochs': [50, 100],
    'model__batch_size': [32, 64]
}

# GridSearch for best parameters
kfold = StratifiedKFold(n_splits=3, shuffle=True, random_state=42)
grid_search = GridSearchCV(estimator=ann, param_grid=param_grid, n_jobs=-1, cv=kfold)

# training data
grid_search.fit(X_train, y_train)

# test data
y_pred = grid_search.predict(X_test)

# Evaluation metrics
test_accuracy = accuracy_score(y_test, y_pred)
recall = recall_score(y_test, y_pred, average='macro')
precision = precision_score(y_test, y_pred, average='macro')
f1 = f1_score(y_test, y_pred, average='macro')

train_accuracy = accuracy_score(y_train, grid_search.predict(X_train))

cm = confusion_matrix(y_test, y_pred)
class_specific_accuracy = cm.diagonal() / cm.sum(axis=1)
```

```python
print(f"Model: ANN")
print(f"Best Parameters: {grid_search.best_params_}")
print(f"Training Accuracy: {train_accuracy}")
print(f"Test Accuracy: {test_accuracy}")
print(f"Recall (Macro): {recall}")
print(f"Precision (Macro): {precision}")
print(f"F1 Score (Macro): {f1}")
for i, class_name in enumerate(class_names):
    print(f"Class-specific accuracy for {class_name}: {class_specific_accuracy[i]}")
print()

# Save result
new_row = {
    'Model': 'ANN',
    'Training Accuracy': train_accuracy,
    'Test Accuracy': test_accuracy,
    'Recall (Macro)': recall,
    'Precision (Macro)': precision,
    'F1 Score (Macro)': f1,
    **{f'Class {class_name}': class_specific_accuracy[i] for i, class_name in enumerate(class_names)}
}
results_df3 = results_df3.append(new_row, ignore_index=True)

# confusion matrix
plt.figure(figsize=(12, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', cbar=False, xticklabels=class_names, yticklabels=class_names)
plt.xlabel('Predicted labels')
plt.ylabel('True labels')
plt.title('Confusion Matrix for ANN')
plt.show()

print("Results DataFrame:")
print(results_df3)
```

**Figure 30: Grid Search Tuning for ANN**

For the comparative evaluation of results from all three model pipeline implementations, their saved results are combined and a bar plot of F1-scores of these models

from Imbalanced, balanced and Grid Search Tuned datasets is displayed to conclude the effect of data oversampling, feature selection and hyperparameter tuning on model performances as shown in Figure 31.

```
combined_df = pd.concat([results_df, results_df2, results_df3], keys=['Imbalanced', 'Balanced', 'GridSearch Tuned'])
combined_df = combined_df.reset_index(level=0).rename(columns={'level_0': 'Dataset'})

sns.set_palette("colorblind")
plt.figure(figsize=(14, 8))
sns.barplot(x='Model', y='F1 Score (Macro)', hue='Dataset', data=combined_df, ci=None)
plt.title('Comparison of F1 Score Imbalanced, Balanced, and Tuned Datasets', fontsize=16)
plt.xlabel('Model', fontsize=14)
plt.ylabel('F1 Score (Macro)', fontsize=14)
plt.xticks(rotation=45)
plt.legend(title='Dataset')
plt.show()
```

**Figure 31: Displaying results from all three model Implementations**

## 4.3 NF-UQ-NIDS-v2 Dataset

Import the NF-UQ-NIDS-v2 Dataset into Jupyter Notebook IDE using the pandas function to read a CSV file as shown in Figure 32. For this dataset, 300,000 rows are taken since it contains millions of records, which would be difficult to process with the resources we have.

```
nrows = 300000
# first 300,000 rows
df = pd.read_csv("C:\\Users\\hp\\Downloads\\NF-UQ-NIDS-v2.csv", nrows=nrows)
df.shape
```

**Figure 32: Read the CSV file for IFD**

After importing the CSV, EDA is done and the following columns in Figure 33 are dropped as they consist of the IP addresses, which are better to remove to avoid any ethical concerns. Also dataset column is removed, which names different small datasets, combined to make this big data.

```
# drop these columns as they are just IP addresses
df.drop(columns=['IPV4_SRC_ADDR', 'IPV4_DST_ADDR', 'Dataset'],inplace=True)
```

**Figure 33: Dropping IP address and irrelevant columns**

After that, we performed label encoding of the target variable as shown in Figure 34.

```
label_encoder = LabelEncoder()
df["Attack"] = label_encoder.fit_transform(df["Attack"])
```

**Figure 34: Encoding Categorical Variables**

Data is split into train and test sets with a split ratio of 0.3, meaning the training set has 70% data and the training set with 30% data as shown in Figure 35.

```
X = df.drop("Attack",axis=1)
y = df["Attack"]
```

## Horizontal Splitting

```
X_train, X_test, y_train, y_test = train_test_split(  X,y, test_size = 0.3, random_state = 32)

print("X_train shape:", X_train.shape)
print("y_train shape:", y_train.shape)
print("X_test shape:", X_test.shape)
print("y_test shape:", y_test.shape)
```

**Figure 35: Splitting Dataset into Train and Test Sets**

This dataset also gave errors due to the presence of some infinite values in the dataset. The mean imputation method shown in Figure 5. of section 4.1 is used to replace all such outliers, infinite and nan values in the dataset.

After that dependent class names and models are defined to create the models pipeline as shown in Figure 36.

```
# models
models = [
    ("Logistic Regression", LogisticRegression(random_state=32, max_iter=10000)),
    ("Naive Bayes", GaussianNB()),
    ("Random Forest", RandomForestClassifier(random_state=32)),
    ("XGBoost", XGBClassifier(random_state=32)),
    ("KNN", KNeighborsClassifier()),
    ("SVM", SVC(random_state=32))
]

# class names
class_names = ['DoS', 'Benign', 'scanning', 'DDoS', 'xss', 'Bot',
               'Reconnaissance', 'password', 'Fuzzers', 'injection', 'Theft',
               'Brute Force', 'Infilteration', 'Exploits', 'Generic', 'Analysis',
               'Backdoor', 'mitm', 'Shellcode', 'ransomware']

# Initialize best_model to None
best_model = None
best_accuracy = 0.0

# store results
results = []
```

**Figure 36: Defining Models and Class Names for Pipeline**

After defining models and class names, the steps from Figures 22 and 23 of section 4.2 are applied to iterate models in the model's agnostic pipeline and save their evaluation into the defined data frames. Similarly, ANN is applied following the code from Figure 24 of section 4.2

ANN is applied following the same steps as other models; it is applied separately as shown in Figure 9. to get clearer output as it runs on many epochs creating longer outputs.

After checking the models on the imbalanced dataset with all the features in it, we will perform feature selection to check its effect on model performance. For this purpose, a random forest is used to create a feature importance plot as shown in Figure 37.

```
model = RandomForestClassifier(random_state=32)

pipeline = Pipeline([
    ("scaler", StandardScaler()),
    ("model", model)
])

pipeline.fit(X_train, y_train)

# feature importance
importances = pipeline.named_steps['model'].feature_importances_
feature_names = np.array(X.columns)
indices = np.argsort(importances)[::-1]

# Plot
plt.figure(figsize=(16, 9))
plt.title("Feature Importances (Random Forest)")
sns.barplot(x=importances[indices], y=feature_names[indices], palette='colorblind')
plt.xlabel('Relative Importance')
plt.ylabel('Features')
plt.xticks(rotation=90)
plt.show()
```

**Figure 37: Feature Importance using Random Forest**

For balancing the dataset, it is imported again in Figure 38 with fewer rows as after oversampling it would take a lot of time to process

```
nrows = 100000
df2 = pd.read_csv("C:\\Users\\hp\\Downloads\\NF-UQ-NIDS-v2.csv", nrows=nrows)
# Again dropping columns Irrelevant
df2.drop(columns=['IPV4_SRC_ADDR', 'IPV4_DST_ADDR', 'Dataset'],inplace=True)
df2.shape
```

**Figure 38: Importing Data for balancing**

In figure 39, noncontributing features identified by the feature importance plot of random forest are removed to avoid data redundancy and reduce the code execution time.

```
df2 = df2.drop(columns=['DST_TO_SRC_SECOND_BYTES', 'SRC_TO_DST_SECOND_BYTES', 'FTP_COMMAND_RET_CODE', 'RETRANSMITTED_IN_PKTS',
                        'RETRANSMITTED_OUT_PKTS', 'ICMP_TYPE', 'ICMP_IPV4_TYPE', 'RETRANSMITTED_OUT_BYTES',
                        'RETRANSMITTED_IN_BYTES', 'NUM_PKTS_1024_TO_1514_BYTES', 'DNS_TTL_ANSWER', 'NUM_PKTS_256_TO_512_BYTES',
                       'NUM_PKTS_512_TO_1024_BYTES'])
```

**Figure 39: Feature Elimination**

After that label encoding of the target variable is done again and data imbalance is handled in Figure 40. using the random oversampling technique that oversampled the classes with fewer instances and makes the dataset balanced.

```
over_sampler = RandomOverSampler(random_state=32)
X_resampled, y_resampled = over_sampler.fit_resample(X, y)

# distribution after oversampling
class_distribution = pd.Series(y_resampled).value_counts()
print("Class distribution after oversampling:")
print(class_distribution)
```

**Figure 40: Oversampling Minority classes using Random Oversampling**

After data balance, the balanced data set is again split into train, and test sets, infinite, Nan and outliers are removed from the newly imported partial dataset following Figure 5 of Subsection 4.1. The same model and class name from Figure 36 are defined again and

executed using the standard scaler pipelines from Figures 22 and 23 along with the ANN pipeline from Figure 24 are applied to the balanced dataset again. However, this time their results are saved into the new 'results_df2' data frame to separate balanced data results from imbalanced ones.

```
models = {
    'Logistic Regression': (LogisticRegression(random_state=32, max_iter=10000), {'C': [0.1, 1, 10]}),
    'Naive Bayes': (GaussianNB(), {}),
    'Random Forest': (RandomForestClassifier(random_state=32), {'n_estimators': [10, 100], 'max_depth': [None, 10, 20]}),
    'XGBoost': (XGBClassifier(random_state=32), {'n_estimators': [10, 100], 'max_depth': [3, 5, 7]}),
    'KNN': (KNeighborsClassifier(), {'n_neighbors': np.arange(3, 30, 2)}),
    'SVM': (SVC(random_state=32), {'kernel': ['rbf', 'poly'], 'C': [0.1, 1, 10]}),
}


class_names = ['DoS', 'Benign', 'scanning', 'DDoS', 'xss', 'Bot',
               'Reconnaissance', 'password', 'Fuzzers', 'injection', 'Theft',
               'Brute Force', 'Infilteration', 'Exploits', 'Generic', 'Analysis',
               'Backdoor', 'mitm', 'Shellcode', 'ransomware']

#  store results
results_df3 = pd.DataFrame(columns=[
    'Model', 'Training Accuracy', 'Test Accuracy', 'Recall (Macro)', 'Precision (Macro)', 'F1 Score (Macro)',
    *[f'Class {class_name}' for class_name in class_names]
])

for name, (model, params) in models.items():
    #  StandardScaler
    pipeline = Pipeline([
        ("scaler", StandardScaler()),
        ("model", model)
    ])

    # GridSearchCV find best hyperparameters
    grid_search = GridSearchCV(pipeline, param_grid={'model__' + k: v for k, v in params.items()}, cv=5, n_jobs=-1)

    grid_search.fit(X_train, y_train)

    y_pred = grid_search.predict(X_test)

    # evaluation metrics
    test_accuracy = accuracy_score(y_test, y_pred)
    recall = recall_score(y_test, y_pred, average='macro')
    precision = precision_score(y_test, y_pred, average='macro')
    f1 = f1_score(y_test, y_pred, average='macro')

    # accuracy using the best estimator
    train_accuracy = accuracy_score(y_train, grid_search.predict(X_train))

    #  confusion matrix
    cm = confusion_matrix(y_test, y_pred)
    class_specific_accuracy = cm.diagonal() / cm.sum(axis=1)

    print(f"Model: {name}")
    print(f"Best Parameters: {grid_search.best_params_}")
    print(f"Training Accuracy: {train_accuracy}")
    print(f"Test Accuracy: {test_accuracy}")
    print(f"Recall (Macro): {recall}")
    print(f"Precision (Macro): {precision}")
    print(f"F1 Score (Macro): {f1}")
    for i, class_name in enumerate(class_names):
        print(f"Class-specific accuracy for {class_name}: {class_specific_accuracy[i]}")
    print()

    # save results
    new_row = {
        'Model': name,
        'Training Accuracy': train_accuracy,
        'Test Accuracy': test_accuracy,
        'Recall (Macro)': recall,
        'Precision (Macro)': precision,
        'F1 Score (Macro)': f1,
        **{f'Class {class_name}': class_specific_accuracy[i] for i, class_name in enumerate(class_names)}
    }
    results_df3 = results_df3.append(new_row, ignore_index=True)

    #  confusion matrix
    plt.figure(figsize=(12, 6))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', cbar=False, xticklabels=class_names, yticklabels=class_names)
    plt.xlabel('Predicted labels')
    plt.ylabel('True labels')
    plt.title(f'Confusion Matrix for {name}')
    plt.show()

print("Results DataFrame:")
print(results_df3)
```

**Figure 41: Hyperparameter Tuning Using Grid Search**

For the final Analysis, hyperparameter tuning of all models is performed as shown in Figures 41 and 42 to find the best parameter for each model and their results are saved into a third data frame.

```
def create_ann(optimizer='adam', init='glorot_uniform', neurons=32, **kwargs):
    model = Sequential()
    model.add(Dense(neurons, input_dim=X_train.shape[1], kernel_initializer=init, activation='relu'))
    model.add(Dense(neurons, kernel_initializer=init, activation='relu'))
    model.add(Dense(len(class_names), kernel_initializer=init, activation='softmax'))
    model.compile(loss='sparse_categorical_crossentropy', optimizer=optimizer, metrics=['accuracy'])
    return model

# Wrap KerasClassifier
ann = KerasClassifier(model=create_ann, verbose=0)

# hyperparameter grid
param_grid = {
    'model__optimizer': ['adam', 'rmsprop'],
    'model__init': ['glorot_uniform', 'normal'],
    'model__neurons': [32, 64, 128],
    'model__epochs': [50, 100],
    'model__batch_size': [32, 64]
}

# GridSearch
grid_search = GridSearchCV(estimator=ann, param_grid=param_grid, n_jobs=-1, cv=5)

grid_search.fit(X_train, y_train)

y_pred = grid_search.predict(X_test)

test_accuracy = accuracy_score(y_test, y_pred)
recall = recall_score(y_test, y_pred, average='macro')
precision = precision_score(y_test, y_pred, average='macro')
f1 = f1_score(y_test, y_pred, average='macro')

train_accuracy = accuracy_score(y_train, grid_search.predict(X_train))

# Confusion matrix
cm = confusion_matrix(y_test, y_pred)
class_specific_accuracy = cm.diagonal() / cm.sum(axis=1)

print(f"Model: ANN")
print(f"Best Parameters: {grid_search.best_params_}")
print(f"Training Accuracy: {train_accuracy}")
print(f"Test Accuracy: {test_accuracy}")
print(f"Recall (Macro): {recall}")
print(f"Precision (Macro): {precision}")
print(f"F1 Score (Macro): {f1}")
for i, class_name in enumerate(class_names):
    print(f"Class-specific accuracy for {class_name}: {class_specific_accuracy[i]}")
print()

# Save results
new_row = {
    'Model': 'ANN',
    'Training Accuracy': train_accuracy,
    'Test Accuracy': test_accuracy,
    'Recall (Macro)': recall,
    'Precision (Macro)': precision,
    'F1 Score (Macro)': f1,
    **{f'Class {class_name}': class_specific_accuracy[i] for i, class_name in enumerate(class_names)}
}
results_df3 = results_df3.append(new_row, ignore_index=True)

# confusion matrix
plt.figure(figsize=(12, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', cbar=False, xticklabels=class_names, yticklabels=class_names)
plt.xlabel('Predicted labels')
plt.ylabel('True labels')
plt.title('Confusion Matrix for ANN')
plt.show()

# updated DF
print("Results DataFrame:")
print(results_df3)
```

**Figure 42: Grid Search Tuning for ANN**

For the comparative evaluation of results from all three model pipeline implementations, their saved results are combined and a bar plot of F1-scores of these models

from Imbalanced, balanced and Grid Search Tuned datasets is displayed to conclude the effect of data oversampling, feature selection and hyperparameter tuning on model performances as shown in Figure 16.

```
combined_df = pd.concat([results_df, results_df2, results_df3], keys=['Imbalanced', 'Balanced', 'GridSearch Tuned'])
combined_df = combined_df.reset_index(level=0).rename(columns={'level_0': 'Dataset'})

sns.set_palette("colorblind")
plt.figure(figsize=(14, 8))
sns.barplot(x='Model', y='F1 Score (Macro)', hue='Dataset', data=combined_df, ci=None)
plt.title('F1 Score for Imbalanced, Balanced & Tuned-Model Datasets', fontsize=16)
plt.xlabel('Model', fontsize=14)
plt.ylabel('F1 Score (Macro)', fontsize=14)
plt.xticks(rotation=45)
plt.legend(title='Dataset', loc='center left', bbox_to_anchor=(1, 0.5))
plt.show()
```

**Figure 43: Displaying results from all three model Implementations**

For the final analysis of these model agnostics pipelines on the three datasets, you can compare the results obtained from Figure 16 of section 4.1, Figure 31 of section 4.2 and Figure 43 of section 4.3.

# References

Moustafa, N. and Slay, J., 2015. UNSW-NB15: A Comprehensive Data Set for Network Intrusion Detection Systems (UNSW-NB15 Network Data Set). *Proceedings of the 2015 Military Communications and Information Systems Conference (MilCIS)*. [online] IEEE Xplore. Available at: https://doi.org/10.1109/MilCIS.2015.7348942.