User Guide

Machine Learning and Explainable AI for Cybersecurity

Table of Contents

1. Introduction

The use of Machine Learning in workplaces and research is continuously growing with the usual goal of having a high accuracy in predictions without much consideration into how they come to the decision. Some models such as Decision Trees offer insight into how they generate their predictions using different features through leaf nodes. However, as we expand this concept into ensemble methods such as Random Forests, and evaluating certain groups within the predictions, the decisions become harder to interpret. This concept also translates to Neural Networks where dense layers are used to cross-communicate many different nodes and apply activation functions. For the most part users are content with seeing a gradual decrease in loss which is the model uncertainty for a certain label, but what about how the different features contribute to this result? By using Explainable AI techniques such as Shapley Additive Explanations (SHAP), users can evaluate the model decision making process and gain a deeper understanding into feature interactions. This becomes particularly useful in domains such as cybersecurity where the different feature contributions give insight into the characteristics of threats. This document is created for the users that may be interested in knowing how to leverage preprocessing techniques, machine learning models, and Explainable AI with SHAP for threat detection.

2. Overview

2.1 Models and Preprocessing Techniques

The use case for machine learning models varies depending on the computational infrastructure used and the type of data that needs to be inferred. In general, for cybersecurity data sets, it is important to try and predict all threat labels correctly and reduce the false positives for normal labels. It is also beneficial to try and include as many features as possible for further evaluation.

Neural Networks are powerful machine learning models that can be customised and scaled extensively. However, one of the weaknesses is their need for continuous features to be scaled and categorical features to be one-hot encoded. This creates some challenges especially with intrusion detection datasets. Firstly, when scaling features the true nature of the numbers is lost in the evaluation phase where the values within the features can only be interpreted as greater or lower with respect to themselves rather than actual points. Noise is also an issue; the data is generally highly imbalanced where most rows are collections of normal network traffic. Some of these normal values can be very dissimilar to the general data points (outliers). By using a cluster visualisation tool such as DB-Scan, scaling techniques can be compared for the normal data so that the appropriate transformation can be selected to try and reduce noise and misclassification between normal labels and threat labels. Another issue which is particularly challenging is one-hot encoding of features. Some features such as ports have over 50, 000 unique values and for a dataset that has 2.5 million rows this becomes cumbersome. When considering that there is destination and source ports, the total number of values in a matrix exceed 250 billion just for these two features alone. A useful tactic to overcome this issue is to only one-hot encode the values within the feature that share a certain level of correlation to the label. This can be done by encoding each value within the feature and drawing the Phi coefficient to the label then keeping it only if it exceeds a given threshold.

Tree models are much more robust than Neural Networks and require very little pre-processing which in turn provides much higher explainability in the evaluation phase. This is particularly useful for cybersecurity datasets that have high cardinalities in categorical features. Tree models do not use distance-based measures to make a prediction for a label which makes them able to natively handle simple label encoding of categories and omit the need for scaling of continuous features. The models are also robust to issues such as noise in the data and in general can achieve very high accuracy on tabular data without the need of extensive optimisation. Additionally, the models can be extended through ensemble methods by increasing the number of estimators which create many trees and choose the best split based across multiple trees. However, this robustness also brings challenges where sampling techniques or using weights have a minimal impact in enforcing the model to prioritise minority labels.

By combining the Neural Networks and Tree Models, an ensemble can be created where a sampling strategy such as synthetic minority oversampling (SMOTE) is used on the Neural Network to predict all threats correctly, then a test set is generated for the Tree Model to classify the rest of the labels. This is an effective method for intrusion detection datasets as detecting threats should be a priority but classifying the type of threat is also important. Since the remaining data is known to contain some of the normal labels, the multi-class model can be used to gain further insight into where the misclassifications are occurring. Table 1 shows an example of a multi-class output for normal labels and threats. The first noticeable thing is that there are a lot more normal labels which shows there is a class imbalance problem. We find that there are a high number of DoS and Exploits misclassifications, but it is amongst themselves. This means that they are still being classified as threats but have uncertainty as to whether it is an Exploit or DoS attack. Meanwhile the most misinterpreted threat is actually Fuzzers where a large number is being classified as normal traffic.

Table 1: A confusion matrix for a multi-class Machine Learning model on different network threats. Values are created for the purpose of an example and may not depict real-world interactions between threats and normal network traffic.

Multi-Class Classifications on Threat Detection Data				
Label	Normal	DoS	Exploits	Fuzzers
Normal	3510	1	2	78
DoS	0	445	351	2
Exploits	1	378	423	1
Fuzzers	122	0	0	954
Total	3633	824	776	1035

This information can be used to further tune the model using either probability scores or grouping. For example, DoS and Exploits could be grouped together for the purpose of simply reducing false positives overall. Or a probability could be used such as if it is a Fuzzers and only has 60% certainty along with certain characteristics amongst its features, it could be a normal label.

2.2 Explainability of Models

Understanding why a cybersecurity model makes a particular prediction is just as critical as the prediction itself. Explainability techniques bridge this gap by providing insights into the factors driving model outputs. SHAP works by attributing the impact of each feature on a model's prediction using a game theory approach. Each attribution is calculated by evaluating a feature’s impact across all possible subsets of the other features. These marginal impacts are then averaged to produce the Shapley value, which represents the final attribution score for that feature. The models explained using SHAP are the Neural Network as the primary model, which includes both the standard and up sampled data versions The secondary model is an XG-Boost multi-class classifier and unlike PyTorch based models, is extremely efficient in calculating the Shapley values. The data incorporates one-hot encoded features in both cases which need to be interpreted with care due to the way the values are represented where a 1 indicates the presence of the variable and 0 as the absence of the variable. Because of this, SHAP portrays these variables as either a high value for 1 or low value for 0. The goal is to use SHAP to help illuminate the feature contributions and interactions in these models.

3. Tools

This section will provide a detailed explanation of the custom Python classes developed to streamline the preprocessing, modelling, and explainability workflows for cybersecurity data. Each class is designed to encapsulate specific functionalities, making the process more modular and user-friendly. We will cover the purpose, methods, and usage of each class in the subsequent subsections.

3.1 Encoders

Encoding is a crucial step in preparing data for machine learning models. Since most algorithms require numerical input, categorical features that have string values must be converted into a numerical format. While widely used techniques like label encoding and standard one-hot encoding are effective for many datasets, they face significant challenges with high-cardinality categorical features, which are common in cybersecurity data. For instance, a standard one-hot encoding approach of a feature with thousands of unique values can create an excessively wide dataset, leading to substantial memory consumption and increased computational time. To address these limitations and ensure the encoded features are both manageable and relevant, this project employs a strategic approach to encoding. In general, the project's work with encoders covers the following key aspects:

- **Prioritized Handling of High-Cardinality:** Recognising that features such as ports often have numerous unique values, a core focus is placed on efficiently encoding these without overwhelming the system.
- **Relevance-Driven Feature Selection:** Instead of a brute-force approach, the project utilises methods that actively select which category values to encode based on their relationship (correlation) with the target variable.
- **Operational Efficiency:** The calculations are run through tensor-based operations ensuring efficiency during the encoding process.
- **Integration with Data Preparation Pipeline:** The process is integrated into the overall data preprocessing pipeline, ensuring categorical features are appropriately handled before being fed into a Neural Network.

3.1.1 Correlation One-hot Encoder

The Correlation One Hot Encoder is a custom encoding technique specifically designed to address the challenges posed by high-cardinality categorical features. This method intelligently selects and encodes the most relevant categories based on their statistical association with the binary target variable. For a binary classification task, it calculates the Phi coefficient, a measure of association for two binary variables between each unique value within a categorical feature and the binary threat label. By focusing on category values with a significant correlation and incorporating parameters to handle sparsity and limit the number of generated features, this encoder effectively manages dimensionality and prioritises features with higher predictive value. This approach is particularly beneficial for features like ports where standard encoding would be computationally prohibitive due to the sheer number of unique values.

3.1.2 Correlation One-hot Encoder Example Usage

Table 2 provides a practical example of utilising the encoder class on the USNW-NB15 dataset. The process begins with the initial data loading, then the categorical columns are identified based on the cardinality of unique values. For each of these features, a separate instance of the encoder is created. After initialisation, the core encoding process is executed using the encode method.

Initialisation:

- **CorrOnehotEncoder(column, target):**

  The initialisation of the encoder object using the following parameters:

  column: The column from the data where the values are drawn and encoded. Each value within the category column is converted to a binary for the calculation.

  target: The binary target label to which the correlation is measured with. Since the absolute correlation is measured, it does not matter how the positive and negative value is mapped.

Encoding:

- **encode(sparse_n, threshold, max_encoded):**

  The encoding function which runs the encoding process for the given variable and target columns with the following parameters:

  sparse_n: This parameter sets the minimum number of occurrences a unique category value must have in the dataset to be considered for encoding. A lower value here can capture a bigger pool of relevant features, especially when working with sampled data.

threshold: The minimum absolute correlation value (Phi) the within variable value must have with the target variable to be encoded. Values that fall below the threshold are skipped.

max_encoded: This parameter limits the total number of encoded features generated by selecting the top-n most correlated values after applying the sparsity and threshold filters.

Table 2: Example code of using the One-hot Correlation Encoder class from initialisation to running the encoder function with parameters for different categorical features.

Correlation One-Hot Encoder

```
# Import the correlation encoder.
from Tools.encoders import CorrOnehotEncoder

# Print the docstring.
help(CorrOnehotEncoder)

# Parameters:
# The minimum number of 1's in the value (deals with sparse data).
sparse_n = 600

# The threshold for the minimum correlation.
threshold = 0.1

# The limit for the encoded features. The highest correlated values are
sorted (highest to lowest) and the maximum is selected.
max_encoded = 30

# Encode the variables.
ce = CorrOnehotEncoder(data['srcip'], data['label'])
ohe1 = ce.encode(sparse_n, threshold, max_encoded)
ce = CorrOnehotEncoder(data['sport'], data['label'])
ohe2= ce.encode(sparse_n, threshold, max_encoded)
ce = CorrOnehotEncoder(data['dstip'], data['label'])
ohe3 = ce.encode(sparse_n, threshold, max_encoded)
ce = CorrOnehotEncoder(data['dsport'], data['label'])
ohe4 = ce.encode(sparse_n, threshold, max_encoded)
ce = CorrOnehotEncoder(data['proto'], data['label'])
ohe5 = ce.encode(sparse_n, threshold, max_encoded)
ce = CorrOnehotEncoder(data['state'], data['label'])
ohe6 = ce.encode(sparse_n, threshold, max_encoded)
ce = CorrOnehotEncoder(data['service'], data['label'])
ohe7 = ce.encode(sparse_n, threshold, max_encoded)
```

The output of the encode method for each feature is a new DataFrame that contains the one-hot encoded binary columns. These resulting DataFrames are then ready to be concatenated with the rest of the features in the subsequent steps of the pipeline. This targeted encoding approach significantly reduces the dimensionality compared to a standard one-hot encoding method, making the dataset more manageable and the modelling process more efficient.

3.2 Cluster Visualisers

The cluster visualisation tool provides a way to view within label data in a 3D plot with either K-means++ clustering or Density-Based Spatial Clustering of Applications with Noise (DB-Scan) from the sci-kit learn library. When it comes to network data such as USNW-NB15, most of the data is normal network traffic. Since there is so much of it, it is expected that there may be rows that are very similar to each other. However, there are some rare cases (outliers or noise) where some points differ significantly and could potentially be misclassified as threat. Scaling is necessary for Neural Networks and generally leads to a better prediction and faster training at the expense of losing the distance measure between points in data. The visualisation tool allows a user to see the effect of the scaling technique on continuous variables in the data for a label. Plots can be generated with K-means++ or DB-Scan to view clusters and evaluate which scaling technique can reduce noise in the data more effectively and in turn lead to a better separation between threats and normal traffic. The clustering tool visualisations are projected on a 3D plane using either t-distributed Stochastic Neighbour Embedding (t-SNE projection) or three principal components (PCA).

The cluster visualisers can be facilitated in several ways including:

- **Unsupervised Grouping of Data:** Unsupervised machine learning can be used to find distinct groups within the data for continuous variables. The clusters can be appended back to the data to evaluate where the within label groups differ.
- **Anomaly Detection:** Evaluation can be done using DB-Scan to find data that differs significantly from within label groups. By setting an epsilon threshold the algorithm can detect outliers. This is particularly useful in selecting scaling techniques or to find potentially risky network traffic that is significantly different from normal traffic.
- **Scaling Selection:** Using the clustering techniques an optimal scaler can be selected for machine learning models. A comparison can easily be made by visualising the different scalers and evaluating the clusters. Reducing noise in normal traffic data is particularly helpful as it reduces the likelihood normal traffic being misclassified as a threat by a model.

- **Dynamic Cluster Selection:** Clusters can be created in a variety of ways using K-means++ and DB-Scan with either t-SNE or PCA dimensionality reduction techniques.

The following sections will outline the implementation of using the tool to create K-means++ and DB-Scan clusters in 3D using t-SNE.

3.2.1 t-SNE with K-means++

K-means++ generates clusters based on the Euclidian distance between the data rows and the created centroids. It forces the data into as many clusters as the number of centroids specified by k. This tool is effective in evaluating points that are closest to each other but faces a few issues as each point must be assigned to a centroid. This means that points that don't share a high similarity to a centroid can potentially be grouped to a centroid even if it is not very similar. However, using the visualisation, this can be evaluated based on the proximity of the point and the colour of the cluster. The tool automatically projects the clusters in a 3D plane using t-SNE.

3.2.2 t-SNE with K-means++ Example Usage

Table 3 provides an example of implementing t-SNE with K-means++ on the USNW-NB15 dataset using the cluster visualiser. For K-means++ to work efficiently only data that is continuous or ordinal can be evaluated which need to be filtered into a separate Data Frame. The columns which have more than a given number of unique values are collected. Here we use 30 but it is recommended to lower this number to 10. First the visualiser class is initialised with the data, a sample size, and the scaler type. Then the data is visualised using the plot function specifying the number of clusters and K-means++ as the cluster type.

Initialisation:

- **PlotClusters(data, scaler_type, sample_size):**

  The initialisation of the visualiser object using the following parameters:

  data: The filtered dataset to visualise.

  scaler_type: The scaling technique to use (either 'standard' or 'minmax').

sample_size: Sample size of the data to reduce calculation times. t-SNE is computationally expensive, and it becomes impractical to compute on very large datasets. The sample size should be large enough to truly represent the data.

Visualising:

- **plot_tsne(apply_pca, n_clusters, lr, perplexity, cluster_type):**

  The plotting function for K-Means++ using t-SNE with the following parameters:

  apply_pca: Whether to apply PCA transformation. This is generally not recommended unless wanting to reduce the computation time for large sample sizes.

  n_clusters: Specifies the number of clusters that K-means++ is forced to create. Centroids are calculated and each point in the space is grouped to the closest centroid based on Euclidean distance.

  lr: The rate at which the positions are adjusted in 3D space during optimisation in t-SNE. A lower learning rate does a better job at projecting the points correctly however it becomes very computationally expensive.

  perplexity: Impacts the shape of the clusters in the 3D space in t-SNE. Low perplexity generally forms tight local clusters whereas higher perplexity can at times form sparse clusters.

  cluster_type: Set to ‘kmeans’ for K-means++ cluster evaluation.

Table 3: Example code of using the t-SNE with K-Means++ from preparing data to initialisation then running the visualisation function with the parameters.

t-SNE K-means++ Evaluation on Normal Rows

```
# Create a new dataset with just normal labels and make sure to drop the
categorical columns from before.
cluster_data = data.drop(columns=categories)

# Now we set it to just Normal labels.
cluster_data = cluster_data[cluster_data['attack_cat'] == 'Normal']

# Filter for mostly continuous variables.
remove = []
for i in cluster_data:
    if len(cluster_data[i].unique()) < 30:
        remove.append(i)
# Drop the varibles collected.
cluster_data = cluster_data.drop(columns=remove)
```

```python
# Import the cluster visualiser and read doc.
from Tools.cluster_visualiser import PlotClusters
# Read the docs.
help(PlotClusters)

# Use Min-Max scaling data.
scaler = 'minmax'

# Select sample size. A limit of 10000 is recommended as TSNE is
computationally expensive.
sample_size = 10000

# Create class.
pc = PlotClusters(cluster_data, scaler, sample_size)

# Ommit PCA conversion (this False by default).
apply_pca = False

# Select number of clusters for K-means++
n_clusters = 3

# Select K-means type clusters.
cluster_type = 'kmeans'

# Run the plot function for the class with the given parameters.
pc.plot_tsne(n_clusters, cluster_type)
```

3.2.3 t-SNE with DB-Scan

DB-Scan is an anomaly detection tool which helps to identify outliers and noise using Euclidean distance between sets. Rows that are very dissimilar to most other data are classed as noise. This is controlled by the epsilon of the ring as the threshold that is formed around the clusters. Data that falls outside of the epsilon are marked as outliers. DB-Scan automatically finds the optimal number of clusters and can be useful in identifying the correct scaling technique to use for reducing noise, removing outliers, or identifying the main groups for within label data. The data is projected into a 3D space using either t-SNE or three PCA components to allow for evaluation of the clusters.

3.2.4 t-SNE with DB-Scan Example Usage

Table 4 shows how t-SNE and DB-Scan can be deployed to select the correct scaling technique for a Neural Network using the same sample as the K-means++ implementation. The data is set to a specific label which in this case is normal, then filtered to only include ordinal and continuous variables. The visualiser class is then initialised with the data, a sample size, and the scaler type. Then the data is visualised using the plot function specifying the number of DB-Scan as the cluster type.

Initialisation:

- **PlotClusters(data, scaler_type, sample_size):**

  The initialisation of the visualiser object using the following parameters:

  data: The filtered dataset to visualise.

  scaler_type: The scaling technique to use (either 'standard' or 'minmax').

  sample_size: Resample size of the data to reduce calculation times. t-SNE is computationally expensive, and it becomes impractical to compute on very large datasets. The sample size should be large enough to truly represent the data.

DB-Scan Visualiser:

- **plot_tsne(apply_pca, lr, perplexity, cluster_type, eps, min_samples):**

  The plotting function for DB-Scan using t-SNE with the following parameters:

  apply_pca: Whether to apply PCA transformation. This is generally not recommended unless wanting to reduce the computation time for large sample sizes.

  lr: The rate at which the positions are adjusted in 3D space during optimisation in t-SNE. A lower learning rate does a better job at projecting the points correctly however it becomes very computationally expensive.

  perplexity: Impacts the shape of the clusters in the 3D space in t-SNE. Low perplexity generally forms tight local clusters whereas higher perplexity can at times form sparse clusters.

  cluster_type: Set to 'dbscan' for anomaly detection.

  eps (epsilon): The threshold for noise and outlier detection in DB-Scan. A smaller epsilon will reduce the ring size that is drawn around the clusters and will flag additional points as noise.

  min_samples: The minimum number of points required to form a cluster in DB-Scan. Using a small value can lead to the generation of many smaller clusters. A larger value can at times create a smaller group of large clusters since DB-Scan automatically creates the centroids.

Table 4:Example code of using the t-SNE with DB-Scan from preparing data to initialisation then running the cluster function with the parameters. This example uses the same preprocessing steps as found in Table 2.

t-SNE DBScan Evaluation on Normal Rows

```
# Use Min-Max scaling data.
scaler = 'minmax'

# Select the same size as all the currently sampled data. A limit of 10000
is recommended as TSNE is computationally expensive.
sample_size = 10000

# Create class.
pc = PlotClusters(cluster_data, scaler, sample_size)

# Ommit PCA conversion.
apply_pca = False

# Learning rate for TSNE - this sometimes impacts the shape we get in the
plots. We will set it to automatic.
learning_rate = 'auto'

# The perplexity directly affects the shape output of TSNE. Can at sometimes
be pretty random but 30 usually works quite well.
perplexity=30

# Select DBScan type clusters.
cluster_type = 'dbscan'

# Epsilon is the threshold for noise. It decides the ring (epsilon) at which
to classify the points as noise or not.
epsilon = 0.5

# Run the plot function for the class with the given parameters.
pc.plot_tsne(apply_pca, n_clusters, learning_rate, perplexity, cluster_type,
epsilon)
```

Figure 1 and 2 provides an example output for t-SNE with DB-Scan on the normal traffic in the USNW-NB15 dataset. The clusters appear in different colours, and the noise is visualised as red points that use a level of alpha transparency to differentiate them from the rest of the clusters. For Figure 1 standard scaling is used and generates a lot of noise in the normal rows along with the formation of 2 large clusters and 5 smaller ones. In Figure 2 min-max scaling is used and shows that the noise is significantly reduced in the normal rows with a smaller number of clusters.

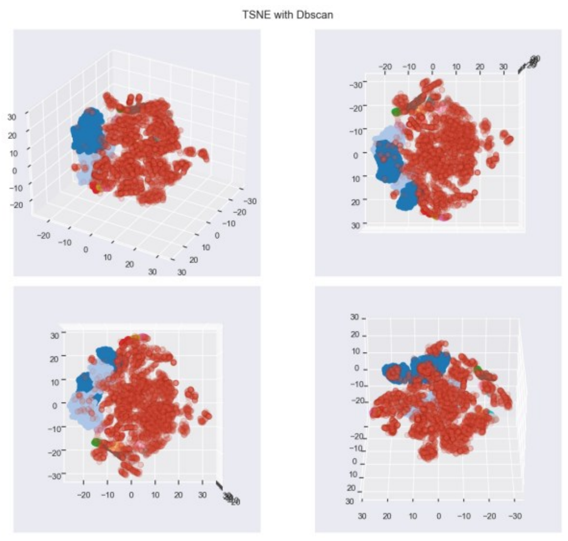

Figure 1: A 3D projection of DB-Scan clusters using standard scaling. Here the visualisation displays noise as red points that have a level of transparency.

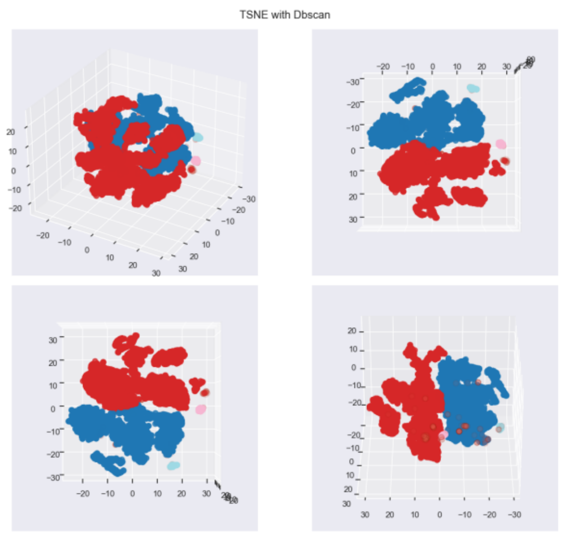

Figure 2: A 3D projection of DB-Scan clusters using min-max scaling. Solid colours indicate actual clusters and that the noise has been reduced.

3.3 SHAP Manager

The SHAP manager class is designed to simplify the computation, analysis, and visualisation of Shapley values for machine learning models. The class abstracts away the complexities of handling different model types and the raw Shapley values, offering a unified interface for users to gain actionable insights into a model's decision-making process.

The SHAP manager class includes the following features:

- **Summary Plots:** Integrates the default plot with extra functions to provide a global overview of the features that are the most important for a specified subset of rows (e.g. correct and incorrect threat predictions). The multi-class version allows for the plots to display the Shapley values for all the labels or alternatively specific labels.
- **Dependence Plots:** Integrates the default plot with extra functions to evaluate how the Shapley value for a single feature varies with the actual values across instances in a scatter plot. These plots can reveal non-linear relationships and interactions with other features. The function automatically finds the given number of best interactions using the approximate interaction function or alternatively a specific interaction can be provided.
- **Stacked Grouping for Binaries:** A function which allows for grouping related binary features into a single representation for visualisation. This simplifies the interpretation of complex categorical features. The process involves calculating the average absolute or sum of Shapley values and selecting the most important

binaries in the group. Then the rest of the data is adjusted by stacking each other variable on top of itself to the number of grouped binaries for plotting.

- **Label Encoded Category Grouping:** When analysing Shapley values across multiple categories, features with low importance can be grouped into an "Other" category. This helps in reducing noise (from many variables) and making the key features more distinguishable in the visualisations. The features are still displayed in the visualisation but are presented as a single group on the x-axis. This function is available for both the binary and multi-class tree-based model evaluations.
- **Custom Colour Mapping:** Custom colours can be applied to dependence plots. This helps in creating clearer visualisations of categorical features.
- **Jitter and Alpha Adjustments:** The jitter and alpha parameters for dependence plots can be adjusted to improve the clarity of visualisations, especially when dealing with many overlapping data points. The jitter function helps to spread out points on the x-axis for categorical or discrete features, whilst alpha controls the transparency, making dense areas easier to interpret.
- **Selecting Interactions:** Interactions can be specified as a number where the top n interactions are shown in dependence plots. Alternatively, specific interactions can be specified.
- **Outlier Filtering:** Outliers can be filtered by specifying maximum and minimum y-axis for the interaction index and x-axis for the variable where the n given percentage is filtered out based on the distribution of the specified axis.
- **Label Selection:** Labels can be easily switched for multi class evaluations. Once the label is selected all other functions will be relative to the given label. This provides granular insight into the model's decision-making process for each category and helps in understanding how the model differentiates between various cyber threats.

3.2.1 SHAP Manager Initialisation Example Usage

The following examples in Table 5 and 6 illustrate creating a background dataset and sampling the predicted threats into SHAP. The SHAP Manager class is initialised with the computed SHAP explainer object with the names of the features used by the model. For tree models, the feature encoder needs to be provided for the custom grouping function. SHAP API indicates that using background data approximates well to the larger dataset. However, this should be approached with caution. Creating background data is useful for the Deep Explainer and Gradient Explainer methods but does not translate in the same way into tree-based models. For the tree-based models it is recommended to compute the Shapely values directly without a background as it decreases complexity. The feature encoders must be provided as a list which contains the encoders for each categorical feature along with the label encoder that was used for the target labels. For multi-class tree-based models, the label type should be set to 'multi' and the parameter set label must specify which target label to set the class to. This selects the label that will be evaluated in the summary and dependence plots.

Initialisation:

- **SHAPmanager(explainer, label_type, feature_encoder, label_encoder, feature_names, set_label):**

  The initialisation of the SHAP manager class using the following parameters:

  explainer: The explainer object which holds both the SHAP values and evaluation data.

  label_type: 'binary' for either a binary tree or binary Neural Network evaluations and 'multi' for multi-class tree evaluations.

  feature_encoder: A list that contains the feature encoders when using tree-based models with label encoding. The attribute is used to convert the categorical features into strings for visualisations.

  label_encoder: When using multi-class tree models, the label encoder must be provided to allow for switching of labels using the string names.

  feature_names: When using the binary Neural Network, the evaluation data is in tensors and feature names must be provided from either the train or test set.

  set_label: Set the initial label for evaluation when using the multi-class SHAP manager which must be provided with multi-class initialisation.

Table 5: Example code of initialising the SHAP Manager class for a binary Neural Networks

SHAP Manager Initialisation for Binary Neural Network

```
# Import shap.
from Tools.shap_manager import SHAPmanager
# read docs.
help(SHAPmanager)

# Create a background sample of the tensor for Neural Network evaluations.
idx = torch.randperm(X_test_tensor.size(0))[:1000]
background = X_test_tensor[idx]
```

```
# Create background.
xp = shap.DeepExplainer(model, background)

# Compute the explainer object on sampled data (e.g. correct predictions).
explainer = xp(correct_threats)

# Feature names must be provided when using tensors.
feature_names = X_test.columns

# label type is set to binary.
label_type = 'binary'

# Initialise the class with the given parameters.
sm = SHAPmanager(explainer, label_type=label_type,
feature_names=feature_names)
```

Table 6: Example code of initialising the SHAP Manager class for binary tree models, and multi-class tree models.

SHAP Manager Initialisation for Binary and Multi-Class Tree Models

```
from Tools.shap_manager import SHAPmanager
# read docs.
help(SHAPmanager)

# We create a smaller sample to speed up the visualisations for the example.
sample = X_test.sample(n=10000)
sample = sample.reset_index(drop=True)

# Compute Shapley values without a background.
xp = shap.TreeExplainer(xgb)
explainer = xp(sample)

## Binary Initialisation. ##

# set label type to binary for evaluation.
label_type = 'binary'

# Make sure to add feature encoder used for the data.
sm = SHAPmanager(explainer, label_type=label_type,
feature_encoder=le_features)

## Multi-class Initialisation. ##

# set label type to multi for multi-class data.
label_type = 'multi'

# Provide the label category to be evaluated if multi-class.
```

```
set_label = 'DoS'

# Make sure to add the label and feature encoder used for the data.
sm = SHAPmanager(explainer, label_type=label_type,
feature_encoder=le_features, label_encoder=le_label, set_label=set_label)
```

3.2.1 SHAP Manager Functions Example Usage

This sub-section provides examples on leveraging the SHAP manager class for the binary Neural Network and multi-class tree models using customised summary and dependency plots for evaluations. The specialised functionality for binary cases includes tools for visualising feature importances and interactions specifically in the context of threat vs. normal classification decisions. Additionally, the examples show how SHAP manager can be used in a multi-class context where insight can be gained through visualisations of the different labels and how the feature attributions change for each label.

<u>Visualisation:</u>

- **plot_summary(max_features, colour, multi, features):**

  A summary plot used to evaluate feature importances for the model's output. The plot is useful in determining the most important features quickly by comparing them together using the following parameters:

  max_features: Specify the maximum number of features to plot. This is useful in generating more readable summary plots since the default is 30. Figure 3 shows a reduced summary plot with just 10 features that display the top 10 features based on the average absolute SHAP value.

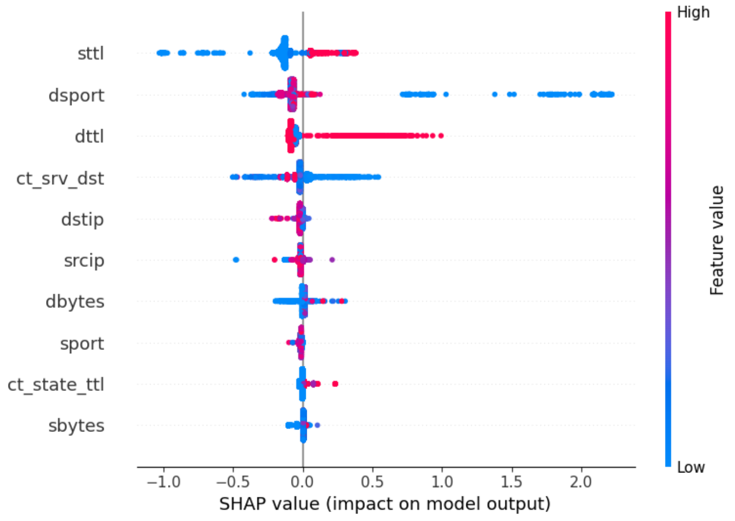

Figure 3: A standard SHAP summary plot using the SHAP manager class with a limit of 10 features.

colour: A c-map colour palette can be used to change the summary plot colour.

multi: A plot specific to multi-class tree models where multiple labels can be plotted together. The plot in Figure 4 displays the features for each threat based on the cumulative absolute average SHAP value. The parameter can either be specified as 'all' or alternatively, a list of the label names can be provided to view specific labels.

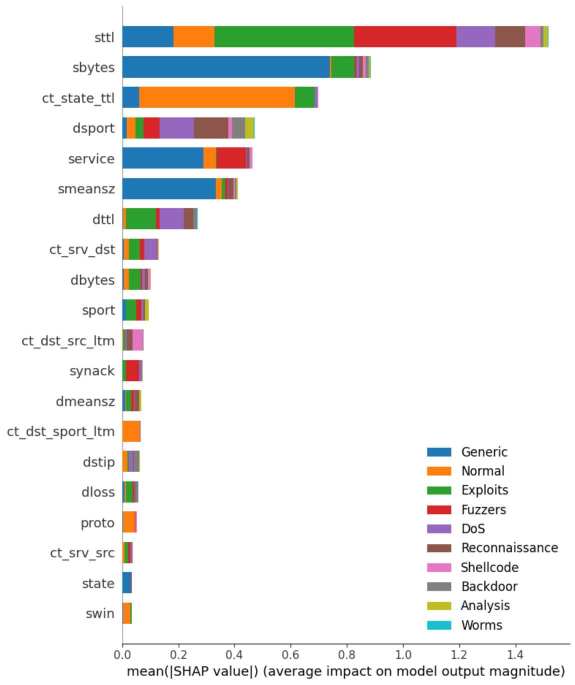

Figure 4: A multi-class version of the SHAP summary plot from the SHAP Manager class that shows the cumulative absolute average SHAP value across all labels.

features: A list of specific features can be provided with this parameter to highlight them for an easier evaluation.

- **plot_dependence(variable, n_interaction, colour, specific_interaction, x_jitter, xmin, xmax, ymin, ymax, alpha):**

  A dependence plot used to evaluate the interactions between multiple features with their actual values and Shapley values using the following parameters:

  variable: The variable to draw the dependency plot for. The variable can be provided as the actual column name as the data is transformed in initialisation allowing for easy indexing. The plot automatically plots the single strongest interaction using the approximate interaction function if no interaction parameters are provided.

  n_interaction: If specified plots the given number of interactions as subplots for the variable by approximating the top n interactions. The number of interactions must either be 1 or a multiple of 2 due to the layout of the subplots. Figure 5 displays a dependency plot using the top 2 interaction indexes.

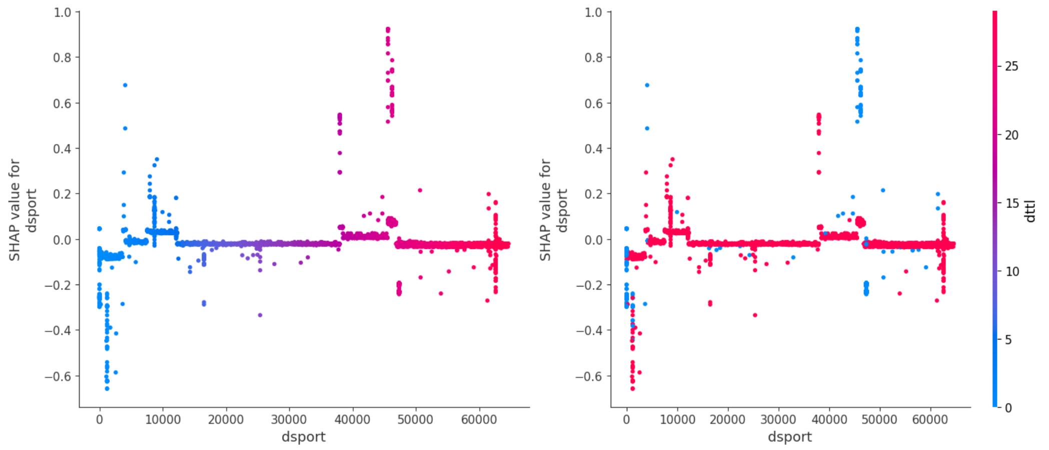

Figure 5: A SHAP dependency plot using the SHAP manager class which projects the visualisations into subplots based on the number of interactions specified.

colour: A custom nominal c-map colour palette can be provided in this parameter to help in displaying categorical interaction indexes. If the values are numeric the plot automatically plots them for a range where a hue should be used.

specific_interaction: A column name for a variable can be provided to plot the interaction index for a specific feature.

x_jitter: Uses a random seed to scatter the data points along the x-axis whilst preserving the position on the y-axis. This function is useful for discreet or binary data points where many appear on the same location.

xmin, xmax: An outlier removal tool for the x-axis variable which removes data points that are positioned within the provided percentile of the distribution given at either end of the tail.

ymin, ymax: An outlier removal tool for the y-axis variable which removes data points that are positioned within the provided percentile of the distribution given at either end of the tail.

alpha: Determines the transparency from 1.0 no transparency to 0.0 invisible. Can help to visualise points that are clustered close together.

Grouping:

- **custom_group(category, type_of, limit, calculation):**

  This function is specific for tree-based Models that use label encoding. The function must be preceded by the string encode function and sets all values that are below the given limit as ‘Other’. The calculation can be based on the top absolute average or absolute sum of SHAP values for each value within the category to select the values with the highest importance. Figure 6 shows an example of a string converted variable that is plotted on a dependency plot. Initially the values cause an error on x-axis as there are too many. When using the grouping function, the x-axis becomes more interpretable as seen in Figure 7.

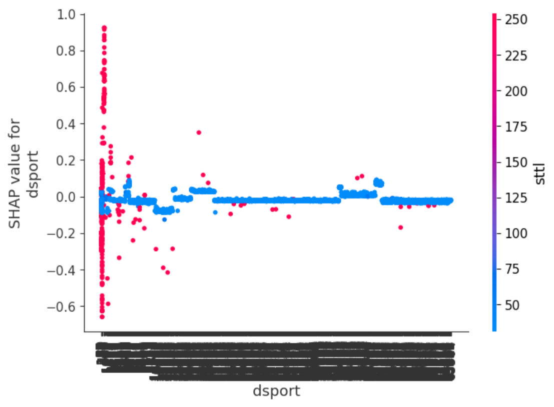

Figure 6: A SHAP dependency plot using the SHAP manager class after converting the destination port variable back into strings. The large number of values creates an unreadable x-axis.

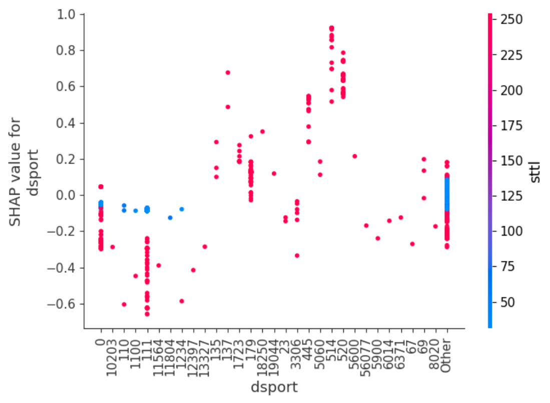

Figure 7: A SHAP dependency plot using the SHAP manager class after using the custom grouping function on destination ports with a limit of 30. The grouped values are stacked in 'Other'.

The following parameters are used for the label encoded grouping function:

category: The categorical variable to use for grouping.

type_of: Specify either 'label-encoded' for automatic grouping based on the calculation to a given limit or 'label-encoded-specific' to specify values within the label encoded category to be grouped.

limit: The limit of variables that are selected as the top values within the category. Other values below this limit are grouped into the string name 'Other'.

calculation: The calculation used to find the top values. Either 'average' for the absolute average or 'sum' for the absolute sum of SHAP values.

- **stacked_group(category, limit, calculation):**

  This function is specific for variables that have been one-hot encoded. The function groups one-hot encoded variables back together by stacking them on top of each other. For each stacked variable the other variables are stacked on top of themselves. Calculation can be based on the absolute average or absolute sum of SHAP values to select the features with the highest importance. Depending on the size of the evaluation sample, this process can be computationally expensive but runs efficiently with smaller sets. After using the function, it is recommended to restore the data so that cumulative stacking does not occur when using the function again. Figure 8 shows an example of how the binaries appear after being grouped for a category variable. grouping function, the x-axis becomes more interpretable as seen in Figure 7.

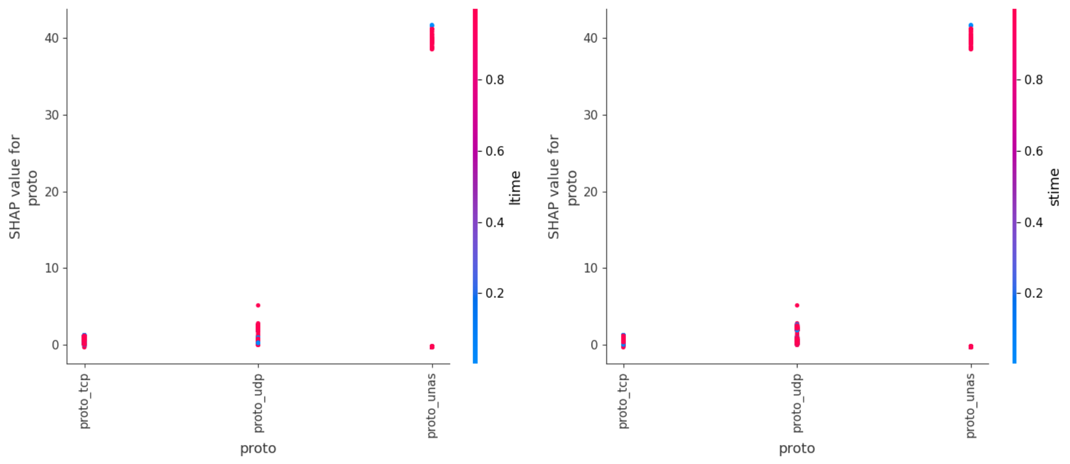

Figure 8: A SHAP dependency plot using the SHAP Manager class and 2 interactions which are projected into subplots. The one-hot encoded protocol variables are stacked using the stacked group function and visualised collectively on one plot.

The following parameters are used for the one-hot encoded grouping function:

category: The original category name that precedes the one-hot encoded variable name (e.g. 'proto' in 'proto_udp').

limit: The limit of variables that are selected as the top values within the category. Other values below this limit are grouped into the string name 'Other'.

calculation: The calculation used to find the top values. Either 'average' for the absolute average or 'sum' for the absolute sum of SHAP values.

Additional Functions:

- **string_encode(categories):**

  This function is used to inverse encode label encoded variables to their original string value names. Once this function is used the dependency plot registers them as categories and plots them as nominals. This function must be used before using the custom group function.

  categories: The list of categorical variables that correspond to the list of label encoders given in the initialisation of the SHAP manager class.

- **restore_data():**

  A function which restores the data back to its original format given in the initialisation of the SHAP manager class. This is useful after running the grouping transformations and should be used between each stacked group function evalution.

- **remove_features(features):**

  A function used to remove features and their corresponding SHAP values. This is useful when trying to view summary plots where there may be features of less interest than others or if wanting to filter features out of the automatic interaction index for dependency plots.

  features: The list of features to filter from the evaluation data.

- **set_label(label):**

  A function used for setting the label to evaluate in the multi-class instances. As the label is changed the SHAP values become drawn specifically for that label in summary plots and dependency plots.

  label: The label to change the evaluation to given as a string.

4. Models

This section delves into the specific machine learning models that were implemented for threat detection with discussions on their strengths and weaknesses in the context of cybersecurity data.

4.1 Neural Network

Neural Networks can learn complex non-linear relationships and have shown promise in various cybersecurity applications. When it comes to the complexity of the model architecture, starting with a lightweight design is often desirable not just for network data, but also for the transition into explainability as the complexity directly affects the calculation of SHAP values and can hinder the iterative process of gaining insights for refinement. The process for designing the architecture is to start simple, where a small architecture is created as a baseline for testing purposes. Neural Networks face challenges with encoding and the need for scaling the data which inherently causes problems in the evaluation phase of cybersecurity datasets. A loss of information is created due to the difficult task of one-hot encoding high cardinality categories and when using scaling only direction of the variables can be measured rather than distance. The models have been found to be very effective using Synthetic Minority Oversampling Technique (SMOTE) from the Imbalanced-learn library where a basic model with a small dense layer and single activation function was able to detect all threats in the USNW-NB15 dataset without a significant increase to false positives. A standard model is also created as a benchmark comparison where results have shown a type of inverse result in that the majority class is predicted well but at the expense of an increase in false predictions in the minority labels. The model is designed for binary classification. It applies the Rectified Linear Unit (ReLU) activation to the expanded dense layer, introducing non-linearity after the input is transformed by the weights and biases. The output layer then applies a Sigmoid function to squash the result into a probability between 0 and 1, representing the likelihood that the input belongs to class 1 (threat). The output logit is then compared to the actual label to calculate the loss using Binary Cross Entropy and a backwards pass occurs using Adam optimiser which adaptively adjusts the gradient. The functionality of the Neural Network class is outlined below:

- **Dynamic Architecture:** The model offers two key components in architecture design that can be modified. The dense layer can easily be changed and passed forward as a parameter in initialisation. The activation function can be changed by setting the activation attribute after creating the class. The architecture is kept simple for preliminary benchmark testing of the sampling method. This is particularly important since the focus of the project is also explainability of models rather than just optimisation.
- **Integration of Sampling:** The class allows the incorporation of a sampler during the training run. The sampling methods introduced are SMOTE and Tomek's Links which help in predicting minority classes accurately. When using the sampling techniques batch size should be chosen carefully as this directly impacts the effectiveness of the technique. Using a smaller batch size introduces dynamic sampling where if there is only 1 class in a batch, the sampler fails and reverts to the standard batch sample. This dynamic nature creates a type of shuffle effect between sampled and unsampled training.
- **Hyperparameter Tuning:** Hyperparameter tuning is done by specifying the learning rate, the number of epochs, batch size, and the parameters for the sampling methods. For the SMOTE model, A high learning rate has been seen to be the most effective along with a small batch size whilst parameters for the sampler are set to an automatic sampling strategy. Although this is effective, the chance of missing the minimum can be high and for this reason evaluation of each epoch is done through the guidance of confusion matrices.
- **Device Selection:** The Neural Network uses CPU for tensors as integration challenges into SHAP were found due to the torch no grad error. Since testing has been done, some results have shown a future model could potentially include GPU integration.

- **Model Saving and Loading:** The model includes functions to save the trained model allowing for the preservation and later loading of specific training checkpoints.
- **Performance Tracking:** The model tracks metrics during training such as training loss, training accuracy, train macro f1 score, and epoch time. The model also stores the predicted and actual values of the test set along with the overall test accuracy which can be used for evaluation.
- **Dataset and Logit Storage:** In the case of using sampling techniques the data is generated in the training run and the model has options to store this dataset as an attribute for the last epoch. The model also stores the raw logits for the test set which can be used to customise the predictions further with rule-based methods or incorporate them as transferred features to a secondary model.
- **Evaluation Functions:** Additional evaluation functions are included which allow for plotting the stored training metrics on a graph and evaluating the predicted test results in a confusion matrix.

4.1.1 Binary SMOTE Neural Network

Predicting threats is critical in cybersecurity environments as any malicious package or disruption to a network traffic can be detrimental for businesses. A Neural Network can be customised as the first barrier to detect all incoming threats and even if it makes some misclassifications on normal network traffic, then at least, the worst scenario has been avoided. SMOTE is particularly effective where there may be only a small number of cases of threat labels compared to normal labels. During training SMOTE can be used to up-sample threats in each imbalanced batch using interpolation. After training the model, the prediction can be done on the standard test set data. Once a model is optimised to predict all threats on dataset using K-fold validation, the predicted threats can be transferred into a secondary model as an ensemble. This is done knowing that some normal traffic is passed along with it, but fine tuning becomes potentially easier after the transfer as the focus shifts to a smaller set of data. Evaluation techniques such as SHAP with tree-based models can be facilitated to gain insight into why the remaining labels are being misclassified. The key features of the primary SMOTE Neural Network are:

- **A Small Dense Layer:** It was found that a basic size of 256 neurons created the optimal SMOTE based model and increasing the layer size further did not yield significantly better results.
- **A High Learning Rate:** A high learning rate of 0.1 gave the best results for the SMOTE Model and caused it to converge within 5 epochs. The effect of the

learning rate and batched shuffling caused the model to oscillate, and the best strategy was to evaluate each epoch in the confusion matrix then select the model that consistently predicted all threats.

- **Small Shuffled Batch Sizes:** A larger batch size caused SMOTE to not be as effective. The dynamic nature that is introduced with small batches contributed to the model effectively predicting all threats. The optimal batch size for the SMOTE model was found to be 128.

4.1.2 Binary SMOTE Neural Network Example Usage

The initialisation of the model consists of deciding on the batch size in the preprocessing steps, the dense layer size, providing a device and creating a folder to save the trained models. Table 7 provides an example of preparing data that has been split into training and test sets and converted to a tensor based. The float value of the tensors specifies the precision of the model which also directly affects the training time and SHAP value calculation time (higher precision leads to an increase in complexity). The following sections explain the implementation of the Neural Network pipeline.

<u>Pre-processing:</u>

- **tensor(data, precision):**

  Tensors are created for the operations in the model with the following parameters:

  data: The data set used for the tensor which should either be the training set or test set.

  precision: The precision of the calculations for the Neural Network model and SHAP values. This needs to be in the form of a float value.

- **TensorDataset(data, labels):**

  A Tensor Dataset object is created with features and labels using the following parameters:

  data: The data that contains the feature values for either the train or test set.

  labels: The labels that correspond to the given feature data.

- **DataLoader(data, batch_size, shuffle):**

A Data Loader object is created to batch the data with the following parameters:

data: The Tensor Dataset which contains both the feature values and the labels for either the train or test set.

batch_size: The size of the batches which are split into the number of rows of the whole dataset divided by the batch size. The last batch is likely to be a different size compared to the rest.

shuffle: A shuffling function which shuffles the order of the batches. This helps in training the model but should only be used in cases when the data is not sequential. The shuffling function should not be used on the test set as in practice we want the test set to represent the true data.

Initialisation:

- **NNdynamic(n_features, fc_size, device, save_dir):**

  The Neural Network class is initialised with the following parameters:

  n_features: The number of features that correspond to the training and test dataset which can be found by indexing the second dimension of the tensor object.

  fc_size: The dense layer size which controls the size of the layer where the activation function occurs.

  device: The device used for the tensor operations which is set to CPU to mitigate errors in SHAP integration.

  save_dir: The directory to save the model weights to for the given factor provided in the training run parameters.

Table 7: Example code for pre-processing data for Pytorch Neural Network using Data Loaders and tensors.

Neural Network Pre-Processing

```
# Select the batch size.
batch_size = 128

# Convert each train and test set into tensors.
X_train_tensor = torch.tensor(X_train.values, dtype=torch.float32)
X_test_tensor = torch.tensor(X_test.values, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train.values, dtype=torch.float32)
y_test_tensor = torch.tensor(y_test.values, dtype=torch.float32)

# Convert to a Tensor Dataset which stores both the train and test sets.
```

```
train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
test_dataset = TensorDataset(X_test_tensor, y_test_tensor)

# Add the Tensor Datasets into a DataLoader object which manages the
batching. Shuffle function is used on the train set but not the test set.
train_loader = DataLoader(train_dataset, batch_size=batch_size,
shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)
```

After initialisation the training process for the model can begin. As outlined in Table 8, the training parameters consists of providing the number of epochs, learning rate, the factor at which to save the weights for each epoch, the sampling method used, and whether to store data which is used for the generated data when using sampling methods.

Training:

- **run(train_loader, learning_rate, epochs, save_factor, sampler, params, store_data):**

  The function to start training the Neural Network with the following parameters:

  train_loader: The train Data Loader object that contains the feature values and labels as tensors.

  learning_rate: The learning rate for the model which is the starting point for the Adam optimiser.

  epochs: The number of times the data is iterated through for training. Due to the nature of shuffling from the Data Loader along with the batch size and sampling method, each epoch becomes slightly different if up sampling is used.

  save_factor: The factor to which to save the weights. For every n save factor, the model weights are saved for that iteration.

  sampler: If given, the sampler the sampling method that is used which is either ‘smote’ for SMOTE or ‘tomeks’ for Tomek’s Links.

  params: The parameters that are used for the sampling method. For SMOTE ‘sampling strategy: auto’ is used which up-samples the minority label until it has the same number of values as the majority label.

  store_data: Used to store the generated dataset created during the sampled training run.

Table 8: Example code for training the SMOTE Neural Network on the USNW-NB15 dataset.

Neural Network Training

```python
# Import the Neural Network and supporting functions.
from Models.models import NNdynamic, plot_metrics, plot_confusion_matrix

# read the Neural Network docs.
help(NNdynamic)

# Set device to CPU to avoid errors in SHAP.
device = torch.device("cpu")

# The number of features which are in the dataset.
n_features = X_train_tensor.shape[1]

# The dense layer size.
fc_size = 256

# This is the directory to save to.
save_dir = './Models/saved_models/Test'

# Create the model with the input parameters.
model = NNdynamic(n_features, fc_size, device, save_dir)

# Set the sampler to 'smote'
sampler = 'smote'

# Set the sampling strategy to 'auto' for the sampler parameters.
params = {'sampling_strategy': 'auto'}

# Set learning rate.
learning_rate = 0.01

# The number of epochs over the whole dataset.
epochs = 5

# The save factor for the model (saves at every factor of the value).
save_factor = 1

# Train the model.
model.run(train_loader, learning_rate, epochs, save_factor, sampler, params,
store_data=False)
```

Testing and evaluation of the model can be done by using the attributes and helper functions. Due to the early convergences and oscillations of loss it is suggested that the test occurs on each epoch where the model is loaded, a test set is run, then the results are evaluated with a confusion matrix. Table 9 provides an example of this procedure.

Evaluation:

- **test(test_loader):**

  The function to predict the test set using the Neural Network with the following parameters:

  test_loader: The Test Loader object that contains the test feature values and labels.

- **plot_metrics(metrics, epoch, title):**

  A function for plotting one of the stored attributes for evaluating with the following parameters:

  metrics: The metric attribute from the model, which is either training loss, training accuracy, training macro f1 score, or training epoch time.

  epoch: The number of epochs that was used in training which helps determine the length of the y-axis.

  title: The custom title that is used for the plot.

- **plot_confusion_matrix(actual_labels, predicted_labels):**

  Plot a confusion matrix using the actual and predicted labels given with the following parameters:

  actual_labels: The original labels of the test set.

  predicted_labels: The labels that the model predicted.

Table 9: Example code of testing and evaluation of the Neural Network model.

Neural Network Testing and Evaluation

```
# Read the plot_metrics doc.
help(plot_metrics)

# Plot loss of the training for each epoch.
plot_metrics(model.train_loss, epochs=epochs, title='Loss')

# Plot accuracy of the training for each epoch.
plot_metrics(model.train_accuracy, epochs=epochs, title='Accuracy')

# Plot F1 macro of the training for each epoch.
```

```python
plot_metrics(model.train_f1, epochs=epochs, title='F1 Macro')

# Plot complexity of the training for each epoch.
plot_metrics(model.epoch_time, epochs=epochs, title='Epoch Time')

# Run model on test set.
model.test(test_loader)

# Plot confusion matrix with the stored predictions of the last epoch.
plot_confusion_matrix(y_test, model.test_predicted)

# Alternatively, plot a confusion matrix for each saved epoch.
for i in range(len(epochs)):
    print(f"Epoch Number: {i+1}")
    model.load_model(f"./Models/saved_models/Test/PB_epoch_{i+1}.pth")
    plot_confusion_matrix(y_test, model.test_predicted)
```

Figure 9 shows an output of a confusion matrix from a trained SMOTE model using the USNW-NB15 dataset where the true negative minority class is predicted 100% and 6262 false predictions occur in the majority class. This result is desirable in the SMOTE model that is used as a primary detector as all threats have been predicted, and the false positives are not significant.

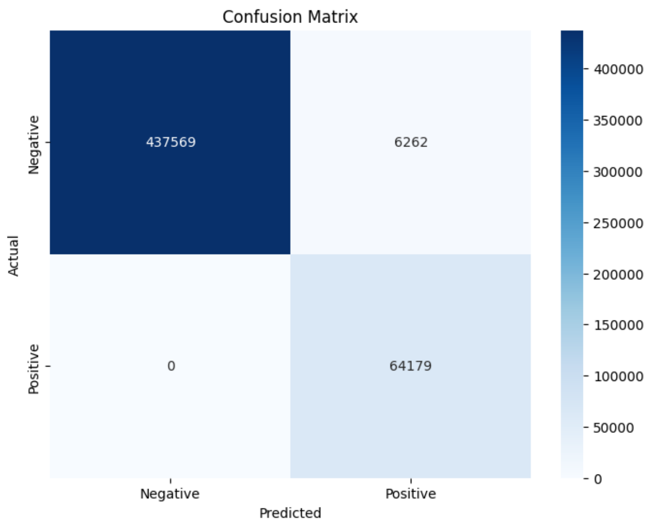

Figure 9: A confusion matrix showing the results of a SMOTE trained Neural Network on the USNW-NB15 dataset predicting all threats correctly.

4.2 Tree-Based Models

Tree-based machine learning models, such as Random Forests and Extreme Gradient Boosting (XG-Boost), are valuable tools in cybersecurity due to their effectiveness in handling complex data. These models operate by recursively partitioning the feature space based on decision rules learned during training. A significant advantage of tree-based models is their capacity to train and predict on native data with minimal pre-processing meaning that continuous variables do not require scaling and categorical features can be label encoded which in turn aids in evaluating in how certain features correspond to correct and incorrect predictions for threats. Although the models achieve high and balanced accuracy, their robustness at times causes it to be ineffective in using techniques to focus on minority classes such as SMOTE. Additionally, the models are penalised through long computation times for Shapley values as the number of estimators are increased. The purpose of the models in this project is to use them as secondary multi-class classifier after detecting all threats to evaluate what caused the remaining false positives.

- **General Approach:** Both the Random Forest and XG-Boost models share the same pre-processing steps and have similar parameters. The models used are implemented through the sci-kit learn library. The models are used as a secondary multi-class classifier after the initial detection of all threats.
- **Preprocessing for Tree Models:** Tree models are generally robust to the input data format and do not strictly require one-hot encoding for categorical features or scaling of numerical values. This allows utilisation of methods like label encoding to convert categorical features. The preprocessing step included detecting all threats with the Neural Network then forwarding the remaining false positives to the tree models for further analysis.
- **Model Selection Considerations:** When selecting between different tree models, such as Random Forest and XG-Boost, practical considerations like computational stability and compatibility with interpretability tools like SHAP are important. Although Random Forests tend to yield a higher accuracy, they encounter more issues with SHAP calculations.

In general, the project's work with tree models uses a standard approach outlined below:

- **Model Training:** The training process involves initialising models with specific parameters, such as setting the number of estimators and adjusting the depth and leaf nodes. There are practical trade-offs during this phase as the complexity directly affects computational efficiency of calculating Shapley values.
- **Model Prediction:** Once trained, the models are used to make predictions on a test set.

- **Model Evaluation:** The performance of the trained models is evaluated using standard classification metrics. This includes assessing accuracy for each class and examining a detailed classification report to see where the misclassifications occur for each threat category. The models are then used in SHAP to gain further insight into feature importances for the different labels.

4.2.1 Random Forest

Random Forest is an ensemble learning method that constructs multiple decision trees during training and outputs the mode of the classes (for classification) or the mean/average prediction (for regression) of the individual trees. This approach reduces the risk of overfitting and often yields balanced and accurate predictions compared to a single Decision Tree.

4.2.2 XG-Boost

XG-Boost is a highly effective ensemble learning algorithm that sequentially builds decision trees, with each new tree designed to correct the residual errors from the preceding trees. It utilises a gradient boosting framework and incorporates regularisation techniques to mitigate overfitting. The XG-Boost model is the recommended tree model found in the SHAP API.

5. Acknowledgments

Project Team Members:

- Mathew Coleman (Project Management, General Documentation, Validation Testing)
- Scott Chandler (Document Management, General Documentation, Validation Testing, Research)
- Edric Laitly (Scrum Management, General Documentation, User Acceptance Testing)
- Ramandeep Singh (UX Development and Design, General Documentation, UX User Guide, Research)
- Adi Selak (Machine Learning and Tools Design, General Documentation, Pipeline Testing, Code User Guide)

We would like to acknowledge the researchers and developers who contributed to the open-source libraries and datasets that made this work possible. Specifically, we appreciate the creators of the USNW-NB15 dataset Nour, M and Slay, J for providing a valuable resource for cybersecurity research. Thanks to Lundberg, S and Lee, S who have made it possible to use SHAP by integrating the values into machine learning methodologies. A special thank you to Chowdhury, A for guiding the team in the project by introducing our goals and verifying the achievements and to Siamak, M and Khayer, B as project sponsors.

A list of the libraries used in Python that helped us reach our goals is provided here:

Pytorch	A machine learning Library for Neural Networks that includes tensors, activation functions, and the Neural Network class
Sci-Kit Learn	A machine learning library with various pre-processing tools, models, and evaluation tools
Imb-Learn	A sampling library that introduces various methods to handle imbalanced datasets
XG-Boost	A library specific to the XG-Boost model used in the project
SHAP	The SHAP library used for the evaluations of machine learning models.

6. Frequently Asked Questions

This guide helps you understand how a smart system uses different techniques to detect cybersecurity threats and explain its decisions.

General Questions

- **What is the main purpose of this system?**
  This system is designed to help businesses and other organisations identify and understand cybersecurity threats. Its core purpose is to provide accurate threat detection using advanced machine learning, and to offer clear explanations for why a threat was flagged. This helps businesses make informed decisions, build trust in automated security, and respond to threats more efficiently, ultimately enhancing their overall cyber resilience.

- **What are the main objectives of the solution?**
  The key objectives include enhancing cybersecurity threat detection using advanced machine learning models, improving transparency by implementing Explainable AI (XAI) to provide interpretable justifications for threat detection, optimising the threat detection process to reduce false positives, improve response times, and supporting regulatory compliance with cybersecurity obligations through Explainable AI-driven security decisions.

- **Why is it important for the system to explain its decisions?**
  It's important to understand why the system identifies a threat, not just that it did. Explanations help you see what factors that led to a specific detection.

- **How does the system generally work?**
  The system uses different 'smart learning' methods to identify patterns in network data and detect threats. It also uses special tools to make sense of these detections.

- **What kind of data does this system analyse?**
  It primarily analyses network traffic data, like the USNW-NB15 dataset, to identify normal activity versus potential threats.

- **What does Explainable AI (XAI) mean in the context of this system?**
  XAI is about making AI models easy for people to understand and trust. For this system, it means providing clear explanations for cyber threats, so security analysts can easily see the reasoning behind the system's decisions using a method called SHAP.

- **What does SHAP explainability mean and what is its role?**
  SHAP (Shapley Additive Explanations) is a tool that helps explain individual AI predictions. It shows how much each piece of information (feature) contributed to a specific threat alert. SHAP makes the AI's reasoning transparent, helping users understand why a certain alert was triggered.

- **How does this system help security analysts?**
  It helps to flag threats so analysts can respond faster. By providing clear explanations, the system enhances their decision-making and builds trust in automated security tools.

- **Does the system use human feedback?**
  Yes, human input is crucial. This system was built with valuable feedback from security analysts to make sure it meets real-world needs. Security analysts can also give ongoing feedback through the 'Feedback' section in the system interface on the website. This input is crucial for continuous improvement.

Understanding the Tools

- **What are encoders and why are they used?**
  Encoders help prepare data for the smart system. Many smart systems need numbers as input, so encoders convert text or categories into a numerical format.

- **What is the 'Correlation One-Hot Encoder'?**
  This is a specialised tool that helps convert categorical data into a binary features. It focuses on converting only the most important categories that are strongly related to whether something is a threat or not, which helps manage very large datasets with high cardinalities.

- **What is the 'Cluster Visualiser' for?**
  These tools help you see patterns in network data, especially normal network traffic. They can show if there are unusual data points (noise or outliers) that might cause the system to incorrectly flag something as a threat. They also help in choosing the best way to prepare data using scalers for the smart system to reduce these misclassifications.

- **What is the 'SHAP Manager'?**
  SHAP Manager is a central tool to the smart system that adds functionalities to the SHAP library. It can simplify how highly technical data categories are viewed with its grouping functions and streamlines the indexing of variables for

visualisations. It can provide overall summaries of important factors or detailed explanations for specific detections.

Understanding the Models

- **What are 'Neural Networks'?**
  Neural Networks are a type of advanced smart system that can learn complex patterns. They are very powerful but often need data to be prepared in specific ways (like making sure numbers are on a similar scale).

- **What are 'tree models' like Random Forest and XG-Boost?**
  Tree models are another type of smart system that are generally easier to understand and require less data preparation compared to Neural Networks. They are good at handling data with many unique categories and are robust to 'noisy' data.

- **Can different smart systems work together?**
  Yes, the guide describes combining Neural Networks and tree models. For example, a Neural Network might be used first to detect all threats, and then a tree model can help provide further insight into the misclassification between labels.

- **How does the system handle false alarms or misclassifications?**
  The system uses various evaluation methods to check its performance. This includes looking at confusion matrices and SHAP visualisations to understand why it misclassifies threats (e.g., mistaking one type of attack for another, or a threat for normal traffic). This information helps in fine-tuning the system to reduce errors.

Using the System's Interface

- **How do I access and understand this system?**
  Open a web browser and go to: https://main.d182cvwsdiq1zi.amplifyapp.com/.

- **What do 'Test Results' show in the UI?**
  The 'Test Results' section in the User Interface displays how well the smart system performs. It shows if the system is correctly identifying threats and avoiding mistakes.

- **What kind of insights do the reports included in the Explainable AI Section provide?**

Multiclass Threat Detection Report: This report helps you understand why the system identifies specific types of threats (like DoS attacks).

Neural Networks Report: This report lets you compare different smart detection methods to see the effect of the sampling techniques on correct and incorrect predictions.