COMP 8085 - Project 1 - Group 3 Report

Mohammed Bajaman
School of Computing and Academic studies
British Columbia Institute of Technology
Burnaby, British Columbia, Canada
`mbajaman@my.bcit.ca`

Jacob Pauls
School of Computing and Academic studies
British Columbia Institute of Technology
Burnaby, British Columbia, Canada
`jpauls14@my.bcit.ca`

Keith Liu
School of Computing and Academic studies
British Columbia Institute of Technology
Burnaby, British Columbia, Canada
`kliu107@my.bcit.ca`

Abstract

This project proposes multiple methods of feature selection and classification that could be used in a Network Intrusion Detection System (NIDS) to identify and categorize potential attacks while scrutinizing their capability, particularly against the UNSW-NB15 dataset. Review of the relevant literature suggested that *AdaBoost*, *K-Neighbors*, and *Multi-layer Perceptron* methods were the most promising for success. During the experiment, feature selection methods were used to reduce the available features, and models were trained, tested, and validated on isolated sets of data for both a binary and a categorical classification task. Findings suggest that *AdaBoost* classification with a decision-tree as base estimator performs exceptionally well when provided a small number of features and *K-Neighbors* as well as *Multi-layer Perceptron* perform with equally comparable results.

1 Task Definition

1.1 Problem Statement

Network protection and defense is an active and constantly growing problem space. An understanding of the available data surrounding attack methods is crucial for determining the nature of future attacks as well as innovating new preventative measures. Furthermore, artificial intelligence plays a key role in measurably keeping up with the diverse nature of these attacks and assisting with identifying and categorizing them early and often. This project lays a framework for a potential Network Intrusion Detection System (NIDS) which employs numerous methods of classification for determining and categorizing the nature of network attacks that range from normal to abnormal in nature.

1.2 Scope

The NIDS system created is capable of training models employing AdaBoost, K-Nearest Neighbors, or Multi-layer Perceptron classification methods. After training, models use data aligning to the specifications of the UNSW-NB15 dataset outlined in *Moustafa et al. [1]* to perform one of two classification tasks. Once selected, models can make predictions as to whether a record of data indicates a network intrusion through a binary classification task or determining the category of a potential attack through a non-binary classification task.

Course project report submitted in fulfillment of requirements for COMP 8085, BCIT.

1.3 Evaluation Metrics

We used a calculated F1 score as the primary method for determining the accuracy of models on both the split test set as well as the unseen validation set.

In the available classification reports for each model, the F1 score is a numerical score between zero and one that can be represented as a percentage accuracy for each class that a model predicts. More specifically, this F1 score is calculated with the precision (metric representing mislabelled positive results) and recall (metric representing correctly classified results) of the model in its predictions which uses the number of true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN) on a certain set of data [2]. As a result, both prediction and recall are bounded by the number of predictions made by the model on a particular dataset. The overall accuracy of each model was also used to determine the overall value of its net true predictions on unseen data.

$$precision = \frac{TP}{TP + FP} \tag{1}$$

$$recall = \frac{TP}{TP + FN} \tag{2}$$

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{3}$$

$$F1 = 2 \times \frac{precision * recall}{precision + recall} \tag{4}$$

Additionally, the cumulative macro average of F1 scores was used to compare and monitor the precision of the model's predictions across all classes for a particular set of data.

1.4 Dataset

The dataset used was acquired from *Moustafa et al. [1]* as a portion of the UNSW-NB15 dataset and contained 449,796 records. In our use of this dataset, 45,000 records were used for validation and a 70/30 split was performed on the remaining 404,796 records for a single round of training and testing. As such, training was completed on 283,357 records (70%) and testing was completed on 121,424 records (30%) after preprocessing. In comparison to the original UNSW-NB15 dataset, the data we used as part of our project represented roughly 17% of all data collected in the cumulative set [1].

2 Infrastructure

2.1 Hardware

Models were trained on hardware with a variety of specifications over the course of the experiment. The lowest performing machine had 8GB of RAM, a six-core AMD CPU, and an NVIDIA GeForce 750ti GPU. Other machines averaged 16GB of ram, eight-core CPUs, and variations of integrated or discrete graphics cards.

2.2 Software

The Python programming language was used for conducting the experiments including both binary and non-binary classification tasks. Within the Python ecosystem we used scikit-learn (classification tasks and feature selection), pandas (data cleaning, framing, and analysis), as well as both matplotlib and seaborn (graphing and visualization of data) with varying levels of use across tasks. Additional dependencies used, including any co-dependencies, are listed in the project's *requirements.txt* file.

2.3 Makefile

The experiments were bootstrapped with a GNU Makefile containing commands for setting up the environment, training models, running pre-trained inference models, and executing feature selection techniques on the provided UNSW-NB15 dataset as well as our extracted training and validation sets. Installation of the virtual environment can be performed using `make install` and the environment can be cleaned with `make clean`. Each command for executing a classification task follows the convention `'<classifier-abbreviation>-<task>'` and optionally contains an 'inf' suffix to load (or train, if not found) a pickled model in inference mode (ex: `make abc-label-inf` would run an inference model for the AdaBoost classifier on the *attack_cat* task).

The output directory of models pickled by the *nids.py* script can be directly changed by modifying the *MODELS_DIR* variable in the Makefile or by leaving it empty. Furthermore, paths for datasets can be added or changed by modifying the *TRAIN_DATA* or *VAL_DATA* variables.

3 Approach

3.1 Dataset

The validation set for each of the classification methods consisted of 45,000 records extracted from the original dataset (records 2 – 45,0001) and the remaining training set consisted of the remaining 404,796 records. Model selection and training were performed with a single round of training/testing using a 70/30 split on the training set and the inference model was validated with the validation set. Employing this dataset allowed further insight into the fit of models on unseen data and further helped in narrowing down the model selection process.

3.2 Feature Selection

Feature analysis was performed to narrow down the available features in the UNSW-NB15 dataset for each classification task using only the initial training set. Three feature elimination/selection techniques were used including, recursive feature elimination (RFE), sequential feature selection (SFS), and correlation feature selection.

3.2.1 Recursive Feature Elimination (RFE)

Recursive feature elimination (RFE) operates in accordance with a provided estimator and a target number of features to select. Initially, the estimator is trained on all provided features and assigns each feature a coefficient to represent their score (or importance) amongst the entire set of features. The algorithm then prunes out the lowest feature scores and proceeds to recurse the feature set into smaller and smaller groups of features until the target number of features is reached.

Despite being relatively effective at capturing a target number of feature sets, RFE's effectiveness and performance are heavily reliant on the estimator being used. Additionally, RFE often begins without a complete understanding of which features are valid, leading many algorithmic implementations to employ a method of cross-validation for evenly determining the proper features to select.

3.2.2 Sequential Feature Selection (SFS)

Sequential feature selection (SFS) adds or removes features to form a feature subset. This is a greedy procedure for feature selection where at each iteration, the selector chooses the best feature to add or remove based on the cross-validation score of a feature as well as the direction it performs the procedure (forwards vs. backwards).

Forward-SFS starts with zero features and finds the one feature that maximizes the cross-validated score when an estimator is trained on this single feature. Once a feature is selected, the process is repeated by adding a new feature to the set of selected features. Backward-SFS on the other hand works inversely where you start with all the features and remove features on each iteration.

Deciding the direction to perform SFS is important as forward and backward selection does not yield equal results. Performance can also vary depending on the direction of feature selection where one might be faster than the others depending on the requested number of features to select.

3.2.3 Correlation Feature Selection

Correlation feature selection selects the most relevant features from a dataset based on their correlation with an identified target feature or each other. Using correlation feature selection we can reduce redundant or irrelevant features from the dataset based on the correlation coefficient. The formula for the standard correlation coefficient, also known as the Pearson method, calculates the correlation coefficient (r) by taking the results of the covariance ($cov_{x,y}$) over the standard deviation (σ) of x and y.

$$r = \frac{cov_{x,y}}{\sigma_x \times \sigma_y} \tag{5}$$

The result will be a number between -1 and 1, where -1 indicates a perfect negative correlation, 0 indicates no correlation, and 1 indicates a perfect positive correlation.

If the correlation coefficient between two features is high (close to 1 or -1), it indicates that the two features are highly correlated and one of them can be eliminated to reduce redundancy as including them does not provide additional information that is useful for predicting the target variable and can lead to overfitting. However, if the correlation coefficient between a feature and the target variable is high, it indicates that the feature is highly predictive of the target variable and should be retained.

A limitation of correlation feature selection is that it only considers linear relationships between variables, so it may not be appropriate for datasets with nonlinear relationships.

3.3 Classification Methods

Various classification methods were selected according to how well they made predictions that appropriately generalized the data from the UNSW-NB15 dataset. Three classification methods were selected on the basis of their performance on the held-out test/validation sets as well as their performance on the larger, unbalanced, set of records. These methods included decision tree classification, AdaBoost classification, and k-neighbors classification.

3.3.1 AdaBoost Classification

AdaBoost (adaptive boosting) is one of many ensemble methods for machine learning that leverages running many base models with a variety of hypotheses that attempt to "boost" themselves on each iteration by learning from the mistakes of its previous model. This occurs by providing each "weak" version of the original estimator with varying sets of the training data and then combining the resultant classifiers from the estimator into a single one.

AdaBoost is often identified as a classifier that becomes the best version of other estimators as a result of its nature to take an existing one and use many weak versions of it, boost them, and result in a better predictor than the original. By doing this, the main effect of AdaBoost, and all algorithms that employ boosting, is a strong reduction in the bias of any weak estimator or learning algorithm that serves as a base for AdaBoost [3]. Correspondingly, AdaBoost is a sequential learning model that learns after each iteration of testing the hypothesis of the provided (or most recent) weak estimator until the ideal one is reached.

Our experiment focused on implementing an AdaBoost classifier on tasks using a decision tree as the "weak" estimator. Decision trees are supervised learning models that form "trees" of rules based on the data they're provided which each map to a particular output. When used in binary classification problems, each sequence of branches on the decision tree maps to either a positive (acceptance) or negative (rejection) result. In contrast, categorical problems have sequences of branches that map to a particular non-binary categorical output. Most decision trees operate by using an algorithm (or a "criterion") to decide the most important feature to expand and make decisions against. Once decided, each feature is expanded from the root and each decision is measured on the acceptance of the available examples being used that map to the current path of decisions.

While decision trees are effective at reaching conclusive decisions, they have some measurable disadvantages. Oftentimes, these trees are sensitive to changes or fluctuations in data as a result of new features or variance in data spawning numerous new nodes and branches. Furthermore, decision trees can easily overfit the data they train on if the depth and complexity of the tree they create is significant enough, leading them to fail at generalizing data when used on unseen data.

3.3.2 K-Neighbors Classification

K-Neighbors is a supervised machine-learning algorithm that can be used for classification. It works by finding the distances between a query (problem) and all the examples it was trained on and then looking at the specified number of neighbors and assigning the most frequent label based on a simple majority vote.

A basic k-neighbors classification uses uniform weights where each neighbor has an equal value irrespective of the distance from the query. In some instances, it might be better to have weighted neighbors based on distance. Additionally, K-Neighbors can utilize different algorithms based on the use case: Brute Force, K-D Tree, or Ball Tree.

Brute force works well for smaller datasets and quickly becomes inefficient when used on a larger dataset. K-D Tree helps address the inefficiencies of the brute-force approach by using tree structures to reduce the number of distance calculations. A shortcoming of using K-D Trees is that it becomes inefficient for larger neighbor searches. To address this, the Ball Tree data structure can be used instead which works by partitioning data in a series of nesting hyper-spheres.

One of the major problems with using K-neighbors is that it becomes significantly slower as the size of the data grows. This can be tackled by reducing k value and running the algorithm on multiple threads/processes to speed up computation or by specifying one of the algorithms mentioned above that would fit the use case the best.

3.3.3 Multi-Layer Perceptron Classification (MLP)

Multi-layer perceptron (MLP) is a supervised deep learning algorithm that constructs an artificial neural network (ANN). As a whole, the main characteristic of deep learning and neural network-based algorithms is that they don't require omniscient or precise input to make meaningful predictions after training. Instead, neural networks are analogous to our understanding of the human brain and attempt to learn the characteristics of the data they're presented. Moreover, artificial neural networks all contain neurons that each represent some logic or decision that can be made in a task of classification or regression.

MLP progresses these ideas by introducing layers of neurons for input and output layers and an N number of non-linear layers between them. In MLP, each feature provided to the neural network is represented as a neuron in the input layer and each corresponding hidden layer is a neuron representing a particular transformation of the input state. In contrast, the output layer retrieves information from the *N-1th* hidden layer and transforms it into a potential output. MLP succeeds and avoids making purely linear decisions on the basis of weights that exist in each neuron which influence a neuron's state and the eventual output. This allows MLP to succeed at concisely learning the characteristics of non-linear sets of data.

The major drawback of MLP classifiers is that they often require tuning of the number of neurons and non-linear hidden layers in order to determine the most effective settings for a particular set of features since their performance is extremely sensitive to the parameters they're defined on [4]. Consequently, the results of MLP classifiers are also extremely sensitive to the methods used to scale the data it's provided.

4 Literature Review

Relevant literature suggests that decision trees as well as AdaBoost methods perform exceptionally well at predicting the existence of network intrusions as well as classifying an intrusion's relevant attack category. In *Sarker et al. [5]* a gini-index based decision tree is proposed using a minimal amount of features from the UNSW-NB15 dataset to reduce the complexity of the tree. As a result, the study proposes a model that effectively generalizes predictions while minimizing computational complexity, achieving a 0.98 F1 score on their test set. Furthermore, in *Ahmad et al. [6]* an AdaBoost classifier implementing a decision tree is used for classification tasks on the UNSW-NB15 dataset, also achieving F1 scores above 0.99 in the process. When comparing other models such as a support vector machine (SVM) as well as an artificial neural network, AdaBoost performs notably better.

In *Kasongo et al. [7]* a K-Neighbors method is employed on the UNSW-NB15 dataset with rounds of incrementing numbers of neighbors (3, 5, 7, 9, and 11). In their results, the model achieved an

83.18% accuracy for categorizing attacks using three neighbors and increased to 84.46% when using nine neighbors without overfitting.

Heuju et al. [8] employs MLP with a small number of hypertuned parameters using a Grid Search algorithm to perform classification on both the binary and categorical tasks in the UNSW-NB15 dataset. Specifically, the neural network employs an 'sgd' solver with 100 neurons in each layer and a learning rate of 1e-3. The neural network performed adequately in both binary and categorical classification (averaging around 80% for the classification of attacks) but results indicate that hyperparameter tuning could have occurred on a wider set of parameters.

5 Experiments and Analysis

5.1 Feature Selection

5.1.1 Recursive Feature Elimination (RFE)

During the feature selection process, we used RFE with five rounds of k-fold cross-validation to retrieve a set of optimal features from the dataset. Our use of the scikit-learn implementation for RFECV leveraged a RandomForest estimator with an entropy-based heuristic [8,9]. This method was picked because random forest ensemble techniques inherently measure feature importance well while also avoiding excluding variables from its prediction equation [10]. Moreover, using a random forest for estimation proved to be a significant performance improvement over other methods such as a support vector machine or a logistic regression-based estimator both of which would require additional computation – especially on large sets of data.

When requesting a minimum of one feature to select, our RFECV returned eight optimal features in relation to the provided training data from the UNSW-NB15 dataset. These features were *srcip*, *dstip*, *sbytes*, *sttl*, *dsport*, *dmeansz*, *Sintpkt*, and *ct_state_ttl* and each feature were equally ranked at the top after cross-validation.

Additionally, data collected from the results of the RFECV algorithm indicate that the optimal number of features exists between seven and nine, as shown in figure 1 where the test accuracy begins to plateau.

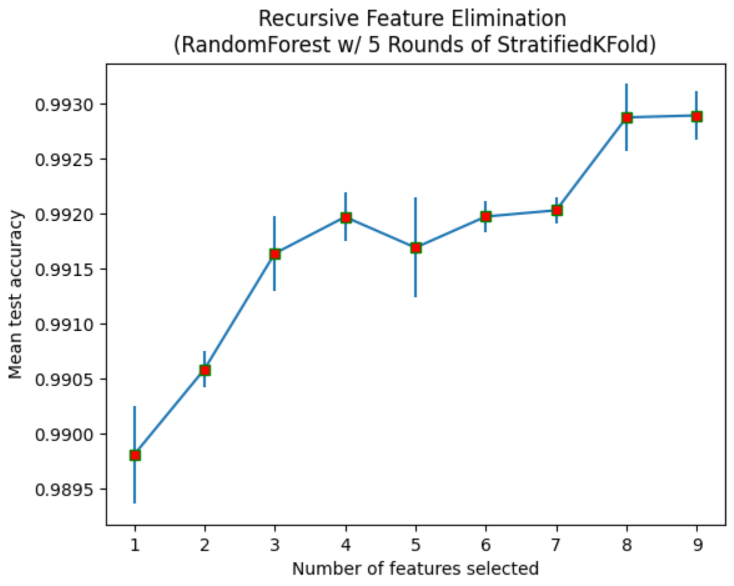

Figure 1: RFECV test accuracy increases until nine features, where accuracy plateaus

5.1.2 Sequential Feature Selection (SFS)

Sequential feature selection was used with a k-neighbors classifier to generate a set of features to be used in the dataset. Both implementations of the classifiers were retrieved from scikit-learn's SequentialFeatureSelector and KNeighborsClassifier [11,12].

The k-neighbors estimator was set up to look at k=3 nearest neighbors along with Forward-SFS setup to select the top 5 features. This particular feature selection method resulted in long run times due to having to train estimators on features through each iteration. One solution was to use another estimator such as a DecisionTreeClassifier which reduced run times significantly.

The 5 most important features returned through SFS were *dstip*, *sttl*, *dttl*, *swin* and *ct_state_ttl* while running it to select the top 10 features resulted with *dstip*, *proto*, *sttl*, *dttl*, *Spkts*, *swin*, *dwin*, *ct_state_ttl* which has overlap with the other feature selection methods and helped contribute to the features that ended up being selected for our classification models.

5.1.3 Correlation Feature Selection

With correlation analysis, we can identify patterns and relationships between variables within the dataset. When used in conjunction with other feature selection methods, we can determine the relevance of specific features that are meaningful to our predictive model.

Using pandas' implementation of the *corr* function, we can calculate the correlation coefficient, using Pearson's method, for each variable to each other (figure 2) [13]. To find the correlation coefficient between a feature and a target variable (like *Label*), we can use the function *corrwith*, and sort them by their absolute values to find the features with the highest correlation to the target variable [14].

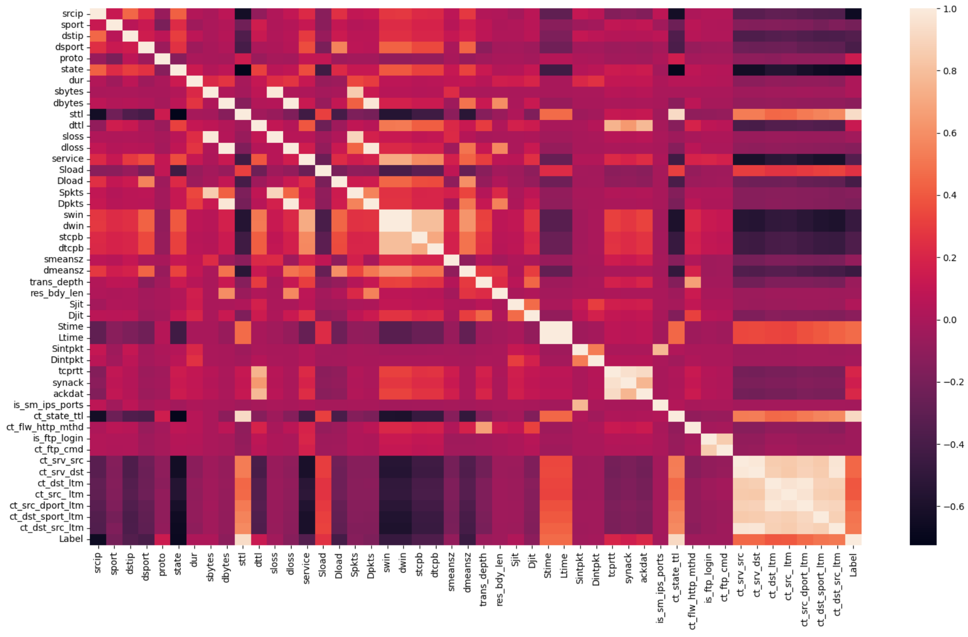

Figure 2: Diagram of correlation between all features with each other

5.1.4 Final Feature Set

Between the results of RFE, SFS, and insight from the correlation between features in the dataset, we selected nine features from the dataset for each classification task: *srcip*, *dstip*, *sbytes*, *sttl*, *dsport*, *dmeansz*, *Sintpkt*, *swin*, and *ct_state_ttl*. We decided to leverage the intersection of features between the three methods while adhering to RFE's prediction for the optimal number of features being between seven and nine.

Table 1: Selected features and their representation in the UNSW-NB15 dataset

Name	Type	Description
srcip	nominal	Source IP address
dstip	nominal	Destination IP address
dsport	integer	Destination port number
sbytes	integer	Source to destination transaction bytes
sttl	integer	Source to destination time to live value
swin	integer	Source TCP window advertisement value
dmeansz	integer	Mean of the flow packet size transmitted by the dst
Sintpkt	float	Source interpacket arrival time (mSec)
ct_state_ttl	integer	No. for each state (6) according to specific range of values for source/destination time to live (10) (11)

5.2 Classification Methods

5.2.1 AdaBoost Classification

Without feature selection, AdaBoost fared well in both the classification of *Label* and *attack_cat*. Interestingly enough, classification of the *Label* remained relatively the same with and without feature selection applied, with only subtle increases in accuracy when the former was applied during training.

Both classification tasks specifically showed the best results when an entropy based decision tree served as the basis for the adaptive boosting algorithm. When combined with feature selection, the use of entropy presumably allowed more important decisions to propagate to the root of the tree faster. Moreover, upon sequential stages of boosting each "edge case" or more difficult case in the tree would have a chance to be tuned with greater success than if a traditional decision tree were to be used. While the entropy based decision tree was a standout parameter, the other parameters that suited this classification task included the *SAMME.R* algorithm for boosting, 50 estimators, and a learning rate of 1.

Table 2: Classifier - AdaBoost (Label Task - 30.137s)

	precision	recall	f-1 score	support
0	0.99	0.99	0.99	60629
1	0.99	0.99	0.99	60795
accuracy			0.99	121424
macro avg	0.99	0.99	0.99	121424
weighted avg	0.99	0.99	0.99	121424

For classifying *attack_cat*, the same parameters were used to a similar effect. Despite the difficulty in classifying the attack category, the model still retained macro F1 scores well above 0.45 and hit as high as 0.67 on the withheld test and validation sets of data.

Table 3: Classifier - AdaBoost (attack_cat Task - 52.817s)

	precision	recall	f1-score	support
None	0.99	0.99	0.99	60629
Generic	1.00	0.99	0.99	40727
Exploits	0.68	0.80	0.74	8507
Reconnaissance	0.90	0.78	0.83	2653
DoS	0.39	0.42	0.41	2945
Fuzzers	0.85	0.81	0.83	4721
Shellcode	0.91	0.84	0.88	290
Analysis	0.47	0.13	0.21	483
Backdoors	0.66	0.11	0.19	447
Worms	0.53	0.73	0.62	22
accuracy			0.94	121424
macro avg	0.74	0.66	0.67	121424
weighted avg	0.95	0.94	0.94	121424

5.2.2 K-Neighbors Classification

For this classification, *k* was set to look at 5 nearest neighbors. In subsequent experiments to improve performance, no noticeable changes were seen by increasing k to 10 or 15 neighbors, or by specifying a weights metric to neighbors (weights calculated by the distance between neighbors and query) for both *Label* and *attack_cat* classifiers.

K-neighbors performed well for *Label* classification tasks in both cases of with and without feature selection, with a score of 0.99 for micro and macro F1 scores on the held-out test set.

Table 4: Classifier - K-Neighbors (Label Task - 15.563s)

	precision	recall	f-1 score	support
0	1.00	0.99	0.99	60629
1	0.99	1.00	0.99	60795
accuracy			0.99	121424
macro avg	0.99	0.99	0.99	121424
weighted avg	0.99	0.99	0.99	121424

For the *attack_cat* classification tasks, without feature selection, the model achieved a 0.49 macro f1 score. However, improvements were seen when selected features were applied resulting in a better score of 0.55.

Table 5: Classifier - K-Neighbors (attack_cat Task - 14.518s)

	precision	recall	f1-score	support
None	0.99	0.99	0.99	60629
Generic	0.99	0.98	0.98	40727
Exploits	0.62	0.77	0.69	8507
Reconnaissance	0.76	0.76	0.76	2653
DoS	0.34	0.27	0.30	2945
Fuzzers	0.80	0.79	0.79	4721
Shellcode	0.69	0.61	0.65	290
Analysis	0.27	0.16	0.20	483
Backdoors	0.16	0.01	0.02	447
Worms	1.00	0.05	0.09	22
accuracy			0.93	121424
macro avg	0.66	0.54	0.55	121424
weighted avg	0.93	0.93	0.93	121424

5.2.3 Multi-Layer Perceptron Classification (MLP)

Due to the requirement for numerous hyperparameters to be tuned on MLP-based classifiers in order to gain optimal performance, we decided to use a grid search cross-validation algorithm provided by scikit-learn to test numerous settings for the classifier on the training set [15]. In the end, we decided to use an MLP with 200 neurons in a single hidden layer as well as the *lbfgs* solver. After numerous tests, we found that these settings resulted in the best scores for both the *Label* and *attack_cat* tasks which both achieved over 0.90 and 0.45 for macro F1 respectively.

MLP is the only classification technique that had a noteworthy decrease in accuracy after many tests when feature selection was applied (0.03 difference). This may have been due to less data being available for neurons to optimize from or make decisions against.

Table 6: Classifier - Multi-Layer Perceptron (Label Task - 607.257s)

	precision	recall	f-1 score	support
0	1.00	0.99	0.99	60629
1	0.99	1.00	0.99	60795
accuracy			0.99	121424
macro avg	0.99	0.99	0.99	121424
weighted avg	0.99	0.99	0.99	121424

The time complexity of MLP is also significantly different than the other models we trained, presumably because of the large amount o data required for the neural network to learn and begin to perceive the characteristics of the dataset.

Table 7: Classifier - Multi-Layer Perceptron (attack_cat Task - 795.4206s)

	precision	recall	f1-score	support
None	1.00	0.99	0.99	60629
Generic	0.99	0.98	0.98	40727
Exploits	0.61	0.79	0.69	8507
Reconnaissance	0.62	0.67	0.64	2653
DoS	0.37	0.26	0.30	2945
Fuzzers	0.77	0.80	0.78	4721
Shellcode	0.73	0.77	0.75	290
Analysis	0.35	0.11	0.16	483
Backdoors	0.33	0.01	0.01	447
Worms	0.00	0.00	0.00	22
accuracy			0.93	121424
macro avg	0.58	0.54	0.53	121424
weighted avg	0.93	0.93	0.93	121424

5.2.4 Discussion

Across classification methods used in the experiment, all methods reported F1 scores around 0.99 on both the unseen test and validation sets. As a result, all classifiers could determine the existence of network intrusions to an exceptional level. In the dataset, classifying the *Label* feature was easier than the *attack_cat* feature, not only because of the reduced complexity involved with binary classification but also because of how evenly distributed genuine attacks are from normal network activity. This allows each model to more comprehensively train with data for each case and significantly improves its accuracy when making predictions on the withheld test and validation sets.

Table 8: Combined results for evaluation metrics across classification methods

	Accuracy	**Macro Average (F1)**	**Weighted Average (F1)**
AdaBoost	0.94	0.67	0.94
K-Neighbors	0.93	0.55	0.93
Multi-layer Perceptron	0.93	0.53	0.93

When performing categorical classification of *attack_cat*, the AdaBoost classifier did the best. In testing, the model often made predictions that reached macro F1 scores of 0.67 on both the unseen validation and test sets. This classifier did the best as a result of the concise set of features selected during the feature selection process which led to a boosted decision tree classifier that generalized the network intrusion data well. In fact, experimentation outside the scope of this report found that the generic decision tree classifier provided by scikit-learn [16] also performed exceptionally well as the target features were narrowed down. The complete comparison of accuracy, precision, recall, and f-1 scores for each classification method on the *attack_cat* task (for the withheld test set) can be found in Appendix A.

In general, the *attack_cat* classifier proved to be challenging to classify, especially considering the near 10% decrease in F1 scores between AdaBoost and the other two classification techniques. While the *attack_cat* task is categorical and increases the level of difficulty for the classification task, this challenge can be attributed further to the innately poor distribution of classes within the UNSW-NB15 dataset. For instance, after reviewing the classification reports for all three of our models, it's evident that *None* and *Generic* attacks had a significantly greater amount of occurrences in the dataset than categories such as *Worms* and *Backdoors*, which skewed the training process of each model. From a preprocessing perspective, classifying the attack category was also made more challenging by the inconsistent formatting and spelling of the various attack categories. Fluctuations in how the data was entered and or received by the model had a greater chance of being marked as *Unknown* (or -1) by our model if it didn't recognize or hadn't previously trained with an understanding of a class encountered in the dataset.

In our solution, each experiment handled training and inference models separately by pickling the trained model. The shape of the pickled model is in the form of a *ClassificationModels* class (line 243 of *nids.py*) and includes information for the model itself, name of the classification method, its training information, as well as the mappings used to factorize the categorical data during its training. Once trained, models are immediately tested using the isolated 30% of the training data and are pickled. When reloaded as an inference model, the passed dataset is factorized according to the loaded model's mappings and the model itself is fed the dataset directly to its feature variables. Once the validation data is factorized, preprocessed, and replaced in the model class, predictions can be instantly made on the requested dataset by the trained model.

6 Contributions

6.1 Jake (60%)

- Conducted the report's literature review surrounding the classification methods used
- Implemented in code and wrote about the *AdaBoost* and *Multi-layer Perceptron* classifiers as classification models in the project
- Implemented *Recursive Feature Elimination* as one of the feature selection techniques
- Implemented pickling for running inference models and properly setting up/factorizing validation data
- Added facilities in code for tuning hyperparameters via grid search cross validation
- Organized and implemented the *Makefile* with commands required to train and validate inference models

6.2 Mohammed (20%)

- Researched and added code for *Sequential Feature Selector*
- Added *K-Neighbors Classifier* as one of the classification models
- Worked on solving the factorization issue on trained models

6.3 Keith (20%)

- Researched *Correlation Feature Selection* methods
- Implemented *Correlation Feature Selection* as one of the feature selection techniques
- Worked on the final report write up

6.4 Obstacles and Roadblocks

Our main obstacles throughout the project involved having a sufficient understanding of the classification methods being used as well as how we could adequately verify their results. Early on in the project, we employed many classifiers at once (support vector, logistic regression, naive Bayes, stochastic gradient descent, etc.) and we weren't sure how to prove which classifiers provided meaningful data. Furthermore, gaining an in-depth understanding of what each classifier did and how it could critically impact or prove beneficial surrounding the UNSW-NB15 dataset proved to be an additional challenge. For some models, trying different parameters was time-consuming (refer to the runtimes for MLP in table 7) and it was difficult to generate an effective procedure for hyperparameter tuning and interpreting the data as a whole.

Finally, the main obstacle in code for the project was understanding the processes involved with data preprocessing and normalization. In the current script, we currently do not meet the requirement of avoiding normalization on test data. This being left in the code stemmed from having numerous crashes and unhandled errors on unseen data and spending too much time trying to debug and avoid normalization on unseen test and validation sets.

References

[1] Moustafa, N., & Slay, J. (2015, November). UNSW-NB15: a comprehensive data set for network intrusion detection systems (UNSW-NB15 network data set). In 2015 military communications and information systems conference (MilCIS) (pp. 1-6). IEEE.

[2] Sokolova, Marina & Japkowicz, Nathalie & Szpakowicz, Stan. (2006). Beyond Accuracy, F-Score and ROC: A Family of Discriminant Measures for Performance Evaluation. AI 2006: Advances in Artificial Intelligence, Lecture Notes in Computer Science. Vol. 304. 1015-1021. 10.1007/11941438_114.

[3] Freund, Y., & Schapire, R.E. (1996). Experiments with a New Boosting Algorithm. In International Conference on Machine Learning.

[4] scikit-learn 1.2.1 Documentation. (2023). 1.17 Neural network models (supervised). `https://scikit-learn.org/stable/modules/neural_networks_supervised.html#neural-networks-supervised`.

[5] Sarker, I. H., Abushark, Y. B., Alsolami, F., & Khan, A. I. (2020). IntruDTree: A Machine Learning Based Cyber Security Intrusion Detection Model. Symmetry, 12(5), 754. MDPI AG. Retrieved from `http://dx.doi.org/10.3390/sym12050754`.

[6] Ahmad, Iftikhar & Haq, Qazi & Imran, Muhammad & Alassafi, Madini & AlGhamdi, Rayed. (2022). An Efficient Network Intrusion Detection and Classification System. Mathematics. 10. 530. 10.3390/math10030530.

[7] Kasongo, S.M., Sun, Y. Performance Analysis of Intrusion Detection Systems Using a Feature Selection Method on the UNSW-NB15 Dataset. J Big Data 7, 105 (2020). `https://doi.org/10.1186/s40537-020-00379-6`.

[8] scikit-learn 1.2.1 Documentation. (2023). sklearn.feature_selection.RFECV. `https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.RFECV.html#sklearn.feature_selection.RFECV`.

[9] scikit-learn 1.2.1 Documentation. (2023). sklearn.ensemble.RandomForestClassifier. `https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html`.

[10] Kuhn, Max & Johnson, Kjell. (2019). Feature Engineering and Selection. 10.1201/9781315108230. `https://bookdown.org/max/FES/`.

[11] scikit-learn 1.2.1 Documentation. (2023). sklearn.feature_selection.SequentialFeatureSelector. `https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.SequentialFeatureSelector.html`.

[12] scikit-learn 1.2.1 Documentation. (2023). sklearn.neighbors.KNeighborsClassifier. `https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html`.

[13] pandas 1.4.1 Documentation. (2023). pandas.DataFrame.corr. `https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.corr.html`

[14] pandas 1.4.1 Documentation. (2023). pandas.DataFrame.corrwith. `https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.corrwith.html`.

[15] scikit-learn 1.2.1 Documentation. (2023). sklearn.model_selection.GridSearchCV. `https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html`.

[16] scikit-learn 1.2.1 Documentation. (2023). sklearn.neighbors.DecisionTreeClassifier. `https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html`.

A Appendix

A.0.1 Comparison of Classification Methods by Evaluation Metrics

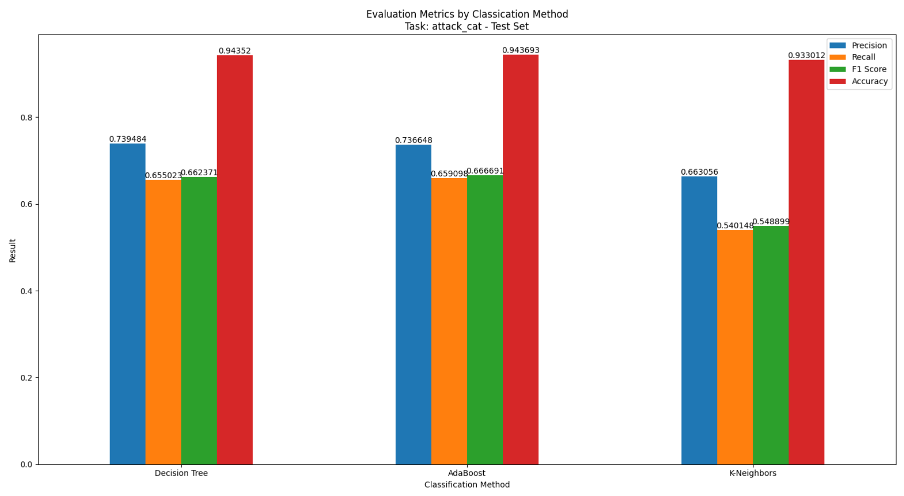

Figure 3: Comparison of evaluation metrics amongst classification methods