Intrusion Detection Systems (IDSs) using NSL-KDD Dataset

Ameer Nasrallah

2022-03-09

Contents

1 Introduction

An intrusion detection system (IDS) is a system that automatically checks and analyzes network flow to detect and prevent abnormal activities [1]. This includes monitoring both user and system behaviors, such as the unauthorized access to network resources, and the analysis of network packet fields (e.g. IP address, flag, ports) [2]. Upon detecting an intrusion, the IDS alarms it to the management [3].

An IDS can be classified based on the detection mechanism to knowledge-based [1] and behavior-based [2]. In knowledge-based (or signature-based) detection, the IDS uses misuse detection to detect known attacks by comparing between received packets and a predefined set of collected data (e.g. signature files). While in behavior-based (or anomaly-based) detection, the IDS uses anomaly detection to detect unknown attacks by comparing the system state with the normal activity profile it builds. Anomaly-based detection suffers from high false alarm rates. Despite that, it is still considered better than knowledge-based detection as it can detect novel or zero-day attacks. Hence, anomaly-based detection got more attention during the past twenty years, and there have been many research efforts to apply machine learning techniques in intrusion detection.

The goal of this project is to build two anomaly-based IDSs. The first system is a binary classification system that will classify a TCP connection (defined by certain attributes such as the port, protocol, etc) as either a normal activity or as an attack. While the second system will be a multi-class classification system that will classify the TCP connection as a normal activity or as one of four known attacks (DoS, Probing, R2L or U2R). Both systems will be built using NSL-KDD dataset [4] which provides the predictions of about 150000 simulated TCP connections.

We will evaluate the performance of each system using six classifiers (Recursive Partitioning, Naive Bayes, KNN, SVM, Random Forest, and Multi-Layer Perceptron), given that the data is already divided into training and testing sets (about 15% of the data). For each classifier, we will apply a tune grid search with 10-fold cross validation on the training set only to find the best tuned parameters. Then, the performance of each tuned classifier will be measured by predicting the testing set output. We will report different metrics, but we will focus mainly on three of them: overall accuracy (max), detection rates (max), and false alarm rates (min).

The rest of this report is organized as follows. We will start by studying the characteristics of NSL-KDD dataset. Then, we will explain the the methodology. Then, we will analyze the training and performance of the binary classification IDS. Then, we will analyze the training and performance of the multi-class classification IDS.. And finally, we will conclude all the work.

2 NSL-KDD Dataset Characteristics

NSL-KDD dataset [4] is one of the most effective datasets in the domain of intrusion detection. It is a modified version of KDDCUP'99 dataset [5], which was created in 1999. KDDCUP'99 is constructed from simulated TCP connections in a military network environment [6]. KDDCUP'99 had been the most widely used dataset to evaluate IDSs until recent years [7]. However, researchers found some deficiencies that make it less reliable [4]:

1. Redundant records: this mainly affects the performance of any classifier such that it is biased towards more frequent records.
2. Low difficulty level: applying simple machine learning methods will give at least 86% accuracy, which makes it difficult to compare the different models as they will fall in the range of 86% to 100%.

To deal with these deficiencies, the following improvements were applied to NSLKDD [4]:

1. Removing all redundant records from train and test sets so that there will be no biasing.

2. Better sampling and distribution for the records which will increase the classification challenge.
3. Reasonable number of records in train and test sets. This makes it affordable to run experiments on the whole dataset without any need for sampling.

NSL-KDD still does not perfectly represent real networks. Nonetheless, it is still a reliable benchmark dataset to compare intrusion detection methods.

2.1 Attacks Categories

NSL-KDD records are labeled as normal or attack. There are 39 different attacks distributed (with some overlap) as 22 attacks in the training set and 37 attacks in the testing set. These attacks fall into four basic categories detailed as follows:

- Denial of Service Attack (DoS): involves attacks which try to keep the machine's memory or computing resources too busy such that the machine cannot serve its legitimate users.
- User to Root Attack (U2R): involves attacks in which the attacker first gains access to a normal user account, and then tries to exploit some vulnerability to gain root access to the system.
- Remote to Local Attack (R2L): involves attacks in which attacker keeps sending packets to a machine over some network. The main purpose in these attacks is to try to find a system vulnerability to gain access as a normal user.
- Probing Attack: these attacks scan the computer networks to find some vulnerability in its security controls.

Table 1 shows the detailed distribution of the different attacks.

Table 1: NSL-KDD Attacks Categories

Attack Category	**Attacks Included**
DoS	neptune, back, land, pod, smurf, teardrop, mailbomb, apache2, processtable, udpstorm, worm
Probing	ipsweep, nmap, portsweep, satan, mscan, saint
R2L	ftp_write, guess_passwd, imap, multihop, phf, spy, warezclient, warezmaster, sendmail, named, snmpgetattack, snmpguess, xlock, xsnoop, httptunnel
U2R	buffer_overflow, loadmodule, perl, rootkit, ps, sqlattack, xterm

Figure 1 shows the distribution of normal and attack connections in NSL-KDD training and testing sets. It is clear that the training set is divided into ~50% normal connections and ~50% for the attacks. Moreover, DoS attack category is dominating the attacks the training set, while R2L and U2R attack categories have few connections. Consequently, it will be challenging to correctly classify R2L and U2R records.

2.2 Features Description

Moreover, NSL-KDD is constructed from 41 attributes or features. These features fall into three main categories as shown in Table 2:

1. Basic features: these attributes are extracted from a TCP/IP connection.
2. Content features: these attributes are extracted from the data portion of the packet. They are very important to detect R2L and U2R attacks. This is because these attacks usually involve a single connection.

3. Traffic features:

   - Time-based traffic features: these attributes are extracted from connections in the past two seconds that have the same destination or same service as current connection.
   - Connection-based traffic features: these attributes are extracted from last 100 connections that has the same destination or same service as current connection. Extracting such attributes contributes more in detecting probing attacks.

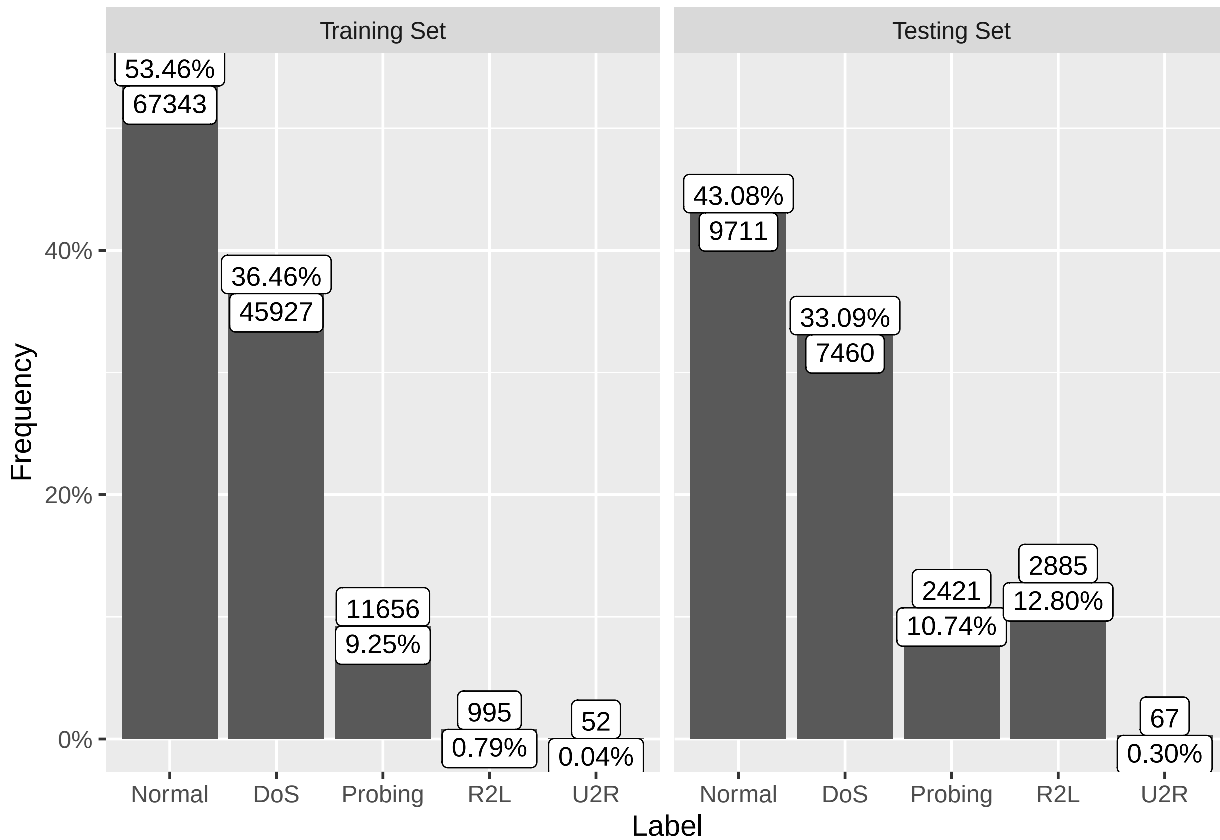

Figure 1: Distribution of Normal/Attacks in NSL-KDD Training and Testing Sets

Table 2: NSL-KDD Attributes

Category	Feature	Type	Range/Values
Basic features	duration	integer	0-42908
	protocol_type	character	tcp, udp, icmp
	service	character	ftp_data, other, private, http, remote_job, name, netbios_ns, eco_i, mtp, telnet, finger, domain_u, supdup, uucp_path, Z39_50, smtp, csnet_ns, uucp, netbios_dgm, urp_i, auth, domain, ftp, bgp, ldap, ecr_i, gopher, vmnet, systat, http_443, efs, whois, imap4, iso_tsap, echo, klogin, link, sunrpc, login, kshell, sql_net, time, hostnames, exec, ntp_u, discard, nntp, courier, ctf, ssh, daytime, shell, netstat, pop_3, nnsp, IRC, pop_2, printer, tim_i, pm_dump, red_i, netbios_ssn, rje, X11, urh_i, http_8001, aol, http_2784, tftp_u, harvest
	flag	character	SF, S0, REJ, RSTR, SH, RSTO, S1, RSTOS0, S3, S2, OTH
	src_bytes	integer	0-1379963888
	dst_bytes	integer	0-1309937401
	land	integer	0,1
	wrong_fragment	integer	0-3
	urgent	integer	0-3
Content features	hot	integer	0-77
	num_failed_logins	integer	0-5
	logged_in	integer	0,1
	num_compromised	integer	0-7479
	root_shell	integer	0,1
	su_attempted	integer	0-2
	num_root	integer	0-7468
	num_file_creations	integer	0-43
	num_shells	integer	0-2
	num_access_files	integer	0-9
	num_outbound_cmds	integer	0-0
	is_host_login	integer	0,1
	is_guest_login	integer	0,1
Time-based traffic features	count	integer	0-511
	srv_count	integer	0-511
	serror_rate	numeric	0-1
	srv_serror_rate	numeric	0-1
	rerror_rate	numeric	0-1
	srv_rerror_rate	numeric	0-1
	same_srv_rate	numeric	0-1
	diff_srv_rate	numeric	0-1
	srv_diff_host_rate	numeric	0-1
Connection-based traffic features	dst_host_count	integer	0-255
	dst_host_srv_count	integer	0-255
	dst_host_same_srv_rate	numeric	0-1
	dst_host_diff_srv_rate	numeric	0-1
	dst_host_same_src_port_rate	numeric	0-1
	dst_host_srv_diff_host_rate	numeric	0-1
	dst_host_serror_rate	numeric	0-1
	dst_host_srv_serror_rate	numeric	0-1
	dst_host_rerror_rate	numeric	0-1
	dst_host_srv_rerror_rate	numeric	0-1

3 Methodology

We will build two IDSs, one for binary classification and the other for multi-class classification. So, the training and testing sets for the first system will have the `label` as `Normal` or `Attack`, while for the second system the `label` will be `Normal` or `DoS` or `Probing` or `R2L` or `U2R`. The performance of each system will be evaluated using six classifiers: Recursive Partitioning, Naive Bayes, KNN, SVM, Random Forest, and Multi-Layer Perceptron. Each classifier needs to be tuned before training it on the whole training set. Therefore, we chose to apply a tune grid search with 10-fold cross validation on each pair of system and classifier. As a result, we will have the best tuned parameters for each classifier to the relevant system. Then, we will train the best tuned classifiers on the relevant training set. Finally, we will apply the prediction on the relevant testing set, and we will report the performance metrics. The following algorithm summarizes the steps followed to tune any classifier.

Algorithm 1: Tune Grid Search with 10-fold Cross-Validation Algorithm

```
Define sets of model parameter values to evaluate
Shuffle training set randomly
Split training set into 10 groups
for each parameter set do
    for each group do
        Take the group as a hold out or test data set
        Take the other 9 groups as a training data set
        Preprocess the training data
        Fit the model on the training data
        Predict the hold out data
    end
    Calculate the average performance across predictions of hold out data
end
Determine the optimal parameter set
Preprocess the whole training set
Fit the final model to the whole training set using the optimal parameter set
```

In the next sections, we will go through the preprocessing stage, the classifiers used, and the evaluation metrics.

3.1 Preprocessing Stage

As noticed from the previous algorithm, we preprocess the data before any training step. This includes both the intermediary training steps performed within cross validation, and the final training step using the optimal tuned parameters. Moreover, the result of the preprocessing stage will be preserved to apply it on any prediction step. For example, if in the preprocessing of the first group `is_host_login` feature was removed as it has zero-variance in that training group, the same feature will be removed from the relevant testing group. Note that removing `is_host_login` feature from one of the groups does not mean it will be removed from the whole training set, because in the full training set it might not have zero-variance.

We chose to apply three main steps in the preprocessing stage which are: removing zero-variance numerical features, normalizing the rest of numerical features, and transforming the nominal features to be numerical. To apply these steps, we decided to use `recipes` package [8] which provides powerful preprocessing capabilities. In the next subsections, we will explain briefly each of these steps.

3.1.1 Remove Zero-Variance Numerical Features

A zero-variance feature is a feature that has only one static value. These features are usually removed before the training because they are seen as having no impact on the output. To apply removing zero-variance features, we used `step_zv` function from `recipes` package. This is an example of how to use `step_zv`.

```
step_zv(all_numeric_predictors())
```

3.1.2 Normalize Numerical Features using Min-Max

Another important step before working with classification algorithms is to normalize numeric features. Normalizing a feature means to scale its values to fall into a smaller range. For example, there are features in NSL-KDD dataset that have wide range of values, such as: `duration`, `src_bytes` and `num_root`. While there are other features that have smaller range of values, such as: `num_failed_logins`, `is_host_login`, and `srv_count`. Keeping the features without normalization may cause biasing towards selecting wide range features which may also affect classification performance. To prevent this dominance, we chose to scale all the numeric features to fall in the range of 0-1 using min-max normalization. Min-max scaling is shown in Equation (1), where x is the value to be scaled in feature X, $MinMax(x)$ is the scaled value of x, $Min(X)$ and $Max(X)$ are the minimum and maximum values respectively in feature X, min and max are the boundaries of the new range.

$$MinMax(x) = min + (max - min)(\frac{x - Min(X)}{Max(X) - Min(X)}) \tag{1}$$

To apply normalizing numerical features, we used `step_range` function from `recipes` package. This is an example of how to use `step_range`.

```
step_range(all_numeric_predictors())
```

3.1.3 Transform Nominal Features using Probability Density Function (PDF)

Many classification algorithms are mathematical-based. Therefore, it is important to transform the nominal features of a dataset into their numerical representation. NSL-KDD dataset has three nominal features (as stated in Table 2): `protocol_type`, `service`, and `flag`.

To avoid biasing the data, we did not encode the data with a static value map (e.g. http takes 1, smtp takes 2, and so on). Rather, we applied probability density function as in Equation (2) [9] such that the most frequent nominal value in a column takes the highest numerical value while still being bounded between 0 and 1. This range goes along with the numerical features normalization.

$$PDF(x) = \frac{occur(x)}{n} \tag{2}$$

where $occur(x)$ is the number of occurrences of value x within a column, and n is the total number of records.

To apply transforming nominal features, we had to build a custom `step` according to official `recipes` documentation[1] as shown below. We called this step `step_nominalpdf`.

```
############################################################
# Normalize PDF Recipe Step Definition
# References: http://cran.nexr.com/web/packages/recipes/vignettes/Custom_Steps.html
#             https://github.com/tidymodels/recipes/blob/main/R/center.R
############################################################

step_nominalpdf <-
```

[1] http://cran.nexr.com/web/packages/recipes/vignettes/Custom_Steps.html

```r
  function(recipe,
           ...,
           role = NA,
           trained = FALSE,
           ref_dist = NULL,
           skip = FALSE,
           id = rand_id("nominalpdf")) {
    add_step(
      recipe,
      step_nominalpdf_new(
        terms = enquos(...),
        trained = trained,
        role = role,
        ref_dist = ref_dist,
        skip = skip,
        id = id
      )
    )
}

step_nominalpdf_new <-
  function(terms, role, trained, ref_dist, skip, id) {
    step(
      subclass = "nominalpdf",
      terms = terms,
      role = role,
      trained = trained,
      ref_dist = ref_dist,
      skip = skip,
      id = id
    )
}

prep.step_nominalpdf <- function(x, training, info = NULL, ...) {
  col_names <- recipes_eval_select(x$terms, training, info)

  ref_dist <- list()
  train_ln <- nrow(training)
  for (i in col_names) {
    # For each column, table will return the count of each value
    # to normalize that count, we divide it by the number of rows
    # For example, if we have (a, b, a, c, d) in a column
    # The output will be a table like this
    #   a   b   c   d
    # 0.4 0.2 0.2 0.2
    ref_dist[[i]] <- table(training[, i]) / train_ln
  }

  ## Always return the updated step
  step_nominalpdf_new(
    terms = x$terms,
    role = x$role,
    trained = TRUE,
```

```r
    ref_dist = ref_dist,
    skip = x$skip,
    id = x$id
  )
}

pdf_by_ref <- function(x, ref) {
  # if we have the following values in ref:
  #   a   b   c   d
  # 0.4 0.2 0.2 0.2
  # And we got x = "a", the function will return 0.4
  # if we got x = "e", the function will return 0
  ifelse(x %in% names(ref), ref[x][[1]], 0)
}

bake.step_nominalpdf <- function(object, new_data, ...) {
  require(tibble)
  vars <- names(object$ref_dist)

  # Transform the columns
  for(i in vars) {
    new_data[, i] <- apply(new_data[, i], 1, pdf_by_ref, ref = object$ref_dist[[i]])
  }
  ## Always convert to tibbles on the way out
  tibble::as_tibble(new_data)
}

print.step_nominalpdf <- function(x, width = max(20, options()$width - 30), ...) {
  cat("PDF for ", sep = "")
  printer(names(x$ref_dist), x$terms, x$trained, width = width)
  invisible(x)
}

tidy.step_nominalpdf <- function(x, ...) {
  if (is_trained(x)) {
    res <- tibble(terms = names(x$ref_dist),
                  value = unname(x$ref_dist))
  } else {
    term_names <- sel2char(x$terms)
    res <- tibble(terms = term_names,
                  value = na_dbl)
  }
  res$id <- x$id
  res
}
```

This is an example of how to use `step_nominalpdf`.

```r
step_nominalpdf(all_nominal_predictors())
```

3.2 Classifiers

In order to perform classification, we used caret package [10] which provides predefined classification algorithms in a convenient way. Table 3 summarizes the classifiers we used indicating the name of the classifier in caret along with its original package (which can be caret itself as for knn).

Table 3: Classifiers Description

Classifier Name	Caret Name	Package::Function	Parameters
Recursive Partitioning	rpart	rpart::rpart [11]	cp
Naive Bayes	naive_bayes	naivebayes::naive_bayes [12]	laplace, usekernel, adjust
KNN	knn	caret::knn3	k
SVM	svmLinear	kernlab::ksvm [13]	C
Random Forest	parRF	randomForest::randomForest [14]	mtry
Multi-Layer Perceptron	mlp	RSNNS::mlp [15]	size

There is a great integration between caret and recipes packages which makes it too easy to build the whole training flow in few lines. Also, caret has strong support for parallel execution.[2] Here is a full example that uses caret, recipes and doParallel package [16].

```
# Define the recipe for the preprocessing steps
data_rec <- recipe(label ~ ., data = nsl_training_data) %>%
              step_zv(all_numeric_predictors()) %>%
              step_range(all_numeric_predictors()) %>%
              step_nominalpdf(all_nominal_predictors())
# Create a cluster
cluster = makePSOCKcluster(detectCores() - 2)
# Register the cluster
registerDoParallel(cluster)
# Register the functions related to step_nominalpdf to the cluster
clusterExport(cl=cluster, varlist=c("step_nominalpdf", "step_nominalpdf_new",
                                  "prep.step_nominalpdf", "pdf_by_ref",
                                  "bake.step_nominalpdf",
                                  "print.step_nominalpdf",
                                  "tidy.step_nominalpdf"), envir=environment())
set.seed(123)
# Train KNN on NSL-KDD data taking the recipe as input (default k for cv is 10)
# Use the default tune grid values for k (the tuning parameter of knn)
model_fit <- train(x = data_rec,
               data = nsl_training_data,
               method = "knn",
               trControl = trainControl(method = 'cv'))
stopCluster(cluster)
```

Also, caret supports convenient abstract way to perform prediction and compute the confusion matrix such as the following example.

```
model_pred <- predict(trained_model, testing_data)
confusionMatrix(model_pred, testing_data$label)
```

[2] http://topepo.github.io/caret/parallel-processing.html

3.3 Evaluation Metrics

The effectiveness of any IDS is mainly measured by [6] [17] [18]: overall accuracy, detection rate, false alarm rate, and training time. A well-performing IDS would achieve a low false alarm rate, and high accuracy and detection rate. The common way to derive the definition of these metrics is through a confusion matrix.

3.3.1 Metrics for Binary Classification IDS

In a binary classification IDS, a record that is labeled as “attack” is a “Positive” record, and a record that is labeled as “normal” is a “Negative” record. Confusion matrix is a two by two matrix that represents the four possible combinations of the actual records and the predicted records.

Table 4: Confusion matrix

		Predicted	
		Negative (normal)	Positive (attack)
Actual	Negative (normal)	TN	FP
	Positive (attack)	FN	TP

Table 4 shows a confusion matrix where:

- True Negative (TN): represents the number of normal records correctly predicted as normal.
- False Positive (FP): represents the number of attack records wrongly predicted as normal.
- False Negative (FN): represents the number of normal records wrongly predicted as attack.
- True Positive (TP): represents the number of attack records correctly predicted as attack.

Based on the confusion matrix, we can define the metrics mentioned above as follows:

- Overall Accuracy: is the percent of correctly classified records. It is calculated by Equation (3).

$$\text{Overall Accuracy} = \frac{TN + TP}{TN + FP + FN + TP} \tag{3}$$

- Detection Rate (DR): also called Recall or sensitivity or true positive rate (TPR). It is the percent of correctly classified attacks to the total number of actual attacks. When it is near 1, it means that the classifier performed well in predicting almost all actual attacks. It is calculated by Equation (4).

$$\text{Detection Rate (DR)} = \frac{TP}{TP + FN} \tag{4}$$

- False Alarm Rate (FAR): also called False Positive Rate (FPR). It is the percentage of wrongly classified normal records. When it is near zero, it means that the classifier performed well in avoiding misprediction of almost all normal records. It is calculated by Equation (5).

$$\text{FAR} = \frac{FP}{FP + TN} \tag{5}$$

3.3.2 Metrics for Multi-class Classification IDS

In a multi-class IDS, all metrics except overall accuracy need to be calculated per class. Therefore, we used a strategy called one-vs-rest, which treats each class as if it is the positive class that we want to detect, and the other classes are negative. We demonstrate this treatment by having an example of a 5x5 confusion matrix on NSL-KDD classes. This confusion matrix is shown in Table 5.

We will break this confusion matrix into 5 smaller matrices each of size 2x2. Tables 6-10 show the resulting confusion matrices for each class in NSL-KDD. And now, we can easily calculate the metrics as follows:

Table 5: Multi-class confusion matrix

		Predicted				
		Normal	DoS	Probing	R2L	U2R
Actual	Normal	21761	286	677	287	106
	DoS	1183	14535	209	74	15
	Probing	510	72	3550	68	23
	R2L	631	77	197	235	24
	U2R	31	1	1	2	1

- Overall Accuracy:
$$\frac{21761 + 14535 + 3550 + 235 + 1}{44556} = \frac{40082}{44556} = 0.8996$$
- Normal DR:
$$\frac{21761}{21761 + 1356} = \frac{21761}{23117} = 0.9413$$
- DoS DR:
$$\frac{14535}{14535 + 1481} = \frac{14535}{16016} = 0.9075$$
- Probing DR:
$$\frac{3550}{3550 + 673} = \frac{3550}{4223} = 0.8407$$
- R2L DR:
$$\frac{235}{235 + 929} = \frac{235}{1164} = 0.2019$$
- U2R DR:
$$\frac{1}{1 + 35} = \frac{1}{36} = 0.0278$$
- Normal FAR:
$$\frac{2355}{2355 + 19084} = \frac{2355}{21439} = 0.1098$$
- DoS FAR:
$$\frac{436}{436 + 28104} = \frac{436}{28540} = 0.0153$$
- Probing FAR:
$$\frac{1084}{1084 + 39249} = \frac{1084}{40333} = 0.0269$$
- R2L FAR:
$$\frac{431}{431 + 42961} = \frac{431}{43392} = 0.0099$$
- U2R FAR:
$$\frac{168}{168 + 44352} = \frac{168}{44520} = 0.0037$$

Table 6: Normal Confusion matrix

		Predicted	
		Other	Normal
Actual	Other	19084	2355
	Normal	1356	21761

Table 7: DoS Confusion matrix

		Predicted	
		Other	DoS
Actual	Other	28104	436
	DoS	1481	14535

Table 8: Probing Confusion matrix

		Predicted	
		Other	Probing
Actual	Other	39249	1084
	Probing	673	3550

Table 9: R2L Confusion matrix

		Predicted	
		Other	R2L
Actual	Other	42961	431
	R2L	929	235

Table 10: U2R Confusion matrix

		Predicted	
		Other	U2R
Actual	Other	44352	168
	U2R	35	1

4 Binary Classification IDS Analysis and Results

We applied the flow proposed in Section 3.2 to the six filters mentioned in Table 3 with the default tuning grid values from `caret` on the binary classification IDS. In this section, we will analyze the binary classification IDS training and its performance results.

4.1 Binary Classification Training Analysis

Table 11 shows the parameters tuning results for each classifier on the binary classification IDS. The accuracy in this table is computed as the average of the 10-folds for the best tuned parameters, it is not for the final model. Also, it is noticeable that random forest and SVM classifiers took the most significant time to tune the parameters.

Table 11: Binary Classification IDS Parameters Tuning

Classifier	**Best Tuned Parameters**	**Accuracy**	**Tuning Time (seconds)**
rpart	cp = 0.0263005287395531	0.9410906	69.927
naive_bayes	laplace = 0, usekernel = TRUE, adjust = 1	0.8903813	58.806
knn	k = 5	0.9953006	1367.774
svmLinear	C = 1	0.9549745	5379.580
parRF	mtry = 21	0.9991189	10722.264
mlp	size = 5	0.9876481	781.724

Figure 2 shows the training time of the best tuned binary classifiers. Again, random forest and SVM classifiers took much more time than other classifiers.

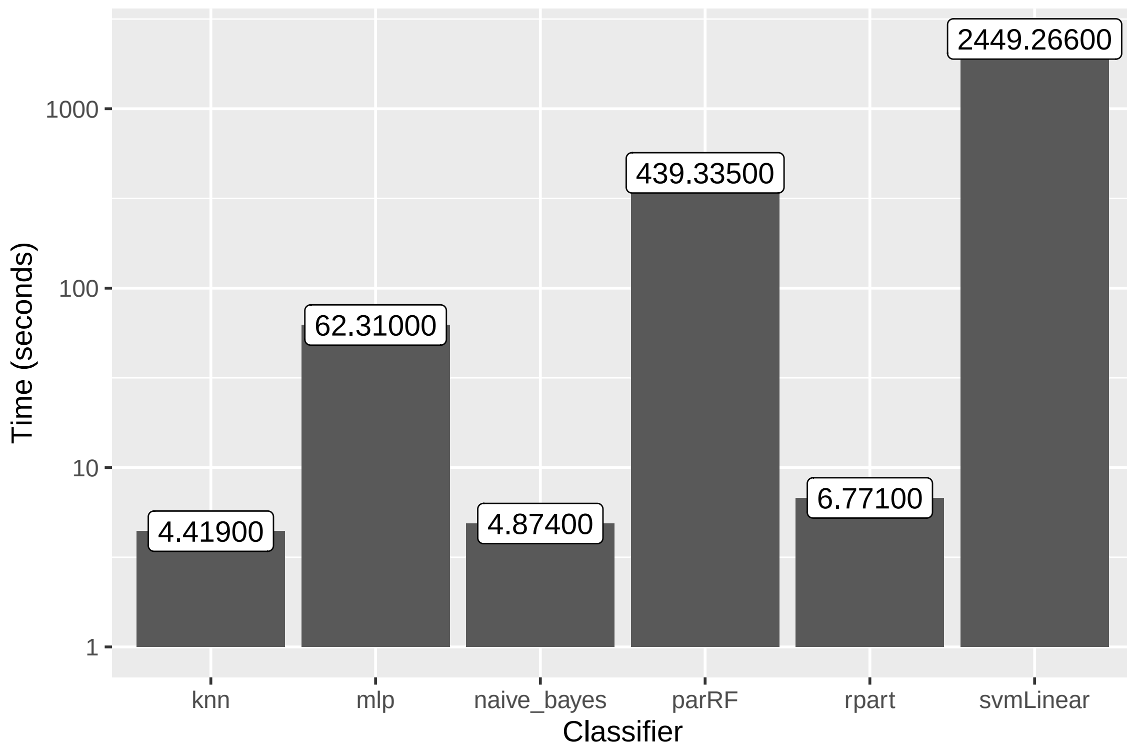

Figure 2: Training Time of Best Tuned Binary Classifiers

Some classifiers such as recursive partitioning `rpart` and random forest `parRF` classifiers have built-in feature importance calculation. Figures 3 and 4 show the top five important features for `rpart` and `parRF` respectively. They actually share the same top feature which is `src_bytes`. But `rpart` gave more importance to other features as well although they are less important in `parRF`, such as `dest_bytes`.

4.2 Binary Classification Performance

We computed the metrics mentioned in Section 3.3.1 using `confusionMatrix`. The accuracy of the binary classification IDS is shown in Figure 5. It is clear that `naive_bayes` had the worst performance with an accuracy of about 68%, while `parRF` had the best performance with an accuracy of about 79.8%. The other classifiers achieved almost the same accuracy as `parRF`.

In terms of the False Alarm Rate (FAR), Figure 6 shows that `mlp` had the worst highest FAR of about 7.58%, while `naive_bayes` had the best lowest FAR of about 0.175% (but with low accuracy). Worths mentioning that `parRF` had the second lowest FAR of about 3.007% while achieving the highest accuracy at the same time.

In terms of the Detection Rate (DR), Figure 7 shows that `naive_bayes` had the worst DR of about 44.21%, while `parRF` had the highest DR of about 66.7%.

Overall, the best classifier for binary classification IDS is Random Forest `parRF` as it achieved a reasonable balance between accuracy, FAR and DR. The only difficulty with `parRF` is that it takes long amount of time to train the model.

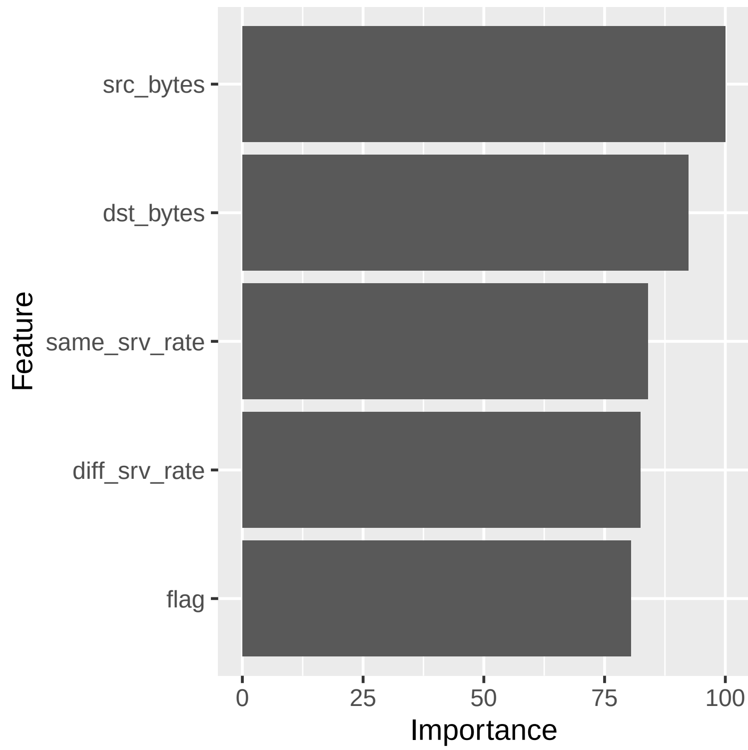

Figure 3: Feature Importance for Binary Classification using rpart

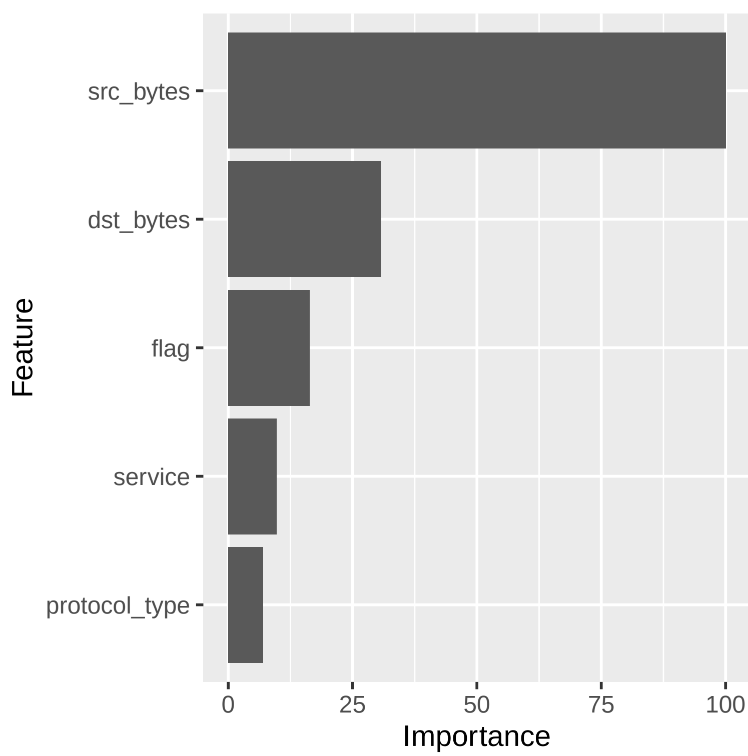

Figure 4: Feature Importance for Binary Classification using parRF

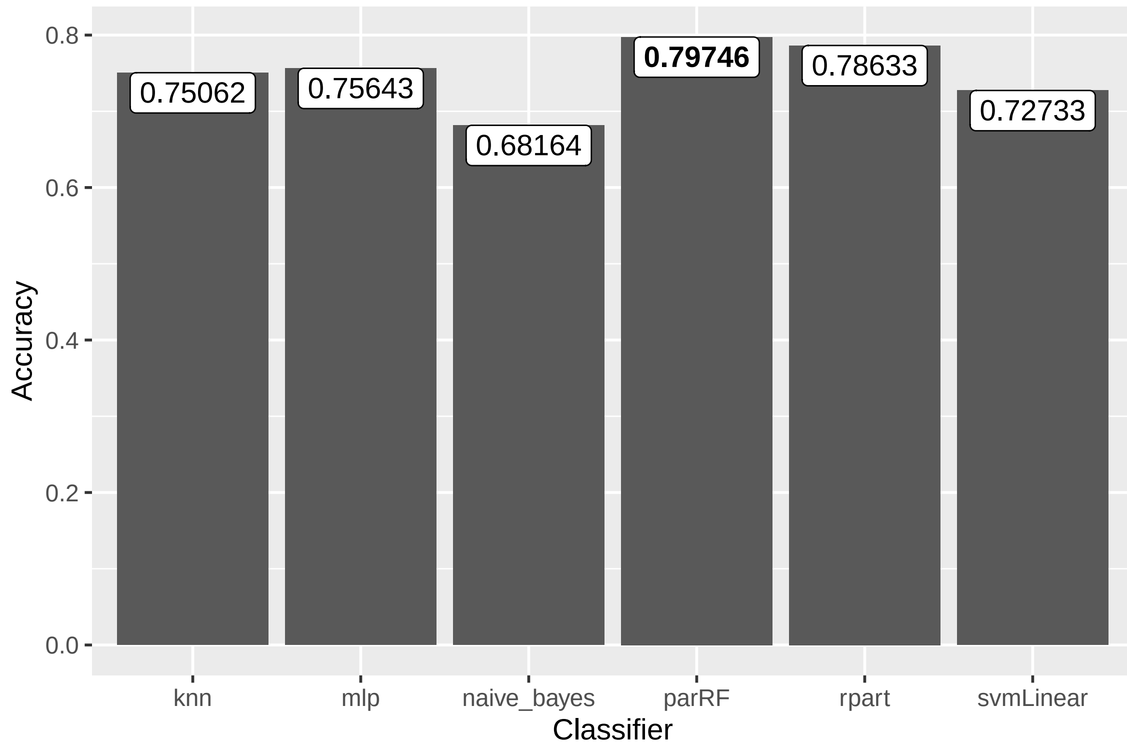

Figure 5: Accuracy of Binary Classfication classifiers

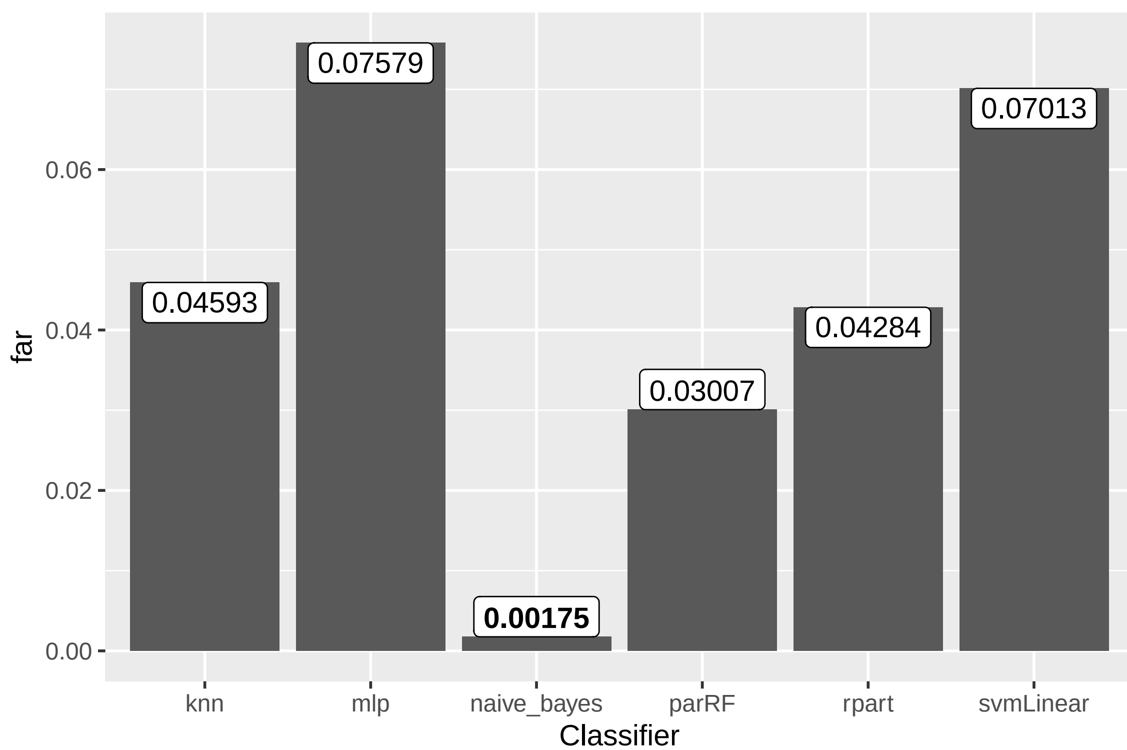

Figure 6: False Alarm Rate (FAR) of Binary Classfication classifiers

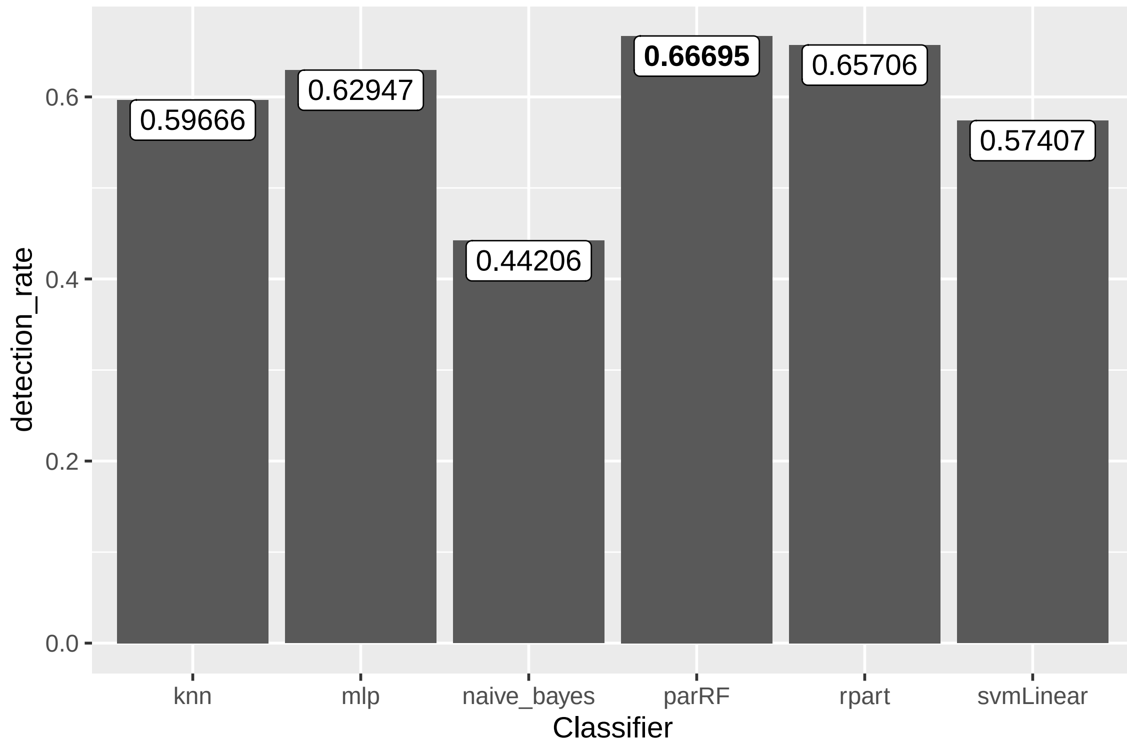

Figure 7: Detection Rate (DR) of Binary Classfication classifiers

5 Multi-class Classification IDS Analysis and Results

We applied the flow proposed in Section 3.2 to the six filters mentioned in Table 3 with the default tuning grid values from `caret` on the multi-class classification IDS. In this section, we will analyze the multi-class classification IDS training and its performance results.

5.1 Multi-class Classification Training Analysis

Table 12 shows the parameters tuning results for each classifier on the multi-class classification IDS. The accuracy in this table is computed as the average of the 10-folds for the best tuned parameters, it is not for the final model. Similar to the binary classification IDS, `parRF` and `svmLinear` took much more time than other classifiers.

Table 12: Multi-class Classification Tuning

Classifier	**Best Tuned Parameters**	**Accuracy**	**Tuning Time (seconds)**
rpart	cp = 0.0498891352549889	0.8998432	67.019
naive_bayes	laplace = 0, usekernel = TRUE, adjust = 1	0.8208108	61.512
knn	k = 5	0.9951101	1334.798
svmLinear	C = 1	0.9018706	1929.012
parRF	mtry = 21	0.9989125	11985.155
mlp	size = 5	0.9829963	796.778

Figure 8 shows the training time of the best tuned multi-class classifiers. Also here, random forest and

SVM classifiers took much more time than other classifiers. Overall the training time of each classifier in multi-class classification is almost the same as binary classification time.

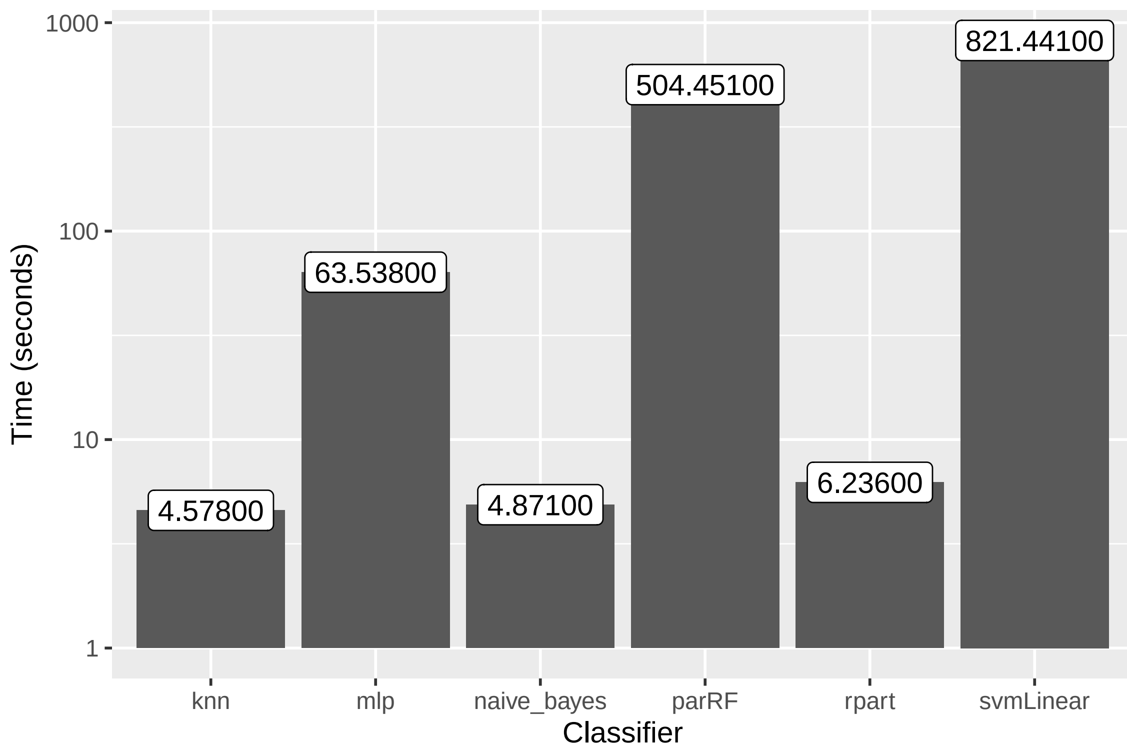

Figure 8: Training Time of Best Tuned Multi-class Classifiers

Figures 9 and 10 show the top five important features for `rpart` and `parRF` respectively. In this case, the order of important features is a bit different between `rpart` and `parRF`.

5.2 Multi-class Classification Performance

We computed the metrics mentioned in Section 3.3.2 using `confusionMatrix`. The accuracy of the multi-class classification IDS is shown in Figure 11. It is clear that `naive_bayes` had the worst performance with an accuracy of about 52.46%, while `parRF` had the best performance with an accuracy of about 76.67%. The other classifiers achieved almost the same accuracy as `parRF`. Moreover, the accuracy of each classiffier in multi-class classification is less than its value in binary classification. This gives an indication that it is usually harder to build a multi-class classification IDS.

Figure 12 shows the FAR for each classifier per attack type. In general, `rpart` has the worst FAR for all attacks. While, `naive_bayes` and `parRF` have the best FARs especially ffor DoS and Probing attacks. The FAR of R2L and U2R attacks is low (near zero) because they do not have a lot of samples.

Figure 13 shows the DR for each classifier per attack type. In general, `naive_bayes` has the worst DR for all attacks. For DoS, most of the classifiers perform almost the same with some preference for `parRF`. For Probing, `parRF` performed much better than other classifiers. For R2L and U2R, it is really difficult to achieve a good DR because of the low number of samples in the training set.

Overall, the best classifier for multi-class classification IDS is also Random Forest `parRF` as it achieved a reasonable balance between accuracy, FAR and DR across all attacks.

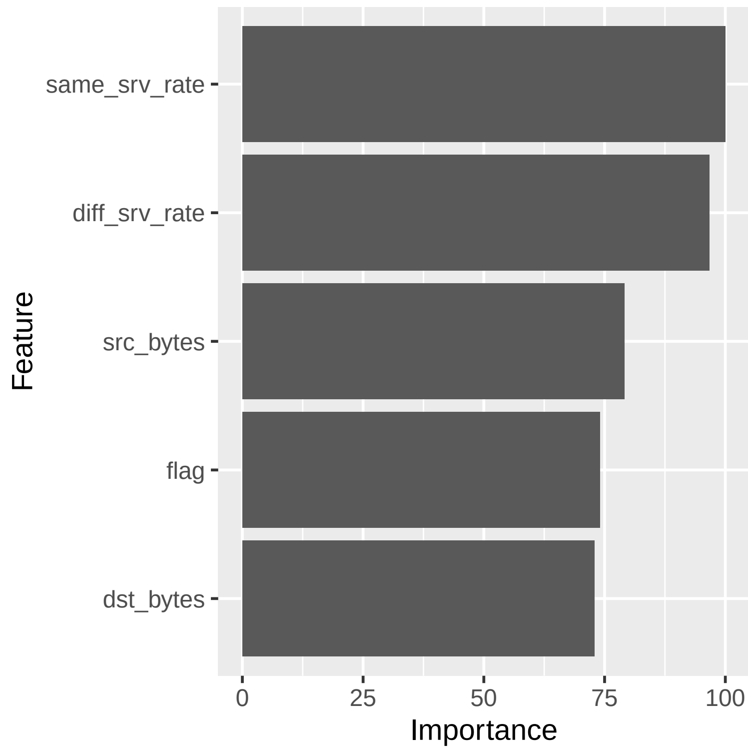

Figure 9: Feature Importance for Multi-class Classification using rpart

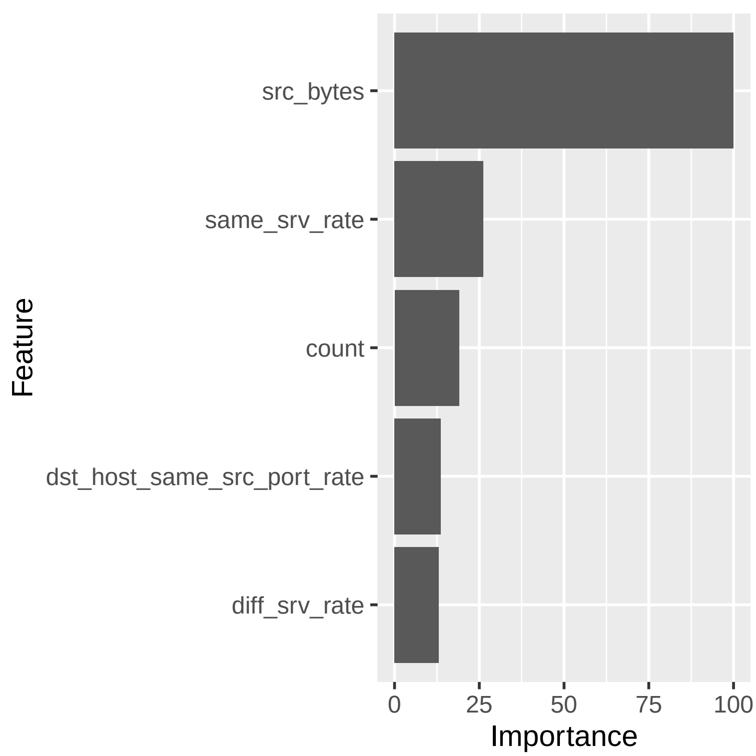

Figure 10: Feature Importance for Binary Classification using parRF

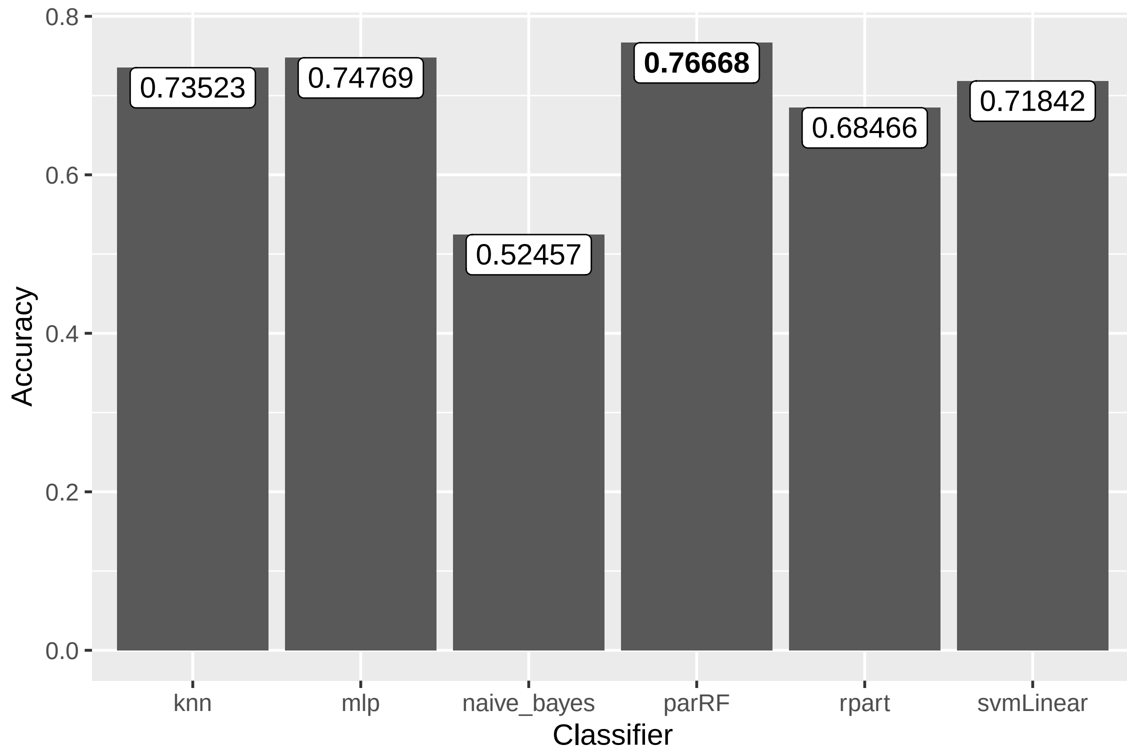

Figure 11: Accuracy of Multi-class Classfication classifiers

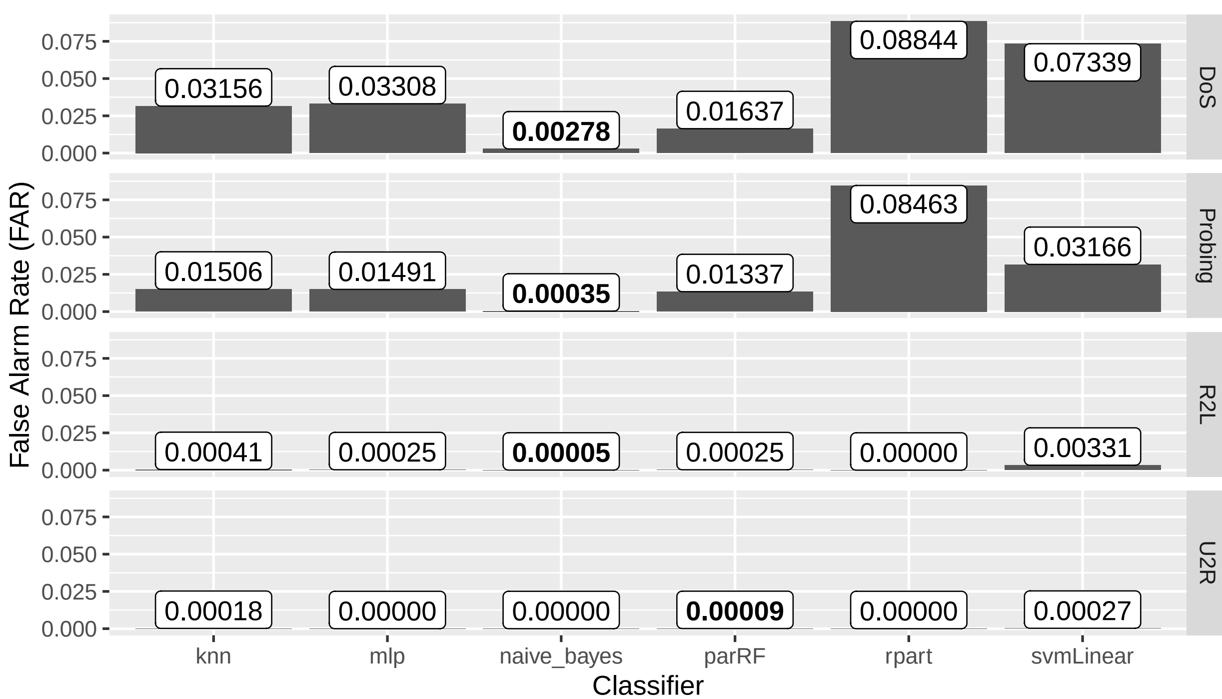

Figure 12: False Alarm Rate (FAR) of Multi-class Classfication classifiers

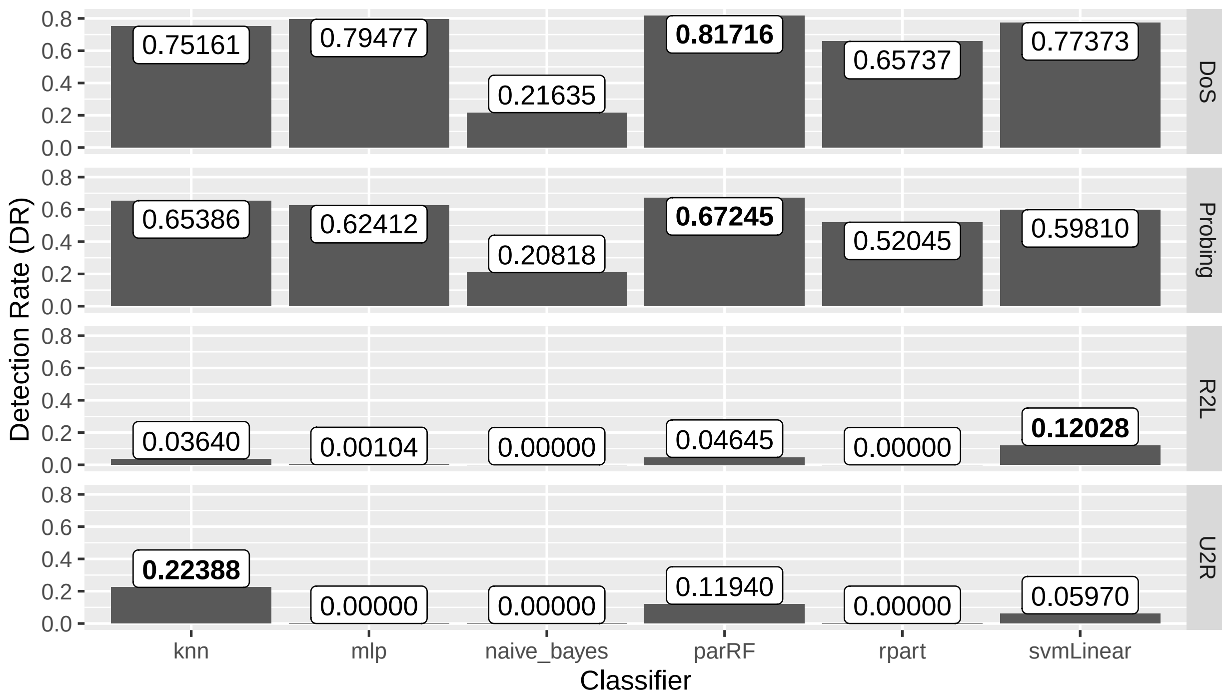

Figure 13: Detection Rate (DR) of Multi-class Classfication classifiers

6 Conclusion

Two IDSs systems were built using NSL-KDD dataset and six different classifiers. The first system was a binary classification IDS where the system predicts if a TCP connection is normal or attack. The best performing classifier for the first system was Random Forest with an accuracy of about 79.8%, a false alarm system of about 3.007%, and a detection rate of about 66.7%. The second system was a multi-class classification IDS where the system predicts if a TCP connection is one of (normal, DoS, Probing, R2L, or U2R). The best performing classifier for the second system was also Random Forest with an accuracy of about 76.67%, and a reasonable false alarm rate and detection rate for DoS and Probing attacks.

References

[1] Z.-H. Chen and C.-W. Tsai, “An effective metaheuristic algorithm for intrusion detection system,” in *2018 IEEE international conference on smart internet of things (SmartIoT)*, 2018, pp. 154–159.

[2] B. Selvakumar and K. Muneeswaran, “Firefly algorithm based feature selection for network intrusion detection,” *Computers & Security*, 2018.

[3] V. Hajisalem and S. Babaie, “A hybrid intrusion detection system based on ABC-AFS algorithm for misuse and anomaly detection,” *Computer Networks*, vol. 136, pp. 37–50, 2018.

[4] M. Tavallaee, E. Bagheri, W. Lu, and A. A. Ghorbani, “A detailed analysis of the KDD CUP 99 data set,” in *2009 IEEE symposium on computational intelligence for security and defense applications*, 2009, pp. 1–6.

[5] “KDD cup 1999.” http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html

[6] S. M. H. Bamakan, H. Wang, T. Yingjie, and Y. Shi, “An effective intrusion detection framework based on MCLP/SVM optimized by time-varying chaos particle swarm optimization,” *Neurocomputing*, vol. 199, pp. 90–102, 2016.

[7] F. Salo, M. Injadat, A. B. Nassif, A. Shami, and A. Essex, “Data mining techniques in intrusion detection systems: A systematic literature review,” *IEEE Access*, vol. 6, pp. 56046–56058, 2018.

[8] M. Kuhn and H. Wickham, *Recipes: Preprocessing and feature engineering steps for modeling*. 2021. Available: https://CRAN.R-project.org/package=recipes

[9] J. K. Seth and S. Chandra, “Intrusion detection based on key feature selection using binary GWO,” in *2016 3rd international conference on computing for sustainable global development (INDIACom)*, 2016, pp. 3735–3740.

[10] M. Kuhn, *Caret: Classification and regression training*. 2021. Available: https://CRAN.R-project.org/package=caret

[11] T. Therneau and B. Atkinson, *Rpart: Recursive partitioning and regression trees*. 2019. Available: https://CRAN.R-project.org/package=rpart

[12] M. Majka, *Naivebayes: High performance implementation of the naive bayes algorithm in r*. 2019. Available: https://CRAN.R-project.org/package=naivebayes

[13] A. Karatzoglou, A. Smola, K. Hornik, and A. Zeileis, “Kernlab – an S4 package for kernel methods in R,” *Journal of Statistical Software*, vol. 11, no. 9, pp. 1–20, 2004, Available: http://www.jstatsoft.org/v11/i09/

[14] A. Liaw and M. Wiener, “Classification and regression by randomForest,” *R News*, vol. 2, no. 3, pp. 18–22, 2002, Available: https://CRAN.R-project.org/doc/Rnews/

[15] C. Bergmeir and J. M. Benítez, “Neural networks in R using the stuttgart neural network simulator: RSNNS,” *Journal of Statistical Software*, vol. 46, no. 7, pp. 1–26, 2012, Available: https://www.jstatsoft.org/v46/i07/

[16] M. Corporation and S. Weston, *doParallel: Foreach parallel adaptor for the 'parallel' package*. 2020. Available: https://CRAN.R-project.org/package=doParallel

[17] E. Hodo, X. J. A. Bellekens, A. W. Hamilton, C. Tachtatzis, and R. C. Atkinson, “Shallow and deep networks intrusion detection system: A taxonomy and survey,” *CoRR*, vol. abs/1701.02145, 2017, Available: http://arxiv.org/abs/1701.02145

[18] T. Hamed, J. B. Ernst, and S. C. Kremer, “A survey and taxonomy of classifiers of intrusion detection systems,” in *Computer and network security essentials*, Springer, 2018, pp. 21–39.