DATA MINING AND DATA WAREHOUSING PROJECT

DATA DRIVEN INTRUSION DETECTION SYSTEM BASED ON CLASSIFICATION ALGORITHMS

SUBMITTED BY:

AAYUSH KALIA (18103004)

ABHISHEK SINGH (18103007)

AMAN KUMAR RAHI (18103015)

ARCHIT JINDAL (18103019)

AVIRAL GUPTA (18103025)

SUBMITTED TO:

DR. NONITA SHARMA

ASSISTANT PROFESSOR

CSE DEPARTMENT

NIT JALANDHAR

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT

DR. B.R. AMBEDKAR NATIONAL INSTITUTE OF TECHNOLOGY

1. Title:
Data Driven Intrusion Detection System based on Classification Algorithms

2. Abstract:

Intrusion is an act of any unauthorized activity on a network. Network intrusion can lead to stealing of valuable and information along with hurting privacy of users on the network. Due to all this, intrusion detection is an important aspect needed to provide security to internet users and organisations. To develop an effective intrusion detection system, we need a dataset which has a large amount of accurate and reliable data. In this assignment we have used NSL-KDD Dataset, which is a successor of KDD'99 Dataset. KDD'99 had a problem of redundancy, which sometimes lead our model to be biased towards a specific property. We have used NSL-KDD dataset to compare the various classification algorithms and find the one with highest accuracy. The classification algorithms compared are:

- Random Forest
- SVM
- K Neighbours

All the work has been done on Google Collab using Python as the programming language.

3. Keywords:

- Intrusion Detection System
- NSL-KDD dataset
- Anomaly
- Protocol
- Random Forest
- SVM
- K Neighbours

4. Introduction

4.1 Real Life Application of Intrusion Detection System

Network-based communication systems are now an Inseparable part of a common man's life. Computer networks are effectively being used for business data processing, education, and learning, collaboration, widespread data acquisition, and entertainment.

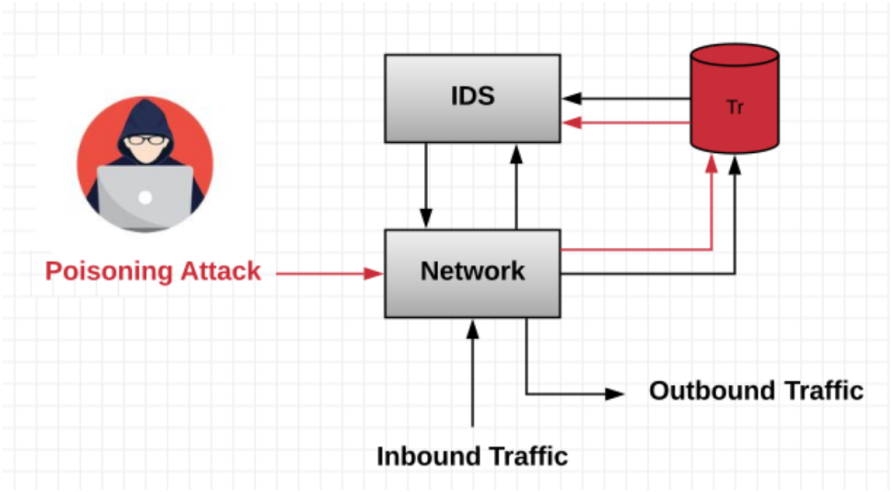
The computer network protocols were designed to be transparent and user-friendly.

This Transparency and flexibility of the protocol have made it vulnerable to the attacks launched by the intruders.

This brings out the need for computer networks to be continuously monitored and protected.

This monitoring process is automated by an **Intrusion Detection System (IDS)**.

The IDS can be made of a combination of hardware and software. At any point in time, a web server can be visited by many clients and they naturally produce heavy traffic. Each network connection can be seen as a set of attributes. The traffic data can be logged and be used to study and classify normal and abnormal traffic. In order to process the voluminous database, various machine learning techniques can be used



Data mining is the process of extracting and discovering patterns in large data sets using machine learning techniques, statistics, and database systems.

The main of an Intrusion Detection System is to raise an alert when it discovers any malicious activity, which has been classified as abnormal behaviour.

So, here in this project, we'll be building a data-driven Intrusion Detection System (IDS), and analyse it based on different classification algorithms like Random Forest, KNN, SVM.

4.2 Dataset Analysis

KDD'99 Dataset – This dataset contains a large amount of network traffic records, collected for the purpose of building a network intrusion detector.

When was KDD'99 Dataset collected?

In 1999, KDD Cup (an International Knowledge Discovery and Data Mining Tools Competition) competition was held with the goal of collecting traffic records.

In that competition, KDD'99 Dataset was collected.

NSL-KDD Dataset – It is an improvement of KDD'99 Dataset. It can be used as an effective benchmark data set to help researchers compare different intrusion detection methods.

- Redundant records are removed to enable the classifiers to produce an un-biased result.
- Sufficient number of records is available in the train and test data sets, which is reasonably rational and enables to execute experiments on the complete set.

List of NSL-KDD dataset files and their description:

S.No.	Name of the File	Description
1	KDDTrain+.ARFF	This file contains full NSL-KDD train set with binary labels in ARFF format
2	KDDTrain+.TXT	This file contains full NSL-KDD train set including attack-type labels and difficulty level in CSV format
3	KDDTrain+_20Percent.ARFF	This file contains a 20% subset of the KDDTrain+.arff file
4	KDDTrain+_20Percent.TXT	This file contains a 20% subset of the KDDTrain+.txt file
5	KDDTest+.ARFF	This file contains full NSL-KDD test set with binary labels in ARFF format
6	KDDTest+.TXT	This file contains full NSL-KDD test set including attack-type labels and difficulty level in CSV format
7	KDDTest-21. ARFF	This file contains a subset of the KDDTest+.arff file which does not include records with difficulty level of 21 out of 21
8	KDDTest-21.TXT	This file contains a subset of the KDDTest+.txt file which does not include records with difficulty level of 21 out of 21

The data set contains 43 features per record, with 41 of the features referring to the traffic input itself and the last two are labels (whether it is a normal or attack) and Score (the severity of the traffic input itself).

Quick Summary About Various Attributes:

1. Duration
2. Protocol_type
3. Service
4. Flag
5. Src_bytes
6. Dst_bytes
7. Land
8. Wrong_fragment
9. Urgent
10. Hot
11. Num_failed _logins
12. Logged_in
13. Num_compromised
14. Root_shell
15. Su_attempted
16. Num_root
17. Num_file_creations
18. Num_shells

19. Num_access_files
20. Num_outbound_cmds
21. Is_hot_login
22. Is_guest_login
23. Count
24. Srv_count
25. Serror_rate
26. Srv_serror_rate
27. Rerror_rate
28. Srv_rerror_rate
29. Same_srv_rate
30. Diff_srv_rate
31. Srv_diff_host_ rate
32. Dst_host_count
33. Dst_host_srv_count
34. Dst_host_same _srv_rate
35. Dst_host_diff_ srv_rate
36. Dst_host_same _src_port_rate
37. Dst_host_srv_ diff_host_rate
38. Dst_host_serror_rate
39. Dst_host_srv_serror_rate
40. Dst_host_rerro r_rate
41. Dst_host_srv_r error_rate
42. Label – Traffic is normal or Attack
43. Score- severity of traffic itself

The features in a traffic record provide the information about the encounter with the traffic input by the IDS and can be broken down into four categories: Intrinsic, Content, Host-based, and Time-based.

- *Intrinsic features:* They can be derived from the header of the packet without looking into the payload itself, and hold the basic information about the packet. This category contains features 1–9.
- *Content features:* They hold information about the original packets, as they are sent in multiple pieces rather than one. With this information, the system can access the payload. This category contains features 10–22.
- *Time-based features:* They hold the analysis of the traffic input over a two-second window and contain information like how many connections it attempted to make to the same host. These features are mostly counts and rates rather than information about the content of the traffic input. This category contains features 23–31.
- *Host-based features:* They are similar to Time-based features, except instead of analysing over a 2-second window, it analyses over a series of connections made (how many requests made to the same host over x-number of connections). These features are designed to access attacks, which span longer than a two-second window time span. This category contains features 32–41.

The Attack classes in NSL-KDD dataset are grouped into 4-categories:

1. **DOS**: Denial of service is an attack category, which depletes the victim's resources thereby making it unable to handle legitimate requests – e.g. syn-flooding.
2. **Probing**: Surveillance and other probing attack's objective is to gain information about the remote victim e.g., port scanning.
3. **U2R**: unauthorized access to local super user (root) privileges is an attack type, by which an attacker uses a normal account to login into a victim system and tries to gain root/administrator privileges by exploiting some vulnerability in the victim e.g., buffer overflow attacks.
4. **R2L**: unauthorized access from a remote machine, the attacker intrudes into a remote machine and gains local access of the victim machine. E.g., password guessing

Breakdown of different subclasses of each attack:

Classes:	DoS	Probe	U2R	R2L
Subclasses	• apache2 • back • land • Neptune • Mailbomb • Pod • Processtable • Smurf • Teardrop • Udpstorm • worm	• ipsweep • mscan • nmap • portsweep • saint • satan	• buffer_overflow • loadmodule • perl • ps • rootkit • sqlattack • xterm	• ftp_write • guess_passwd • imap • multihop • named • phf • sendmail • Snmpgetattack • Spy • Snmpguess • Warezclient • Warezmaster • Xlock • xsnoop
Total	11	6	7	15

Attribute Value Types:

Type	**Features**
Nominal	Protocol_type(2), Service(3), Flag(4)
Binary	Land(7), logged_in(12), root_shell(14), su_attempted(15), is_host_login(21), is_guest_login(22)
numeric	Duration(1), src_bytes(5), dst_bytes(6), wrong_fragment(8), urgent(9), hot(10), num_failed_logins(11), num_compromised(13), num_root(16), num_file_creations(17), num_shells(18), num_access_files(19), num_outbound_cmds(20), count(23) srv_count(24), serror_rate(25), srv_serror_rate(26), rerror_rate(27), srv_rerror_rate(28), same_srv_rate(29) diff_srv_rate(30), srv_diff_host_rate(31), dst_host_count(32), dst_host_srv_count(33), dst_host_same_srv_rate(34), dst_host_diff_srv_rate(35), dst_host_same_src_port_rate(36), dst_host_srv_diff_host_rate(37), dst_host_serror_rate(38), dst_host_srv_serror_rate(39), dst_host_rerror_rate(40), dst_host_srv_rerror_rate(41)

Distribution of normal and Attack Data in NSL-KDD dataset :

Data Set Type	Total No. of					
	Records	Normal Class	DoS Class	Probe Class	U2R Class	R2L Class
KDD Train+ 20%	25192	13449	9234	2289	11	209
		53.39 %	36.65 %	9.09%	0.04 %	0.83 %
KDD Train+	125973	67343	45927	11656	52	995
		53.46 %	36.46	9.25%	0.04 %	0.79 %
KDD Test+	22544	9711	7458	2421	200	2754
		43.08 %	33.08 %	10.74 %	0.89 %	12.22 %

Classification

Classification is a supervised learning method, which involves categorizing a given dataset into classes. It can be performed on both structured and unstructured data. The process starts with predicting the class of given data points.

Various Classification Methods:

1. Logistic Regression: Logistic regression is a machine learning algorithm for classification. In this algorithm, the probabilities describing the possible outcomes of a single trial are modelled using a logistic function.
   Advantages: Probabilistic approach gives information about statistical significance of features.
   Disadvantages: The assumption of linearity between dependent and independent variables

2. Naïve Bayes: Naive Bayes algorithm based on Bayes' theorem with the assumption of independence between every pair of features.

   Advantages: Efficient, Not biased by outliers, works on non-linear problems, probabilistic approach.

   Disadvantages: Based on the assumption that features have small statistical relevance.

3. K-Nearest Neighbours: Neighbours based classification is a type of lazy learning as it does not attempt to construct a general internal model, but simply stores instances of the training data. Classification is computed from a simple majority vote of the k nearest neighbours of each point.

   Advantages: Simple to understand, fast and efficient

   Disadvantages: Need to manually choose 'k'

4. Decision Tree: Given a data of attributes together with its classes, a decision tree produces a sequence of rules that can be used to classify the data.

   Advantages: Interoperability, no need of feature scaling, works on both linear/non-linear problems.

   Disadvantages: Poor results on every small dataset, overfitting

5. Random Forest: Random forest classifier is a meta-estimator that fits a number of decision trees on various sub-samples of datasets and uses average to improve the predictive accuracy of the model and controls over-fitting.

   Advantages: Less prone to overfitting, outputs the importance features.

   Disadvantages: Computations can become very complex

6. Support Vector Machine (SVM): Support vector machine is a representation of the training data as points in space separated into categories by a clear gap that is as wide as possible. New examples are then mapped into that same space and predicted to belong to a category based on which side of the gap they fall.
   Advantages: Not biased to outliers, not sensitive to overfitting
   Disadvantages: Not a good choice when large number of features

4.3 Structure of project Report

The following table shows the structure of project report from 5th section onwards:

Section	Name	Detail
5	Classification Methods	Discussion about various classification algorithms
6	Experimentation	Complete demonstration of the Project
7	Results and Discussion	Comparison between intrusion detection models built on KNN, SVM and Random Forest.
8	Conclusion and Future Scope	Final project summary and future scope of improvement.

5. Methods of Classification

a) Decision Tree

A DT is a predictive model that is represented as a recursive partition of the feature space into subspaces that form the prediction base. A DT stands for a directed tree with roots.

Internal nodes in DTs are those with outgoing edges. The DT's other nodes are terminal nodes or leaves.

The attributes are classified by DTs using a series of hierarchical decisions. The split criteria is determined by judgments taken at internal nodes.

Each leaf in a DT is allocated to a class or chance.

Small changes in the training set cause various splits, which result in a different DT. As a result, for DTs, the error contribution due to variance is important. Ensemble learning, discussed in the next section, can help palliate the error due to variance.

A general algorithm for a decision tree can be described as follows:

1. Pick the best attribute/feature. The best attribute is one which best splits or separates the data.
2. Ask the relevant question.
3. Follow the answer path.
4. Go to step 1 until you arrive to the answer.

The best split is one which separates two different labels into two sets.

How does a tree decide where to split?

The dataset can be splitted using entropy and information gain.

1. **Information Gain**

Entropy is a measure of how much difference there is in a set of data.

The weighted entropies of each branch are subtracted from the original entropy to quantify Information Gain for a break. By using these metrics to train a Decision Tree, the best split is determined by optimising Information Gain.

$$Entropy = \sum_{i=1}^{C} -p_i * \log_2(p_i)$$

$$Gain(S, A) = Entropy(S) - \sum_{vcValues(A)} \frac{|S_v|}{|S|}.Entropy(S_v)$$

ii. Gini Index

- Gini Index is a metric to measure how often a randomly chosen element would be incorrectly identified.
- It means an attribute with lower Gini index should be preferred.
- Sklearn supports "Gini" criteria for Gini Index and by default, it takes "gini" value.
- The Formula for the calculation of the of the Gini Index is given below.

$$GiniIndex = 1 - \sum_j p_j^2$$

Advantages

- It is easy to grasp because it follows a constant method that somebody follows whereas creating any call-in real-life.
- It is terribly helpful for the resolution of decision-related issues.
- It helps to place confidence in all the attainable outcomes for a haul.
- There is less demand for knowledge cleansing compared to alternative algorithms.

Disadvantages

- The decision tree contains legion layers, which makes it advanced.
- It may have an associate overfitting issue, which might be resolved exploitation the Random Forest formula.
- For a lot of category labels, the process quality of the choice tree could increase.

b) Naïve Bayes algorithm

Naïve Bayes algorithm is a classification technique based on applying Bayes' theorem with a strong assumption that all the predictors are independent to each other. To put it another way, the presumption is that the inclusion of a function in a class is unrelated to the existence of other features in the same class. A phone, for example, can be called smart if it has a touch screen, internet access, and a decent camera. Despite the fact that all of these functions are interdependent, they each add to the likelihood that the phone is a smart phone.

The key goal of Bayesian classification is to determine the posterior probabilities, or the likelihood of a mark provided certain observed characteristics, P(L | features). We can express this in quantitative form using Bayes' theorem as follows:

wP(L|features)=P(L)P(features|L) / P(features)

Here, ($L \mid features$) is the posterior probability of class.

$P(L)$ is the prior probability of class.

$P(features|L)$ is the likelihood which is the probability of predictor given class.

$P(features)$ is the prior probability of predictor.

Applications of Naïve Bayes classification

The following are some common applications of Naïve Bayes classification −

- **Real-time prediction** − Due to its ease of implementation and fast computation, it can be used to do prediction in real-time.
- **Multi-class prediction** − Naïve Bayes classification algorithm can be used to predict posterior probability of multiple classes of target variable.
- **Text classification** − Due to the feature of multi-class prediction, Naïve Bayes classification algorithms are well suited for text classification. That is why it is also used to solve problems like spam-filtering and sentiment analysis.
- **Recommendation system** − Along with the algorithms like collaborative filtering, Naïve Bayes makes a Recommendation system which can be used to filter unseen information and to predict weather a user would like the given resource or not.

c) K Nearest Neighbours Algorithm

The KNN algorithm is a type of supervised machine learning algorithm that can be used to solve both classification and regression predictive problems. However, in industry, it is mostly used to solve classification and prediction problems. The following two characteristics would be a good way to describe KNN:

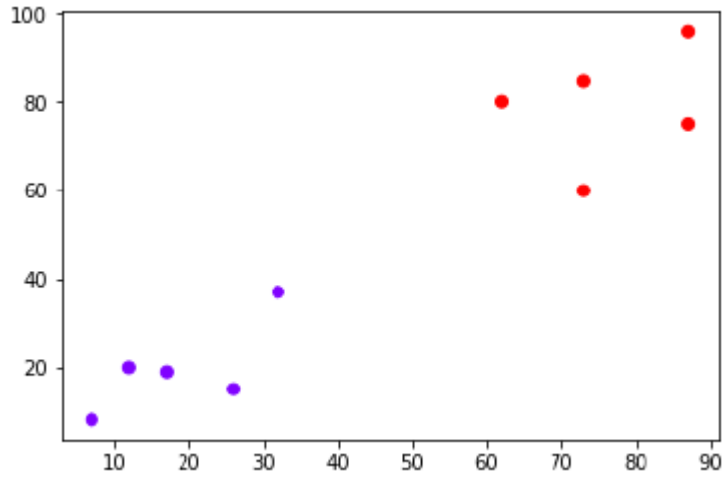
KNN is a lazy learning algorithm since it doesn't have a dedicated training process and instead uses all of the data for training and classification.

KNN is also a non-parametric learning algorithm since it makes no assumptions about the underlying results.

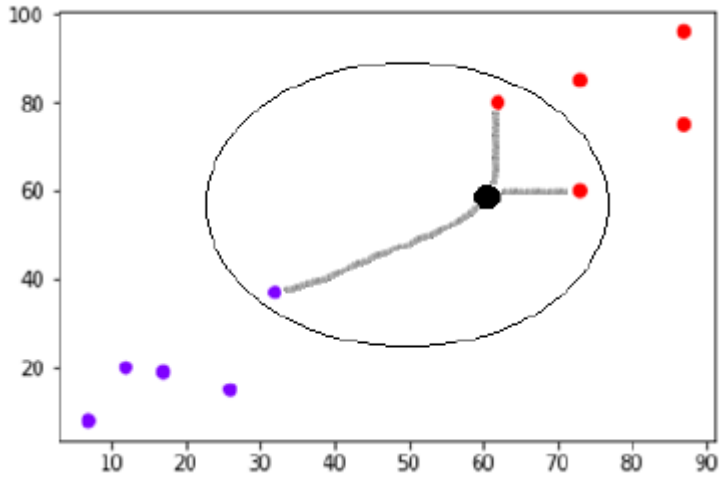
Working of KNN

The following is an example to understand the concept of K and working of KNN algorithm –

Let's say we have a dataset that can be plotted like this:



Now we must sort the new data point with the black dot (at 60,60) into the blue or red categories. We are assuming K = 3 i.e. it would find three nearest data points. It is shown in the next diagram –



We can see in the above diagram the three nearest neighbors of the data point with black dot. Among those three, two of them lies in Red class hence the black dot will also be assigned in red class.

d) Random Forest Algorithm

"Random Forest is a classifier that combines a variety of decision trees on different subsets of a dataset and averages them to increase the dataset's predictive accuracy." Instead of depending on a single decision tree, the random forest takes the predictions from each tree and forecasts the final performance based on the majority votes of predictions.

Working of Random Forest Algorithm:

The following steps will help us understand how the Random Forest algorithm works.

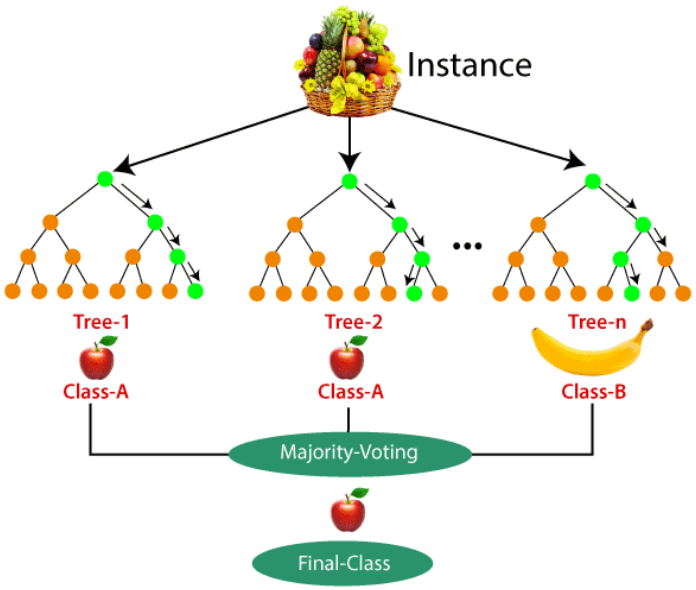
Step 1: Begin by selecting random samples from a dataset.

Step 2: For each sample, this algorithm will build a decision tree. The prediction outcome from each decision tree would then be obtained.

Step 3: Voting will be done for each expected outcome in this step.

Step 4: Then, choose the prediction outcome with the most votes as the actual prediction result.

Consider the following scenario: there is a dataset containing several fruit images. As a result, the Random forest classifier is given this dataset. Each decision tree is assigned a subset of the dataset to work with. During the training process, each decision tree generates a prediction result, and when a new data point appears, the Random Forest classifier forecasts the final decision based on the majority of outcomes. Get the following illustration:



6. Experimentation:

6.1 Random Forest

Random forests, also known as random decision forests, are an ensemble learning method for classification, regression, and other tasks that works by constructing a large number of decision trees during training and then outputting the class that is the mean/average prediction of the individual trees.

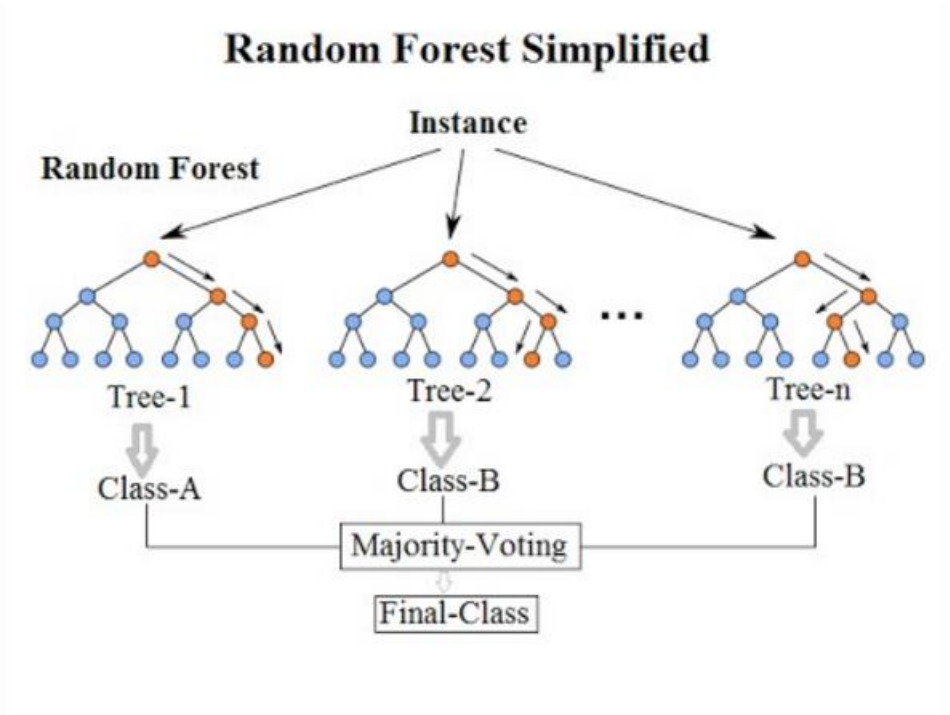
The Random Forest Classifier

Random forest consists of many individual decision trees that operate as an ensemble. Every individual tree in the random forest spits out a class prediction and the class with the maximum votes becomes our model's prediction.

The prerequisites for random forest are:

Our features must have some real signal in order for models designed with them to outperform random guessing.

Individual tree predictions must have low correlations with one another.



Pros:

- Robust against outliers.
- Works well with non-linear data and reduces the chance of over fitting.
- On a huge dataset, it performs well.
- Better than other classification algorithms in terms of accuracy.

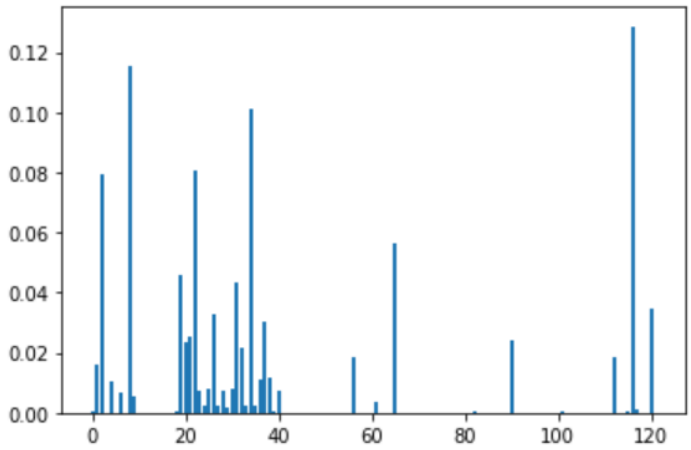
Cons:

- When dealing with categorical variables, random forests are considered to be biased.
- It is not suitable for linear methods with a lot of sparse features

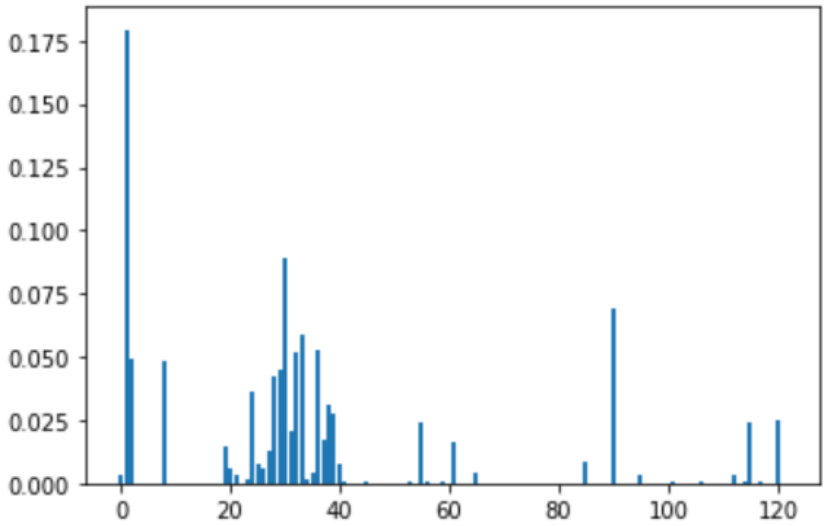
```
from sklearn.ensemble import RandomForestClassifier

# Build the model
clf_DOS = RandomForestClassifier(n_estimators=10, n_jobs=2)
clf_Probe = RandomForestClassifier(n_estimators=10, n_jobs=2)
clf_R2L = RandomForestClassifier(n_estimators=10, n_jobs=2)
clf_U2R = RandomForestClassifier(n_estimators=10, n_jobs=2)

clf_DOS.fit(X_DOS_train, Y_DOS_train.astype(int))
clf_Probe.fit(X_Probe_train, Y_Probe_train.astype(int))
clf_R2L.fit(X_R2L_train, Y_R2L_train.astype(int))
clf_U2R.fit(X_U2R_train, Y_U2R_train.astype(int))
```

```
RandomForestClassifier(bootstrap=True, ccp_alpha=0.0, class_weight=None,
                       criterion='gini', max_depth=None, max_features='auto',
                       max_leaf_nodes=None, max_samples=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, n_estimators=10, n_jobs=2,
                       oob_score=False, random_state=None, verbose=0,
                       warm_start=False)
```

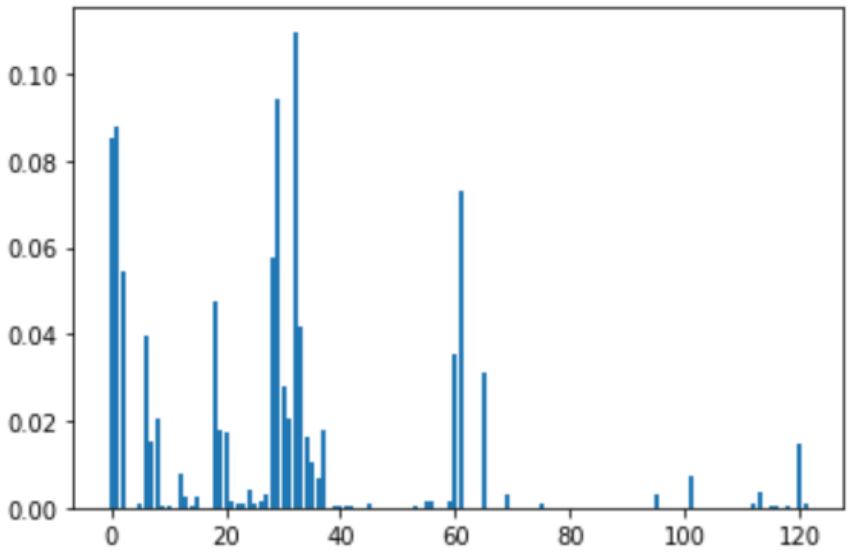
```
importance_DOS = clf_DOS.feature_importances_
pyplot.bar([x for x in range(len(importance_DOS))], importance_DOS)
pyplot.show()
```



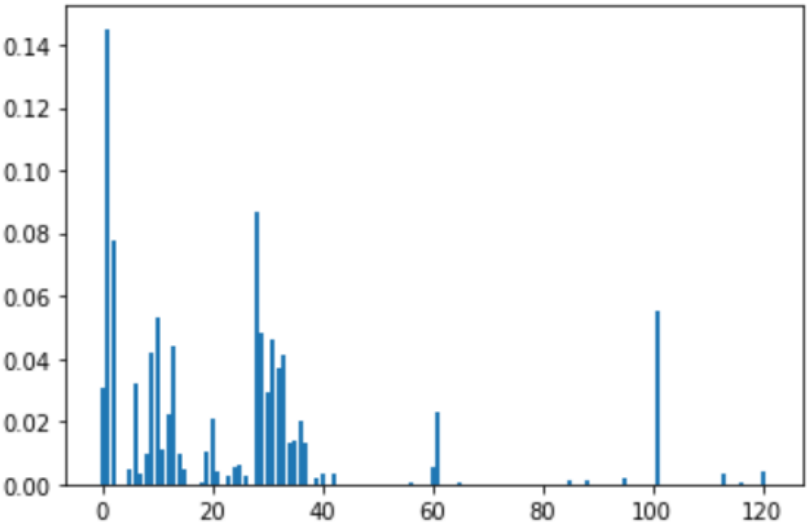

```
importance_Probe = clf_Probe.feature_importances_
pyplot.bar([x for x in range(len(importance_Probe))], importance_Probe)
pyplot.show()
```



```
importance_R2L = clf_R2L.feature_importances_
pyplot.bar([x for x in range(len(importance_R2L))], importance_R2L)
pyplot.show()
```




```
importance_U2R = clf_U2R.feature_importances_
pyplot.bar([x for x in range(len(importance_U2R))], importance_U2R)
pyplot.show()
```



Prediction and Evaluation

```
# DOS
# Apply the classifier to the test data
Y_DOS_pred = clf_DOS.predict(X_DOS_test)
pd.crosstab(Y_DOS_test, Y_DOS_pred, rownames=['Actual Attacks'], colnames=['Predicted Attacks'])
```

Predicted Attacks	0	1
Actual Attacks		
0	9682	29
1	6035	1425

```
# Probe
Y_Probe_pred = clf_Probe.predict(X_Probe_test)
pd.crosstab(Y_Probe_test, Y_Probe_pred, rownames=['Actual Attacks'], colnames=['Predicted Attacks'])
```

Predicted Attacks	0	2
Actual Attacks		
0	9453	258
2	978	1443

```
# R2L
Y_R2L_pred = clf_R2L.predict(X_R2L_test)
pd.crosstab(Y_R2L_test, Y_R2L_pred, rownames=['Actual Attacks'], colnames=['Predicted Attacks'])
```

Predicted Attacks	0
Actual Attacks	
0	9711
3	2885

```
# U2R
Y_U2R_pred = clf_U2R.predict(X_U2R_test)
pd.crosstab(Y_U2R_test, Y_U2R_pred, rownames=['Actual Attacks'], colnames=['Predicted Attacks'])
```

Predicted Attacks	0	4
Actual Attacks		
0	9711	0
4	65	2

Cross Validation: Accuracy, Precision, Recall, F-measure

```
# DOS
from sklearn.model_selection import cross_val_score
from sklearn import metrics
accuracy = cross_val_score(clf_DOS, X_DOS_test, Y_DOS_test, cv=10, scoring='accuracy')
print("Accuracy: %0.5f (+/- %0.5f)" % (accuracy.mean(), accuracy.std() * 2))
precision = cross_val_score(clf_DOS, X_DOS_test, Y_DOS_test, cv=10, scoring='precision')
print("Precision: %0.5f (+/- %0.5f)" % (precision.mean(), precision.std() * 2))
recall = cross_val_score(clf_DOS, X_DOS_test, Y_DOS_test, cv=10, scoring='recall')
print("Recall: %0.5f (+/- %0.5f)" % (recall.mean(), recall.std() * 2))
f = cross_val_score(clf_DOS, X_DOS_test, Y_DOS_test, cv=10, scoring='f1')
print("F-measure: %0.5f (+/- %0.5f)" % (f.mean(), f.std() * 2))
```

```
Accuracy: 0.99796 (+/- 0.00197)
Precision: 0.99906 (+/- 0.00172)
Recall: 0.99678 (+/- 0.00401)
F-measure: 0.99772 (+/- 0.00257)
```

```
Accuracy_RF_DOS = accuracy.mean()
```

```
# Probe
accuracy = cross_val_score(clf_Probe, X_Probe_test, Y_Probe_test, cv=10, scoring='accuracy')
print("Accuracy: %0.5f (+/- %0.5f)" % (accuracy.mean(), accuracy.std() * 2))
precision = cross_val_score(clf_Probe, X_Probe_test, Y_Probe_test, cv=10, scoring='precision_macro')
print("Precision: %0.5f (+/- %0.5f)" % (precision.mean(), precision.std() * 2))
recall = cross_val_score(clf_Probe, X_Probe_test, Y_Probe_test, cv=10, scoring='recall_macro')
print("Recall: %0.5f (+/- %0.5f)" % (recall.mean(), recall.std() * 2))
f = cross_val_score(clf_Probe, X_Probe_test, Y_Probe_test, cv=10, scoring='f1_macro')
print("F-measure: %0.5f (+/- %0.5f)" % (f.mean(), f.std() * 2))
```

```
Accuracy: 0.99662 (+/- 0.00238)
Precision: 0.99603 (+/- 0.00470)
Recall: 0.99370 (+/- 0.00516)
F-measure: 0.99508 (+/- 0.00491)
```

```
Accuracy_RF_Probe = accuracy.mean()
```

```
# U2R
accuracy = cross_val_score(clf_U2R, X_U2R_test, Y_U2R_test, cv=10, scoring='accuracy')
print("Accuracy: %0.5f (+/- %0.5f)" % (accuracy.mean(), accuracy.std() * 2))
precision = cross_val_score(clf_U2R, X_U2R_test, Y_U2R_test, cv=10, scoring='precision_macro')
print("Precision: %0.5f (+/- %0.5f)" % (precision.mean(), precision.std() * 2))
recall = cross_val_score(clf_U2R, X_U2R_test, Y_U2R_test, cv=10, scoring='recall_macro')
print("Recall: %0.5f (+/- %0.5f)" % (recall.mean(), recall.std() * 2))
f = cross_val_score(clf_U2R, X_U2R_test, Y_U2R_test, cv=10, scoring='f1_macro')
print("F-measure: %0.5f (+/- %0.5f)" % (f.mean(), f.std() * 2))
```

```
Accuracy: 0.99693 (+/- 0.00224)
Precision: 0.96088 (+/- 0.10669)
Recall: 0.84985 (+/- 0.13520)
F-measure: 0.92658 (+/- 0.11544)
```

```
[ ]  Accuracy_RF_U2R = accuracy.mean()
```

```
[ ]  # R2L
     accuracy = cross_val_score(clf_R2L, X_R2L_test, Y_R2L_test, cv=10, scoring='accuracy')
     print("Accuracy: %0.5f (+/- %0.5f)" % (accuracy.mean(), accuracy.std() * 2))
     precision = cross_val_score(clf_R2L, X_R2L_test, Y_R2L_test, cv=10, scoring='precision_macro')
     print("Precision: %0.5f (+/- %0.5f)" % (precision.mean(), precision.std() * 2))
     recall = cross_val_score(clf_R2L, X_R2L_test, Y_R2L_test, cv=10, scoring='recall_macro')
     print("Recall: %0.5f (+/- %0.5f)" % (recall.mean(), recall.std() * 2))
     f = cross_val_score(clf_R2L, X_R2L_test, Y_R2L_test, cv=10, scoring='f1_macro')
     print("F-measure: %0.5f (+/- %0.5f)" % (f.mean(), f.std() * 2))

     Accuracy: 0.98015 (+/- 0.00598)
     Precision: 0.97534 (+/- 0.00980)
     Recall: 0.96846 (+/- 0.01388)
     F-measure: 0.97082 (+/- 0.00915)
```

```
     Accuracy_RF_R2L = accuracy.mean()
```

6.2 KNeighbours

The K-Nearest Neighbour algorithm is based on the Supervised Learning technique and is one of the most basic Machine Learning algorithms. The K-NN algorithm assumes that the new case/data and existing cases are identical and places the new case in the category that is most similar to the existing categories.

Choosing the correct value for K

We run the KNN algorithm several times with different values of K to find the K that decreases the number of errors we encounter while preserving the algorithm's ability to correctly make predictions when given data it hasn't seen before.

How does KNN Algorithm works?

The K-nearest neighbour algorithm basically boils down to forming a majority vote between the K most similar instances to a given "unseen" observation in the classification environment. A distance metric between two data points is used to establish similarity. The Euclidean distance method is a common one.

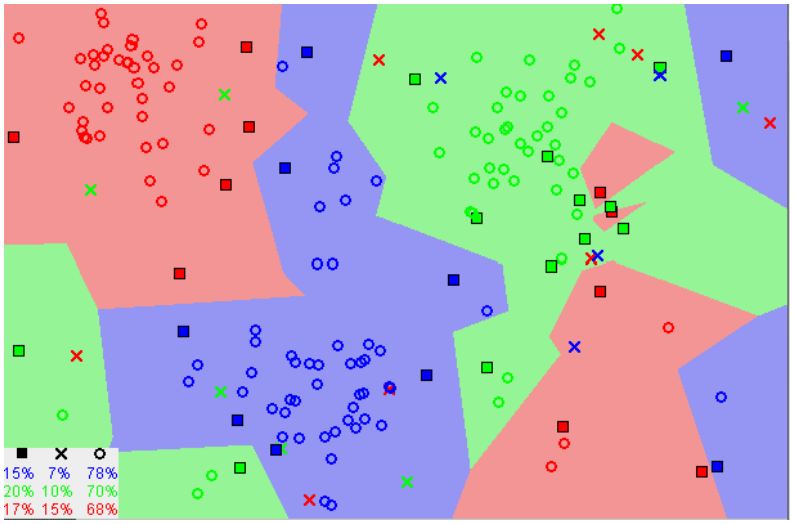
Manhattan, Minkowski, and Hamming distance methods are examples of other distance methods. The hamming distance must be used for categorical variables.

Pros of KNN

- Simple to incorporate Flexible in terms of feature/distance choices
- Handles multi-class cases naturally
- With enough representative data, it is possible to perform well in practice.

Cons of KNN

- The cost of computing the distance between each query instance and all training samples is very high since we need to calculate the value of parameter K (number of nearest neighbours).
- Data storage
- It's important to know that we have a useful distance feature.




```
from sklearn.neighbors import KNeighborsClassifier

clf_KNN_DOS = KNeighborsClassifier()
clf_KNN_Probe = KNeighborsClassifier()
clf_KNN_R2L = KNeighborsClassifier()
clf_KNN_U2R = KNeighborsClassifier()

clf_KNN_DOS.fit(X_DOS_train, Y_DOS_train.astype(int))
clf_KNN_Probe.fit(X_Probe_train, Y_Probe_train.astype(int))
clf_KNN_R2L.fit(X_R2L_train, Y_R2L_train.astype(int))
clf_KNN_U2R.fit(X_U2R_train, Y_U2R_train.astype(int))
```

```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                     metric_params=None, n_jobs=None, n_neighbors=5, p=2,
                     weights='uniform')
```

```
# DOS
Y_DOS_pred_KNN = clf_KNN_DOS.predict(X_DOS_test)
pd.crosstab(Y_DOS_test, Y_DOS_pred_KNN, rownames=['Actual attacks'], colnames=['Predicted Attacks'])
```

Predicted Attacks	0	1
Actual attacks		
0	9422	289
1	1573	5887

```
# Probe
Y_Probe_pred_KNN = clf_KNN_Probe.predict(X_Probe_test)
pd.crosstab(Y_Probe_test, Y_Probe_pred_KNN, rownames=['Actual Attacks'], colnames=['Predicted Attacks'])
```

```
# R2L
Y_R2L_pred_KNN = clf_KNN_R2L.predict(X_R2L_test)
pd.crosstab(Y_R2L_test, Y_R2L_pred_KNN, rownames=['Actual Attacks'], colnames=['Predicted Attacks'])
```

```
# U2R
Y_U2R_pred_KNN = clf_KNN_U2R.predict(X_U2R_test)
pd.crosstab(Y_U2R_test, Y_U2R_pred_KNN, rownames=['Actual Attacks'], colnames=['Predicted Attacks'])
```

Cross Validation: Accuracy, Precision, Recall, F-measure

```
# DOS
from sklearn.model_selection import cross_val_score
from sklearn import metrics
accuracy = cross_val_score(clf_KNN_DOS, X_DOS_test, Y_DOS_test, cv=10, scoring='accuracy')
print("Accuracy: %0.5f (+/- %0.5f)" % (accuracy.mean(), accuracy.std() * 2))
precision = cross_val_score(clf_KNN_DOS, X_DOS_test, Y_DOS_test, cv=10, scoring='precision')
print("Precision: %0.5f (+/- %0.5f)" % (precision.mean(), precision.std() * 2))
recall = cross_val_score(clf_KNN_DOS, X_DOS_test, Y_DOS_test, cv=10, scoring='recall')
print("Recall: %0.5f (+/- %0.5f)" % (recall.mean(), recall.std() * 2))
f = cross_val_score(clf_KNN_DOS, X_DOS_test, Y_DOS_test, cv=10, scoring='f1')
print("F-measure: %0.5f (+/- %0.5f)" % (f.mean(), f.std() * 2))
```

```
Accuracy: 0.99715 (+/- 0.00278)
Precision: 0.99678 (+/- 0.00383)
Recall: 0.99665 (+/- 0.00344)
F-measure: 0.99672 (+/- 0.00320)
```

```
Accuracy_KNN_DOS = accuracy.mean()
```

```
# Probe
accuracy = cross_val_score(clf_KNN_Probe, X_Probe_test, Y_Probe_test, cv=10, scoring='accuracy')
print("Accuracy: %0.5f (+/- %0.5f)" % (accuracy.mean(), accuracy.std() * 2))
precision = cross_val_score(clf_KNN_Probe, X_Probe_test, Y_Probe_test, cv=10, scoring='precision_macro')
print("Precision: %0.5f (+/- %0.5f)" % (precision.mean(), precision.std() * 2))
recall = cross_val_score(clf_KNN_Probe, X_Probe_test, Y_Probe_test, cv=10, scoring='recall_macro')
print("Recall: %0.5f (+/- %0.5f)" % (recall.mean(), recall.std() * 2))
f = cross_val_score(clf_KNN_Probe, X_Probe_test, Y_Probe_test, cv=10, scoring='f1_macro')
print("F-measure: %0.5f (+/- %0.5f)" % (f.mean(), f.std() * 2))
```

```
Accuracy: 0.99077 (+/- 0.00403)
Precision: 0.98606 (+/- 0.00675)
Recall: 0.98508 (+/- 0.01137)
F-measure: 0.98553 (+/- 0.00645)
```

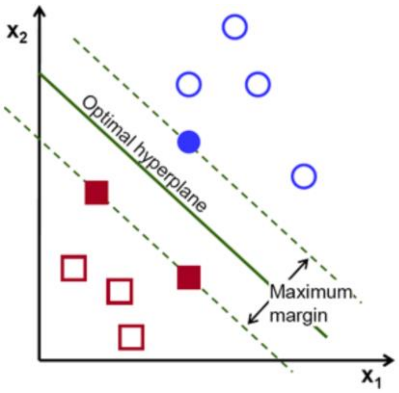
```
Accuracy_KNN_Probe = accuracy.mean()
```

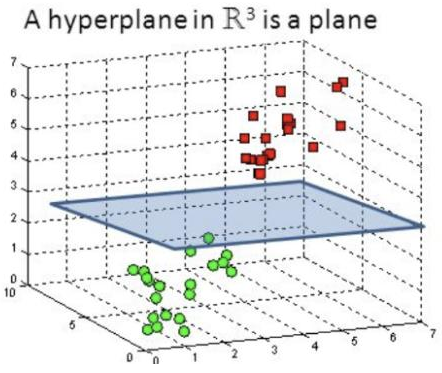
```
# R2L
accuracy = cross_val_score(clf_KNN_R2L, X_R2L_test, Y_R2L_test, cv=10, scoring='accuracy')
print("Accuracy: %0.5f (+/- %0.5f)" % (accuracy.mean(), accuracy.std() * 2))
precision = cross_val_score(clf_KNN_R2L, X_R2L_test, Y_R2L_test, cv=10, scoring='precision_macro')
print("Precision: %0.5f (+/- %0.5f)" % (precision.mean(), precision.std() * 2))
recall = cross_val_score(clf_KNN_R2L, X_R2L_test, Y_R2L_test, cv=10, scoring='recall_macro')
print("Recall: %0.5f (+/- %0.5f)" % (recall.mean(), recall.std() * 2))
f = cross_val_score(clf_KNN_R2L, X_R2L_test, Y_R2L_test, cv=10, scoring='f1_macro')
print("F-measure: %0.5f (+/- %0.5f)" % (f.mean(), f.std() * 2))
```

```
Accuracy: 0.96737 (+/- 0.00730)
Precision: 0.95311 (+/- 0.01274)
Recall: 0.95484 (+/- 0.01326)
F-measure: 0.95389 (+/- 0.01030)
```

6.3 Support Vector Machine (SVM)

The objective of SVM is to find a hyperplane in N-dimensions that can distinctly classify the data points with great accuracy. For the separation of the data points, we need to draw some boundaries and those boundaries are also know as hyperplanes. So basically, the main motive is to find such kind of plain that has the maximum margin. Below are the visualizations of the Hyperplanes in 2D and 3D.





(2D hyperplane) (3D hyperplane)

Now applying the SVM to our NSL-KDD training dataset-

For the implementation's sake, we have used the sklearn library in order to perform the computations most optimally-

Code from sklearn official website describing all the possible parameters of SVC-

class sklearn.svm.**SVC**(*, C=1.0, kernel='rbf', degree=3, gamma='scale', coef0=0.0, shrinking=True, probability=False, tol=0.001, cache_size=200, class_weight=None, verbose=False, max_iter=- 1, decision_function_shape='ovr', break_ties=False, random_state=None)

Now, next is step by step explanation regarding the implementation of SVC and fitting to the training-data.

Step by Step Explanation of implementation using SVC (Sklearn) -

1)

```
[35] from sklearn.svm import SVC

     clf_SVM_DOS=SVC(kernel='linear', C=1.0, random_state=0)
     clf_SVM_Probe=SVC(kernel='linear', C=1.0, random_state=0)
     clf_SVM_R2L=SVC(kernel='linear', C=1.0, random_state=0)
     clf_SVM_U2R=SVC(kernel='linear', C=1.0, random_state=0)
```

So, firstly SVC is imported from sklearn.svm, then for the different type of attacks i.e. DOS, Probe, R2L, U2R, we have created four separate instances of the SVC.

Kernel- A kernel transforms an input data space into the required form. Kernel trick is the trick that is mainly used by SVM. Here, the kernel takes a low-dimensional input space and transforms it into a higher dimensional space. In other words, you can say that it converts non separable problems to separable problems by adding more dimension to it. I Useful in the cases of non-linear separation problem. With the help of Kernel trick, a more accurate classifier can be built. **So here we are using the 'linear' kernel.**

Linear Kernel- It may be used as a normal dot product any two given observations.

2)

```
clf_SVM_DOS.fit(X_DOS_train, Y_DOS_train.astype(int))
clf_SVM_Probe.fit(X_Probe_train, Y_Probe_train.astype(int))
clf_SVM_R2L.fit(X_R2L_train, Y_R2L_train.astype(int))
clf_SVM_U2R.fit(X_U2R_train, Y_U2R_train.astype(int))
```

```
SVC(C=1.0, break_ties=False, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma='scale', kernel='linear',
    max_iter=-1, probability=False, random_state=0, shrinking=True, tol=0.001,
    verbose=False)
```

This is basically the fitting step, we give our training and testing sets to SVM for all the four different possible attacks and get our four models ready.

3) Now the time for making predictions using our four models-

```
[36] # DOS
     Y_DOS_pred_SVM = clf_SVM_DOS.predict(X_DOS_test)
     pd.crosstab(Y_DOS_test, Y_DOS_pred_SVM, rownames=['Actual Attacks'], colnames=['Predicted Attacks'])
```

Predicted Attacks	0	1
Actual Attacks		
0	9455	256
1	1359	6101

```
[37] # Probe
     Y_Probe_pred_SVM = clf_SVM_Probe.predict(X_Probe_test)
     pd.crosstab(Y_Probe_test, Y_Probe_pred_SVM, rownames=['Actual Attacks'], colnames=['Predicted Attacks'])
```

Predicted Attacks	0	2
Actual Attacks		
0	9576	135
2	1285	1136

```
[38] # R2L
     Y_R2L_pred_SVM = clf_SVM_R2L.predict(X_R2L_test)
     pd.crosstab(Y_R2L_test, Y_R2L_pred_SVM, rownames=['Actual Attacks'], colnames=['Predicted Attacks'])
```

Predicted Attacks	0	3
Actual Attacks		
0	9639	72
3	2737	148

```
[39] # U2R
     Y_U2R_pred_SVM = clf_SVM_U2R.predict(X_U2R_test)
     pd.crosstab(Y_U2R_test, Y_U2R_pred_SVM, rownames=['Actual Attacks'], colnames=['Predicted Attacks'])
```

Predicted Attacks	0	4
Actual Attacks		
0	9710	1
4	67	0

pd.crosstab( ) -> This basically helps in the generation of a confusion matrix so as to measure the different parameters for our classifier.

Now, Cross Validation: Accuracy, Precision, Recall, F-measure calculations-

```
[40] # DOS
     from sklearn.model_selection import cross_val_score
     from sklearn import metrics
     accuracy = cross_val_score(clf_SVM_DOS, X_DOS_test, Y_DOS_test, cv=10, scoring='accuracy')
     print("Accuracy: %0.5f (+/- %0.5f)" % (accuracy.mean(), accuracy.std() * 2))
     precision = cross_val_score(clf_SVM_DOS, X_DOS_test, Y_DOS_test, cv=10, scoring='precision')
     print("Precision: %0.5f (+/- %0.5f)" % (precision.mean(), precision.std() * 2))
     recall = cross_val_score(clf_SVM_DOS, X_DOS_test, Y_DOS_test, cv=10, scoring='recall')
     print("Recall: %0.5f (+/- %0.5f)" % (recall.mean(), recall.std() * 2))
     f = cross_val_score(clf_SVM_DOS, X_DOS_test, Y_DOS_test, cv=10, scoring='f1')
     print("F-measure: %0.5f (+/- %0.5f)" % (f.mean(), f.std() * 2))
```

```
Accuracy: 0.99371 (+/- 0.00375)
Precision: 0.99107 (+/- 0.00785)
Recall: 0.99450 (+/- 0.00388)
F-measure: 0.99278 (+/- 0.00428)
```

```
[41] Accuracy_SVM_DOS = accuracy.mean()
```

```
[42] # Probe
     accuracy = cross_val_score(clf_SVM_Probe, X_Probe_test, Y_Probe_test, cv=10, scoring='accuracy')
     print("Accuracy: %0.5f (+/- %0.5f)" % (accuracy.mean(), accuracy.std() * 2))
     precision = cross_val_score(clf_SVM_Probe, X_Probe_test, Y_Probe_test, cv=10, scoring='precision_macro')
     print("Precision: %0.5f (+/- %0.5f)" % (precision.mean(), precision.std() * 2))
     recall = cross_val_score(clf_SVM_Probe, X_Probe_test, Y_Probe_test, cv=10, scoring='recall_macro')
     print("Recall: %0.5f (+/- %0.5f)" % (recall.mean(), recall.std() * 2))
     f = cross_val_score(clf_SVM_Probe, X_Probe_test, Y_Probe_test, cv=10, scoring='f1_macro')
     print("F-measure: %0.5f (+/- %0.5f)" % (f.mean(), f.std() * 2))
```

```
Accuracy: 0.98450 (+/- 0.00526)
Precision: 0.96907 (+/- 0.01031)
Recall: 0.98365 (+/- 0.00686)
F-measure: 0.97613 (+/- 0.00800)
```

```
[43] Accuracy_SVM_Probe = accuracy.mean()
```

```
[44] # R2L
     accuracy = cross_val_score(clf_SVM_R2L, X_R2L_test, Y_R2L_test, cv=10, scoring='accuracy')
     print("Accuracy: %0.5f (+/- %0.5f)" % (accuracy.mean(), accuracy.std() * 2))
     precision = cross_val_score(clf_SVM_R2L, X_R2L_test, Y_R2L_test, cv=10, scoring='precision_macro')
     print("Precision: %0.5f (+/- %0.5f)" % (precision.mean(), precision.std() * 2))
     recall = cross_val_score(clf_SVM_R2L, X_R2L_test, Y_R2L_test, cv=10, scoring='recall_macro')
     print("Recall: %0.5f (+/- %0.5f)" % (recall.mean(), recall.std() * 2))
     f = cross_val_score(clf_SVM_R2L, X_R2L_test, Y_R2L_test, cv=10, scoring='f1_macro')
     print("F-measure: %0.5f (+/- %0.5f)" % (f.mean(), f.std() * 2))
```

```
Accuracy: 0.96793 (+/- 0.00738)
Precision: 0.94854 (+/- 0.00994)
Recall: 0.96264 (+/- 0.01388)
F-measure: 0.95529 (+/- 0.01048)
```

```
[45] Accuracy_SVM_R2L = accuracy.mean()
```

```
     # U2R
     accuracy = cross_val_score(clf_SVM_U2R, X_U2R_test, Y_U2R_test, cv=10, scoring='accuracy')
     print("Accuracy: %0.5f (+/- %0.5f)" % (accuracy.mean(), accuracy.std() * 2))
     precision = cross_val_score(clf_SVM_U2R, X_U2R_test, Y_U2R_test, cv=10, scoring='precision_macro')
     print("Precision: %0.5f (+/- %0.5f)" % (precision.mean(), precision.std() * 2))
     recall = cross_val_score(clf_SVM_U2R, X_U2R_test, Y_U2R_test, cv=10, scoring='recall_macro')
     print("Recall: %0.5f (+/- %0.5f)" % (recall.mean(), recall.std() * 2))
     f = cross_val_score(clf_SVM_U2R, X_U2R_test, Y_U2R_test, cv=10, scoring='f1_macro')
     print("F-measure: %0.5f (+/- %0.5f)" % (f.mean(), f.std() * 2))
```

```
Accuracy: 0.99632 (+/- 0.00390)
Precision: 0.91056 (+/- 0.17934)
Recall: 0.82909 (+/- 0.21833)
F-measure: 0.84869 (+/- 0.16029)
```

```
[47] Accuracy_SVM_U2R = accuracy.mean()
```

Hence, by using the cross_val_score library, we got our different scores for all the four types of attacks.

7. RESULTS & DISCUSSIONS

In this section, accuracies of various algorithms are being compared by using graphical analysis.

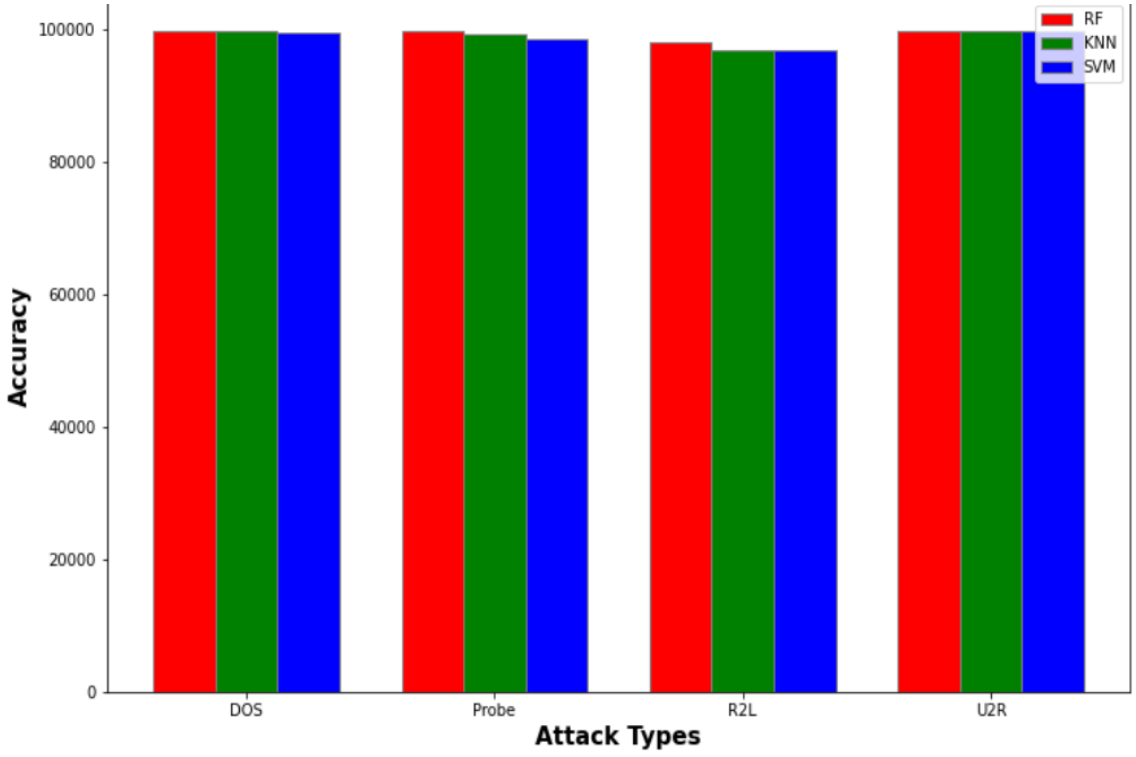
```
[48] # Tabular Comparison
     from tabulate import tabulate
```

```
[49] table = [['classification Algorithm', 'Class name', 'Test Accuracy'], ['Random Forest', 'DOS', Accuracy_RF_DOS], ['Random Forest',
     print(tabulate(table, headers='firstrow', tablefmt='fancy_grid'))
```

classification Algorithm	Class name	Test Accuracy
Random Forest	DOS	0.997379
Random Forest	Probe	0.997032
Random Forest	R2L	0.979994
Random Forest	U2R	0.997341
KNN	DOS	0.997146
KNN	Probe	0.990768
KNN	R2L	0.96737
KNN	U2R	0.997034
SVM	DOS	0.99371
SVM	Probe	0.984504
SVM	R2L	0.967926
SVM	U2R	0.996318

Here we can see that accuracy is almost similar using various algorithms, next we will visualize this using a graph for better visualizations.

```
[ ]  # Graphical Comparison

     barWidth = 0.25
     fig = pyplot.subplots(figsize =(12, 8))

     RF = [Accuracy_RF_DOS*100000, Accuracy_RF_Probe*100000, Accuracy_RF_R2L*100000, Accuracy_RF_U2R*100000]
     KNN = [Accuracy_KNN_DOS*100000, Accuracy_KNN_Probe*100000, Accuracy_KNN_R2L*100000, Accuracy_KNN_U2R*100000]
     SVM = [Accuracy_SVM_DOS*100000, Accuracy_SVM_Probe*100000, Accuracy_SVM_R2L*100000, Accuracy_SVM_U2R*100000]

     # Set position of bar on X axis
     br1 = np.arange(len(RF))
     br2 = [x + barWidth for x in br1]
     br3 = [x + barWidth for x in br2]

     # Make the plot
     pyplot.bar(br1, RF, color ='r', width = barWidth,
             edgecolor ='grey', label ='RF')
     pyplot.bar(br2, KNN, color ='g', width = barWidth,
             edgecolor ='grey', label ='KNN')
     pyplot.bar(br3, SVM, color ='b', width = barWidth,
             edgecolor ='grey', label ='SVM')

     pyplot.xlabel('Attack Types', fontweight ='bold', fontsize = 15)
     pyplot.ylabel('Accuracy', fontweight ='bold', fontsize = 15)
     pyplot.xticks([r + barWidth for r in range(len(RF))],
             ['DOS', 'Probe', 'R2L', 'U2R'])

     pyplot.legend()
     pyplot.show()
```



(Multiplied by 100000 because for less precision (decimal points) accuracy was almost same)

8. Conclusion

We observe that Random Forest has the highest accuracy when compared to other classifiers (KNN and SVM) for intrusion detection systems. Thus, an Intrusion Detection System (IDS) driven by NSL-KDD dataset can be constructed by training with Random Forest Algorithm.

Future Scope of Improvement

We can work on exploring new techniques having better accuracy rate than our current model.

One such technique that can be applied is to use **feature elimination**, which will provide us with a better accuracy.

Feature elimination can be done using **Recursive Feature Elimination (RFE)**.

Availability of a better dataset than NSL-KDD (fresher and more relevant) will also make our model more efficient.