Network Intrusion Detection

AI & CYBERSECURITY (DSCI 6015)

Final Project Report - Manasa Rangineni (00716955)

Submitted to

Dr. Vahid Behzadan

ASSISTANT PROFESSOR

UNIVERSITY OF NEW HAVEN

Fall 2021

Abstract

Network security is vital not only for business continuity, but also for the thousands of additional systems and applications that run continually through the network to supply services. Intrusion detection and prevention systems are one method of implementing and enforcing network security. Traditional intrusion detection systems are often rule-based and ineffective in detecting novel or previously undisclosed intrusion occurrences. Data mining techniques and machine algorithms have recently gotten a lot of interest as a new way to detect network security problems proactively. Six data mining algorithms are used in this study to detect and classify network intrusion using the NSL-KDD dataset: Support Vector Machine (SVM), Decision Tree and Random Forest, Naive Baye, K-Nearest Neighbor (KNN), and Logistic Regression classifiers. In general, the results show that models are biased towards classes with low distribution in the dataset.

Introduction

Intrusion is a term used to describe malicious activities that are directed at a computer network system in order to compromise its integrity, availability, or confidentiality. Intrusion detection systems can be used to enforce network security (IDS). Essentially intrusion detection device or application examines all incoming and outgoing network data, looking for known and unknown events in packets. Known events and violations are typically logged in a SIEM (security information and event management) system. Depending on the options enabled on the system, malicious activities or unknown events may be set up to inform the system administrator or the corresponding packets may be dropped. Efficient intrusion detection becomes critical for enterprises to deal with security risks in their networks in a proactive manner. Many existing intrusion detection systems, on the other hand, are rule-based and aren't very good at identifying novel intrusion events that aren't encoded in the existing rules. Furthermore, developing intrusion detection algorithms takes effort and is limited to only knowing about known intrusions.

Data mining approaches, on the other hand, have been found to be effective in identifying and distinguishing known and new intrusions from network event records or data using supervised and unsupervised learning algorithms. As a result, it's worth looking into using data mining techniques to detect known and potential network attacks as a viable option. The goal of this project is to implement data mining techniques in building models to classify network intrusions using algorithms such as Support Vector Machine (SVM), Decision Tree and Random Forest, Naive Baye, K-Nearest Neighbor (KNN) and Logistic Regression.

Classification Algorithms

Support Vector Machines (SVM): The key objective of SVM is to draw a hyperplane that separates the two classes optimally such that the margin is maximum between the hyperplane and the observations. Although, it is possible to highlight different hyperplanes within a given space, the goal of SVM is to find the one that produces a high margin. As all cases of classifications cannot be handled linearly, SVM employs some kernel functions like polynomial or radial basis function to transform the linear algorithms to nonlinear ones.

SVM belongs to the category of supervised learning algorithms that can analyze data and recognize patterns and are used for classification and regression analysis tasks. The

effectiveness of SVM depends on the selection of kernel, the kernel's parameters, and soft margin parameter.
A potential drawback of the SVM is that the parameters of a solved model are difficult to interpret. Despite this, SVM has found a wide applications especially in text classification tasks.

Decision trees (DT): Decision tree constructs a tree-like structure where each internal node denotes a test on an attribute, each branch corresponds to an outcome of the test, and each external (leaf) node denotes a class prediction. At each node, the algorithm chooses the best attribute to classify data into individual classes. Decision tree is one of the most commonly used machine learning approaches in the field of intrusion detection.
Decision trees are used in data mining either for classification where the predicted outcome is the class to which the data belongs or regression analysis where the predicted outcome can be considered a real number.

Random Forests: Random Forests are an ensemble learning method for classification, regression and other tasks, that operate by constructing a multitude of decision trees at training time and outputting the class that is the mode of the classes (classification) or mean prediction (regression) of the individual trees. Random Forests overcome the decision trees habit of overfitting to their training set.
One of the key advantages of Random Forests is that they can be used to rank the importance of variables in a regression or classification problem.

Project Methodology

The implementation methodology used in the project follows these functional steps: Environment Setup, Data Loading, Data Preprocessing, Training the Models, Evaluating Models and Testing the Models. The project is implemented using Python programming language.

Project Implementation and Results

KDD Dataset

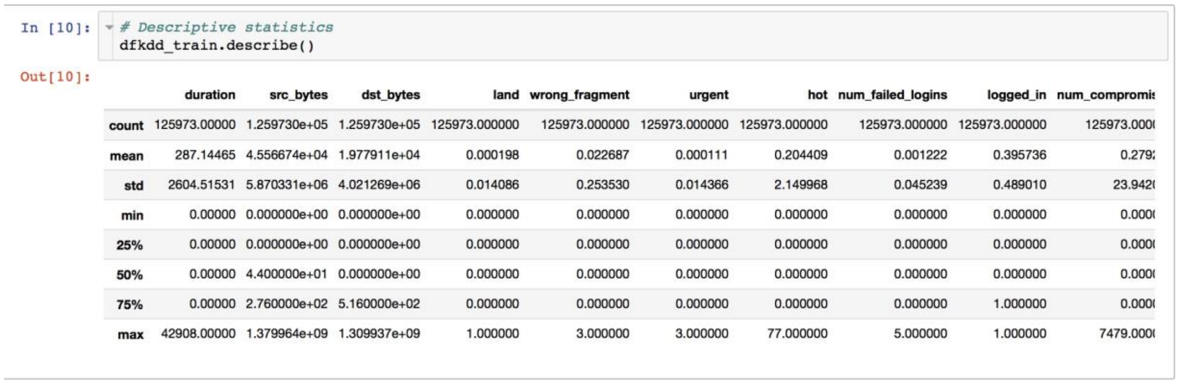
The dataset used in this project is the NSL-KDD dataset from Kaggle. The dataset is an improved version of the well known KDDCUP'99. The original KDD dataset is perhaps the most widely used dataset for machine learning intrusion detection tasks. However, it was revealed that the dataset is fraught with redundant records that can lead to poor evaluation of anomaly detection tasks. The NSL-KDD dataset used in this project consists of 125,973 records training set and 22,544 records test set with 42 attributes/features for each connection sample including class label containing the attack types.

Data Load and Preprocessing

A. Mapping intrusion types to attack classes

After loading the dataset into Python (Jupyter Notebook) development environment, the first task performed was mapping various attack types in the dataset into four attack classes.

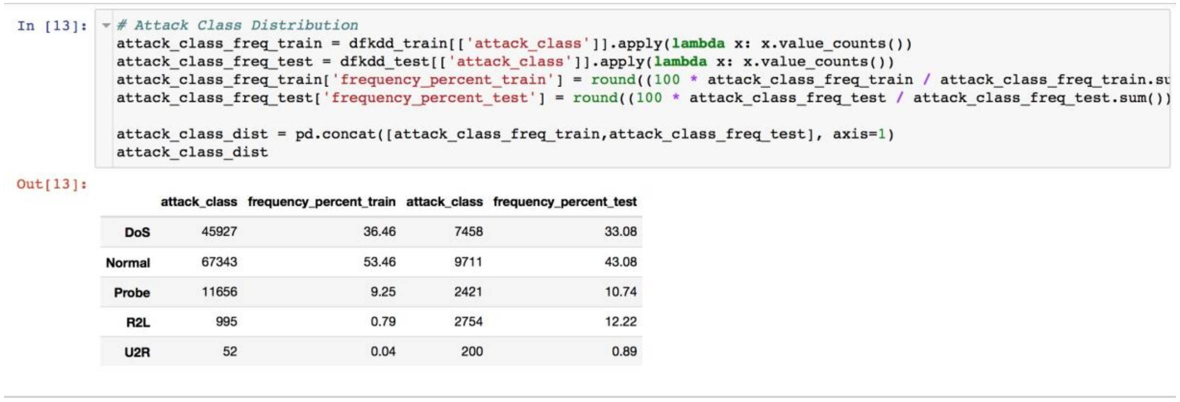
1. **Denial of Service (DoS)**: It is an attack in which an adversary directed a deluge of traffic requests to a system in order to make the computing or memory resource too busy or too full to handle legitimate requests and in the process, denies legitimate users access to a machine.
2. **Probing Attack (Probe)**: probing network of computers to gather information to be used to compromise its security controls.
3. **User to Root Attack (U2R)**: It is a class of exploit in which the adversary starts out with access to a normal user account on the system and is able to exploit some vulnerability to gain root access to the system.
4. **Remote to Local Attack (R2L)**: occurs when an attacker who has the ability to send packets to a machine over a network but who does not have an account on that machine exploits some vulnerability to gain local access as a user of that machine.



```
In [10]: # Descriptive statistics
         dfkdd_train.describe()
```

Out[10]:

	duration	src_bytes	dst_bytes	land	wrong_fragment	urgent	hot	num_failed_logins	logged_in	num_compromis
count	125973.00000	1.259730e+05	1.259730e+05	125973.000000	125973.000000	125973.000000	125973.000000	125973.000000	125973.000000	125973.000
mean	287.14465	4.556674e+04	1.977911e+04	0.000198	0.022687	0.000111	0.204409	0.001222	0.395736	0.279
std	2604.51531	5.870331e+06	4.021269e+06	0.014086	0.253530	0.014366	2.149968	0.045239	0.489010	23.942
min	0.00000	0.000000e+00	0.000000e+00	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000
25%	0.00000	0.000000e+00	0.000000e+00	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000
50%	0.00000	4.400000e+01	0.000000e+00	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000
75%	0.00000	2.760000e+02	5.160000e+02	0.000000	0.000000	0.000000	0.000000	0.000000	1.000000	0.000
max	42908.00000	1.379964e+09	1.309937e+09	1.000000	3.000000	3.000000	77.000000	5.000000	1.000000	7479.000



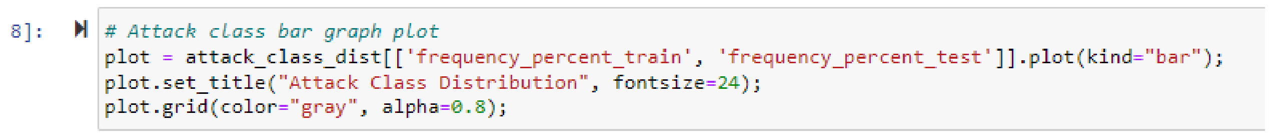
```
In [13]: # Attack Class Distribution
         attack_class_freq_train = dfkdd_train[['attack_class']].apply(lambda x: x.value_counts())
         attack_class_freq_test = dfkdd_test[['attack_class']].apply(lambda x: x.value_counts())
         attack_class_freq_train['frequency_percent_train'] = round((100 * attack_class_freq_train / attack_class_freq_train.su
         attack_class_freq_test['frequency_percent_test'] = round((100 * attack_class_freq_test / attack_class_freq_test.sum())

         attack_class_dist = pd.concat([attack_class_freq_train,attack_class_freq_test], axis=1)
         attack_class_dist
```

Out[13]:

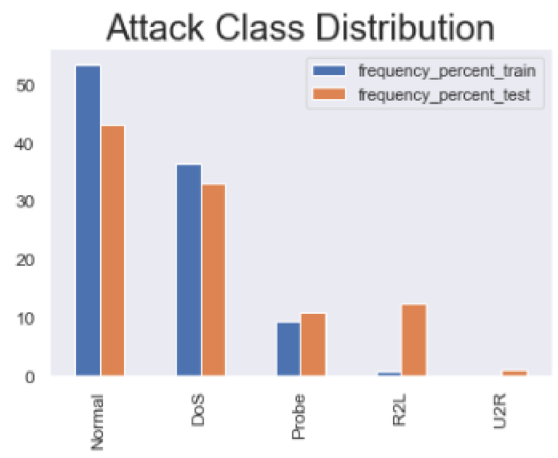
	attack_class	frequency_percent_train	attack_class	frequency_percent_test
DoS	45927	36.46	7458	33.08
Normal	67343	53.46	9711	43.08
Probe	11656	9.25	2421	10.74
R2L	995	0.79	2754	12.22
U2R	52	0.04	200	0.89

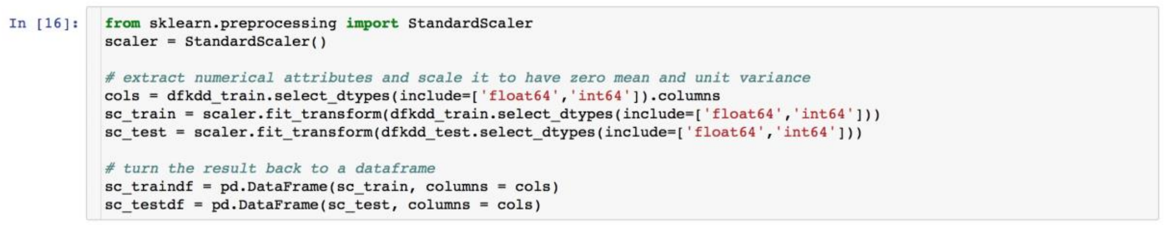
B. Exploratory Data Analysis (EDA)

Basic exploratory data analyses are carried out among other things to understand the descriptive statistics of the dataset, find instances of missing values and redundant features, explore the data type and structure and investigate the distribution of attack class in the dataset.



```
8]:   # Attack class bar graph plot
      plot = attack_class_dist[['frequency_percent_train', 'frequency_percent_test']].plot(kind="bar");
      plot.set_title("Attack Class Distribution", fontsize=24);
      plot.grid(color="gray", alpha=0.8);
```





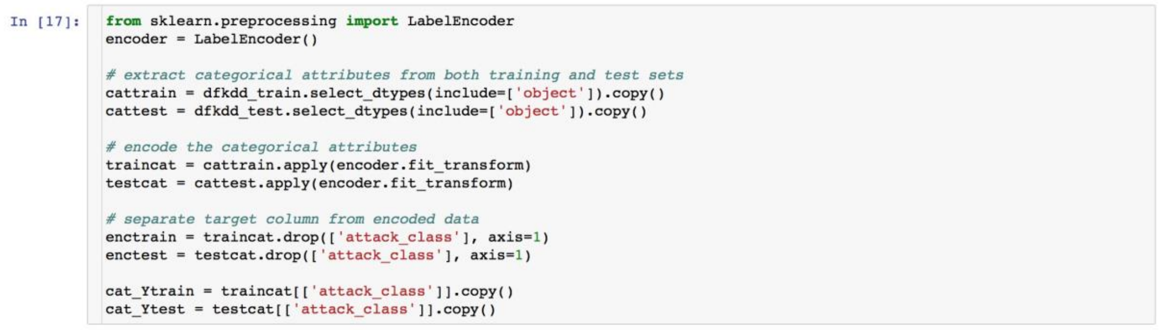
```
In [16]:  from sklearn.preprocessing import StandardScaler
          scaler = StandardScaler()

          # extract numerical attributes and scale it to have zero mean and unit variance
          cols = dfkdd_train.select_dtypes(include=['float64','int64']).columns
          sc_train = scaler.fit_transform(dfkdd_train.select_dtypes(include=['float64','int64']))
          sc_test = scaler.fit_transform(dfkdd_test.select_dtypes(include=['float64','int64']))

          # turn the result back to a dataframe
          sc_traindf = pd.DataFrame(sc_train, columns = cols)
          sc_testdf = pd.DataFrame(sc_test, columns = cols)
```

Standardization of Numerical Attributes

The numerical features in the dataset were extracted and standardized to have zero mean and unit variance. This is a common requirement for many machine learning algorithms implemented in Scikit-learn python module.

Encoding Categorical Attributes

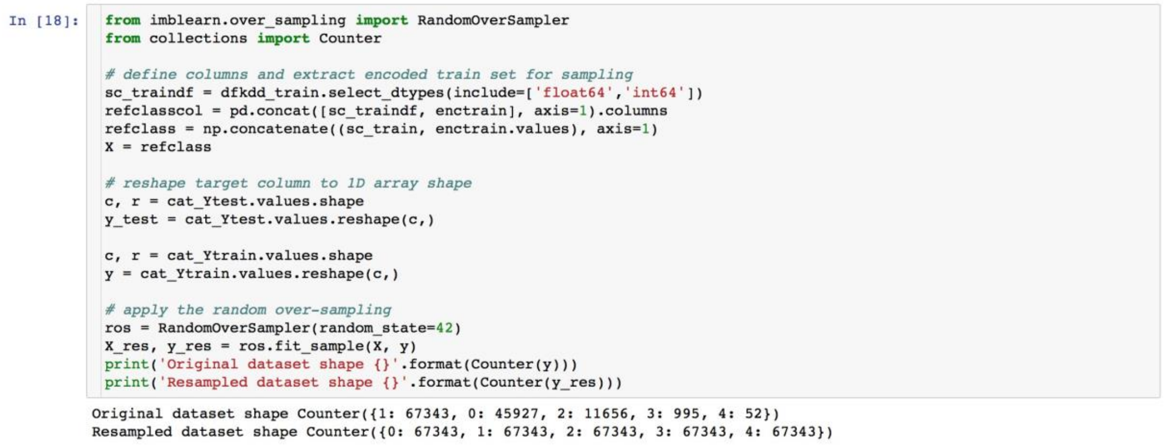
The categorical features in the dataset were encoded to integers. This is also a common requirement for many machine learning algorithms implemented in Scikit-learn.



```
In [17]: from sklearn.preprocessing import LabelEncoder
         encoder = LabelEncoder()

         # extract categorical attributes from both training and test sets
         cattrain = dfkdd_train.select_dtypes(include=['object']).copy()
         cattest = dfkdd_test.select_dtypes(include=['object']).copy()

         # encode the categorical attributes
         traincat = cattrain.apply(encoder.fit_transform)
         testcat = cattest.apply(encoder.fit_transform)

         # separate target column from encoded data
         enctrain = traincat.drop(['attack_class'], axis=1)
         enctest = testcat.drop(['attack_class'], axis=1)

         cat_Ytrain = traincat[['attack_class']].copy()
         cat_Ytest = testcat[['attack_class']].copy()
```

C. Data Sampling

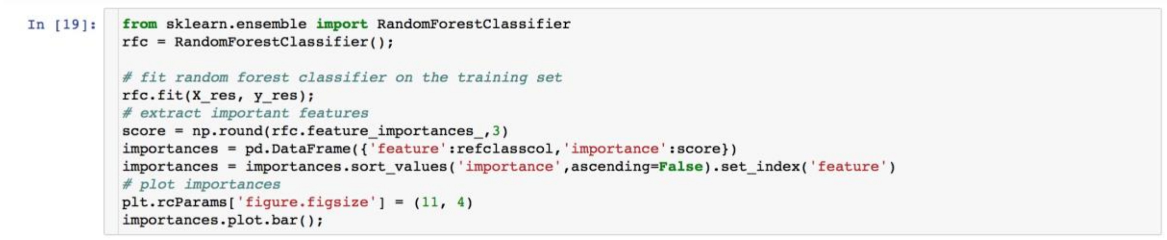
The sparse distribution of certain attack classes such as *U2R* and *L2R* in the dataset while others such as *Normal*, *DoS* and *Probe* are significantly represented inherently leads to the situation of imbalance dataset. While this scenario is not unexpected in data mining tasks involving identification or classification of instances of deviations from normal patterns in a given dataset, research has shown that supervised learning algorithms are often biased against the target class that is weakly represented in a given dataset.
The use of sampling involves modification of an imbalanced data set by some mechanisms in order to provide a balanced distribution. While oversampling replicates and increases data in class label with low distribution, random undersampling removes data from the original dataset with high class frequency.
In this project random oversampling technique was used to balance class distribution in the training dataset as shown below.



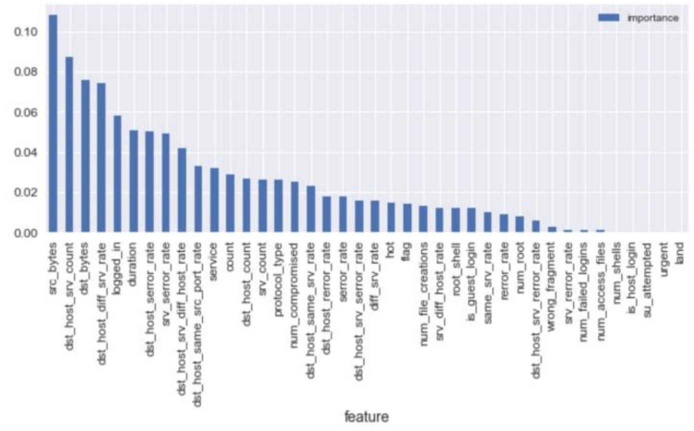
```
In [18]: from imblearn.over_sampling import RandomOverSampler
         from collections import Counter

         # define columns and extract encoded train set for sampling
         sc_traindf = dfkdd_train.select_dtypes(include=['float64','int64'])
         refclasscol = pd.concat([sc_traindf, enctrain], axis=1).columns
         refclass = np.concatenate((sc_train, enctrain.values), axis=1)
         X = refclass

         # reshape target column to 1D array shape
         c, r = cat_Ytest.values.shape
         y_test = cat_Ytest.values.reshape(c,)

         c, r = cat_Ytrain.values.shape
         y = cat_Ytrain.values.reshape(c,)

         # apply the random over-sampling
         ros = RandomOverSampler(random_state=42)
         X_res, y_res = ros.fit_sample(X, y)
         print('Original dataset shape {}'.format(Counter(y)))
         print('Resampled dataset shape {}'.format(Counter(y_res)))

         Original dataset shape Counter({1: 67343, 0: 45927, 2: 11656, 3: 995, 4: 52})
         Resampled dataset shape Counter({0: 67343, 1: 67343, 2: 67343, 3: 67343, 4: 67343})
```

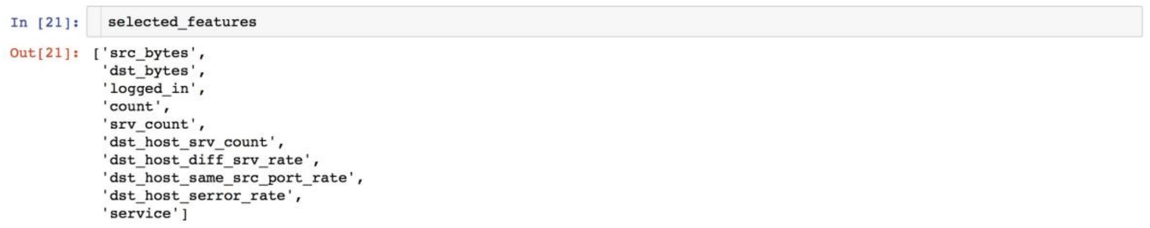
D. Feature Selection

Feature selection is a key data preprocessing step in data mining task involving selection of important features as a subset of original features according to certain criteria to reduce dimension in order to improve the efficiency of data mining algorithms. Most of the data includes irrelevant, redundant, or noisy features. Feature selection reduces the number of features, removes irrelevant, redundant, or noisy features, and improves learning accuracy, and leading to better model comprehensibility.

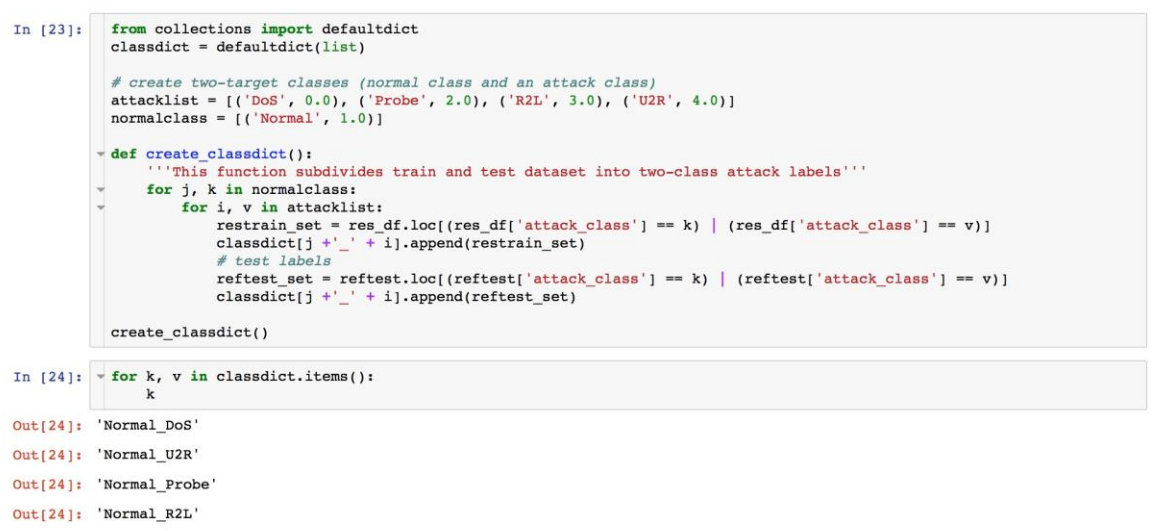


```
In [19]:  from sklearn.ensemble import RandomForestClassifier
          rfc = RandomForestClassifier();

          # fit random forest classifier on the training set
          rfc.fit(X_res, y_res);
          # extract important features
          score = np.round(rfc.feature_importances_,3)
          importances = pd.DataFrame({'feature':refclasscol,'importance':score})
          importances = importances.sort_values('importance',ascending=False).set_index('feature')
          # plot importances
          plt.rcParams['figure.figsize'] = (11, 4)
          importances.plot.bar();
```





```
In [21]:  selected_features

Out[21]: ['src_bytes',
          'dst_bytes',
          'logged_in',
          'count',
          'srv_count',
          'dst_host_srv_count',
          'dst_host_diff_srv_rate',
          'dst_host_same_src_port_rate',
          'dst_host_serror_rate',
          'service']
```



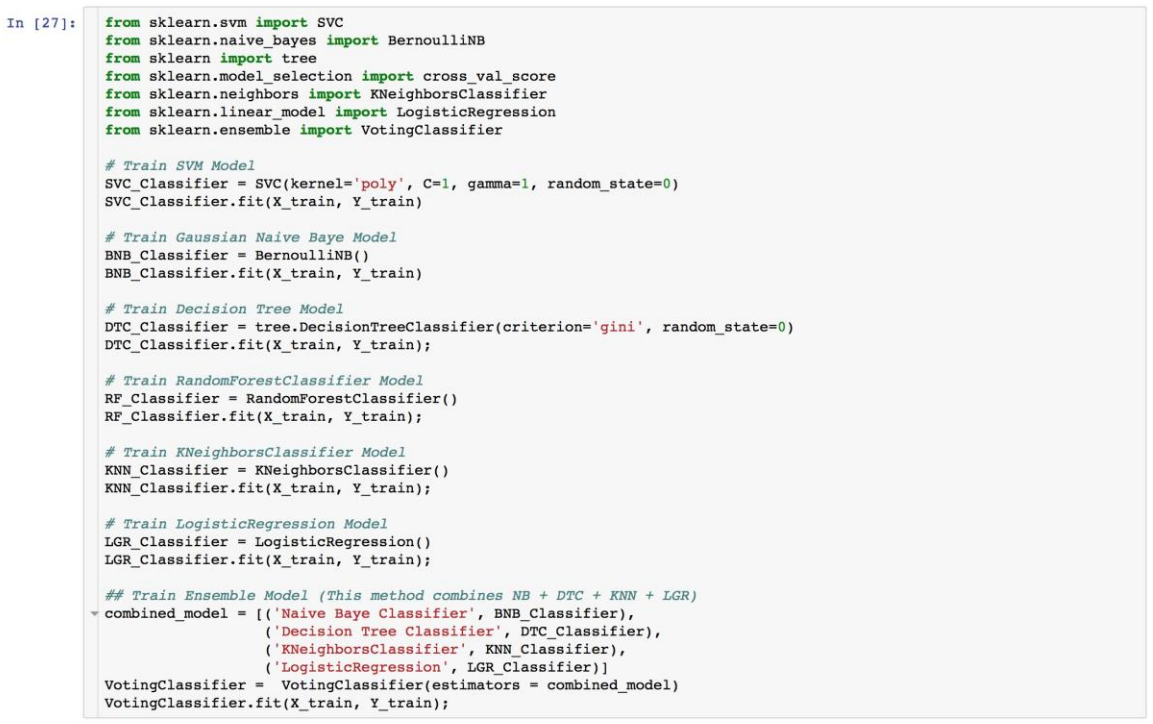
```
In [23]: from collections import defaultdict
         classdict = defaultdict(list)

         # create two-target classes (normal class and an attack class)
         attacklist = [('DoS', 0.0), ('Probe', 2.0), ('R2L', 3.0), ('U2R', 4.0)]
         normalclass = [('Normal', 1.0)]

         def create_classdict():
             '''This function subdivides train and test dataset into two-class attack labels'''
             for j, k in normalclass:
                 for i, v in attacklist:
                     restrain_set = res_df.loc[(res_df['attack_class'] == k) | (res_df['attack_class'] == v)]
                     classdict[j +'_' + i].append(restrain_set)
                     # test labels
                     reftest_set = reftest.loc[(reftest['attack_class'] == k) | (reftest['attack_class'] == v)]
                     classdict[j +'_' + i].append(reftest_set)

         create_classdict()

In [24]: for k, v in classdict.items():
             k

Out[24]: 'Normal_DoS'

Out[24]: 'Normal_U2R'

Out[24]: 'Normal_Probe'

Out[24]: 'Normal_R2L'
```

E. Data Partition

After selection of relevant features, The resampled dataset based on the selected features was partitioned into two-target classes (normal class and an attack class) for all the attack classes in the dataset to facilitate binary classification.

Train Models

The training dataset is used to train the following classifier algorithms: Support Vector Machine (SVM), Decision Tree, Random Forest, Naïve Baye, Logistic Regression and k-Nearest Neighbor (kNN).

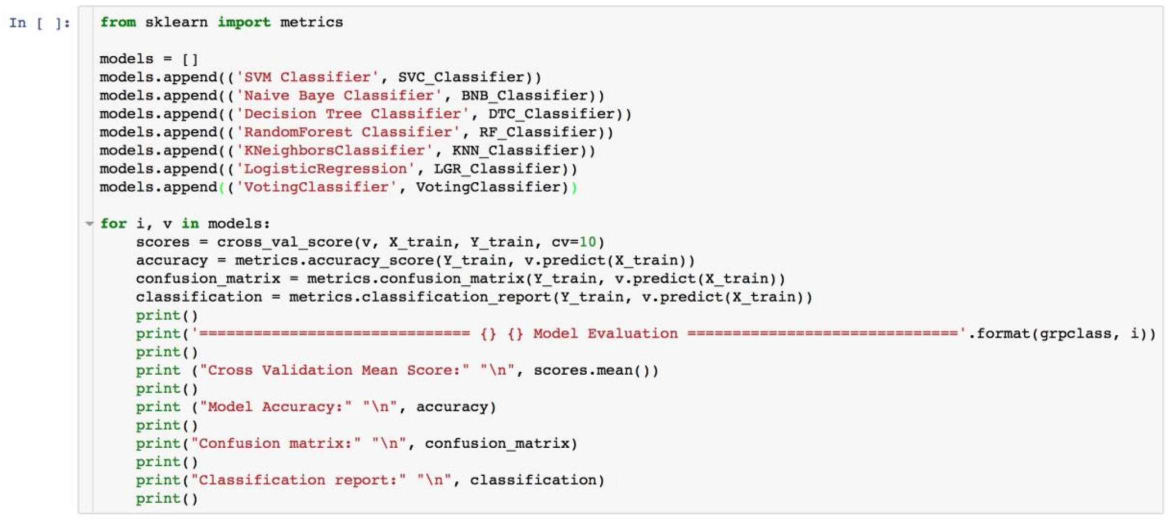


```
In [27]: from sklearn.svm import SVC
         from sklearn.naive_bayes import BernoulliNB
         from sklearn import tree
         from sklearn.model_selection import cross_val_score
         from sklearn.neighbors import KNeighborsClassifier
         from sklearn.linear_model import LogisticRegression
         from sklearn.ensemble import VotingClassifier

         # Train SVM Model
         SVC_Classifier = SVC(kernel='poly', C=1, gamma=1, random_state=0)
         SVC_Classifier.fit(X_train, Y_train)

         # Train Gaussian Naive Baye Model
         BNB_Classifier = BernoulliNB()
         BNB_Classifier.fit(X_train, Y_train)

         # Train Decision Tree Model
         DTC_Classifier = tree.DecisionTreeClassifier(criterion='gini', random_state=0)
         DTC_Classifier.fit(X_train, Y_train);

         # Train RandomForestClassifier Model
         RF_Classifier = RandomForestClassifier()
         RF_Classifier.fit(X_train, Y_train);

         # Train KNeighborsClassifier Model
         KNN_Classifier = KNeighborsClassifier()
         KNN_Classifier.fit(X_train, Y_train);

         # Train LogisticRegression Model
         LGR_Classifier = LogisticRegression()
         LGR_Classifier.fit(X_train, Y_train);

         ## Train Ensemble Model (This method combines NB + DTC + KNN + LGR)
         combined_model = [('Naive Baye Classifier', BNB_Classifier),
                           ('Decision Tree Classifier', DTC_Classifier),
                           ('KNeighborsClassifier', KNN_Classifier),
                           ('LogisticRegression', LGR_Classifier)]
         VotingClassifier =  VotingClassifier(estimators = combined_model)
         VotingClassifier.fit(X_train, Y_train);
```

Evaluate Models

The trained models were evaluated using a 10-fold cross validation technique. The concept of *k*-fold cross validation involves generation of validation set out of training dataset to assess the model before it is exposed to test data.

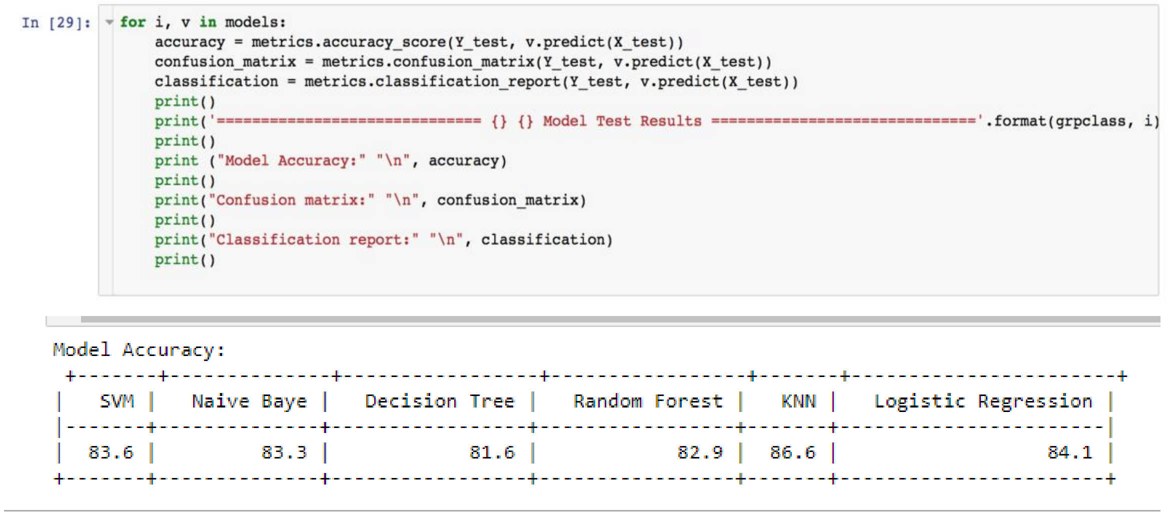
In *k*-fold cross-validation, the training dataset is randomly divided into *k* equal sized subsamples. Out of the *k* subsamples, a single subsample is retained as the validation data for testing the model, and the remaining *k*− 1 subsamples are used as training data. The cross-validation process is then repeated *k* times (the folds), with each of the *k* subsamples used exactly once as the validation data. The *k* results from the folds can then be averaged or combined to generate a single value. The advantage of this method is that all samples are used for both training and validation [5].



```
In [ ]:  from sklearn import metrics

         models = []
         models.append(('SVM Classifier', SVC_Classifier))
         models.append(('Naive Baye Classifier', BNB_Classifier))
         models.append(('Decision Tree Classifier', DTC_Classifier))
         models.append(('RandomForest Classifier', RF_Classifier))
         models.append(('KNeighborsClassifier', KNN_Classifier))
         models.append(('LogisticRegression', LGR_Classifier))
         models.append(('VotingClassifier', VotingClassifier))

         for i, v in models:
             scores = cross_val_score(v, X_train, Y_train, cv=10)
             accuracy = metrics.accuracy_score(Y_train, v.predict(X_train))
             confusion_matrix = metrics.confusion_matrix(Y_train, v.predict(X_train))
             classification = metrics.classification_report(Y_train, v.predict(X_train))
             print()
             print('============================== {} {} Model Evaluation =============================='.format(grpclass, i))
             print()
             print ("Cross Validation Mean Score:" "\n", scores.mean())
             print()
             print ("Model Accuracy:" "\n", accuracy)
             print()
             print("Confusion matrix:" "\n", confusion_matrix)
             print()
             print("Classification report:" "\n", classification)
             print()
```

Test Models Performance

The trained models were used to classify labels in a test dataset that has not been previously exposed to the algorithms. Performance metrics such as accuracy, confusion matrix and classification report were generated for each two-target attack class and for each model.



```
In [29]:  for i, v in models:
              accuracy = metrics.accuracy_score(Y_test, v.predict(X_test))
              confusion_matrix = metrics.confusion_matrix(Y_test, v.predict(X_test))
              classification = metrics.classification_report(Y_test, v.predict(X_test))
              print()
              print('============================== {} {} Model Test Results =============================='.format(grpclass, i)
              print()
              print ("Model Accuracy:" "\n", accuracy)
              print()
              print("Confusion matrix:" "\n", confusion_matrix)
              print()
              print("Classification report:" "\n", classification)
              print()
```

```
Model Accuracy:
 +-------+--------------+----------------+----------------+-------+----------------------+
|  SVM  |  Naive Baye  |  Decision Tree |  Random Forest |  KNN  |  Logistic Regression |
|-------+--------------+----------------+----------------+-------+----------------------|
|  83.6 |         83.3 |           81.6 |           82.9 |  86.6 |                 84.1 |
+-------+--------------+----------------+----------------+-------+----------------------+
```

Results and Discussion

From the results obtained, all the models evaluated achieved an average of 99% on training set while model's performance on test set indicates an average of more than 80% across all the four attack groups investigated.

The test accuracy for 'Normal_U2R' across all the models showed a value of more than 90%. However, a review of performance as shown by the confusion matrix indicated that the 'U2R' detection rate was very poor across Decision Tree, RandomForest, KNN and Logistic Regression models. The attack class 'R2L' detection rate is also not far from being poor as 'Normal' and other classes with higher distribution got more attention of the models to the detriment of the low distribution classes.

The results generally showed that sampling may improve model training accuracy. However, a poor detection rate in detecting U2R and R2L minority attacks are inevitable because of their large bias available in the dataset.

Conclusion

A project was carried out to build models for classifying network intrusions using NSL-KDD dataset. Classifier algorithms such as Support Vector Machine (SVM), Naive Baye, Decision Tree, K-Nearest Neighbor, Logistic Regression and Random Forest were trained, evaluated and tested to determine each model performance.

A key lesson learned in this project is that, for classification task, accuracy is not a good measure of a model performance where there is an imbalance dataset. Accuracy value may be high for the model but the class with lower samples may not be effectively classified. Metrics such as Confusion Matrix is more realistic in evaluating classifier model performance than accuracy measure.