

ADFA-LD - Model Evaluation

In [1]:

```
1 import seaborn as sns
2 import matplotlib.pyplot as plt
3 import pandas as pd
4 from IPython.display import display
5 pd.options.display.max_columns = None
6 from sklearn.ensemble import RandomForestRegressor, RandomForestClassifier
7 from IPython.display import display
8 from sklearn import metrics
9 from sklearn.model_selection import train_test_split
10 import statistics
11 import numpy as np
12 from sklearn import metrics
13 from sklearn.preprocessing import MinMaxScaler, StandardScaler, LabelEncoder
14 from sklearn.feature_selection import SelectKBest
15 from sklearn.pipeline import Pipeline
16 from sklearn.model_selection import train_test_split, GridSearchCV, RandomizedSearchCV
```

In [2]:

```
1 import numpy as np
2 import pandas as pd
3 import matplotlib.pyplot as plt
4 import seaborn as sns
5 from keras.preprocessing.text import Tokenizer
6 from keras.preprocessing.sequence import pad_sequences
7 from keras.models import Sequential
8 from keras.layers import Dense, Embedding, LSTM, SpatialDropout1D
9 from sklearn.model_selection import train_test_split
10 from keras.utils.np_utils import to_categorical
11 from keras.callbacks import EarlyStopping
12 from keras.layers import Dropout
13 import re
```

Using TensorFlow backend.

In [3]:

```

1  import glob
2  import math
3  from collections import Counter
4  import csv
5
6  import numpy as np
7
8
9  def plot_confusion_matrix(cm,
10                          target_names,
11                          title='Confusion matrix',
12                          cmap=None,
13                          normalize=True):
14      import matplotlib.pyplot as plt
15      import numpy as np
16      import itertools
17
18      accuracy = np.trace(cm) / float(np.sum(cm))
19      misclass = 1 - accuracy
20
21      if cmap is None:
22          cmap = plt.get_cmap('Blues')
23
24      plt.figure(figsize=(8, 6))
25      plt.imshow(cm, interpolation='nearest', cmap=cmap)
26      plt.title(title)
27      plt.colorbar()
28
29      if target_names is not None:
30          tick_marks = np.arange(len(target_names))
31          plt.xticks(tick_marks, target_names, rotation=45)
32          plt.yticks(tick_marks, target_names)
33
34      if normalize:
35          cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
36
37
38      thresh = cm.max() / 1.5 if normalize else cm.max() / 2
39      for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
40          if normalize:
41              plt.text(j, i, "{:0.4f}".format(cm[i, j]),
42                      horizontalalignment="center",
43                      color="white" if cm[i, j] > thresh else "black")
44          else:
45              plt.text(j, i, "{:,}".format(cm[i, j]),
46                      horizontalalignment="center",
47                      color="white" if cm[i, j] > thresh else "black")
48      plt.tight_layout()
49      plt.ylabel('True label')
50      plt.xlabel('Predicted label\naccuracy={:0.4f}; misclass={:0.4f}'.format(accuracy,
51      plt.show()
52
53  # returns a dictionary of n-grams frequency for any list
54  def ngrams_freq(listname, n):
55      counts = dict()
56      # make n-grams as string iteratively
57      grams = [' '.join(listname[i:i+n]) for i in range(len(listname)-n)]
58      for gram in grams:
59          if gram not in counts:

```

```

60         counts[gram] = 1
61     else:
62         counts[gram] += 1
63     return counts
64
65 # returns the values of features for any list
66 def feature_freq(listname,n,features):
67     counts = dict()
68     # make n-grams as string iteratively
69     grams = [' '.join(listname[i:i+n]) for i in range(len(listname)-n)]
70     for gram in grams:
71         counts[gram] = 0
72     for gram in grams:
73         if gram in features:
74             counts[gram] += 1
75     return counts
76
77 # values of n for finding n-grams
78 n_values = [1]
79
80 # Base address for attack data files
81 add = "ADFA-LD/ADFA-LD/Attack_Data_Master/"
82 # list of attacks
83 attack = ['Adduser','Hydra_FTP','Hydra_SSH','Java_Meterpreter','Meterpreter','Web_Shell']
84
85 # initializing dictionary for n-grams from all files
86 tra indict = {}
87
88 Attack_list_new = []
89 print("Generating Training Data .....")
90 for term in attack:
91     print(" Training data from " + term)
92     globals()['%s_list' % term] = []
93     in_address = add+term
94     k = 1
95     # finding list of data from all files
96     for i in range (1,11):
97         read_files = glob.glob(in_address+"_"+str(i)+"/*.txt")
98         for f in read_files:
99             with open(f, "r") as infile:
100                 globals()['%s_list_array' % term+str(k)] = ALine =infile.read()
101                 #ALine = ALine[:820]
102                 Attack_list_new.append(term + ',' + str(ALine))
103                 globals()['%s_list' % term].extend(globals()['%s_list_array' % term+str(k)])
104                 k += 1
105     # number of lists for distinct files
106     globals()['%s_size' % term] = k-1
107     # combined list of all files
108     listname = globals()['%s_list' % term]
109     # finding n-grams and extracting top 30%
110     for n in n_values:
111         #print("      Extracting top 30% "+str(n)+"-grams from "+term+".....")
112         dictname = ngrams_freq(listname,n)
113         top = math.ceil(0.3*len(dictname))
114         dictname = Counter(dictname)
115         for k, v in dictname.most_common(top):
116             tra indict.update({k : v})
117
118 # finding training data for Normal file
119 print(" Training data from Normal")
120 Normal_list = []

```

```

121 Normal_list_new = []
122 in_address = "ADFA-LD/ADFA-LD/Training_Data_Master/"
123 k = 1
124 read_files = glob.glob(in_address+"/*.txt")
125 for f in read_files:
126     with open(f, "r") as infile:
127         globals()['Normal%s_list_array' % str(k)] = Line = infile.read()
128         Normal_list_new.append('Normal,'+ str(Line))
129         Normal_list.extend(globals()['Normal%s_list_array' % str(k)])
130         k += 1
131
132 # number of lists for distinct files
133 Normal_list_size = k-1
134 # combined list of all files
135 listname = Normal_list
136
137
138 print("\nnew_train.csv created.....\n")
139

```

Generating Training Data

```

Training data from Adduser
Training data from Hydra_FTP
Training data from Hydra_SSH
Training data from Java_Meterpreter
Training data from Meterpreter
Training data from Web_Shell
Training data from Normal

```

new_train.csv created.....

In [4]:

```

1 new_train_list = []
2 new_train_list = Normal_list_new + Attack_list_new
3 #new_train_list[1]
4 #Attack_list_new[1]
5

```

In [5]:

```

1 new_train_list = []
2 new_train_list = Normal_list_new + Attack_list_new
3
4
5 with open('new_train.csv', 'w') as f:
6     for item in new_train_list:
7         f.write("%s\n" % item)

```

In [6]:

```

1 train = pd.read_csv("./new_train.csv", sep=',', error_bad_lines=False, header=None, name
2 train.head(5)
3 train.shape
4 #train.info()
5
6 #train.describe(include = 'all')
7 train_df = train.copy()
8 train['Label'] = train['Label'].astype('category')
9 train['CallTrace'] = train['CallTrace'].astype('category')
10
11 train['Label'].value_counts()
12 #train['CallTrace'].value_counts()

```

Out[6]:

```

Normal      833
Hydra_SSH   176
Hydra_FTP   162
Java_Meterpreter 124
Web_Shell   118
Adduser     91
Meterpreter 75
Name: Label, dtype: int64

```

In [7]:

```

1 train['Label_Codes'] = train['Label'].cat.codes
2 train['CallTrace_Codes'] = train['CallTrace'].cat.codes
3 train['Label_Codes'].value_counts()

```

Out[7]:

```

5      833
2      176
1      162
3      124
6      118
0       91
4       75
Name: Label_Codes, dtype: int64

```

In [8]:

```

1 train.head()

```

Out[8]:

	Label	CallTrace	Label_Codes	CallTrace_Codes
0	Normal	6 6 63 6 42 120 6 195 120 6 6 114 114 1 1 252 ...	5	1407
1	Normal	54 175 120 175 175 3 175 175 120 175 120 175 1...	5	1239
2	Normal	6 11 45 33 192 33 5 197 192 6 33 5 3 197 192 1...	5	1286
3	Normal	7 174 174 5 197 197 6 13 195 4 4 118 6 91 38 5...	5	1465
4	Normal	11 45 33 192 33 5 197 192 6 33 5 3 197 192 192...	5	93

Multinomial Logistic Regression

In [9]:

```
1 import warnings
2 warnings.filterwarnings("ignore")
3
4 # split the dataset in train and test
5 X = train.iloc[:, [3]].values
6 y = train.iloc[:, 2].values
7
8
9 # Splitting the dataset into the Training set and Test set
10 from sklearn.model_selection import train_test_split
11 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state=0)
12
13 # Feature Scaling
14 from sklearn.preprocessing import StandardScaler
15 sc = StandardScaler()
16 X_train = sc.fit_transform(X_train)
17 X_test = sc.transform(X_test)
18
19 # Fitting Logistic Regression to the Training set
20 from sklearn.linear_model import LogisticRegression
21 classifier = LogisticRegression(multi_class='ovr', solver = 'lbfgs')
22 classifier.fit(X_train, y_train)
23
24 # Predicting the Test set results
25 y_pred = classifier.predict(X_test)
26
27 # How did our model perform?
28 from sklearn import metrics
29 count_misclassified = (y_test != y_pred).sum()
30 print('Misclassified samples: {}'.format(count_misclassified))
31 accuracy = metrics.accuracy_score(y_test, y_pred)
32 print('Accuracy: {:.2f}'.format(accuracy))
33
34
35
```

Misclassified samples: 145

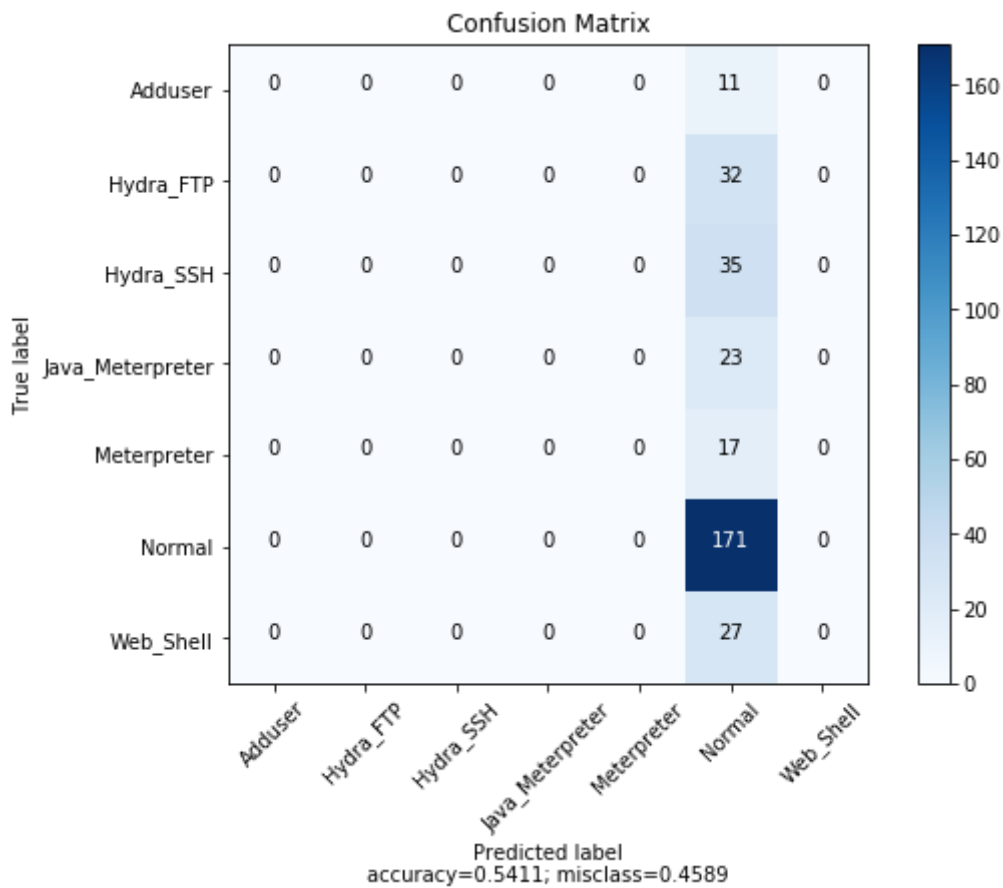
Accuracy: 0.54

In [10]:

```

1 #classifier.predict_proba(X_test)
2
3 # Making the Confusion Matrix
4 from sklearn.metrics import confusion_matrix
5 cm = confusion_matrix(y_test, y_pred)
6 plot_confusion_matrix(cm,
7                        normalize      = False,
8                        target_names  = ['Adduser', 'Hydra_FTP', 'Hydra_SSH', 'Java_Meterpreter',
9                        title         = "Confusion Matrix")

```



Logistic Regression Binary Classification

In [11]:

```
1 train.loc[train.Label != 'Normal','Label_Binary']= 1
2 train.loc[train.Label == 'Normal','Label_Binary']= 0
3 train['Label_Binary'].value_counts()
4 #train.head()
```

Out[11]:

0.0 833
1.0 746
Name: Label_Binary, dtype: int64

In [12]:

```
1 train.head()
```

Out[12]:

	Label	CallTrace	Label_Codes	CallTrace_Codes	Label_Binary
0	Normal	6 6 63 6 42 120 6 195 120 6 6 114 114 1 1 252 ...	5	1407	0.0
1	Normal	54 175 120 175 175 3 175 175 120 175 120 175 1...	5	1239	0.0
2	Normal	6 11 45 33 192 33 5 197 192 6 33 5 3 197 192 1...	5	1286	0.0
3	Normal	7 174 174 5 197 197 6 13 195 4 4 118 6 91 38 5...	5	1465	0.0
4	Normal	11 45 33 192 33 5 197 192 6 33 5 3 197 192 192...	5	93	0.0

In [13]:

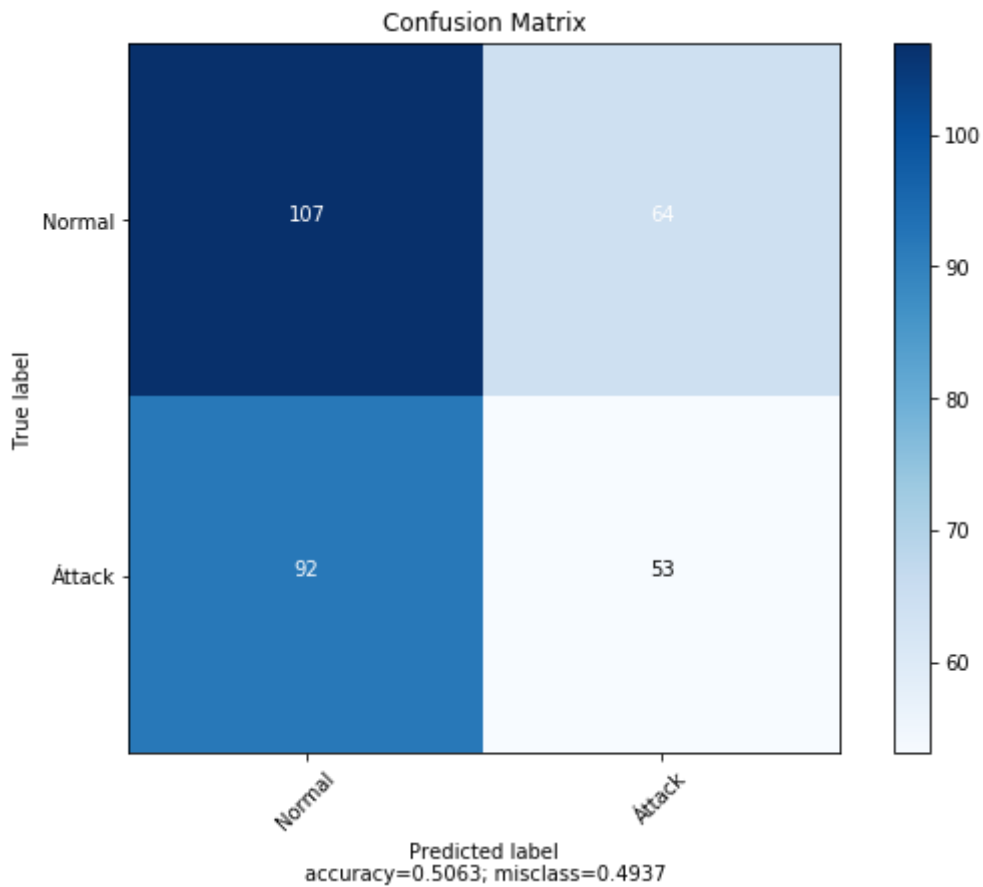
```
1 import warnings
2 warnings.filterwarnings("ignore")
3
4 # split the dataset in train and test
5 X = train.iloc[:, [3]].values
6 y = train.iloc[:, 4].values
7
8
9 # Splitting the dataset into the Training set and Test set
10 from sklearn.model_selection import train_test_split
11 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state=0)
12
13 # Feature Scaling
14 from sklearn.preprocessing import StandardScaler
15 sc = StandardScaler()
16 X_train = sc.fit_transform(X_train)
17 X_test = sc.transform(X_test)
18
19 # Fitting Logistic Regression to the Training set
20 from sklearn.linear_model import LogisticRegression
21 #classifier = LogisticRegression(multi_class='ovr', solver = 'lbfgs')
22 classifier = LogisticRegression()
23 classifier.fit(X_train, y_train)
24
25 # Predicting the Test set results
26 y_pred = classifier.predict(X_test)
27
28 # How did our model perform?
29 from sklearn import metrics
30 count_misclassified = (y_test != y_pred).sum()
31 print('Misclassified samples: {}'.format(count_misclassified))
32 accuracy = metrics.accuracy_score(y_test, y_pred)
33 print('Accuracy: {:.2f}'.format(accuracy))
34
35
36
```

Misclassified samples: 156

Accuracy: 0.51

In [14]:

```
1 # Making the Confusion Matrix
2 from sklearn.metrics import confusion_matrix
3 cm = confusion_matrix(y_test, y_pred)
4 plot_confusion_matrix(cm,
5                       normalize = False,
6                       target_names = ['Normal', 'Attack'],
7                       title = "Confusion Matrix")
```



In [15]:

```
1 print(metrics.classification_report(y_pred, y_test))
```

	precision	recall	f1-score	support	
	0.0	0.63	0.54	0.58	199
	1.0	0.37	0.45	0.40	117
micro avg	0.51	0.51	0.51		316
macro avg	0.50	0.50	0.49		316
weighted avg	0.53	0.51	0.51		316

OneHotEncoding for LogisticRegression

In [16]:

```
1 # Split into predictor and response dataframes.
2 train_df_enc = train_df.copy()
3 X_df = train_df_enc.drop('Label', axis=1)
4 y = train_df_enc['Label']
5
6 X_df.shape,y.shape
```

Out[16]:

((1579, 1), (1579,))

In [17]:

```
1 X_df.head()
```

Out[17]:

	CallTrace
0	6 6 63 6 42 120 6 195 120 6 6 114 114 1 1 252 ...
1	54 175 120 175 175 3 175 175 120 175 120 175 1...
2	6 11 45 33 192 33 5 197 192 6 33 5 3 197 192 1...
3	7 174 174 5 197 197 6 13 195 4 4 118 6 91 38 5...
4	11 45 33 192 33 5 197 192 6 33 5 3 197 192 192...

In [18]:

```
1 train_df.head()
```

Out[18]:

	Label	CallTrace
0	Normal	6 6 63 6 42 120 6 195 120 6 6 114 114 1 1 252 ...
1	Normal	54 175 120 175 175 3 175 175 120 175 120 175 1...
2	Normal	6 11 45 33 192 33 5 197 192 6 33 5 3 197 192 1...
3	Normal	7 174 174 5 197 197 6 13 195 4 4 118 6 91 38 5...
4	Normal	11 45 33 192 33 5 197 192 6 33 5 3 197 192 192...

In [19]:

```
1 # Map response variable to integers 0,1.
2 y = pd.Series(np.where(y.values != 'Normal',1,0), y.index)
3 y.value_counts()
```

Out[19]:

```
0    833
1    746
dtype: int64
```

In [20]:

```
1 # Label Encode instead of dummy variables
2
3 mappings = []
4
5 from sklearn.preprocessing import LabelEncoder
6
7 label_encoder = LabelEncoder()
8
9 label_df = train.drop('Label', axis=1)
10 label_df = train.drop('Label_Binary', axis=1)
11 label_df = train.drop('Label_Codes', axis=1)
12 label_df['CallTrace'] = label_df['CallTrace_Codes']
13 label_df = X_df.copy()
14 for i, col in enumerate(label_df):
15     if label_df[col].dtype == 'object':
16         label_df[col] = label_encoder.fit_transform(np.array(label_df[col].astype(str)))
17         mappings.append(dict(zip(label_encoder.classes_, range(1, len(label_encoder.cl
```

In [21]:

```
1 label_df.head()
```

Out[21]:

	CallTrace
0	1407
1	1239
2	1286
3	1465
4	93

In [22]:

```
1 from sklearn.preprocessing import OneHotEncoder
2
3
4 onehot_encoder = OneHotEncoder()
5 for i, col in enumerate(label_df):
6     if label_df[col].dtype == 'object':
7         label_df[col] = onehot_encoder.fit_transform(np.array(label_df[col].astype(str)))
8         mappings.append(dict(zip(onehot_encoder.classes_, range(1, len(onehot_encoder.classes_)))))
```

In [23]:

```
1 label_df[col].head()
```

Out[23]:

```
0    1407
1    1239
2    1286
3    1465
4      93
Name: CallTrace, dtype: int32
```

In [24]:

```
1 X_train, X_test, y_train, y_test = train_test_split(label_df, y, test_size = 0.2, random_state=42)
2 X_train.shape, X_test.shape, y_train.shape, y_test.shape
```

Out[24]:

```
((1263, 1), (316, 1), (1263,), (316,))
```

In [25]:

```
1 clf = LogisticRegression()
2 model_mix = clf.fit(X_train, y_train)
3 # y_pred = model_norm.predict(X_test)
4 print("Model accuracy is", model_mix.score(X_test, y_test))
```

```
Model accuracy is 0.5569620253164557
```

In [26]:

```
1 model_mix
```

Out[26]:

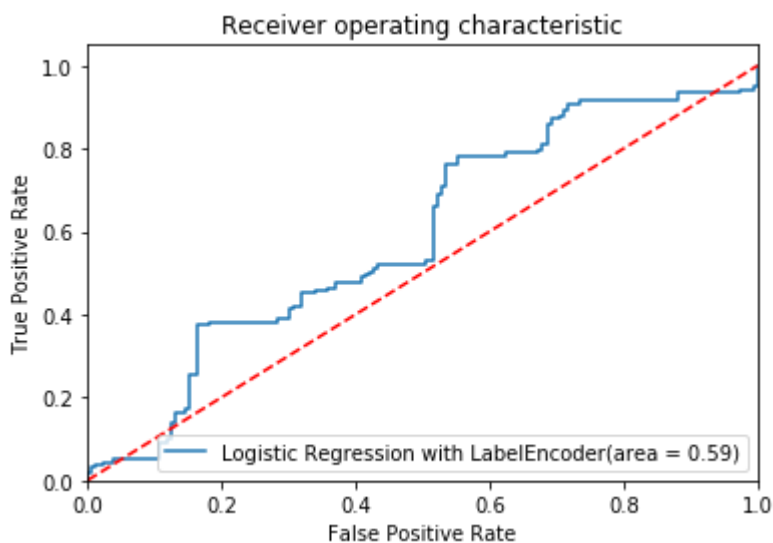
```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
    intercept_scaling=1, max_iter=100, multi_class='warn',
    n_jobs=None, penalty='l2', random_state=None, solver='warn',
    tol=0.0001, verbose=0, warm_start=False)
```

In [27]:

```
1 # logit_roc_auc = roc_auc_score(y_test, model_norm.predict(X_test))
2 # fpr, tpr, thresholds = roc_curve(y_test, model_norm.predict_proba(X_test)[:,-1])
3
4 classes = model_mix.predict(X_test)
5 probs = model_mix.predict_proba(X_test)
6 preds = probs[:,1]
7 #preds
```

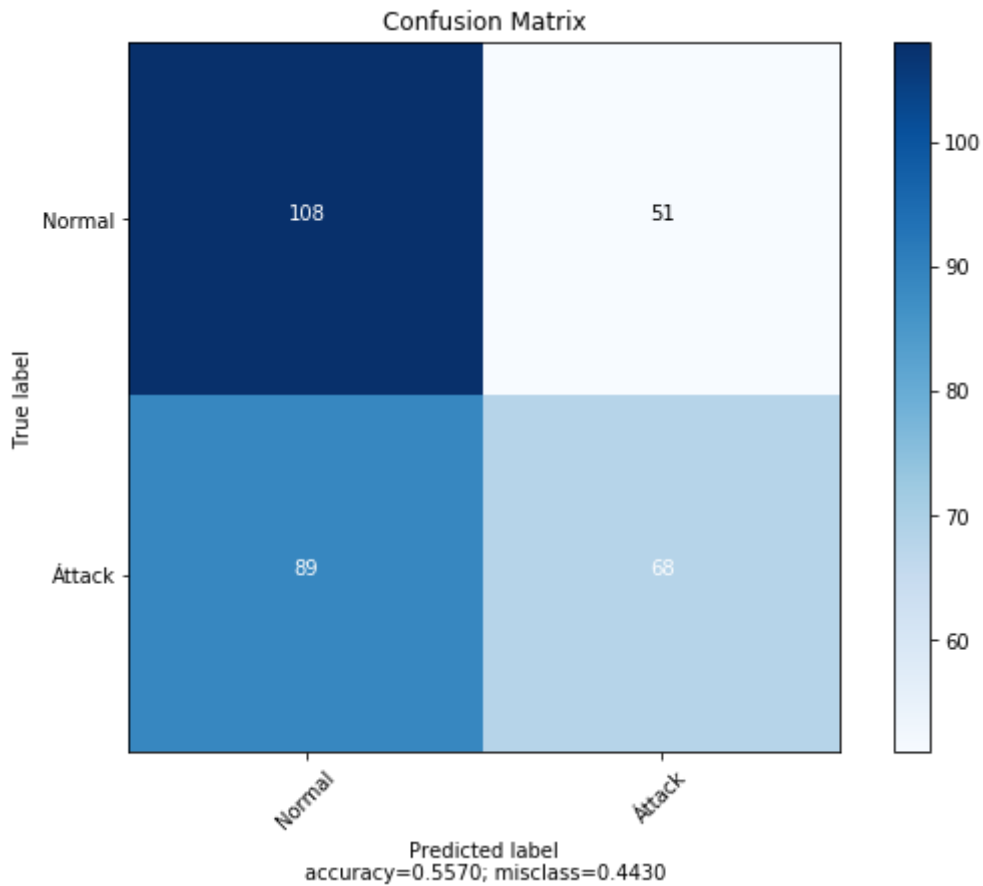
In [28]:

```
1 labelfpr, labeltpr, labelthreshold = metrics.roc_curve(y_test, preds)
2 label_roc_auc = metrics.auc(labelfpr, labeltpr)
3
4 plt.figure()
5 plt.plot(labelfpr, labeltpr, label='Logistic Regression with LabelEncoder(area = %0.2f'
6 plt.plot([0, 1], [0, 1], 'r--')
7 plt.xlim([0.0, 1.0])
8 plt.ylim([0.0, 1.05])
9 plt.xlabel('False Positive Rate')
10 plt.ylabel('True Positive Rate')
11 plt.title('Receiver operating characteristic')
12 plt.legend(loc="lower right")
13 plt.savefig('Log_ROC')
14 plt.show()
```



In [29]:

```
1 # Making the Confusion Matrix
2 from sklearn.metrics import confusion_matrix
3 cm = confusion_matrix(y_test, classes)
4 plot_confusion_matrix(cm,
5                       normalize = False,
6                       target_names = ['Normal', 'Attack'],
7                       title = "Confusion Matrix")
```



In [30]:

```
1 X_train.shape, X_test.shape, y_train.shape, y_test.shape
```

Out[30]:

```
((1263, 1), (316, 1), (1263,), (316,))
```

In [31]:

```
1 print(metrics.classification_report(classes, y_test))
```

	precision	recall	f1-score	support
0	0.68	0.55	0.61	197
1	0.43	0.57	0.49	119
micro avg	0.56	0.56	0.56	316
macro avg	0.56	0.56	0.55	316
weighted avg	0.59	0.56	0.56	316

RandomForest Classification

In [32]:

```
1 # Normalize using MinMaxScaler to constrain values to between 0 and 1.
2 from sklearn.preprocessing import MinMaxScaler, StandardScaler
3
4 scaler = MinMaxScaler(feature_range = (0,1))
5
6 scaler.fit(X_train)
7 X_train = scaler.transform(X_train)
8 X_test = scaler.transform(X_test)
```

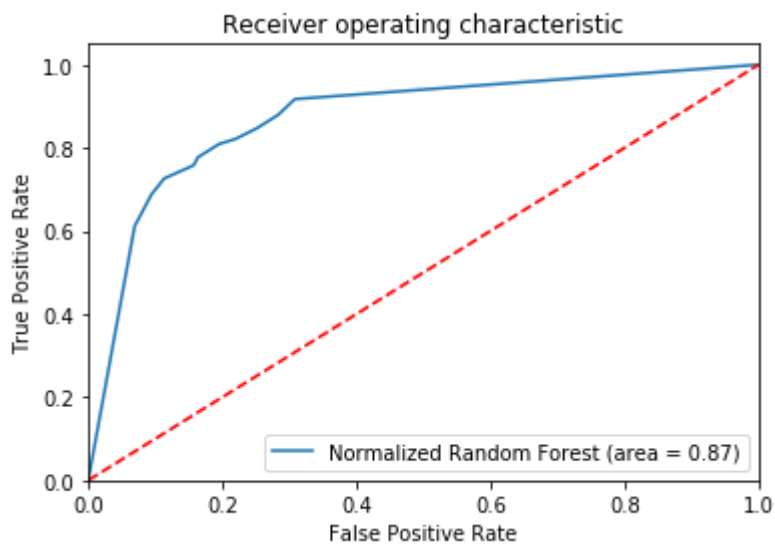
In [33]:

```
1 clf = RandomForestClassifier(n_jobs=-1)
2 model_rf = clf.fit(X_train, y_train)
3 print('Model accuracy is', model_rf.score(X_test, y_test))
```

Model accuracy is 0.8069620253164557

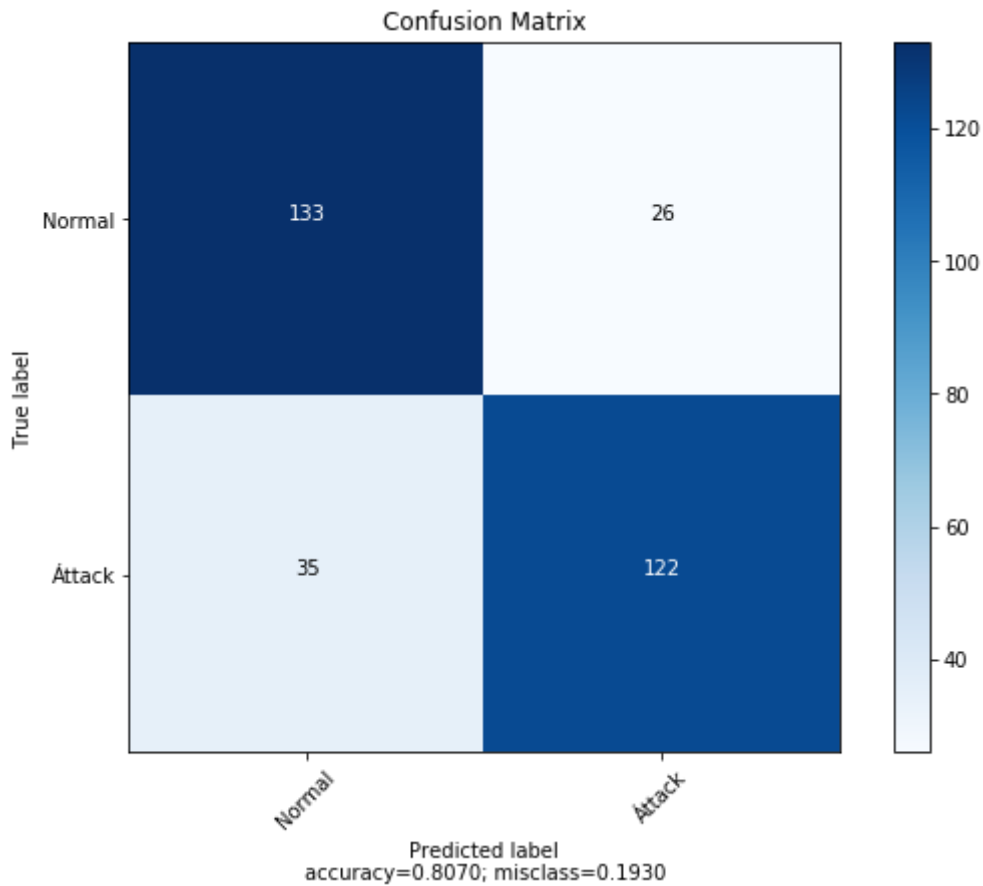
In [34]:

```
1 probs = model_rf.predict_proba(X_test)
2 preds = probs[:,1]
3 rffpr, rftpr, rfthreshold = metrics.roc_curve(y_test, preds)
4 rf_roc_auc = metrics.auc(rffpr, rftpr)
5
6 plt.figure()
7 plt.plot(rffpr, rftpr, label='Normalized Random Forest (area = %0.2f)' % rf_roc_auc)
8 plt.plot([0, 1], [0, 1], 'r--')
9 plt.xlim([0.0, 1.0])
10 plt.ylim([0.0, 1.05])
11 plt.xlabel('False Positive Rate')
12 plt.ylabel('True Positive Rate')
13 plt.title('Receiver operating characteristic')
14 plt.legend(loc="lower right")
15 plt.savefig('Log_ROC')
16 plt.show()
```



In [35]:

```
1 # Making the Confusion Matrix
2 from sklearn.metrics import confusion_matrix
3 classes = model_rf.predict(X_test)
4 cm = confusion_matrix(y_test, classes)
5 plot_confusion_matrix(cm,
6                       normalize = False,
7                       target_names = ['Normal', 'Attack'],
8                       title = "Confusion Matrix")
```



In [36]:

```
1 print(metrics.classification_report(classes, y_test))
```

	precision	recall	f1-score	support
0	0.84	0.79	0.81	168
1	0.78	0.82	0.80	148
micro avg	0.81	0.81	0.81	316
macro avg	0.81	0.81	0.81	316
weighted avg	0.81	0.81	0.81	316

Train Data with ngrams

In [37]:

```
1 from sklearn.datasets import make_classification
2 from sklearn.model_selection import train_test_split
3 from sklearn.linear_model import LogisticRegression
4 from sklearn.metrics import accuracy_score, confusion_matrix, recall_score, roc_auc_score
5
6 X, y = make_classification(
7     n_classes=2, class_sep=1.5, weights=[0.1, 0.9],
8     n_features=20, n_samples=1000, random_state=10
9 )
10
11 #X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=10)
12
13 clf = LogisticRegression(class_weight="balanced")
14 clf.fit(X_train, y_train)
15 THRESHOLD = 0.5
16 preds = np.where(clf.predict_proba(X_test)[:,:1] > THRESHOLD, 1, 0)
17
18 pd.DataFrame(data=[accuracy_score(y_test, preds), recall_score(y_test, preds),
19                     precision_score(y_test, preds), roc_auc_score(y_test, preds)],
20             index=["accuracy", "recall", "precision", "roc_auc_score"])
```

Out[37]:

	0
accuracy	0.531646
recall	0.579618
precision	0.526012
roc_auc_score	0.531947

In [38]:

```

1  from sklearn import model_selection, preprocessing, linear_model, naive_bayes, metrics
2  from sklearn.feature_extraction.text import TfidfVectorizer, CountVectorizer
3  from sklearn import decomposition, ensemble
4
5  import pandas, xgboost, numpy, textblob, string
6  from keras.preprocessing import text, sequence
7  from keras import layers, models, optimizers
8
9  def train_model(classifier, feature_vector_train, label, feature_vector_valid, is_neur:
10     # fit the training dataset on the classifier
11     classifier.fit(feature_vector_train, label)
12
13     # predict the labels on validation dataset
14     predictions = classifier.predict(feature_vector_valid)
15
16     if is_neural_net:
17         predictions = predictions.argmax(axis=-1)
18
19     return metrics.accuracy_score(predictions, valid_y)
20
21 # Load the dataset
22 #data = open('data/corpus').read()
23 #labels, texts = [], []
24 #for i, line in enumerate(data.split("\n")):
25 #    content = line.split()
26 #    labels.append(content[0])
27 #    texts.append(" ".join(content[1:]))
28
29 # create a dataframe using texts and labels
30 #trainDF = pandas.DataFrame()
31 #trainDF['text'] = texts
32 #trainDF['label'] = labels

```

In [39]:

```
1 X_df.head()
```

Out[39]:

CallTrace

```

0    6 6 63 6 42 120 6 195 120 6 6 114 114 1 1 252 ...
1    54 175 120 175 175 3 175 175 120 175 120 175 1...
2     6 11 45 33 192 33 5 197 192 6 33 5 3 197 192 1...
3     7 174 174 5 197 197 6 13 195 4 4 118 6 91 38 5...
4    11 45 33 192 33 5 197 192 6 33 5 3 197 192 192...

```

In [40]:

```

1  # create a dataframe using texts and lables
2  trainDF = train_df.copy()
3
4  trainDF['CallTrace_T'] = trainDF.CallTrace.str.split(' ').str.join(',').astype(str)
5  #X_df = trainDF.drop('Label', axis=1)
6  X_df = trainDF.drop(['Label', 'CallTrace'], axis=1)
7  y = trainDF['Label']
8
9  # split the dataset into training and validation datasets
10 train_x, valid_x, train_y, valid_y = model_selection.train_test_split(X_df, y)
11
12 # label encode the target variable
13 encoder = preprocessing.LabelEncoder()
14 train_y = encoder.fit_transform(train_y)
15 valid_y = encoder.fit_transform(valid_y)
16
17 X_df.head()
18 #list(encoder.classes_)
19 #le_name_mapping = dict(zip(encoder.classes_, encoder.transform(encoder.classes_)))
20 #print(le_name_mapping)

```

Out[40]:

	CallTrace_T
0	6,6,63,6,42,120,6,195,120,6,6,114,114,1,1,252,...
1	54,175,120,175,175,3,175,175,120,175,120,175,1...
2	6,11,45,33,192,33,5,197,192,6,33,5,3,197,192,1...
3	7,174,174,5,197,197,6,13,195,4,4,118,6,91,38,5...
4	11,45,33,192,33,5,197,192,6,33,5,3,197,192,192...

In [41]:

```
1 train_x.shape, valid_x.shape, train_y.shape, valid_y.shape
```

Out[41]:

```
((1184, 1), (395, 1), (1184,), (395,))
```

In [42]:

```
1 trainDF.head()
```

Out[42]:

	Label	CallTrace	CallTrace_T
0	Normal	6 6 63 6 42 120 6 195 120 6 6 114 114 1 1 252 ...	6,6,63,6,42,120,6,195,120,6,6,114,114,1,1,252,...
1	Normal	54 175 120 175 175 3 175 175 120 175 120 175 1...	54,175,120,175,175,3,175,175,120,175,120,175,1...
2	Normal	6 11 45 33 192 33 5 197 192 6 33 5 3 197 192 1...	6,11,45,33,192,33,5,197,192,6,33,5,3,197,192,1...
3	Normal	7 174 174 5 197 197 6 13 195 4 4 118 6 91 38 5...	7,174,174,5,197,197,6,13,195,4,4,118,6,91,38,5...
4	Normal	11 45 33 192 33 5 197 192 6 33 5 3 197 192 192...	11,45,33,192,33,5,197,192,6,33,5,3,197,192,192...

Feature Engineering - 1n, 2n, 3n-grams

In [43]:

```
1 trainDF.head()
```

Out[43]:

	Label	CallTrace	CallTrace_T
0	Normal	6 6 63 6 42 120 6 195 120 6 6 114 114 1 1 252 ...	6,6,63,6,42,120,6,195,120,6,6,114,114,1,1,252,...
1	Normal	54 175 120 175 175 3 175 175 120 175 120 175 1...	54,175,120,175,175,3,175,175,120,175,120,175,1...
2	Normal	6 11 45 33 192 33 5 197 192 6 33 5 3 197 192 1...	6,11,45,33,192,33,5,197,192,6,33,5,3,197,192,1...
3	Normal	7 174 174 5 197 197 6 13 195 4 4 118 6 91 38 5...	7,174,174,5,197,197,6,13,195,4,4,118,6,91,38,5...
4	Normal	11 45 33 192 33 5 197 192 6 33 5 3 197 192 192...	11,45,33,192,33,5,197,192,6,33,5,3,197,192,192...

In [44]:

```

1 train_1n = pd.read_csv("./train_1n.csv")
2 train_1n.columns
3 train_1n_bkp = train_1n.copy()
4 train_1n.head()

```

Out[44]:

	Label	168	265	3	54	162	142	309	146	114	175	43	104	5	78	102	13	6	2
0	Adduser	193	75	0	0	0	0	0	0	0	0	0	0	0	0	10	0	0	
1	Adduser	0	110	139	0	0	286	0	55	0	64	0	50	0	0	0	0	0	
2	Adduser	249	133	112	0	0	0	0	0	0	0	60	0	0	0	0	0	0	
3	Adduser	0	1	51	809	0	0	202	0	0	0	0	0	0	0	0	0	0	
4	Adduser	426	234	157	0	0	0	0	0	0	0	0	0	0	2	0	0	0	

In [45]:

```
1 train_1n.columns
```

Out[45]:

```

Index(['Label', '168', '265', '3', '54', '162', '142', '309', '146', '114',
      '175', '43', '104', '5', '78', '102', '13', '6', '240', '4', '192',
      '195', '91', '85', '125', '197', '140', '19', '174', '301', '221', '3
3',
      '180', '45', '196', '120', '7', '220', '42', '63', '11', '1', '252',
      '201', '243', '199', '308', '122', '118', '219'],
      dtype='object')

```

Modelling Logistic Regression - 1n-grams

In [46]:

```

1 import warnings
2 warnings.filterwarnings("ignore")
3
4 # split the dataset in train and test
5
6 #y = train_1n.iloc[:, 0].values
7 #train_1n_no_y = train_1n.drop('Label', axis=1)
8 #X = train_1n_no_y.iloc[:, :].values
9 y = train_1n.iloc[:, 0]
10 train_1n_no_y = train_1n.drop('Label', axis=1)
11 X = train_1n_no_y.iloc[:, :]
12
13
14 # Splitting the dataset into the Training set and Test set
15 from sklearn.model_selection import train_test_split
16 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state=42)
17

```

In [47]:

```
1 X_test_bkp = X_test
```

In [48]:

```
1 X_train.shape, X_test.shape, y_train.shape, y_test.shape, type(X), type(y)
```

Out[48]:

```
((1070, 49),  
(268, 49),  
(1070,),  
(268,),  
pandas.core.frame.DataFrame,  
pandas.core.series.Series)
```

In [49]:

```
1  
2 # Feature Scaling  
3 from sklearn.preprocessing import StandardScaler  
4 sc = StandardScaler()  
5 X_train = sc.fit_transform(X_train)  
6 X_test = sc.transform(X_test)  
7  
8 # Fitting Logistic Regression to the Training set  
9 from sklearn.linear_model import LogisticRegression  
10 classifier = LogisticRegression(multi_class='ovr', solver = 'lbfgs')  
11 classifier.fit(X_train, y_train)  
12  
13 # Predicting the Test set results  
14 y_pred = classifier.predict(X_test)  
15  
16 # How did our model perform?  
17 from sklearn import metrics  
18 count_misclassified = (y_test != y_pred).sum()  
19 print('Misclassified samples: {}'.format(count_misclassified))  
20 accuracy = metrics.accuracy_score(y_test, y_pred)  
21 print('Accuracy: {:.2f}'.format(accuracy))  
22
```

Misclassified samples: 72

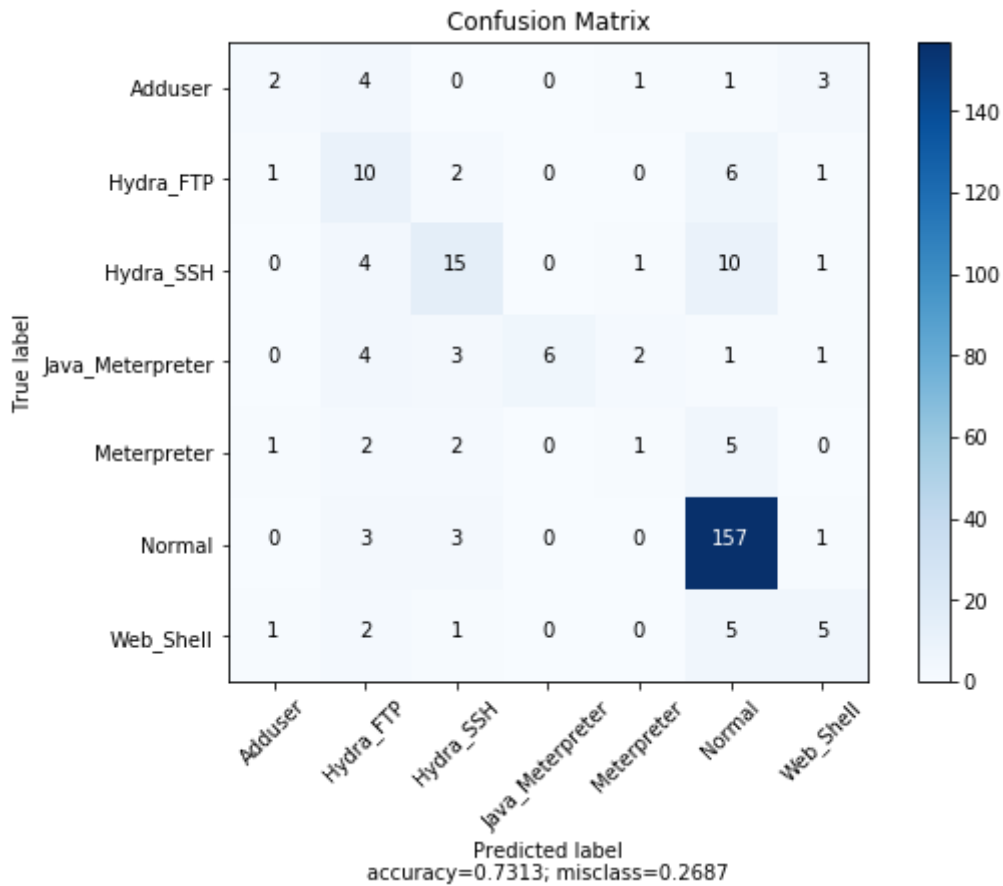
Accuracy: 0.73

In [50]:

```

1 #classifier.predict_proba(X_test)
2
3 # Making the Confusion Matrix
4 from sklearn.metrics import confusion_matrix
5 cm = confusion_matrix(y_test, y_pred)
6 plot_confusion_matrix(cm,
7                        normalize      = False,
8                        target_names  = ['Adduser', 'Hydra_FTP', 'Hydra_SSH', 'Java_Meterpreter',
9                        title         = "Confusion Matrix")

```



In [51]:

```
1 y_pred.shape, y_test.shape, type(y_test)
```

Out[51]:

```
((268,), (268,), pandas.core.series.Series)
```

In [52]:

```
1 # Merge predicted results into original dataframe
2 # y_test['preds'] = y_pred
3 # df_out = pd.merge(train_1n, y_test[['preds']], how = 'left', right_index = True)
```

In [53]:

```
1 train_1n.index
```

Out[53]:

```
RangeIndex(start=0, stop=1338, step=1)
```

In [54]:

```
1 train_2n = pd.read_csv("./train_2n.csv")
2 train_2n.columns
3 train_2n_bkp = train_2n.copy()
4 train_2n.head()
```

Out[54]:

	Label	168 168	54 54	168 265	162 162	265 168	3 168	168 3	265 265	3 3	265 3	3 265	54 309	309 54	114 162	162 114	142 142
0	Adduser	138	0	48	0	47	0	0	24	0	0	0	0	0	0	0	0
1	Adduser	0	0	0	0	0	0	0	24	45	17	20	0	0	0	0	126
2	Adduser	110	0	60	0	55	48	52	28	16	31	32	0	0	0	0	0
3	Adduser	0	594	0	0	0	0	0	0	1	0	0	172	165	0	0	0
4	Adduser	236	0	117	0	119	69	71	69	38	46	48	0	0	0	0	0

In [55]:

```

1 train_3n = pd.read_csv("./train_3n.csv")
2 train_3n.columns
3 train_3n_bkp = train_3n.copy()
4 train_3n.head()

```

Out[55]:

		168	54	162	168	265	168	168	168	3	54	54	265	309	168	265	162
	Label	168	54	162	265	168	168	3	168	168	309	54	168	54	265	265	114
		168	54	162	168	168	265	168	3	168	54	309	265	54	265	168	162
0	Adduser	101	0	0	31	34	31	0	0	0	0	0	12	0	14	14	0
1	Adduser	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	Adduser	49	0	0	25	26	25	22	23	21	0	0	12	0	11	14	0
3	Adduser	0	431	0	0	0	0	0	0	0	137	128	0	124	0	0	0
4	Adduser	132	0	0	63	68	60	33	42	36	0	0	32	0	32	31	0

In [56]:

```

1 train_1n.shape, train_2n.shape, train_3n.shape

```

Out[56]:

```
((1338, 50), (1338, 800), (1338, 4148))
```

In [70]:

```

1 train_1n_30 = train_1n.iloc[:, 0:30]
2 train_2n_30 = train_2n.iloc[:, 0:30]
3 train_3n_30 = train_3n.iloc[:, 0:30]

```

Modelling Logistic Regression/SVM/RandomForrest - 1n-grams + 2n-grams + 3n-grams

In [71]:

```

1 frames=[train_1n_30, train_2n_30, train_3n_30]
2 result=pd.concat(frames, axis=1)
3 result.shape

```

Out[71]:

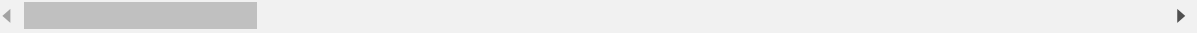
```
(1338, 90)
```

In [72]:

```
1 result.head()
```

Out[72]:

	Label	168	265	3	54	162	142	309	146	114	175	43	104	5	78	102	13	6	2
0	Adduser	193	75	0	0	0	0	0	0	0	0	0	0	0	0	10	0	0	
1	Adduser	0	110	139	0	0	286	0	55	0	64	0	50	0	0	0	0	0	
2	Adduser	249	133	112	0	0	0	0	0	0	0	60	0	0	0	0	0	0	
3	Adduser	0	1	51	809	0	0	202	0	0	0	0	0	0	0	0	0	0	
4	Adduser	426	234	157	0	0	0	0	0	0	0	0	0	0	2	0	0	0	



In [73]:

1 result.info()

```

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1338 entries, 0 to 1337
Data columns (total 90 columns):
Label                1338 non-null object
168                  1338 non-null int64
265                  1338 non-null int64
3                    1338 non-null int64
54                   1338 non-null int64
162                  1338 non-null int64
142                  1338 non-null int64
309                  1338 non-null int64
146                  1338 non-null int64
114                  1338 non-null int64
175                  1338 non-null int64
43                   1338 non-null int64
104                  1338 non-null int64
5                    1338 non-null int64
78                   1338 non-null int64
102                  1338 non-null int64
13                   1338 non-null int64
6                    1338 non-null int64
240                  1338 non-null int64
4                    1338 non-null int64
192                  1338 non-null int64
195                  1338 non-null int64
91                   1338 non-null int64
85                   1338 non-null int64
125                  1338 non-null int64
197                  1338 non-null int64
140                  1338 non-null int64
19                   1338 non-null int64
174                  1338 non-null int64
301                  1338 non-null int64
Label                1338 non-null object
168 168              1338 non-null int64
54 54                1338 non-null int64
168 265              1338 non-null int64
162 162              1338 non-null int64
265 168              1338 non-null int64
3 168                1338 non-null int64
168 3                1338 non-null int64
265 265              1338 non-null int64
3 3                  1338 non-null int64
265 3                1338 non-null int64
3 265                1338 non-null int64
54 309               1338 non-null int64
309 54               1338 non-null int64
114 162              1338 non-null int64
162 114              1338 non-null int64
142 142              1338 non-null int64
142 3                1338 non-null int64
3 142                1338 non-null int64
142 265              1338 non-null int64
265 142              1338 non-null int64
3 54                 1338 non-null int64
174 174              1338 non-null int64
309 309              1338 non-null int64

```

```

43 168      1338 non-null int64
168 146      1338 non-null int64
142 146      1338 non-null int64
146 3        1338 non-null int64
146 142      1338 non-null int64
175 175      1338 non-null int64
Label      1338 non-null object
168 168 168  1338 non-null int64
54 54 54     1338 non-null int64
162 162 162  1338 non-null int64
168 265 168  1338 non-null int64
265 168 168  1338 non-null int64
168 168 265  1338 non-null int64
168 3 168    1338 non-null int64
168 168 3    1338 non-null int64
3 168 168    1338 non-null int64
54 309 54    1338 non-null int64
54 54 309    1338 non-null int64
265 168 265  1338 non-null int64
309 54 54    1338 non-null int64
168 265 265  1338 non-null int64
265 265 168  1338 non-null int64
162 114 162  1338 non-null int64
114 162 162  1338 non-null int64
162 162 114  1338 non-null int64
3 168 265    1338 non-null int64
168 265 3    1338 non-null int64
265 3 168    1338 non-null int64
3 265 168    1338 non-null int64
265 168 3    1338 non-null int64
168 3 265    1338 non-null int64
265 265 265  1338 non-null int64
3 168 3      1338 non-null int64
3 3 168      1338 non-null int64
168 3 3      1338 non-null int64
3 3 3        1338 non-null int64
dtypes: int64(87), object(3)
memory usage: 940.9+ KB

```

In [74]:

```

1 import warnings
2 warnings.filterwarnings("ignore")
3
4 # split the dataset in train and test
5 result = result.loc[:,~result.columns.duplicated()]
6
7 y = result.iloc[:, 0].values
8 result_no_y = result.drop('Label', axis=1)
9 X = result_no_y.iloc[:, :].values

```

In [75]:

```

1 #result

```

In [76]:

```

1 # Splitting the dataset into the Training set and Test set
2 from sklearn.model_selection import train_test_split
3 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state=0)
4 X_train.shape, X_test.shape, y_train.shape, y_test.shape, type(X), type(y)

```

Out[76]:

```
((1070, 87), (268, 87), (1070,), (268,), numpy.ndarray, numpy.ndarray)
```

In [77]:

```

1 # Feature Scaling
2 from sklearn.preprocessing import StandardScaler
3 from sklearn.svm import SVC
4
5 sc = StandardScaler()
6 X_train = sc.fit_transform(X_train)
7 X_test = sc.transform(X_test)
8
9 # Fitting Logistic Regression to the Training set
10 from sklearn.linear_model import LogisticRegression
11 #classifier = LogisticRegression(multi_class='ovr', solver = 'lbfgs')
12 #classifier = SVC(kernel = 'linear', random_state = 0)
13 #classifier = SVC(kernel = 'rbf', random_state = 0)
14 clf = RandomForestClassifier(n_jobs=-1)
15 classifier.fit(X_train, y_train)
16
17 # Predicting the Test set results
18 y_pred = classifier.predict(X_test)
19
20 # How did our model perform?
21 from sklearn import metrics
22 count_misclassified = (y_test != y_pred).sum()
23 print('Misclassified samples: {}'.format(count_misclassified))
24 accuracy = metrics.accuracy_score(y_test, y_pred)
25 print('Accuracy: {:.2f}'.format(accuracy))
26 #y_pred

```

Misclassified samples: 73

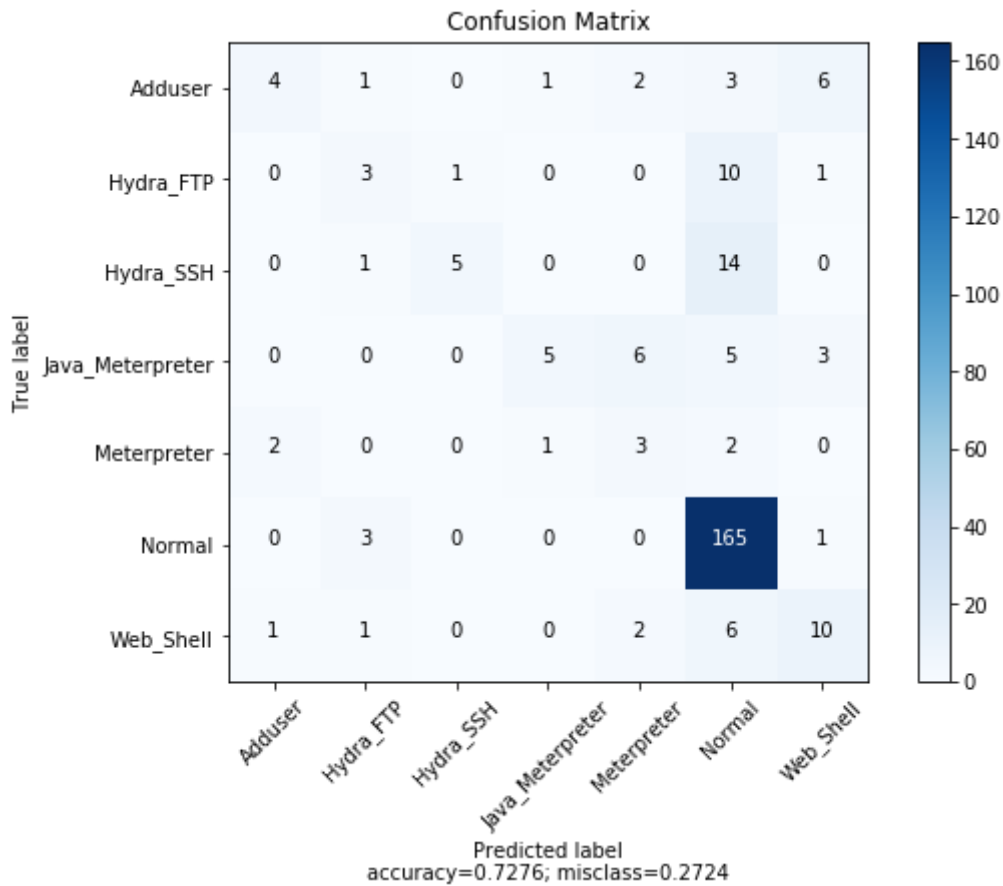
Accuracy: 0.73

In [78]:

```

1 #classifier.predict_proba(X_test)
2
3 # Making the Confusion Matrix
4 from sklearn.metrics import confusion_matrix
5 cm = confusion_matrix(y_test, y_pred)
6 plot_confusion_matrix(cm,
7                        normalize      = False,
8                        target_names  = ['Adduser', 'Hydra_FTP', 'Hydra_SSH', 'Java_Meterpreter',
9                        title         = "Confusion Matrix")

```



Applying 10-Fold cross-validation

In [79]:

```
1 from sklearn.model_selection import cross_val_score
2 import numpy as np
3
4 print(np.mean(cross_val_score(clf, X_train, y_train, cv=10)))
```

0.8121539825388545

Comparing Different Models Binary Classification - 1n-grams + 2n-grams + 3n-grams

In [80]:

```
1 # Compare Algorithms
2 # https://machinelearningmastery.com/compare-machine-learning-algorithms-python-scikit
3
4 import pandas
5 import matplotlib.pyplot as plt
6 from sklearn import model_selection
7 from sklearn.linear_model import LogisticRegression
8 from sklearn.tree import DecisionTreeClassifier
9 from sklearn.neighbors import KNeighborsClassifier
10 from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
11 from sklearn.naive_bayes import GaussianNB
12 from sklearn.svm import SVC
13
14 # Load dataset
15 Y = result.iloc[:, 0].values
16 result_no_y = result.drop('Label', axis=1)
17 X = result_no_y.iloc[:, :].values
18 #X = array[:,0:8]
19 #Y = array[:,8]
20
21 # Prepare configuration for cross validation test harness
22 seed = 7
23
24
```

In [81]:

```
1 # Prepare models
2 models = []
3 models.append(('LR', LogisticRegression()))
4 models.append(('LDA', LinearDiscriminantAnalysis()))
5 models.append(('KNN', KNeighborsClassifier()))
6 models.append(('CART', DecisionTreeClassifier()))
7 models.append(('NB', GaussianNB()))
8 models.append(('SVM', SVC()))
9 models.append(('RandomForest', RandomForestClassifier()))
10
11
```

In [82]:

```

1 # Evaluate each model in turn
2 results = []
3 names = []
4 scoring = 'accuracy'
5 for name, model in models:
6     kfold = model_selection.KFold(n_splits=10, random_state=seed)
7     cv_results = model_selection.cross_val_score(model, X, Y, cv=kfold, scoring=scoring)
8     results.append(cv_results)
9     names.append(name)
10    msg = "%s: %f (%f)" % (name, cv_results.mean(), cv_results.std())
11    print(msg)
12
13

```

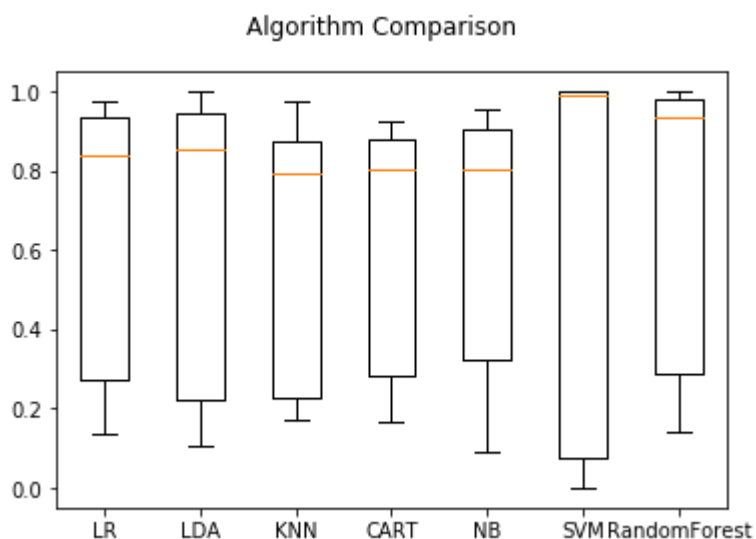
LR: 0.629806 (0.349178)
 LDA: 0.626826 (0.370299)
 KNN: 0.602828 (0.329431)
 CART: 0.616250 (0.311622)
 NB: 0.616266 (0.331033)
 SVM: 0.625373 (0.459750)
 RandomForest: 0.675362 (0.364782)

In [83]:

```

1 # Boxplot algorithm comparison
2 fig = plt.figure()
3 fig.suptitle('Algorithm Comparison')
4 ax = fig.add_subplot(111)
5 plt.boxplot(results)
6 ax.set_xticklabels(names)
7 plt.show()

```



Random Forest & Logistic Regression gave best accuracy so far

Random Forest Model Parameter Tuning

In [84]:

```
1 train_1n_30 = train_1n.iloc[:, 0:30]
2 train_2n_30 = train_2n.iloc[:, 0:30]
3 train_3n_30 = train_3n.iloc[:, 0:30]
```

In [85]:

```
1 train_1n_30.shape
```

Out[85]:

(1338, 30)

In [86]:

```
1 train_2n_30.shape
```

Out[86]:

(1338, 30)

In [87]:

```
1 train_3n_30.shape
```

Out[87]:

(1338, 30)

In [88]:

```

1 #frames = [train_1n, train_2n, train_3n]
2 frames = [train_1n_30, train_2n_30, train_3n_30]
3 result = pd.concat(frames, axis=1)
4
5 result = result.loc[:,~result.columns.duplicated()]
6
7 result.loc[result.Label != 'Normal', 'Label'] = 1
8 result.loc[result.Label == 'Normal', 'Label'] = 0
9 result.head()
10 #result['Label_Binary'].value_counts()
11
12 # Extract features and labels
13 labels = result['Label']
14 features = result.drop('Label', axis = 1)
15 #features.head(5)
16 #labels.head(5)
17
18
19 # One Hot Encoding
20 #features = pd.get_dummies(result)
21 #features.head(5)
22
23
24 #from sklearn import preprocessing
25
26 #le = preprocessing.LabelEncoder()
27 #features['Label_Normal'] = le.fit_transform(features['Label_Normal'])
28
29
30 #cols_drop = [ 'Label_Meterpreter', 'Label_Web_Shell', 'Label_Adduser',
31 #             'Label_Hydra_FTP', 'Label_Hydra_SSH', 'Label_Java_Meterpreter', 'Label_Normal'
32
33 # Extract features and labels
34 #labels = features['Label_Normal']
35 #labels['Label_Normal'].astype(object).astype(int)
36 #labels = labels.loc[:,~labels.columns.duplicated()]
37 #features = features.drop(cols_drop, axis = 1)
38 #features.head(5)
39 #labels.head(5)
40

```

In [89]:

```

1 # Convert to numpy arrays
2 import numpy as np
3
4 features = np.array(features)
5 labels = np.array(labels)
6
7 # Training and Testing Sets
8 from sklearn.model_selection import train_test_split
9
10 train_features, test_features, train_labels, test_labels = train_test_split(features,
11                                                                           test_size =

```

In [90]:

```
1 print('Training Features Shape:', train_features.shape)
2 print('Training Labels Shape:', train_labels.shape)
3 print('Testing Features Shape:', test_features.shape)
4 print('Testing Labels Shape:', test_labels.shape)
```

Training Features Shape: (1003, 87)

Training Labels Shape: (1003,)

Testing Features Shape: (335, 87)

Testing Labels Shape: (335,)

Examine the Default Random Forest to Determine Parameters

In [91]:

```
1 # Reference : https://towardsdatascience.com/hyperparameter-tuning-the-random-forest-in-
2
3 from sklearn.ensemble import RandomForestClassifier
4 from pprint import pprint
5
6 rf = RandomForestClassifier(random_state=42)
7
8 #Look at parameters used by our current forest
9 pprint(rf.get_params())
```

```
{'bootstrap': True,
 'class_weight': None,
 'criterion': 'gini',
 'max_depth': None,
 'max_features': 'auto',
 'max_leaf_nodes': 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': 'warn',
 'n_jobs': None,
 'oob_score': False,
 'random_state': 42,
 'verbose': 0,
 'warm_start': False}
```

Random Search with Cross Validation

In [92]:

```

1  from sklearn.model_selection import RandomizedSearchCV
2
3  # Number of trees in random forest
4  n_estimators = [int(x) for x in np.linspace(start = 200, stop = 2000, num = 10)]
5  # Number of features to consider at every split
6  max_features = ['auto', 'sqrt']
7  # Maximum number of levels in tree
8  max_depth = [int(x) for x in np.linspace(10, 110, num = 11)]
9  max_depth.append(None)
10 # Minimum number of samples required to split a node
11 min_samples_split = [2, 5, 10]
12 # Minimum number of samples required at each leaf node
13 min_samples_leaf = [1, 2, 4]
14 # Method of selecting samples for training each tree
15 bootstrap = [True, False]
16
17 # Create the random grid
18 random_grid = {'n_estimators': n_estimators,
19                'max_features': max_features,
20                'max_depth': max_depth,
21                'min_samples_split': min_samples_split,
22                'min_samples_leaf': min_samples_leaf,
23                'bootstrap': bootstrap}
24
25 pprint(random_grid)

```

```

{'bootstrap': [True, False],
 'max_depth': [10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, None],
 'max_features': ['auto', 'sqrt'],
 'min_samples_leaf': [1, 2, 4],
 'min_samples_split': [2, 5, 10],
 'n_estimators': [200, 400, 600, 800, 1000, 1200, 1400, 1600, 1800, 2000]}

```

In [93]:

```

1  # Use the random grid to search for best hyperparameters
2  # First create the base model to tune
3  rf = RandomForestClassifier(random_state = 42)
4  # Random search of parameters, using 3 fold cross validation,
5  # search across 100 different combinations, and use all available cores
6  rf_random = RandomizedSearchCV(estimator=rf, param_distributions=random_grid,
7                                n_iter = 100, scoring='accuracy',
8                                cv = 3, verbose=2, random_state=42, n_jobs=-1,
9                                return_train_score=True)
10
11 # Fit the random search model
12 rf_random.fit(train_features, train_labels);

```

Fitting 3 folds for each of 100 candidates, totalling 300 fits

```

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done 25 tasks      | elapsed: 10.1s
[Parallel(n_jobs=-1)]: Done 146 tasks    | elapsed: 47.8s
[Parallel(n_jobs=-1)]: Done 300 out of 300 | elapsed: 1.6min finished

```

Best Parameters Identified

In [94]:

```
1 rf_random.best_params_
```

Out[94]:

```
{'n_estimators': 400,  
 'min_samples_split': 2,  
 'min_samples_leaf': 1,  
 'max_features': 'sqrt',  
 'max_depth': None,  
 'bootstrap': False}
```

In [95]:

```
1 #rf_random.cv_results_
```

Evaluate Random Search

To determine if random search yielded a better model, we compare the base model with the best random search model.

In [96]:

```
1 def evaluate(model, test_features, test_labels):  
2     predictions = model.predict(test_features)  
3     accuracy = metrics.accuracy_score(test_labels, predictions)  
4     print('Accuracy: {:.2f}'.format(accuracy))  
5
```

Evaluate the Default Model

In [97]:

```
1 base_model = RandomForestClassifier(n_estimators = 10, random_state = 42)  
2 base_model.fit(train_features, train_labels)  
3 base_accuracy = evaluate(base_model, test_features, test_labels)  
4
```

Accuracy: 0.96

Evaluate the Best Random Search Model

In [98]:

```
1 best_random = rf_random.best_estimator_  
2 random_accuracy = evaluate(best_random, test_features, test_labels)
```

Accuracy: 0.97

Grid Search

We can now perform grid search building on the result from the random search. We will test a range of hyperparameters around the best values returned by random search.

In [99]:

```

1  from sklearn.model_selection import GridSearchCV
2
3  # Create the parameter grid based on the results of random search
4  param_grid = {
5      'bootstrap': [True],
6      'max_depth': [80, 90, 100, 110],
7      'max_features': [2, 3],
8      'min_samples_leaf': [3, 4, 5],
9      'min_samples_split': [8, 10, 12],
10     'n_estimators': [100, 200, 300, 1000]
11 }
12
13 # Create a base model
14 rf = RandomForestClassifier(random_state = 42)
15
16 # Instantiate the grid search model
17 grid_search = GridSearchCV(estimator = rf, param_grid = param_grid,
18                             cv = 3, n_jobs = -1, verbose = 2, return_train_score=True)

```

In [100]:

```

1  # Fit the grid search to the data
2  grid_search.fit(train_features, train_labels);

```

Fitting 3 folds for each of 288 candidates, totalling 864 fits

```

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done 34 tasks      | elapsed:    2.5s
[Parallel(n_jobs=-1)]: Done 177 tasks    | elapsed:   14.0s
[Parallel(n_jobs=-1)]: Done 380 tasks    | elapsed:   29.6s
[Parallel(n_jobs=-1)]: Done 663 tasks    | elapsed:   51.8s
[Parallel(n_jobs=-1)]: Done 864 out of 864 | elapsed:  1.1min finished

```

In [101]:

```
1  grid_search.best_params_
```

Out[101]:

```

{'bootstrap': True,
 'max_depth': 80,
 'max_features': 3,
 'min_samples_leaf': 3,
 'min_samples_split': 8,
 'n_estimators': 300}

```

Evaluate the Best Model from Grid Search

In [102]:

```

1  best_grid = grid_search.best_estimator_
2  grid_accuracy = evaluate(best_grid, test_features, test_labels)

```

Accuracy: 0.91

Final Model

In [103]:

```
1 final_model = grid_search.best_estimator_  
2  
3 print('Final Model Parameters:\n')  
4 pprint(final_model.get_params())  
5 print('\n')  
6 grid_final_accuracy = evaluate(final_model, test_features, test_labels)
```

Final Model Parameters:

```
{'bootstrap': True,  
 'class_weight': None,  
 'criterion': 'gini',  
 'max_depth': 80,  
 'max_features': 3,  
 'max_leaf_nodes': None,  
 'min_impurity_decrease': 0.0,  
 'min_impurity_split': None,  
 'min_samples_leaf': 3,  
 'min_samples_split': 8,  
 'min_weight_fraction_leaf': 0.0,  
 'n_estimators': 300,  
 'n_jobs': None,  
 'oob_score': False,  
 'random_state': 42,  
 'verbose': 0,  
 'warm_start': False}
```

Accuracy: 0.91

Deep Autoencoder Model

In [104]:

```
1 # https://www.kaggle.com/kredy10/simple-lstm-for-text-classification  
2 # https://www.curiously.com/posts/credit-card-fraud-detection-using-autoencoders-in-k  
3
```

In [107]:

```

1 import pandas as pd
2 import numpy as np
3 import pickle
4 import matplotlib.pyplot as plt
5 from scipy import stats
6 import tensorflow as tf
7 import seaborn as sns
8 from pylab import rcParams
9 from sklearn.model_selection import train_test_split
10 from keras.models import Model, load_model
11 from keras.layers import Input, Dense
12 from keras.callbacks import ModelCheckpoint, TensorBoard
13 from keras import regularizers
14
15 %matplotlib inline
16
17 sns.set(style='whitegrid', palette='muted', font_scale=1.5)
18
19 rcParams['figure.figsize'] = 14, 8
20
21 RANDOM_SEED = 42
22 LABELS = ["Normal", "Attack"]

```

In [109]:

```

1 #result = result.loc[:,~result.columns.duplicated()]
2
3 #result.loc[result.Label != 'Normal', 'Label'] = 1
4 #result.loc[result.Label == 'Normal', 'Label'] = 0
5 result.head()
6
7

```

Out[109]:

	Label	168	265	3	54	162	142	309	146	114	175	43	104	5	78	102	13	6	240
0	1	193	75	0	0	0	0	0	0	0	0	0	0	0	0	10	0	0	0
1	1	0	110	139	0	0	286	0	55	0	64	0	50	0	0	0	0	0	0
2	1	249	133	112	0	0	0	0	0	0	0	60	0	0	0	0	0	0	0
3	1	0	1	51	809	0	0	202	0	0	0	0	0	0	0	0	0	0	0
4	1	426	234	157	0	0	0	0	0	0	0	0	0	0	2	0	0	0	0

In [110]:

```

1 attack = result[result.Label == 1]
2 normal = result[result.Label == 0]

```

In [111]:

```
1 attack.shape
```

Out[111]:

(505, 88)

In [112]:

```
1 normal.shape
```

Out[112]:

(833, 88)

In []:

```
1
```

In [113]:

```
1 X_train, X_test = train_test_split(result, test_size=0.2, random_state=RANDOM_SEED)
2 X_train = X_train[X_train.Label == 0]
3 X_train = X_train.drop(['Label'], axis=1)
4
5 y_test = X_test['Label']
6 X_test = X_test.drop(['Label'], axis=1)
7
8 X_train = X_train.values
9 X_test = X_test.values
10
```

In [114]:

```
1 X_train.shape
```

Out[114]:

(662, 87)

Building the model Our Autoencoder uses 4 fully connected layers with 14, 7, 7 and 29 neurons respectively. The first two layers are used for our encoder, the last two go for the decoder. Additionally, L1 regularization will be used during training:

In [115]:

```
1 input_dim = X_train.shape[1]
2 encoding_dim = 341
```

In [116]:

```
1 input_layer = Input(shape=(input_dim, ))
2
3 encoder = Dense(encoding_dim, activation="tanh",
4                 activity_regularizer=regularizers.l1(10e-5))(input_layer)
5 encoder = Dense(int(encoding_dim / 2), activation="relu")(encoder)
6
7 decoder = Dense(int(encoding_dim / 2), activation='tanh')(encoder)
8 decoder = Dense(input_dim, activation='relu')(decoder)
9
10 autoencoder = Model(inputs=input_layer, outputs=decoder)
```

WARNING:tensorflow:From C:\Users\kuna\AppData\Local\Continuum\anaconda3\lib\site-packages\tensorflow\python\framework\op_def_library.py:263: colocate_with (from tensorflow.python.framework.ops) is deprecated and will be removed in a future version.

Instructions for updating:

Colocations handled automatically by placer.

In [117]:

```

1  nb_epoch = 100
2  batch_size = 32
3
4  autoencoder.compile(optimizer='adam',
5                      loss='mean_squared_error',
6                      metrics=['accuracy'])
7
8  checkpointer = ModelCheckpoint(filepath="model.h5",
9                                verbose=0,
10                               save_best_only=True)
11  tensorboard = TensorBoard(log_dir='./logs',
12                             histogram_freq=0,
13                             write_graph=True,
14                             write_images=True)
15
16  history = autoencoder.fit(X_train, X_train,
17                           epochs=nb_epoch,
18                           batch_size=batch_size,
19                           shuffle=True,
20                           validation_data=(X_test, X_test),
21                           verbose=1,
22                           callbacks=[checker, tensorboard]).history

```

```

- acc: 0.8021 - val_loss: 1171.8919 - val_acc: 0.7425
Epoch 27/100
662/662 [=====] - 0s 139us/step - loss: 1440.6810
- acc: 0.8051 - val_loss: 1167.5713 - val_acc: 0.7575
Epoch 28/100
662/662 [=====] - 0s 130us/step - loss: 1432.7027
- acc: 0.8127 - val_loss: 1162.3112 - val_acc: 0.7537
Epoch 29/100
662/662 [=====] - 0s 131us/step - loss: 1425.0881
- acc: 0.8278 - val_loss: 1155.5788 - val_acc: 0.7575
Epoch 30/100
662/662 [=====] - 0s 143us/step - loss: 1417.5828
- acc: 0.8233 - val_loss: 1150.9050 - val_acc: 0.7575
Epoch 31/100
662/662 [=====] - 0s 125us/step - loss: 1410.3577
- acc: 0.8278 - val_loss: 1144.9596 - val_acc: 0.7649
Epoch 32/100
662/662 [=====] - 0s 144us/step - loss: 1403.0831
- acc: 0.8233 - val_loss: 1139.3417 - val_acc: 0.7575
- . . . . .

```

In [118]:

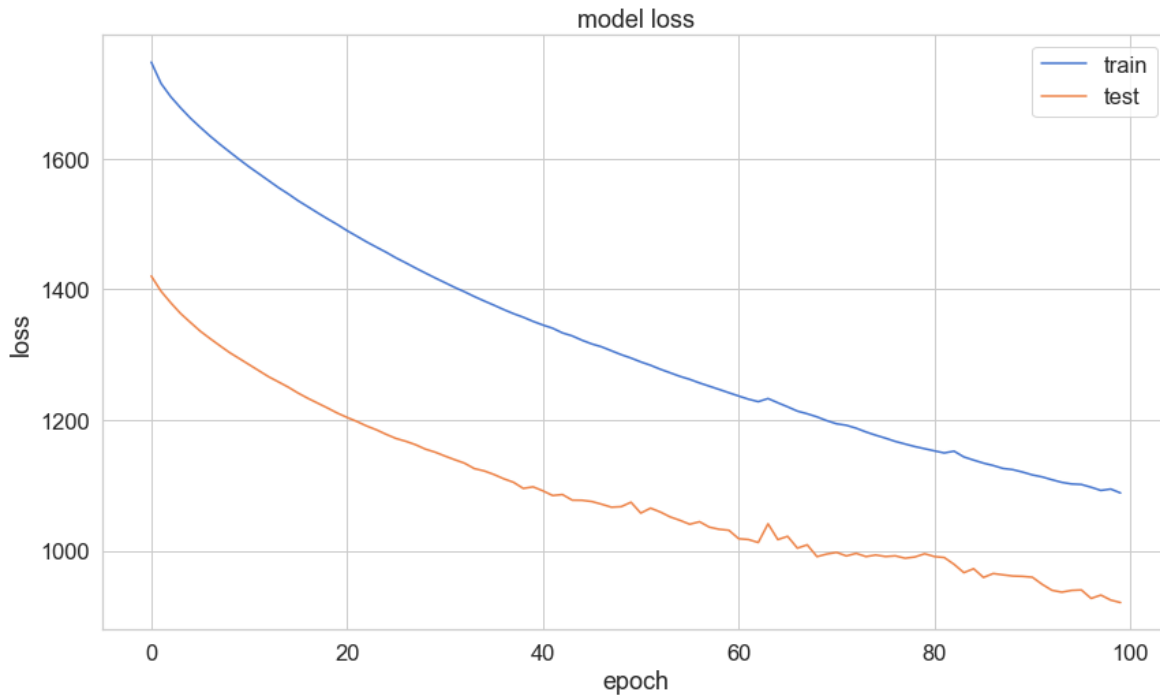
```
1 # Evaluation
```

In [119]:

```

1 plt.plot(history['loss'])
2 plt.plot(history['val_loss'])
3 plt.title('model loss')
4 plt.ylabel('loss')
5 plt.xlabel('epoch')
6 plt.legend(['train', 'test'], loc='upper right');

```



The reconstruction error on our training and test data seems to converge nicely. Is it low enough? Let's have a closer look at the error distribution:

In [120]:

```
1 predictions = autoencoder.predict(X_test)
```

In [121]:

```

1 mse = np.mean(np.power(X_test - predictions, 2), axis=1)
2 error_df = pd.DataFrame({'reconstruction_error': mse,
3                           'true_class': y_test})

```

In [122]:

```
1 error_df.describe()
```

Out[122]:

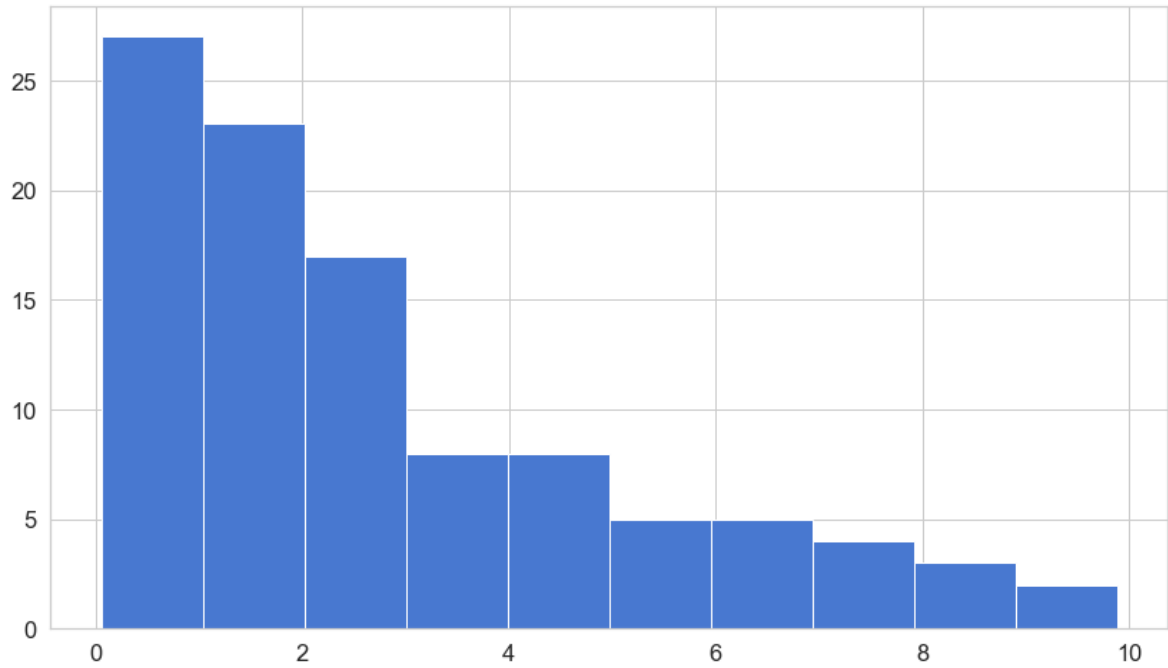
	reconstruction_error	true_class
count	268.000000	268.000000
mean	919.749266	0.361940
std	3652.005677	0.481461
min	0.049912	0.000000
25%	2.512415	0.000000
50%	15.145470	0.000000
75%	408.687194	1.000000
max	39546.021756	1.000000

In [123]:

```
1 # Reconstruction error without Attack
```

In [124]:

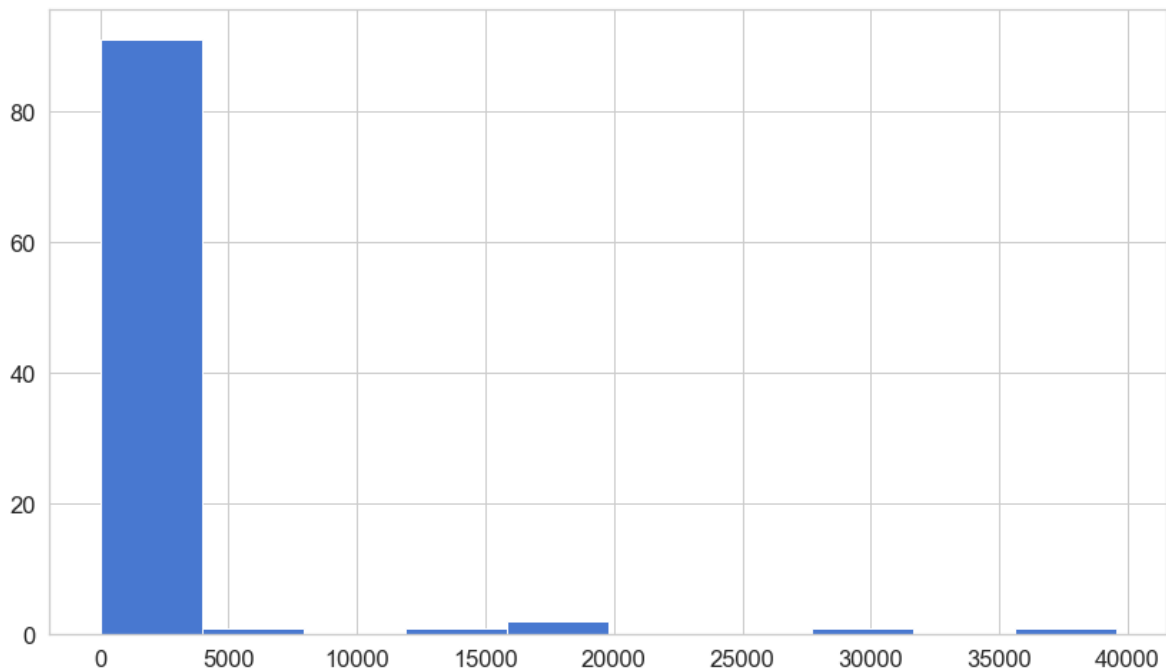
```
1 fig = plt.figure()
2 ax = fig.add_subplot(111)
3 normal_error_df = error_df[(error_df['true_class']== 0) & (error_df['reconstruction_error'] < 10)]
4 _ = ax.hist(normal_error_df.reconstruction_error.values, bins=10)
```



Reconstruction error with Attack

In [125]:

```
1 fig = plt.figure()
2 ax = fig.add_subplot(111)
3 fraud_error_df = error_df[error_df['true_class'] == 1]
4 _ = ax.hist(fraud_error_df.reconstruction_error.values, bins=10)
```



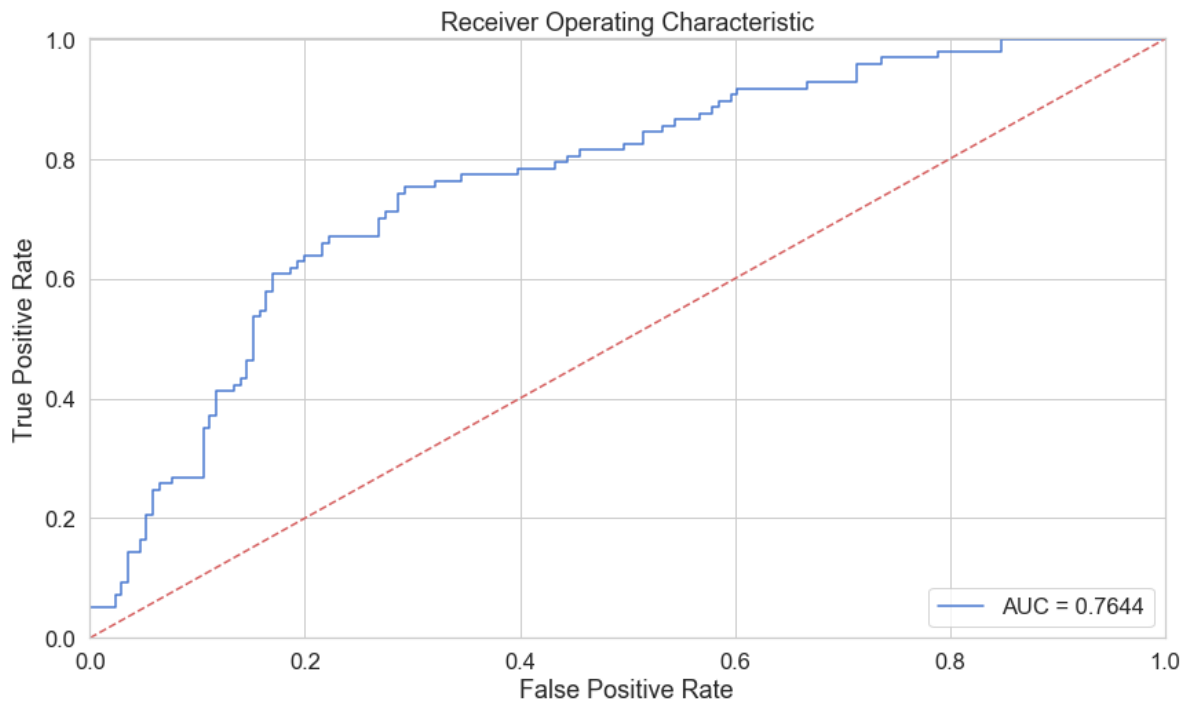
In [126]:

```
1 from sklearn.metrics import (confusion_matrix, precision_recall_curve, auc,
2                               roc_curve, recall_score, classification_report, f1_score,
3                               precision_recall_fscore_support)
```

ROC curves are very useful tool for understanding the performance of binary classifiers. However, our case is a bit out of the ordinary. We have a very imbalanced dataset. Nonetheless, let's have a look at our ROC curve:

In [127]:

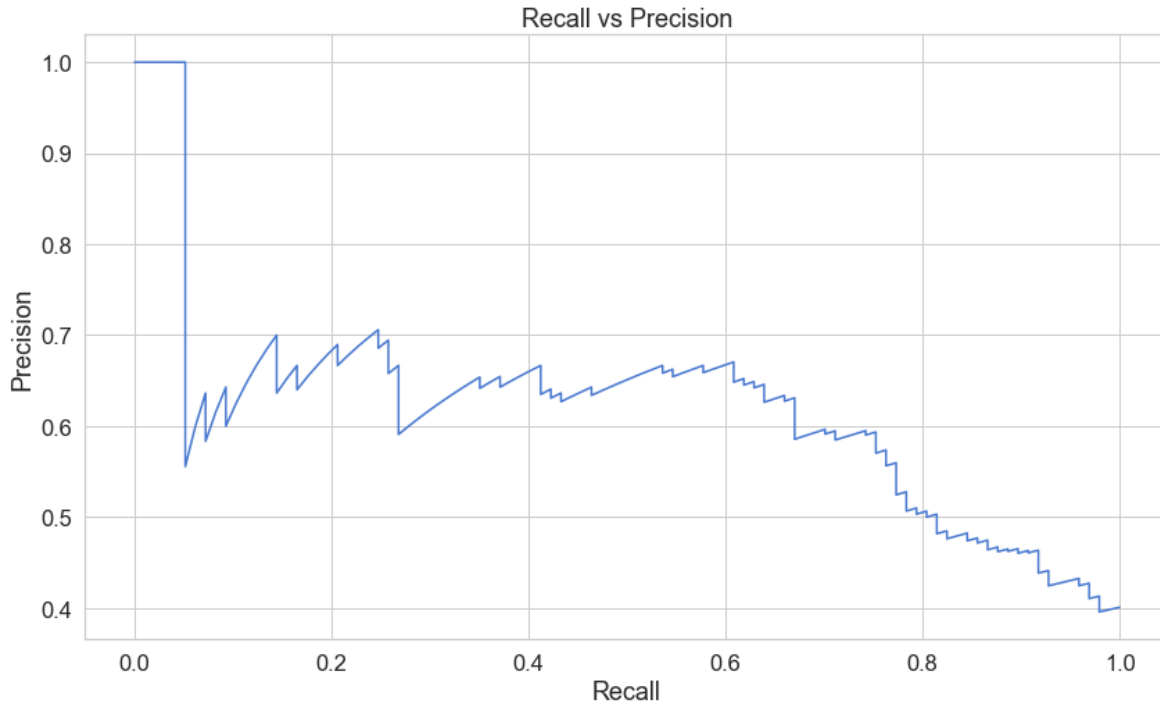
```
1 fpr, tpr, thresholds = roc_curve(error_df.true_class, error_df.reconstruction_error)
2 roc_auc = auc(fpr, tpr)
3
4 plt.title('Receiver Operating Characteristic')
5 plt.plot(fpr, tpr, label='AUC = %0.4f' % roc_auc)
6 plt.legend(loc='lower right')
7 plt.plot([0,1],[0,1], 'r--')
8 plt.xlim([-0.001, 1])
9 plt.ylim([0, 1.001])
10 plt.ylabel('True Positive Rate')
11 plt.xlabel('False Positive Rate')
12 plt.show();
```



The ROC curve plots the true positive rate versus the false positive rate, over different threshold values. Basically, we want the blue line to be as close as possible to the upper left corner. While our results look pretty good, we have to keep in mind of the nature of our dataset. ROC doesn't look very useful for us. Onward...

In [128]:

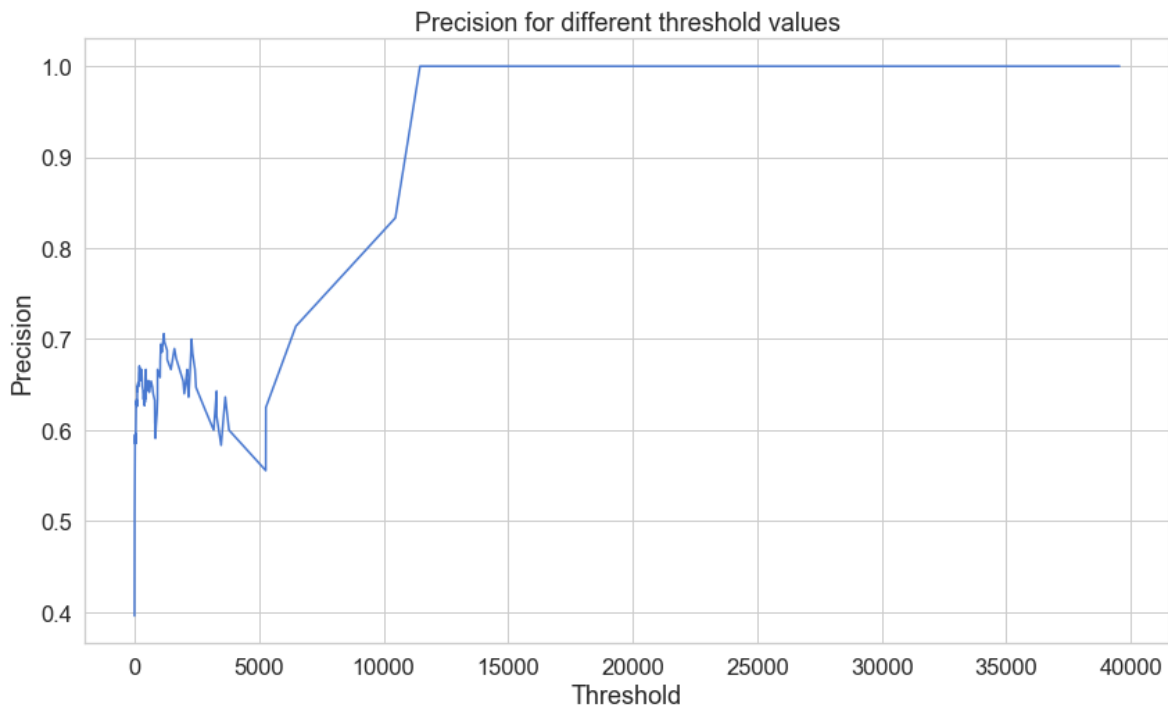
```
1 precision, recall, th = precision_recall_curve(error_df.true_class, error_df.reconstru
2 plt.plot(recall, precision, 'b', label='Precision-Recall curve')
3 plt.title('Recall vs Precision')
4 plt.xlabel('Recall')
5 plt.ylabel('Precision')
6 plt.show()
```



A high area under the curve represents both high recall and high precision, where high precision relates to a low false positive rate, and high recall relates to a low false negative rate. High scores for both show that the classifier is returning accurate results (high precision), as well as returning a majority of all positive results (high recall).

In [129]:

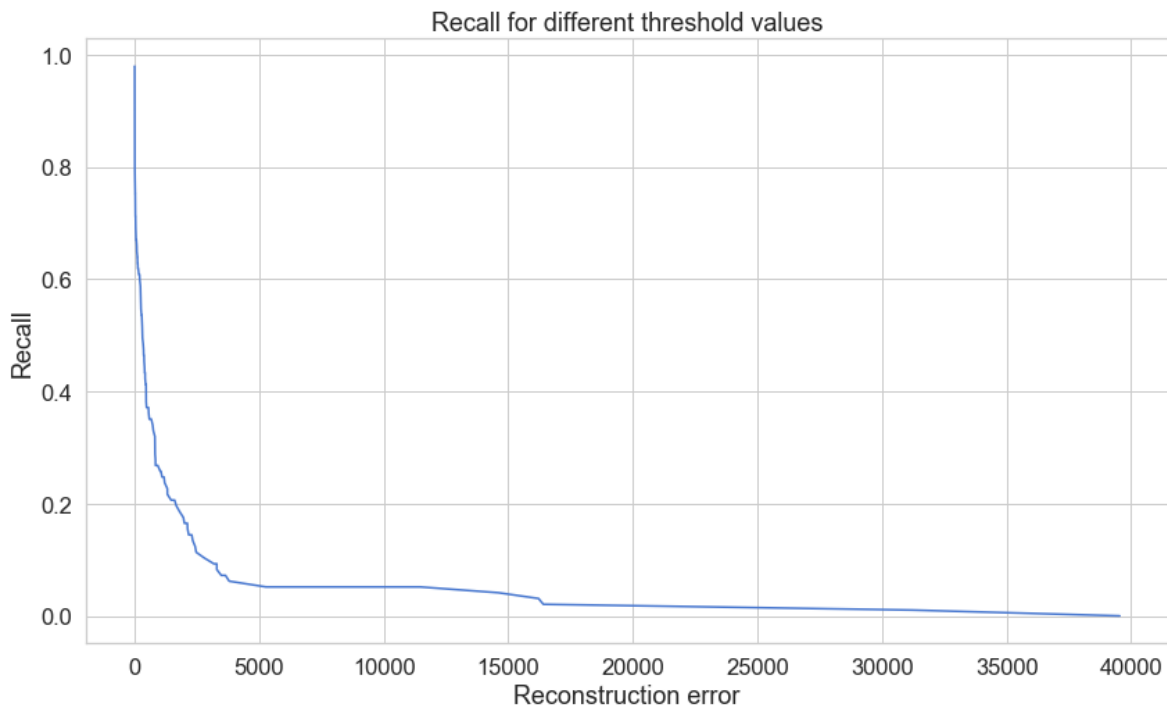
```
1 plt.plot(th, precision[1:], 'b', label='Threshold-Precision curve')
2 plt.title('Precision for different threshold values')
3 plt.xlabel('Threshold')
4 plt.ylabel('Precision')
5 plt.show()
```



You can see that as the reconstruction error increases our precision rises as well. Let's have a look at the recall:

In [130]:

```
1 plt.plot(th, recall[1:], 'b', label='Threshold-Recall curve')
2 plt.title('Recall for different threshold values')
3 plt.xlabel('Reconstruction error')
4 plt.ylabel('Recall')
5 plt.show()
```



Here, we have the exact opposite situation. As the reconstruction error increases the recall decreases.

Prediction Our model is a bit different this time. It doesn't know how to predict new values. But we don't need that. In order to predict whether or not a new/unseen system call sequence is normal or attack, we'll calculate the reconstruction error from the systemcall data itself. If the error is larger than a predefined threshold, we'll mark it as a attack (since our model should have a low error on normal transactions). Let's pick that value:

In [139]:

```
1 threshold = 20
```

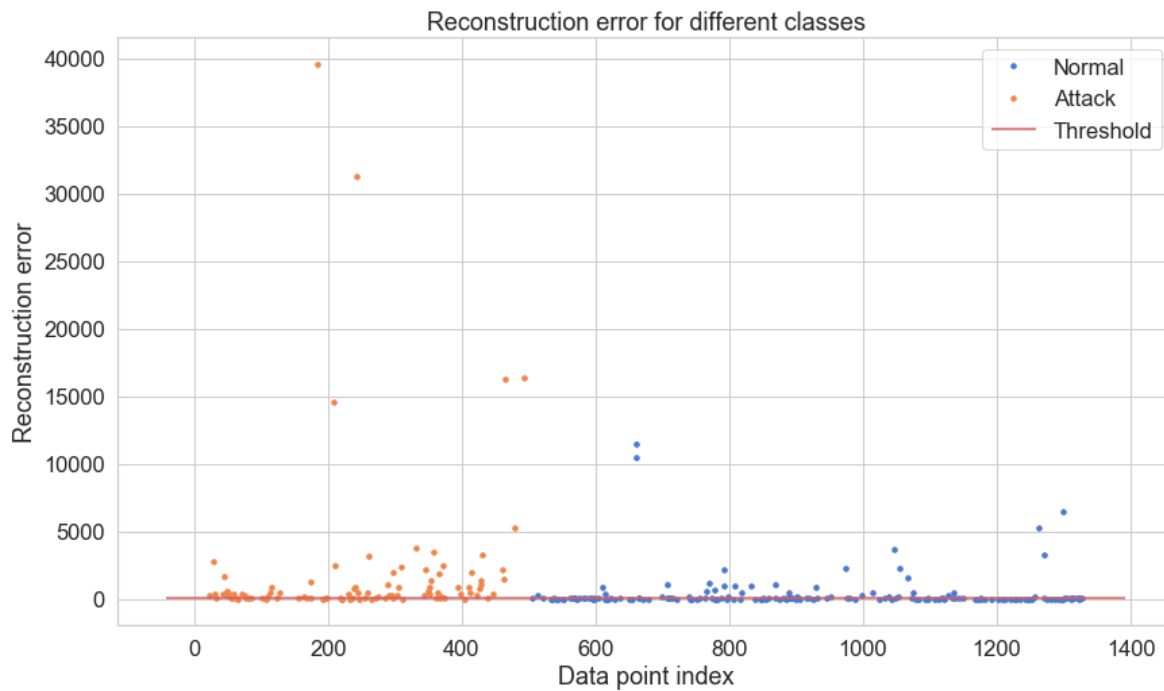
And see how well we're dividing the two types of transactions:

In [140]:

```

1 groups = error_df.groupby('true_class')
2 fig, ax = plt.subplots()
3
4 for name, group in groups:
5     ax.plot(group.index, group.reconstruction_error, marker='o', ms=3.5, linestyle='',
6             label= "Attack" if name == 1 else "Normal")
7 ax.hlines(threshold, ax.get_xlim()[0], ax.get_xlim()[1], colors="r", zorder=100, label='Threshold')
8 ax.legend()
9 plt.title("Reconstruction error for different classes")
10 plt.ylabel("Reconstruction error")
11 plt.xlabel("Data point index")
12 plt.show();

```



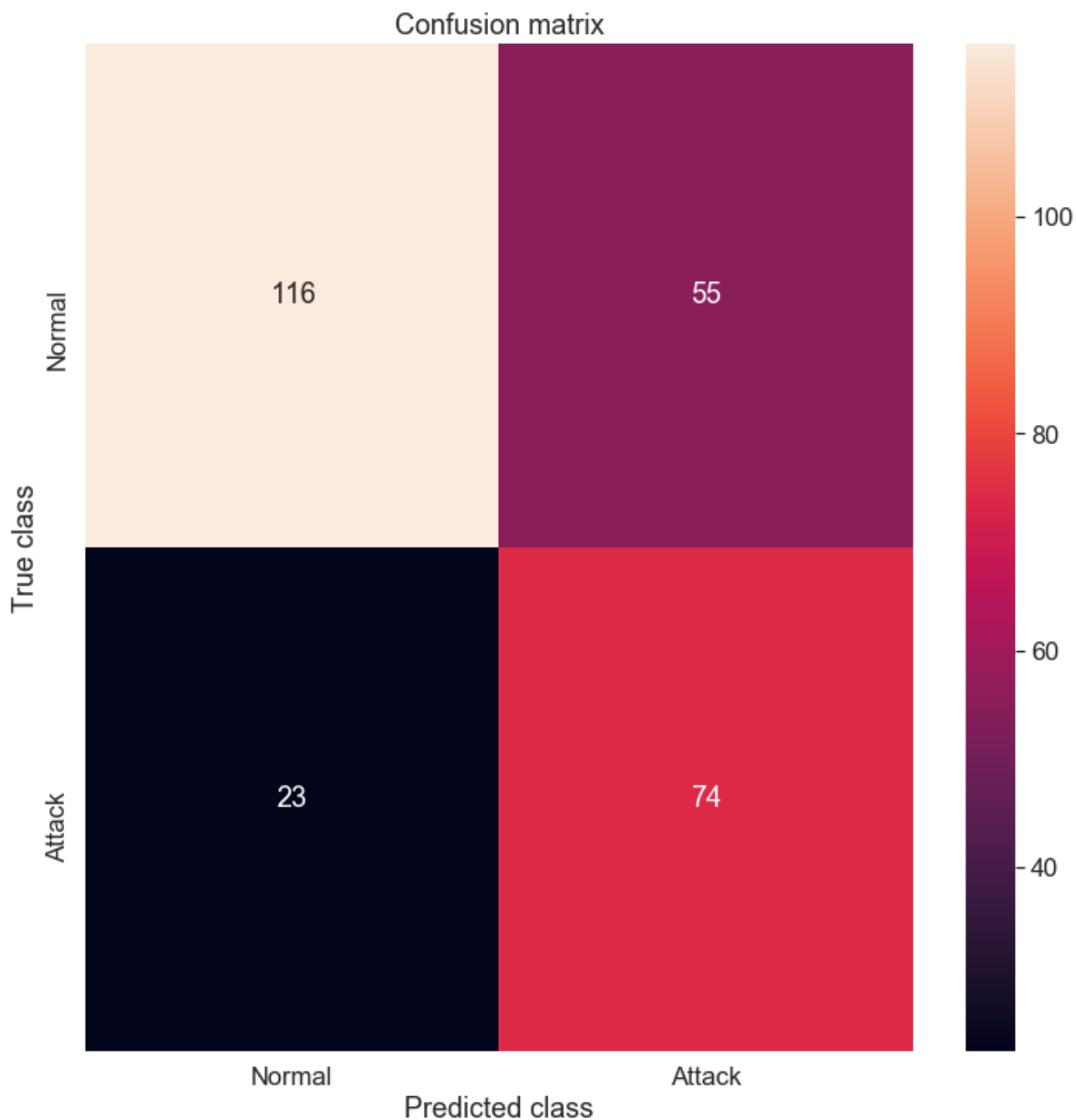
I know, that chart might be a bit deceiving. Let's have a look at the confusion matrix:

In [141]:

```

1 y_pred = [1 if e > threshold else 0 for e in error_df.reconstruction_error.values]
2 conf_matrix = confusion_matrix(error_df.true_class, y_pred)
3
4 plt.figure(figsize=(12, 12))
5 sns.heatmap(conf_matrix, xticklabels=LABELS, yticklabels=LABELS, annot=True, fmt="d");
6 plt.title("Confusion matrix")
7 plt.ylabel('True class')
8 plt.xlabel('Predicted class')
9 plt.show()

```



Our model seems to catch a lot of the attack cases. Of course, there is a catch (see what I did there?). The number of normal transactions classified as Attack is really high. Is this really a problem? Probably it is. You might want to increase or decrease the value of the threshold, depending on the problem. That one is up to you.

Conclusion

We've created a very simple Deep Autoencoder in Keras that can reconstruct what normal system call looks like. Initially, I was a bit skeptical about whether or not this whole thing is gonna work out, but it kinda did. Think about it, we gave a lot of one-class examples (normal systemcalls) to a model and it learned (somewhat) how to discriminate whether or not new examples belong to that same class. Isn't that cool? Our dataset was kind of magical, though. We really don't know what the original features look like.

Keras gave us very clean and easy to use API to build a non-trivial Deep Autoencoder. You can search for TensorFlow implementations and see for yourself how much boilerplate you need in order to train one. Can you apply a similar model to a different problem?

References

Building Autoencoders in Keras Stanford tutorial on Autoencoders Stacked Autoencoders in TensorFlow

LSTM Autoencoder Classifier Model

In [185]:

```
1 # LSTM autoencoder - https://machinelearningmastery.com/lstm-autoencoders/  
2 # LSTM Autoencoder - https://towardsdatascience.com/lstm-autoencoder-for-extreme-rare-c
```

Prepare data for LSTM models

```
1  
2  
3 LSTM is a bit more demanding than other models. Significant amount of time and  
  attention goes in preparing the data that fits an LSTM.  
4  
5 First, we will create the 3-dimensional arrays of shape: (samples x timesteps x  
  features). Samples mean the number of data points. Timesteps is the number of time  
  steps we look back at any time t to make a prediction. This is also referred to as  
  lookback period. The features is the number of features the data has, in other words,  
  the number of predictors in a multivariate data.
```

In [142]:

```
1 result.head()
```

Out[142]:

	Label	168	265	3	54	162	142	309	146	114	175	43	104	5	78	102	13	6	240
0	1	193	75	0	0	0	0	0	0	0	0	0	0	0	0	10	0	0	0
1	1	0	110	139	0	0	286	0	55	0	64	0	50	0	0	0	0	0	0
2	1	249	133	112	0	0	0	0	0	0	0	60	0	0	0	0	0	0	0
3	1	0	1	51	809	0	0	202	0	0	0	0	0	0	0	0	0	0	0
4	1	426	234	157	0	0	0	0	0	0	0	0	0	0	2	0	0	0	0

In [148]:

```
1 input_X = result.loc[:, result.columns != 'Label'].values # converts the df to a numpy array
2 input_y = result['Label'].values
3
4 n_features = input_X.shape[1] # number of features
5
```

In [144]:

```
1 def temporalize(X, y, lookback):
2     output_X = []
3     output_y = []
4     for i in range(len(X)-lookback-1):
5         t = []
6         for j in range(1,lookback+1):
7             # Gather past records upto the Lookback period
8             t.append(X[(i+j+1)], :])
9         output_X.append(t)
10        output_y.append(y[i+lookback+1])
11    return output_X, output_y
```

In LSTM, to make prediction at any time t , we will look at data from $(t-\text{lookback}):t$. In the following, we have an example to show how the input data are transformed with the temporalize function with $\text{lookback}=5$. For the modeling, we may use a longer lookback.

In [146]:

```
1 '''
2 Test: The 3D tensors (arrays) for LSTM are forming correctly.
3 '''
4 print('First instance of y = 1 in the original data')
5 display(result.iloc[(np.where(np.array(input_y) == 1)[0][0]-5):(np.where(np.array(input_y) == 1)[0][0])])
6
7 lookback = 5 # Equivalent to 10 min of past data.
8 # Temporalize the data
9 X, y = temporalize(X = input_X, y = input_y, lookback = lookback)
10
11 print('For the same instance of y = 1, we are keeping past 5 samples in the 3D predictor array, X.')
12 display(pd.DataFrame(np.concatenate(X[np.where(np.array(y) == 1)[0][0]], axis=0 )))
```

First instance of y = 1 in the original data

Label 168 265 3 54 162 142 309 146 114 175 43 104 5 78 102 13 6 240 4

For the same instance of y = 1, we are keeping past 5 samples in the 3D predictor array, X.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	249	133	112	0	0	0	0	0	0	0	60	0	0	0	0	0	0	0	0	0	4
1	0	1	51	809	0	0	202	0	0	0	0	0	0	0	0	0	0	4	0	0	0
2	426	234	157	0	0	0	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0
3	227	115	90	1	3	0	0	40	0	0	0	0	0	10	0	0	0	0	4	0	0
4	0	0	0	0	325	0	0	0	98	0	0	0	0	0	0	0	0	0	0	0	0

Divide the data into train, valid, and test

In [177]:

```
1 %matplotlib inline
2 import matplotlib.pyplot as plt
3 import seaborn as sns
4
5 import pandas as pd
6 import numpy as np
7 from pylab import rcParams
8
9 import tensorflow as tf
10 from keras import optimizers, Sequential
11 from keras.models import Model
12 from keras.utils import plot_model
13 from keras.layers import Dense, LSTM, RepeatVector, TimeDistributed
14 from keras.callbacks import ModelCheckpoint, TensorBoard
15
16 from sklearn.preprocessing import StandardScaler
17 from sklearn.model_selection import train_test_split
18 from sklearn.metrics import confusion_matrix, precision_recall_curve
19 from sklearn.metrics import recall_score, classification_report, auc, roc_curve
20 from sklearn.metrics import precision_recall_fscore_support, f1_score
21
22 from numpy.random import seed
23 seed(7)
24 from tensorflow import set_random_seed
25 set_random_seed(11)
26
27 from sklearn.model_selection import train_test_split
28
29 SEED = 123 #used to help randomly select the data points
30 DATA_SPLIT_PCT = 0.2
31
32 rcParams['figure.figsize'] = 8, 6
33 LABELS = ["Normal", "Attack"]
34
35 X_train, X_test, y_train, y_test = train_test_split(np.array(X), np.array(y), test_size=0.2)
36 X_train, X_valid, y_train, y_valid = train_test_split(X_train, y_train, test_size=DATA_SPLIT_PCT, random_state=SEED)
```

In [151]:

```
1 X_train.shape
```

Out[151]:

```
(852, 5, 1, 87)
```

In [152]:

```
1 X_train_y0 = X_train[y_train==0]
2 X_train_y1 = X_train[y_train==1]
3
4 X_valid_y0 = X_valid[y_valid==0]
5 X_valid_y1 = X_valid[y_valid==1]
```

In [153]:

```
1 X_train_y0.shape
```

Out[153]:

```
(542, 5, 1, 87)
```

Reshaping the data

The tensors we have here are 4-dimensional. We will reshape them into the desired 3-dimensions corresponding to sample x lookback x features.

In [154]:

```
1 X_train = X_train.reshape(X_train.shape[0], lookback, n_features)
2 X_train_y0 = X_train_y0.reshape(X_train_y0.shape[0], lookback, n_features)
3 X_train_y1 = X_train_y1.reshape(X_train_y1.shape[0], lookback, n_features)
4
5 X_test = X_test.reshape(X_test.shape[0], lookback, n_features)
6
7 X_valid = X_valid.reshape(X_valid.shape[0], lookback, n_features)
8 X_valid_y0 = X_valid_y0.reshape(X_valid_y0.shape[0], lookback, n_features)
9 X_valid_y1 = X_valid_y1.reshape(X_valid_y1.shape[0], lookback, n_features)
```

In [155]:

```
1 n_features
```

Out[155]:

```
87
```

Standardize the data It is usually better to use a standardized data (transformed to Gaussian, mean 0 and sd 1) for autoencoders.

One common mistake is: we normalize the entire data and then split into train-test. This is not correct. Test data should be completely unseen to anything during the modeling. We should normalize the test data using the feature summary statistics computed from the training data. For normalization, these statistics are the mean and variance for each feature.

The same logic should be used for the validation set. This makes the model more stable for a test data.

To do this, we will require two UDFs.

flatten: This function will re-create the original 2D array from which the 3D arrays were created. This function is the inverse of temporalize, meaning $X = \text{flatten}(\text{temporalize}(X))$. scale: This function will scale a 3D array that we created as inputs to the LSTM.

In [156]:

```
1 def flatten(X):
2     '''
3     Flatten a 3D array.
4
5     Input
6     X          A 3D array for lstm, where the array is sample x timesteps x features
7
8     Output
9     flattened_X A 2D array, sample x features.
10    '''
11    flattened_X = np.empty((X.shape[0], X.shape[2])) # sample x features array.
12    for i in range(X.shape[0]):
13        flattened_X[i] = X[i, (X.shape[1]-1), :]
14    return(flattened_X)
15
16 def scale(X, scaler):
17     '''
18     Scale 3D array.
19
20     Inputs
21     X          A 3D array for lstm, where the array is sample x timesteps x features
22     scaler      A scaler object, e.g., sklearn.preprocessing.StandardScaler, sklearn.
23
24     Output
25     X          Scaled 3D array.
26    '''
27    for i in range(X.shape[0]):
28        X[i, :, :] = scaler.transform(X[i, :, :])
29
30    return X
```

In [157]:

```
1 # Initialize a scaler using the training data.
2 scaler = StandardScaler().fit(flatten(X_train_y0))
```

In [158]:

```
1 X_train_y0_scaled = scale(X_train_y0, scaler)
2 X_train_y1_scaled = scale(X_train_y1, scaler)
3 X_train_scaled = scale(X_train, scaler)
```

In [159]:

```

1  '''
2  Test: Check if the scaling is correct.
3
4  The test succeeds if all the column means
5  and variances are 0 and 1, respectively, after
6  flattening.
7  '''
8  a = flatten(X_train_y0_scaled)
9  print('colwise mean', np.mean(a, axis=0).round(6))
10 print('colwise variance', np.var(a, axis=0))

```

```
colwise mean [0.169742 0.142066 0.195572 0.156827 0.051661 0.073801 0.055351
0.125461
```

```

0.136531 0.132841 0.          0.          0.175277 0.178967 0.177122 0.178967
0.114391 0.145756 0.154982 0.180812 0.182657 0.125461 0.051661 0.142066
0.145756 0.190037 0.04797  0.193727 0.042435 0.156827 0.075646 0.123616
0.042435 0.119926 0.114391 0.140221 0.105166 0.162362 0.081181 0.079336
0.          0.          0.          0.          0.073801 0.051661 0.046125 0.
0.          0.068266 0.208487 0.042435 0.          0.110701 0.          0.077491
0.          0.154982 0.180812 0.057196 0.042435 0.101476 0.062731 0.094096
0.107011 0.127306 0.136531 0.          0.          0.114391 0.          0.070111
0.079336 0.          0.          0.          0.094096 0.090406 0.105166 0.081181
0.081181 0.092251 0.081181 0.110701 0.086716 0.114391 0.145756]

```

```
colwise variance [0.71657521 0.77502008 0.62595825 0.7226379  0.95674079 0.9
4289293
```

```

0.96003595 0.84403807 0.7931673  0.83475171 0.          0.
0.67592013 0.70413325 0.69925518 0.67461295 0.81717297 0.75919786
0.72505821 0.60937351 0.64006822 0.8071377  0.9382906 0.70121935
0.68908716 0.64838782 0.98293869 0.53627061 0.97421399 0.68942757
0.85590134 0.80944227 0.97421399 0.83617121 0.82086301 0.784766
0.81366335 0.70795605 0.87532849 0.87746967 0.          0.
0.          0.          0.90230253 0.98626108 0.97757724 0.
0.          0.92338408 0.61520472 0.97421399 0.          0.82538364
0.          0.90543429 0.          0.76564862 0.73114473 0.92846298
0.97421399 0.84394616 0.95178443 0.85276957 0.78928664 0.83065658
0.80054738 0.          0.          0.82455304 0.          0.92128375
0.91068  0.          0.          0.          0.88966994 0.89035076
0.8690139 0.89746872 0.90115875 0.86602851 0.91222886 0.90656445
0.89469438 0.87621356 0.77395801]

```

The test succeeded. Now we will *scale* the validation and test sets.

In [160]:

```

1  X_valid_scaled = scale(X_valid, scaler)
2  X_valid_y0_scaled = scale(X_valid_y0, scaler)
3
4  X_test_scaled = scale(X_test, scaler)

```

LSTM Autoencoder training

First we will initialize the Autoencoder architecture. We are building a simple autoencoder. More complex architectures and other configurations should be explored.

In [161]:

```

1 timesteps = X_train_y0_scaled.shape[1] # equal to the lookback
2 n_features = X_train_y0_scaled.shape[2] # 87
3
4 epochs = 200
5 batch = 64
6 lr = 0.0001

```

In [162]:

```
1 n_features
```

Out[162]:

87

In [166]:

```

1 lstm_autoencoder = Sequential()
2 # Encoder
3 lstm_autoencoder.add(LSTM(32, activation='relu', input_shape=(timesteps, n_features),
4 lstm_autoencoder.add(LSTM(16, activation='relu', return_sequences=False))
5 lstm_autoencoder.add(RepeatVector(timesteps))
6 # Decoder
7 lstm_autoencoder.add(LSTM(16, activation='relu', return_sequences=True))
8 lstm_autoencoder.add(LSTM(32, activation='relu', return_sequences=True))
9 lstm_autoencoder.add(TimeDistributed(Dense(n_features)))
10
11 lstm_autoencoder.summary()

```

Layer (type)	Output Shape	Param #
=====		
lstm_3 (LSTM)	(None, 5, 32)	15360
lstm_4 (LSTM)	(None, 16)	3136
repeat_vector_1 (RepeatVecto	(None, 5, 16)	0
lstm_5 (LSTM)	(None, 5, 16)	2112
lstm_6 (LSTM)	(None, 5, 32)	6272
time_distributed_1 (TimeDist	(None, 5, 87)	2871
=====		
Total params: 29,751		
Trainable params: 29,751		
Non-trainable params: 0		

As a rule-of-thumb, look at the number of parameters. If not using any regularization, keep this less than the number of samples. If using regularization, depending on the degree of regularization you can let more parameters in the model that is greater than the sample size. For example, if using dropout with 0.5, you can have up to double the sample size (loosely speaking).

In [168]:

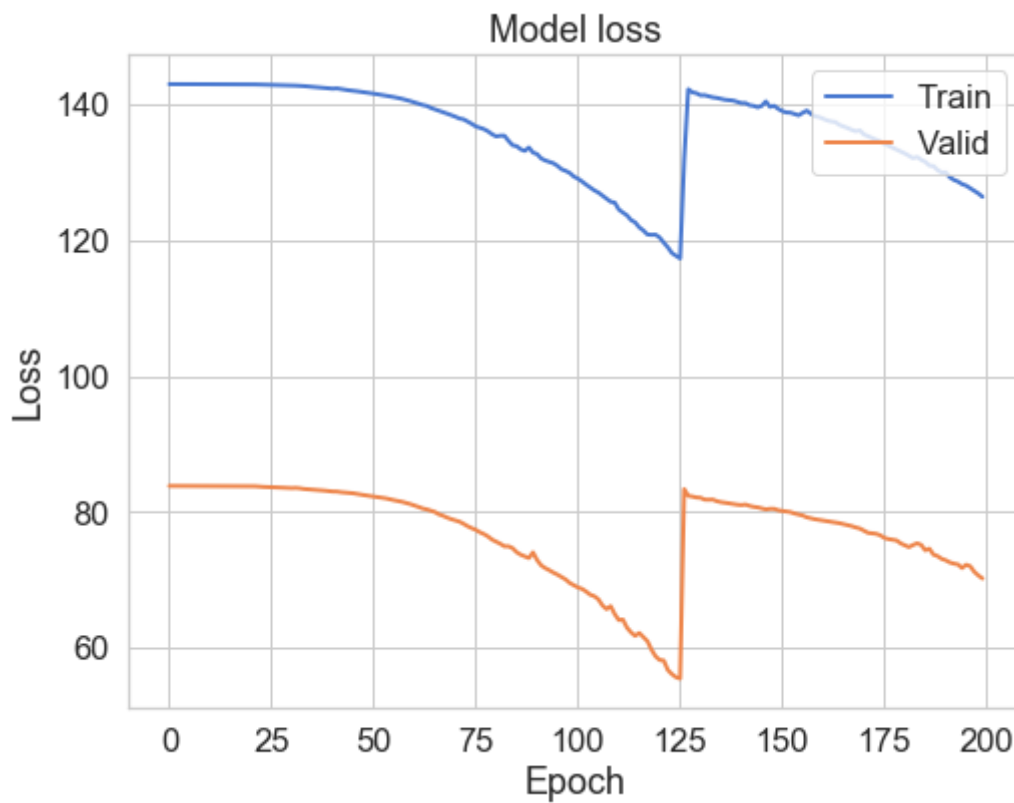
```
1 adam = optimizers.Adam(lr)
2 lstm_autoencoder.compile(loss='mse', optimizer=adam)
3
4 cp = ModelCheckpoint(filepath="lstm_autoencoder_classifier.h5",
5                       save_best_only=True,
6                       verbose=0)
7
8 tb = TensorBoard(log_dir='./logs',
9                  histogram_freq=0,
10                  write_graph=True,
11                  write_images=True)
12
13 lstm_autoencoder_history = lstm_autoencoder.fit(X_train_y0_scaled, X_train_y0_scaled,
14                                                epochs=epochs,
15                                                batch_size=batch,
16                                                validation_data=(X_valid_y0_scaled, X_valid_y0_scaled),
17                                                verbose=2).history
```

```
Epoch 141/200
- 0s - loss: 140.2282 - val_loss: 81.0555
Epoch 142/200
- 0s - loss: 140.2263 - val_loss: 81.1333
Epoch 143/200
- 0s - loss: 139.9190 - val_loss: 80.9368
Epoch 144/200
- 0s - loss: 139.8253 - val_loss: 80.7971
Epoch 145/200
- 0s - loss: 139.6162 - val_loss: 80.7180
```

Plotting the change in the loss over the epochs.

In [169]:

```
1 plt.plot(lstm_autoencoder_history['loss'], linewidth=2, label='Train')
2 plt.plot(lstm_autoencoder_history['val_loss'], linewidth=2, label='Valid')
3 plt.legend(loc='upper right')
4 plt.title('Model loss')
5 plt.ylabel('Loss')
6 plt.xlabel('Epoch')
7 plt.show()
```



Sanity check

Doing a sanity check by validating the reconstruction error on the train data. Here we will reconstruct the entire train data with both 0 and 1 labels.

Expectation: the reconstruction error of 0 labeled data should be smaller than 1.

Caution: do not use this result for model evaluation. It may result into overfitting issues.

In [171]:

```

1 train_x_predictions = lstm_autoencoder.predict(X_train_scaled)
2 mse = np.mean(np.power(flatten(X_train_scaled) - flatten(train_x_predictions), 2), axis=1)
3
4 error_df = pd.DataFrame({'Reconstruction_error': mse,
5                          'True_class': y_train.tolist()})
6
7 groups = error_df.groupby('True_class')
8 fig, ax = plt.subplots()
9
10 for name, group in groups:
11     ax.plot(group.index, group.Reconstruction_error, marker='o', ms=3.5, linestyle='',
12            label= "Attack" if name == 1 else "Normal")
13 ax.legend()
14 plt.title("Reconstruction error for different classes")
15 plt.ylabel("Reconstruction error")
16 plt.xlabel("Data point index")
17 plt.show();

```



Predictions using the Autoencoder

Here we show how we can use an Autoencoder reconstruction error for the rare-event classification. We follow this concept: the autoencoder is expected to reconstruct a noif the reconstruction error is high, we will classify it as a sheet-break.

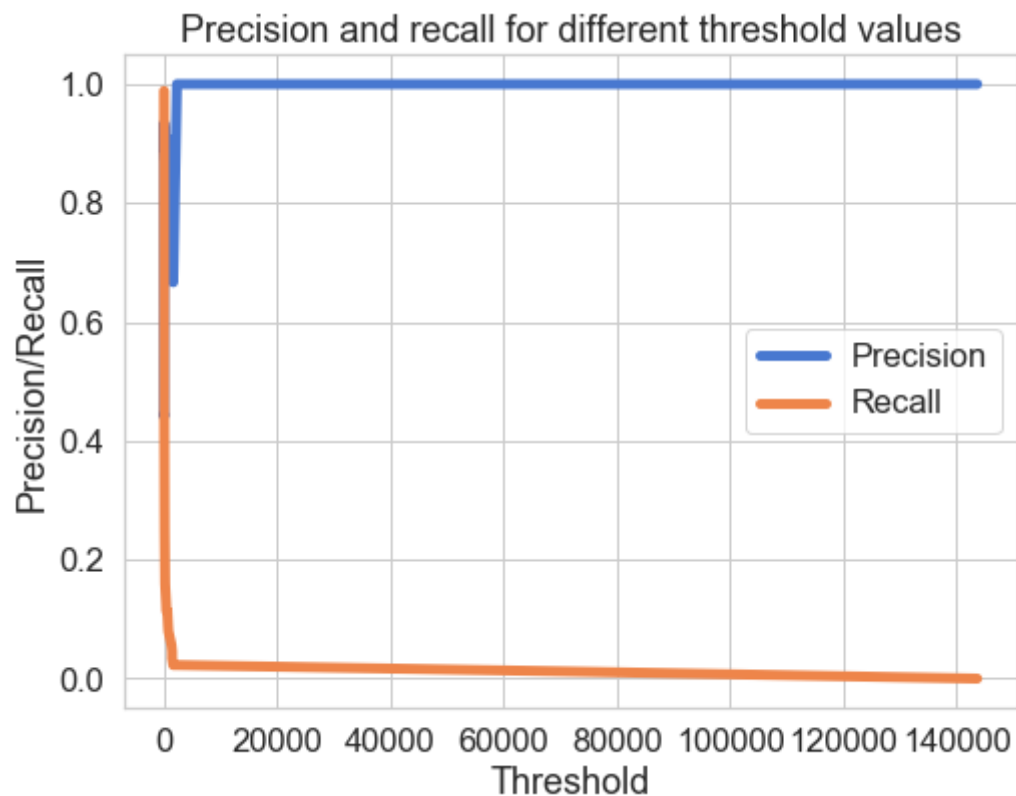
We will need to determine the threshold for this. Also, note that here we will be using the entire validation set containing both $y = 0$ or 1 .

In [172]:

```

1 valid_x_predictions = lstm_autoencoder.predict(X_valid_scaled)
2 mse = np.mean(np.power(flatten(X_valid_scaled) - flatten(valid_x_predictions), 2), axis=
3
4 error_df = pd.DataFrame({'Reconstruction_error': mse,
5                          'True_class': y_valid.tolist()})
6
7 precision_rt, recall_rt, threshold_rt = precision_recall_curve(error_df.True_class, en
8 plt.plot(threshold_rt, precision_rt[1:], label="Precision",linewidth=5)
9 plt.plot(threshold_rt, recall_rt[1:], label="Recall",linewidth=5)
10 plt.title('Precision and recall for different threshold values')
11 plt.xlabel('Threshold')
12 plt.ylabel('Precision/Recall')
13 plt.legend()
14 plt.show()

```



In [174]:

```

1 test_x_predictions = lstm_autoencoder.predict(X_test_scaled)
2 mse = np.mean(np.power(flatten(X_test_scaled) - flatten(test_x_predictions), 2), axis=
3
4 error_df = pd.DataFrame({'Reconstruction_error': mse,
5                           'True_class': y_test.tolist()})
6
7 threshold_fixed = 0.3
8 groups = error_df.groupby('True_class')
9 fig, ax = plt.subplots()
10
11 for name, group in groups:
12     ax.plot(group.index, group.Reconstruction_error, marker='o', ms=3.5, linestyle='',
13             label= "Attack" if name == 1 else "Normal")
14 ax.hlines(threshold_fixed, ax.get_xlim()[0], ax.get_xlim()[1], colors="r", zorder=100,
15 ax.legend()
16 plt.title("Reconstruction error for different classes")
17 plt.ylabel("Reconstruction error")
18 plt.xlabel("Data point index")
19 plt.show();

```



The orange and blue dot above the threshold line represents the True Positive and False Positive, respectively. As we can see, we have good number of false positives.

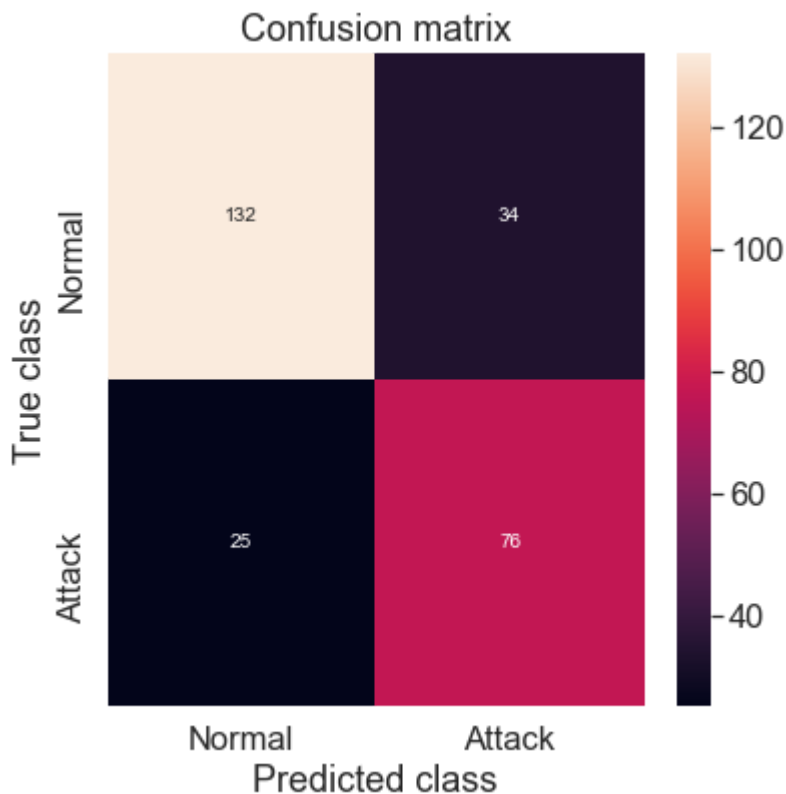
Let's see the accuracy results.

In [175]:

```
1 pred_y = [1 if e > threshold_fixed else 0 for e in error_df.Reconstruction_error.values]
```

In [178]:

```
1 conf_matrix = confusion_matrix(error_df.True_class, pred_y)
2
3 plt.figure(figsize=(6, 6))
4 sns.heatmap(conf_matrix, xticklabels=LABELS, yticklabels=LABELS, annot=True, fmt="d");
5 plt.title("Confusion matrix")
6 plt.ylabel('True class')
7 plt.xlabel('Predicted class')
8 plt.show()
```

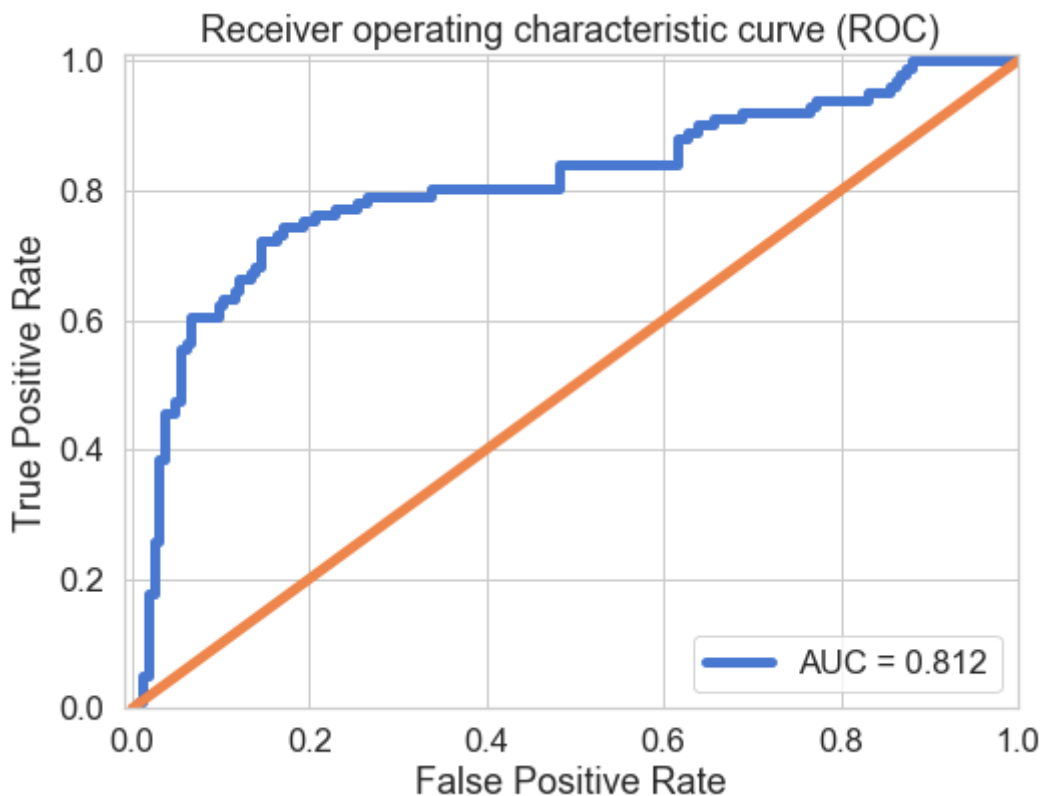


In [179]:

```

1 false_pos_rate, true_pos_rate, thresholds = roc_curve(error_df.True_class, error_df.Re
2 roc_auc = auc(false_pos_rate, true_pos_rate,)
3
4 plt.plot(false_pos_rate, true_pos_rate, linewidth=5, label='AUC = %0.3f'% roc_auc)
5 plt.plot([0,1],[0,1], linewidth=5)
6
7 plt.xlim([-0.01, 1])
8 plt.ylim([0, 1.01])
9 plt.legend(loc='lower right')
10 plt.title('Receiver operating characteristic curve (ROC)')
11 plt.ylabel('True Positive Rate')
12 plt.xlabel('False Positive Rate')
13 plt.show()

```



Conclusion

From the Confusion Matrix in Figure above, we could predict 76 out of 101 Attack instances.

The primary reason is LSTM model has more parameters to estimate. It becomes important to use regularization with LSTMs.