Python Programs for ‘DDoS Detection and Mitigation Using Machine Learning Tools’

Arpit Gawande

March 19, 2018

1 Data extraction from Wireshark capture file

Library imports

```
#Pandas for creating data-frames
import pandas as pd

#Pyshark to capture packets
import pyshark
#For file operarions
import os
import gc
import sys
```

Define required attributed needed to extract from capture.

```
required_keys = ['ip.dst', 'ip.proto', 'tcp.flags.syn', 'tcp.flags.ack']
```

Reading packets from pre-captured file using Pyshark

```
file_cap = pyshark.FileCapture('captures/botnet-capture-20110810-neris.pcap')
```

Extract IP packets from the captured file for further processing

```
%time
#Write extraction result to
base_directory = 'converted/test3/attack_samples/1/'
#Debugger for pyshark
#file_cap.set_debug()
#Attributes for sample size
startTime = 0.0
endTime = 0.0
i = -1
first = True
#Sample Number
init_sample = 1
dfList = []
while(True):
    i += 1
    #Each packet can have multiple layers (e.g eth, ip, tcp etc.).
    #Combine them in list for a single packet.
    layerList = []
    #pyshark not able to handle AttributeError, AssertionError and KeyError(index values)
    try:
        #Iterate through layer of packet
        for layer in file_cap[i]:
            # Slice Data according to time
            t = float(file_cap[i].sniff_timestamp)
            #Initial setup
            if startTime == 0.0:
                startTime = t
                endTime = startTime + 300 # 300 sec(5 min) is the sample size
                sample = init_sample
                print(sample, startTime, endTime)
            # Write every sample to csv file
            elif t > endTime:
                if not os.path.exists(base_directory):
                    os.makedirs(base_directory)
                dfList.to_csv(base_directory+str(sample))
                #Clear list after writing this would save memory
                startTime = endTime
                endTime = startTime + 300
                sample += 1
                #Reset after writing to file
                first = True
                print(sample, startTime, endTime)
                gc.collect()
            #We need only ip layer only
            if layer._layer_name == 'ip':
                #Layer values are in the form of dictionary. Filter the attributes
                layer_dict = {key:value for key, value in layer._all_fields.items()
                              if key in required_keys}
                #Create data-frame from dictionary
                layerList.append(pd.DataFrame(layer_dict, index=[0]))
                #Add timestamp and sample number
                layerList.append(pd.DataFrame({'sniff_timestamp':file_cap[i].sniff_timestamp},
                                              index=[0]))
                layerList.append(pd.DataFrame({'sample':sample}, index=[0]))
                #Build packet dataframe from layer frames. Its single row dataframe
                cDf = pd.concat(layerList, axis=1);
                if first:
                    dfList = cDf
                    first = False
                else:
                    dfList = dfList.append(cDf, ignore_index=True)
    except (AttributeError, AssertionError) as e:
        continue  #print('Ipv4 packet does not exist')
    except  KeyError:
        break;

#If sample data is not written to file because for frame size
if(sample == init_sample):
    dfList.to_csv(base_directory+str(init_sample))
```

2 DDoS attack Detection based on destination IP address

Library imports

```
#Pandas for creating data-frames
import pandas as pd

#Pyshark to capture packets
import pyshark
#For file operations
import os
import gc
import sys
```

Declare and define file/folders for storing/retrieving data for analysis

```
#Base Folder Paths
base_folder_name = 'converted'
test_folder_name = 'test2'
base_path = os.path.join(base_folder_name, test_folder_name)
#Normal Sample and cluster paths
sample_dir_name = 'samp'
sample_path = os.path.join(base_path, sample_dir_name)
#If sample folder exist
if os.path.isdir(sample_path):
    #Test and Train data Folders
    sample_train_path = os.path.join(base_path, sample_dir_name+'_train')
    os.makedirs(sample_train_path, exist_ok=True)
    sample_test_path = os.path.join(base_path, sample_dir_name+'_test')
    os.makedirs(sample_test_path, exist_ok=True)
    #Clustering folders
    cluster_train_path = os.path.join(sample_train_path, 'cluster')
    os.makedirs(cluster_train_path, exist_ok=True)
    cluster_test_path = os.path.join(sample_test_path, 'cluster')
    os.makedirs(cluster_test_path, exist_ok=True)
```

Defining features we want to work with

```
features = [1, 6, 17] #(ICMP:1, TCP:6, UDP:17)
```

Creating feature tables which would be input to k-means learning algorithms

```
def get_feature_dataframe(sample_file, features):
    """
    Create feature table from the sample file which has flow information

    Parameters
    ----------

    sample_file: string
        file(full path) containing IP layer information destination address,
        protocol, time stamp.

    features: list
        list of feature we required for anlysis.
    """
    df = pd.read_csv(sample_file, index_col=0)
    #Filter Columns
    df = df[['ip.dst', 'ip.proto', 'sniff_timestamp', 'sample']]
    #Remove null destinations
    df = df[df['ip.dst'].notnull()]
    #Rename Columns
    df.columns = ['ip', 'protocol', 'time_stamp', 'sample']
    #Get count for each ip
    df = df.groupby(['ip', 'protocol']).size().unstack().fillna(0).astype(int)
    #Drop row for given IP
    #df = filter_ip(df, '147.32.84.165')
    if(set(df.columns) != set(features)):
        non_columns = set(features) - set(df.columns)
        for c in non_columns:
            df.insert(loc=features.index(c), column=c, value=0)
    #Select only required protocols which would be used as features
    df = df[features]
    return df

#Merge sample files to create bigger samples
def merge_files(files, merge_count, features):
    """
    Merge feature sample file to form bigger sample.

    Parameters
    ----------
    files: list
        all the sample files,

    merge_count: int
        number of files to be merged.

    features: list
        list of features

    """
    if len(files) < merge_count:
        print('Too few file to merge')
        return
    dfs = []
    count = 0
    for file in files:
        if count == 0:
            df = get_feature_dataframe(file, features)
            count += 1
        else:
            temp_df = get_feature_dataframe(file, features)
            df = df.append(temp_df)
            count += 1
        if count == merge_count:
            df = df.groupby(df.index).sum()
            dfs.append(df)
            count = 0
    return dfs

def create_train_test(sample_path, features, merge_count=1):
    """
    Seperate data into train and test for the evaluation of the algorithms

    Parameters
    ----------

    sample_path: string
        location where all the packet sample files are kept
    merge_count: int, default 1
        Number of files to be merged to create a feature table

    feature: list
        List of features to be considered for creating feature table
    """
    files = sorted(glob.glob(os.path.join(sample_path,'*')),  key=os.path.getmtime)

    test_files = files[:merge_count]
    train_files = files[merge_count:]

    test_dfs = merge_files(test_files, merge_count, features)
    train_dfs = merge_files(train_files, merge_count, features)

    return train_dfs, test_dfs, merge_count
```

Create train and test data frames

```
train_dfs, test_dfs, merge_count = create_train_test(sample_path, features)
```

Save merge count which will be used for merging the files to form larger sample.

```
np.savetxt(os.path.join(base_path, 'merge_count'),
           np.asarray(merge_count).reshape(1,), fmt='%d')
```

Store train and test tables on the file system

```
#Write train and test data to file
train_files = []
for i, df in enumerate(train_dfs, 1):
    train_file = os.path.join(sample_train_path,str(i))
    df.to_csv(train_file)
    train_files.append(train_file)
test_files = []
for i, df in enumerate(test_dfs, 1):
    test_file = os.path.join(sample_test_path,str(i))
    df.to_csv(test_file)
    test_files.append(test_file)
```

3 k-means clustering

Functions for finding optimal number of clusters for k-means clustering algorithm using elbow method.

```
#Find optimal number of clusters for k-means clustering using elbow method.
def elbow_method(X_trans, ax, title):
    """
    For the k-means algorithm, elbow method check the percent of
    variance explained as function of the number of clusters.
    Variance for each cluster number is calculated and the cluster
    number which produce less variance for the next cluster number
    is selected as best choice.

    Parameters
    ----------

    X_tran: vector
        Standardize vector.
    """
    elbow_count = 0
    range_val = 10
    nc = range(1, range_val)
    kmeans = [KMeans(n_clusters=i) for i in nc]
    score = [kmeans[i].fit(X_trans).score(X_trans) for i in range(len(kmeans))]
    total_diff = abs(score[0] - score[len(score) -1])
    for i in range(range_val - 2):
        percent_diff = abs(score[i] - score[i+1])/total_diff
        if percent_diff < 0.01:
            opt_clust_count = i
            break
    ax.plot(nc,score)
    ax.set_xlabel('Number of Clusters')
    ax.set_ylabel('Score')
    ax.set_title(title)
    return opt_clust_count

def get_optimal_cluster_count(df_list, count):
    """
    Apply elbow method on all the sample vectors and get their mean as
    the best cluster number for kmeans algorithm.

    Parameters
    ----------

    df_list: list of dataframes
        contain tables/dataframes for each sample data.

    count: int
        Number of sample to loop t.
    """
    elbow_vals = []
    row_count = math.ceil(count/2)
    fig = plt.figure(figsize=(10, 4*row_count), dpi=80, facecolor='w', edgecolor='k')
    fig.subplots_adjust(hspace=.5) #Adjust space between the subplot
    for i, df in enumerate(df_list[:count], 1):
        X = df.values
        #Create scaling and transforme
        X_trans = preprocessing.StandardScaler().fit_transform(X)
        #Create subplot
        ax = fig.add_subplot(row_count, 2, i)
        title = 'Sample:'+str(i)
        fig.suptitle('Elbow Method', fontsize=16)
        elbow = elbow_method(X_trans, ax, title)
        elbow_vals.append(elbow)
    plt.savefig('elbow-method.png')
    return int(np.floor(np.mean(elbow_vals)))
```

Get the optimal cluster count

```
sample_count = 4 #Number of samples to be used for elbow method
cluster_count = get_optimal_cluster_count(train_dfs, sample_count)
```

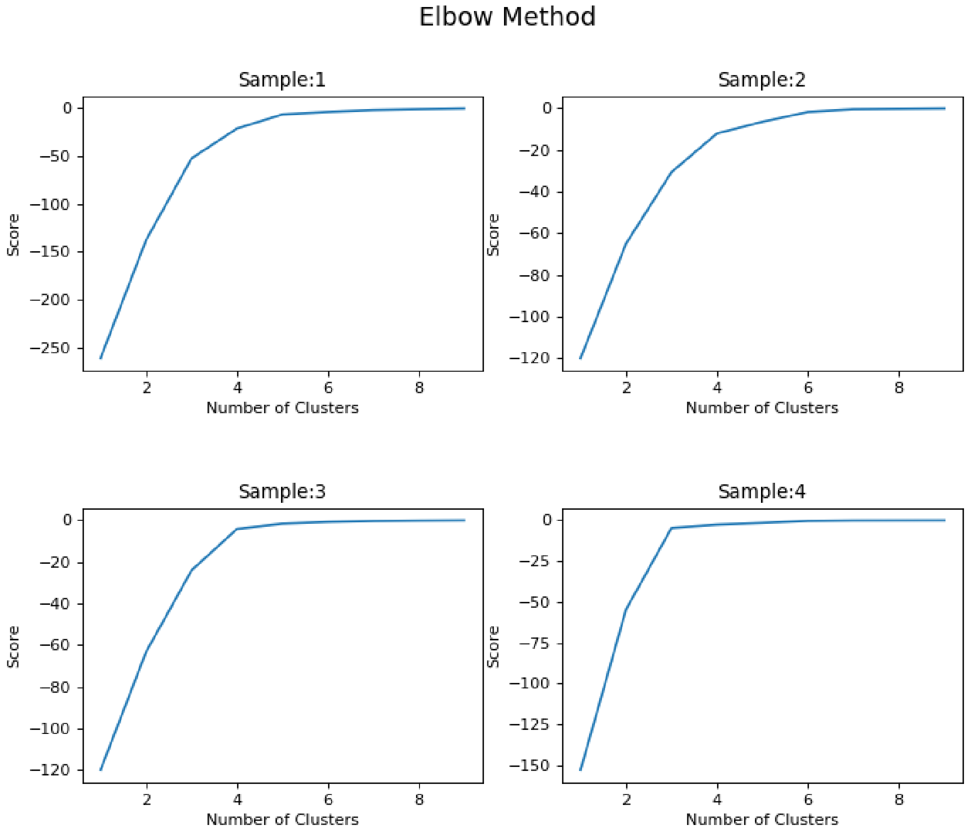

Figure 1: Elbow Method for cluster count detection

Functions for finding and storing the centroid which would be used for all k-means clustering.

```
def get_kmeans_centroid(feature_df, cluster_count):
    """
    Run the k-means algorithm on input data vector and find out the
    centroid for each cluster then return that centroid in the
    form of table

    Parameters
    ----------

    feature_df: table/dataframe
        vector on which the k-means algorithm is be applied

    cluster_count: int
        Number of clusters to be used for k-means clustering
    """
    df_centroid = {}
    X = feature_df.values
    #Create scaling
    scaler = preprocessing.StandardScaler().fit(X)
    #Transform Training data
    X_trans = scaler.transform(X)
    #Data Fitting using K-means
    kmeans = KMeans(n_clusters=cluster_count)
    kmeans.fit(X_trans)
    #Create cluster data-frame for saving on file system
    #Dataframe 0 contain all the clusters centers associated with 0th cluster
    first = True
    for i in range(kmeans.cluster_centers_.shape[0]):
        s = pd.Series(kmeans.cluster_centers_[i], index=feature_df.columns)
        if(first):
            df_centroid = pd.DataFrame(columns=feature_df.columns)
            first = False
        df_centroid = df_centroid.append(s,ignore_index=True)
    return df_centroid

def get_kmeans_cluster_centroids(sample_frames, features, cluster_count):
    """
    This function will first find all cluster centroids for all the
    training sample vectors and then calculate the median of each
    cluster over all the samples. This median would be the best guess
    of the centroid for all the training vectors and also would be use
    for evaluation.

    Parameters
    ----------

    sample_frames: list
        list of feature vector datafames.

    features: list
        list of feature.

    cluster_count: int
        number of clusters to be considered for k-means clustering.

    """
    df_concat = pd.DataFrame(columns=features)
    for df in sample_frames:
        #Run kmeans and get centroid
        df_centroid = get_kmeans_centroid(df, cluster_count)
        #Create list of centroids
        df_concat = df_concat.append(df_centroid)
    centroids = []
    #Find median for each centroid and store them in file
    for c in range(cluster_count):
        med = np.median(df_concat.loc[c], axis=0) # e.g. df_concat.loc[0] is df of clister 0
        centroids.append(med)
    return centroids
```

Find the median of all the centroids. This median is the better estimation of the centroids for all future k-means clustering. Store it on the file system. Also store the feature list on file system:

```
#Already define base directory path
base_path = base_path
#File to store centroids
centroid_filename = 'centroids.csv'
#File to store feature list
features_filename = 'features.csv'

#Call to find median of centroid
centroids = get_kmeans_cluster_centroids(train_dfs, features, cluster_count)

#Save centroids and features in files for future use
np.savetxt(os.path.join(base_path, centroid_filename),
           np.asarray(centroids), delimiter=",")
np.savetxt(os.path.join(base_path, features_filename),
           np.asarray(list(features)), delimiter=",")

def read_centroid_features(base_path, centroid_filename, features_filename):
    """
    Read previously calculated centroids and stored feature from the file system.

    Parameters
    ----------

    base_path: string
        folder location in which both centroids and feature file are present

    centroid_filename: string
        Name of the file in which centroid vector is stored.

    features_filename: string
        Name of the file in which feature list is stored.

    """
    centroids = np.genfromtxt(os.path.join(base_path,centroid_filename), delimiter=',')
    features = np.genfromtxt(os.path.join(base_path,features_filename), delimiter=',')
    features = list(features.astype('int64'))
    return centroids, features
```

After finding cluster count and the common centroids we will be doing kmeans clustering using previously calculated centroids

```
#Base directory path
base_path = base_path #Already define
```

```
#File to store centroids
centroid_filename = centroid_filename #Already define
#File to store feature list
features_filename = features_filename #Already define
#Get centroids and feature
centroids, features = read_centroid_features(base_path,
                                              centroid_filename, features_filename)
```

Plot k-means clustering for samples

```
def draw_clusters(X, pre_centroids, ax, title):
    """
    Draw clusters in 2D

    Parameters
    ----------

    X: vector
        vector to be clustered and plot.

    ax: object
        pyplot subplot object for plotting the clusters.

    title: string
        title of the plot.

    """
    if X.shape[1] > 2:
        #Use PCA component analysis for 2D visuals
        reduced_X = PCA(n_components=2).fit_transform(X)
        km = KMeans(n_clusters=pre_centroids.shape[0])
        km.fit(reduced_X)
    else:
        reduced_X = X
        km = KMeans(n_clusters=pre_centroids.shape[0], init=pre_centroids)
        km.fit(reduced_X)

    # Step size of the mesh. Decrease to increase the quality of the VQ.
    h = .01     # point in the mesh [x_min, x_max]x[y_min, y_max].

    # Plot the decision boundary. For that, we will assign a color to each
    x_min, x_max = reduced_X[:, 0].min() - 1, reduced_X[:, 0].max() + 1
    y_min, y_max = reduced_X[:, 1].min() - 1, reduced_X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))

    # Obtain labels for each point in mesh. Use last trained model.
    Z = km.predict(np.c_[xx.ravel(), yy.ravel()])

    # Put the result into a color plot
    Z = Z.reshape(xx.shape)
    plt.imshow(Z, interpolation='nearest',
               extent=(xx.min(), xx.max(), yy.min(), yy.max()),
               cmap=plt.cm.Paired,
               aspect='auto', origin='lower')

    #Plot the data points
    ax.plot(reduced_X[:,0],reduced_X[:,1],  'k.', markersize=3)
    # Plot the centroids as a white X
    centroids = km.cluster_centers_
    ax.scatter(centroids[:, 0], centroids[:, 1], marker='x', s=169,
               linewidths=3, color='w', zorder=10)
    #Set tile and boundaries of the plot
    ax.set_title(title)
    ax.set_xlim(x_min, x_max)
    ax.set_ylim(y_min, y_max)
    ax.set_xticks(())
    ax.set_yticks(())

def plot_kmeans_clusters(to_print_df):
    fig = plt.figure(figsize=(10, 8), dpi=80, facecolor='w', edgecolor='k')
    row_count = math.ceil(len(to_print_df)/2)
    for i, df in enumerate(to_print_df, 1):
        X = df.values
        #Create scaling
        X_trans = preprocessing.StandardScaler().fit_transform(X)
        #Create subplot
        ax = fig.add_subplot(row_count, 2, i)
        plt.suptitle('k-means++ clustering (with PCA-reduced data)', fontsize=16)
        title = 'Sample:'+str(i)
        draw_clusters(X_trans, centroids, ax, title)
    plt.savefig('kemans-clustering.png')
```

Plot only few clusters as not all required.

```
plot_count = 4
to_print_df = train_dfs[:plot_count]
plot_kmeans_clusters(to_print_df)
```

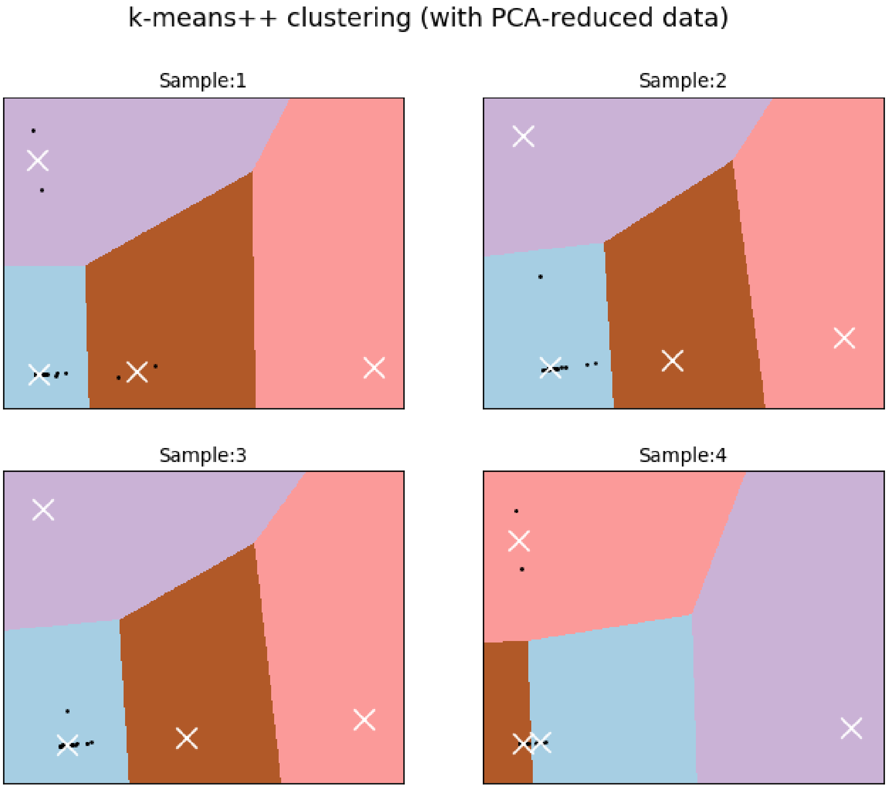

Figure 2: Clustering using k-means++ algorithm

Do k-means++ clustering and save clustered output in the files

```
def kmeans_clustering(feature_df, centroids):
    """
    Clustering input vector using k-means algorithm

    Parameter
    ---------
    feature_df: Object
        Panda dataframe object containing input vector which
        is to clustered using k-means.

    centroids: vector
        Vector containing centroid values. These centroids are
        used for clutering.

    """
    X = feature_df.values
    #Create scaling
    scaler = preprocessing.StandardScaler().fit(X)
    #Transform Training data
    X_trans = scaler.transform(X)
    #k means clustering using provided centroids
    kmeans = KMeans(n_clusters=centroids.shape[0], init=centroids)
    clusters = kmeans.fit_predict(X_trans)
    #Getting the labels/clusters for each IP
    cluster_df = pd.DataFrame({'cluster': kmeans.labels_})
    #Attaching labels to existing dataframe and return new dataframe
    df = pd.concat([feature_df.reset_index(), cluster_df], axis=1).set_index('ip')
    return df
```

Apply clustering on both training and test data. Store clustered information on the file system

```
#First retrieve the centroids and features
centroids, features = read_centroid_features(centroid_filename, features_filename)
```

Cluster all the samples and store them on file.

```
centroids, features = read_centroid_features(centroid_filename, features_filename)
for i, train_df in enumerate(train_dfs):
    clustered_df = kmeans_clustering(train_df, centroids)
    clustered_df.to_csv(os.path.join(cluster_train_path,str(i+1)))
for i, test_df in enumerate(test_dfs):
    clustered_df = kmeans_clustering(test_df, centroids)
    clustered_df.to_csv(os.path.join(cluster_test_path,str(i+1)))
```

Now combine clustering of all the train/test samples and find out the best cluster and average packet count for each IP

```
def get_best_cluster_sample_count(cluster_path):
    """
    Get all the trained IP address and then combine training data
    from all the sample to make best judgment about the cluster
    in which IP address should be placed into. It also gives the
    average number of packets flowing during the analysis time
    which would help to determine if its truly attack or
    just a misjudgment.

    Parameters
    ----------
    cluster_path: string
        directory location where the clustered samples are stored.

    """
    files = sorted(glob.glob(os.path.join(cluster_path,'*')),  key=os.path.getmtime)
    first = True
    for file in files:
        if first:
            df = pd.read_csv(file, index_col=0)
            first = False
        else:
            temp_df = pd.read_csv(file, index_col=0)
            df = df.append(temp_df)
    #Fist reindex all data with IP
    df = df.reset_index().set_index(['ip'])

    new_columns =  ['cluster', 'packet_count']
    features = list(df.columns[:-1])
    cluster_column = list(df.columns[-1:])
    #Average packet count across all the samples
    packet_count = df[features].groupby('ip').mean().sum(axis=1).astype('int64')
    #most occurring cluster across all samples
    cluster = df[cluster_column].groupby('ip').median().astype('int64')
    #Create new dataframe with cluster and packet count and return new table/dataframe
    new_df = pd.concat([cluster, packet_count], axis=1)
    new_df.columns = new_columns
    return new_df
```

Get tables containing best judged cluster and average packet count

```
train_df = get_best_cluster_sample_count(cluster_train_path)
test_df = get_best_cluster_sample_count(cluster_test_path)
```

Function for finding RI index

```
import numpy as np
from scipy.misc import comb

def get_rand_index_score(train_df, test_df):
    r = pd.merge(train_df, test_df, left_index=True, right_index=True, how='right')
    r = r.dropna()
    r = r.astype(int)
    actual_clusters = r.iloc[:,0].tolist()
    predicted_clusters = r.iloc[:,1].tolist()

    tp_plus_fp = comb(np.bincount(actual_clusters), 2).sum()
    tp_plus_fn = comb(np.bincount(predicted_clusters), 2).sum()
    A = np.c_[(actual_clusters, predicted_clusters)]
    tp = sum(comb(np.bincount(A[A[:, 0] == i, 1]), 2).sum()
             for i in set(actual_clusters))
    fp = tp_plus_fp - tp
    fn = tp_plus_fn - tp
    tn = comb(len(A), 2) - tp - fp - fn
    ri =  (tp + tn) / (tp + fp + fn + tn)
    return ri
```

Find the RI index of the clustering using the train and test comparison

```
get_rand_index_score(train_df, test_df)
```

Save the table obtained in previous step to file system. This table would be used to compare each flow with the past to identify if its threat or not.

```
#Save train result to file
train_tag_filename = 'ip_cluster_tag_train'
tag_file = os.path.join(base_path,train_tag_filename)
train_df.to_csv(tag_file)
#Save test result to file
test_tag_filename = 'ip_cluster_tag_test'
tag_file = os.path.join(base_path,test_tag_filename)
test_df.to_csv(tag_file)
```

Anomaly Detection Using One Class SVM

```
#Create feature vector corrosponding to each cluster.
#This feature vector would be used to define boundray using One Class SVM for the cluster.
def get_clusters_feature_vectors(cluster_path):
    cluster_dict = dict()
    filenames = sorted(glob.glob(os.path.join(cluster_path,'*')),  key=os.path.getmtime)
    first = True
    for filename in filenames:
        if first:
            df = pd.read_csv(filename, index_col=0)
            first = False
        else:
            temp_df = pd.read_csv(filename, index_col=0)
            df = df.append(temp_df)
            #print(df)
            break
    df = df.reset_index().set_index(['cluster','ip'])
    clusters = df.index.get_level_values(0).unique()
    for c in clusters:
        cluster_dict[c] = df.loc[c].values
    return cluster_dict

def plot_outlier_detecton(X, ax, title):
    if X.shape[1] > 2:
        reduced_X = PCA(n_components=2).fit_transform(X)
    else:
        reduced_X = X

    clf = svm.OneClassSVM(nu=0.01, kernel="rbf", gamma=0.1)
    clf.fit(reduced_X)
    xx, yy = np.meshgrid(np.linspace(-5, 5, 500), np.linspace(-5, 5, 500))

    # plot the levels lines and the points
    Z = clf.decision_function(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    ax.set_title(title)
    ax.contourf(xx, yy, Z, levels=np.linspace(Z.min(), 0, 7), cmap=plt.cm.PuBu)
    a = ax.contour(xx, yy, Z, levels=[0], linewidths=2, colors='darkred')
    ax.contourf(xx, yy, Z, levels=[0, Z.max()], colors='palevioletred')

    s = 40
    b1 = ax.scatter(reduced_X[:, 0], reduced_X[:, 1], c='white', s=s, edgecolors='k')
    #ax.set_axis('tight')
    ax.set_xlim((-5, 5))
    ax.set_ylim((-5, 5))
    ax.legend([a.collections[0], b1],
              ["learned frontier", "training observations"],
             loc="upper left")
```

Get cluster data for training One Class SVM model

```
cluster_feature_dict = get_clusters_feature_vectors(cluster_train_path)
```

```
fig = plt.figure(figsize=(10, 8), dpi=80, facecolor='w', edgecolor='k')
row_count = math.ceil(len(cluster_feature_dict)/2)
for i, (cluster, X) in enumerate(cluster_feature_dict.items(), 1):
    #Create scaling
    X_trans = preprocessing.StandardScaler().fit_transform(X)
    ax = fig.add_subplot(row_count, 2, i)
    plt.suptitle('One Class SVM (with PCA-reduced data)', fontsize=16)
    title = 'Cluster:'+str(cluster)
    if (X_trans.shape[0] >= X_trans.shape[1]):
        plot_outlier_detecton(X_trans, ax, title)
plt.savefig('one-class-SVM.png')
```

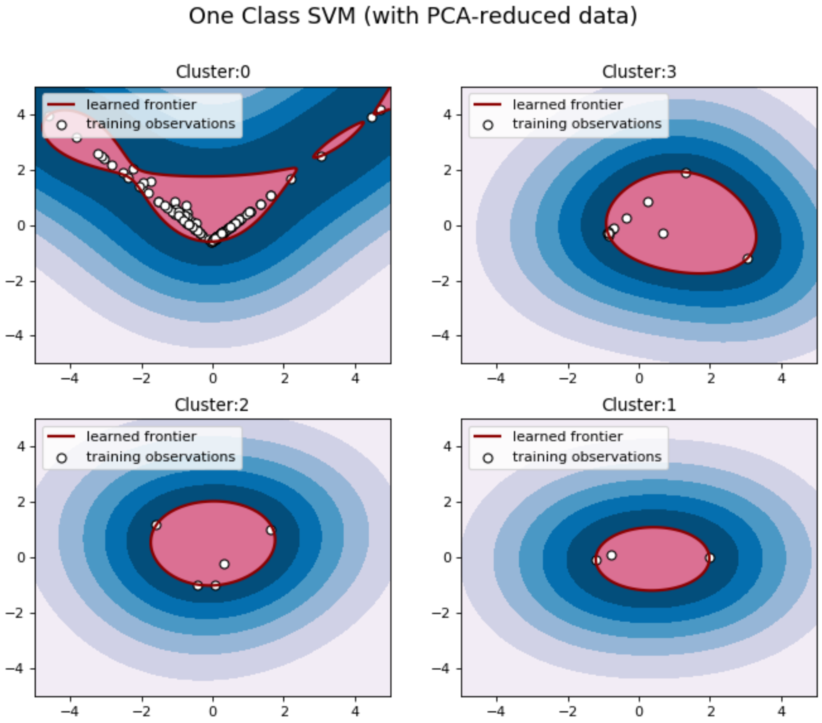

Figure 3: One Class SVM

```
def one_class_svm(feature_vector, cluster):
    X_train = feature_vector
    #Get scaler
    scaler = preprocessing.StandardScaler().fit(X_train)
    #Transform Traning data
    X_trans = scaler.transform(X_train)
    # fit the model
    clf = svm.OneClassSVM(nu=0.01, kernel="rbf", gamma=0.1)
    clf.fit(X_trans)
    clf.name = 'svm'
    return clf, scaler
```

Run One Class SVM

```
clf_dict = dict()
scaler_dict = dict()
for cluster, df in cluster_feature_dict.items():
    clf_dict[cluster], scaler_dict[cluster] = one_class_svm(df, cluster)
```

Clustering the data captured during attack

```
attack_sample_dir_name = 'attack_samp'
attack_sample_path = os.path.join(base_path, attack_sample_dir_name)
attack_sample_train_path = os.path.join(base_path, attack_sample_dir_name+'_train')
os.makedirs(attack_sample_train_path, exist_ok=True)
attack_cluster_path = os.path.join(attack_sample_train_path, 'cluster')
os.makedirs(attack_cluster_path, exist_ok=True)

files = sorted(glob.glob(os.path.join(attack_sample_path,'*')),  key=os.path.getmtime)
merge_count = np.genfromtxt(os.path.join(base_path,'merge_count'))

#Get centroids and feature list
centroids, features = read_centroid_features(centroid_filename, features_filename)

attack_dfs = merge_files(files, merge_count, features)

#Cluster all the samples and store them
centroids, features = read_centroid_features(centroid_filename, features_filename)
for i, attack_df in enumerate(attack_dfs):
    clustered_df = kmeans_clustering(attack_df, centroids)
    clustered_df.to_csv(os.path.join(attack_cluster_path,str(i+1)))

#Read clustered attack file
files = sorted(glob.glob(os.path.join(attack_cluster_path,'*')),  key=os.path.getmtime)
df = pd.read_csv(files[0], index_col=0)

train_tag_filename = 'ip_cluster_tag_train'
train_tag_file = os.path.join(base_path,train_tag_filename)
train_df = pd.read_csv(train_tag_file, index_col=0)
```

Detect an attack

```
bad_ip_df = pd.DataFrame(columns=['ip', 'packet_count', 'cluster'])
for ip, row in df.iterrows():
    feature_row = row[:-1]
    cluster_value = row[-1]
    #cluster test
    if ip in train_df.index:
        packet_count = feature_row.sum()
        train_cluster = train_df.loc[ip]['cluster']
        #Compare each ip data with the train cluster
        if(cluster_value == train_cluster):
            #SVM test: Check if its inside the boundary using one class svm
            X_test_tran = scaler_dict[cluster_value].transform([feature_row.values])
            prediction = clf_dict[cluster_value].predict(X_test_tran)
            if prediction == - 1:
                s = pd.Series([ip, packet_count, cluster_value], index=bad_ip_df.columns)
                bad_ip_df = bad_ip_df.append(s, ignore_index=True)
        else:
            s = pd.Series([ip, packet_count, cluster_value], index=bad_ip_df.columns)
            bad_ip_df = bad_ip_df.append(s, ignore_index=True)

bad_ip_df = bad_ip_df.set_index('ip')
print(bad_ip_df)
```