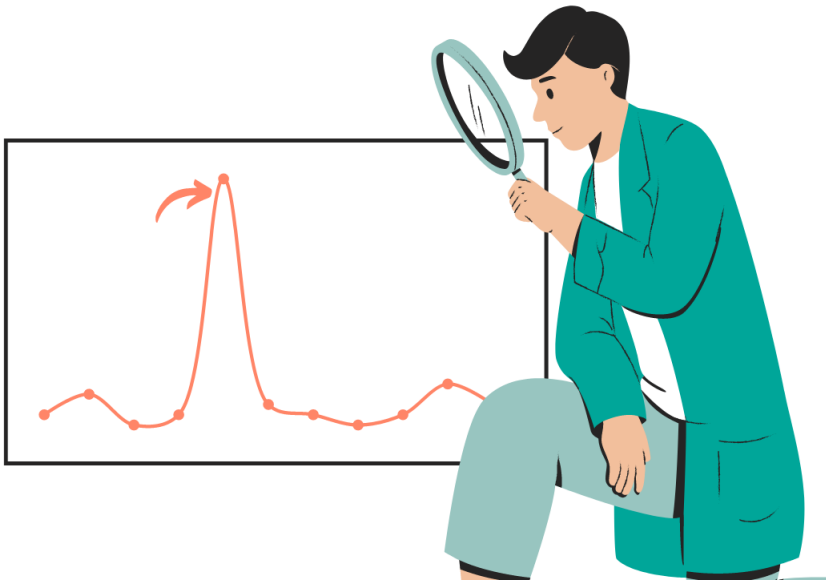
Pattern Recognition

Assignment 2

Anomalies Detection



Prepared by:

Name	ID
Youssef Samuel Nachaat Labib	6978
Youssef Amr Ismail Othman	6913
Arsany Mousa Fathy Rezk	6927

1. kmeans

Function Purpose: Perform k-means clustering on a given data matrix.

Input Parameters:

- **Data_Matrix**: A numpy array representing the input data matrix.
- **k**: An integer representing the number of clusters to form.
- **max_iterations**: An integer specifying the maximum number of iterations.
- **threshold**: A float representing the threshold for convergence.
- **rand_restart**: An integer specifying the number of times the algorithm is restarted with different initial centroids. This is done due to fact that k-means may stuck at local minimum

Output:

- Returns the final centroids and cluster assignments as numpy arrays. Cluster assignments is an array with size equal to the size of the data, and for each data sample a value is stored representing the cluster assigned to it.

Algorithm/Logic:

- Initialize the minimum sum of squared errors (SSE) and the best clusters and centroids.
- Iterate for a given number of random restarts:
    - Randomly initialize centroids.
    - Iterate for a given number of maximum iterations:
        - Calculate the Euclidean distance between data points and centroids.
        - Assign data points to clusters based on the minimum distance. **O(knd)**
        - Update centroids by calculating the mean of data points in each cluster. **O(nd)**
        - Calculate SSE and update the best clusters and centroids if necessary.
        - Check for convergence using a threshold on centroid updates.
- Remove empty clusters. Any cluster, where no data assigned to it, should be removed. So, the number of centroids returned may be less than the required number of clusters.
- Return the final centroids and cluster assignments.

Related Functions:

- **remove_empty_clusters**: A function that removes empty clusters from the final centroids and cluster assignments.

2. remove_empty_clusters

Function Purpose: Remove empty clusters from the final centroids and update cluster assignments accordingly.

Input Parameters:

- **Data_Matrix**: A numpy array representing the input data matrix.
- **best_centroids**: A numpy array representing the centroids of the best clusters.
- **data_cluster**: A numpy array representing the cluster assignments for each data point.

Output:

- Returns the updated centroids and cluster assignments as numpy arrays.

Algorithm/Logic:

- Initialize a count of data points in each cluster.
- Iterate through the cluster assignments and update the count.
- Determine the number of non-empty clusters.
- Create a new numpy array to store the centroids of non-empty clusters.
- Iterate through the counts and append centroids of non-empty clusters to the new array.
- Calculate the Euclidean distance between data points and updated centroids.
- Update cluster assignments based on the minimum distance.
- Return the updated centroids and cluster assignments.

Note: This function is used in the **kmeans** function to remove empty clusters from the final centroids and cluster assignments before returning the results.

3. classify_centroid

Function Purpose: Classify the centroids based on majority class of their corresponding data cluster.

Input Parameters:

- **centroids**: A numpy array representing the centroids of the clusters returned from the kmeans function.
- **data_cluster**: A numpy array representing the cluster assignments for each data point.
- **training_labels**: A numpy array representing the ground truth labels for the training data.

Output:

- Returns a numpy array **cluster_identification** containing the majority class label for each centroid.

Algorithm/Logic:

- Initialize an array **cluster_identification** to store the majority class label for each centroid.
- Get the unique cluster labels and their counts from **data_cluster**.
- For each unique cluster label, extract the indices of data points belonging to that cluster.
- Create a binary mask **cluster_mask** to mark the data points belonging to the current cluster.
- Multiply the **cluster_mask** with **training_labels** to get the ground truth labels for the data points in the current cluster.
- Get the unique class labels and their counts from the ground truth labels of the current cluster.
- Iterate through the unique class labels and find the majority class label with the highest count.
- Store the majority class label in **cluster_identification** for the current cluster.
- Repeat the above steps for all unique cluster labels.
- Return the **cluster_identification** array containing the majority class label for each centroid.

4. assign_centroids

Function Purpose: Assign the testing data points to the nearest centroid and calculate clustering statistics.

Input Parameters:

- **testing_set**: A numpy array representing the testing data points.
- **centroids**: A numpy array representing the centroids of the clusters.
- **cluster_identification**: A numpy array representing the majority class labels for each centroid, obtained from the **classify_centroid** function.
- **label_list**: A list of class labels.

Output:

- **cluster_index**: A numpy array containing the cluster assignments for each testing data point.
- **clustering_statistics**: A numpy array containing the number of data samples detected for each class label. For example, the number of data points classified as ‘normal’.

Algorithm/Logic:

- Initialize an empty list **cluster_index** to store the cluster assignments for each testing data point.
- Initialize a numpy array **clustering_statistics** with zeros to store the clustering statistics for each class.

- Compute the Euclidean distance between each testing data point and the centroids using **cdist** function.
- For each testing data point, find the index of the centroid with the minimum Euclidean distance.
- Append the index of the nearest centroid to **cluster_index**, representing the cluster assignment for the current testing data point.
- Update the **clustering_statistics** by incrementing the count for the cluster corresponding to the majority class label of the nearest centroid.
- Repeat the above steps for all testing data points.

5. Normalized_cut

Function Purpose: Perform spectral clustering using normalized cut algorithm on the given data.

Input Parameters:

- **Data_matrix**: A numpy array representing the data matrix.
- **k**: An integer representing the number of clusters to form.
- **nearest_K**: An integer representing the number of nearest neighbors to consider for constructing the similarity matrix.

Output:

- **best_centroids**: A numpy array representing the centroids of the clusters obtained from k-means algorithm applied on the spectral embeddings.
- **data_cluster**: A numpy array representing the cluster assignments for each data point.

Algorithm/Logic:

- Initialize an empty similarity matrix **similarity_matrix** and a degree matrix **Degree_matrix**.
- Compute the Euclidean distance between each data point and all other data points using **cdist** function.
- For each data point, find the indices of the nearest **nearest_K** neighbors based on the sorted distances.
- Update the similarity matrix by setting the corresponding entries to 1, indicating that the data points are similar to their nearest neighbors.
- Set the diagonal entries of the similarity matrix to 0, as each data point is not similar to itself.
- Compute the degree matrix by summing the rows of the similarity matrix.
- Update the similarity matrix by adding its transpose to make it symmetric.
- Compute the Laplacian matrix **L**. (degree - similarity)
- Compute the normalized Laplacian matrix.

- Compute the eigenvalues and eigenvectors of **Norm_L** using **np.linalg.eig** function.
- Sort the eigenvalues and corresponding eigenvectors in ascending order based on the eigenvalues.
- Extract the eigenvectors corresponding to the lowest **k** eigenvalues and normalize.
- Apply k-means algorithm on the normalized eigenvectors.
- Return the centroids obtained from k-means and the cluster assignments.

6. assignment_statistics

Function Purpose: Compute the statistics of cluster assignments based on the cluster identification and cluster assignments obtained from the spectral clustering.

Input Parameters:

- **data_cluster_Ncut**: A numpy array representing the cluster assignments obtained from the spectral clustering algorithm.
- **cluster_identification**: A numpy array representing the cluster identification or ground truth labels for each data point.
- **label_list**: A list representing the unique labels or cluster identifications present in the ground truth data.

Output:

- Returns a numpy array representing the clustering statistics, i.e., the count of data points assigned to each cluster.

Algorithm/Logic:

- Initialize an empty numpy array **clustering_statistics** with zeros, of length equal to the number of unique labels in the ground truth data.
- For each data point in **data_cluster_Ncut**:
    - Get the cluster assignment for the data point from **data_cluster_Ncut**.
    - Get the cluster identification or ground truth label for the data point from **cluster_identification**.
    - Update the corresponding entry in **clustering_statistics** by incrementing the count.
- Return the **clustering_statistics** representing the count of data points assigned to each cluster.

7. DBSCAN_clustring

This code implements the Density-Based Spatial Clustering of Applications with Noise (DBSCAN) algorithm, which is a popular clustering algorithm used in machine learning and data analysis.

The DBSCAN algorithm is designed to cluster together data points that are close to each other in a high-density region, while also identifying and labeling outliers or noise points.

The function **dbscan()** takes four parameters as input:

- **data**: a two-dimensional numpy array containing the data points to be clustered.
- **eps**: a float value representing the maximum distance between two points for them to be considered neighbors.
- **min_pts**: an integer value representing the minimum number of points required for a point to be considered a core point.
- **num_clusters**: an integer value representing the desired number of clusters.

The function starts by initializing two arrays:

- **visited**: a boolean array of the same length as the input data, used to keep track of which data points have been visited during the clustering process.
- **cluster_labels**: an integer array of the same length as the input data, initially set to all zeros. This array will be used to label each data point with its cluster membership.

Next, the function calculates the neighbors of each point using the Euclidean distance metric. For each data point, it finds all other points within a distance of **eps**, and stores the indices of these points in a list called **neighbors**.

The function then iterates over all unvisited points in the input data, and for each unvisited point, it checks if it is a core point (i.e., if it has at least **min_pts** neighbors within a distance of **eps**). If the point is a core point, it is assigned a new cluster label, and all of its neighbors are added to a list called **current_neighbors**. The function then expands the cluster by iterating over **current_neighbors**, finding their neighbors, and adding them to **current_neighbors** if they are also core points. Each new neighbor that is added to **current_neighbors** is also assigned the same cluster label as the original core point. This process continues until no more core points can be added to the cluster.

If a point is not a core point, it is assigned a special noise label (-1) to indicate that it does not belong to any cluster.

After all points have been visited and assigned a cluster label, any remaining unassigned points are assigned to the noise label.

Finally, the function checks if the number of clusters found is greater than the desired number of clusters (**num_clusters**). If so, it repeatedly merges the smallest cluster with its nearest neighbor (i.e., the cluster with the fewest points that is closest to another cluster), until the desired number of clusters is reached.

The function returns the **cluster_labels** array, which contains the cluster labels for each data point in the input data.

8. agglomerative_clustering

Function Purpose: Perform agglomerative clustering on input data to group data points into k clusters.

Input Parameters:

- **data**: A numpy array representing the input data points to be clustered.
- **k**: An integer representing the desired number of clusters.

Output:

- **clusters**: A list of lists, where each list represents a cluster containing indices of data points belonging to that cluster.
- **cluster_index_for_each_point**: A numpy array of length equal to the number of data points, where each entry represents the cluster index assigned to the corresponding data point in the input data.

Algorithm/Logic:

- Initialize **clusters** as a list of individual data points, where each data point is initially considered as a separate cluster.
- Calculate the pairwise distances between all data points in the input data using Euclidean distance and store it in the **distances** numpy array.
- While the number of clusters is greater than k:
    - Find the closest pair of clusters based on the average distance between all pairs of data points in the two clusters.
    - Merge the closest pair of clusters by appending the data points of one cluster to the other cluster, and then delete the second cluster from the list of clusters.
- Assign a cluster index to each data point in the input data based on the final merged clusters.
- Return the **clusters** and **cluster_index_for_each_point** representing the final clusters and cluster index for each data point, respectively.

9. Evaluation Functions

Input Parameters:

- **labels**: A numpy array representing the cluster labels for each data point.
- **ground_truth**: A numpy array representing the ground truth labels for each data point.

a. Conditional Entropy

Returns a scalar value representing the conditional entropy between the cluster labels and ground truth labels.

Algorithm/Logic:

- Get the unique values and their counts in the **labels** numpy array.
- Initialize **conditional_ent** as 0.
- For each unique cluster label:
    - Get the cluster number and the count of data points in that cluster.
    - Create a mask for the cluster by setting 1 for data points with the cluster label, and 0 for others.
    - Multiply the cluster mask with the ground truth labels to get the ground truth labels for the data points in that cluster.
    - Get the unique values and their counts in the ground truth labels for the cluster.
    - Initialize **cond_ent_clust** as 0.
    - For each unique ground truth label in the cluster:
        - Calculate the entropy contribution of the ground truth label in the cluster using the formula:
            - **ent = (counts_clust[k] / cluster_sum) * math.log2(counts_clust[k] / cluster_sum)**
        - Add the entropy contribution to **cond_ent_clust**.
    - Update **conditional_ent** by adding the weighted average of **cond_ent_clust** based on the count of data points in the cluster.
- Return the **conditional_ent** as the final conditional entropy between the cluster labels and ground truth labels.

b. Purity

Returns a scalar value representing the purity between the cluster labels and ground truth labels.

Algorithm/Logic:

- Get the unique values and their counts in the **labels** numpy array.
- Initialize **purity** as 0.
- For each unique cluster label:
    - Get the cluster number and the count of data points in that cluster.

    - Create a mask for the cluster by setting 1 for data points with the cluster label, and 0 for others.
    - Multiply the cluster mask with the ground truth labels to get the ground truth labels for the data points in that cluster.
    - Get the unique values and their counts in the ground truth labels for the cluster.
    - Initialize **majority** as -1.
    - For each unique ground truth label in the cluster:
        - If the count of the ground truth label is greater than **majority**, update **majority** with the count.
    - Calculate the cluster purity as **majority** divided by the count of data points in the cluster.
    - Update **purity** by adding the weighted average of cluster purity based on the count of data points in the cluster.
- Return the **purity** as the final purity between the cluster labels and ground truth labels.

c. F-measure

Returns a scalar value representing the F-measure between the cluster labels and ground truth labels.

Algorithm/Logic:

- Get the unique values and their counts in the **labels** numpy array.
- Initialize **sigma_f** as 0.
- Get the total number of clusters.
- For each unique cluster label:
    - Get the cluster number and the count of data points in that cluster.
    - Create a mask for the cluster by setting 1 for data points with the cluster label, and 0 for others.
    - Multiply the cluster mask with the ground truth labels to get the ground truth labels for the data points in that cluster.
    - Get the unique values and their counts in the ground truth labels for the cluster.
    - Initialize **majority** as -1.
    - For each unique ground truth label in the cluster:
        - If the count of the ground truth label is greater than **majority**, update **majority** with the count, and store the index of the ground truth label.
    - Calculate the precision as **majority** divided by the count of data points in the cluster.
    - Get the total occurrence of the ground truth label with the maximum count in the cluster.
    - Calculate the recall as **majority** divided by the total occurrence of the ground truth label.
    - Calculate the F-measure as the harmonic mean of precision and recall.
    - Update **sigma_f** by adding the F-measure.

- Return the average F-measure by dividing **sigma_f** by the total number of clusters.

d. pairwise_measures

Returns two scalar values: Jaccard coefficient and Rand index as tuple.

Algorithm/Logic:

- Get the unique values and their counts in the **labels** numpy array.
- Initialize true positive (TP), false positive (FP), false negative (FN), and true negative (TN) counts as 0.
- Initialize an empty list **cluster_counts** to store counts of ground truth labels for each cluster.
- For each unique cluster label:
    - Get the cluster number and the count of data points in that cluster.
    - Create a mask for the cluster by setting 1 for data points with the cluster label, and 0 for others.
    - Multiply the cluster mask with the ground truth labels to get the ground truth labels for the data points in that cluster.
    - Get the unique values and their counts in the ground truth labels for the cluster.
    - Create a matrix of unique ground truth labels and their counts for the cluster.
    - Append the matrix to the **cluster_counts** list.
    - For each unique ground truth label in the cluster:
        - Calculate the number of true positive pairs as the combination of counts_clust[k] choose 2 (math.comb) since we want to calculate the number of pairs of data points in the cluster that share the same ground truth label.
        - Calculate the number of false positive pairs as counts_clust[k] times (cluster_sum - counts_clust[k]) since we want to calculate the number of pairs of data points in the cluster that do not share the same ground truth label.
        - Update TP and FP counts accordingly.
- Convert **cluster_counts** list to a numpy array for further processing.
- For each unique pair of ground truth labels:
    - If the ground truth labels are the same (excluding diagonal elements), update FN count as the product of their counts.
    - If the ground truth labels are different and the second ground truth label is not 0 (excluding diagonal elements), update TN count as the product of their counts.
- Divide FN and TN counts by 2 to avoid double counting of pairs.
- Calculate Jaccard coefficient as TP divided by the sum of FN, TP, and FP counts.
- Calculate Rand index as the sum of TP and TN counts divided by the sum of FN, TN, TP, and FP counts.
- Return Jaccard coefficient and Rand index as a tuple.

e. Beta-CV

returns the calculated Beta CV value as a float.

Algorithm/Logic:

- The function first maps the input labels to a unique set of labels, starting from 1, to ensure consistency.
- It then calculates the total number of intra-cluster pairs (N_in) and inter-cluster pairs (N_out) based on the counts of each unique label.
- Next, it computes a proximity matrix to measure the distances between data points within and across clusters.
- Using the proximity matrix, it calculates the within-cluster sum of distances (W_in) and the between-cluster sum of distances (W_out).
- Finally, it computes the Beta CV as the ratio of W_in divided by N_in, divided by W_out divided by N_out.

f. N-cut Measure

returns the calculated Beta CV value as a float.

Algorithm/Logic:

- The function first maps the input labels to a unique set of labels, starting from 1, to ensure consistency.
- It then calculates the total number of intra-cluster pairs (N_in) and inter-cluster pairs (N_out) based on the counts of each unique label.
- Next, it computes a proximity matrix to measure the distances between data points within and across clusters.
- Using the proximity matrix, it calculates the within-cluster sum of distances (W_in) and the between-cluster sum of distances (W_out).
- Finally, it computes the Beta CV as the ratio of W_in divided by N_in, divided by W_out divided by N_out.

10.Loading Function

Function Purpose: The **load()** function is used to load and preprocess data from two gzip-compressed CSV files (**data_path** and **test_path**). It performs one-hot encoding on the categorical features, applies label encoding to the label column, and returns the preprocessed data, label column, and label list for further analysis.

Input Parameters:

- **data_path**: A string representing the file path of the first gzip-compressed CSV file containing the data.
- **test_path**: A string representing the file path of the second gzip-compressed CSV file containing the test data.
- **limit**: A boolean flag indicating whether to limit the number of rows read from the first data file to 5% of the total rows or not. If **limit** is set to **True**, only 5% of the rows will be read from the first data file; otherwise, all rows from both data files will be read.

Function Logic:

- The function reads the data from the first gzip-compressed CSV file (**data_path**) into a pandas DataFrame (**df1**).
- If **limit** is set to **True**, it calculates the number of rows to be read as 5% of the total rows in **df1** and stores it in the variable **num_rows**. Otherwise, it reads the data from the second gzip-compressed CSV file (**test_path**) into another pandas DataFrame (**df2**), and concatenates both dataframes along the rows axis to create a combined DataFrame (**df**).
- The function identifies the categorical and numeric columns in the DataFrame (**df**) and stores them in separate lists (**categorical_cols** and **numeric_cols**).
- It then extracts the numeric data from **df** using the numeric column indices (**numeric_cols**) and stores it in a numpy array (**numeric_data**).
- The categorical data is extracted from **df** using the categorical column indices (**categorical_cols**) and stored in another DataFrame (**categorical_data**).
- One-hot encoding is applied to the categorical features in **categorical_data** using **pd.get_dummies()** function, and the resulting one-hot encoded data is stored in another DataFrame (**one_hot_data**).
- Label encoding is applied to the label column (column 41) in **df** using **LabelEncoder()** from **sklearn.preprocessing** and the encoded label column is stored in a new column named 'col41_encoded'.
- The function creates a list of labels by removing the last character from each label in the encoded label column, and stores it in the variable **label_list**.
- The label column (**col41_encoded**) is converted to a numpy array (**label_column**) and each label is incremented by 1 to ensure consistency with the unique labels returned by **LabelEncoder()**.
- Finally, the numeric data, one-hot encoded data, and label column are concatenated along the columns axis to create the preprocessed data (**data**) and the preprocessed data (**data**), label

column (**label_column**), label list (**label_list**), and total number of rows (**num_rows**) are returned as outputs.

Output or Return Value:

The function returns four values:

- **data**: A numpy array representing the preprocessed data, where each row represents a data point and each column represents a feature.
- **label_column**: A numpy array representing the label column of the data, where each element represents the label of the corresponding data point.
- **label_list**: A list of unique labels extracted from the label column, with the last character removed.
- **num_rows**: An integer representing the total number of rows in the

anomaly-detection-code

April 15, 2023

{“username”:“arsanymousafathy”,“key”:“7dad8a66f62a0da359c47074cd914b32”}

[ ]:
```
!pip install opendatasets
import gzip
import opendatasets as od
#download given dataset for the original problem
od.download(
    "https://www.kaggle.com/datasets/galaxyh/kdd-cup-1999-data?
 ↪resource=download")
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-
wheels/public/simple/
Requirement already satisfied: opendatasets in /usr/local/lib/python3.9/dist-
packages (0.1.22)
Requirement already satisfied: click in /usr/local/lib/python3.9/dist-packages
(from opendatasets) (8.1.3)
Requirement already satisfied: kaggle in /usr/local/lib/python3.9/dist-packages
(from opendatasets) (1.5.13)
Requirement already satisfied: tqdm in /usr/local/lib/python3.9/dist-packages
(from opendatasets) (4.65.0)
Requirement already satisfied: python-dateutil in /usr/local/lib/python3.9/dist-
packages (from kaggle->opendatasets) (2.8.2)
Requirement already satisfied: certifi in /usr/local/lib/python3.9/dist-packages
(from kaggle->opendatasets) (2022.12.7)
Requirement already satisfied: requests in /usr/local/lib/python3.9/dist-
packages (from kaggle->opendatasets) (2.27.1)
Requirement already satisfied: python-slugify in /usr/local/lib/python3.9/dist-
packages (from kaggle->opendatasets) (8.0.1)
Requirement already satisfied: urllib3 in /usr/local/lib/python3.9/dist-packages
(from kaggle->opendatasets) (1.26.15)
Requirement already satisfied: six>=1.10 in /usr/local/lib/python3.9/dist-
packages (from kaggle->opendatasets) (1.16.0)
Requirement already satisfied: text-unidecode>=1.3 in
/usr/local/lib/python3.9/dist-packages (from python-
slugify->kaggle->opendatasets) (1.3)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.9/dist-
packages (from requests->kaggle->opendatasets) (3.4)
Requirement already satisfied: charset-normalizer~=2.0.0 in
```

```
/usr/local/lib/python3.9/dist-packages (from requests->kaggle->opendatasets)
(2.0.12)
Skipping, found downloaded files in "./kdd-cup-1999-data" (use force=True to
force download)
```

[16]:
```
import numpy as np
import random
import copy
from scipy.spatial.distance import cdist
import math
from sklearn.model_selection import train_test_split
import pandas as pd
from sklearn.preprocessing import LabelEncoder
np.set_printoptions(threshold=np.inf)
```

#Kmeans

##Classify Each Centroid

[1]:
```
def classify_centroid(centroids, data_cluster, training_labels):
  num_clusters = len(centroids)
  cluster_identification = np.zeros([num_clusters, 1])
  unique, counts = np.unique(data_cluster[:], return_counts=True)
  for j in range(len(unique)):
    cluster_number = unique[j]
    cluster_sum = counts[j]
    indices = np.where(data_cluster[:] == cluster_number)
    indices = np.array(indices)
    indices=indices[0,:]
    cluster_mask = np.zeros(data_cluster.shape[0])
    cluster_mask[indices] = 1
    ground_truth_clust = cluster_mask * training_labels
    unique_clust, counts_clust = np.unique(ground_truth_clust,␣
 ↪return_counts=True)
    majority_count = -1
    majority_index = -1
    for k in range(len(unique_clust)):
      if unique_clust[k] > 0:
        if counts_clust[k] > majority_count:
          majority_count = counts_clust[k]
          majority_index = unique_clust[k]
    cluster_identification[j] = majority_index  #Cluster identification to␣
 ↪contain the class of the majority of the cluster
  return cluster_identification
```

##Assign each test sample to a centroid

[2]:
```
def assign_centroids(testing_set, centroids, cluster_identification,␣
↪label_list):
  cluster_index = []
  clustering_statistics = np.zeros([len(label_list)])
  euc_dist = cdist(testing_set, centroids, metric = 'euclidean')
  for i in range(len(testing_set)):
    min_index = np.argmin(euc_dist[i])
    cluster_index.append(min_index+1)
    clustering_statistics[int(cluster_identification[min_index]-1)] =␣
↪clustering_statistics[int(cluster_identification[min_index]-1)] + 1
  return np.array(cluster_index).astype(int), clustering_statistics
```

##Remove empty clusters

[3]:
```
def remove_empty_clusters(Data_Matrix, best_centroids, data_cluster):
  counts = np.zeros([len(best_centroids), 1])
  for i in range(len(data_cluster)):
    counts[int(data_cluster[i]-1), 0] = counts[int(data_cluster[i]-1), 0] + 1
  len_centroids = 0
  for i in range(len(counts)):
    if counts[i] != 0:
      len_centroids = len_centroids + 1
  centroids_kmeans = np.zeros([len_centroids, best_centroids.shape[1]])
  c = 0
  for i in range(len(counts)):
    if counts[i] != 0:
      centroids_kmeans[c] = best_centroids[i]
      c = c+1

  # Get the euclidean distance between each record and the current centroids
  euc_dist = cdist(Data_Matrix, centroids_kmeans, metric = 'euclidean')
  # Cluster Assignment
  for i in range(Data_Matrix.shape[0]):
    min_index = np.argmin(euc_dist[i])
    data_cluster[i] = min_index + 1

  return centroids_kmeans, data_cluster
```

##Kmeans function

[4]:
```
def kmeans(Data_Matrix, k, max_iterations, threshold, rand_restart):
  min_sse = float('inf')
  best_clusters =  [[] for _ in range(k)]
  best_centroids = np.zeros([k, Data_Matrix.shape[1]])

  for r in range(rand_restart):
    data_size = len(Data_Matrix)
```

```
    centroids_indices = random.sample(range(data_size), k)
    centroids = [Data_Matrix[i] for i in centroids_indices]
    old_centroids = np.array(centroids).astype(float)
    new_centroids = np.array(centroids).astype(float)
    clusters = [[] for _ in range(k)]

    for m in range(max_iterations):
      # Get the euclidean distance between each record and the current centroids
      euc_dist = cdist(Data_Matrix, old_centroids, metric = 'euclidean')
      # Cluster Assignment
      for i in range(data_size):
        min_index = np.argmin(euc_dist[i])
        clusters[min_index].append(i)
      sse_centroids = 0

      SSE = 0.0
      # Calculate SSE
      for i in range(k):
        for j in range(len(clusters[i])):
          SSE = SSE + euc_dist[clusters[i][j], i]
      if SSE < min_sse:
        min_sse = SSE
        best_clusters = copy.deepcopy(clusters)
        best_centroids = copy.deepcopy(old_centroids)
      # Centroid Update
      for i in range(k):
        if len(clusters[i]) != 0:
          new_centroids[i] = np.mean(Data_Matrix[clusters[i]], axis=0)
          sse_centroids = sse_centroids + np.linalg.norm(old_centroids[i] -␣
↪new_centroids[i])**2
          old_centroids[i] = new_centroids[i]
      clusters = [[] for _ in range(k)]
      if sse_centroids <= threshold:
        break
  data_cluster = np.zeros([Data_Matrix.shape[0], 1])
  for i in range(k):
    for j in range(len(best_clusters[i])):
      data_cluster[best_clusters[i][j], 0] = i+1


  centroids_kmeans, data_cluster = remove_empty_clusters(Data_Matrix,␣
↪best_centroids, data_cluster)

  return centroids_kmeans, data_cluster.astype(int)
```

#Normalized Cut

##Assignment statistics

[5]:
```
def assignment_statistics(data_cluster_Ncut, cluster_identification,␣
↪label_list):
 clustering_statistics = np.zeros([len(label_list)])
 for i in range(len(data_cluster_Ncut)):
  ␣
↪clustering_statistics[int(cluster_identification[data_cluster_Ncut[i]-1]-1)]␣
↪=␣
↪clustering_statistics[int(cluster_identification[data_cluster_Ncut[i]-1]-1)]␣
↪+ 1
 return clustering_statistics
```

##Normalized Cut function

[6]:
```
def Normalized_cut(Data_matrix, k, nearest_K):
  similarity_matrix = np.zeros([Data_matrix.shape[0], Data_matrix.shape[0]])
  Degree_matrix = np.zeros([Data_matrix.shape[0], Data_matrix.shape[0]])
  euc_distance = cdist(Data_matrix, Data_matrix, metric='euclidean')

  for i in range(Data_matrix.shape[0]):
    sorted_distances = np.argsort(euc_distance[i])
    sorted_distances = sorted_distances[1:nearest_K+1]
    for j in range(len(sorted_distances)):
      similarity_matrix[i, sorted_distances[j]] = 1

  for i in range(Data_matrix.shape[0]):
    similarity_matrix[j, j] = 0
    Degree_matrix[i,i] = sum(similarity_matrix[i,:])
  similarity_matrix = similarity_matrix + similarity_matrix.transpose()

  L =  Degree_matrix - similarity_matrix
  Degree_inverse = np.linalg.inv(Degree_matrix)
  Norm_L = Degree_inverse @ L

  eig_values, eig_vectors = np.linalg.eig(Norm_L)
  # The imaginary part of the eigenvectors indicates the orientation of the␣
↪vector in the complex plane.
  eig_values = np.real(eig_values)
  eig_vectors = np.real(eig_vectors)
  indices = eig_values.argsort()
  eig_values_vector = eig_values[indices]
  eig_vectors = eig_vectors[:,indices]
  # Normalization of the eigenvectors corresponding to the lowest k eigenvalue
  eig_vectors = eig_vectors[:, 0:k]
  for j in range(eig_vectors.shape[0]):
    norm = np.linalg.norm(eig_vectors[j])
    eig_vectors[j] = eig_vectors[j] / norm
  max_iterations = 30
```

```
    threshold = 0.01
    rand_restart = 5
    best_centroids, data_cluster = kmeans(eig_vectors, k, max_iterations,␣
    ↪threshold, rand_restart)
    return best_centroids, data_cluster
```

#DBSCAN Clustring

[32]:
```
def dbscan(data, eps, min_pts, num_clusters):
    # initialize visited and cluster labels
    visited = np.zeros(len(data), dtype=bool)
    cluster_labels = np.zeros(len(data), dtype=int)

    # find neighbors for each point
    neighbors = []
    for i in range(len(data)):
        dist = np.linalg.norm(data - data[i], axis=1)
        neighbors.append(np.where(dist <= eps)[0])

    # iterate over unvisited points
    cluster = 0
    for i in range(len(data)):
        if not visited[i]:
            visited[i] = True

            # find neighbors of current point
            current_neighbors = neighbors[i]

            # check if current point is a core point
            if len(current_neighbors) >= min_pts:
                cluster += 1
                cluster_labels[i] = cluster

                # expand cluster
                j = 0
                while j < len(current_neighbors):
                    neighbor = current_neighbors[j]
                    if not visited[neighbor]:
                        visited[neighbor] = True

                        # find neighbors of neighbor
                        neighbor_neighbors = neighbors[neighbor]

                        # check if neighbor is a core point
                        if len(neighbor_neighbors) >= min_pts:
                            # add neighbor's neighbors to current_neighbors
                            for nn in neighbor_neighbors:
```

```
                            if nn not in current_neighbors:
                                current_neighbors = np.
↪append(current_neighbors, nn)

                    # add neighbor to cluster
                    if cluster_labels[neighbor] == 0:
                        cluster_labels[neighbor] = cluster

                    j += 1

            # assign noise label to non-core point
            else:
                cluster_labels[i] = -1

    # assign remaining unassigned points to noise
    cluster_labels[cluster_labels == 0] = -1

    # assign points to desired number of clusters
    unique_labels = np.unique(cluster_labels)
    num_clusters_found = len(unique_labels) - 1  # exclude noise label
    while num_clusters_found > num_clusters:
        # find smallest cluster
        smallest_cluster_size = np.inf
        smallest_cluster_label = None
        for label in unique_labels:
            if label == -1:
                continue
            cluster_size = np.sum(cluster_labels == label)
            if cluster_size < smallest_cluster_size:
                smallest_cluster_size = cluster_size
                smallest_cluster_label = label

        # merge smallest cluster with nearest neighbor
        nearest_neighbor_dist = np.inf
        nearest_neighbor_label = None
        for label in unique_labels:
            if label == smallest_cluster_label or label == -1:
                continue
            dist = np.min(np.linalg.norm(data[cluster_labels ==␣
↪smallest_cluster_label] -
                                         data[cluster_labels == label],␣
↪axis=1))
            if dist < nearest_neighbor_dist:
                nearest_neighbor_dist = dist
                nearest_neighbor_label = label
```

```
        cluster_labels[cluster_labels == smallest_cluster_label] =␣
↪nearest_neighbor_label
        unique_labels = np.unique(cluster_labels)
        num_clusters_found = len(unique_labels) - 1  # exclude noise label

    # return cluster labels for each point
    return cluster_labels
```

#Agglomerative Clustering

[7]:
```
def agglomerative_clustering(data, k):
    # initialize clusters
    clusters = [[i] for i in range(len(data))]
    # calculate distances between points
    euc_dist = cdist(data, data)
    # merge clusters until k clusters remain
    while len(clusters) > k:
        # find the closest clusters
        min_dist = np.inf
        for i in range(len(clusters)):
            for j in range(i+1, len(clusters)):
                # find the distance between the two clusters
                dist = 0
                for a in clusters[i]:
                    for b in clusters[j]:
                        dist += euc_dist[a][b]
                dist /= (len(clusters[i]) * len(clusters[j]))

                # update minimum distance and closest clusters
                if dist < min_dist:
                    min_dist = dist
                    closest_clusters = (i, j)

        # merge the closest clusters
        clusters[closest_clusters[0]] += clusters[closest_clusters[1]]
        del clusters[closest_clusters[1]]


    cluster_index_for_each_point = np.zeros(len(data))
    for u in range(len(clusters)):
      for t in range(len(clusters[u])):
        cluster_index_for_each_point [clusters[u][t]] = u

    # return the final clusters and cluster index for each point
    return clusters,cluster_index_for_each_point
```

#Evaluation Functions

##Conditional Entropy

[8]:
```
def conditional_entropy(labels, ground_truth):
  unique, counts = np.unique(labels[:], return_counts=True)
  conditional_ent = 0
  for j in range(len(unique)):
    cluster_number = unique [j]
    cluster_sum = counts[j]
    indices = np.where(labels[:] == cluster_number)
    indices = np.array(indices)
    indices=indices[0,:]
    cluster_mask = np.zeros(labels.shape[0])
    cluster_mask[indices] = 1
    ground_truth_clust = cluster_mask * ground_truth
    unique_clust, counts_clust = np.unique(ground_truth_clust,␣
 ↪return_counts=True)
    cond_ent_clust = 0
    for k in range(len(unique_clust)):
      if unique_clust[k] > 0:
        ent = (counts_clust[k] / cluster_sum) * math.log2(counts_clust[k] /␣
 ↪cluster_sum)
        cond_ent_clust = cond_ent_clust + -1*ent
    conditional_ent = conditional_ent + (cluster_sum/labels.
 ↪shape[0])*cond_ent_clust
  return conditional_ent
```

##Purity

[9]:
```
def purity(labels, ground_truth):
  unique, counts = np.unique(labels[:], return_counts=True)
  purity = 0
  for j in range(len(unique)):
    cluster_number = unique [j]
    cluster_sum = counts[j]
    indices = np.where(labels[:] == cluster_number)
    indices = np.array(indices)
    indices=indices[0,:]
    cluster_mask = np.zeros(labels.shape[0])

    cluster_mask[indices] = 1
    ground_truth_clust =  cluster_mask * ground_truth
    unique_clust, counts_clust = np.unique(ground_truth_clust,␣
 ↪return_counts=True)
    majority = -1
    for k in range(len(unique_clust)):
      if unique_clust[k] > 0:
        if counts_clust[k] > majority:
```

```
            majority = counts_clust[k]
    cluster_purity = majority / cluster_sum
    purity = purity + (cluster_sum/labels.shape[0])*cluster_purity
  return purity
```

##F-Measure

[10]:
```
def f_measure(labels, ground_truth):
  unique, counts = np.unique(labels[:], return_counts=True)
  sigma_f = 0
  clusters = len(unique)
  for j in range(len(unique)):
    cluster_number = unique [j]
    cluster_sum = counts[j]
    indices = np.where(labels[:] == cluster_number)
    indices = np.array(indices)
    indices=indices[0,:]
    cluster_mask = np.zeros(labels.shape[0])
    cluster_mask[indices] = 1
    ground_truth_clust = cluster_mask * ground_truth
    unique_clust, counts_clust = np.unique(ground_truth_clust,␣
 ↪return_counts=True)
    majority = -1
    for k in range(len(unique_clust)):
      if unique_clust[k] > 0:
        if counts_clust[k] > majority:
          majority = counts_clust[k]
          index = k
    prec = majority / cluster_sum
    total_occurence = np.count_nonzero(ground_truth == unique_clust[index])
    rec = majority / total_occurence
    F_measure = (2 * prec * rec) / (prec + rec)
    sigma_f = sigma_f + F_measure
  return (sigma_f/clusters)
```

##Pairwise Measures

[11]:
```
def pairwise_measures(labels, ground_truth):
  unique, counts = np.unique(labels[:], return_counts=True)
  TP = 0
  FP = 0
  FN = 0
  TN = 0
  cluster_counts = []
  for j in range(len(unique)):
    cluster_number = unique [j]
    cluster_sum = counts[j]
```

```
    indices = np.where(labels[:] == cluster_number)
    indices = np.array(indices)
    indices=indices[0,:]
    cluster_mask = np.zeros(labels.shape[0])
    cluster_mask[indices] = 1
    ground_truth_clust = cluster_mask * ground_truth
    unique_clust, counts_clust = np.unique(ground_truth_clust,␣
↪return_counts=True)
    matrix = np.column_stack((unique_clust, counts_clust))
    for i in range(matrix.shape[0]):
      cluster_counts.append(matrix[i])

    FP_temp = 0
    for k in range(len(unique_clust)):
      if unique_clust[k] > 0:
        TP = TP + math.comb(counts_clust[k], 2)
        FP_temp = FP_temp + (counts_clust[k] * (cluster_sum - counts_clust[k]))

    FP = FP + FP_temp / 2

  cluster_counts = np.array(cluster_counts)
  for m in range (cluster_counts.shape[0]):
    if cluster_counts[m,0] > 0:
      for n in range  (cluster_counts.shape[0]):
        if cluster_counts[m,0] == cluster_counts[n,0] and m!=n:
          FN = FN + cluster_counts[m,1] * cluster_counts[n,1]
        if cluster_counts[m,0] != cluster_counts[n,0] and m!=n and␣
↪cluster_counts[n,0] != 0:
          TN = TN + cluster_counts[m,1] * cluster_counts[n,1]
  FN = FN / 2
  TN = (TN / 2)- FP

  jacc = TP / (FN + TP + FP)
  rand = (TP + TN) / (FN + TN + TP + FP)
  return jacc, rand
```

##Beta CV

[12]:
```
def beta_cv(Data_matrix, labels):
  unique, counts = np.unique(labels[:], return_counts=True)
  for i in range(len(unique)):
    labels = np.where(labels == unique[i], i+1, labels)
  unique, counts = np.unique(labels[:], return_counts=True)
  N_in = 0
  N_out = 0
  for i in range(len(counts)):
    N_in = N_in + math.comb(counts[i],2)
```

```
    for j in range(len(counts)):
      if i!=j:
        N_out = N_out + counts[i] * counts[j]
  N_out = N_out / 2
  proximity_matrix = np.zeros([len(unique), len(unique)])
  for i in range(sum(counts)):
    for j in range(sum(counts)):
      if labels[i] == labels[j] and i!=j:
        distance = np.linalg.norm(Data_matrix[i] - Data_matrix[j])
        proximity_matrix[int(labels[i]) - 1, int(labels[i]) - 1] += distance
      elif labels[i] != labels[j]:
        distance = np.linalg.norm(Data_matrix[i] - Data_matrix[j])
        proximity_matrix[int(labels[i]) - 1, int(labels[j]) - 1] += distance
        proximity_matrix[int(labels[j]) - 1, int(labels[i]) - 1] += distance
  W_in = np.trace(proximity_matrix) / 2
  W_out = (np.sum(proximity_matrix) - np.trace(proximity_matrix)) / 4

  BETACV = (W_in / N_in) / (W_out / N_out)
  return BETACV
```

##N_cut measure

[13]:
```
def N_cut(Data_matrix, labels):
  unique, counts = np.unique(labels[:], return_counts=True)
  for i in range(len(unique)):
    labels = np.where(labels == unique[i], i+1, labels)
  unique, counts = np.unique(labels[:], return_counts=True)
  proximity_matrix = np.zeros([len(unique), len(unique)])
  for i in range(sum(counts)):
    for j in range(sum(counts)):
      if labels[i] == labels[j] and i!=j:
        distance = np.linalg.norm(Data_matrix[i] - Data_matrix[j])
        proximity_matrix[int(labels[i]) - 1, int(labels[i]) - 1] += distance
      elif labels[i] != labels[j]:
        distance = np.linalg.norm(Data_matrix[i] - Data_matrix[j])
        proximity_matrix[int(labels[i]) - 1, int(labels[j]) - 1] += distance
        proximity_matrix[int(labels[j]) - 1, int(labels[i]) - 1] += distance
  Ncut_measure = 0
  for i in range(len(counts)):
    Ncut_measure = Ncut_measure + (sum(proximity_matrix[i,:])-␣
↪proximity_matrix[i,i]) / sum(proximity_matrix[i,:])

  return Ncut_measure
```

#Loading Dataset

[17]:
```
def load(data_path, test_path, train_only):
  le = LabelEncoder()
  if train_only == True:
    df = pd.read_csv(data_path, compression='gzip', header=None)
    num_rows = df.shape[0]
  else:
    df1 = pd.read_csv(data_path, compression='gzip', header=None)
    num_rows = df1.shape[0]
    df2 = pd.read_csv(test_path, compression='gzip', header=None)
    df = pd.concat([df1, df2], axis=0)
  # identify the categorical and numeric columns
  categorical_cols = [col for col in df.columns if df[col].dtype == 'object']
  categorical_cols = categorical_cols [:-1]
  numeric_cols = df.select_dtypes(include='number').columns.tolist()

  numeric_data = df.iloc[:, numeric_cols].values
  categorical_data = df.iloc[:, categorical_cols]

  # perform one-hot encoding on the categorical features except the label column
  one_hot_data = pd.get_dummies(categorical_data)

  # Apply label encoding to the label column
  df['col41_encoded'] = le.fit_transform(df[41])

  # Get the encoded label of the normal class
  label_list = []
  for label, encoded_label in zip(le.classes_, le.transform(le.classes_)):
      label_list.append(label[:-1])
  label_column = np.array(df['col41_encoded'])  # To make it an np array
  data = np.concatenate((numeric_data, one_hot_data), axis=1)
  for i in range(len(label_column)):
    label_column[i] = label_column[i] + 1
  return data, label_column, label_list, num_rows
```

#Sample Runs

##Datasets for Kmeans

[ ]:
```
test_path = '/content/kdd-cup-1999-data/corrected.gz'
training_path_kmeans = '/content/kdd-cup-1999-data/kddcup.data_10_percent.gz'
data_kmeans, labels_kmeans, label_list, num_rows = load(training_path_kmeans,␣
 ↪test_path, False)
training_data_kmeans = data_kmeans[0:num_rows, :]
training_labels_kmeans = labels_kmeans[0:num_rows]
testing_data_kmeans = data_kmeans[num_rows:, :]
testing_labels_kmeans = labels_kmeans[num_rows:]
```

##Kmeans Sample Run

```
[ ]: k = [7, 15, 23, 31, 45]
max_iterations = 10
threshold = 0.01
rand_restart = 5
print('K-MEANS')
for i in range(len(k)):
  print('For ', k[i], 'clusters: \n')
  centroids_Kmeans, data_cluster = kmeans(training_data_kmeans, k[i],␣
 ↪max_iterations, threshold, rand_restart)
  cluster_identification = classify_centroid(centroids_Kmeans, data_cluster,␣
 ↪training_labels_kmeans)
  cluster_labels, clustering_statistics = assign_centroids(testing_data_kmeans,␣
 ↪centroids_Kmeans, cluster_identification, label_list)
  ground_truth = testing_labels_kmeans

  print('\tNORMAL TRAFFIC DETECTED:\t\t{} occurence\n'.
 ↪format(int(clustering_statistics[label_list.index('normal')])))
  print('\tANOMALIES DETECTED: \n')
  for j in range(len(label_list)):
    if j != label_list.index('normal') and clustering_statistics[j] != 0:
      print('\n\t\t{} anomaly:\t{} occurence'.format(label_list[j],␣
 ↪int(clustering_statistics[j])))

  print('\n\n\tEXTERNAL MEASURES: \n')
  print('\n\t\tPurity: ', purity(cluster_labels, testing_labels_kmeans))
  print('\n\t\tConditional Entropy:', conditional_entropy(cluster_labels,␣
 ↪testing_labels_kmeans))
  print('\n\t\tF_measure: ', f_measure(cluster_labels, testing_labels_kmeans))
  jacc_Kmeans, rand_Kmeans = pairwise_measures(cluster_labels,␣
 ↪testing_labels_kmeans)
  print('\n\t\tJaccard Index: ', jacc_Kmeans)
  print('\n\t\tRand Index: ', rand_Kmeans)
  print('\n\n')
```

```
K-MEANS
For  7 clusters:

        NORMAL TRAFFIC DETECTED:                4220 occurence

        ANOMALIES DETECTED:


                back anomaly:   2194 occurence

                neptune anomaly:        132513 occurence

                smurf anomaly:  172088 occurence
```

```
        warezclient anomaly:    6 occurence

        warezmaster anomaly:    8 occurence

    EXTERNAL MEASURES:

        Purity:  0.7309961450539981

        Conditional Entropy: 1.0124654355054392

        F_measure:  0.39667079378460857

        Jaccard Index:  0.6932512962511278

        Rand Index:  0.8477954855410705


For  15 clusters:
    NORMAL TRAFFIC DETECTED:                66933 occurence
    ANOMALIES DETECTED:

        back anomaly:   2170 occurence

        neptune anomaly:        110028 occurence

        smurf anomaly:  131889 occurence

        warezclient anomaly:    6 occurence

        warezmaster anomaly:    3 occurence

    EXTERNAL MEASURES:

        Purity:  0.8171553134916679

        Conditional Entropy: 0.7636341547248007

        F_measure:  0.36292649422549295
```

Jaccard Index: 0.3529501908846765

Rand Index: 0.7200281269913461

For 23 clusters:

NORMAL TRAFFIC DETECTED: 115110 occurence

ANOMALIES DETECTED:

back anomaly: 2173 occurence

neptune anomaly: 61870 occurence

smurf anomaly: 131867 occurence

warezclient anomaly: 6 occurence

warezmaster anomaly: 3 occurence

EXTERNAL MEASURES:

Purity: 0.9167794642943263

Conditional Entropy: 0.381059115751135

F_measure: 0.3023690114114509

Jaccard Index: 0.28483224827824183

Rand Index: 0.7405316473172973

For 31 clusters:

NORMAL TRAFFIC DETECTED: 81960 occurence

ANOMALIES DETECTED:

back anomaly: 2169 occurence

```
        neptune anomaly:        62154 occurence

        smurf anomaly:  164737 occurence

        warezclient anomaly:    6 occurence

        warezmaster anomaly:    3 occurence

    EXTERNAL MEASURES:

        Purity:  0.918007645589318

        Conditional Entropy: 0.36481764478440687

        F_measure:  0.24227116462712286

        Jaccard Index:  0.27876607190684016

        Rand Index:  0.7392031009439175


For  45 clusters:

    NORMAL TRAFFIC DETECTED:                114698 occurence

    ANOMALIES DETECTED:

        back anomaly:   2174 occurence

        neptune anomaly:        60430 occurence

        satan anomaly:  1805 occurence

        smurf anomaly:  131914 occurence

        warezclient anomaly:    6 occurence

        warezmaster anomaly:    2 occurence

    EXTERNAL MEASURES:

        Purity:  0.9208241032186708
```

```
Conditional Entropy: 0.36976154096021896

F_measure:  0.24479212142049758

Jaccard Index:  0.27157681268312217

Rand Index:  0.7357416462651797
```

[ ]:
```
# Not used due to its very high complexity O(n^2) where n denotes the number of␣
 ↪testing points (311029 point)
print('\n\n\tINTERNAL MEASURES: \n')
print('\n\t\tBetaCV: ', beta_cv(testing_data_kmeans, cluster_labels))
print('\n\t\tNormalized Cut: ', N_cut(testing_data_kmeans, cluster_labels))
```

##Datasets for Spectral Clustering

[ ]:
```
training_path_ncut = '/content/kdd-cup-1999-data/kddcup.data.gz'
training_data_ncut, training_labels_ncut, label_list, num_rows=␣
 ↪load(training_path_ncut, None,  True)
```

##Spectral Clustering Sample Run

[ ]:
```
training_data_ncut, _, training_labels_ncut, _ =␣
 ↪train_test_split(training_data_ncut, training_labels_ncut, test_size=0.999,␣
 ↪random_state=42, shuffle = True, stratify = training_labels_ncut)
NN_similarity_metric = 3
k_clusters = 23
centroids_Ncut, data_cluster_Ncut = Normalized_cut(training_data_ncut,␣
 ↪k_clusters, NN_similarity_metric)
cluster_identification = classify_centroid(centroids_Ncut, data_cluster_Ncut,␣
 ↪training_labels_ncut)
clustering_statistics = assignment_statistics(data_cluster_Ncut,␣
 ↪cluster_identification, label_list)
print('NORMALIZED CUT\n')
print('\tNORMAL TRAFFIC DETECTED:\t\t{} occurence\n'.
 ↪format(int(clustering_statistics[label_list.index('normal')])))
print('\tANOMALIES DETECTED: \n')
for j in range(len(label_list)):
  if j != label_list.index('normal') and clustering_statistics[j] != 0:
    print('\n\t\t{} anomaly:\t{} occurence'.format(label_list[j],␣
 ↪int(clustering_statistics[j])))

print('\n\n\tEXTERNAL MEASURES: \n')
```

```
print('\n\t\tPurity: ', purity(data_cluster_Ncut, training_labels_ncut))
print('\n\t\tConditional Entropy:', conditional_entropy(data_cluster_Ncut,␣
 ↪training_labels_ncut))
print('\n\t\tF_measure: ', f_measure(data_cluster_Ncut, training_labels_ncut))
jacc_ncut, rand_ncut = pairwise_measures(data_cluster_Ncut,␣
 ↪training_labels_ncut)
print('\n\t\tJaccard Index: ', jacc_ncut)
print('\n\t\tRand Index: ', rand_ncut)
print('\n\n')
#########################################################################
# Compared to K-Means on the same data
max_iterations = 30
threshold = 0.01
rand_restart = 5
print('K-MEANS\n')
centroids_Kmeans, data_cluster = kmeans(training_data_ncut, k_clusters,␣
 ↪max_iterations, threshold, rand_restart)
cluster_identification = classify_centroid(centroids_Kmeans, data_cluster,␣
 ↪training_labels_ncut)
clustering_statistics = assignment_statistics(data_cluster_Ncut,␣
 ↪cluster_identification, label_list)
ground_truth = training_labels_ncut
print('\tNORMAL TRAFFIC DETECTED:\t\t{} occurence\n'.
 ↪format(int(clustering_statistics[label_list.index('normal')])))
print('\tANOMALIES DETECTED: \n')
for j in range(len(label_list)):
  if j != label_list.index('normal') and clustering_statistics[j] != 0:
    print('\n\t\t{} anomaly:\t{} occurence'.format(label_list[j],␣
 ↪int(clustering_statistics[j])))

print('\n\n\tEXTERNAL MEASURES: \n')
print('\n\t\tPurity: ', purity(data_cluster, training_labels_ncut))
print('\n\t\tConditional Entropy:', conditional_entropy(data_cluster,␣
 ↪training_labels_ncut))
print('\n\t\tF_measure: ', f_measure(data_cluster, training_labels_ncut))
jacc_Kmeans, rand_Kmeans = pairwise_measures(data_cluster, training_labels_ncut)
print('\n\t\tJaccard Index: ', jacc_Kmeans)
print('\n\t\tRand Index: ', rand_Kmeans)
print('\n\n')
```

```
NORMALIZED CUT

        NORMAL TRAFFIC DETECTED:                1292 occurence

        ANOMALIES DETECTED:
```

neptune anomaly: 1006 occurence

smurf anomaly: 2600 occurence

EXTERNAL MEASURES:

Purity: 0.9316047366271947

Conditional Entropy: 0.25248455525168145

F_measure: 0.1883921618730085

Jaccard Index: 0.4333502875076269

Rand Index: 0.762650243859771

K-MEANS

NORMAL TRAFFIC DETECTED: 1846 occurence

ANOMALIES DETECTED:

neptune anomaly: 1869 occurence

portsweep anomaly: 8 occurence

smurf anomaly: 1175 occurence

EXTERNAL MEASURES:

Purity: 0.9775418538178848

Conditional Entropy: 0.1266675184729025

F_measure: 0.22832223670255933

Jaccard Index: 0.5998512236173286

Rand Index: 0.8317536432210353

##Datasets for DBSCAN Clustering

[ ]:
```
training_path_DB = '/content/kdd-cup-1999-data/kddcup.data.gz'
training_data_DB, training_labels_DB, label_list, num_rows=␣
 ↪load(training_path_DB, None,  True)
```

##DBSCAN Clustring Sample Run

[ ]:
```
training_data_dbscan, _, training_labels_dbscan, _ =␣
 ↪train_test_split(training_data_DB, training_labels_DB, test_size=0.999,␣
 ↪random_state=42, shuffle = True, stratify = training_labels_DB)
k_clusters = 23
epsilon=8
min_per_cluster=20
labels = dbscan(training_data_dbscan, epsilon, min_per_cluster, k_clusters)
print('DBSCAN Clustring\n')
clusters_no_array = np.zeros(k_clusters)
cluster_identification = classify_centroid(clusters_no_array, labels,␣
 ↪training_labels_dbscan)
int_array = labels.astype(int)
clustering_statistics = assignment_statistics(int_array,␣
 ↪cluster_identification, label_list)
print('\tNORMAL TRAFFIC DETECTED:\t\t{} occurence\n'.
 ↪format(int(clustering_statistics[label_list.index('normal')])))
print('\tANOMALIES DETECTED: \n')
for j in range(len(label_list)):
  if j != label_list.index('normal') and clustering_statistics[j] != 0:
    print('\n\t\t{} anomaly:\t{} occurence'.format(label_list[j],␣
 ↪int(clustering_statistics[j])))
print('\n\n\tEXTERNAL MEASURES: \n')
print('\n\t\tPurity: ', purity(labels, training_labels_dbscan))
print('\n\t\tConditional Entropy:', conditional_entropy(labels,␣
 ↪training_labels_dbscan))
print('\n\t\tF_measure: ', f_measure(labels, training_labels_dbscan))
jacc_ncut, rand_ncut = pairwise_measures(labels, training_labels_dbscan)
print('\n\t\tJaccard Index: ', jacc_ncut)
print('\n\t\tRand Index: ', rand_ncut)
print('\n\n')
```

```
DBSCAN Clustring

        NORMAL TRAFFIC DETECTED:                2332 occurence

        ANOMALIES DETECTED:
```

```
neptune anomaly:        169 occurence

smurf anomaly:  1333 occurence

warezmaster anomaly:    1064 occurence


EXTERNAL MEASURES:


Purity:  0.9661086157615354

Conditional Entropy: 0.20117618743700807

F_measure:  0.3885991751741702

Jaccard Index:  0.6460736690321968

Rand Index:  0.8481580501157657
```

##Datasets for Agglomerative Clustering

[18]:
```
training_path_Agg = '/content/kdd-cup-1999-data/kddcup.data.gz'
training_data_Agg, training_labels_Agg, label_list, num_rows=␣
 ↪load(training_path_Agg, None,  True)
```

##Agglomerative Clustring Sample Run

[24]:
```
training_data_agg, _, training_labels_agg, _ =␣
 ↪train_test_split(training_data_Agg, training_labels_Agg, test_size=0.9999,␣
 ↪random_state=42, shuffle = True, stratify = training_labels_Agg)
k_clusters = 23
clusters,labels = agglomerative_clustering(training_data_agg, k_clusters)
print('AGGLOMERATIVE Clustring\n')
clusters_no_array = np.zeros(k_clusters)
cluster_identification = classify_centroid(clusters_no_array, labels,␣
 ↪training_labels_agg)
int_array = labels.astype(int)
clustering_statistics = assignment_statistics(int_array,␣
 ↪cluster_identification, label_list)
print('\tNORMAL TRAFFIC DETECTED:\t\t{} occurence\n'.
 ↪format(int(clustering_statistics[label_list.index('normal')])))
print('\tANOMALIES DETECTED: \n')
for j in range(len(label_list)):
  if j != label_list.index('normal') and clustering_statistics[j] != 0:
```

```
    print('\n\t\t{} anomaly:\t{} occurence'.format(label_list[j],␣
 ↪int(clustering_statistics[j])))
print('\n\n\tEXTERNAL MEASURES: \n')
print('\n\t\tPurity: ', purity(labels, training_labels_agg))
print('\n\t\tConditional Entropy:', conditional_entropy(labels,␣
 ↪training_labels_agg))
print('\n\t\tF_measure: ', f_measure(labels, training_labels_agg))
jacc_ncut, rand_ncut = pairwise_measures(labels, training_labels_agg)
print('\n\t\tJaccard Index: ', jacc_ncut)
print('\n\t\tRand Index: ', rand_ncut)
print('\n\n')
```

```
AGGLOMERATIVE Clustring

        NORMAL TRAFFIC DETECTED:                207 occurence

        ANOMALIES DETECTED:


                neptune anomaly:        279 occurence

                smurf anomaly:  3 occurence


        EXTERNAL MEASURES:


                Purity:  0.9243353783231081

                Conditional Entropy: 0.297681633438901

                F_measure:  0.13660089611025555

                Jaccard Index:  0.8414802643862729

                Rand Index:  0.9288444131549835
```