Assignment 2: Network Anomaly Detection

●	Farah Moustafa	6888
●	Asala Ahmed	6916
●	Zahraa elHareedy	6895

Table of Contents

Notebook link: https://colab.research.google.com/drive/1PMK-2-FcIftDWB6FvJ4q98hrdJU-7uiF?usp=sharing

The ”KDD Cup 1999” Dataset

The KDD Cup 1999 dataset consists of network traffic data that was captured on a simulated network environment. The data was collected over a seven-week period and includes a mix of normal and malicious traffic.
Each connection record includes 41 different attributes, or features, which describe various aspects of the network traffic. These features include information such as the duration of the connection, the protocol used, the number of bytes sent and received, and other information that can be used to identify patterns of normal or malicious traffic.
The format of the dataset is a plain text file with comma-separated values (CSV). Each row in the file represents a single connection record, and each column represents a different attribute.

Different types of anomalies

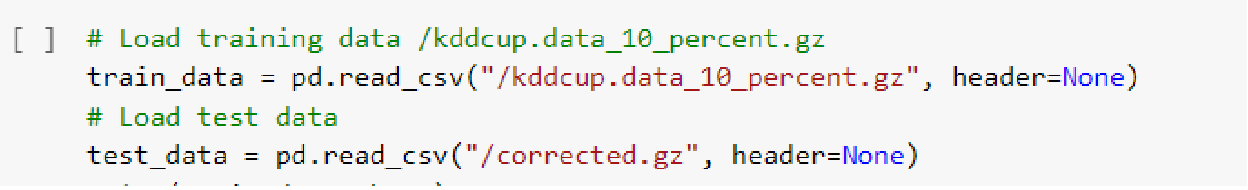
The KDD Cup 1999 dataset includes attack traffic generated by a total of 22 different tools. These tools were used to simulate various types of attacks on a network, including DoS, probing, and remote-to-local attacks. Here is the complete list of tools included in the dataset:

1. Back: This tool is a remote access trojan (RAT) that allows an attacker to take control of a victim's computer.
2. Land: This tool generates a DoS attack by sending a TCP packet with the same source and destination IP address and port.
3. Neptune: This tool generates a DoS attack by flooding a target with TCP, UDP, or ICMP packets.
4. Pod: This tool exploits a vulnerability in the TCP/IP protocol stack to cause a system to crash or become unresponsive.
5. Smurf: This tool exploits a vulnerability in the Internet Control Message Protocol (ICMP) to launch a DoS attack.
6. Teardrop: This tool sends malformed packets to a target system that can cause it to crash or become unresponsive.
7. Apache2: This tool simulates an HTTP DoS attack.
8. Backdoor: This tool is a remote access trojan (RAT) that allows an attacker to take control of a victim's computer.
9. FTP_write: This tool exploits a vulnerability in the File Transfer Protocol (FTP) to gain unauthorized write access to an FTP server.
10. Guess_passwd: This tool attempts to guess the password for a user account on a target system.
11. httptunnel: This tool can be used to bypass firewalls and other network security measures by encapsulating arbitrary data in HTTP requests and responses.

12. Named: This tool exploits a vulnerability in the DNS server to gain unauthorized access to a target system.
13. IPSweep: This tool is a program that can be used to scan a range of IP addresses to determine which ones have active hosts.
14. MQSeries: This tool exploits a vulnerability in IBM's MQSeries messaging system to gain unauthorized access to a target system.
15. NetBus: This tool is a remote access trojan (RAT) that allows an attacker to take control of a victim's computer.
16. NetCat: This tool is a command-line utility that can be used to create TCP/IP connections and transfer data between systems.
17. Satan: This tool can be used for port scanning and reconnaissance to gather information about a target system.
18. Snmpgetattack: This tool exploits a vulnerability in the Simple Network Management Protocol (SNMP) to gain unauthorized access to a target system.
19. Spy: This tool is a remote access trojan (RAT) that allows an attacker to take control of a victim's computer.
20. Stacheldraht: This tool is a command-and-control (C&C) tool used to coordinate DoS attacks.
21. UDPstorm: This tool generates a DoS attack by flooding a target with UDP packets.
22. Worm: This tool is a self-replicating malware that spreads over a network by exploiting vulnerabilities in target systems.

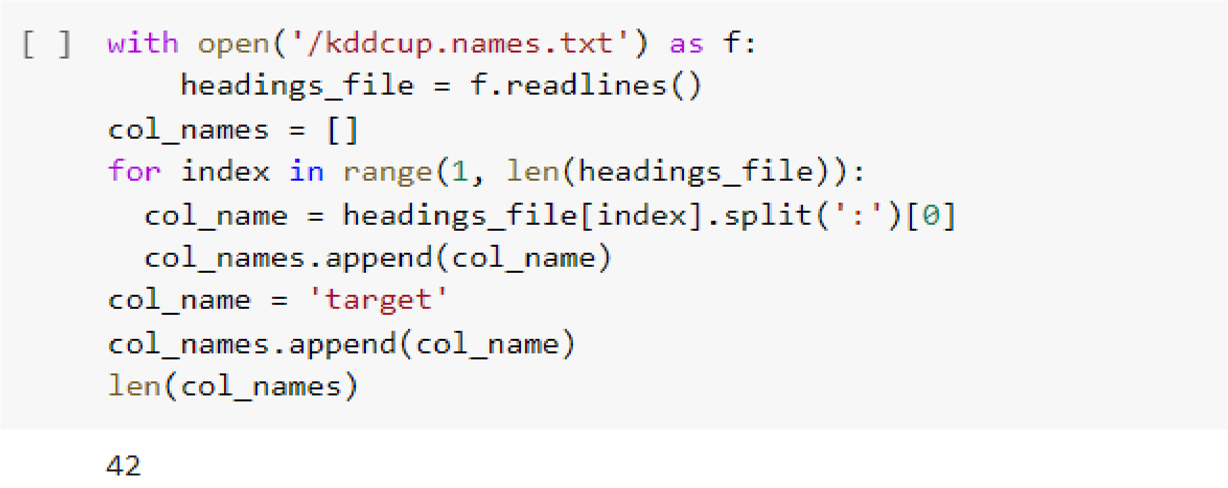
Download Dataset

Unzip the Datasets



```
# Load training data /kddcup.data_10_percent.gz
train_data = pd.read_csv("/kddcup.data_10_percent.gz", header=None)
# Load test data
test_data = pd.read_csv("/corrected.gz", header=None)
```

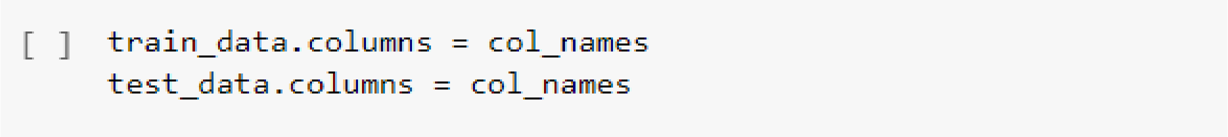
Load and parse columns' headings



```
with open('/kddcup.names.txt') as f:
    headings_file = f.readlines()
col_names = []
for index in range(1, len(headings_file)):
  col_name = headings_file[index].split(':')[0]
  col_names.append(col_name)
col_name = 'target'
col_names.append(col_name)
len(col_names)
```

```
42
```

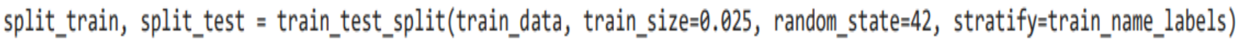
Assign column headings



```
train_data.columns = col_names
test_data.columns = col_names
```

Taking only a percentage for the normalized cut

Note that for the normalized cut, only 2.5% of the data is taken.



```
split_train, split_test = train_test_split(train_data, train_size=0.025, random_state=42, stratify=train_name_labels)
```

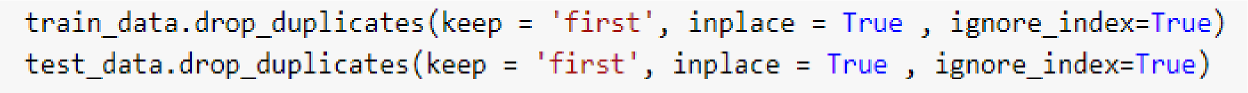
Preprocessing

Note: For each of the following preprocessing steps, the same steps are repeated for the split_train.

Dropping Duplicates

Removing duplicates in data preprocessing is a crucial step to enhance data accuracy, and ultimately leading to more reliable analysis results.

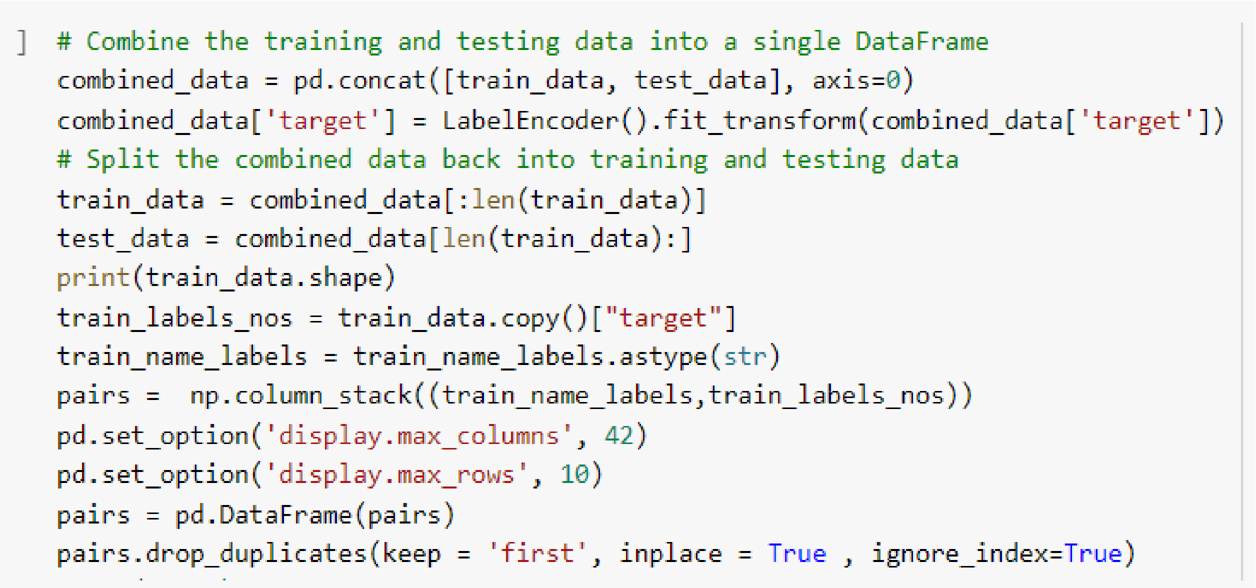
- Accuracy: Duplicate data can lead to inaccurate analysis results by artificially inflating the count of specific categories, causing them to appear more significant than they actually are. This can lead to misleading analysis results that are not representative of the true nature of the data.
- Efficiency: Duplicate data can also cause computational inefficiencies, particularly when performing calculations or analysis on large datasets. Redundant data can result in unnecessary extra work, slowing down processing times and reducing overall efficiency.
- Data size: Duplicate data can unnecessarily increase the size of a dataset, which can be problematic when working with large datasets that take up significant memory or storage space. This can result in slower processing times, decreased system performance, and increased costs associated with data storage and management.



```
train_data.drop_duplicates(keep = 'first', inplace = True , ignore_index=True)
test_data.drop_duplicates(keep = 'first', inplace = True , ignore_index=True)
```

Label Encoding for targets

Label encoding is used in preprocessing the target column to convert categorical data into numerical data.



```
] # Combine the training and testing data into a single DataFrame
  combined_data = pd.concat([train_data, test_data], axis=0)
  combined_data['target'] = LabelEncoder().fit_transform(combined_data['target'])
  # Split the combined data back into training and testing data
  train_data = combined_data[:len(train_data)]
  test_data = combined_data[len(train_data):]
  print(train_data.shape)
  train_labels_nos = train_data.copy()["target"]
  train_name_labels = train_name_labels.astype(str)
  pairs =  np.column_stack((train_name_labels,train_labels_nos))
  pd.set_option('display.max_columns', 42)
  pd.set_option('display.max_rows', 10)
  pairs = pd.DataFrame(pairs)
  pairs.drop_duplicates(keep = 'first', inplace = True , ignore_index=True)
```

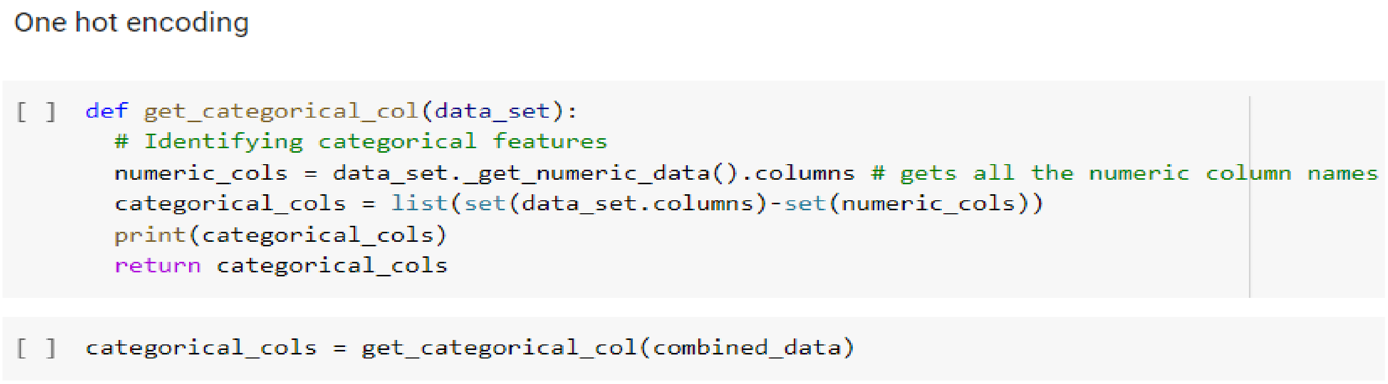
In the previous code, The reason for combining the training and testing data into a single DataFrame is to ensure that any data transformations or preprocessing steps that are applied to the training data are also applied to the testing data. This helps to avoid any discrepancies or errors that could arise due to differences in the data processing between the training and testing datasets.

The pairs DataFrame is created by stacking the train_name_labels and train_labels_nos columns. This DataFrame contains a pair of values for each row, where the first value corresponds to the name of the target class and the second value corresponds to the numerical label assigned to that class by the label encoding process. This DataFrame is useful for mapping between the original target class names and the numerical labels used in the machine learning model.

One-Hot encoding for categorical features

Applying label encoding to feature columns can introduce ordinality in the data, which can result in incorrect model outputs. Therefore, it is crucial to use one-hot encoding or other techniques to preprocess categorical data in feature columns.

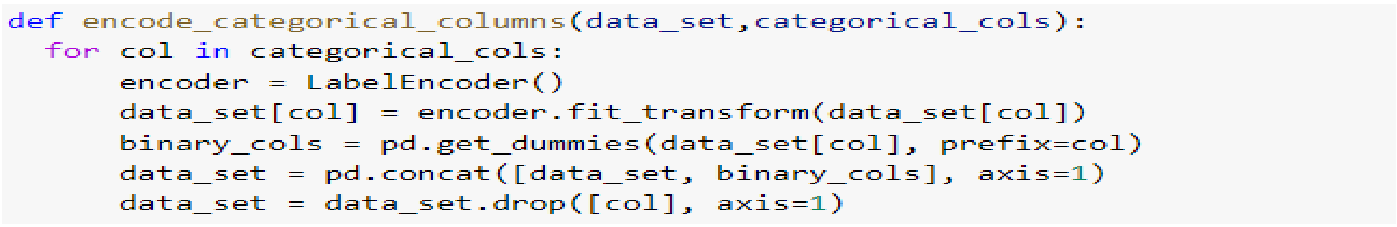
One-hot encoding creates a binary vector for each categorical value in the feature, where all elements are zero except for the element corresponding to the value, which is set to one. This encoding ensures that the numerical representation of the categorical feature is unique and independent of the original ordering of the categorical values.



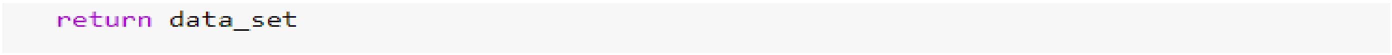
One hot encoding

```
[ ]  def get_categorical_col(data_set):
       # Identifying categorical features
       numeric_cols = data_set._get_numeric_data().columns # gets all the numeric column names
       categorical_cols = list(set(data_set.columns)-set(numeric_cols))
       print(categorical_cols)
       return categorical_cols
```

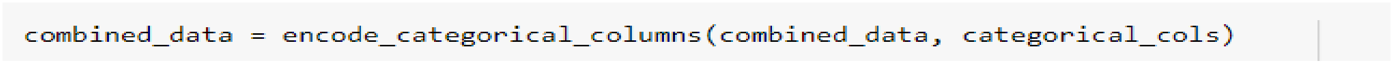
```
[ ]  categorical_cols = get_categorical_col(combined_data)
```



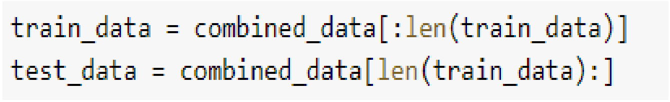
```
def encode_categorical_columns(data_set,categorical_cols):
  for col in categorical_cols:
      encoder = LabelEncoder()
      data_set[col] = encoder.fit_transform(data_set[col])
      binary_cols = pd.get_dummies(data_set[col], prefix=col)
      data_set = pd.concat([data_set, binary_cols], axis=1)
      data_set = data_set.drop([col], axis=1)
```



```
   return data_set
```



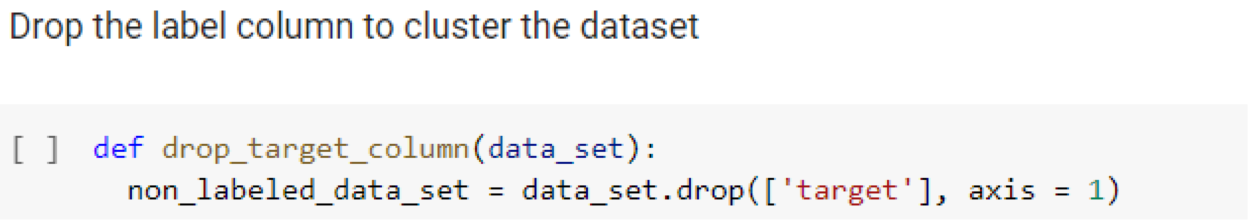
```
combined_data = encode_categorical_columns(combined_data, categorical_cols)
```



```
train_data = combined_data[:len(train_data)]
test_data = combined_data[len(train_data):]
```

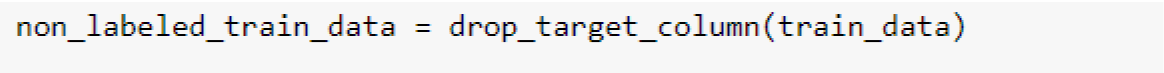
Dropping the targets column

Clustering is unsupervised learning, thus, it is done without labels.



Drop the label column to cluster the dataset

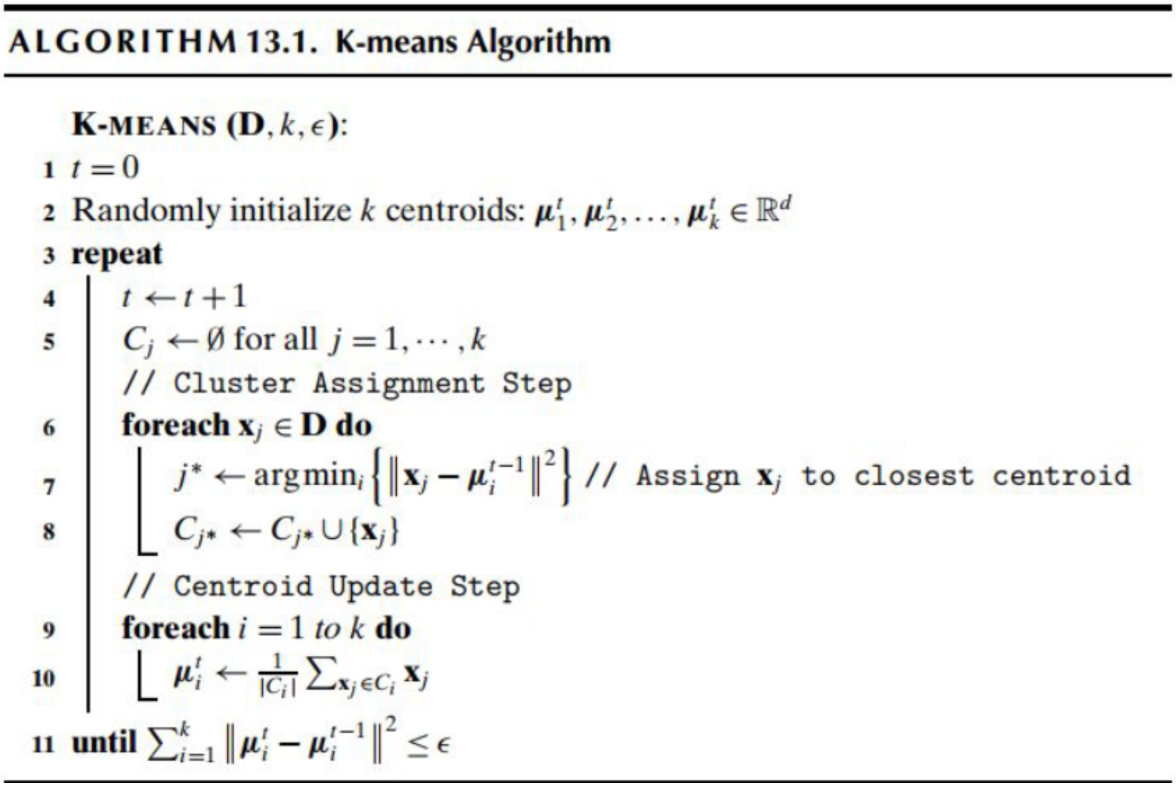
```
[ ]  def drop_target_column(data_set):
       non_labeled_data_set = data_set.drop(['target'], axis = 1)
```



```
non_labeled_train_data = drop_target_column(train_data)
```

K-Means

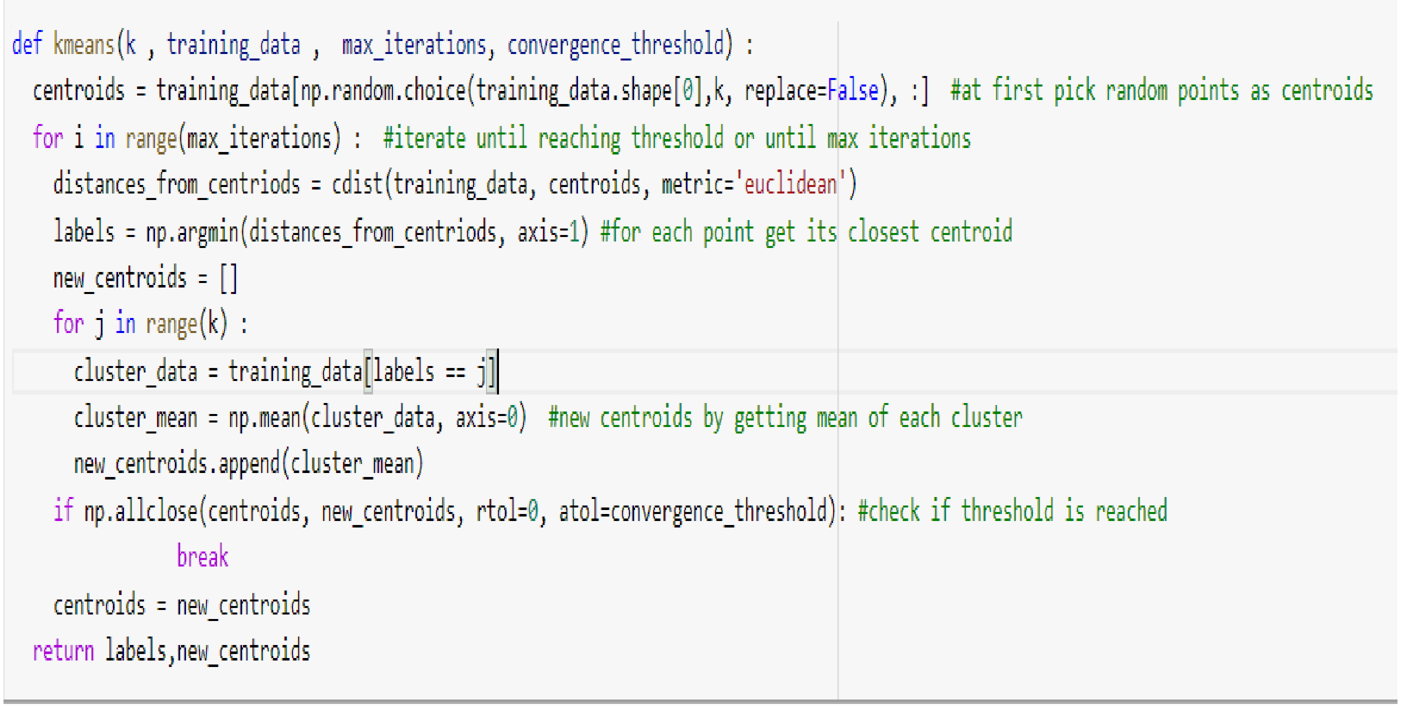
Algorithm



ALGORITHM 13.1. K-means Algorithm

```
K-MEANS ($\mathbf{D}, k, \epsilon$):
$t = 0$
Randomly initialize $k$ centroids: $\boldsymbol{\mu}_1^t, \boldsymbol{\mu}_2^t, \dots, \boldsymbol{\mu}_k^t \in \mathbb{R}^d$
repeat
    $t \leftarrow t + 1$
    $C_j \leftarrow \emptyset$ for all $j = 1, \cdots, k$
    // Cluster Assignment Step
    foreach $\mathbf{x}_j \in \mathbf{D}$ do
        $j^* \leftarrow \arg\min_i \left\{ \left\| \mathbf{x}_j - \boldsymbol{\mu}_i^{t-1} \right\|^2 \right\}$ // Assign $\mathbf{x}_j$ to closest centroid
        $C_{j^*} \leftarrow C_{j^*} \cup \{\mathbf{x}_j\}$
    // Centroid Update Step
    foreach $i = 1$ to $k$ do
        $\boldsymbol{\mu}_i^t \leftarrow \frac{1}{|C_i|} \sum_{\mathbf{x}_j \in C_i} \mathbf{x}_j$
until $\sum_{i=1}^k \left\| \boldsymbol{\mu}_i^t - \boldsymbol{\mu}_i^{t-1} \right\|^2 \leq \epsilon$
```

Implementation



```
def kmeans(k , training_data ,  max_iterations, convergence_threshold) :
  centroids = training_data[np.random.choice(training_data.shape[0],k, replace=False), :]  #at first pick random points as centroids
  for i in range(max_iterations) :  #iterate until reaching threshold or until max iterations
    distances_from_centriods = cdist(training_data, centroids, metric='euclidean')
    labels = np.argmin(distances_from_centriods, axis=1) #for each point get its closest centroid
    new_centroids = []
    for j in range(k) :
      cluster_data = training_data[labels == j]
      cluster_mean = np.mean(cluster_data, axis=0)  #new centroids by getting mean of each cluster
      new_centroids.append(cluster_mean)
    if np.allclose(centroids, new_centroids, rtol=0, atol=convergence_threshold): #check if threshold is reached
                break
    centroids = new_centroids
  return labels,new_centroids
```

Explanation

The kmeans function starts by randomly selecting k data points from the training data as the initial centroids. It then iteratively performs the following steps until convergence or until the maximum number of iterations is reached:

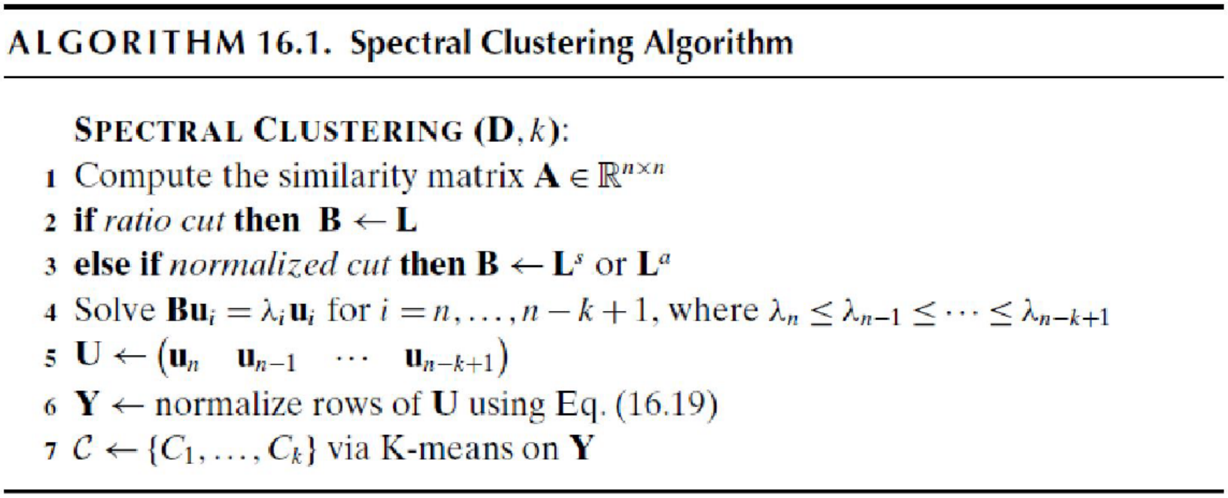
- Calculates the distance between each data point and each centroid, using the cdist function from the scipy.spatial.distance module.
- Assigns each data point to its nearest centroid, using the argmin function to obtain the index of the centroid with the minimum distance.
- Computes the new centroids by taking the mean of the data points assigned to each cluster.
- Checks for convergence by comparing the new centroids to the previous centroids, using the np.allclose function with the specified convergence threshold.
- Updates the centroids to the new centroids.

The function returns two values:

- labels: a 1D NumPy array containing the cluster assignments for each data point.
- new_centroids: a 2D NumPy array containing the final centroids for each cluster.

Normalized Cut

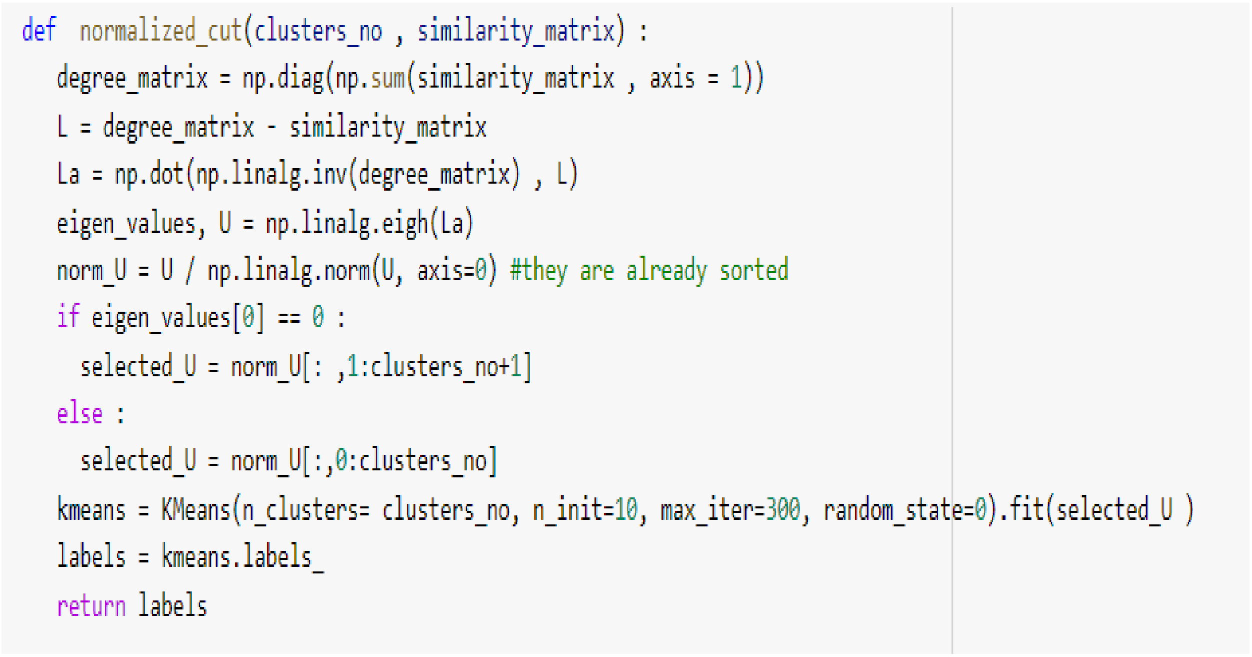
Algorithm



ALGORITHM 16.1. Spectral Clustering Algorithm

```
SPECTRAL CLUSTERING ($\mathbf{D}, k$):
Compute the similarity matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$
if ratio cut then $\mathbf{B} \leftarrow \mathbf{L}$
else if normalized cut then $\mathbf{B} \leftarrow \mathbf{L}^s$ or $\mathbf{L}^a$
Solve $\mathbf{B}\mathbf{u}_i = \lambda_i \mathbf{u}_i$ for $i = n, \ldots, n-k+1$, where $\lambda_n \le \lambda_{n-1} \le \cdots \le \lambda_{n-k+1}$
$\mathbf{U} \leftarrow \begin{pmatrix} \mathbf{u}_n & \mathbf{u}_{n-1} & \cdots & \mathbf{u}_{n-k+1} \end{pmatrix}$
$\mathbf{Y} \leftarrow$ normalize rows of $\mathbf{U}$ using Eq. (16.19)
$\mathcal{C} \leftarrow \{C_1, \ldots, C_k\}$ via K-means on $\mathbf{Y}$
```

Implementation



```python
def  normalized_cut(clusters_no , similarity_matrix) :
   degree_matrix = np.diag(np.sum(similarity_matrix , axis = 1))
   L = degree_matrix - similarity_matrix
   La = np.dot(np.linalg.inv(degree_matrix) , L)
   eigen_values, U = np.linalg.eigh(La)
   norm_U = U / np.linalg.norm(U, axis=0) #they are already sorted
   if eigen_values[0] == 0 :
     selected_U = norm_U[: ,1:clusters_no+1]
   else :
     selected_U = norm_U[:,0:clusters_no]
   kmeans = KMeans(n_clusters= clusters_no, n_init=10, max_iter=300, random_state=0).fit(selected_U )
   labels = kmeans.labels_
   return labels
```

Explanation

The function starts by calculating the degree matrix of the similarity matrix, which is a diagonal matrix whose diagonal elements are the sum of the similarities of each data point to all other data points. The degree matrix is then used to compute the Laplacian matrix of the similarity matrix, which is the difference between the degree matrix and the similarity matrix. The Laplacian matrix is a matrix that describes the structure of the data and is often used in spectral clustering.

The next step in the algorithm is to calculate the eigenvectors and eigenvalues of the Laplacian matrix. The eigenvectors are sorted based on their corresponding eigenvalues, and the first clusters_no eigenvectors are selected. The selected eigenvectors are then clustered using k-means clustering to obtain the final cluster assignments.

Clustering Evaluations

What are these evaluations?

Clustering evaluation is the process of assessing the quality of clustering results obtained from a clustering algorithm. Clustering evaluation is important because it allows us to determine how well the algorithm has performed, and whether the resulting clusters are meaningful and useful.

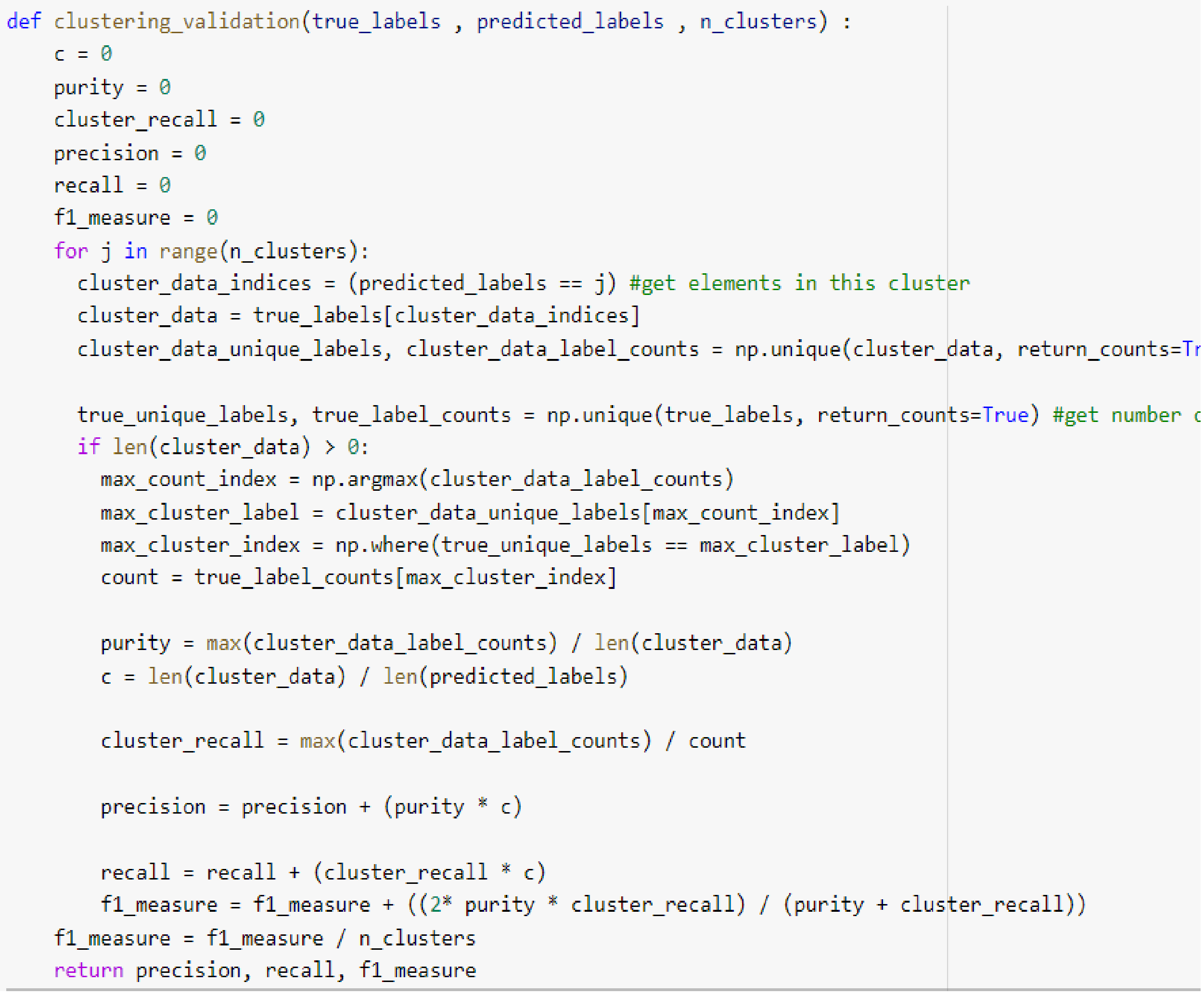
Precision, recall, and F1 measure

Precision, recall, and F1 measure are commonly used evaluation metrics in classification tasks. Precision is the number of true positive predictions divided by the total number of positive predictions, which means it measures the proportion of correctly predicted positive instances out of all predicted positive instances. A high precision indicates that the model is very selective in predicting positive instances.

Recall is the number of true positive predictions divided by the total number of actual positive instances in the dataset, which means it measures the proportion of correctly predicted positive instances out of all actual positive instances. A high recall indicates that the model is very sensitive in detecting positive instances.

F1 measure is the harmonic mean of precision and recall, which means it provides a balance between precision and recall. It is calculated as 2 * (precision * recall) / (precision + recall). A high F1 score indicates that the model is both selective and sensitive in predicting positive instances.

Implementation



```
def clustering_validation(true_labels , predicted_labels , n_clusters) :
    c = 0
    purity = 0
    cluster_recall = 0
    precision = 0
    recall = 0
    f1_measure = 0
    for j in range(n_clusters):
      cluster_data_indices = (predicted_labels == j) #get elements in this cluster
      cluster_data = true_labels[cluster_data_indices]
      cluster_data_unique_labels, cluster_data_label_counts = np.unique(cluster_data, return_counts=Tr

      true_unique_labels, true_label_counts = np.unique(true_labels, return_counts=True) #get number c
      if len(cluster_data) > 0:
        max_count_index = np.argmax(cluster_data_label_counts)
        max_cluster_label = cluster_data_unique_labels[max_count_index]
        max_cluster_index = np.where(true_unique_labels == max_cluster_label)
        count = true_label_counts[max_cluster_index]

        purity = max(cluster_data_label_counts) / len(cluster_data)
        c = len(cluster_data) / len(predicted_labels)

        cluster_recall = max(cluster_data_label_counts) / count

        precision = precision + (purity * c)

        recall = recall + (cluster_recall * c)
        f1_measure = f1_measure + ((2* purity * cluster_recall) / (purity + cluster_recall))
    f1_measure = f1_measure / n_clusters
    return precision, recall, f1_measure
```

Explanation

1. purity: the proportion of correctly assigned samples in a cluster, which is the ratio of the highest count of a true label in the predicted cluster to the total number of samples in the predicted cluster.
2. c: the proportion of samples in the predicted cluster to the total number of samples.
3. cluster_recall: the ratio of the highest count of a true label in the predicted cluster to the number of samples in the corresponding true cluster.

Using purity and c, it calculates precision for each cluster by taking their product and summing over all clusters. Similarly, it calculates recall for each cluster using cluster_recall and c, and sums over all clusters.

Finally, it calculates the f1_measure as the harmonic mean of precision and recall over all clusters.
The function returns precision, recall, and f1_measure as the evaluation metrics for the clustering algorithm.

Conditional Entropy

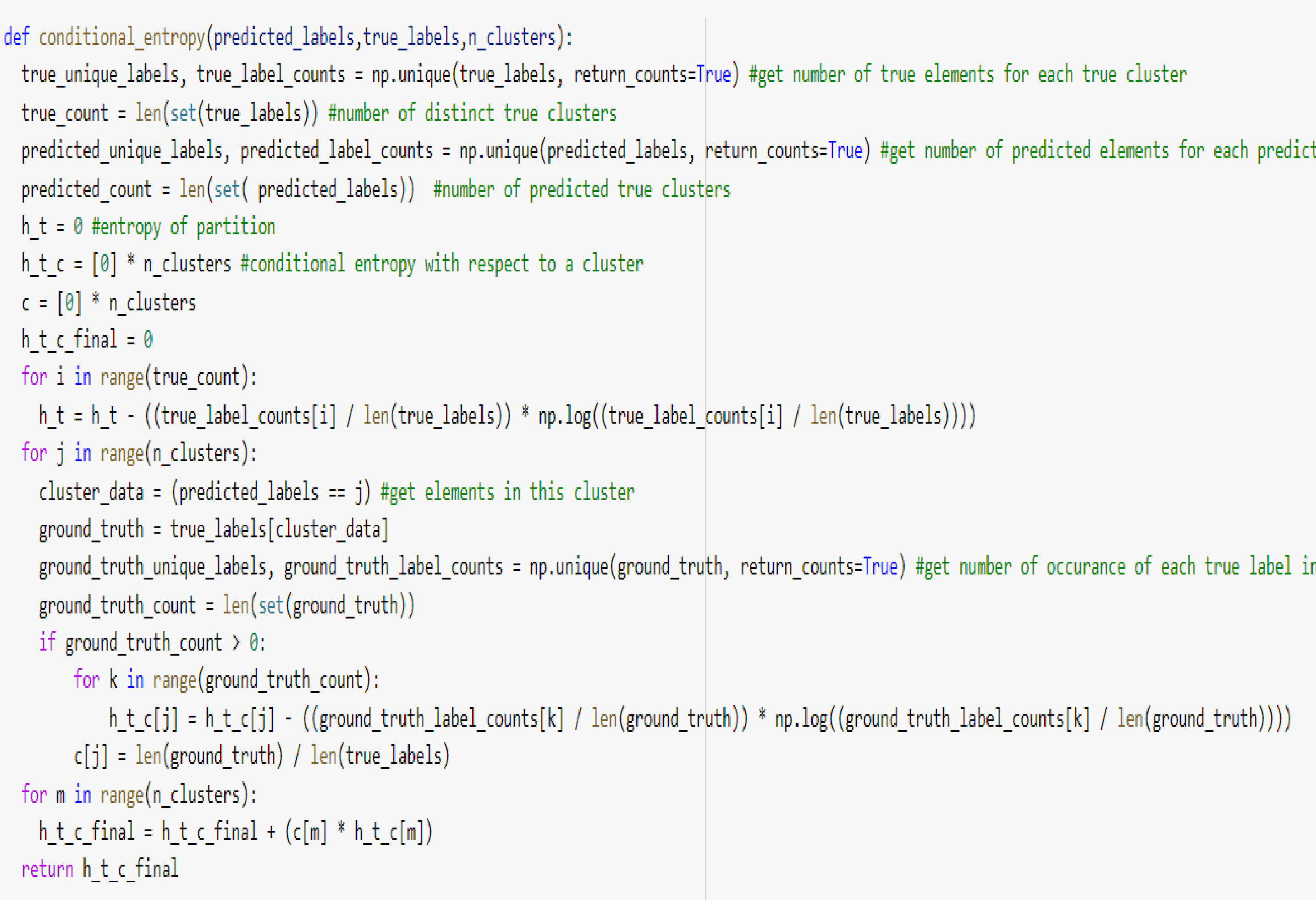
Conditional entropy is a measure of the amount of uncertainty remaining in a random variable Y given that the value of another random variable X is known. It can be defined as the average amount of entropy in Y, given the value of X:

$H(Y|X) = - \sum p(x,y) \log(p(y|x))$

where p(x,y) is the joint probability distribution of X and Y, and p(y|x) is the conditional probability distribution of Y given X.

In other words, conditional entropy measures the amount of uncertainty in the values of Y that remain once we know the value of X. If X provides a lot of information about Y, then the conditional entropy H(Y|X) will be low, indicating that there is little uncertainty in the values of Y given X. On the other hand, if X provides little information about Y, then the conditional entropy H(Y|X) will be high, indicating that there is still a lot of uncertainty in the values of Y given X.

Implementation



```
def conditional_entropy(predicted_labels,true_labels,n_clusters):
  true_unique_labels, true_label_counts = np.unique(true_labels, return_counts=True) #get number of true elements for each true cluster
  true_count = len(set(true_labels)) #number of distinct true clusters
  predicted_unique_labels, predicted_label_counts = np.unique(predicted_labels, return_counts=True) #get number of predicted elements for each predict
  predicted_count = len(set( predicted_labels))  #number of predicted true clusters
  h_t = 0 #entropy of partition
  h_t_c = [0] * n_clusters #conditional entropy with respect to a cluster
  c = [0] * n_clusters
  h_t_c_final = 0
  for i in range(true_count):
    h_t = h_t - ((true_label_counts[i] / len(true_labels)) * np.log((true_label_counts[i] / len(true_labels))))
  for j in range(n_clusters):
    cluster_data = (predicted_labels == j) #get elements in this cluster
    ground_truth = true_labels[cluster_data]
    ground_truth_unique_labels, ground_truth_label_counts = np.unique(ground_truth, return_counts=True) #get number of occurance of each true label in
    ground_truth_count = len(set(ground_truth))
    if ground_truth_count > 0:
        for k in range(ground_truth_count):
            h_t_c[j] = h_t_c[j] - ((ground_truth_label_counts[k] / len(ground_truth)) * np.log((ground_truth_label_counts[k] / len(ground_truth))))
        c[j] = len(ground_truth) / len(true_labels)
  for m in range(n_clusters):
    h_t_c_final = h_t_c_final + (c[m] * h_t_c[m])
  return h_t_c_final
```

Explanation

The function first calculates the entropy of the true partition h_t by iterating through the true labels and using the entropy formula.
Then, for each predicted cluster, it calculates the conditional entropy with respect to that cluster by iterating through the unique true labels within that cluster and using the conditional entropy formula. It also calculates the proportion of data points in that predicted cluster c[j].
Finally, the function computes the overall conditional entropy by taking a weighted sum of the conditional entropy with respect to each cluster, where the weights are given by c[j].
The output of the function is the overall conditional entropy. A lower conditional entropy indicates that the clustering is better at separating the data points into distinct groups.

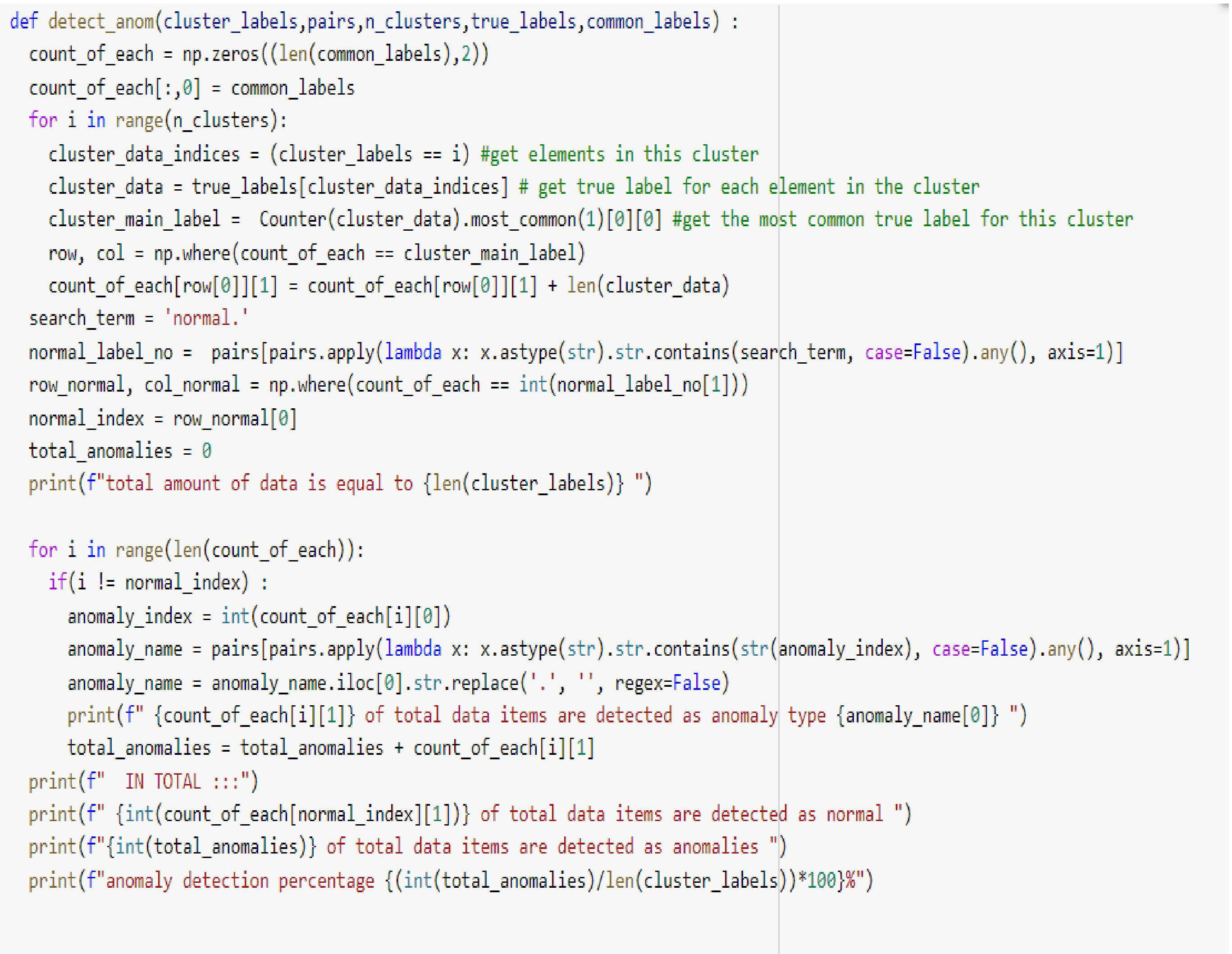
Detecting anomalies

As mentioned and explained earlier, The KDD Cup 1999 dataset includes attack traffic generated by the following 22 different tools:

1. smurf
2. neptune
3. back
4. teardrop
5. pod
6. land
7. apache2
8. udpstorm
9. processtable
10. mailbomb
11. guess_passwd
12. ftp_write
13. multihop
14. rootkit
15. buffer_overflow
16. loadmodule
17. perl
18. spy
19. warezclient
20. warezmaster
21. snmpgetattack
22. named

Which we identify all as anomalies.
So the goal here is to see how the clustering algorithms identifies them and how many normal it does.

Implementation



```
def detect_anom(cluster_labels,pairs,n_clusters,true_labels,common_labels) :
  count_of_each = np.zeros((len(common_labels),2))
  count_of_each[:,0] = common_labels
  for i in range(n_clusters):
    cluster_data_indices = (cluster_labels == i) #get elements in this cluster
    cluster_data = true_labels[cluster_data_indices] # get true label for each element in the cluster
    cluster_main_label =  Counter(cluster_data).most_common(1)[0][0] #get the most common true label for this cluster
    row, col = np.where(count_of_each == cluster_main_label)
    count_of_each[row[0]][1] = count_of_each[row[0]][1] + len(cluster_data)
  search_term = 'normal.'
  normal_label_no =  pairs[pairs.apply(lambda x: x.astype(str).str.contains(search_term, case=False).any(), axis=1)]
  row_normal, col_normal = np.where(count_of_each == int(normal_label_no[1]))
  normal_index = row_normal[0]
  total_anomalies = 0
  print(f"total amount of data is equal to {len(cluster_labels)} ")

  for i in range(len(count_of_each)):
    if(i != normal_index) :
      anomaly_index = int(count_of_each[i][0])
      anomaly_name = pairs[pairs.apply(lambda x: x.astype(str).str.contains(str(anomaly_index), case=False).any(), axis=1)]
      anomaly_name = anomaly_name.iloc[0].str.replace('.', '', regex=False)
      print(f" {count_of_each[i][1]} of total data items are detected as anomaly type {anomaly_name[0]} ")
      total_anomalies = total_anomalies + count_of_each[i][1]
  print(f"  IN TOTAL :::")
  print(f" {int(count_of_each[normal_index][1])} of total data items are detected as normal ")
  print(f"{int(total_anomalies)} of total data items are detected as anomalies ")
  print(f"anomaly detection percentage {(int(total_anomalies)/len(cluster_labels))*100}%")
```

Explanation

The function first initializes a 2D array count_of_each with the size of common_labels by 2, where the first column contains the true label values and the second column is initially filled with zeros. It then loops through all clusters, and for each cluster, it finds the most common true label using Counter from Python's collections module. The row index of the true label in count_of_each is then found using np.where, and the corresponding count value in the second column of that row is incremented by the number of data points in the current cluster.

Next, the function finds the true label for normal traffic by searching for the string "normal." in the pairs dataframe. It then finds the row index of the normal traffic true label in count_of_each, which will be used later to calculate the total number of anomalies.

The function then loops through all true labels in common_labels and checks if the true label is not the normal traffic true label. If so, the function finds the index of the corresponding row in count_of_each using np.where, and prints the number of data points that are detected as the corresponding anomaly type. The total number of anomalies is incremented by this count value. Finally, the function prints the total number of data points, the number of data points detected as normal traffic, the number of data points detected as anomalies, and the percentage of data points detected as anomalies.

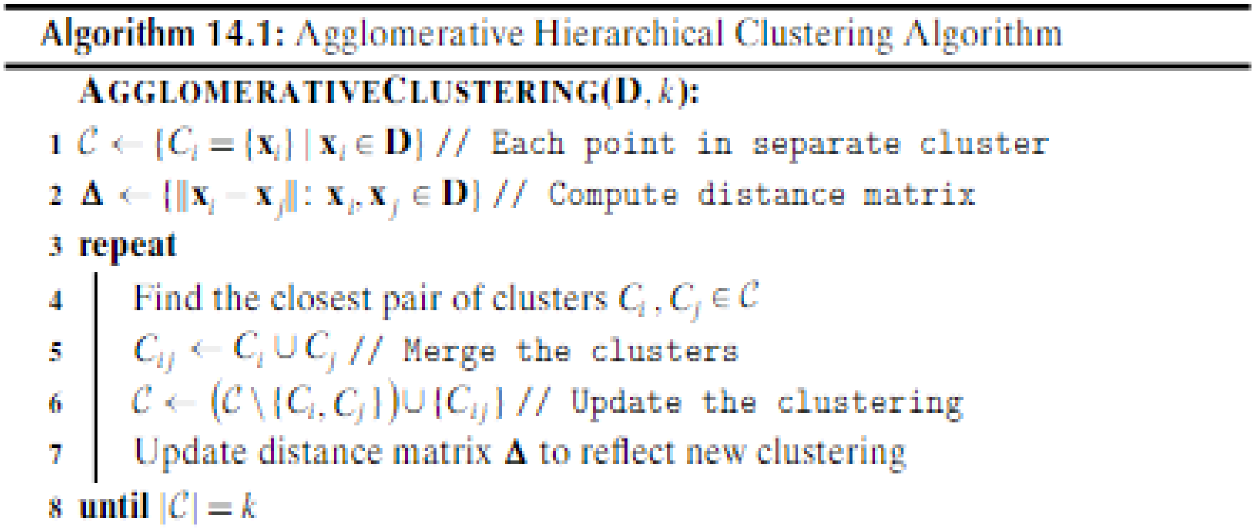
Hierarchical Agglomerative Clustering (HAC)

Description

Hierarchical Agglomerative Clustering (HAC) is a clustering technique that seeks to build a hierarchy of clusters by iteratively merging the two closest clusters based on a distance metric until a stopping criterion is met.

The algorithm starts with each data point as a separate cluster and then merges the closest pairs of clusters based on a distance metric, such as Euclidean distance or another similarity measure. The merging process continues until all the data points belong to a single cluster, or until a predefined number of clusters is reached.

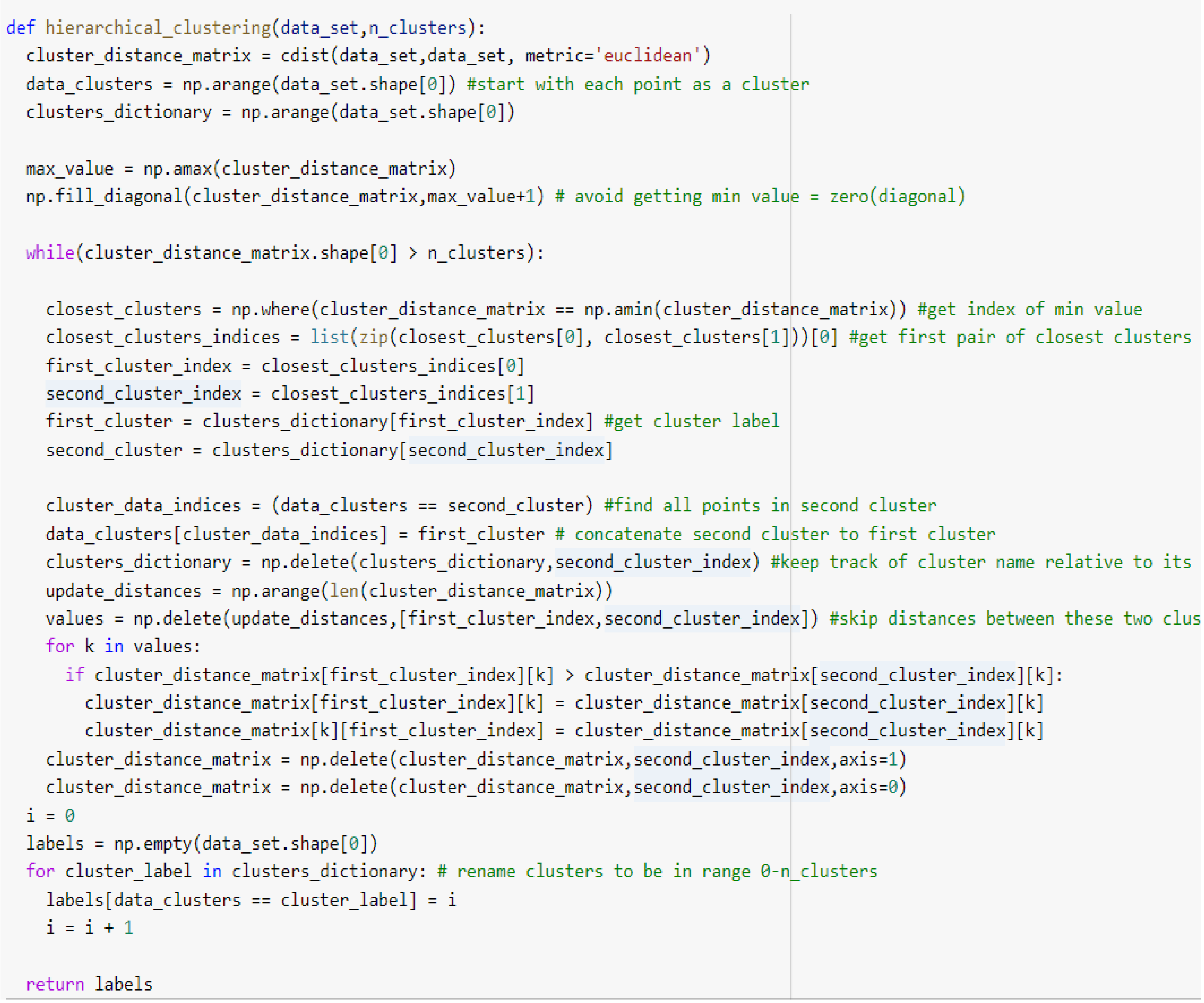
Algorithm



Algorithm 14.1: Agglomerative Hierarchical Clustering Algorithm

```
AGGLOMERATIVECLUSTERING($\mathbf{D}, k$):
$\mathcal{C} \leftarrow \{C_i = \{\mathbf{x}_i\} \mid \mathbf{x}_i \in \mathbf{D}\}$ // Each point in separate cluster
$\Delta \leftarrow \{\|\mathbf{x}_i - \mathbf{x}_j\| : \mathbf{x}_i, \mathbf{x}_j \in \mathbf{D}\}$ // Compute distance matrix
repeat
    Find the closest pair of clusters $C_i, C_j \in \mathcal{C}$
    $C_{ij} \leftarrow C_i \cup C_j$ // Merge the clusters
    $\mathcal{C} \leftarrow (\mathcal{C} \setminus \{C_i, C_j\}) \cup \{C_{ij}\}$ // Update the clustering
    Update distance matrix $\Delta$ to reflect new clustering
until $|\mathcal{C}| = k$
```

Implementation



```python
def hierarchical_clustering(data_set,n_clusters):
  cluster_distance_matrix = cdist(data_set,data_set, metric='euclidean')
  data_clusters = np.arange(data_set.shape[0]) #start with each point as a cluster
  clusters_dictionary = np.arange(data_set.shape[0])

  max_value = np.amax(cluster_distance_matrix)
  np.fill_diagonal(cluster_distance_matrix,max_value+1) # avoid getting min value = zero(diagonal)

  while(cluster_distance_matrix.shape[0] > n_clusters):

    closest_clusters = np.where(cluster_distance_matrix == np.amin(cluster_distance_matrix)) #get index of min value
    closest_clusters_indices = list(zip(closest_clusters[0], closest_clusters[1]))[0] #get first pair of closest clusters
    first_cluster_index = closest_clusters_indices[0]
    second_cluster_index = closest_clusters_indices[1]
    first_cluster = clusters_dictionary[first_cluster_index] #get cluster label
    second_cluster = clusters_dictionary[second_cluster_index]

    cluster_data_indices = (data_clusters == second_cluster) #find all points in second cluster
    data_clusters[cluster_data_indices] = first_cluster # concatenate second cluster to first cluster
    clusters_dictionary = np.delete(clusters_dictionary,second_cluster_index) #keep track of cluster name relative to its
    update_distances = np.arange(len(cluster_distance_matrix))
    values = np.delete(update_distances,[first_cluster_index,second_cluster_index]) #skip distances between these two clus
    for k in values:
      if cluster_distance_matrix[first_cluster_index][k] > cluster_distance_matrix[second_cluster_index][k]:
        cluster_distance_matrix[first_cluster_index][k] = cluster_distance_matrix[second_cluster_index][k]
        cluster_distance_matrix[k][first_cluster_index] = cluster_distance_matrix[second_cluster_index][k]
    cluster_distance_matrix = np.delete(cluster_distance_matrix,second_cluster_index,axis=1)
    cluster_distance_matrix = np.delete(cluster_distance_matrix,second_cluster_index,axis=0)
  i = 0
  labels = np.empty(data_set.shape[0])
  for cluster_label in clusters_dictionary: # rename clusters to be in range 0-n_clusters
    labels[data_clusters == cluster_label] = i
    i = i + 1

  return labels
```

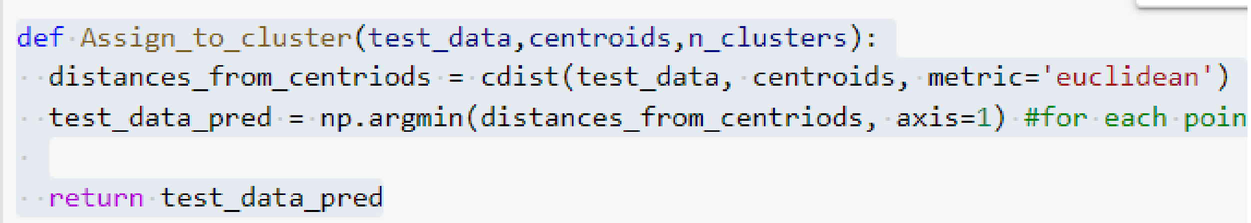
Explanation

First, the function calculates the distance matrix between all pairs of data points using the Euclidean distance metric. It then initializes each point as its own cluster and maintains a dictionary to keep track of the cluster names relative to their index in the distance matrix.
In each iteration, the two closest clusters are identified and merged by concatenating the points of the second cluster to the first cluster. The distance matrix is updated by replacing the distances between the first cluster and other clusters with the minimum distance between the first and second clusters. The distance matrix is then reduced in size by removing the row and

column of the second cluster, and the process continues until the desired number of clusters is reached.

Finally, the code assigns a label to each data point based on its corresponding cluster index in the clusters dictionary, and returns an array of labels for each data point indicating its assigned cluster.

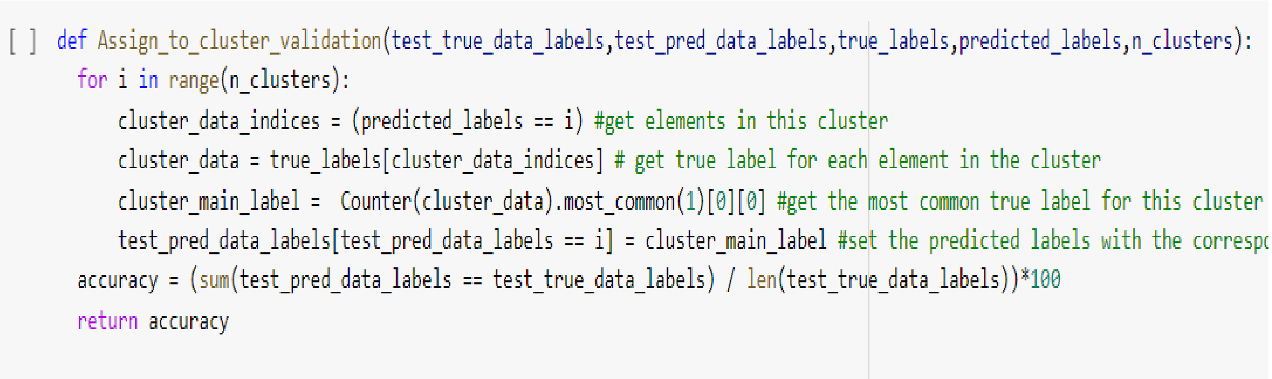
Combining test data to the train clusters



```
def Assign_to_cluster(test_data,centroids,n_clusters):
  distances_from_centriods = cdist(test_data, centroids, metric='euclidean')
  test_data_pred = np.argmin(distances_from_centriods, axis=1) #for each poin

  return test_data_pred
```

The function assigns each data point to its closest centroid by finding the index of the minimum distance in each row of distances_from_centroids. The resulting assignments are returned as test_data_pred, which is a one-dimensional array of length equal to the number of data points in test_data

Get accuracy of clustering by comparing assigned labels to true ones

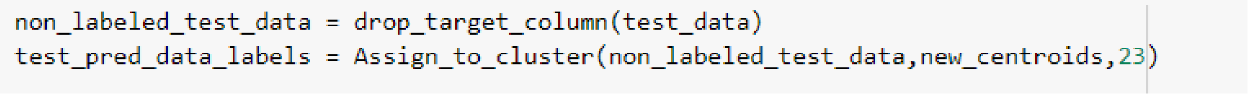


```
[ ] def Assign_to_cluster_validation(test_true_data_labels,test_pred_data_labels,true_labels,predicted_labels,n_clusters):
      for i in range(n_clusters):
          cluster_data_indices = (predicted_labels == i) #get elements in this cluster
          cluster_data = true_labels[cluster_data_indices] # get true label for each element in the cluster
          cluster_main_label =  Counter(cluster_data).most_common(1)[0][0] #get the most common true label for this cluster
          test_pred_data_labels[test_pred_data_labels == i] = cluster_main_label #set the predicted labels with the correspc
      accuracy = (sum(test_pred_data_labels == test_true_data_labels) / len(test_true_data_labels))*100
      return accuracy
```

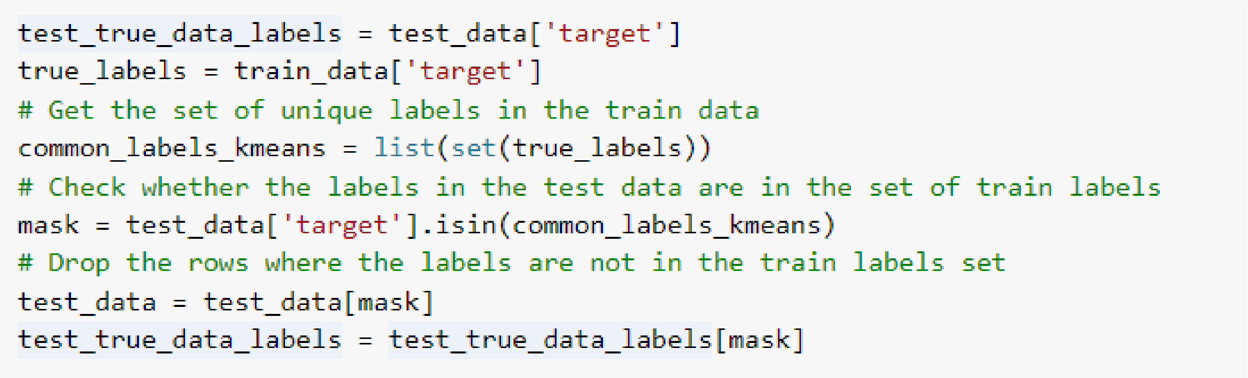
The function first loops through each cluster (i) and gets the indices of the elements in that cluster (cluster_data_indices). It then extracts the true labels of the elements in that cluster (cluster_data) and finds the most common true label (cluster_main_label) for that cluster using the Counter class from the collections module.

Next, the function sets the predicted labels for the elements in the current cluster to be the corresponding cluster_main_label. This is done by setting the predicted labels with cluster index i to cluster_main_label.

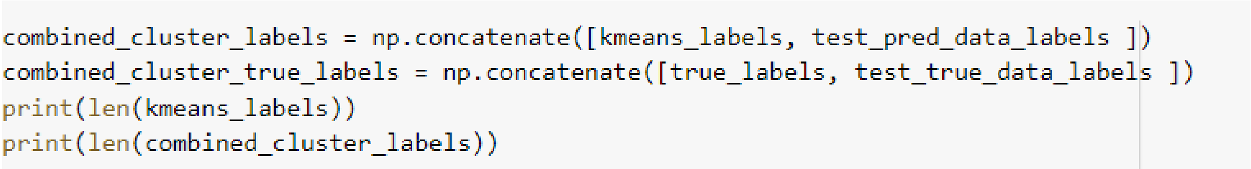
Finally, the function calculates the accuracy of the predicted labels of the test data by comparing them with the true labels of the test data. The accuracy is calculated as the percentage of correctly predicted labels out of the total number of test data points. The result is returned as accuracy.



```
non_labeled_test_data = drop_target_column(test_data)
test_pred_data_labels = Assign_to_cluster(non_labeled_test_data,new_centroids,23)
```

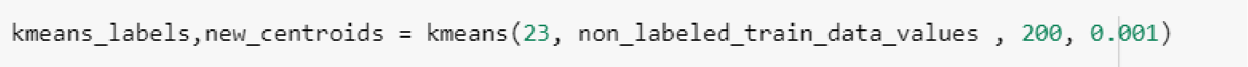


```
test_true_data_labels = test_data['target']
true_labels = train_data['target']
# Get the set of unique labels in the train data
common_labels_kmeans = list(set(true_labels))
# Check whether the labels in the test data are in the set of train labels
mask = test_data['target'].isin(common_labels_kmeans)
# Drop the rows where the labels are not in the train labels set
test_data = test_data[mask]
test_true_data_labels = test_true_data_labels[mask]
```

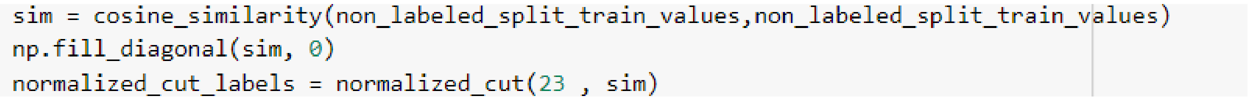


```
combined_cluster_labels = np.concatenate([kmeans_labels, test_pred_data_labels ])
combined_cluster_true_labels = np.concatenate([true_labels, test_true_data_labels ])
print(len(kmeans_labels))
print(len(combined_cluster_labels))
```

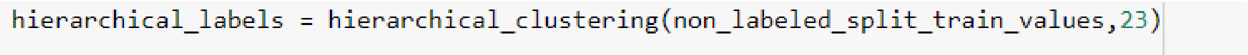
Running Clustering using previous described functions and prepared data



```
kmeans_labels,new_centroids = kmeans(23, non_labeled_train_data_values , 200, 0.001)
```

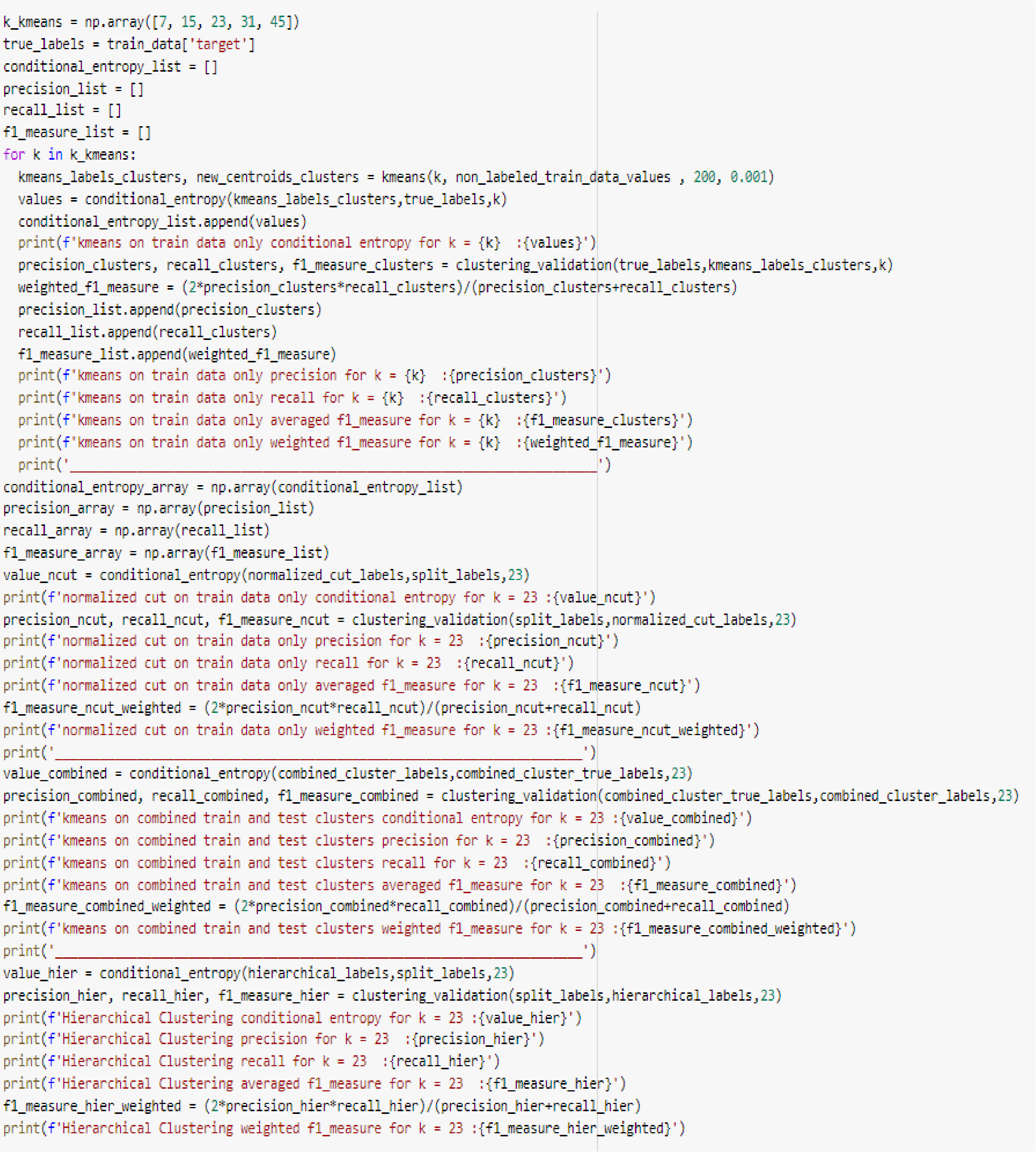


```
sim = cosine_similarity(non_labeled_split_train_values,non_labeled_split_train_values)
np.fill_diagonal(sim, 0)
normalized_cut_labels = normalized_cut(23 , sim)
```



```
hierarchical_labels = hierarchical_clustering(non_labeled_split_train_values,23)
```

Getting clustering evaluation for different ks in K means, normalized cut, hierarchical , and on combined test and train clusters



```
k_kmeans = np.array([7, 15, 23, 31, 45])
true_labels = train_data['target']
conditional_entropy_list = []
precision_list = []
recall_list = []
f1_measure_list = []
for k in k_kmeans:
  kmeans_labels_clusters, new_centroids_clusters = kmeans(k, non_labeled_train_data_values , 200, 0.001)
  values = conditional_entropy(kmeans_labels_clusters,true_labels,k)
  conditional_entropy_list.append(values)
  print(f'kmeans on train data only conditional entropy for k = {k}  :{values}')
  precision_clusters, recall_clusters, f1_measure_clusters = clustering_validation(true_labels,kmeans_labels_clusters,k)
  weighted_f1_measure = (2*precision_clusters*recall_clusters)/(precision_clusters+recall_clusters)
  precision_list.append(precision_clusters)
  recall_list.append(recall_clusters)
  f1_measure_list.append(weighted_f1_measure)
  print(f'kmeans on train data only precision for k = {k}  :{precision_clusters}')
  print(f'kmeans on train data only recall for k = {k}  :{recall_clusters}')
  print(f'kmeans on train data only averaged f1_measure for k = {k}  :{f1_measure_clusters}')
  print(f'kmeans on train data only weighted f1_measure for k = {k}  :{weighted_f1_measure}')
  print('_______________________________________________________________________')
conditional_entropy_array = np.array(conditional_entropy_list)
precision_array = np.array(precision_list)
recall_array = np.array(recall_list)
f1_measure_array = np.array(f1_measure_list)
value_ncut = conditional_entropy(normalized_cut_labels,split_labels,23)
print(f'normalized cut on train data only conditional entropy for k = 23 :{value_ncut}')
precision_ncut, recall_ncut, f1_measure_ncut = clustering_validation(split_labels,normalized_cut_labels,23)
print(f'normalized cut on train data only precision for k = 23  :{precision_ncut}')
print(f'normalized cut on train data only recall for k = 23  :{recall_ncut}')
print(f'normalized cut on train data only averaged f1_measure for k = 23  :{f1_measure_ncut}')
f1_measure_ncut_weighted = (2*precision_ncut*recall_ncut)/(precision_ncut+recall_ncut)
print(f'normalized cut on train data only weighted f1_measure for k = 23 :{f1_measure_ncut_weighted}')
print('_______________________________________________________________________')
value_combined = conditional_entropy(combined_cluster_labels,combined_cluster_true_labels,23)
precision_combined, recall_combined, f1_measure_combined = clustering_validation(combined_cluster_true_labels,combined_cluster_labels,23)
print(f'kmeans on combined train and test clusters conditional entropy for k = 23 :{value_combined}')
print(f'kmeans on combined train and test clusters precision for k = 23  :{precision_combined}')
print(f'kmeans on combined train and test clusters recall for k = 23  :{recall_combined}')
print(f'kmeans on combined train and test clusters averaged f1_measure for k = 23  :{f1_measure_combined}')
f1_measure_combined_weighted = (2*precision_combined*recall_combined)/(precision_combined+recall_combined)
print(f'kmeans on combined train and test clusters weighted f1_measure for k = 23 :{f1_measure_combined_weighted}')
print('_______________________________________________________________________')
value_hier = conditional_entropy(hierarchical_labels,split_labels,23)
precision_hier, recall_hier, f1_measure_hier = clustering_validation(split_labels,hierarchical_labels,23)
print(f'Hierarchical Clustering conditional entropy for k = 23 :{value_hier}')
print(f'Hierarchical Clustering precision for k = 23  :{precision_hier}')
print(f'Hierarchical Clustering recall for k = 23  :{recall_hier}')
print(f'Hierarchical Clustering averaged f1_measure for k = 23  :{f1_measure_hier}')
f1_measure_hier_weighted = (2*precision_hier*recall_hier)/(precision_hier+recall_hier)
print(f'Hierarchical Clustering weighted f1_measure for k = 23 :{f1_measure_hier_weighted}')
```

Evaluation Outputs

```
kmeans on train data only conditional entropy for k = 7 :0.851738690791461
kmeans on train data only precision for k = 7 :0.6068165895072328
kmeans on train data only recall for k = 7 :0.9777445
kmeans on train data only averaged f1_measure for k = 7 :0.35334183
 kmeans on train data only weighted f1_measure for k = 7 :0.74886555
```

```
 kmeans on train data only conditional entropy for k = 15 :0.7942444571016202
kmeans on train data only precision for k = 15 :0.6093168299149643
kmeans on train data only recall for k = 15 :0.74338716
kmeans on train data only averaged f1_measure for k = 15 :0.18940071
kmeans on train data only weighted f1_measure for k = 15 :0.66970795
```

```
kmeans on train data only conditional entropy for k = 23 :0.6556523843483423
kmeans on train data only precision for k = 23 :0.6579478796038082
kmeans on train data only recall for k = 23 :0.73375688
kmeans on train data only averaged f1_measure for k = 23 :0.14790925
kmeans on train data only weighted f1_measure for k = 23 :0.69378764
```

```
kmeans on train data only conditional entropy for k = 31 :0.4554062040123989
kmeans on train data only precision for k = 31 :0.8462764276784855
kmeans on train data only recall for k = 31 :0.57865685
kmeans on train data only averaged f1_measure for k = 31 :0.13714868
kmeans on train data only weighted f1_measure for k = 31 :0.68733555
```

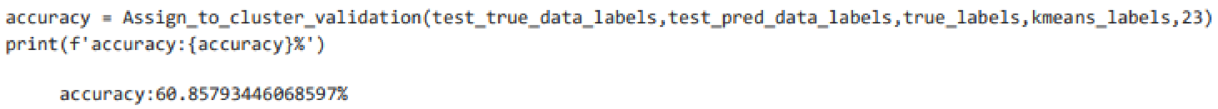
```
kmeans on train data only conditional entropy for k = 45 :0.1784441159200027
kmeans on train data only precision for k = 45 :0.9584094624483125
kmeans on train data only recall for k = 45 :0.20927238
kmeans on train data only averaged f1_measure for k = 45 :0.15399331
kmeans on train data only weighted f1_measure for k = 45 :0.34353301
```

```
normalized cut on train data only conditional entropy for k = 23 :0.22693097111691535
normalized cut on train data only precision for k = 23 :0.9455894476504534
normalized cut on train data only recall for k = 23 :0.20070266
normalized cut on train data only averaged f1_measure for k = 23 :0.13999656
normalized cut on train data only weighted f1_measure for k = 23 :0.33112383
```

```
kmeans on combined train and test clusters conditional entropy for k = 23 :0.6875418824139613
kmeans on combined train and test clusters precision for k = 23 :0.6577551346782117
kmeans on combined train and test clusters recall for k = 23 :0.4113767
kmeans on combined train and test clusters averaged f1_measure for k = 23 :0.13114442
kmeans on combined train and test clusters weighted f1_measure for k = 23 :0.50617733
```

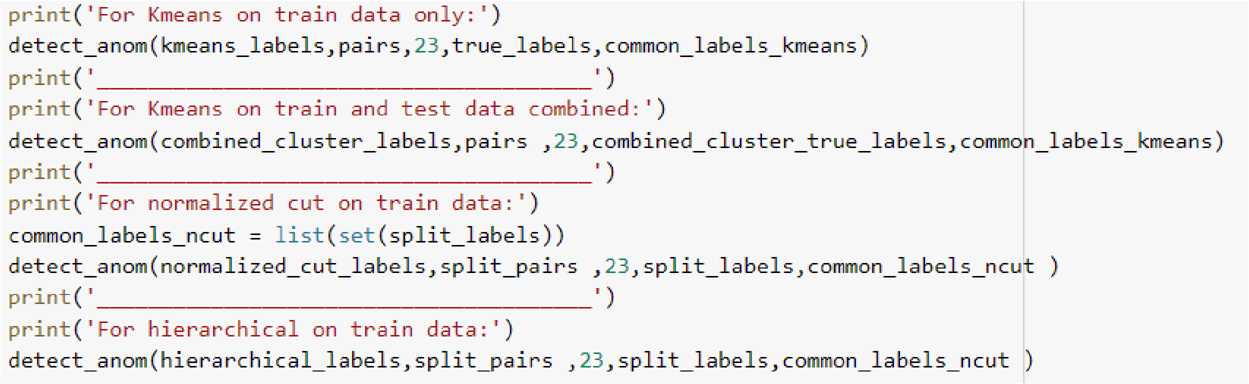
```
Hierarchical Clustering conditional entropy for k = 23 :0.8452998341312433
Hierarchical Clustering precision for k = 23 :0.6106073097004678
Hierarchical Clustering recall for k = 23 :0.98159216
Hierarchical Clustering averaged f1_measure for k = 23 :[0.09065632]
Hierarchical Clustering weighted f1_measure for k = 23 :0.75287972
```

Getting accuracy by comparing true labels to assigned labels



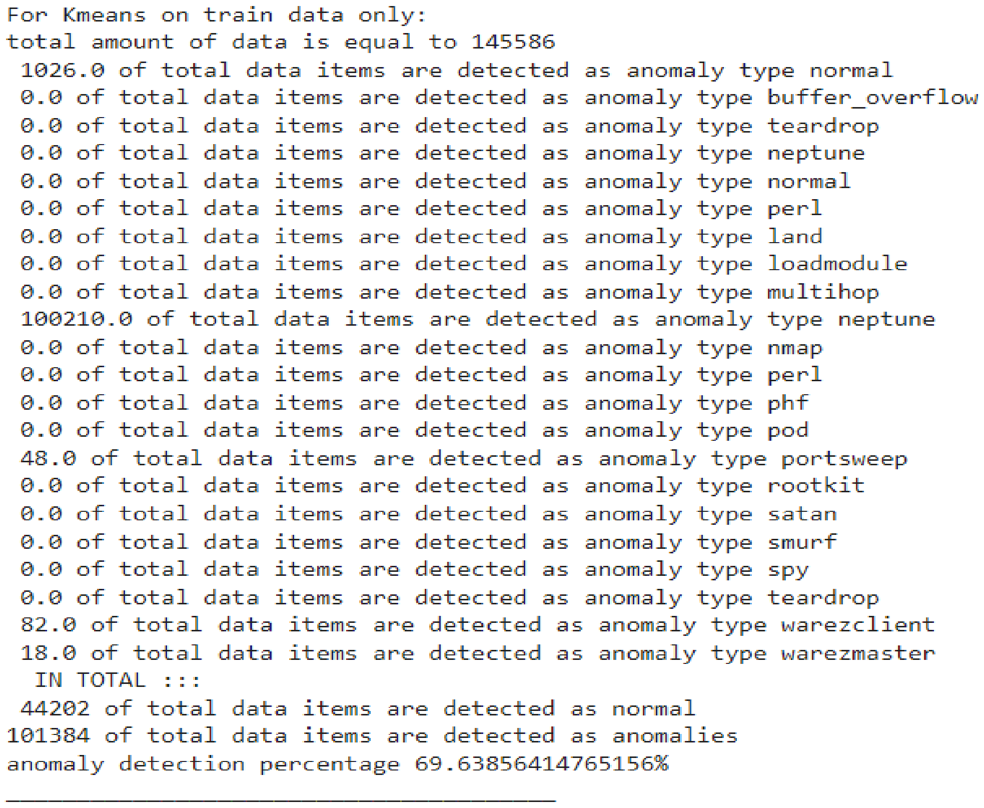
```
accuracy = Assign_to_cluster_validation(test_true_data_labels,test_pred_data_labels,true_labels,kmeans_labels,23)
print(f'accuracy:{accuracy}%')

     accuracy:60.85793446068597%
```

Comparing the different algorithms by Detecting anomalies

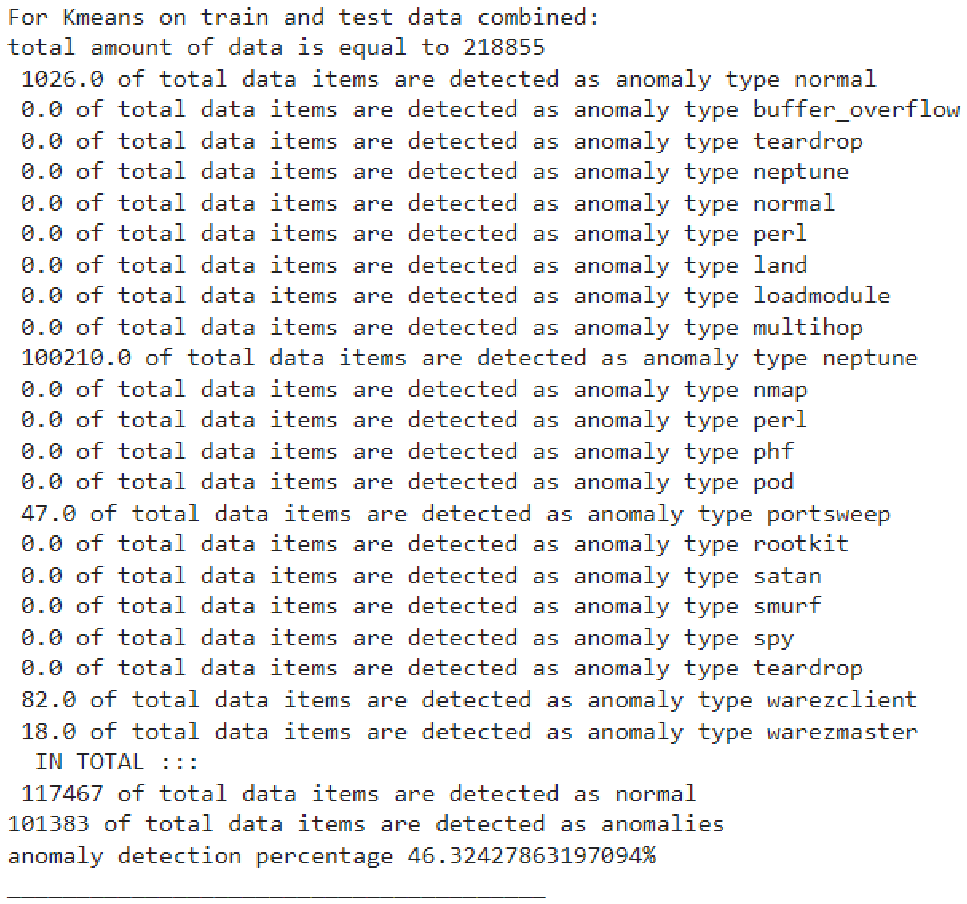


```
print('For Kmeans on train data only:')
detect_anom(kmeans_labels,pairs,23,true_labels,common_labels_kmeans)
print('________________________________________')
print('For Kmeans on train and test data combined:')
detect_anom(combined_cluster_labels,pairs ,23,combined_cluster_true_labels,common_labels_kmeans)
print('________________________________________')
print('For normalized cut on train data:')
common_labels_ncut = list(set(split_labels))
detect_anom(normalized_cut_labels,split_pairs ,23,split_labels,common_labels_ncut )
print('________________________________________')
print('For hierarchical on train data:')
detect_anom(hierarchical_labels,split_pairs ,23,split_labels,common_labels_ncut )
```

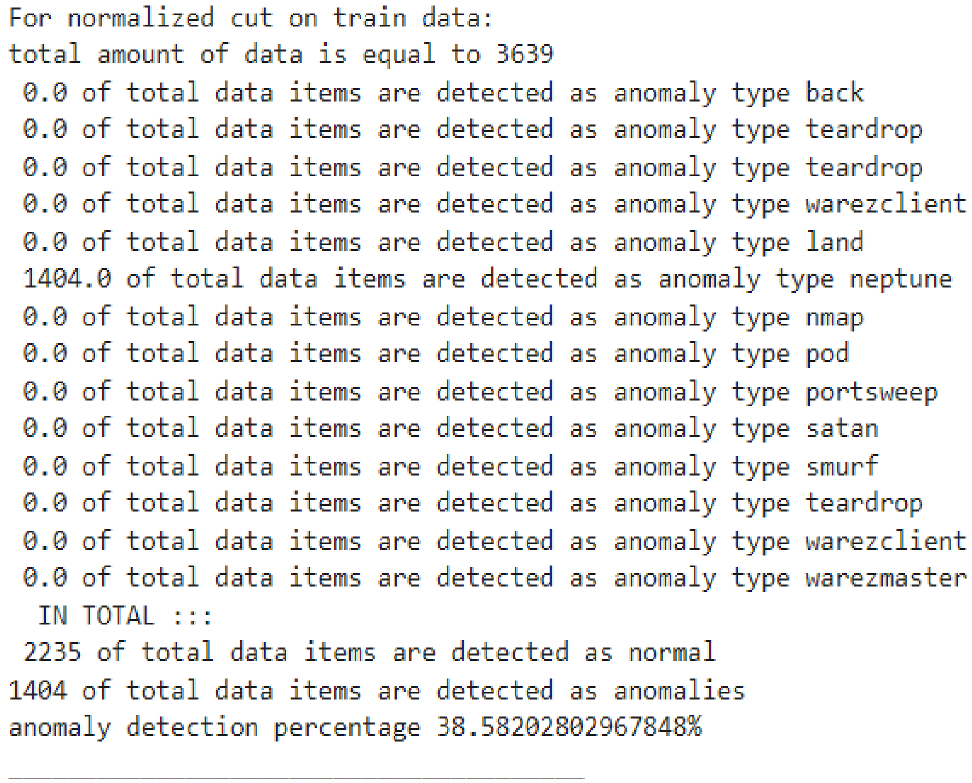
Output:



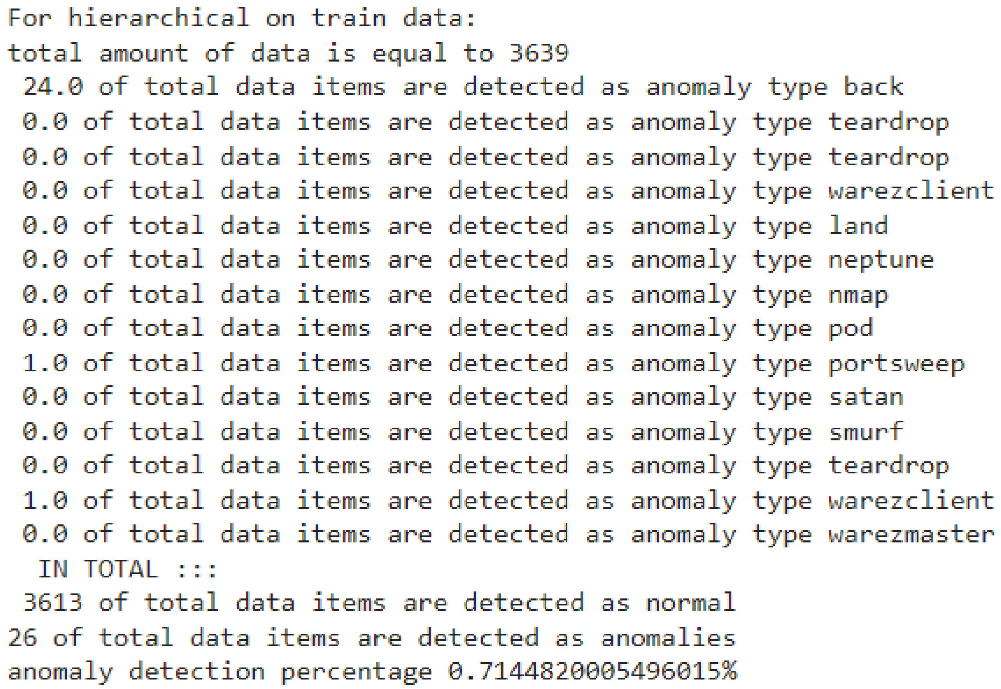
```
For Kmeans on train data only:
total amount of data is equal to 145586
 1026.0 of total data items are detected as anomaly type normal
 0.0 of total data items are detected as anomaly type buffer_overflow
 0.0 of total data items are detected as anomaly type teardrop
 0.0 of total data items are detected as anomaly type neptune
 0.0 of total data items are detected as anomaly type normal
 0.0 of total data items are detected as anomaly type perl
 0.0 of total data items are detected as anomaly type land
 0.0 of total data items are detected as anomaly type loadmodule
 0.0 of total data items are detected as anomaly type multihop
 100210.0 of total data items are detected as anomaly type neptune
 0.0 of total data items are detected as anomaly type nmap
 0.0 of total data items are detected as anomaly type perl
 0.0 of total data items are detected as anomaly type phf
 0.0 of total data items are detected as anomaly type pod
 48.0 of total data items are detected as anomaly type portsweep
 0.0 of total data items are detected as anomaly type rootkit
 0.0 of total data items are detected as anomaly type satan
 0.0 of total data items are detected as anomaly type smurf
 0.0 of total data items are detected as anomaly type spy
 0.0 of total data items are detected as anomaly type teardrop
 82.0 of total data items are detected as anomaly type warezclient
 18.0 of total data items are detected as anomaly type warezmaster
  IN TOTAL :::
 44202 of total data items are detected as normal
101384 of total data items are detected as anomalies
anomaly detection percentage 69.63856414765156%

_______________________________________
```



```
For Kmeans on train and test data combined:
total amount of data is equal to 218855
 1026.0 of total data items are detected as anomaly type normal
 0.0 of total data items are detected as anomaly type buffer_overflow
 0.0 of total data items are detected as anomaly type teardrop
 0.0 of total data items are detected as anomaly type neptune
 0.0 of total data items are detected as anomaly type normal
 0.0 of total data items are detected as anomaly type perl
 0.0 of total data items are detected as anomaly type land
 0.0 of total data items are detected as anomaly type loadmodule
 0.0 of total data items are detected as anomaly type multihop
 100210.0 of total data items are detected as anomaly type neptune
 0.0 of total data items are detected as anomaly type nmap
 0.0 of total data items are detected as anomaly type perl
 0.0 of total data items are detected as anomaly type phf
 0.0 of total data items are detected as anomaly type pod
 47.0 of total data items are detected as anomaly type portsweep
 0.0 of total data items are detected as anomaly type rootkit
 0.0 of total data items are detected as anomaly type satan
 0.0 of total data items are detected as anomaly type smurf
 0.0 of total data items are detected as anomaly type spy
 0.0 of total data items are detected as anomaly type teardrop
 82.0 of total data items are detected as anomaly type warezclient
 18.0 of total data items are detected as anomaly type warezmaster
  IN TOTAL :::
 117467 of total data items are detected as normal
101383 of total data items are detected as anomalies
anomaly detection percentage 46.32427863197094%
________________________________________
```

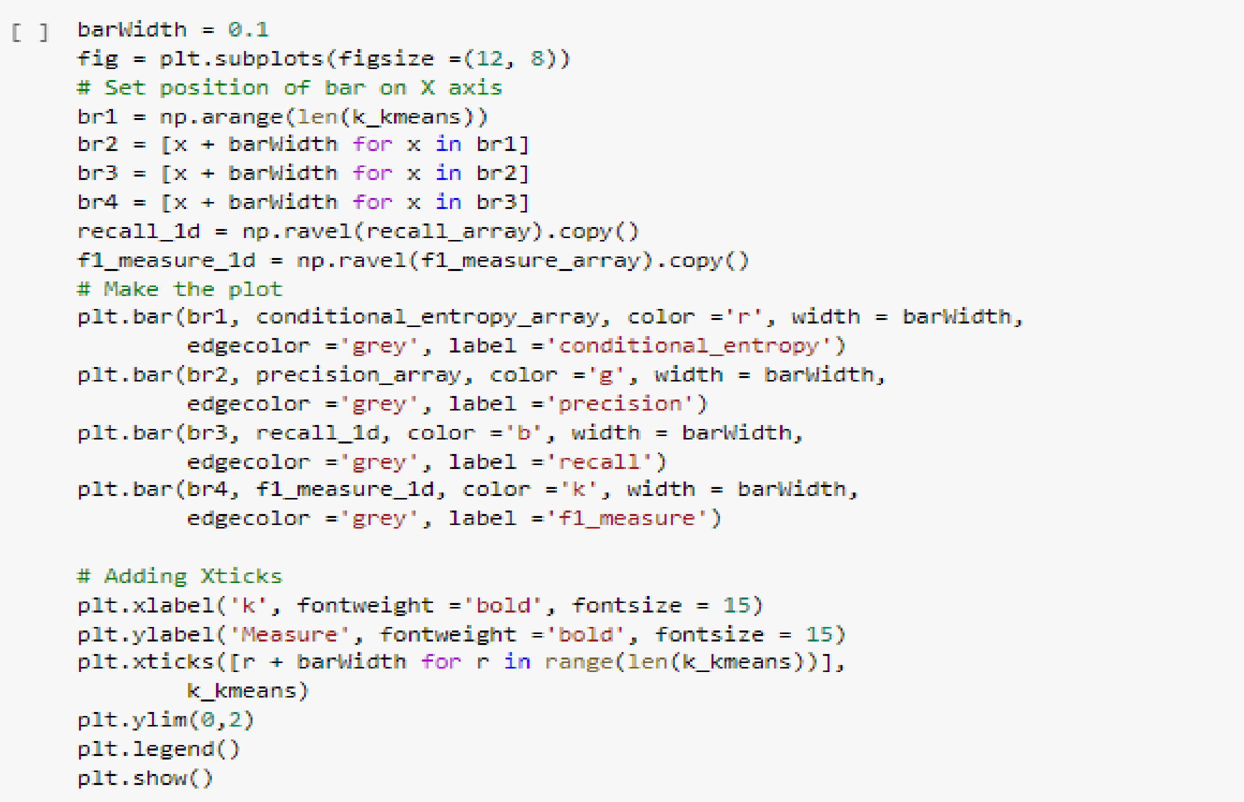


```
For normalized cut on train data:
total amount of data is equal to 3639
 0.0 of total data items are detected as anomaly type back
 0.0 of total data items are detected as anomaly type teardrop
 0.0 of total data items are detected as anomaly type teardrop
 0.0 of total data items are detected as anomaly type warezclient
 0.0 of total data items are detected as anomaly type land
 1404.0 of total data items are detected as anomaly type neptune
 0.0 of total data items are detected as anomaly type nmap
 0.0 of total data items are detected as anomaly type pod
 0.0 of total data items are detected as anomaly type portsweep
 0.0 of total data items are detected as anomaly type satan
 0.0 of total data items are detected as anomaly type smurf
 0.0 of total data items are detected as anomaly type teardrop
 0.0 of total data items are detected as anomaly type warezclient
 0.0 of total data items are detected as anomaly type warezmaster
  IN TOTAL :::
 2235 of total data items are detected as normal
1404 of total data items are detected as anomalies
anomaly detection percentage 38.5820280296784%
________________________________________
```

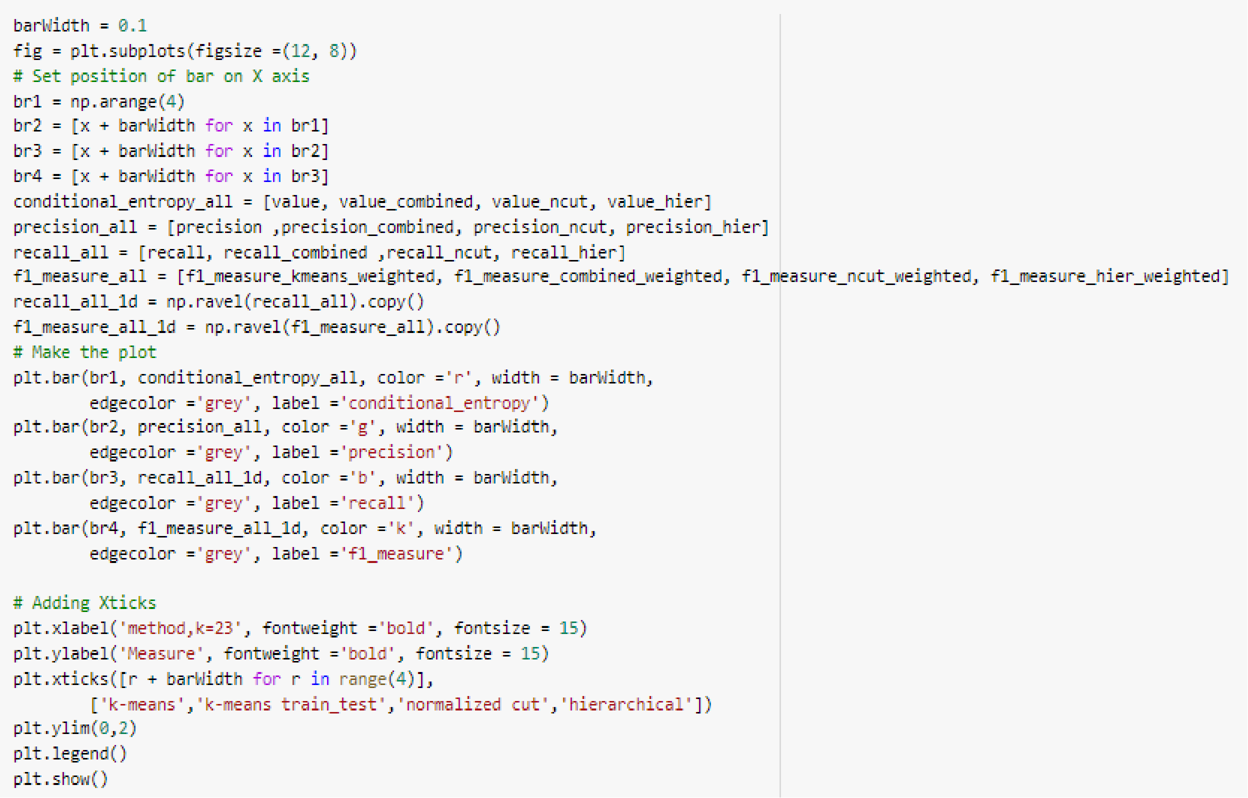


```
For hierarchical on train data:
total amount of data is equal to 3639
 24.0 of total data items are detected as anomaly type back
 0.0 of total data items are detected as anomaly type teardrop
 0.0 of total data items are detected as anomaly type teardrop
 0.0 of total data items are detected as anomaly type warezclient
 0.0 of total data items are detected as anomaly type land
 0.0 of total data items are detected as anomaly type neptune
 0.0 of total data items are detected as anomaly type nmap
 0.0 of total data items are detected as anomaly type pod
 1.0 of total data items are detected as anomaly type portsweep
 0.0 of total data items are detected as anomaly type satan
 0.0 of total data items are detected as anomaly type smurf
 0.0 of total data items are detected as anomaly type teardrop
 1.0 of total data items are detected as anomaly type warezclient
 0.0 of total data items are detected as anomaly type warezmaster
  IN TOTAL :::
 3613 of total data items are detected as normal
26 of total data items are detected as anomalies
anomaly detection percentage 0.7144820005496015%
```

Bar Charts

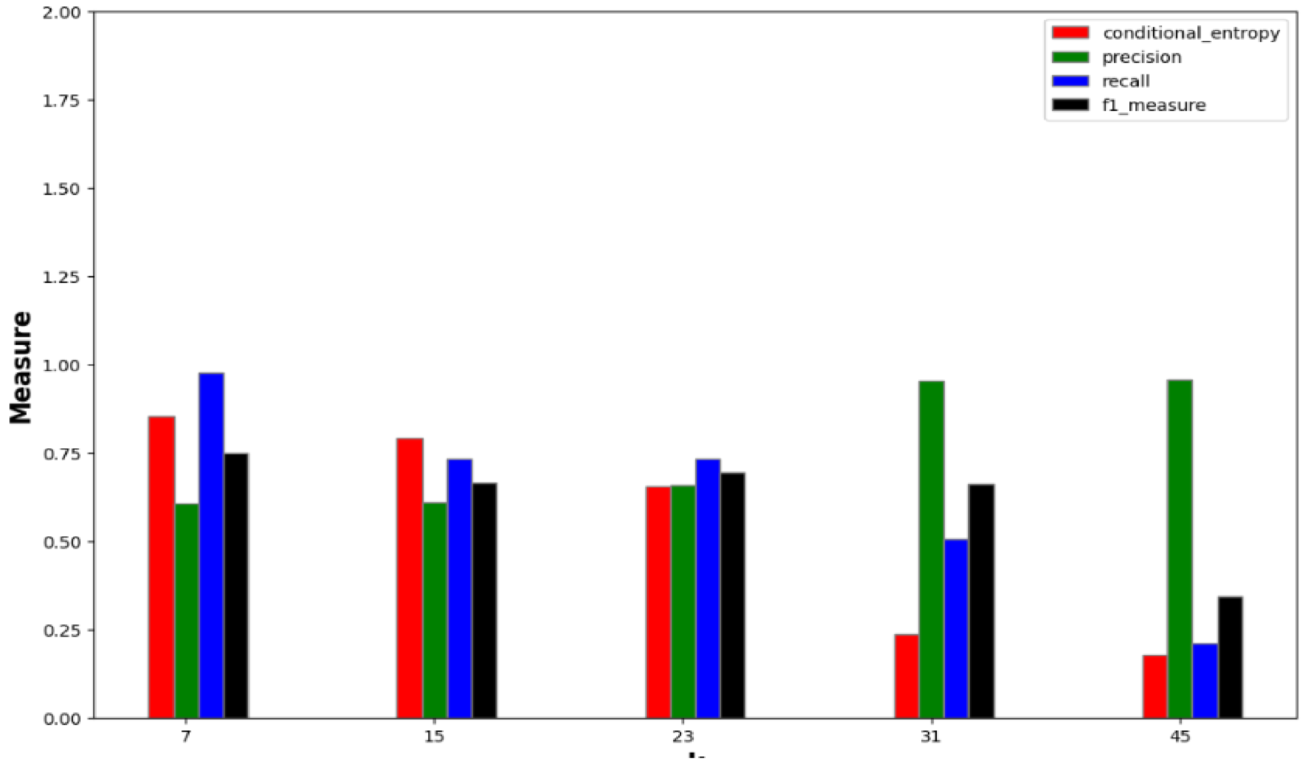


```
barWidth = 0.1
fig = plt.subplots(figsize =(12, 8))
# Set position of bar on X axis
br1 = np.arange(len(k_kmeans))
br2 = [x + barWidth for x in br1]
br3 = [x + barWidth for x in br2]
br4 = [x + barWidth for x in br3]
recall_1d = np.ravel(recall_array).copy()
f1_measure_1d = np.ravel(f1_measure_array).copy()
# Make the plot
plt.bar(br1, conditional_entropy_array, color ='r', width = barWidth,
    edgecolor ='grey', label ='conditional_entropy')
plt.bar(br2, precision_array, color ='g', width = barWidth,
    edgecolor ='grey', label ='precision')
plt.bar(br3, recall_1d, color ='b', width = barWidth,
    edgecolor ='grey', label ='recall')
plt.bar(br4, f1_measure_1d, color ='k', width = barWidth,
    edgecolor ='grey', label ='f1_measure')

# Adding Xticks
plt.xlabel('k', fontweight ='bold', fontsize = 15)
plt.ylabel('Measure', fontweight ='bold', fontsize = 15)
plt.xticks([r + barWidth for r in range(len(k_kmeans))],
    k_kmeans)
plt.ylim(0,2)
plt.legend()
plt.show()
```

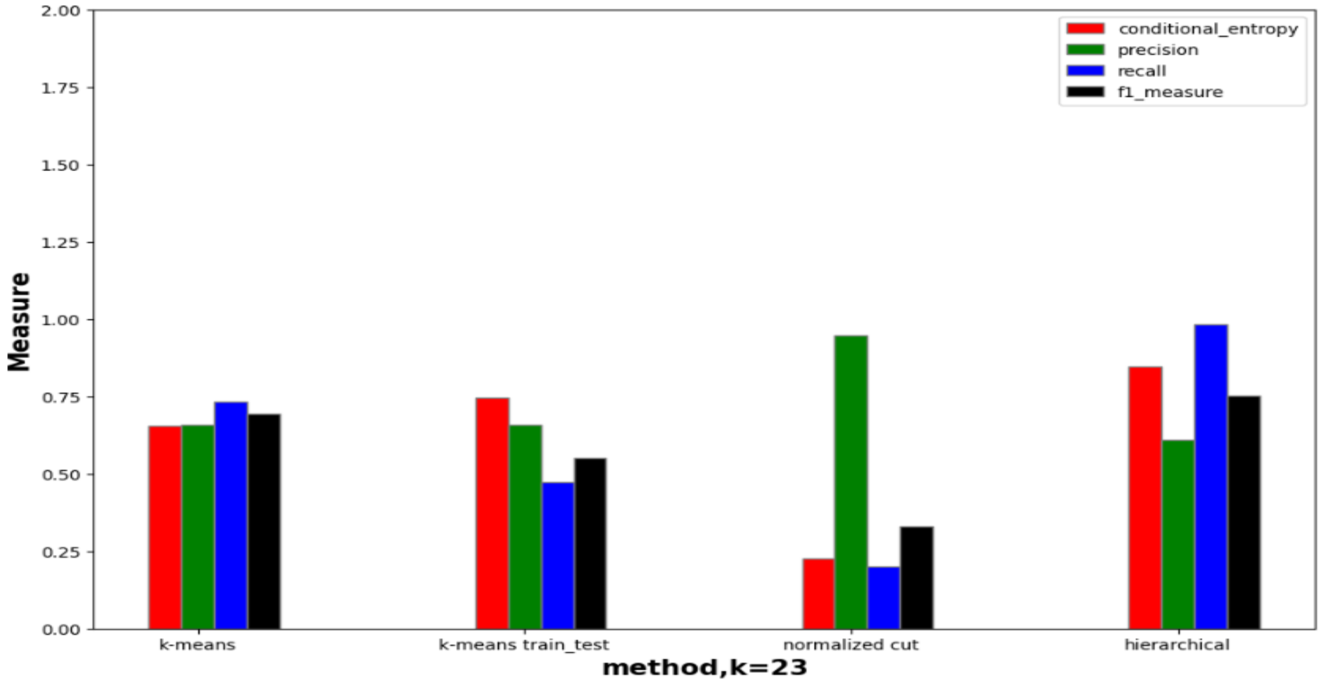


```
barWidth = 0.1
fig = plt.subplots(figsize =(12, 8))
# Set position of bar on X axis
br1 = np.arange(4)
br2 = [x + barWidth for x in br1]
br3 = [x + barWidth for x in br2]
br4 = [x + barWidth for x in br3]
conditional_entropy_all = [value, value_combined, value_ncut, value_hier]
precision_all = [precision ,precision_combined, precision_ncut, precision_hier]
recall_all = [recall, recall_combined ,recall_ncut, recall_hier]
f1_measure_all = [f1_measure_kmeans_weighted, f1_measure_combined_weighted, f1_measure_ncut_weighted, f1_measure_hier_weighted]
recall_all_1d = np.ravel(recall_all).copy()
f1_measure_all_1d = np.ravel(f1_measure_all).copy()
# Make the plot
plt.bar(br1, conditional_entropy_all, color ='r', width = barWidth,
        edgecolor ='grey', label ='conditional_entropy')
plt.bar(br2, precision_all, color ='g', width = barWidth,
        edgecolor ='grey', label ='precision')
plt.bar(br3, recall_all_1d, color ='b', width = barWidth,
        edgecolor ='grey', label ='recall')
plt.bar(br4, f1_measure_all_1d, color ='k', width = barWidth,
        edgecolor ='grey', label ='f1_measure')

# Adding Xticks
plt.xlabel('method,k=23', fontweight ='bold', fontsize = 15)
plt.ylabel('Measure', fontweight ='bold', fontsize = 15)
plt.xticks([r + barWidth for r in range(4)],
        ['k-means','k-means train_test','normalized cut','hierarchical'])
plt.ylim(0,2)
plt.legend()
plt.show()
```

Comparison between the Evaluation measures against different ks



Comparison between the Evaluation measures against different clustering algorithms


Measure
conditional_entropy
precision
recall
f1_measure
k-means
k-means train_test
normalized cut
hierarchical
method,k=23