NETWORK ANOMALY DETECTION

COLLABORATORS

i	Name	ID
1	Zeyad Zidan	19015709
2	Abdelaziz Mohammed	19015941
3	Omar Khairat	19016063

Problem Statement

The exponential growth of network traffic has led to increased network anomalies, such as cyber-attacks, network failures, and hardware malfunctions. Network anomaly detection is a critical task for maintaining the security and stability of computer networks. This assignment aims to help students understand how K-Means and Normalized Cut algorithms can be used for network anomaly detection.

Understanding Dataset

After downloading the dataset and understanding the format, we split the training and testing data into training data, testing data, training labels, and testing labels.

Now the data is categorical, but to perform K-Means, Normalized Cut, and DB Scan it is required to convert the data into numerical data.

A function to convert each training feature and its corresponding testing feature from categorical to numerical form is provided below.

```python
import numpy as np
from sklearn.preprocessing import LabelEncoder

def cat_to_num(trcolumn, tscolumn):
    """
    Converts 2 categorical columns of the same types into numerical columns

    Args:
        trcolumn (ndarray): ndarray of values of the first column.
        tscolumn (ndarray): ndarray of values of the second column.

    Returns:
        tuple: a tuple of 2 ndarrays
    """
    encoder = LabelEncoder()
    categories = set(np.unique(trcolumn)).union(set(np.unique(tscolumn)))
    encoder.fit(list(categories))
    return encoder.transform(trcolumn), encoder.transform(tscolumn)
```

Now the function shall convert the categorical features to numerical ones.

```python
# Copy the data into another dataframe to convert its categorical values into numerical.
num_training = training.copy()
num_testing = testing.copy()

# Convert the categorical features.
for i in range(1, 4):
    values = cat_to_num(num_training.iloc[:, i].values, num_testing.iloc[:, i].values)
    num_training.isetitem(i, values[0])
    num_testing.isetitem(i, values[1])

# Convert the labels.
num_trlabels, num_tslabels = cat_to_num(trlabels, tslabels)
```

K-Means

The code for K-Means is provided below in which :

This code implements the K-means algorithm to cluster data into k labels. The algorithm iteratively updates the centroids by assigning each data point to the nearest centroid and then computing the mean of the data points assigned to each centroid.

The function takes three arguments: X (the data matrix), k (the number of clusters), and epsilon (a tolerance fraction for convergence). It returns the final centroids after performing K-means on the data.

The algorithm initializes the centroids by randomly choosing k data points from X. Then it computes the Euclidean distance from each sample to each centroid, assigns each sample to the nearest centroid, and updates the centroids by computing the mean of the assigned samples. It repeats this process until the centroids do not change by more than epsilon.

```python
def k_means(X, k, epsilon):

    n_samples, n_features = X.shape

    # Randomly choose k data points as the initial centroids
    centroids = X[np.random.choice(n_samples, k, replace=False)]
    distances = np.zeros((n_samples, k))
    labels = np.zeros(n_samples)
    old_centroids = np.zeros((k, n_features))

    # Continue until the centroids don't change by more than epsilon
    while np.linalg.norm(centroids - old_centroids) > epsilon:
        old_centroids = centroids.copy()

        # Calculate the Euclidean distances from each sample to each centroid
        for i in range(k):
            distances[:, i] = np.linalg.norm(X - centroids[i], axis=1)

        # Assign each sample to the nearest centroid
        labels = np.argmin(distances, axis=1)

        # Update the centroids to be the mean of the samples assigned to them
        for i in range(k):
            X_i = X[labels == i]
            if len(X_i) == 0:
                centroids[i] = old_centroids[i]
            else:
                centroids[i] = np.mean(X_i, axis=0)

    return centroids
```

```python
from sklearn.metrics.cluster import contingency_matrix
from scipy.optimize import linear_sum_assignment

labels = []
train = np.array(num_training_10)
test = np.array(num_testing_10)
results = []
for K in [7,15,23,31,45]:
    centroids = k_means(train, K, 0.001)
    distances = np.linalg.norm(test[:, np.newaxis, :] - centroids, axis=2)
    labels = np.argmin(distances, axis=1)
    contingency = contingency_matrix(num_tslabels_10, labels)
    row_ind, col_ind = linear_sum_assignment(-contingency)
    y_pred = np.zeros_like(labels)
    for i, j in zip(row_ind, col_ind):
        y_pred[labels == j] = i
    print("K:",K)
    print("precision:",precision(y_pred.tolist(), num_tslabels.tolist()))
    print("recall",recall(y_pred.tolist(), num_tslabels.tolist()))
    print("f1_score",f1(y_pred.tolist(), num_tslabels.tolist()))
    print("conditional_entropy",conditional_entropy(y_pred.tolist(), num_tslabels.tolist()))
    print()
```

Normalized Cut

In the normalized cut, we set the random seed by 42. Using the train_test_split function we produce a training dataset representing 0.15% of the dataset. Then we calculate the cosine similarity matrix, degree matrix, and the Laplacian matrix of the training dataset. After calculating the Laplacian matrix, we get the normalized sorted eigenvectors and perform K-Means on them and finally return the labels of the clustering.

```python
def ncut(training_data, k):
    """
    Splits the data into a training set and testing set with ratio 0.15% for training dataset,
    then applies the normalized cut algorithm on the reduced training dataset.

    Args:
        data (pd.DataFrame): pd.DataFrame containing the original dataset.
        k (int): number of clusters.

    Returns:
        predicted, true: nparrays of predicted labels after applying the normalized cut algorithm and the true labels.
    """
    training = tts(training_data, random_state=42, train_size=0.0015)[0]

    # Get the true labels
    true = training.iloc[:, 41].values

    # Convert the data into numpy arrays
    training = np.array(training)

    # Construct the similarity graph
    S = cosine_similarity(training)

    # Construct the degree matrix
    degrees = np.sum(S, axis=1)
    D = np.diag(degrees)

    # Compute Laplacian Matrix
    L = D - S

    # Compute sorted eigenvectors of the Laplacian Matrix then normalize them
    values, vectors = np.linalg.eigh(L)
    eigvectors = vecsort(vectors, values)

    normalized = norm(eigvectors[:, :k])

    # Perform K-means clustering on eigenvectors
    centroids = k_means(normalized, k, 0.01)
    distances = np.linalg.norm(normalized[:, np.newaxis, :] - centroids, axis=2)
    predicted = np.argmin(distances, axis=1)

    return predicted, true
```

Evaluation of Normalized Cut Results

- Precision = 0.9248672927725603
- Recall = 0.7068238647290678
- F1 Measure = 0.30294307548555777
- Conditional Entropy = 0.3441486877728897

```python
num_training['labels'] = num_trlabels
ncut_clustering = ncut(num_training, 11)
ncut_predicted = np.array(ncut_clustering[0])
ncut_true = np.array(ncut_clustering[1])
```

```python
print("Precision:", precision(ncut_predicted.tolist(), ncut_true.tolist()))
print("Recall", recall(ncut_predicted.tolist(), ncut_true.tolist()))
print("F1 Measure",f1(ncut_predicted.tolist(), ncut_true.tolist()))
print("Conditional Entropy",conditional_entropy(ncut_predicted.tolist(), ncut_true.tolist()))
```

```
Precision: 0.9248672927725603
Recall 0.7068238647290678
cluster: 0 pre: 0.8037735849056604   rec: 0.9545454545454546   f1: 0.8726953467954346
cluster: 1 pre: 0.9421221864951769   rec: 0.13739742086752638   f1: 0.23981993042766525
cluster: 2 pre: 1.0   rec: 0.11969380654140571   f1: 0.21379738968303294
cluster: 3 pre: 0.9552238805970149   rec: 0.08907446068197634   f1: 0.16295353278166774
cluster: 4 pre: 1.0   rec: 0.015309672929714684   f1: 0.03015764222069911
cluster: 5 pre: 0.6820083682008368   rec: 0.11343075852470424   f1: 0.19451073985680192
cluster: 6 pre: 0.9532163742690059   rec: 0.11343075852470424   f1: 0.20273631840796016
cluster: 7 pre: 0.9942528735632183   rec: 0.12038970076548365   f1: 0.21477343265052762
cluster: 8 pre: 0.9491525423728814   rec: 0.03897007654836465   f1: 0.07486631016042782
cluster: 9 pre: 0.9880952380952381   rec: 0.11551844119693806   f1: 0.20685358255451713
cluster: 10 pre: 0.9849906191369606   rec: 0.861664712778429   f1: 0.9192096048024013
F1 Measure 0.30294307548555777
Conditional Entropy 0.3441486877728897
```

DB Scan

The code defines a function called DB Scan that implements the density-based spatial clustering of applications with noise (DBSCAN) algorithm. The function takes three inputs: data, which is a pandas DataFrame containing the data to be clustered, eps, which is the maximum distance between two points for them to be considered as part of the same cluster, and min_samples, which is the minimum number of points required for a cluster to be formed. The function initializes some variables and then loops over each point in the data. For each point, it checks if it has already been assigned to a cluster. If not, it finds all the neighboring points within a distance of eps and determines whether the point is a "core point" (i.e., it has at least min_samples neighbors) or not. If the point is a core point, a new cluster is formed and all its connection points are assigned to the same cluster. If the point is not a core point, it is marked as an outlier. The function also calls another function called expand_cluster to expand the cluster starting from the current core point. The expand_cluster function takes the current point, its neighbors, and some other parameters and recursively adds new points to the cluster until no new points can be added. Finally, the function returns a list of cluster labels for each point in the data, where the label is an integer that indicates the cluster to which the point belongs. If a point is an outlier, its label is -1. We consider the outliers as one of the clusters.

```python
vis = []
my_dict = {}
def dbscan(data, eps, min_samples):
    """
    Performs DB Scan clustering algorithm on a given dataset.

    Args:
        data (nparray): the dataset to be clustered
        eps (float): a tolerance extent.
        min_samples (number): a number indicating the minimum number of samples.

    Returns:
        labels: the labels of the dataset after clustering.
    """
    X = data.values
    global vis
    global my_dict
    vis = [0] * len(X)
    my_dict = {i: [] for i in range(len(X))}
    labels = [0] * len(X)
    cluster_id = 0
    for i in range(len(X)):
        if labels[i] != 0:
            continue
        # Find all neighbors of the current point within eps distance
        neighbors = get_neighbors(X, i, eps)
        # If the point is not a core point, mark it as an outlier
        if len(neighbors) < min_samples:
            labels[i] = -1
            continue
        # Expand the cluster starting from the current core point
        cluster_id += 1
        labels[i] = cluster_id

        expand_cluster(X, labels, i, neighbors, eps, min_samples, cluster_id)

    return labels
```

```python
def expand_cluster(X, labels, i, neighbors, eps, min_samples, cluster_id):
    """
    Auxiliary function for the DB Scan to expand the core points clusters.

    Args:
        X (nparray): the dataset
        labels (nparray): labels of the clustering
        i (number): cluster's id
        neighbors (nparray): neighbors of the core point
        eps (float): a tolerance extent
        min_samples (number): the number of minimum samples
        cluster_id (number): the cluster's id
    """
    # Loop over each neighbor of the core point
    for j in neighbors:
        if labels[j] == -1:
            labels[j] = cluster_id
        elif labels[j] == 0:
            labels[j] = cluster_id
            # Find all neighbors of the current point within eps distance
            new_neighbors = get_neighbors(X, j, eps)
            # If the point is a core point, add its neighbors to the list of neighbors
            if len(new_neighbors) >= min_samples:
                neighbors += new_neighbors
```

```python
def get_neighbors(X, i, eps):
    """
    This functions gets the neighbour of the ith instance within given epsilon

    Args:
        X (nparray): the dataset
        i (number): cluster's id
        eps (float): a tolerance extent

    Returns:
        neighbors: nparray of neighbors of the ith instance
    """
    global vis
    global my_dict
    if vis[i] == 1:
        return my_dict[i]
    neighbors = []
    for j in range(len(X)):
        if i == j:
            continue

        dist = np.linalg.norm(X[i] - X[j])
        if dist <= eps:
            neighbors.append(j)
    vis[i] = 1
    my_dict[i] = neighbors
    return neighbors
```

Evaluation of DBSCAN Results

- Precision = 0.9897917517354023
- Recall = 0.7563745809565455
- F1 Measure = 0.23751813503832142
- Conditional Entropy = 0.08650543363578882

Evaluation

Clusterize

Auxiliary function to annotate the predicted labels with the true labels.

```python
def clusterize(pred_labels, true_labels):
    """
    Annotate the predicted with the true labels

    Args:
        pred_labels (list): list of predicted labels
        true_labels (list): list of true labels

    Raises:
        ValueError: The two lists must be equal.

    Returns:
        clusters, clusters_set: clusters is a dictionary of the annotation, and clusters_set is a set of the predicted labels.
    """
    if len(pred_labels) != len(true_labels):
        raise ValueError("The two list should be equal")
    clusters_set = set(pred_labels)
    # num_clusters = len(set(pred_labels))
    clusters = {}
    for cluster in clusters_set:
        clusters[cluster] = []
    for i in range(len(pred_labels)):
        clusters[pred_labels[i]].append(true_labels[i])
    return clusters, clusters_set
```

Precision

```python
def precision(pred_labels, true_labels):
    """
    Evaluates the precision of a clustering given the true labels.

    Args:
        pred_labels (list): list of predicted labels
        true_labels (list): list of true labels

    Returns:
        res: the precision value of the clustering
    """
    clusters, clusters_set = clusterize(pred_labels, true_labels)
    res = 0
    for cluster in clusters_set:
        most_common = max(set(clusters[cluster]), key = clusters[cluster].count)
        count = clusters[cluster].count(most_common)
        res += (len(clusters[cluster]) / len(true_labels)) * (count / len(clusters[cluster]))
    return res
```

Recall

```python
def recall(pred_labels, true_labels):
    """
    Evaluates the recall of a clustering given the true labels.

    Args:
        pred_labels (list): list of predicted labels
        true_labels (list): list of true labels

    Returns:
        res: the recall value of the clustering
    """
    clusters, clusters_set = clusterize(pred_labels, true_labels)
    res = 0
    r = len(clusters_set)
    for cluster in clusters_set:
        most_common = max(set(clusters[cluster]), key = clusters[cluster].count)
        count = clusters[cluster].count(most_common)
        count_total = true_labels.count(most_common)
        res += (len(clusters[cluster]) / len(true_labels)) * (count / count_total)
    return res
```

F1 Measure

```python
def f1(pred_labels, true_labels):
    """
    Evaluates the F1-Measure of a clustering given the true labels.

    Args:
        pred_labels (list): list of predicted labels
        true_labels (list): list of true labels

    Returns:
        res: the F1-Measure value of the clustering
    """
    clusters, clusters_set = clusterize(pred_labels, true_labels)
    res = 0
    r = len(clusters_set)
    for cluster in clusters_set:
        most_common = max(set(clusters[cluster]), key = clusters[cluster].count)
        count = clusters[cluster].count(most_common)
        count_total = true_labels.count(most_common)
        precision =  count / len(clusters[cluster])
        recall = count / count_total
        f1 = (2 * precision * recall) / (precision + recall)
        print(f"cluster: {cluster} pre: {precision}   rec: {recall}   f1: {f1}")
        res += (float(f1) / float(r))
    return res
```

Conditional Entropy

```python
from math import log2
def conditional_entropy(pred_labels, true_labels):
    """
    Evaluates the conditional entropy of a clustering given the true labels.

    Args:
        pred_labels (list): list of predicted labels
        true_labels (list): list of true labels

    Returns:
        res: the conditional entropy value of the clustering
    """
    clusters, clusters_set = clusterize(pred_labels, true_labels)
    res = 0
    true_labels_set = set(true_labels)
    for cluster in clusters_set:
        temp = 0
        for t in true_labels_set:
            t_count = clusters[cluster].count(t)
            if t_count != 0:
                temp += -(t_count / len(clusters[cluster])) * log2(t_count / len(clusters[cluster]))
        res += (len(clusters[cluster]) / len(true_labels)) * temp
    return res
```