DDA3020 Machine Learning Assignment 4

Ma Xinxian 121090408@link.cuhk.edu.cn

June 13, 2023

Contents

1 TASK DESCRIPTION

Clustering on UCI seed dataset with **PCA** and **K-means**. To contact me, send to my email on the cover page. Welcome any suggestions and requirements!

2 PCA

2.1 WHAT AND WHY

PCA aims to find a low-dimensional data vector to represent the original high-dimensional data vector so that it may help visualization, alleviate overfitting or reduce the computational cost.

2.2 PROCEDURE

Step1: Calculate the empirical covariance matrix [*numpy*]

$$\Sigma = \frac{1}{N}\sum_{n=1}^{N}(x^{(n)} - \mu)(x^{(n)} - \mu)^T$$

Step2: Do SVD decomposition of Σ to obtain its D eigenvalues λ_i and eigenvectors q_i by using *np.linalg.eig(cov)* function in numpy, and rank them from large to small according to the eigenvalues.

Step3: Pick the top-K eigenvectors to form the matrix $U = [q_1, ..., q_K]$ by *argmin* function.

Step4: The new representation of $x^{(n)}$ is $U^T(x^{(n)})$−μ).

2.3 CODE WITH PYTHON

```
X = np.subtract(X, np.mean(X, axis=0))
# Step1
COV = np.dot(X.T, X)/X.shape[0]
# Step2
eigenvalue, eigenvector = np.linalg.eig(COV)
print(eigenvalue)
```

```
[1.07419301e+01 2.11931485e+00 7.32794138e-02 1.28261257e-02
 2.73513989e-03 1.56297146e-03 2.95142261e-05]
```

```
#Step3 & 4
# from above, the first and the second are top two
z1 = np.dot(X, eigenvector.T[0])
```

```
z2 = np.dot(X, eigenvector.T[1])
new_X = np.array([z1,z2]).T
print(new_X)
```

3 K-MEANS

3.1 WHAT AND WHY

K-means clustering is a method of vector quantization, that aims to partition n observations/samples into k clusters in which each observation belongs to the cluster with the nearest mean.

3.2 PROCEDURE WITH PYTHON

Step1: First, choose K — the number of clusters. Then you randomly put K feature vectors, called centroids, to the feature space. Here, 3.

Step2: Next, compute the distance from each example x to each centroid c using the Euclidean distance. In the code, this is implemented by *cal_dis()*

```
def cal_dis(dataSet, centroids, k):
    # output's distance corresponding to the index of the center
    # sort_distance_index ouputs a sorted center's index list

    distance_list = []
    sort_distance_index = []

    for i in range(dataSet.shape[0]):
        ith_dis = []
        for center in centroids:
            ith_dis.append(np.linalg.norm(dataSet[i] - center))

        distance_list.append(ith_dis)
        sort_distance_index.append(sorted(range(len(ith_dis)), key=
            lambda k: ith_dis[k], reverse=False))

    return distance_list, sort_distance_index
```

Then we **assign** the closest centroid to each example.

Step3: For each centroid, we calculate the average feature vector of the examples labeled

with it. These average feature vectors become the new locations of the centroids. In the code, this is implemented by *classify()*. New centroids are *newCentroids*.

```
def classify(dataSet, centroids, k):

    distance_list, sort_distance_index = cal_dis(dataSet, centroids
        , k)

    minDistIndices = np.argmin(distance_list, axis=1)
    newCentroids = pd.DataFrame(dataSet).groupby(minDistIndices).
        mean()

    newCentroids = newCentroids.values
    changed = newCentroids - centroids

    return changed, newCentroids
```

Step4: We recompute the distance from each example to each centroid, modify the assignment and repeat the procedure until the assignments don't change after the centroid locations are recomputed. This is implemented by the while loop in the *k-means* function.

```
def kmeans(dataSet, k):
    cluster = [[],[],[]]
    centroids = random.sample(list(dataSet), k)

    # initial assignment
    changed, newCentroids = classify(dataSet, centroids, k)

    while np.any(changed != 0):
        changed, newCentroids = classify(dataSet, newCentroids, k)

    centroids = newCentroids.tolist()
    distance_list, sort_distance_index = cal_dis(dataSet, centroids
        , k)
    minDistIndices = np.argmin(distance_list, axis=1)

    for i, j in enumerate(minDistIndices):
        cluster[j].append(dataSet[i].tolist())

    return centroids, cluster
```

Step5: Finally, we output the centroids and the cluster (too long, see in the code file). And we also plot the classification.

```
centroids, cluster = kmeans(new_X, 3)
print('Centroids{}'.format(centroids))

for i in range(3):
    print('Cluster{}:{}'.format(i, cluster[i]))

color_list = ['green', 'blue', 'purple']
for i in range(3):
    clus = cluster[i]
    color = color_list[i]
    for point in clus:
        plt.scatter(point[0],point[1], marker = 'o',color = color,
           s = 40, label = 'orginal point')

for i in range(len(centroids)):
    plt.scatter(centroids[i][0],centroids[i][1],marker='x',color='
       red', s=50,label='centroids')

plt.show()
```

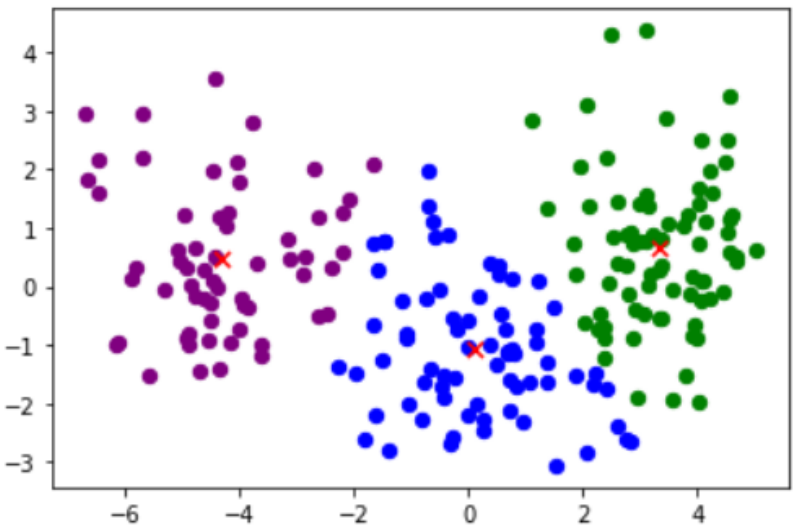

FIGURE 1: CLASSIFICATION.

4 PERFORMANCE EVALUATION

4.1 SILHOUETTE COEFFICIENT

Given a clustering, we define

- **a**: The mean distance between a point and all other points in the same cluster.

- **b**: The mean distance between a point and all other points in the next nearest cluster.

Then, Silhouette coefficients for a single sample is formulated as:

$$s = \frac{b - a}{max(a, b)}$$

And Silhouette coefficient s for a set of samples is defined as the mean of the Silhouette Coefficient for each sample. Thus, we implement it as following:

```
# Silhouette Coefficient
distance_list, sort_distance_index = cal_dis(new_X, centroids, k)
s_sum = 0
for i in range(new_X.shape[0]):
    s_data = new_X[i]
    smallest_index = sort_distance_index[i][0]
    second_index = sort_distance_index[i][1]
    a_s_sum = 0
    b_s_sum = 0
    #a
    for oth_point in cluster[smallest_index]:
        a_s_sum += np.linalg.norm(s_data - oth_point)
    a_s = a_s_sum/len(cluster[smallest_index])
    #b
    for oth_point in cluster[second_index]:
        b_s_sum += np.linalg.norm(s_data - oth_point)
    b_s = b_s_sum/len(cluster[second_index])

    s_s = (b_s-a_s)/max(a_s, b_s)
    s_sum += s_s

s = s_sum/new_X.shape[0]
print(s)
```

```
0.4875992369330414
```

It is easy to know that $s \in (-1, 1)$, and larger s value indicates better clustering performance. So this classification is relatively good.

4.2 RAND INDEX

Given a set of n samples $S = o_1, o_2, ..., o_n$, there are **two clusterings of S to compare**, including: X with r clusters and Y with s clusters. Let us define:

- ***a***: The number of pairs of elements in S that are in the same subset in X and Y
- ***b***: The number of pairs of elements in S that are in the different subset in X and Y
- ***c***: The number of pairs of elements in S that are in the same subset in X but different in Y
- ***d***: The number of pairs of elements in S that are in the different subset in X but same in Y The rand index (RI) can be computed as follows:

$$RI = \frac{a+b}{a+b+c+d} = \frac{a+b}{\frac{(1+n)n}{2}}$$

Note that $RI \in [0, 1]$, and **higher score corresponds higher similarity**. In this task, we do not have two clustering, but the true label and PCA & K-means clustering. If we do rand index to compare them, we can evaluate the performance of our clustering (similarity with true label). Thus, we implement the rand index as followings:

```
a = 0
b = 0
label = txtDF[txtDF.columns[-1]]
for i in range(new_X.shape[0]):
    for j in range(i+1, new_X.shape[0]):
        if sort_distance_index[i][0] == sort_distance_index[j][0]
           and label[i] == label[j]:
            a +=1
        elif sort_distance_index[i][0] != sort_distance_index[j][0]
            and label[i] != label[j]:
            b +=1

RI = 2* (a + b)/ (new_X.shape[0] * (new_X.shape[0] - 1))
print(RI)
```

```
0.8743677375256322
```

Which has a high score, so the classification is good.

<END>