University of Exeter

College of Engineering, Mathematics and Physical Sciences

ECM3420 - Learning From Data

Coursework 2 - Clutering

Enter your candidate number here: 054573

Task 1

In [1]:
```
import numpy as np
import pandas as pd
from scipy.spatial import distance
from sklearn import datasets, model_selection, preprocessing
from sklearn.decomposition import PCA
from sklearn.cluster import KMeans

from math import sqrt
import time
import matplotlib.pyplot as plt
```

In [2]:
```
class DataPoint:
    def __init__(self, values, centroids):
        """
        Initialize a Point object with a value and centroid.
        :param values: values/coordinates of a datapoint
        :param centroids: centroids in the dataset
        """
        # the data point values/coordinates
        self.values = values

        # the centroid the cluster is associated to
        self.centroid = None

        # upon intialization the nearest centroid is selected to start
        self.assign_to_cluster(centroids)


    def euclidian_distance(self,point_a,point_b):
        """
        Calculate distance between two points using the euclidian distance formula
        :param point_a,point_b: coordinates/values of a datapoint
        """
        # euclidian distance formula
        return sqrt(sum(np.square(point_b - point_a)))


    def assign_to_cluster(self, centroids):
        """
        Update the Centroid/Cluster this datapoint is assigned to.
        Closest Centroid (Euclidian Distance) is selected.

        :param centroids: centroids in the dataset
        :return : False or True
        """
        closest_centroid = None
        closest_distance = 0
        change= False

        # go through all centroids to find the one that is closest to this point
        for centroid in centroids:
            centroid_distance = self.euclidian_distance(self.values, centroid.values)

            if closest_centroid is None or centroid_distance  < closest_distance:
```

```
                closest_centroid = centroid
                closest_distance = centroid_distance


        # assign new cluster and log whether point has changed cluster
        change = self.centroid != closest_centroid
        self.centroid = closest_centroid

        # return whether a change occured
        return change
```

In [3]:

```
class Centroid:
    def __init__(self, label, values):
        """
        Initialize a Centroid object with a value/coordinates and label.
        :param label: randomly allocated label (int)
        :param centroids: centroids in the dataset
        """
        # the label for a cluster
        self.label = label
        # associated array depicting the coordinate of the centroid
        self.values = values


    def update_centroid(self, datapoints):
        """
        Find the average coordinate of all of the points in the cluster and set it to be the centroids
        coordinate

        :param datapoints: A list of all of the datapoint objects
        """
        # create an array of all of the points in this centroid's corresponding cluster
        cluster = [point.values for point in datapoints if point.centroid is self]

        # in the first run there is a chance that all centroids are not set to a particular cluster
        if len(cluster) == 0:
            return

        # take the average point of the cluster, this is the new centroid value/coordinate
        new_centroid = np.mean(cluster, axis=0)

        # assign new coordinate
        self.values = new_centroid
```

In [4]:

```
def random_centroids(data, k, random_state):
    """
    Pick k random centroids using a random state from a collection of datapoints
    :param data: List of datapoint values
    :param k: Number of centroids to choose
    :param random_state: int representing random state
    """
    # initialize random generator
    rng = np.random.default_rng(random_state)

    # choose random point
    return rng.choice(data, k)
```

In [5]:

```
def update_centroids(data, centroids):
    """
    Iterate through all clusters and update their centroid values
    :param data: List of Datapoint objects
    :param centroids: A list of Centroid objects
    """
    for centroid in centroids:
        centroid.update_centroid(data)
```

In [6]:
```
def incremental_kmeans(data, k, max_itr=100, random_state=None):
    """
    An implementation of the incremenral kmeans algorithm

    :param x: the data to be clustered (data points)
    :param k: the number of clusters
    :param max_itr: the max number of iterations
    :param random_state: determines the random number generation for centroid initialization
    :return cluster_labels: the cluster membership labels for each element in the data x
    :return n_iter: number of iterations run
    """

    # choose k random centroids out of the data
    r_centroids = random_centroids(data, k, random_state)

    # initialize centroid objects and assign them labels and coordinates
    centroids = [Centroid(i, r_centroids[i]) for i in range(len(r_centroids))]

    # initialize the datapoints of the set and assign each to closest centroid/cluster
    datapoints = [DataPoint(point,centroids) for point in data]

    # reassign centroids to these new clusters
    update_centroids(datapoints, centroids)

    itr = 0
    change_occured = True #Set this flag to true for the first iteration

    # while stopping conditions unfullfilled
    while itr < max_itr and change_occured == True:

        # increment iteration
        itr += 1
        change_occured = False #set flag to false

        for point in datapoints:
            # assign the point to closest centroid and log whether there has been a change
            change = point.assign_to_cluster(centroids)

            if change:
                change_occured = True
                # there is no need to update centroids if no points have moved in the previous step
                update_centroids(datapoints, centroids)

    labels = np.array([point.centroid.label for point in datapoints])

    return labels, itr
```

Task 2

In order to analyze the runtime of the two KMeans variations here I:

- Import the dataset
- Split the datatset into variables and labels
- Split these into a training and testing set
- Normalize the two training sets (best ppractice for clustering)

In [7]:
```
# import the datatset
iris = datasets.load_iris()

# split the dataset into data and target variables
iris_X = iris.data
iris_y = iris.target

# split the dataset into a training and testing set
X_train, X_test, y_train, y_test = model_selection.train_test_split(iris_X, iris_y, test_size = 0.3, ra

# Load the Normalizer
n = preprocessing.Normalizer().fit(X_train)

# normalize the data
```

```
X_train_nz = n.transform(X_train)
X_test_nz = n.transform(X_test)

# swap dataset for scaled dataset
X_train=X_train_nz
X_test=X_test_nz
```

Task 2.1 Draw a Table of Average Metrics on the Dataset

Here I define functions for measuring runtime metrics of both algorithms and print them out in a table

In [8]:

```
def std_metrics(data, m, k):
    """
    :param data: the data to be clustered
    :param m: the number of test runs
    :param k: the number of clusters
    """

    # initialize variables to be logged
    no_iterations = 0
    time_elapsed = 0

    for i in range(m):
        # begin measuring time
        t0 = time.time()

        # run algorithm
        kmeans = KMeans(n_clusters=k, random_state=i)
        kmeans.fit(X_train)

        # calculate time elapsed and add to the sum of time over all runs
        t1 = time.time() - t0
        time_elapsed += t1 * 1000

        # add number of iterations for each run to sum of all runs
        no_iterations += kmeans.n_iter_

    # calculate average over m runs
    avg_iter = no_iterations/m
    avg_time = time_elapsed/m

    return round(avg_iter,4), avg_time

def inc_metrics(data, m, k):
    """
    :param data: the data to be clustered
    :param m: the number of test runs
    :param k: the number of clusters used in the run
    """

    # initialize variables to be logged
    no_iterations = 0
    time_elapsed = 0

    for i in range(m):
        #begin measuring time
        t0 = time.time()

        #run algorithm
        y_means, n_iter = incremental_kmeans(data, k, max_itr=100, random_state=i)


        # calculate time elapsed and add to the sum of time over all runs
        t1 = time.time() - t0
        time_elapsed += t1 *1000

        # add number of iterations for each run to sum of all runs
        no_iterations += n_iter

    # calculate average over m runs
    avg_iter = no_iterations/m
```

```
    avg_time = time_elapsed/m

    return round(avg_iter,4), avg_time
```

In [9]:

```
def compare_implementations(m, k,data):
    """
    :param m: the number of test runs
    :param k: a list depicting how many clusters are to be used in experiments for each run
    :param data: data to compare the implmentations on
    """
    std_kmeans = []
    inc_kmeans = []

    # for each cluster in test
    for cluster_n in k:

        # calculate average time and iterations over m runs and append to tracker list for std
        std_iter, std_time = std_metrics(data, m, cluster_n)
        std_kmeans.append([cluster_n,std_iter,std_time])

        # calculate average time and iterations over m runs and append to tracker list for inc
        inc_iter, inc_time = inc_metrics(data, m, cluster_n)
        inc_kmeans.append([cluster_n,inc_iter,inc_time])

    # plot the data for Standard KMeans
    table = pd.DataFrame(std_kmeans, columns =['Clusters', 'Average Iterations', 'Average Time(ms)'])
    table = table.set_index('Clusters')
    table = table.transpose()
    print("Standard K-Means")
    print("-----------------------------------------------------------")
    print(table, "\n")

    # plot the data for Incremental KMeans
    table = pd.DataFrame(inc_kmeans, columns =['Clusters', 'Average Iterations', 'Average Time(ms)'])
    table = table.set_index('Clusters')
    table = table.transpose()
    print("Incremental K-Means")
    print("-----------------------------------------------------------")
    print(table)
```

In [10]:

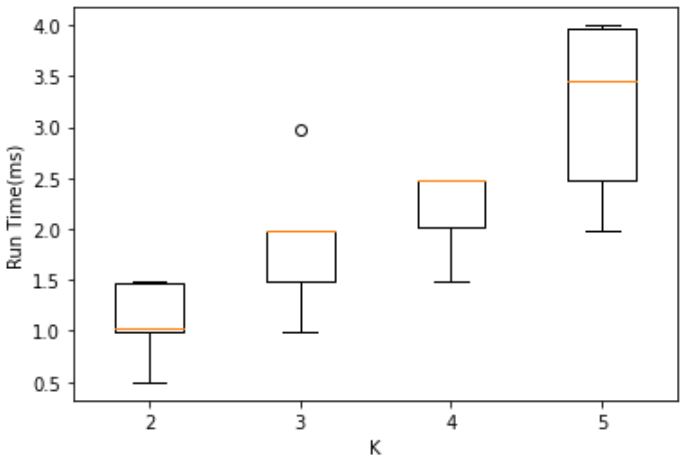
```
# run the comparison
compare_implementations(5,[2,3,4,5], X_test)
```

```
Standard K-Means
-----------------------------------------------------------
Clusters                  2         3          4          5
Average Iterations  2.000000   3.80000   6.200000   4.800000
Average Time(ms)    8.823442  12.39996  16.169739  19.641638

Incremental K-Means
-----------------------------------------------------------
Clusters                  2        3         4         5
Average Iterations  1.600000  2.60000  2.400000  3.000000
Average Time(ms)    0.992012  1.79038  2.073574  3.273392
```

In [11]:

```
def plot_inc_kmeans(m,k,data):
    """
    Run the incremental kmeans algorithm and gather runtime metrics to plot on a graph.
    :param m: the number of test runs
    :param k: a list depicting how many clusters are to be used in experiments for each run
    :param data: data to compare the implmentations on
    """

    results = []

    for cluster_n in k:
        cluster_results = []
        for rand_state in range(m):

            # measure the time of each algorithm iteration
```

```
            t0 = time.time()
            y_means, _ = incremental_kmeans(data, cluster_n, max_itr=100, random_state=rand_state)
            t = time.time() - t0

            # append to list for this cluster
            cluster_results.append(t*1000)

        # append to results to plot latter
        results.append(cluster_results)

    # plot the results
    fig1,ax1 = plt.subplots()
    ax1.boxplot(results, labels=k)

    plt.xlabel('K')
    plt.ylabel('Run Time(ms)')

    plt.show()

# run the algorithm and plot as specified in coursewoek
plot_inc_kmeans(5,[2,3,4,5], X_test)
```



Task 3

In [12]:

```
def jaccard_index_cw2(y_true,y_pred):
    """
    An impleementation of the jaccard index as defined in class.
    Using the formula a/(a+b+c), we firstly identify the sets a, and bc.
    :param y_true: A list of ground truth labels
    :param y_true: A list of predicted labels
    :return : Jaccard Score of the two sets (here, it is used to provide the labels)
    """
    a=0
    bc=0

    for i in range(len(y_true)):
        for j in range(i+1, len(y_true)):
            # SS (a)
            if y_true[i] == y_true[j] and y_pred[i] == y_pred[j]:
                a += 1

            #(not d)
            elif not(y_true[i] != y_true[j] and y_pred[i] != y_pred[j]):
                bc+= 1

        return a/(a+bc)


def jaccard_index_alternative(y_true,y_pred):
    """
    The jaccard score, as defined by the internet as the number of overlapping points in 2 sets,
    divided by the union of the two sets. This one is implemented just for fun.
    """
    correct = 0
```

```
    for i in range(len(y_true)):
        if y_true[i] == y_pred[i]:
            correct += 1

    return correct/len(y_true)
```

Task 4

In [13]:

```
from sklearn.metrics import jaccard_score
from sklearn.metrics.cluster import contingency_matrix
from sklearn.metrics import plot_confusion_matrix
from sklearn.metrics import jaccard_score
from tabulate import tabulate
from IPython.core.display import HTML, display
```

In [14]:

```
# the labels of the clusters given by each algorithm are different so we need to make sure that the lab
# correctly in order to get a reasonable result with the sk learn version of the jaccard score, which u
# principle as the internet defined version of the jaccard index

# align the results of ground truth and incremental kmeans
y_pred, _ = incremental_kmeans(X_test, 3, max_itr=100, random_state=0)

# return the intersections of the predicted labels and true solution
align = np.argmax(contingency_matrix(y_pred,y_test), axis=1)

# use the values from the contingency matrix to create a dictionary of correct mappings for the cluster
mapping = {i:align[i] for i in range(len(align))}

# use the mapping to convert the prediction clusters to correct labels
y_pred = np.array([mapping.get(n,n) for n in y_pred])

# align the results of ground truth and standard kmeans
kmeans = KMeans(n_clusters=3,max_iter=100, random_state=0)
y_means = kmeans.fit_predict(X_test)
align = np.argmax(contingency_matrix(y_means,y_test), axis=1)

# use the values from the contingency matrix to create a dictionary of correct mappings for the cluster
mapping = {i:align[i] for i in range(len(align))}

# use the mapping to convert the prediction clusters to correct labels
y_means = np.array([mapping.get(n,n) for n in y_means])
```

In [15]:

```
def generate_label_table(y_true, y_pred, cluster_n, method):
    """
    Generates a table depicting the counts of each label in each cluster, from the list of ground truth
    and the list of predicted labels.

    :param y_true: A 1-D Array of Labels
    :param y_pred: A 1-D Array of Labels
    :param cluster_n: Number of Labels/Clusters
    :param method: String method,
    """
    table = [[method]]
    # create table of label listings
    labels = np.zeros((3,3))

    for truth,cluster in zip(y_true, y_pred):
        labels[cluster][truth] += 1

    table[0].extend(["Label " + str(i + 1) for i in range(cluster_n)])

    for i in range(cluster_n):
        table.append(["Cluster " + str(i+1)] + list(labels[i]))

    return table

display(HTML(tabulate(generate_label_table(y_test, y_pred, 3, "Incremental KMeans"), tablefmt="html")))
```

```
display(HTML(tabulate(generate_label_table(y_test, y_means, 3, "Standard KMeans"), tablefmt="html")))

print("Jaccard Score (Implemented):")
print("Incrmental Kmeans:",jaccard_index_cw2(y_test,y_pred))
print("Standard Kmeans:",jaccard_index_cw2(y_test,y_means))

print("Jaccard Score (SKLearn)")
print("Incrmental Kmeans:", jaccard_score(y_test,y_pred,average='micro'))
print("Standard Kmeans:",jaccard_score(y_test,y_means, average='micro'))
```

Incremental KMeans	Label 1	Label 2	Label 3
Cluster 1	18.0	0.0	0.0
Cluster 2	0.0	10.0	2.0
Cluster 3	0.0	0.0	15.0

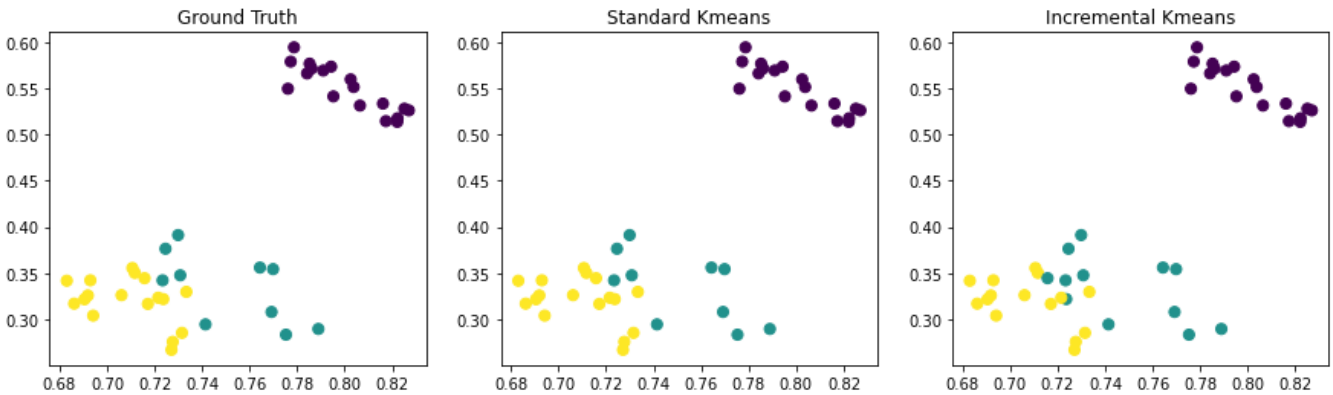
Standard KMeans	Label 1	Label 2	Label 3
Cluster 1	18.0	0.0	0.0
Cluster 2	0.0	10.0	0.0
Cluster 3	0.0	0.0	17.0

```
Jaccard Score (Implemented):
Incrmental Kmeans: 0.8181818181818182
Standard Kmeans: 1.0
Jaccard Score (SKLearn)
Incrmental Kmeans: 0.9148936170212766
Standard Kmeans: 1.0
```

In [16]:

```
fig, (ax1, ax2, ax3) = plt.subplots(1, 3,figsize=(15, 4))
print("Kmeans CLustering Reuslts:")
ax1.set_title("Ground Truth")
ax2.set_title("Standard Kmeans")
ax3.set_title("Incremental Kmeans")
ax1.scatter(X_test[:, 0], X_test[:, 1], c=y_test, s=50)
ax2.scatter(X_test[:, 0], X_test[:, 1], c=y_means, s=50)
ax3.scatter(X_test[:, 0], X_test[:, 1], c=y_pred, s=50)
plt.show()
```

```
Kmeans CLustering Reuslts:
```



Writeup:

Jaccard Score Parameter Choice

The Sklean implmentation of the jaccard index is slightly different than the one outlined in class, and requires an argument *average* to be specified in order for the function to run correclty. The options here were, binary (which was not applicable as the labels are themselves not binary), weighted (which was overkill in this scenario as we knew we would be performing jaccard comparisons on sets with no label-imbalance), samples (which was not appropriate seeing as we were not dealing in multilabel classification), macro (more appropriate than the above and useful) and micro (the one that was most similar to the jaccard score discussed in class and therefore used).

Conclusions as seen from Data and tables

It seems, looking at results of task two that the method incremental Kmeans uses is indeed more efficient than Standard Kmeans. This can be seen to be caused by the consistently lower amount of iterations needed to

approximate clusters, which also then can have the side effect of the algorithm performing significantly faster in terms of runtime as well.

Moreover, as the incremental kmeans algorithm can afford to skip unecessary reassignment of centroids in the instance of a datapoint not changing cluster assignment, we can also reduce the amount of computation, and with that, runtime, on each of the passes of the algorithm.

This can further be confirmed not to be a simple fluke in Task 4, both in regards to the number of labels in each cluster, showing similar clusters in both std kmeans and incremental kmeans. Similarily the jaccard score also points towards the incremental kmeans having excellent accuracy on this particular dataset. On this particular random state (0), there were only 2 instances of mislabelled samples.

As well as that it is clear that both versions of the algorithm correctly form clusters, and those clusters are similar/identical to the ground truth.

With all of this information in mind, it is reasonable to conclude that the incremental KMeans Algorithm outperforms the Std Kmeans version in terms of iteration and runtime efficiency. However, it would be advisable to perform tests similar to the above on more tricky datasets, larger datasets, or using a higher spread of random states to confirm this conclusion still holds. However, on this particular dataset and test set, Incremental Kmeans seems to be a much better choice.