DMO - Programming project - Clustering methods

November 30, 2019

1 Deterministic Models and Optimization

2 Programming project - Clustering methods

Alexandros Pappas Aron Pap Pedro Freitas

2.1 Short summary

In this programming project we implemented MST and k-means clustering for a 2-dimensional and a 5-dimensional dataset and we have seen that k-means seems to perform better for the 2-dimensional case with 15 clusters, while MST does a better job for the 5-dimensional case with 3 clusters (however for that dataset 2,3, 8 and 11 are all seemed to be potentially good number of clusters based on our experiments).

2.2 Implementation

In the following cell, we will import some standard libraries from which we will use some standard procedures, functions.

[4]:
```
# Import some modules which we will use later

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import math
import copy
from python_algorithms.basic.union_find import UF

%matplotlib inline
```

2.3 Functions

In the following cell we define the functions we wrote to implement the algorithms.

[1]:
```
### Define custom function

## Basic functions for all algorithms

def euclidean_dist(x,y):
    """Calculates Euclidean distance between 2 n-dimensional vectors"""
    temp = x-y
    temp_2 = temp**2
    temp_sum = sum(temp_2)
    temp_sqrt = math.sqrt(temp_sum)

    return temp_sqrt

def benchmark_data_generation(n,p,K,std):
    """Generates benchmak data for clustering"""

    from sklearn.datasets.samples_generator import make_blobs
    X, y = make_blobs(n_samples=n, centers=K, n_features = p,
                  random_state=0, cluster_std=std)

    return X, y

## Functions for the MST algorithm

def adjacency_matrix(X):
    """Calculates distance matrix of input data points"""
    n = X.shape[0]
    X = np.array(X)
    dist_matrix = np.zeros((n, n))
    for i in range(0,n):
        for j in range(i+1,n):
            dist_matrix[i,j]= np.linalg.norm(X[i]-X[j])

    return dist_matrix

def adjacency_matrix1(X):
    """Calculates distance matrix of input data points"""
    n = X.shape[0]
    dist_matrix = np.zeros((n, n))
    for i in range(0,n):
        for j in range(i+1,n):
            dist_matrix[i,j]=euclidean_dist(np.array(X.iloc[i,]),np.array(X.
 ↪iloc[j,]))

    return dist_matrix

def edge_vector(dist_matrix):
```

```
    """Creates list of edges/vertices with the corresponding distance"""
    n=len(dist_matrix)
    n_rows = int(n*(n-1)*(0.5))
    edge_dist = np.zeros((n_rows, 3))

    counter = 0

    for i in range(0,n):
        for j in range(i+1,n):

            edge_dist[counter,0] = i
            edge_dist[counter,1] = j
            edge_dist[counter,2] = dist_matrix[i,j]
            counter += 1

    return edge_dist

def edge_sorting(edge_vector):
    """Sorts the list of edges/distances"""
    sorted_matrix = edge_vector[edge_vector[:, -1].argsort(kind='mergesort')]

    return sorted_matrix


## Functions for MST/Kruskal algorithm
def mst_cluster_UF(X, k):
    """Kruskal algorithm for MST"""
    X = pd.DataFrame(X)

    # intiliaze empty tree
    tree = UF(len(X))

    # calculate distances between two points and store in a vector
    matrix = adjacency_matrix(X)
    vector = edge_vector(matrix)

    # sort distances from smaller to bigger
    sorted_vector = edge_sorting(vector)

    # start finds and unions
    i = 0
    while tree.count() > k:

        # get nodes following the order of the sorted vector by distances
        node1 = sorted_vector[i][0].astype('int')
        node2 = sorted_vector[i][1].astype('int')
```

```
        # find the tree for each of the nodes
        tree1 = tree.find(node1)
        tree2 = tree.find(node2)
        i = i + 1

        # check if they are already connected: if not, join trees
        if tree1 != tree2:
            tree.union(node1, node2)

    # arrange results in the format we want for runnning the indices later
    temp =  np.ones((len(X),2), dtype=int)
    for point in range(len(X)):
        temp[point][0] = int(point)
        temp[point][1] = int(tree.find(point))
    return temp


def mst_cluster(X, k, d):
    X = pd.DataFrame(X)

    matrix = adjacency_matrix(X)
    vector = edge_vector(matrix)
    sorted_vector = edge_sorting(vector)



    i = 0
    #initialize 1 cluster for each point
    points_set = X.join(pd.DataFrame(np.arange(0,len(X)), columns=['Cluster']))

    #set stopping point
    while points_set['Cluster'].nunique() > k:

        #start from firs row or sorted vector, get nodes
        node1 = sorted_vector[i][0].astype('int')
        node2 = sorted_vector[i][1].astype('int')

        #identify to which cluster each point belongs
        cluster1 = points_set.iloc[node1,d]
        cluster2 = points_set.iloc[node2,d]

        #if they are from the same cluster already, no connection can be made␣
↪to avoid cycle
        if cluster1 != cluster2:
            # if they are different, the smaller cluster label will be given to␣
↪the larger
            if cluster1 < cluster2:
```

```
                points_set.iloc[node2,d] = cluster1
            else:

                points_set.iloc[node1,d] = cluster2
        i = i +1


    col_list = list(np.arange(0,d,1))

    points_set.drop(columns=col_list)
    points_set['point_index'] = points_set.index
    points_set = points_set[['point_index', 'Cluster']]
    points_set = np.array(points_set)
    return points_set


def centroid_calc_for_mst(K,d,result,X):
    """Calculates the centroids of the clusters
    found by the MST method"""

    # Get current unique labels
    uniques = pd.DataFrame(result).iloc[:,1]
    uniques = uniques.unique()

    # Loop over the clusters table and assign cluster names
    # going from 0-15 (basically just renaming clusters for convenience)
    for i in range(0,len(result)):
        # Getting the current cluster label
        index = int(result[i,1])
        # Searching the current cluster label in the list of unique labels
        unique_places = np.array([i==index for i in uniques])

        # Assign the relative location of the current label to the range of 0-15
        temp = np.where(unique_places)[0]
        result[i,1] = temp

    # Initialize the centroids to be 0
    centroids2 = np.zeros((K, d))

    for i in range(0,K):
                # Filter for points in the given cluster
                temp_table2 = result[result[:,1]==i]

                # Calculating new centroids
                temp_array = np.zeros((len(temp_table2),d))
                counter = 0
```

```
                for j in temp_table2[:,0]:

                    temp_array[counter,:] = X[j,:]
                    counter += 1

                # Assign the new centroids - coordinate-wise mean of points in␣
 ↪the cluster
                centroids2[i,:] = np.mean(temp_array, axis=0)

    return centroids2

## Functions for the k-means algorithms

def k_means_clustering(K,X):
    n = X.shape[0] # Number of observations
    d = X.shape[1] # dimensions
    X = np.array(X) # Convert pandas dataframe to numpy array

    # STEP 0
    # Get initial starting points from the data points
    initial_indices = np.random.randint(0,high=n,size=K) # Initial indices
    print(initial_indices)

    # Initialize centroids
    centroids = np.zeros((K, d))

    counter = 0

    # Get initial centroids as the randomly chosen datapoints
    for i in initial_indices:
        centroids[counter,:] = X[i,:]
        counter += 1


    # STEP 1
    # Initialize table for holding the datapoint indexes and
    # their corresponding clusters
    cluster_table = np.zeros((n, 2))
    cluster_table[:,0] = np.arange(0,n,1)
    cluster_table = cluster_table.astype(int)


    change_tracker = True
```

```
    cost_values_db = [] # Initialize list for DB-index values during iterations
    cost_values_dunn = [] # Initialize list for Dunn-index values during␣
↪iterations

    while change_tracker: #

        # Initialize counter for the changes in cluster assignments
        overall_change = 0

        # Loop over data points
        for i in range(0,n):
            current_cluster = int(cluster_table[i,1])


            # Calculate distance for a point from its current cluster centroid
            current_dist = np.linalg.norm(X[i,]-centroids[current_cluster,])

            # Loop over clusters
            for j in range(0,K):


                # Calculate distance for a point and all potential cluster␣
↪centroids
                temp = np.linalg.norm(X[i,]-centroids[j,])

                # Assign a point to the closest cluster
                if temp<current_dist:
                    cluster_table[i,1]=j
                    current_dist = temp

                    # If a point changes cluster in the iteration, increment␣
↪the counter
                    overall_change += 1

        # If no point changes cluster in a given iteration, the stopping␣
↪criteria is met
        if overall_change == 0:
            change_tracker = False

        # STEP 2
        # Recalculate centroids

        # Loop over clusters
        for i in range(0,K):

            # Filter for points in the given cluster
            temp_table2 = cluster_table[cluster_table[:,1]==i]
```

```
            # Initialize temporary table for points in the given cluster
            temp_array = np.zeros((len(temp_table2),d))
            counter = 0

            # Put the values of points into the temporary table
            for j in temp_table2[:,0]:
                temp_array[counter,:] = X[j,:]
                counter += 1

            # Calculate cluster centroids as average of points in the cluster
            if len(temp_array)>0:
                centroids[i,:] = np.mean(temp_array, axis=0)

        # Append index values from the current iteration to the list
        cost_values_db.append(DB_index(X,centroids,cluster_table))
        cost_values_dunn.append(dunn_index(X,centroids,cluster_table))

    return centroids,cluster_table,cost_values_db,cost_values_dunn

## Functions for the Davies-Boulding-index

# Within cluster distances
def within_cluster_dist(X,centroids,cluster_table):
    """Calculates distance for points from their index"""

    # Initialize list for the cluster sums
    sums = []

    # Loop over clusters
    for i in range(0,len(centroids)):

        # Filter for datapoints in the given cluster
        temp = cluster_table[cluster_table[:,1]==i]
        temp_sum = 0

        # Calculate distance between point and its corresponding cluster
        # Sum them for a given cluster
        if len(temp)!=0:
            for j in range(0,len(temp)):
                index = int(temp[j,0])

                current_dist = np.linalg.norm(X[index,:]-centroids[i,:])
                temp_sum += current_dist

            temp_sum = math.sqrt(temp_sum/len(temp))
```

```
        # Add current value to the list
        sums.append(temp_sum)

    return sums

# Cluster separation
def cluster_separation(centroids):
    """Calculates cluster separation metrics based on the centroids"""

    # Get combinations of clusters
    combinations = int(len(centroids)*(len(centroids)-1)/2)

    separations = np.zeros((combinations,3))


    counter = 0

    # Loop over cluster combinations
    # Calculate distance between their centroids
    for i in range(0, len(centroids)):
        for j in range(i+1, len(centroids)):
            separations[counter,0] = i
            separations[counter,1] = j

            separations[counter,2] = np.linalg.norm(centroids[i,]-centroids[j,])

            counter += 1

    return separations

# Calculates Davies-Bouldien index
def DB_index(X,centroids,cluster_table):
    """Calculates Davies-Bouldien index for the clustering"""

    # Calculates within cluster distances
    sums = within_cluster_dist(X,centroids,cluster_table)

    # Calculates cluster centroid distances
    separations = cluster_separation(centroids)

    # Calculate intermediary metric
    r_s = separations.copy()
    for i in range(0, len(r_s)):
        index_1 = int(r_s[i,0])
        index_2 = int(r_s[i,1])
        r_s[i,2] = (sums[index_1]+sums[index_2])/separations[i,2]
```

```
    d = []

    # Loop over clusters
    for j in range(0,len(centroids)):

        # Calculate DB-index values for the clusters
        if j==0:
            temp1 = r_s[r_s[:,0]==j]
            max1 = max(temp1[:,2])
            d.append(max1)
        elif j==len(centroids)-1:
            temp2 = r_s[r_s[:,1]==j]
            max2 = max(temp2[:,2])
            d.append(max2)
        else:
            temp1 = r_s[r_s[:,0]==j]
            max1 = max(temp1[:,2])
            temp2 = r_s[r_s[:,1]==j]
            max2 = max(temp2[:,2])
            final_max = max(max1,max2)
            d.append(final_max)

    # Average DB index values across clusters
    DB_index_val = sum(d)/len(centroids)

    return DB_index_val


## Functions for the Dunn-index
def max_within_distances(X,centroids,cluster_table):
    """Calculates maximum distance between any 2 points in the same cluster"""

    # Initialize list for holding result for each cluster
    max_within_distances = []

    # Loop over clusters
    for k in range(0,len(centroids)):

        # Filter for points inside the given cluster
        temp_table = cluster_table[cluster_table[:,1]==k]
        temp_max = 0

        # Loop over points in the given cluster
        for i in range(0,len(temp_table)):
            for j in range(i+1,len(temp_table)):
                index_1 = temp_table[i,0]
                index_2 = temp_table[j,0]
```

```
                current_value = np.linalg.norm(X[index_1,]-X[index_2,])

                # Get the maximum among all
                if current_value>temp_max:
                    temp_max = current_value

        # Append to the lists of results
        max_within_distances.append(temp_max)

    return max_within_distances


def min_between_distances(X,centroids,cluster_table):
    """Calculates minimum distance between any 2 point in different clusters"""

    # Get combination of clusters
    combinations = int(len(centroids)*(len(centroids)-1)/2)

    # Initialize separation metric
    separations = np.zeros((combinations,3))

    counter = 0

    # Loop over cluster combination
    for i in range(0, len(centroids)):
        for j in range(i+1, len(centroids)):
            separations[counter,0] = i
            separations[counter,1] = j

            temp_table1 = cluster_table[cluster_table[:,1]==i]
            temp_table2 = cluster_table[cluster_table[:,1]==j]

            # Calculate distances between all points in different clusters

            if len(temp_table1)>0 and len(temp_table2)>0:
                index_1 = temp_table1[0,0]
                index_2 = temp_table2[0,0]

                temp_min = np.linalg.norm(X[index_1,]-X[index_2,])
            else:
                temp_min = 1000

            for l in range(0,len(temp_table1)):
                for h in range(0,len(temp_table2)):
                    index_1 = temp_table1[l,0]
                    index_2 = temp_table2[h,0]
```

```
                    current_value = np.linalg.norm(X[index_1,]-X[index_2,])

                    # Choose the overall minimum
                    if current_value<temp_min:
                        temp_min = current_value



            separations[counter,2] = temp_min

            counter += 1

    return separations

def dunn_index(X,centroids,cluster_table):
    """Calculates the Dunn-index for the clustering"""

    # Calculates minimum distance between points in different clusters
    separations_temp = min_between_distances(X,centroids,cluster_table)

    # Calculates maximum distance between points in the same cluster
    max_within_distances_temp = max_within_distances(X,centroids,cluster_table)

    result = min(separations_temp[:,2])/max(max_within_distances_temp)

    return result


### Function for checking indexes, effects of different parameter values

## Hyperparameter optimization functions for MST
def mst_index_params(X,n=300,p=2,K_true=3,std=0.9,K_hat=5, d=5):

    result = mst_cluster(X,k = K_hat,d=d)
    mst_centroids = centroid_calc_for_mst(K=K_hat,d=d,result=result,X=X)
    db_final = DB_index(X,mst_centroids,result)
    dunn_final = dunn_index(X,mst_centroids,result)

    return db_final,dunn_final

def mst_index_params_plotter(param,start_value,end_value,X):
    dbs=[]
    dunns=[]
    integers = ['n','p','K_true','K_hat']
    floats = ['std']
```

```
    if param in integers:
        iterators = np.arange(start_value,end_value,1)
    else:
        iterators = np.arange(start_value,end_value,0.1)

    for i in iterators:

        temp_dict = {str(param) : i}
        db_final,dunn_final = mst_index_params(**temp_dict,X=X)

        dbs.append(db_final)
        dunns.append(dunn_final)

    plt.plot(iterators,dbs)
    plt.plot(iterators,dunns)
    plt.xlabel('Parameter: {}'.format(param))
    plt.ylabel('Index value')
    plt.legend(['DB-index','Dunn-index'])
    plt.title("Index values of the final K-means clustering as a function of:␣
 ↪{}".format(param))
    plt.show()


## Hyperparameter optimization functions for K-means
def index_params(X,n=300,p=2,K_true=3,std=0.9,K_hat=5):

    centroids, cluster_table, cost_values_db, cost_values_dunn =␣
 ↪k_means_clustering(K_hat,X)
    db_final = cost_values_db[-1]
    dunn_final = cost_values_dunn[-1]

    return db_final,dunn_final



def index_params_plotter(param,start_value,end_value,X):
    dbs=[]
    dunns=[]
    integers = ['n','p','K_true','K_hat']
    floats = ['std']

    if param in integers:
        iterators = np.arange(start_value,end_value,1)
    else:
        iterators = np.arange(start_value,end_value,0.1)

    for i in iterators:
```

```
        temp_dict = {str(param) : i}

        db_final,dunn_final = index_params(X=X, **temp_dict)

        dbs.append(db_final)
        dunns.append(dunn_final)

    plt.plot(iterators,dbs)
    plt.plot(iterators,dunns)
    plt.xlabel('Parameter: {}'.format(param))
    plt.ylabel('Index value')
    plt.legend(['DB-index','Dunn-index'])
    plt.title("Index values of the final K-means clustering as a function of:␣
↪{}".format(param))
    plt.show()
```

2.4 Testing

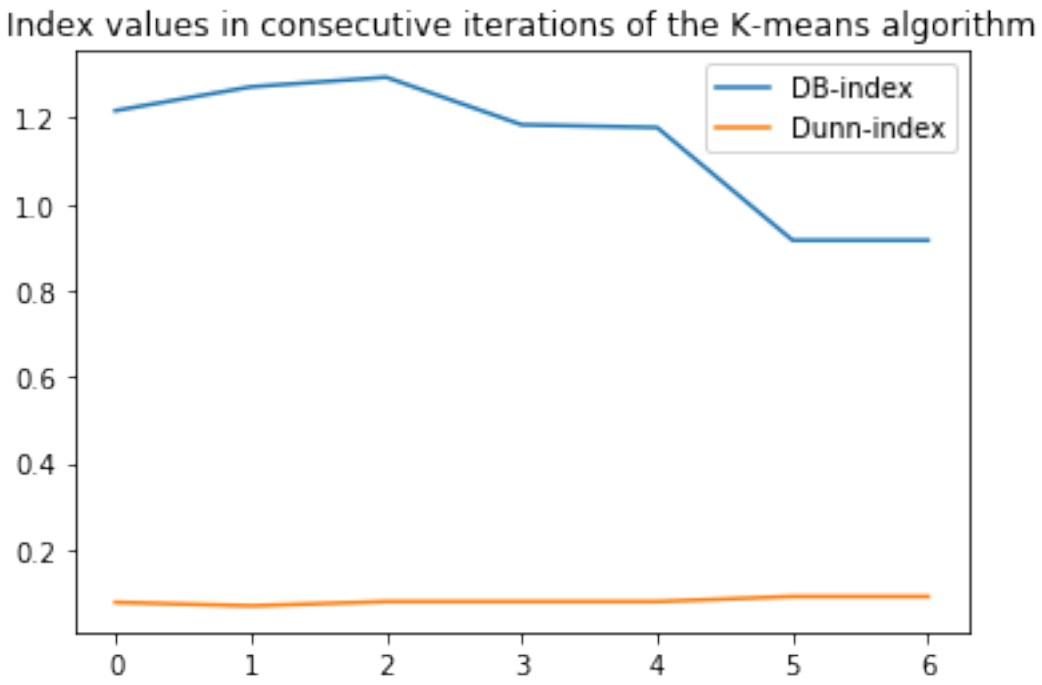
In the following cells we tested our functions on datasets that we generated.

[2]:
```
X, y = benchmark_data_generation(n=60,p=5,K=3,std = 0.9)
```

[5]:
```
centroids, cluster_table, cost_values_db, cost_values_dunn =␣
 ↪k_means_clustering(3,X)
print(centroids)
#print(cluster_table)
print(cost_values_db)
plt.plot(cost_values_db)
plt.plot(cost_values_dunn)
plt.legend(['DB-index','Dunn-index'])
plt.title("Index values in consecutive iterations of the K-means algorithm")
plt.show()

print(cost_values_dunn[-1])
```

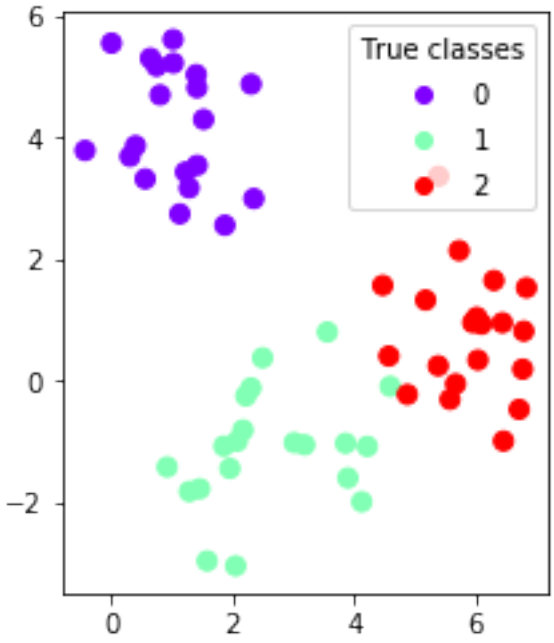
```
[11 44 12]
[[ 0.94911493  3.89817053  2.53632861  0.81152158 -1.11692219]
 [ 1.28569847  4.87781436  1.41575479  1.44381288 -2.14817799]
 [ 4.25398686 -0.16892258  4.56324773  8.71641963 -5.46704232]]
[1.216508301285701, 1.2716500855396795, 1.2939148119467905, 1.183920398478965,
1.1775926227234323, 0.9164733363684815, 0.9164733363684815]
```

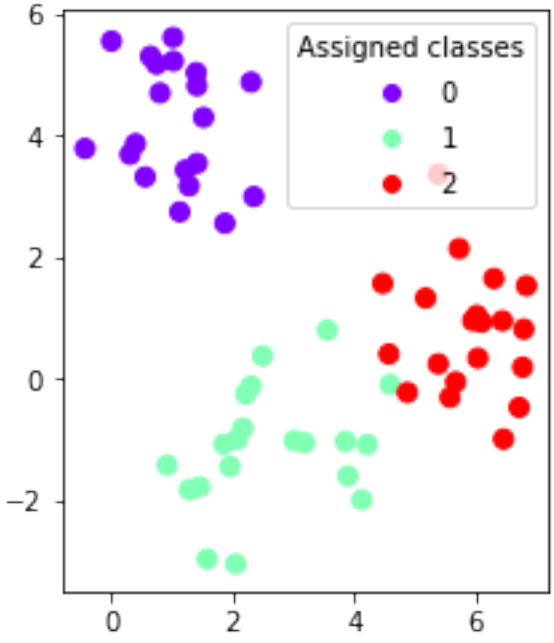



```
0.09175944544178387
```

[33]:
```
# Scatter plot - ORIGINAL
fig,ax = plt.subplots()
plt.axes().set_aspect('equal')
scatter=plt.scatter(X[:, 0], X[:, 1], c=y, s=50 , cmap='rainbow');
# produce a legend with the unique colors from the scatter
legend1 = plt.legend(*scatter.legend_elements(),
                     loc="upper right", title="True classes")
plt.show()

# Scatter plot - K-MEANS
fig,ax = plt.subplots()
plt.axes().set_aspect('equal')
scatter=plt.scatter(X[:, 0], X[:, 1], c=cluster_table[:,1], s=50 ,␣
 ↪cmap='rainbow');
# produce a legend with the unique colors from the scatter
legend1 = plt.legend(*scatter.legend_elements(),
                     loc="upper right", title="Assigned classes")
plt.show()
```


True classes
0
1
2
6
4
2
0
−2
0
2
4
6


Assigned classes
0
1
2
6
4
2
0
−2
0
2
4
6

2.5 Running algorithms with the real benchmark data

In the following section we will use the real benchmark data (Synthetic and Thyriod) for our algorithms and functions. We will read in the data, run MST and k-means clustering, then compare the index values for the different methods on the different datasets.

2.6 The 2-dimensional dataset (Synthetic)

This section is for the 2-dimensional (Synthetic) dataset.

Reading in data

[3]:
```
## Read in the data
synthetic = open("./Data/Synthetic.rtf").read()

## Transform the data into the format that our functions use

# Deleting the unnecessary parts of the text,
# which do not contain values
synthetic = synthetic[395:]
synthetic = synthetic.split("\\")

# Clean string values
synthetic = [w.replace('\n', '') for w in synthetic]
synthetic = [w.replace(' ', '') for w in synthetic]
synthetic = synthetic[:-1]

# Turn the strings into integer variables
synthetic_col1 = [int(w[:6]) for w in synthetic]
synthetic_col2 = [int(w[6:]) for w in synthetic]

# Check the length of the lists to see if method worked
print(len(synthetic_col1))
print(len(synthetic_col2))

# Convert the lists into numpy arrays and
synthetic = np.asarray((synthetic_col1,synthetic_col2))
synthetic = synthetic.T



print(synthetic.shape)
print(synthetic)
```

```
5000
5000
(5000, 2)
```

```
[[664159 550946]
 [665845 557965]
 [597173 575538]
 …
 [650661 861267]
 [599647 858702]
 [684091 842566]]
```

2.6.1 K-means method

Running the K-means algorithm

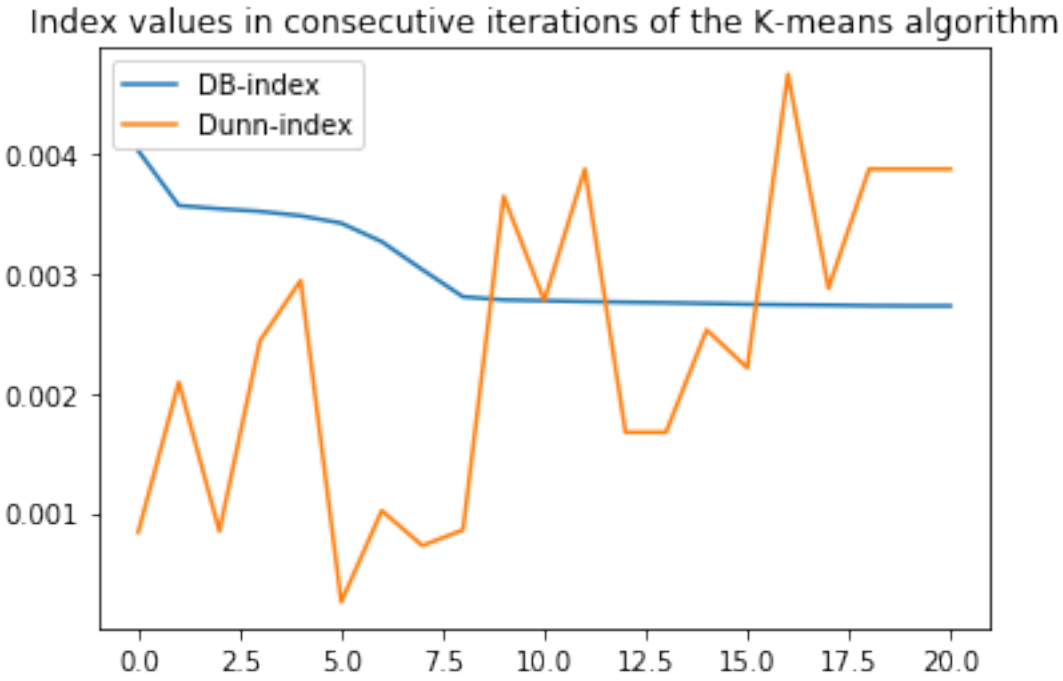
[4]:
```
X = synthetic
```

[35]:
```
centroids, cluster_table, cost_values_db, cost_values_dunn =␣
↪k_means_clustering(15,X)
```

```
[1113 1358 4739   57 1515 3965 3073 1628 2754 4588 3955  924  156 1933
 4681]
```

Calculating index values

[36]:
```
plt.plot(cost_values_db)
plt.plot(cost_values_dunn)
plt.legend(['DB-index','Dunn-index'])
plt.title("Index values in consecutive iterations of the K-means algorithm")
plt.show()
```



As we can see on the graph above, the index values are moving to the "right" direction as the algorithm iterates.

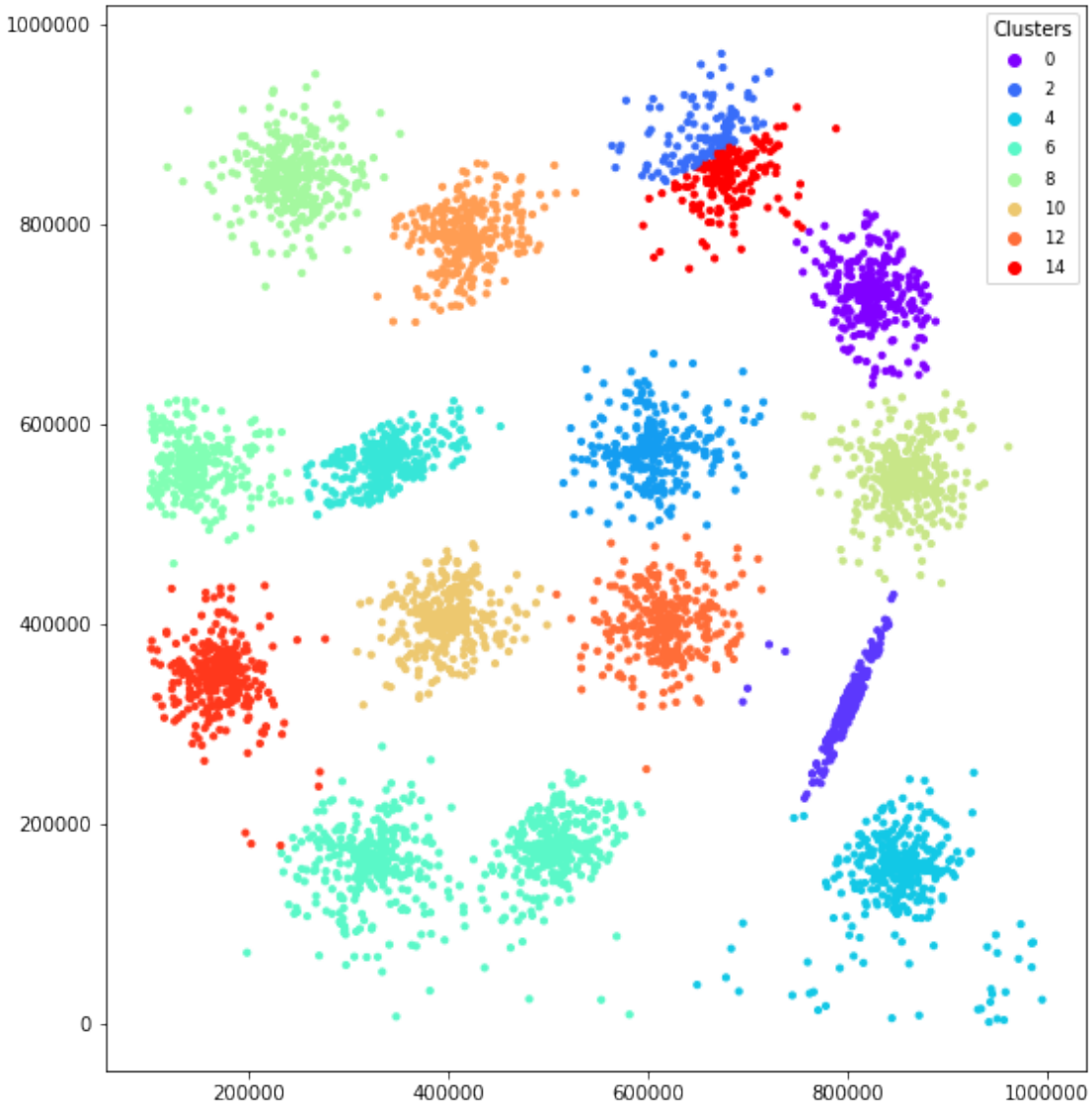
Visualizing the clusters

[37]:
```
# Scatter plot - K-MEANS
fig,ax = plt.subplots(figsize=(10,10))
plt.axes().set_aspect('equal')

scatter=plt.scatter(X[:, 0], X[:, 1], c=cluster_table[:,1], s=10 ,␣
↪cmap='rainbow');

# Produce a legend with the unique colors for each cluster
legend1 = plt.legend(*scatter.legend_elements(),
                     fontsize = 'small',
                    loc="upper right", title="Clusters")


plt.show()
```



As we can see on the graph, the k-means algorithm does a pretty good job in identifying the 15 clusters (the color scale can be misleading, but there are indeed 15 clusters). However there are minor issues (e.g. the green cluster in the lower-left part could be splitted into 2, while the red-blue clusters in the upper-right part could be combined). Probably choosing different starting points could help making the algorithm even better.

[38]:
```
# Sanity-check for the clusters
# Printing cluster centroids

c=cluster_table[:,1]
np.unique(c)
print(centroids)
```

```
[[823867.75709779 730721.7192429 ]
 [801616.78164557 321123.34177215]
 [654914.40336134 890818.42016807]
 [606574.95622896 574455.16835017]
 [853279.48209366 146410.25344353]
 [337858.94189602 562276.80428135]
 [416725.13688761 166630.40778098]
 [148488.48366013 557102.04901961]
 [244654.8856305  847642.04105572]
 [858947.9713467  546259.65902579]
 [398555.94857143 404855.06857143]
 [417799.69426752 787001.99363057]
 [617546.43916914 399072.54599407]
 [169810.22686567 345693.58507463]
 [679734.11914894 848011.50212766]]
```

2.6.2 Minimum Spanning Tree clustering

In the following cells the MST will be implemented for the 2-dimensional dataset.

Running the Kruskal-algorithm

[39]:
```
from datetime import datetime

X = synthetic

print(datetime.now())

result = mst_cluster_UF(X, k=15)


print(datetime.now())
```

```
2019-11-28 12:31:52.027677
2019-11-28 12:33:21.894338
```

Calculating index values

[40]:
```
mst_centroids = centroid_calc_for_mst(K=15,d=2,result=result,X=X)
print(mst_centroids)
```

```
[[616988.37306677 524459.85439457]
 [327946.20642202 818425.62538226]
 [850803.28869048 153712.10119048]
 [246507.31962025 559933.13765823]
 [958332.72222222  43664.5       ]
 [139601.         914203.        ]
```

```
[415389.79076479 168369.87445887]
[679941.2         57837.2       ]
[858691.           6050.        ]
[567751.          15690.        ]
[198355.          70290.        ]
[364875.5         19233.        ]
[569075.          86609.        ]
[481435.          23870.        ]
[789095.         895634.        ]]
```

[41]:
```
## Calculate index values for MST
db_temp = DB_index(X,mst_centroids,result)
dunn_temp = dunn_index(X,mst_centroids,result)

print(db_temp)
print(dunn_temp)
```

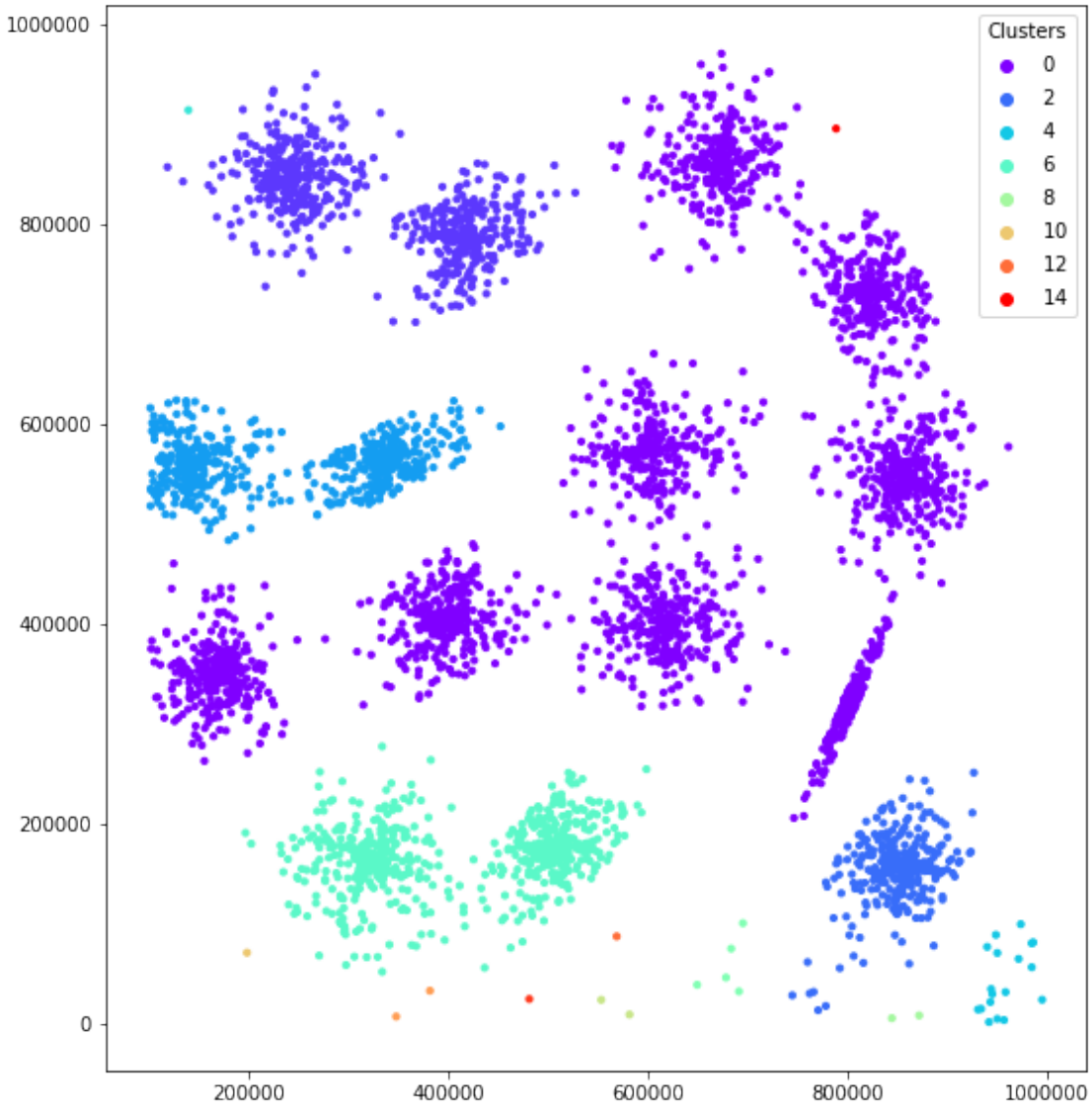
```
0.002238924867906986
0.04991540360875162
```

[42]:
```
# Calculate index values for K-Means
print(cost_values_db[-1])
print(cost_values_dunn[-1])
```

```
0.0027281758285410624
0.0038690461254900304
```

Visualizing the clusters

[43]:
```
# Scatter plot - MST/Kruskal
fig,ax = plt.subplots(figsize=(10,10))
plt.axes().set_aspect('equal')
scatter=plt.scatter(X[:, 0], X[:, 1], c=result[:,1], s=10 , cmap='rainbow');
# produce a legend with the unique colors from the scatter
legend1 = plt.legend(*scatter.legend_elements(),
                     loc="upper right", title="Clusters")
plt.show()
```



As we can see from the figure, the MST makes one huge cluster, some medium-sized ones and some tiny clusters (with only a few elements). From the figure it seems that the k-means clustering does a better job, however we need to compare the index values to have an objective comparison.

2.6.3 Comparing index values for K-means and MST

[44]:
```
### Comparing index values

## K-means
# Calculate index values for K-Means
db_kmeans = cost_values_db[-1]
dunn_kmeans = cost_values_dunn[-1]
```

```
## MST
# Calculate index values for MST
db_mst = DB_index(X,centroids,result)
dunn_mst = dunn_index(X,centroids,result)

print("The DB-index for K-means is {} and for MST it is {}".
 ↪format(db_kmeans,db_mst))
print("The Dunn-index for K-means is {} and for MST it is {}".
 ↪format(dunn_kmeans,dunn_mst))
```

```
The DB-index for K-means is 0.0027281758285410624 and for MST it is
0.009739720936506723
The Dunn-index for K-means is 0.0038690461254900304 and for MST it is
0.04991540360875162
```

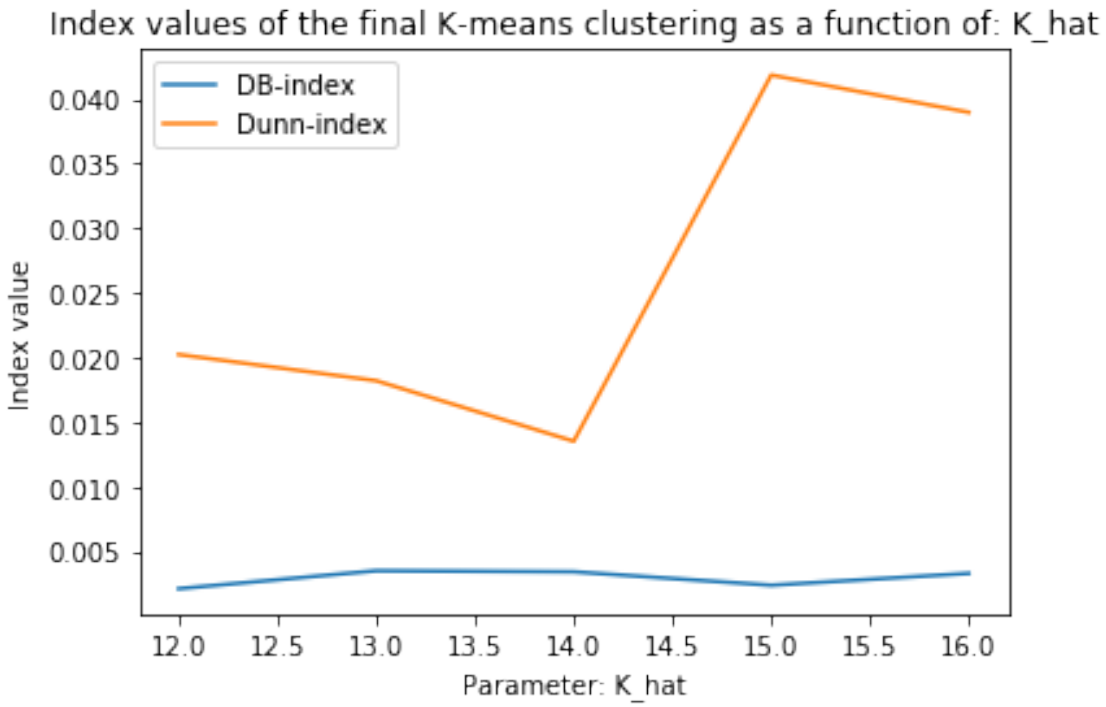
Based on the indices, we have different results, since the k-means algorithm performs better on the DB-index (lower the better), whereas MST performs better in the Dunn-index (higher the better).

2.6.4 Optimizing over K (number of hypothetical clusters)

In this section we check whether our functions can recover that for the Synthetic dataset 15 clusters is the best (best index values). If this is true, we can have confidence that repeating the same procedure will help us to find the optimal, but a priori unknown number of clusters in the Thyroid dataset. We are using a random sample with 1000 observations for this experiment and we are checking potential cluster numbers from 12 to 17.

[13]:
```
X_test = X[np.random.choice(X.shape[0], 1000, replace=False), :]
print(X_test.shape)
index_params_plotter('K_hat',12,17, X_test)
```

```
(1000, 2)
[750 784 979 605 655 153  13 419 529 886 990 471]
[360 693 718 975 962 493 374 114 720 680 380 234 466]
[370 568 854 529 857 972  98  69 570 534 130 859 921 106]
[ 63  11 606 254 407 461 943 982  94 868 410 231 435 888 201]
[544 636  19  77 649 714 859 432 547 785 734 205 362 995 586 763]
```



We can see from the figure that our procedure is able to recover the true values since the DB-index is lowest at 15 and the Dunn-index is highest at 15 as the number of clusters. Therefore we can have confidence that the same procedure will be able to find the optimal number of clusters in the other case as well.

2.7 The 5 dimensional dataset (Thyroid)

In the following section we will use our clustering methods and algorithms on the 5-dimensional dataset.

Reading in data

[46]:
```
## Read in the data
thyroid = pd.read_csv('./Data/Thyroid_new.csv', header=None)
print(thyroid.head())
print(thyroid.shape)
```

```
         0        1        2        3        4
0  3195023  3455331  3497964  3068822  3206710
1  3651455  3412754  4131996  3248619  3603214
2  4716462  4051411  3638860  3150548  2946503
3  3347167  2433481  3075276  3150548  3058020
4  3042879  2859252  3004828  3166893  2859768
(215, 5)
```

2.7.1 K-means method

[47]:
```
X_5d = np.array(thyroid)
```

Running the k-means algorithm

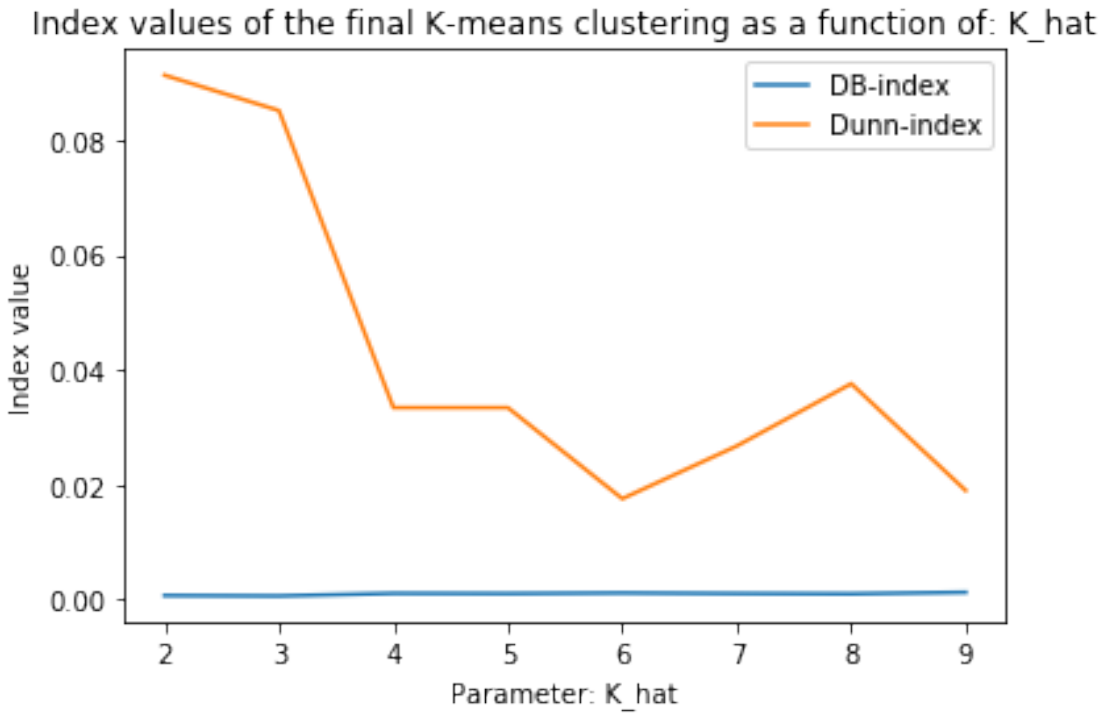
[48]:
```
X_5d = np.array(thyroid)
print(X_5d.shape)
centroids, cluster_table, cost_values_db, cost_values_dunn =␣
↪k_means_clustering(3,X_5d)
```

```
(215, 5)
[ 4 77 72]
```

Optimizing over K (number of hypothetical clusters)

[49]:
```
index_params_plotter(X = X_5d, param = 'K_hat',start_value = 2,end_value = 10)
```

```
[206  49]
[105 169  95]
[ 19 201  24 173]
[ 51 117 183  95  50]
[46 90 60 87 73 33]
[118 142 132  21  53 166  42]
[135  55  98 207 127 204  38 139]
[212  96 184  91  12  69  18 214 145]
```



From this figure we can see that the optimal number of clusters is either 2,3 or maybe 8.

Fitting the optimal model

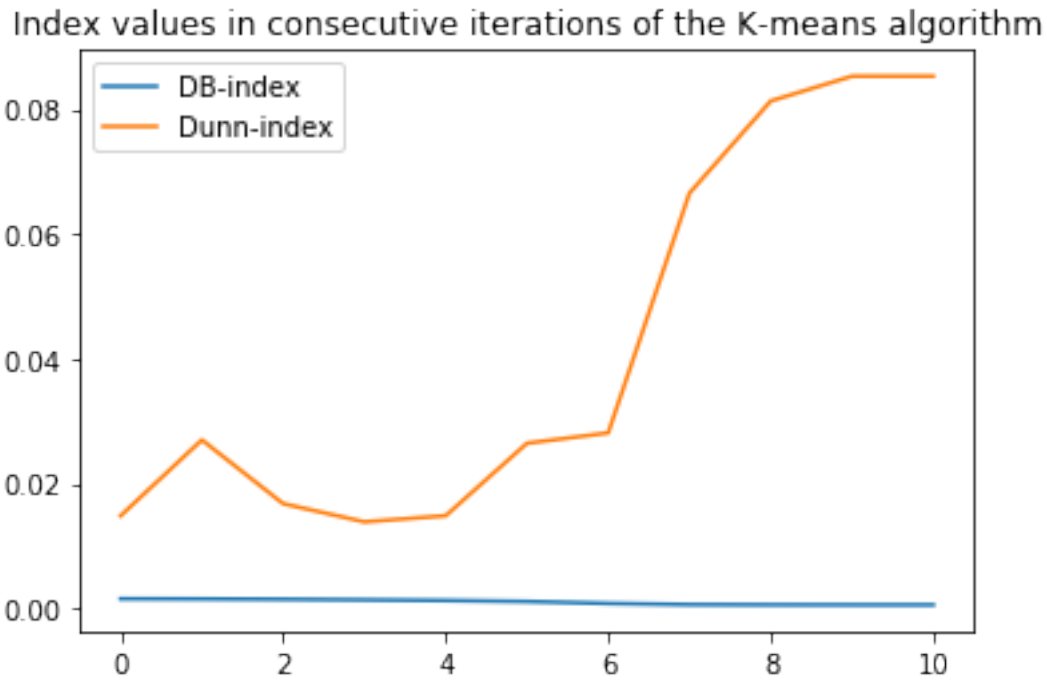
[50]:
```
centroids, cluster_table, cost_values_db, cost_values_dunn =␣
↪k_means_clustering(3,X_5d)
```

```
[ 14  69 147]
```

Calculating the index values

[51]:
```
plt.plot(cost_values_db)
plt.plot(cost_values_dunn)
plt.legend(['DB-index','Dunn-index'])
plt.title("Index values in consecutive iterations of the K-means algorithm")
plt.show()

#print(cost_values_dunn[-1])
```



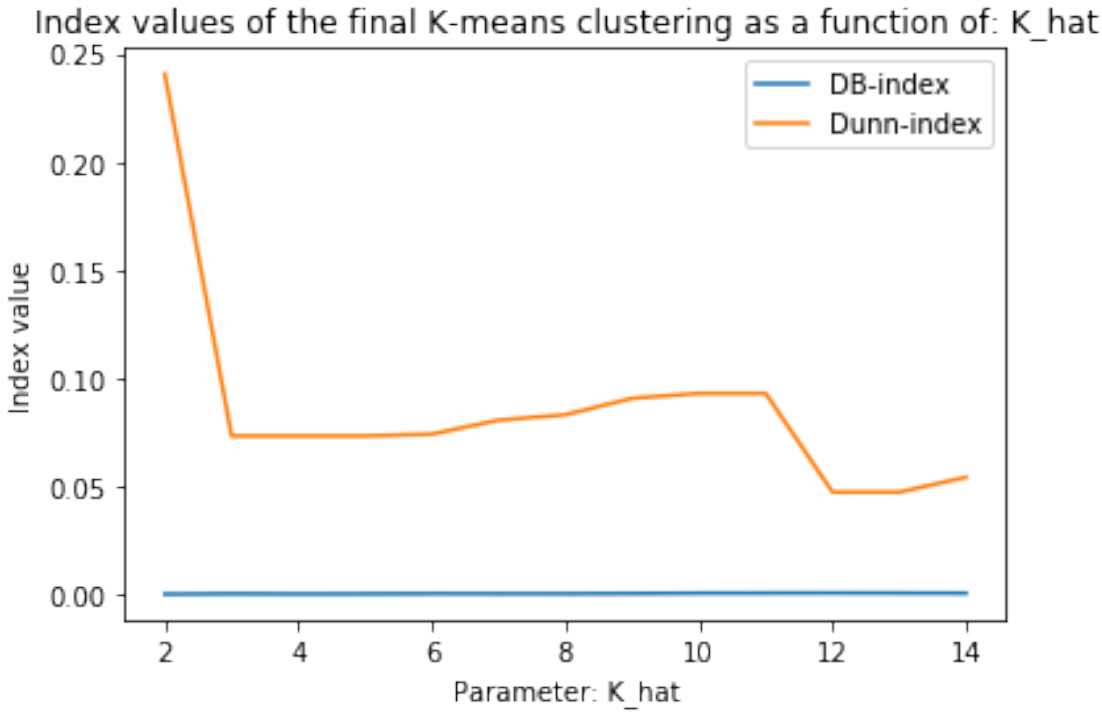
We can see from this figure that as the algorithm iterates, the indices move to the "right" direction.

2.7.2 Minimum Spanning Tree clustering

Running the Kruskal algorithm

Optimizing over K (number of hypothetical clusters)

[53]:
```
mst_index_params_plotter(param = 'K_hat',start_value = 2,end_value = 15,X=X_5d)
```



[52]:
```
# Using the optimised number of clusters from k-means
result_5d = mst_cluster_UF(X_5d, k=3)
```

Based on this analysis the optimal number of clusters is 2 or maybe 11, so MST finds a different optimum than K-means (which found 2,3 or 8 as the best number of clusters for MST).

We are using one of the optimal number of clusters found by our k-means procedure (**3 clusters in this case**), to be able to compare the index values for k-means and MST.

Calculating the index values

[54]:
```
mst_centroids_5d = centroid_calc_for_mst(K=3,d=5, result=result_5d, X=X_5d)
print(mst_centroids_5d)

## Calculate index values for MST
db_temp = DB_index(X_5d,mst_centroids_5d,result_5d)
dunn_temp = dunn_index(X_5d,mst_centroids_5d,result_5d)
```

```
print(db_temp)
print(dunn_temp)
```

```
[[ 3388238.39906103  3408656.6056338   3401387.48356808  3322133.55399061
   3373723.2629108 ]
 [ 4107886.          1475497.          2441244.         12140369.
   5548567.        ]
 [ 3575383.          1858691.          2441244.          9623219.
   5226406.        ]]
0.00016728039090530914
0.24070869611275858
```

2.7.3 Comparing index values for K-means and MST

[55]:
```
### Comparing index values

## K-means
# Calculate index values for K-Means
db_kmeans = cost_values_db[-1]
dunn_kmeans = cost_values_dunn[-1]

## MST
# Calculate index values for MST
db_mst = DB_index(X_5d,mst_centroids_5d,result_5d)
dunn_mst = dunn_index(X_5d,mst_centroids_5d,result_5d)

print("The DB-index for K-means is {} and for MST it is {}".
 ↪format(db_kmeans,db_mst))
print("The Dunn-index for K-means is {} and for MST it is {}".
 ↪format(dunn_kmeans,dunn_mst))
```

```
The DB-index for K-means is 0.0006178983874467176 and for MST it is
0.00016728039090530914
The Dunn-index for K-means is 0.08526102745007716 and for MST it is
0.24070869611275858
```

Based on the index values MST has better performance in both DB-, and Dunn-index in the 5-dimensional dataset (Thyriod) than k-means.

2.8 Summary

In this programming project we implemented MST and k-means clustering for a 2-dimensional and 5-dimensional dataset. K-means clustering seemed to work better for the 2-dimensional dataset with 15 clusters, whereas MST clustering was performing better for the 5 dimensional dataset with 3 clusters.