HW02

July 11, 2020

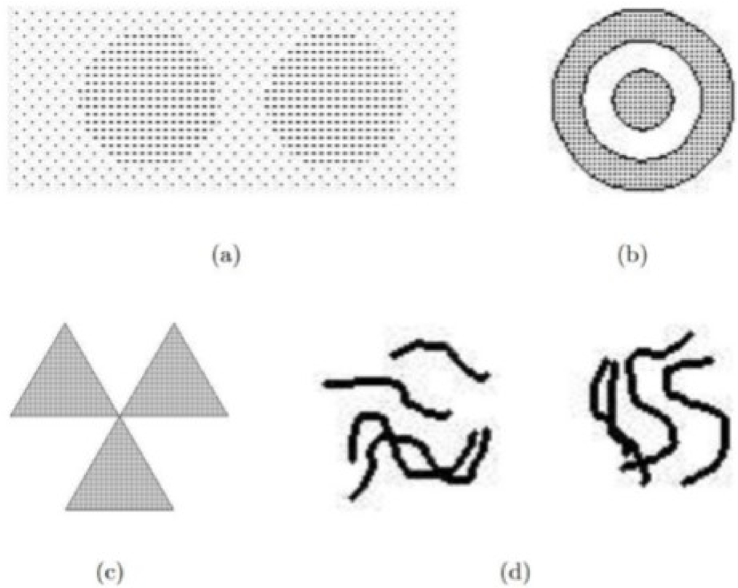
1 Data Mining - HW02

98722278 - Mohammad Doosti Lakhani

1.1 1

For simplicity, 1, 2, 3, 4 correspond to a, b, c, d in original question and A, B, C are center, contiguity, and density based clustering approachs respectively. Entries with star represent the best solutions (my own view).

1. Fig a)
    1. 2 clusters: The centers will be add the center of two circles and noises in the rectangle will be incorporated into clusters.
    2. 1 cluster: Due to existance of noise between to circles, they will be merged into a single cluster.
    3. 2 clusters: 2 circles as the only dense area will be chosen as clusters and noises will be detected to separated from clusters. *
2. Fig b)
    1. 1 cluster: The center will be at the center of concentric area of both circles otherwise, outer circle will be considered as noise.
    2. 2 clusters: Each ring will represent a separate cluster. *
    3. 2 clusters: Each ring will represent a separate cluster as they are dense w.r.t. their neighbors. * PS. result of 2 and 3 would be similar due to non-existance of noise.
3. Fig c)
    1. 1 or 3 clusters:
        1. 1 cluster where center is the joint point.
        2. 3 is the better answer which each center will be at the center of each triangle to maintain equal distance. *
    2. 1 cluster: The joint area of three triangle will cause it
    3. 1 or 3 clusters:
        1. 1 cluster: If the triangles *overlap* and the joint area is not a single point, then it is possible w.r.t. to different paramters.
        2. 3 clusters: Each triangle is the most dense area.
4. Fig d)



1.1

$$J = \frac{M_{11}}{M_{01} + M_{10} + M_{11}}$$

2.1

1. 2 clusters: Each center will be near the center of a circle containig all the threads.
2. 4 or 5 clusters: Assume that threads with joint area as a separate cluster, then left side will have 2 or 3 clusters and right side 2 clusters.
3. 2 or 3 clusters: If we choose min distance large enough, then 2 if not it can be 3 as there is a considerable distance between threads in left side but best answer would be 2. *

1.2 2

Let's say we have two binary vectors:

`M11` represents the total number of attributes where A and B both have a value of 1. `M01` represents the total number of attributes where the attribute of A is 0 and the attribute of B is 1. `M10` represents the total number of attributes where the attribute of A is 1 and the attribute of B is 0. `M00` represents the total number of attributes where A and B both have a value of 0.

Jaccard:

SMC:

As we can see, the only difference between SMC and Jaccard is that SMC adds `M00` in numinator and denominator.

On the other side, if we define Hamming *similiarty* (1 - Hamming Distance) as the number of similar bits, the `Hamming / len(vector) = SMC`. Hence, `SMC` is normalized `Hamming`. Note that the Hamming distance between two equal-length strings of symbols is the number of positions at which the corresponding symbols are different.

And Also, we can define Cosine similarity as normalized dot product which demonstrates

$$\mathrm{SMC} = \frac{\text{number of matching attributes}}{\text{number of attributes}}$$
$$= \frac{M_{00} + M_{11}}{M_{00} + M_{01} + M_{10} + M_{11}}$$

2.2

$$\text{similarity} = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}},$$

2.3

similarities of 1s in vectors which is `M11` where is holding the definition of Jaccard.

1.3 3

1.4 4

The answer would be NO. The main point again is the distance metric which in our case we assume it is Euclidean distance. In this scenario, euclidean distance will produce higher results for attributes with higher mean or variance toward particular magnitude of space such as skewness.

If we do not normalized our data, then one variable with higher variance/mean tend to have higher impact on the result and so the clustering is biased toward that particular variable. By removing this bias/variance using normalization, we can ensure that all attributes are incroporated with same proportion.

Also, Kmeans using euclidean distance tends to create circular clusters so if data has skewness then normalization will have much higher impact as normalized data is distributed around center of coordinate (mean=0, variance=1) which is not true about unnormalized data.

Note that in all of interpretation, different mean and variance for different attributes mean different physical/real world intrepretation. If two attributes have different mean/variance but are in same domain, then standardazation might not be necessary as the one with higher/lower mean/variance may have more/lower importance.

In the end, all of these views depend on the prespective of tasks but normally, normalization helps numerical stability of algorithm.

1.5 5

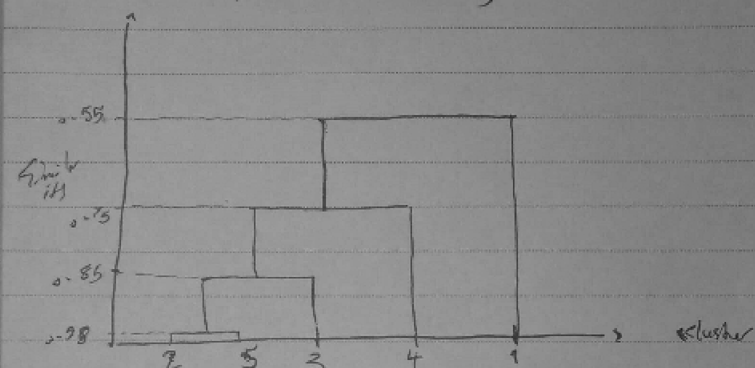
1. Robustness to noise:
    1. Single Link: If we look at the definition which is merging two clusters whose two closest points have the smallet distance, we can find out that this approach do not care about largest distances which directly corresponds to the outliers and noises. Hence, this approach is not robust to noises.
    2. Complete Link: Again, its definition states that we only merge two clusters where merger cluster has smaller diameter and if we try to demonstrate outliers and noises,

3

Min :

1. Sort descending :

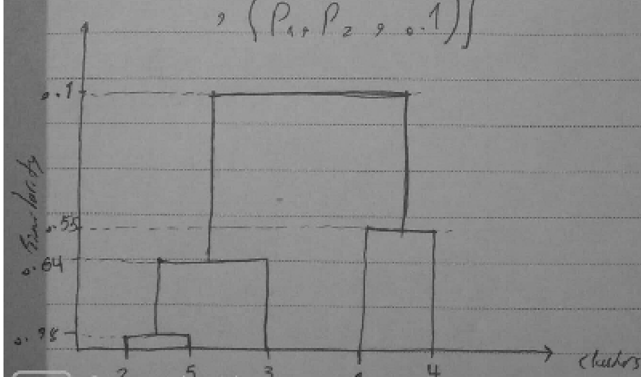
[(P_2, P_5, 0.98), (P_3, P_5, 0.85), (P_4, P_5, 0.76), (P_4, P_1, 0.55)]



Max :

1. Sort descending

[(P_2, P_5, 0.98), (P_3, P_2, 0.64), (P_1, P_4, 0.55), (P_1, P_2, 0.1)]



Page:

3

we will see them in a distance away from the dense diameter of a cluster, then this approach works as a threshold and removes most of the noises and outliers.
3. Average Link is a combo of two previously mentioned approachs which can be controlled to have robustness to noise w.r.t. complete link attribute or tendency of single link approach which try to construct clusters in form of long threads.

2. Time Complexity:

1. Single Link: In this scenario, in each step we compute a distance between clusters which is `n^2`. Then we one list of smallest distances for each cluster for merging time which takes `n` to be updated. Also, in each step, we need to update proximity matrix which takes `n` too. But all of this can be done while distances are computing and then sequentially so worst case would be `O(n^2)`.
2. Complete Link: The main difference from Single link is that in Single Link if `i` and `j` are merged, then the best merger for `k` is either `i` and `j` and that is why we only need to save smallest distance for each cluster rather than a list for each cluster. This is not true for complete link because any cluster rather than `i` or `j` can be best merger for `k`, hence we need to maintain a sorted list of smallest distances for each cluster which takes `log n` time so we have `O(log(n).n^2)`.
3. Average Groupd Link: Based on image, same time complexity can be considered for Average Group Link

ref: Introduction to Information Retrieval, By Christopher D. Manning, Prabhakar Raghavan & Hinrich Schütze

1.6 DBSCAN

```
In [2]: import numpy as np
        import pandas as pd
        from copy import deepcopy
        import matplotlib.pyplot as plt

        from sklearn.metrics.pairwise import euclidean_distances
        from sklearn import cluster
```

```
In [5]: data = pd.read_csv('../data/data.csv', header=None)
        print('Number of samples: ', len(data))
        data.plot.scatter(0, 1, figsize=(15, 10))
        plt.show()

        data = data.values
```

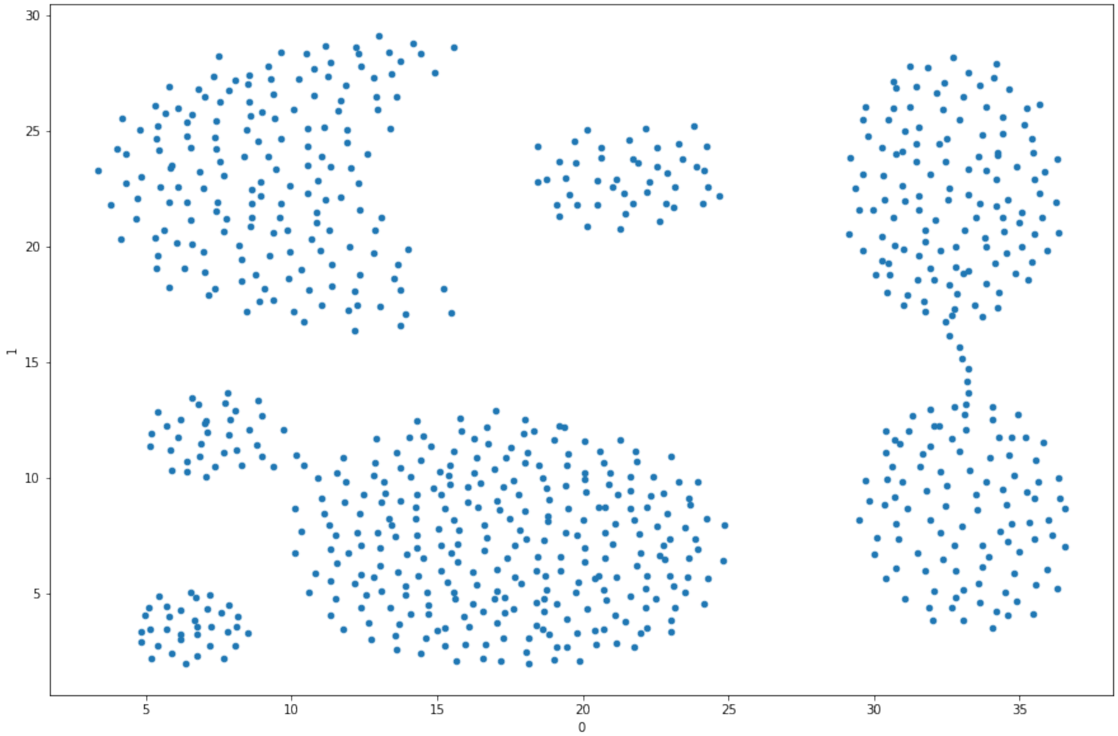
```
Number of samples:  788
```

```
EfficientHAC(d⃗_1, ..., d⃗_N)
 1  for n ← 1 to N
 2  do for i ← 1 to N
 3       do C[n][i].sim ← d⃗_n · d⃗_i
 4          C[n][i].index ← i
 5       I[n] ← 1
 6       P[n] ← priority queue for C[n] sorted on sim
 7       P[n].Delete(C[n][n]) (don't want self-similarities)
 8  A ← []
 9  for k ← 1 to N − 1
10  do k_1 ← arg max_{k:I[k]=1} P[k].Max().sim
11     k_2 ← P[k_1].Max().index
12     A.Append(⟨k_1, k_2⟩)
13     I[k_2] ← 0
14     P[k_1] ← []
15     for each i with I[i] = 1 ∧ i ≠ k_1
16     do P[i].Delete(C[i][k_1])
17        P[i].Delete(C[i][k_2])
18        C[i][k_1].sim ← Sim(i, k_1, k_2)
19        P[i].Insert(C[i][k_1])
20        C[k_1][i].sim ← Sim(i, k_1, k_2)
21        P[k_1].Insert(C[k_1][i])
22  return A
```

clustering algorithm	$\text{SIM}(i, k_1, k_2)$
single-link	$\max(\text{SIM}(i, k_1), \text{SIM}(i, k_2))$
complete-link	$\min(\text{SIM}(i, k_1), \text{SIM}(i, k_2))$
centroid	$(\frac{1}{N_m}\vec{v}_m) \cdot (\frac{1}{N_i}\vec{v}_i)$
group-average	$\frac{1}{(N_m+N_i)(N_m+N_i-1)}[(\vec{v}_m + \vec{v}_i)^2 - (N_m + N_i)]$

compute $C[5]$	1	2	3	4	5
	0.2	0.8	0.6	0.4	1.0
create $P[5]$ (by sorting)	2	3	4	1	
	0.8	0.6	0.4	0.2	
merge 2 and 3, update similarity of 2, delete 3	2	4	1		
	0.3	0.4	0.2		
delete and reinsert 2	4	2	1		
	0.4	0.3	0.2		

► **Figure 17.5** The priority-queue algorithm for HAC. Top: The algorithm. Center: Four different similarity measures. Bottom: An example for processing steps 6 and 16–19. This is a made up example showing $P[5]$ for a 5×5 matrix C.



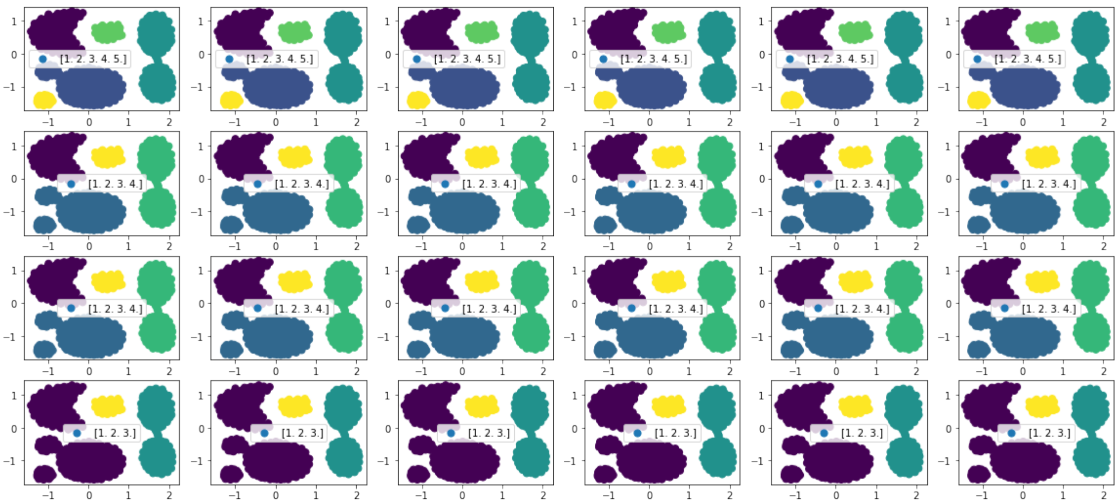

```
In [7]: mean = data.mean()
        std = data.std()
        data = (data - mean) / std
```

```
In [8]: class DBSCAN:
            def __init__(self, min_samples, eps):
                """
                Constructs DBSCAN given parameters of neighborhood

                :param min_samples: Minimum samples within eps radius to be consider as a core
                :param eps: Radius of core point
                """
                self.min_samples = min_samples
                self.eps = eps

                self.labels = None  # '0': Haven't processed, '-1': noise, 'C': cluster number
                self.core_points = None

            def fit_predict(self, x, *args, **kwargs):
                """
                Fits the data using DBSCAN and returns labels and core points
                Order of data matter!

                Algorithm:
```

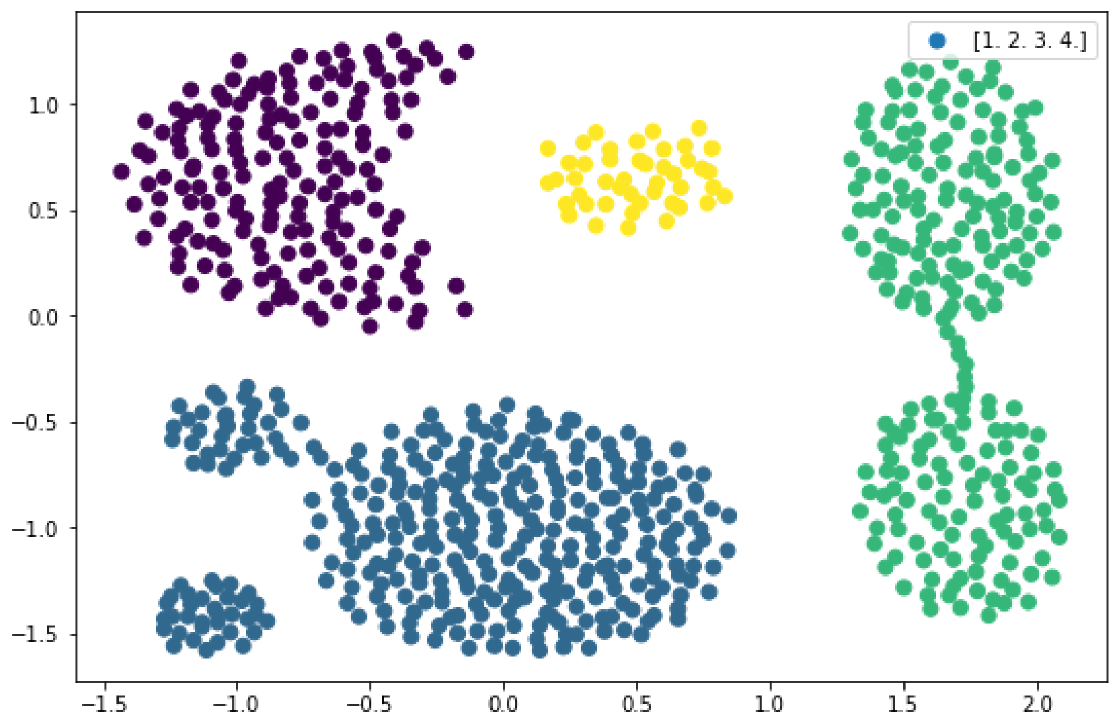
```
        1. Consider a list of points that have not been seen yet
        2. Read an arbitrary point until there is no unseen point left
        3. If there are at least ``min_samples`` points within a radius of ``eps``
           then all these points are from same cluster
        4. Expand this cluster for its all core points for all neighbors
        5. Repeat

    :param x: N-dimensional numpy array

    :return: A tuple of labels of each point and index of core points
        where label=-1 corresponds to noise data and label=N N>=1 demonstrates clu
    """

    self.labels = np.zeros((len(x),))
    self.core_points = np.zeros((len(x),))
    current_cluster = 1  # we use 1->inf

    for pnt in range(len(x)):
        # if self.labels[pnt] == -1 or self.labels[pnt] >= 1:
        if self.labels[pnt] == 0:
            neighbor_indices = self.__nearest_neighbors(x, x[pnt])

            if len(neighbor_indices) >= self.min_samples:
                self.__expand(x, pnt, current_cluster)
                current_cluster += 1
            else:  # noise/outlier scenario
                self.labels[pnt] = -1
    return self.labels, self.core_points

def __nearest_neighbors(self, data, point):
    """
    Finds points near to the point ``point`` within the range of ``eps``

    :param point: A point
    :param: All points

    :return: Indices of nearest neighbor points
    """
    distances = euclidean_distances(data, point.reshape(1, -1))
    neighbors = distances <= self.eps
    topk = np.argsort(distances, axis=0)
    neighbors_idx = np.max(neighbors[topk].nonzero()[0]) + 1
    return topk[:neighbors_idx].flatten()

def __expand(self, data, point_idx, current_cluster):
    """
    Expands ``current_cluster`` using given point w.r.t. ``eps`` and ``min_samples
    Algorithm:
```

```
            1. Get a point as the start point for ``current_cluster``
            2. Get its neighbors and go through them one by one using queue logic
            3. If the neighbor is noise, then add it to the current cluster, if it is 
               then add them to the list of neighbors of original point
            4. Repeat step 2 and 3 until all points in the list of neighbors are proce

        :param data: Whole data to be clustered
        :param point_idx: The index of a point of the current cluster as the start poi
        :param current_cluster: The label of current cluster
        :return: None
        """

        self.labels[point_idx] = current_cluster
        neighbors_indices = deepcopy(self.__nearest_neighbors(data, data[point_idx]))

        while len(neighbors_indices) > 0:
            neighbor_point = neighbors_indices[0]
            neighbors_indices = np.delete(neighbors_indices, 0, 0)
            if self.labels[neighbor_point] == -1:
                self.labels[neighbor_point] = current_cluster
            elif self.labels[neighbor_point] == 0:
                self.labels[neighbor_point] = current_cluster
                neighbors_indices_neighbor_point = self.__nearest_neighbors(data, data
                if len(neighbors_indices_neighbor_point) >= self.min_samples:
                    neighbors_indices = np.concatenate((neighbors_indices, neighbors_in
                    self.core_points[neighbor_point] = 1
```

In [9]:
```
parameters = {'eps': [0.25, 0.3, 0.35, 0.4], 'min_samples': [3, 4, 5, 6, 7, 10]}

plt.figure(figsize=(20, 9))
i = 0
for dist in parameters['eps']:
    for min_pnt in parameters['min_samples']:
        dbscan = DBSCAN(min_samples=min_pnt, eps=dist)
        y_dbscan, _ = dbscan.fit_predict(data)
        plt.subplot(len(parameters['eps']), len(parameters['min_samples']), i+1)
        i += 1
        plt.scatter(data[:, 0], data[:, 1], c=y_dbscan, s=50, cmap='viridis', label=np
        plt.legend()
plt.show()
```




```
In [12]: plt.figure(figsize=(9, 6))
         dbscan = DBSCAN(min_samples=5, eps=0.3)
         y_dbscan, centers = dbscan.fit_predict(data)
         plt.scatter(data[:, 0], data[:, 1], c=y_dbscan, s=50, cmap='viridis', label=np.unique
         plt.legend()
         plt.show()
```

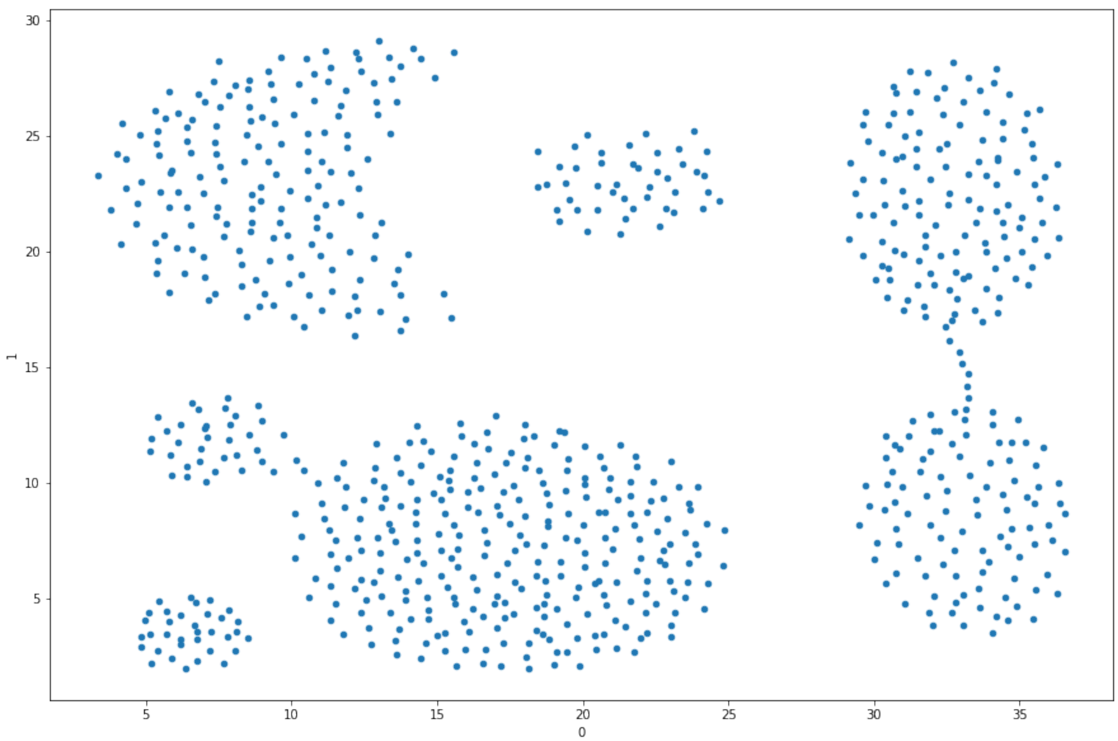


1.7 KMeans

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
```

In [3]:
```
data = pd.read_csv('../data/data.csv', header=None)
print('Number of samples: ',len(data))
data.plot.scatter(0, 1, figsize=(15, 10))
plt.show()

data = data.values
```

```
Number of samples:  788
```

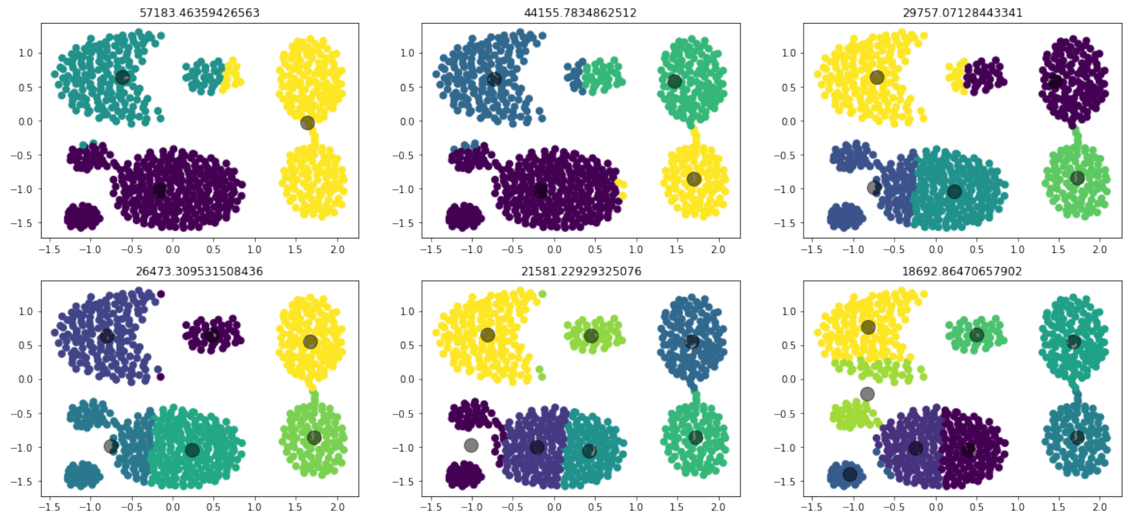


In [4]:
```
# normalization
mean = data.mean()
std = data.std()
data = (data - mean) / std
```

In [19]:
```
# DBI
from sklearn.metrics.pairwise import euclidean_distances
from sklearn.metrics import make_scorer
```

```
from sklearn.cluster import KMeans

def DB_loss_function(estimator, X, y_true=None):
    """
    Computes Davis-Bolding Index for a fitted KMeans

    args:
        model: Fitted KMeans object
        x: input data for evaluation
    """
    preds = estimator.predict(X)
    n_clusters = int(np.max(preds))+1

    db_values = []
    for i in range(n_clusters):
        for j in range(i+1, n_clusters):
            cluster_i = X[preds == i]
            cluster_j = X[preds == j]

            avg_cluster_i = 2 * np.sum(euclidean_distances(cluster_i, cluster_i)) / le
            avg_cluster_j = 2 * np.sum(euclidean_distances(cluster_j, cluster_j)) / le

            u_cluster_i = np.sum(cluster_i, axis=0) / len(cluster_i)
            u_cluster_j = np.sum(cluster_j, axis=0) / len(cluster_j)
            db = (avg_cluster_i + avg_cluster_j) / np.sum(euclidean_distances(
                u_cluster_i.reshape(-1,1),
                u_cluster_j.reshape(-1,1)))
            db_values.append(db)
    dbi = np.sum(np.array(db_values)) / n_clusters
    return dbi

DBI_Scorer = make_scorer(DB_loss_function, greater_is_better=False)

n_clusters = np.arange(3, 9)
plt.figure(figsize=(20, 9))

for i, n in enumerate(n_clusters):
    kmeans = KMeans(n_clusters=n, init='random')
    kmeans.fit(data)
    y_kmeans = kmeans.predict(data)

    plt.subplot(2, 3, i+1)
    plt.scatter(data[:, 0], data[:, 1], c=y_kmeans, s=50, cmap='viridis')

    centers = kmeans.cluster_centers_
    plt.scatter(centers[:, 0], centers[:, 1], c='black', s=200, alpha=0.5)
    plt.title(DB_loss_function(kmeans, data))
```

```
plt.show()
```




```
In [ ]: # grid search model training

        from sklearn.model_selection import PredefinedSplit
        from sklearn.model_selection import GridSearchCV

        parameters = {'n_clusters':list(np.arange(3, 10)), 'init':['random'],
                      'algorithm':('elkan', 'full'), 'tol':[0.0001, 0.001, 0.00005, 0.00001, 0

        best_model = None
        best_score = np.inf

        for n_c in parameters['n_clusters']:
            for al in parameters['algorithm']:
                for t in parameters['tol']:
                    for i in parameters['max_iter']:
                        kmeans = KMeans(**{'n_clusters':n_c, 'algorithm':al, 'tol': t, 'init':
                        kmeans.fit(data)
                        dbi = DB_loss_function(kmeans, data)
                        if dbi < best_score:
                            print('model config: ', str(kmeans), '\n', 'dbi score:', dbi)
                            best_score = dbi
                            best_model = kmeans
        best_model
```

```
model config:  KMeans(algorithm='elkan', copy_x=True, init='random', max_iter=300,
       n_clusters=3, n_init=10, n_jobs=None, precompute_distances='auto',
       random_state=None, tol=0.0001, verbose=0)
 dbi score: 57183.46359426563
```

```
model config:  KMeans(algorithm='full', copy_x=True, init='random', max_iter=300, n_clusters=3
       n_init=10, n_jobs=None, precompute_distances='auto', random_state=None,
       tol=0.0001, verbose=0)
 dbi score: 57081.73694047507
model config:  KMeans(algorithm='elkan', copy_x=True, init='random', max_iter=300,
       n_clusters=4, n_init=10, n_jobs=None, precompute_distances='auto',
       random_state=None, tol=0.0001, verbose=0)
 dbi score: 44155.783486251195
model config:  KMeans(algorithm='elkan', copy_x=True, init='random', max_iter=300,
       n_clusters=4, n_init=10, n_jobs=None, precompute_distances='auto',
       random_state=None, tol=5e-05, verbose=0)
 dbi score: 43970.91716060179
model config:  KMeans(algorithm='elkan', copy_x=True, init='random', max_iter=300,
       n_clusters=5, n_init=10, n_jobs=None, precompute_distances='auto',
       random_state=None, tol=0.0001, verbose=0)
 dbi score: 29757.0712844334
model config:  KMeans(algorithm='elkan', copy_x=True, init='random', max_iter=1000,
       n_clusters=5, n_init=10, n_jobs=None, precompute_distances='auto',
       random_state=None, tol=0.0001, verbose=0)
 dbi score: 29727.338048239977
model config:  KMeans(algorithm='full', copy_x=True, init='random', max_iter=300, n_clusters=5
       n_init=10, n_jobs=None, precompute_distances='auto', random_state=None,
       tol=0.0001, verbose=0)
 dbi score: 29727.338048239973
model config:  KMeans(algorithm='elkan', copy_x=True, init='random', max_iter=300,
       n_clusters=6, n_init=10, n_jobs=None, precompute_distances='auto',
       random_state=None, tol=0.0001, verbose=0)
 dbi score: 26433.63656724541
model config:  KMeans(algorithm='elkan', copy_x=True, init='random', max_iter=1000,
       n_clusters=6, n_init=10, n_jobs=None, precompute_distances='auto',
       random_state=None, tol=0.001, verbose=0)
 dbi score: 26339.338568787352
model config:  KMeans(algorithm='elkan', copy_x=True, init='random', max_iter=300,
       n_clusters=6, n_init=10, n_jobs=None, precompute_distances='auto',
       random_state=None, tol=0.0005, verbose=0)
 dbi score: 26339.33856878734
model config:  KMeans(algorithm='elkan', copy_x=True, init='random', max_iter=300,
       n_clusters=7, n_init=10, n_jobs=None, precompute_distances='auto',
       random_state=None, tol=0.0001, verbose=0)
 dbi score: 21686.59445720419
model config:  KMeans(algorithm='elkan', copy_x=True, init='random', max_iter=300,
       n_clusters=7, n_init=10, n_jobs=None, precompute_distances='auto',
       random_state=None, tol=0.001, verbose=0)
 dbi score: 21648.295649490286
model config:  KMeans(algorithm='elkan', copy_x=True, init='random', max_iter=1000,
       n_clusters=7, n_init=10, n_jobs=None, precompute_distances='auto',
       random_state=None, tol=5e-05, verbose=0)
 dbi score: 21581.229293250763
```

```
model config:  KMeans(algorithm='full', copy_x=True, init='random', max_iter=300, n_clusters=7
       n_init=10, n_jobs=None, precompute_distances='auto', random_state=None,
       tol=0.001, verbose=0)
 dbi score: 21581.22929325076
model config:  KMeans(algorithm='elkan', copy_x=True, init='random', max_iter=300,
       n_clusters=8, n_init=10, n_jobs=None, precompute_distances='auto',
       random_state=None, tol=0.0001, verbose=0)
 dbi score: 17244.949445195867
model config:  KMeans(algorithm='elkan', copy_x=True, init='random', max_iter=300,
       n_clusters=8, n_init=10, n_jobs=None, precompute_distances='auto',
       random_state=None, tol=0.001, verbose=0)
 dbi score: 17230.001778258502
model config:  KMeans(algorithm='full', copy_x=True, init='random', max_iter=300, n_clusters=8
       n_init=10, n_jobs=None, precompute_distances='auto', random_state=None,
       tol=0.0001, verbose=0)
 dbi score: 17184.318107677605
model config:  KMeans(algorithm='full', copy_x=True, init='random', max_iter=300, n_clusters=8
       n_init=10, n_jobs=None, precompute_distances='auto', random_state=None,
       tol=0.001, verbose=0)
 dbi score: 17162.25184721984
model config:  KMeans(algorithm='elkan', copy_x=True, init='random', max_iter=300,
       n_clusters=9, n_init=10, n_jobs=None, precompute_distances='auto',
       random_state=None, tol=0.0001, verbose=0)
 dbi score: 16081.128539591753
model config:  KMeans(algorithm='elkan', copy_x=True, init='random', max_iter=1000,
       n_clusters=9, n_init=10, n_jobs=None, precompute_distances='auto',
       random_state=None, tol=0.0001, verbose=0)
 dbi score: 16064.42404214852
model config:  KMeans(algorithm='elkan', copy_x=True, init='random', max_iter=300,
       n_clusters=9, n_init=10, n_jobs=None, precompute_distances='auto',
       random_state=None, tol=0.001, verbose=0)
 dbi score: 15811.632441043039
model config:  KMeans(algorithm='elkan', copy_x=True, init='random', max_iter=1000,
       n_clusters=9, n_init=10, n_jobs=None, precompute_distances='auto',
       random_state=None, tol=0.001, verbose=0)
 dbi score: 14904.520570886412
model config:  KMeans(algorithm='elkan', copy_x=True, init='random', max_iter=300,
       n_clusters=9, n_init=10, n_jobs=None, precompute_distances='auto',
       random_state=None, tol=1e-05, verbose=0)
 dbi score: 13705.64565308541
model config:  KMeans(algorithm='elkan', copy_x=True, init='random', max_iter=300,
       n_clusters=9, n_init=10, n_jobs=None, precompute_distances='auto',
       random_state=None, tol=0.0005, verbose=0)
 dbi score: 13690.561580760907
```

```
Out[ ]: KMeans(algorithm='elkan', copy_x=True, init='random', max_iter=300,
               n_clusters=9, n_init=10, n_jobs=None, precompute_distances='auto',
```

```
        random_state=None, tol=0.0005, verbose=0)
```

```
In [ ]: best_model
```

```
Out[ ]: KMeans(algorithm='elkan', copy_x=True, init='random', max_iter=300,
               n_clusters=9, n_init=10, n_jobs=None, precompute_distances='auto',
               random_state=None, tol=0.0005, verbose=0)
```

```
In [ ]: y_kmeans = best_model.predict(data)

        plt.figure(figsize=(15, 10))
        plt.scatter(data[:, 0], data[:, 1], c=y_kmeans, s=50, cmap='viridis')
        centers = best_model.cluster_centers_
        plt.scatter(centers[:, 0], centers[:, 1], c='black', s=200, alpha=0.5, label='centers')
        plt.legend()

        DB_loss_function(best_model, data)
```

```
Out[ ]: 13690.561580760907
```