Homework #6 Answers

Chongye Feng

In [1]:
```
# importing libraries
import numba
import numpy as np
import pandas as pd
from pylab import *
from mpl_toolkits.mplot3d import axes3d
from scipy.optimize import minimize

import seaborn as sns

from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import accuracy_score
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split

# setting the random seed
np.random.seed(0)
```

Q1

In [2]:
```
compounds = pd.read_csv('compounds.csv')
compounds.head(3)
```

Out[2]:

	A	B	C	D	type
0	6.4	2.9	4.3	1.3	amide
1	5.7	4.4	1.5	0.4	phenol
2	6.7	3.0	5.2	2.3	ether

a:

In [3]:
```
compounds.columns
```

Out[3]:
```
Index(['A', 'B', 'C', 'D', 'type'], dtype='object')
```

In [4]:
```
features = ['A', 'B', 'C', 'D']

for feature in features:
    up_limit = compounds[feature].max()
    bot_limit = compounds[feature].min()
    compounds[feature] = (compounds[feature] - bot_limit) / (up_limit

compounds.head(3)
```

Out[4]:

	A	B	C	D	type
0	0.583333	0.375000	0.559322	0.500000	amide
1	0.388889	1.000000	0.084746	0.125000	phenol
2	0.666667	0.416667	0.711864	0.916667	ether

In [5]:
```
from sklearn.decomposition import PCA
```

In [6]:
```
features_df = compounds[features]
features_df.head(3)
```

Out[6]:

	A	B	C	D
0	0.583333	0.375000	0.559322	0.500000
1	0.388889	1.000000	0.084746	0.125000
2	0.666667	0.416667	0.711864	0.916667

In [7]:
```
pca = PCA(n_components=None)

x_pca = pca.fit_transform(features_df)

x_pca.shape
```

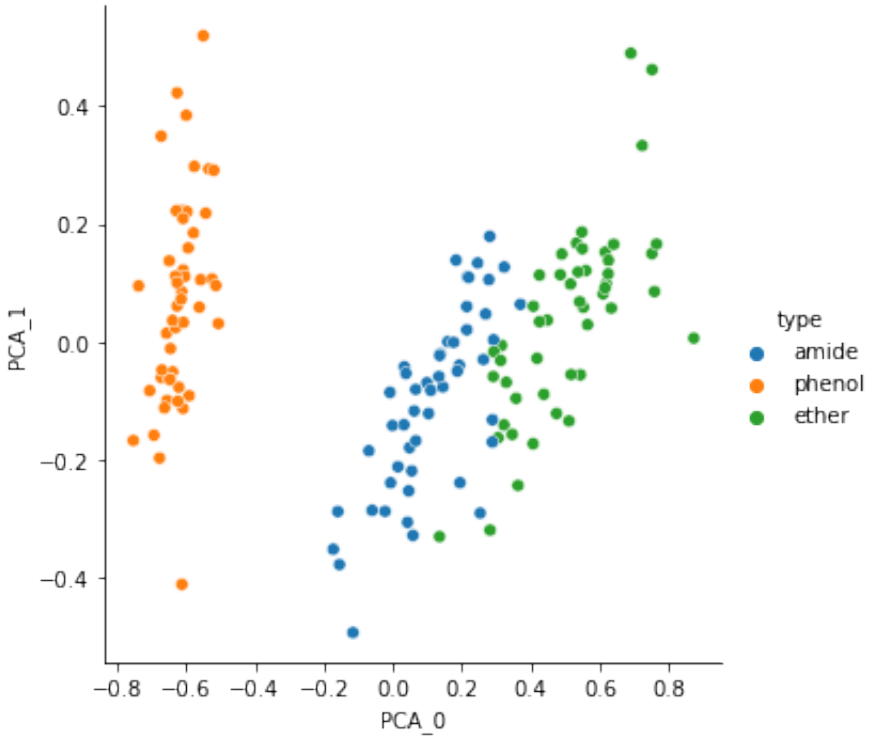
Out[7]:
```
(150, 4)
```

In [8]:
```
X_pca = pd.DataFrame(x_pca,index = features_df.index,columns = [f'PCA_

X_pca.head(3)
```

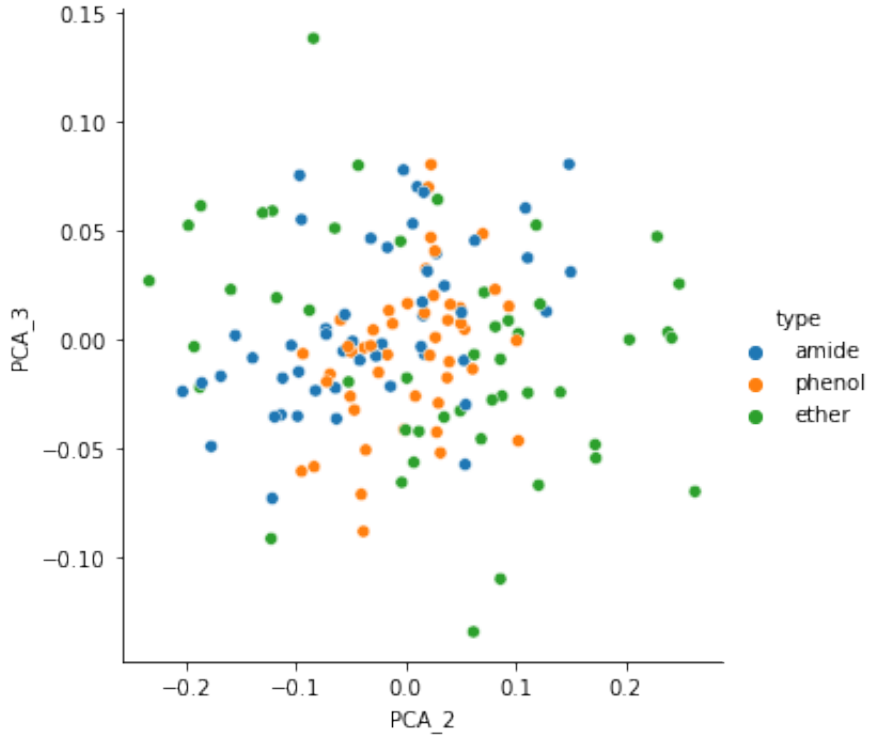
Out[8]:

	PCA_0	PCA_1	PCA_2	PCA_3
0	0.158971	0.000675	-0.112446	-0.017676
1	-0.549943	0.518968	0.039335	-0.010145
2	0.551755	0.058990	0.085968	-0.109943

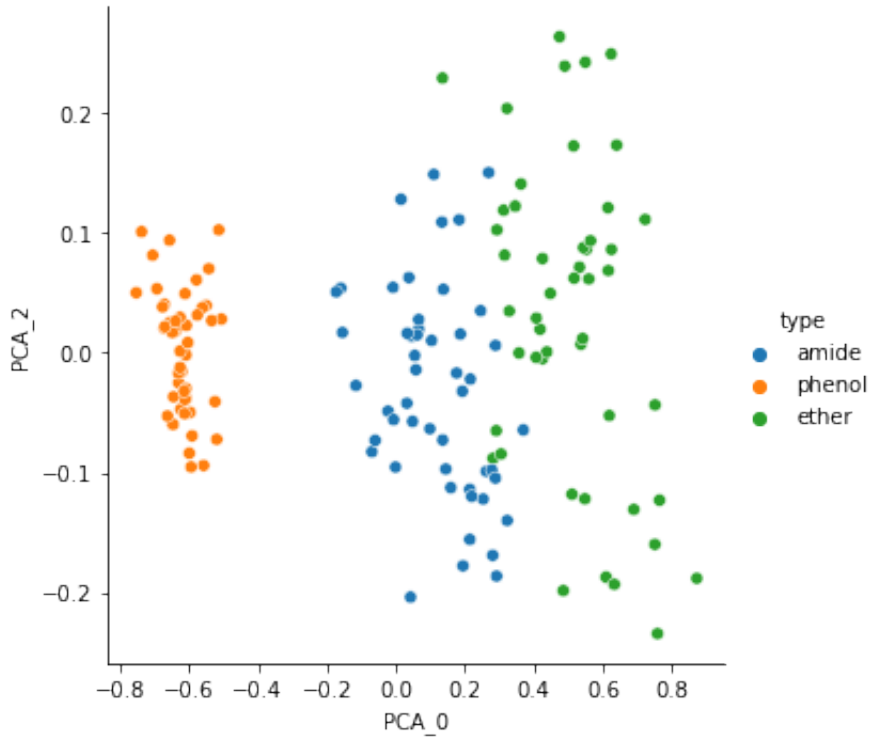
In [9]:
```
ax1 = sns.relplot(data=X_pca, x="PCA_0", y="PCA_1",hue=compounds['type
```



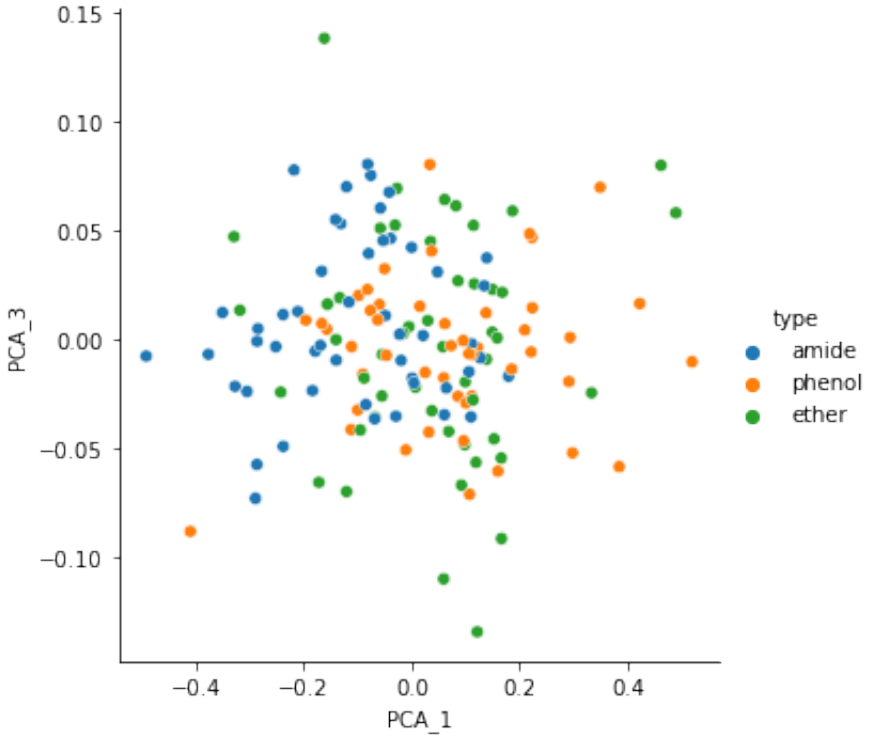
In [10]:
```
ax2 = sns.relplot(data=X_pca, x="PCA_2", y="PCA_3",hue=compounds['type
```



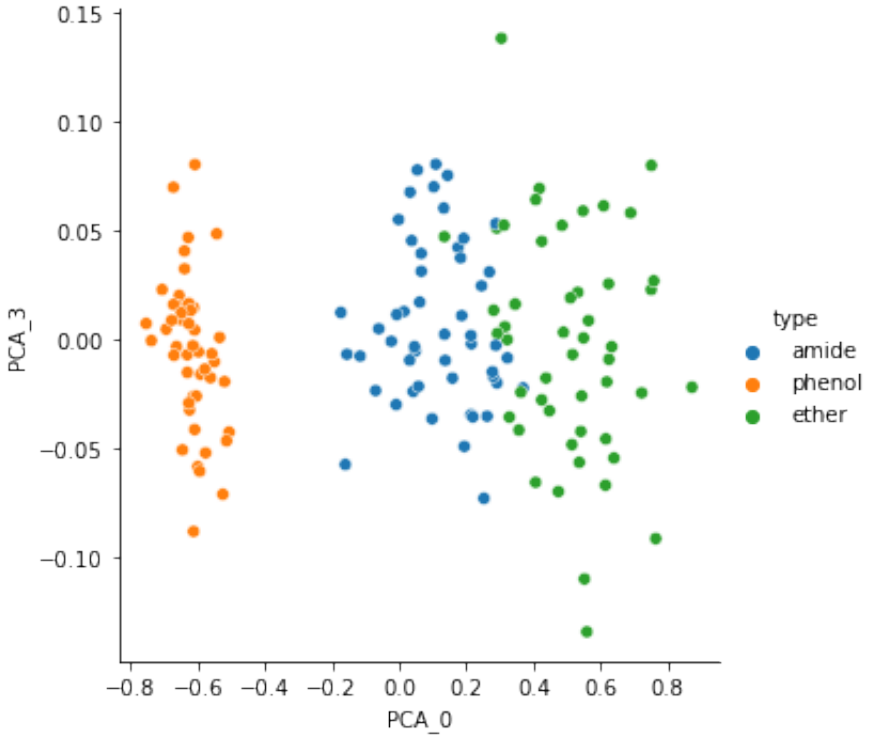
In [11]:
```
ax3 = sns.relplot(data=X_pca, x="PCA_0", y="PCA_2",hue=compounds['type
```



In [12]:
```
ax4 = sns.relplot(data=X_pca, x="PCA_1", y="PCA_3",hue=compounds['type
```

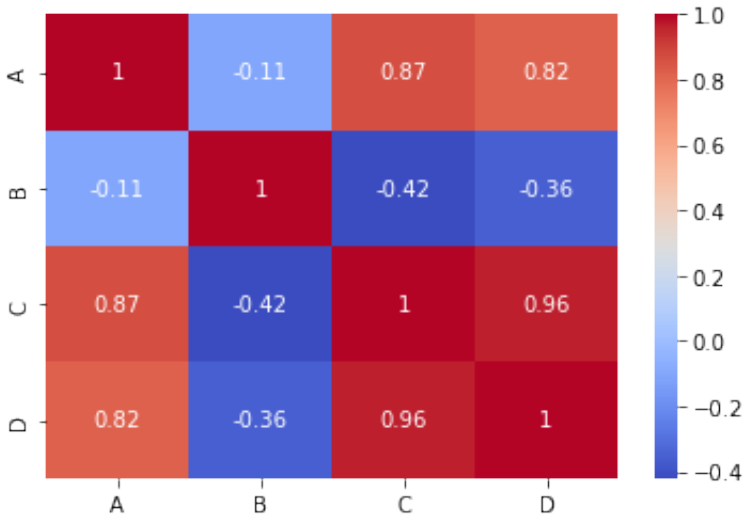


In [13]:
```
ax5 = sns.relplot(data=X_pca, x="PCA_0", y="PCA_3",hue=compounds['type
```



In [14]:
```
sns.heatmap(features_df.corr(), cmap='coolwarm', annot=True)
```

Out[14]: <AxesSubplot:>



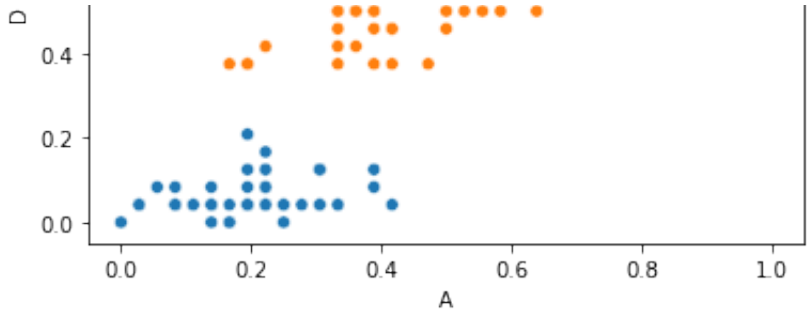
Based on the heatmap, we can see that there is a strong positive correlation between reagents (A, C), (A, D), as well as between reagents (C, D). We can also observe some negative correlation between the reagents, like (B, C). Based on the relplots, we can see that the compounds of type ether and amide are more similar to each other compared to phenol compounds in terms of their feature values.

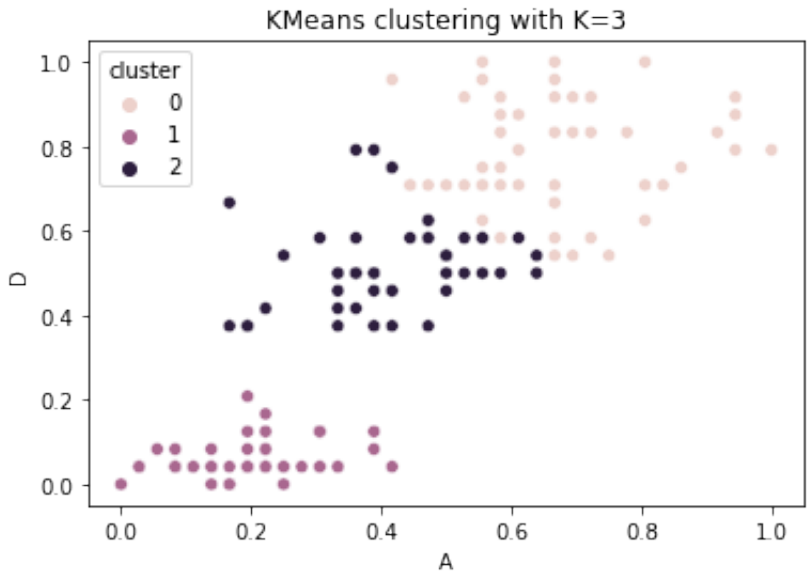
b:

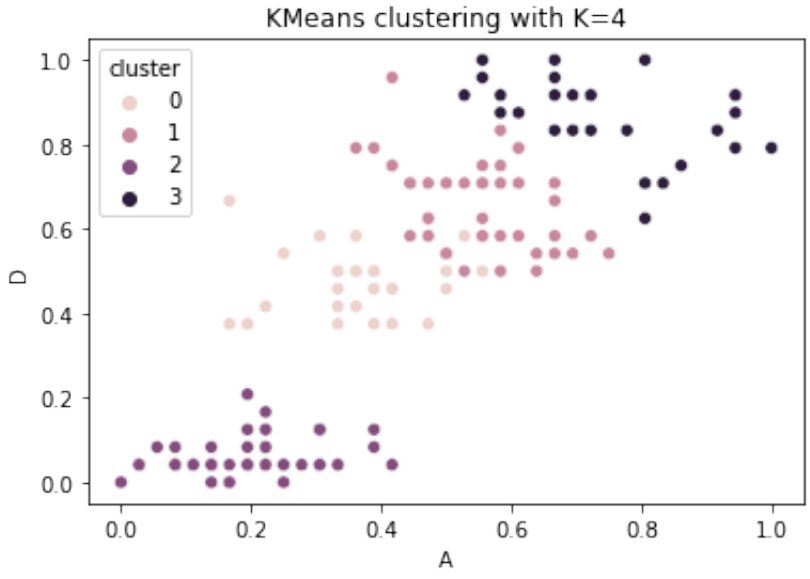
In [15]:
```
import warnings

class KMeans():
    def __init__(self, K, maximum_iters=100):
        # K: number of clusters to be created
        # distance matrix is Eucledian distance
        self.K = K
        self.maximum_iters = maximum_iters

    def cluster(self, input_points):
        """ Do KMeans clustering
        input_points: np.array shape(ndata,nfeatures).
            Each feature is assumed to be normalized within range of [
        """
        centroids = np.random.random((self.K, input_points.shape[1]))
        assignments = np.zeros_like(input_points.shape[0])
        new_assignments = self.create_new_assignments(centroids, input

        # restart if run into bad initialization
```

```
        # Comment out this part for Q1.(a)
        if len(np.unique(new_assignments))<self.K:
            return self.cluster(input_points)

        n_iters = 1
        while (new_assignments != assignments).any() and n_iters < sel
            ### Compute the centroid given new assignment ###
            centroids = np.array([input_points[new_assignments == k].m
            assignments = new_assignments
            ### Update the assignment with current centroids ###
            new_assignments = self.create_new_assignments(centroids, i
            if len(np.unique(new_assignments))< self.K:
                warnings.warn('At least one centroid vanishes')
            n_iters += 1
            if n_iters == self.maximum_iters:
                print("Warning: Maximum number of iterations reached!"

        return new_assignments


    def create_new_assignments(self, centroids, data_points):
        """ Assign each datapoint to its nearest centroid.
        centroid: 2d array of the current centroid for each cluster
        data_points: 2d arrays recording the features of each data poi
        """
        ###Compute the distances that stores the Eucledian distances b
        #shape (ndata,ncentroid)
        distances = np.linalg.norm(data_points[:, np.newaxis, :] - cen
        new_assignments = np.argmin(distances, axis=-1)
        return new_assignments
```

In [16]:
```
# Perform KMeans clustering with K=2,3,4
k_values = [2, 3, 4]
for k in k_values:
    model = KMeans(k)
    clusters = model.cluster(features_df.values)
    compounds['cluster'] = clusters

    # Visualize clusters using two features
    sns.scatterplot(data=compounds, x='A', y='D', hue='cluster')
    plt.title(f"KMeans clustering with K={k}")
    plt.show()
```

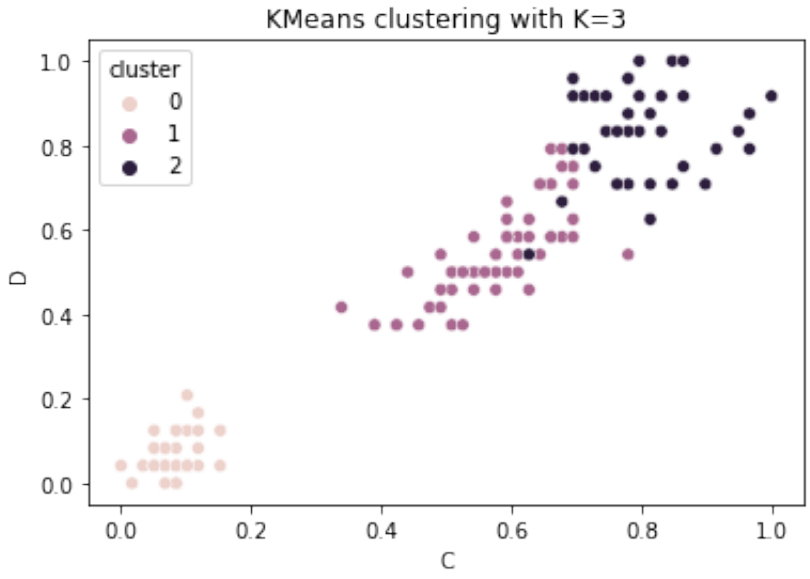

D
A
0.0
0.2
0.4
0.6
0.8
1.0

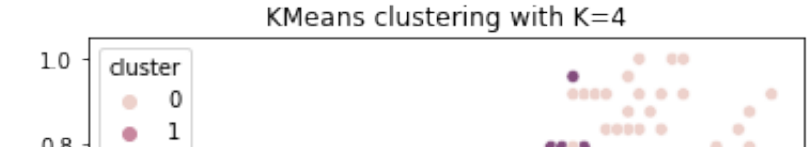

KMeans clustering with K=3
cluster
0
1
2
D
A

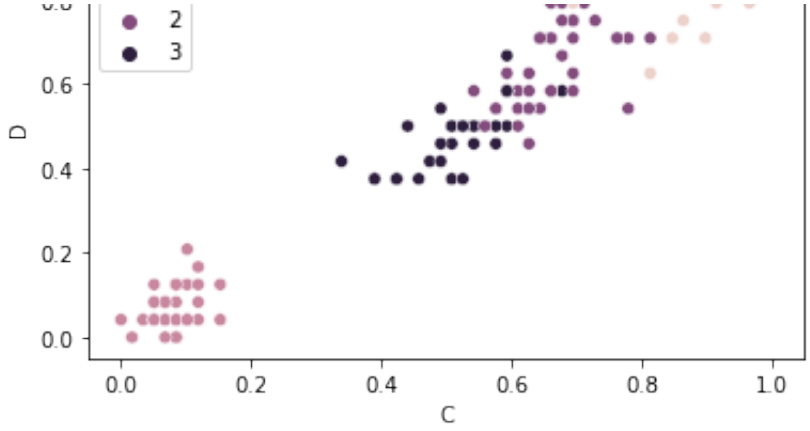

KMeans clustering with K=4
cluster
0
1
2
3
D
A

In [17]:

```
# Perform KMeans clustering with K=2,3,4
k_values = [2, 3, 4]
for k in k_values:
    model = KMeans(k)
    clusters = model.cluster(features_df.values)
    compounds['cluster'] = clusters

    # Visualize clusters using two features
    sns.scatterplot(data=compounds, x='C', y='D', hue='cluster')
    plt.title(f"KMeans clustering with K={k}")
    plt.show()
```







Based on my observation of reagents A and D, a K value of 3 would make the most sense. While a K value of 2 was also effective, the large group with values ranging from 0.4 to 1.0 could be further clustered with a K value of 3, resulting in a clearer boundary between clusters 0 and 2.

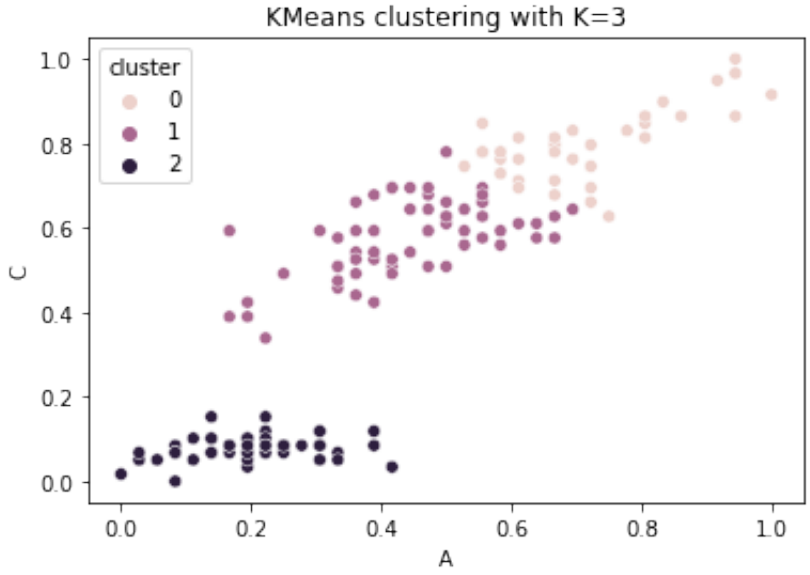
On the other hand, for reagents C and D, a K value of 2 would make the most sense. When the K value was increased to 3, the boundary between cluster 0 and 2 became less clear.

c:

In [18]:
```
def validate(y_hat,y):
    """print accuracy of prediction for each class for the compounds c
    yhat: np.array shape(ndata). Your prediction of classes
    y: np.array of str shape(ndata). data labels / groudn truths.
    """
    # correct classification
    compounds = np.unique(y) # should be ['amide','phenol','ether'] fo
    clusters =[np.where((y==c)) for c in compounds]
    pred_class = np.unique(y_hat)

    #remove -1 for noise point in DBSCAN
    pred_class= np.delete(pred_class,np.where(pred_class==-1))
    assert len(pred_class) == len(compounds), f'y_hat has less or more

    for i in range(3):
        #loop over solutions
        counts=[]
        scores=[]
        for j in range(3):
            #loop over clusters of true assignments
            sol_i= np.where((y_hat==pred_class[i]))
            counts.append(len(np.intersect1d(sol_i, clusters[j])))
            scores.append(counts[-1]/len(clusters[j]))
        idx = np.argmax(scores)
        print(f'Class {pred_class[i]} - {compounds[idx]}: {counts[idx]
of {np.count_nonzero(clusters[idx])} are classified correctly')
```

Using feature A and C, highest correlation based on the heatmap of Q(a). Setting k = 3.

In [19]:
```
k = 3
model = KMeans(k)
clusters = model.cluster(features_df.values)
compounds['cluster'] = clusters

# Visualize clusters using two features
sns.scatterplot(data=compounds, x='A', y='C', hue='cluster')
plt.title(f"KMeans clustering with K={k}")
plt.show()
```

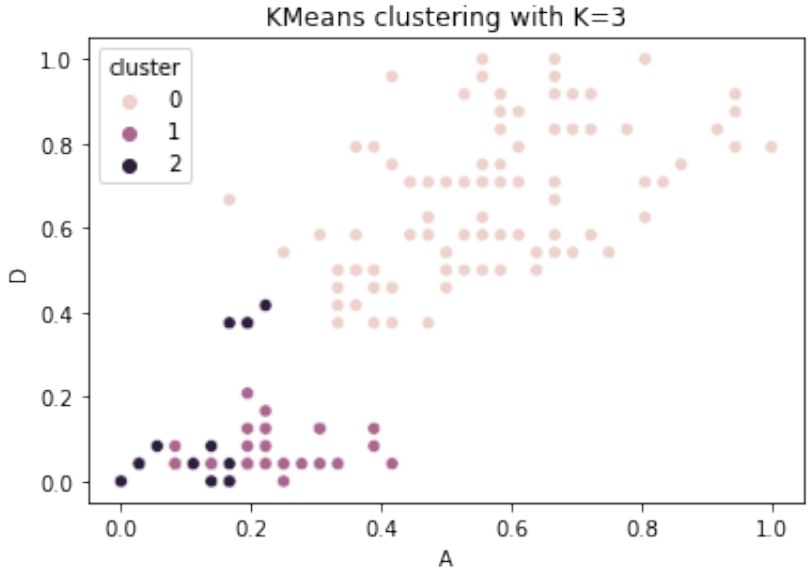


In [20]:
```
validate(clusters, compounds['type'])
```

```
Class 0 - ether: 36 out of 50 are classified correctly
Class 1 - amide: 47 out of 49 are classified correctly
Class 2 - phenol: 50 out of 50 are classified correctly
```

Let's try clustering based on A and D.

In [21]:
```
k = 3
model = KMeans(k)
clusters = model.cluster(features_df.values)
compounds['cluster'] = clusters

# Visualize clusters using two features
sns.scatterplot(data=compounds, x='A', y='D', hue='cluster')
plt.title(f"KMeans clustering with K={k}")
plt.show()
```



In [22]:
```
validate(clusters, compounds['type'])
```

```
Class 0 - ether: 50 out of 50 are classified correctly
Class 1 - phenol: 32 out of 50 are classified correctly
Class 2 - phenol: 18 out of 50 are classified correctly
```

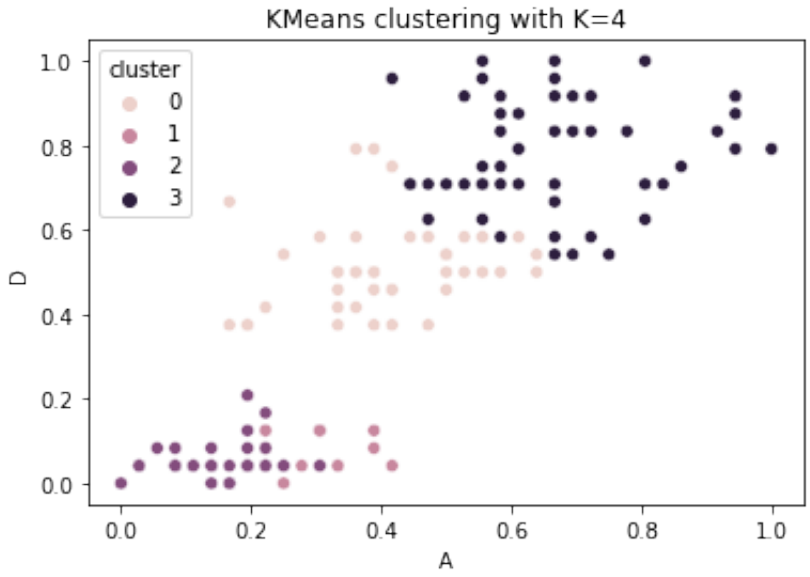
The classification achieved by the K-Means method is relatively good. The method is able to classify 80% of ether and amide correctly, which are similar to each other. Additionally, phenol can be classified with 100% accuracy. The method is also fast, and the results are good enough for qualitative analysis.

d:

In [23]:
```
import warnings

class KMeans_sp():
    def __init__(self, K, maximum_iters=100):
        # K: number of clusters to be created
        # distance matrix is Eucledian distance
        self.K = K
```

```
        self.K = K
        self.maximum_iters = maximum_iters

    def cluster(self, input_points):
        """ Do KMeans clustering
        input_points: np.array shape(ndata,nfeatures).
            Each feature is assumed to be normalized within range of [
        """
        centroids = np.random.random((self.K, input_points.shape[1]))
        assignments = np.zeros_like(input_points.shape[0])
        new_assignments = self.create_new_assignments(centroids, input

        # restart if run into bad initialization
        # Comment out this part for Q1.(d)
#         if len(np.unique(new_assignments))<self.K:
#             return self.cluster(input_points)

        n_iters = 1
        while (new_assignments != assignments).any() and n_iters < sel
            ### Compute the centroid given new assignment ###
            centroids = np.array([input_points[new_assignments == k].m
            assignments = new_assignments
            ### Update the assignment with current centroids ###
            new_assignments = self.create_new_assignments(centroids, i
            if len(np.unique(new_assignments))< self.K:
                warnings.warn('At least one centroid vanishes')
            n_iters += 1
            if n_iters == self.maximum_iters:
                print("Warning: Maximum number of iterations reached!"

        return new_assignments


    def create_new_assignments(self, centroids, data_points):
        """ Assign each datapoint to its nearest centroid.
        centroid: 2d array of the current centroid for each cluster
        data_points: 2d arrays recording the features of each data poi
        """
        ###Compute the distances that stores the Eucledian distances b
        #shape (ndata,ncentroid)
        distances = np.linalg.norm(data_points[:, np.newaxis, :] - cen
        new_assignments = np.argmin(distances, axis=-1)
        return new_assignments
```

In [24]:
```
k = 4
model = KMeans_sp(k)
clusters = model.cluster(features_df.values)
compounds['cluster'] = clusters

# Visualize clusters using two features
sns.scatterplot(data=compounds, x='A', y='D', hue='cluster')
plt.title(f"KMeans clustering with K={k}")
plt.show()
```



There is an error in the code (very possible). The KMeans algorithm fails to cluster with K = 4.

If the initial centroids are chosen poorly, for example, if they are located on the border-line between two clusters, two or more original clusters may be classified as a single cluster. This occurs because the algorithm is trapped in a local minimum. Therefore, it is essential to choose good initial centroids for the KMeans clustering.

Q2

a:

In [25]:
```
from sklearn.cluster import DBSCAN

db = DBSCAN(eps=0.2, min_samples=2)
clustering = db.fit(features_df)
# Cluster labels for each point in the dataset given to fit().  Noisy
clustering.labels_
# Indices of core samples.
clustering.core_sample_indices_
```

Out[25]:
```
array([  0,   1,   2,   3,   4,   5,   6,   7,   8,   9,  10,  11,  1
2,
        13,  14,  15,  16,  17,  18,  19,  20,  21,  22,  23,  24,  2
5,
        26,  27,  28,  29,  30,  31,  32,  33,  34,  35,  36,  37,  3
8,
        39,  40,  41,  42,  43,  44,  45,  46,  47,  48,  49,  51,  5
2,
        53,  54,  55,  56,  57,  58,  59,  60,  61,  62,  63,  64,  6
5,
        66,  67,  68,  69,  70,  71,  72,  73,  74,  75,  76,  77,  7
8,
        79,  80,  81,  82,  83,  84,  85,  86,  87,  88,  89,  90,  9
1,
        92,  93,  94,  95,  96,  97,  98,  99, 100, 101, 102, 103, 10
4,
       105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 11
7,
       118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 13
0,
       131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 14
3,
       144, 145, 146, 147, 148, 149])
```

In [26]:
```
n_clusters = len(set(clustering.labels_)) - (1 if -1 in clustering.lab
print(n_clusters)
```

```
3
```

Found that eps = 0.2 can have 3 clusters.

Further adjusting the model:

In [27]:
```
db = DBSCAN(eps=0.12, min_samples=5)
clustering = db.fit(features_df)
```

In [28]:
```
n_core = len(clustering.core_sample_indices_)
n_border = np.sum(clustering.labels_ != -1) - n_core
n_noise = np.sum(clustering.labels_ == -1)

print(f"Number of core points: {n_core}")
print(f"Number of border points: {n_border}")
print(f"Number of noise points: {n_noise}")
```

```
Number of core points: 85
Number of border points: 29
Number of noise points: 36
```

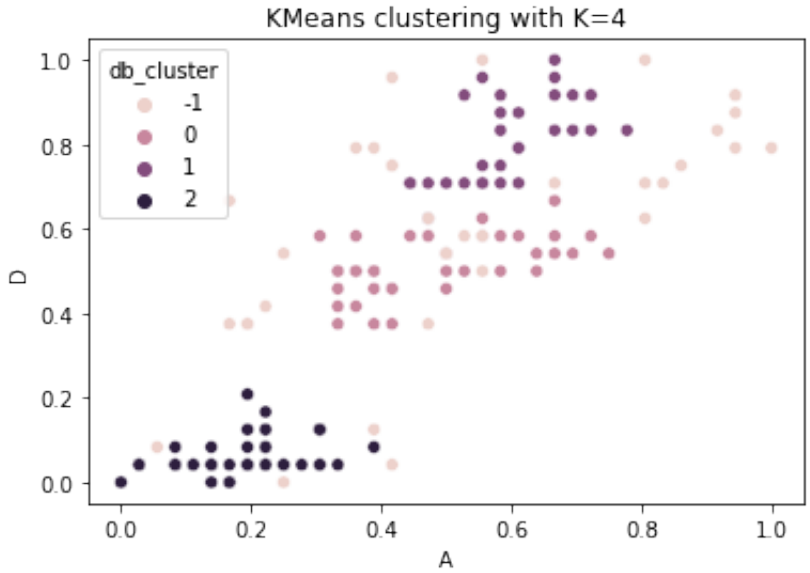
In [29]:
```
n_clusters = len(set(clustering.labels_)) - (1 if -1 in clustering.lab
print(n_clusters)
```

```
3
```

In [30]:
```
validate(clustering.labels_, compounds['type'])
```

```
Class 0 - amide: 38 out of 49 are classified correctly
Class 1 - ether: 28 out of 50 are classified correctly
Class 2 - phenol: 45 out of 50 are classified correctly
```

In [31]:
```
compounds['db_cluster'] = clustering.labels_

# Visualize clusters using two features
sns.scatterplot(data=compounds, x='A', y='D', hue='db_cluster')
plt.title(f"KMeans clustering with K={k}")
plt.show()
```



The initial setting of DBSCAN: eps=0.12, min_samples=5

Number of core points: 85

Number of border points: 29

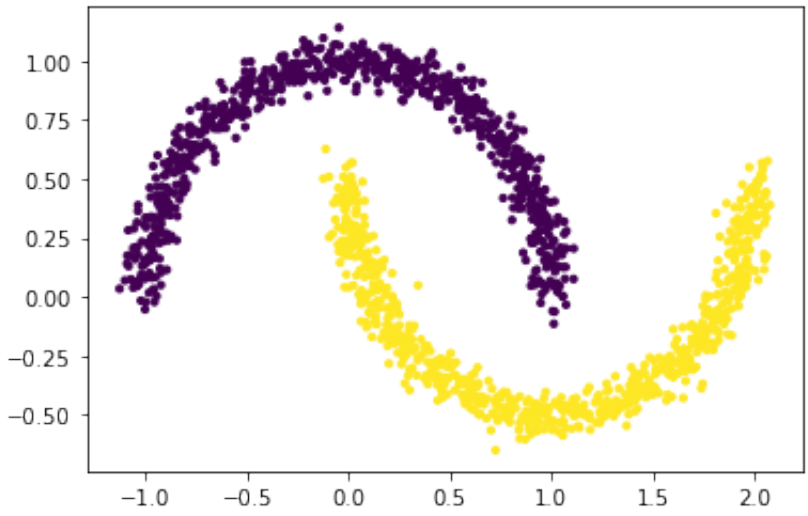
Number of noise points: 36

Compared to KMeans, DBSCAN is not more effective in this problem. When the eps value is too large, too few clusters are identified. On the other hand, if the eps value is good enough, like 0.12 in this case, many data points will be identified as noise points.

b:

In [32]:
```
from sklearn import cluster, datasets, mixture
from sklearn.preprocessing import StandardScaler
from itertools import cycle, islice
from pylab import *

np.random.seed(0)

# ============
# Generate datasets. We choose the size big enough to see the scalabil
# of the algorithms, but not too big to avoid too long running times
# ============
n_samples = 1500
noisy_circles = datasets.make_circles(n_samples=n_samples, factor=0.5,
noisy_moons = datasets.make_moons(n_samples=n_samples, noise=0.05)
# print(noisy_moons)
X,y=noisy_moons
plt.scatter(X[:, 0], X[:, 1], s=10,c=y)
```

Out[32]:
```
<matplotlib.collections.PathCollection at 0x7f95b0811910>
```



In [33]:
```
noisy_moons
```

Out[33]:
```
(array([[ 0.49627131, -0.34275349],
        [-0.16629956,  0.92234209],
        [ 0.71895601,  0.66529038],
        ...,
        [ 1.90950927,  0.02989686],
        [ 0.54623069, -0.36003133],
        [ 0.04090016,  0.37069297]]),
 array([1, 0, 0, ..., 1, 1, 1]))
```

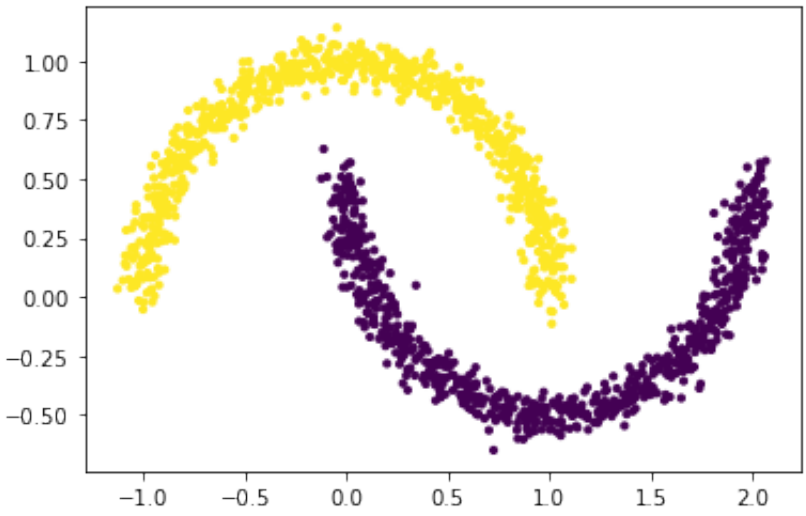
DBSCAN:

In [34]:
```
db = DBSCAN(eps=0.14, min_samples=5)
clustering = db.fit(X)

n_core = len(clustering.core_sample_indices_)
n_border = np.sum(clustering.labels_ != -1) - n_core
n_noise = np.sum(clustering.labels_ == -1)
n_clusters = len(set(clustering.labels_)) - (1 if -1 in clustering.lab

print(f"Number of core points: {n_core}")
print(f"Number of border points: {n_border}")
print(f"Number of noise points: {n_noise}")
print(f"Number of clusters: {n_clusters}")
```

```
Number of core points: 1500
Number of border points: 0
Number of noise points: 0
Number of clusters: 2
```

In [35]:
```
plt.scatter(X[:, 0], X[:, 1], s=10,c=clustering.labels_)
```

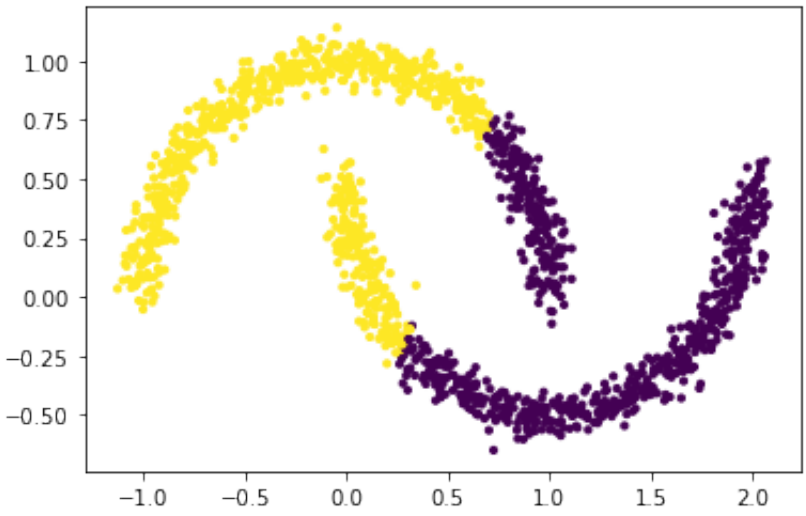
Out[35]:
```
<matplotlib.collections.PathCollection at 0x7f95b0f19ac0>
```



KMEANS:

In [36]:
```
model = KMeans(2)
clusters = model.cluster(X)

# Visualize clusters using two features
plt.scatter(X[:, 0], X[:, 1], s=10,c=clusters)
```

Out[36]: <matplotlib.collections.PathCollection at 0x7f95b0f53c70>



This time, DBSCAN performed better than KMeans. It highlights one of DBSCAN's important pros - it performs well with arbitrary shaped clusters.