# Chem277B: Machine Learning Algorithms

## Homework assignment #6: Clustering

In [168…
```python
import numpy as np
import pandas as pd
import math
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.model_selection import train_test_split, KFold
import warnings
from sklearn.cluster import DBSCAN
from sklearn import cluster, datasets, mixture
from sklearn.preprocessing import StandardScaler
from itertools import cycle, islice
from pylab import *
import seaborn as sns

np.random.seed(0)
sns.set()
```

### 1. KMeans.

(a) By normalizing the data and visualizing their correlation, I noticed that all three types are correlated by features `C` and `D`. In addition, amides and ethers are correlated by features `A` and `C`, but phenols are not correlated by these two features.

In [169…
```python
compounds = pd.read_csv('compounds.csv')
compounds.head()
```

Out[169]:

| | 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 |
| **3** | 5.8 | 2.8 | 5.1 | 2.4 | ether |
| **4** | 6.4 | 3.2 | 5.3 | 2.3 | ether |

```
In [170…  compounds.loc[:,'A':'D'].max(axis=0)
```

```
Out[170]: A    7.9
          B    4.4
          C    6.9
          D    2.5
          dtype: float64
```

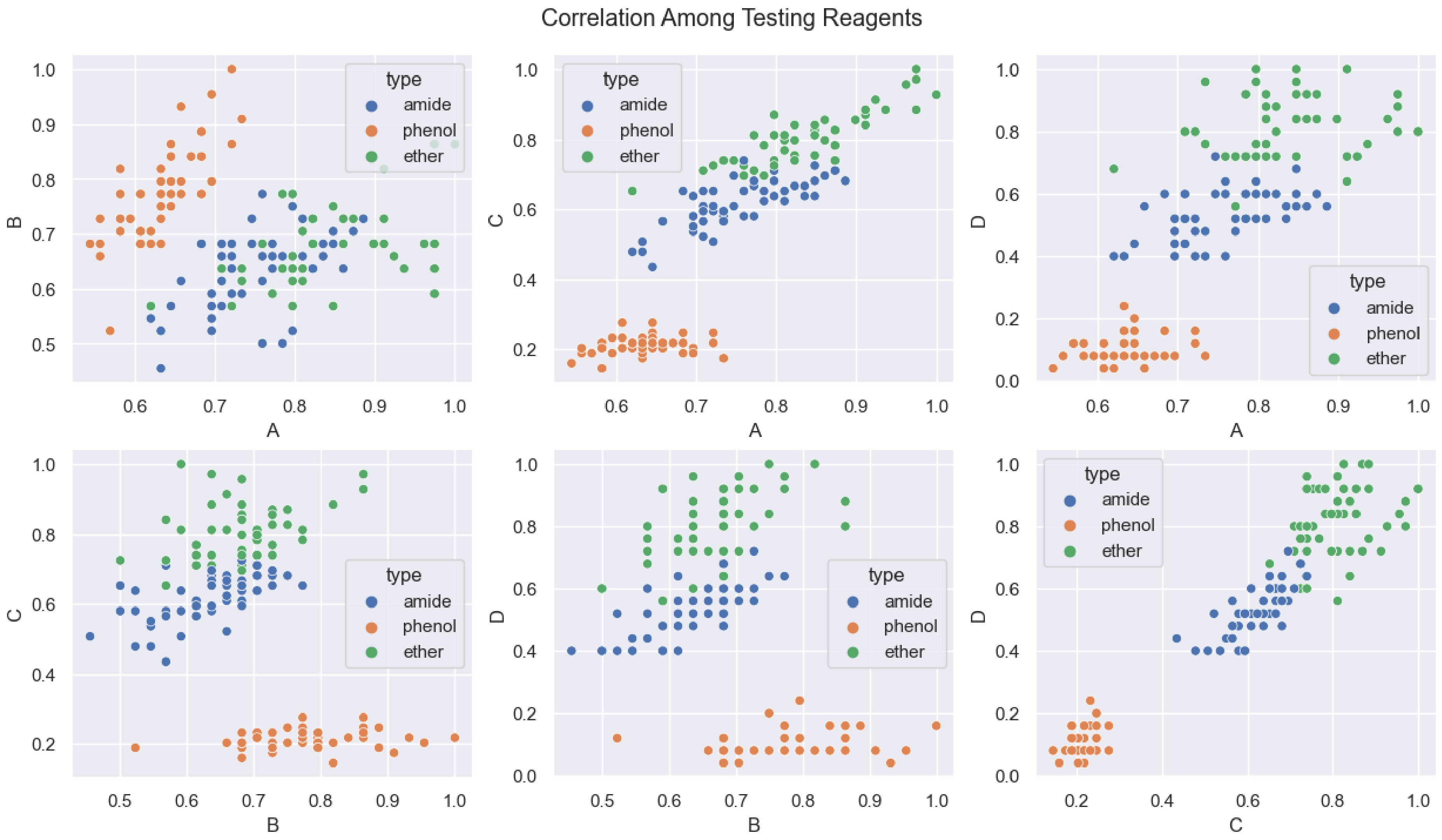
```
In [171…  compounds.loc[:,'A':'D'] = compounds.loc[:,'A':'D'] / compounds.loc[:,'A':'D'].max(axis=0)
          compounds.head()
```

Out[171]:

| | A | B | C | D | type |
|---|---|---|---|---|---|
| **0** | 0.810127 | 0.659091 | 0.623188 | 0.52 | amide |
| **1** | 0.721519 | 1.000000 | 0.217391 | 0.16 | phenol |
| **2** | 0.848101 | 0.681818 | 0.753623 | 0.92 | ether |
| **3** | 0.734177 | 0.636364 | 0.739130 | 0.96 | ether |
| **4** | 0.810127 | 0.727273 | 0.768116 | 0.92 | ether |

```
In [172…  fig, axes = plt.subplots(2, 3, figsize=(15,8))
          fig.suptitle('Correlation Among Testing Reagents', y=0.93)
          sns.scatterplot(ax=axes[0,0], data = compounds, x='A', y='B', hue='type')
          sns.scatterplot(ax=axes[0,1], data = compounds, x='A', y='C', hue='type')
          sns.scatterplot(ax=axes[0,2], data = compounds, x='A', y='D', hue='type')
          sns.scatterplot(ax=axes[1,0], data = compounds, x='B', y='C', hue='type')
          sns.scatterplot(ax=axes[1,1], data = compounds, x='B', y='D', hue='type')
          sns.scatterplot(ax=axes[1,2], data = compounds, x='C', y='D', hue='type')
```

```
Out[172]: <AxesSubplot: xlabel='C', ylabel='D'>
```



(b) I have finished the KMeans clustering codes as shown below. I also chose K = 2, 3, 4 and generated 3 new columns in the compounds dataframe. After visualizing the codes, I noticed that K = 2 is actually the best clustering because each cluster is very distant apart from the other in all the correlation graphs.

In [173…

```
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 [0,1]
        """
        centroids = np.random.random((self.K, input_points.shape[1]))
        assignments = np.zeros(input_points.shape[0], dtype = np.int32)
        new_assignments = self.create_new_assignments(centroids, input_points)

        # 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 < self.maximum_iters:
            ### Compute the centroid given new assignment ###
            centroids = np.array([input_points[new_assignments == k].mean(axis=0) for \
                                     k in range(self.K)])
            assignments = new_assignments
            ### Update the assignment with current centroids ###
            new_assignments = self.create_new_assignments(centroids, input_points)
            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 point.
        """
        ###Compute the distances that stores the Eucledian distances between each datapoints and the centroid ###
        #shape (ndata,ncentroid)
        distances = np.sqrt(((data_points[:, np.newaxis, :] - centroids)**2).sum(axis = -1))
        new_assignments = np.argmin(distances, axis=-1)
        return new_assignments
```

In [174…

```
compounds_ndarray = np.array(compounds.loc[:,'A':'D'])
KMeans_2 = KMeans(2, maximum_iters=100)
compounds['type_2'] = KMeans_2.cluster(compounds_ndarray)
KMeans_3 = KMeans(3, maximum_iters=100)
compounds['type_3'] = KMeans_3.cluster(compounds_ndarray)
KMeans_4 = KMeans(4, maximum_iters=100)
compounds['type_4'] = KMeans_4.cluster(compounds_ndarray)
compounds.head()
```

Out[174]:

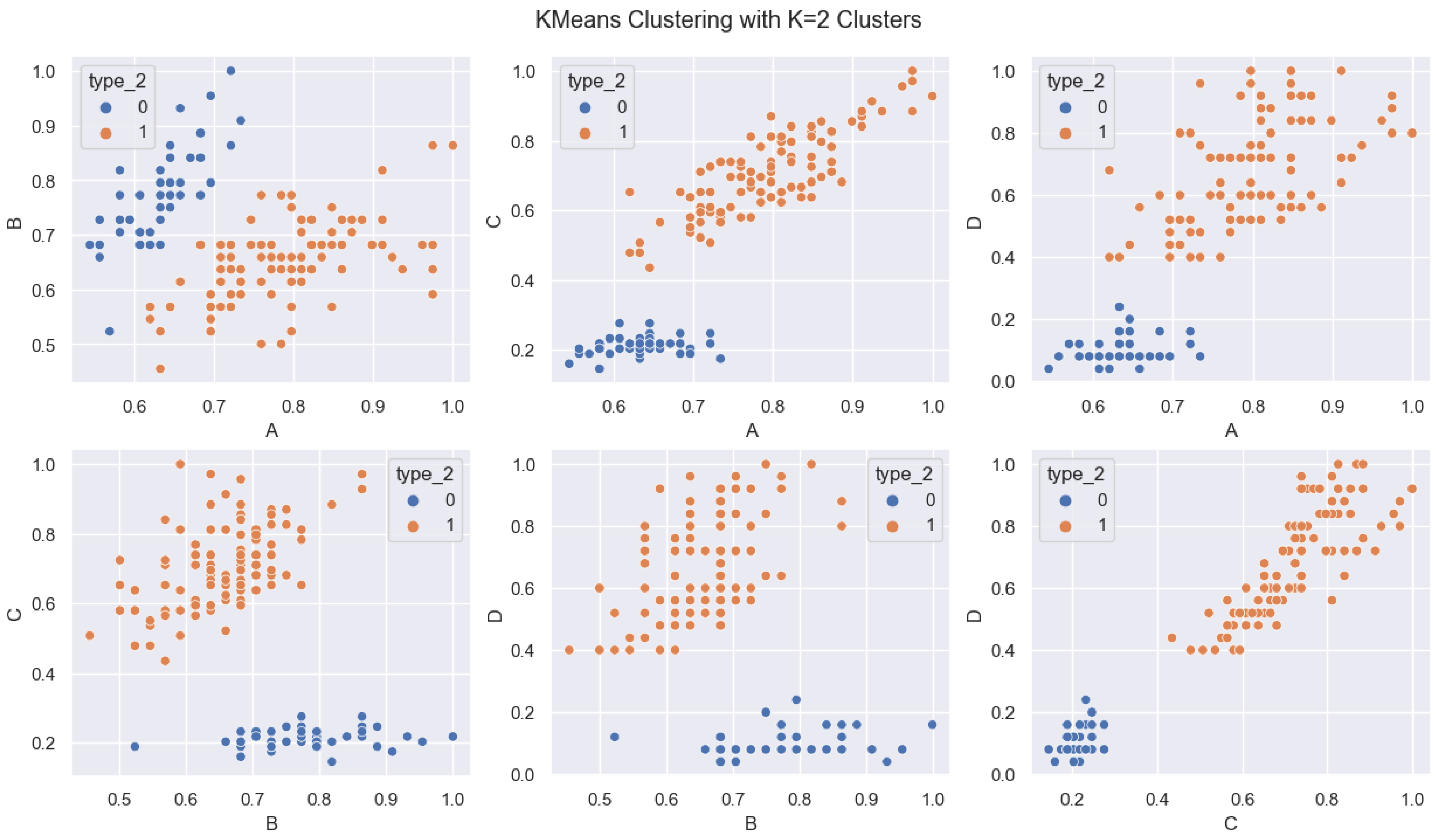
| | A | B | C | D | type | type_2 | type_3 | type_4 |
|---|---|---|---|---|---|---|---|---|
| **0** | 0.810127 | 0.659091 | 0.623188 | 0.52 | amide | 1 | 2 | 3 |
| **1** | 0.721519 | 1.000000 | 0.217391 | 0.16 | phenol | 0 | 0 | 0 |
| **2** | 0.848101 | 0.681818 | 0.753623 | 0.92 | ether | 1 | 1 | 2 |
| **3** | 0.734177 | 0.636364 | 0.739130 | 0.96 | ether | 1 | 1 | 2 |
| **4** | 0.810127 | 0.727273 | 0.768116 | 0.92 | ether | 1 | 1 | 2 |

In [175…

```
fig, axes = plt.subplots(2, 3, figsize=(15,8))
fig.suptitle('KMeans Clustering with K=2 Clusters', y=0.93)
sns.scatterplot(ax=axes[0,0], data = compounds, x='A', y='B', hue='type_2')
sns.scatterplot(ax=axes[0,1], data = compounds, x='A', y='C', hue='type_2')
sns.scatterplot(ax=axes[0,2], data = compounds, x='A', y='D', hue='type_2')
sns.scatterplot(ax=axes[1,0], data = compounds, x='B', y='C', hue='type_2')
sns.scatterplot(ax=axes[1,1], data = compounds, x='B', y='D', hue='type_2')
sns.scatterplot(ax=axes[1,2], data = compounds, x='C', y='D', hue='type_2')
```

Out[175]:

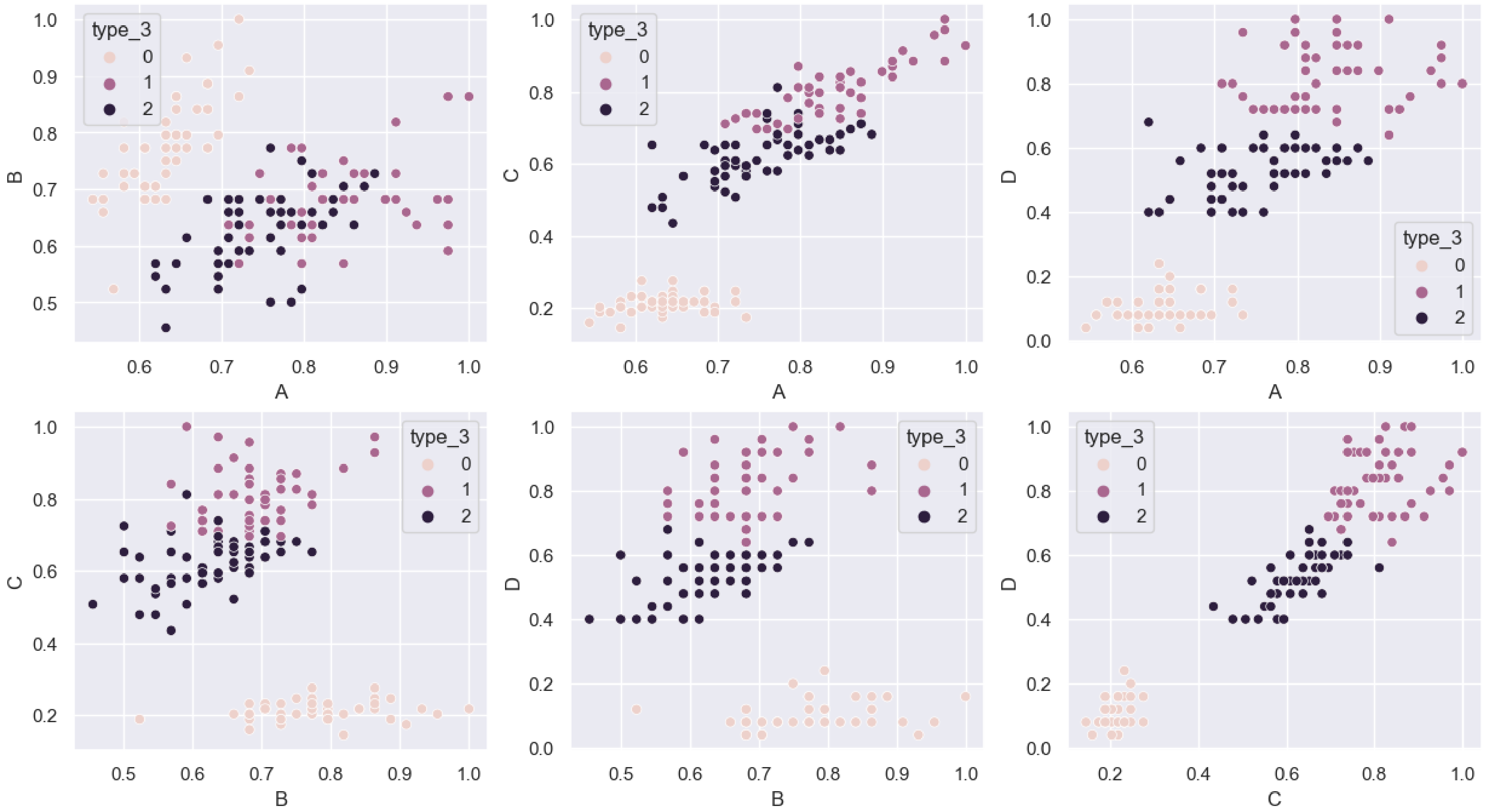
```
<AxesSubplot: xlabel='C', ylabel='D'>
```



```
In [176…  fig, axes = plt.subplots(2, 3, figsize=(15,8))
          fig.suptitle('KMeans Clustering with K=3 Clusters', y=0.93)
          sns.scatterplot(ax=axes[0,0], data = compounds, x='A', y='B', hue='type_3')
          sns.scatterplot(ax=axes[0,1], data = compounds, x='A', y='C', hue='type_3')
          sns.scatterplot(ax=axes[0,2], data = compounds, x='A', y='D', hue='type_3')
          sns.scatterplot(ax=axes[1,0], data = compounds, x='B', y='C', hue='type_3')
          sns.scatterplot(ax=axes[1,1], data = compounds, x='B', y='D', hue='type_3')
          sns.scatterplot(ax=axes[1,2], data = compounds, x='C', y='D', hue='type_3')
```

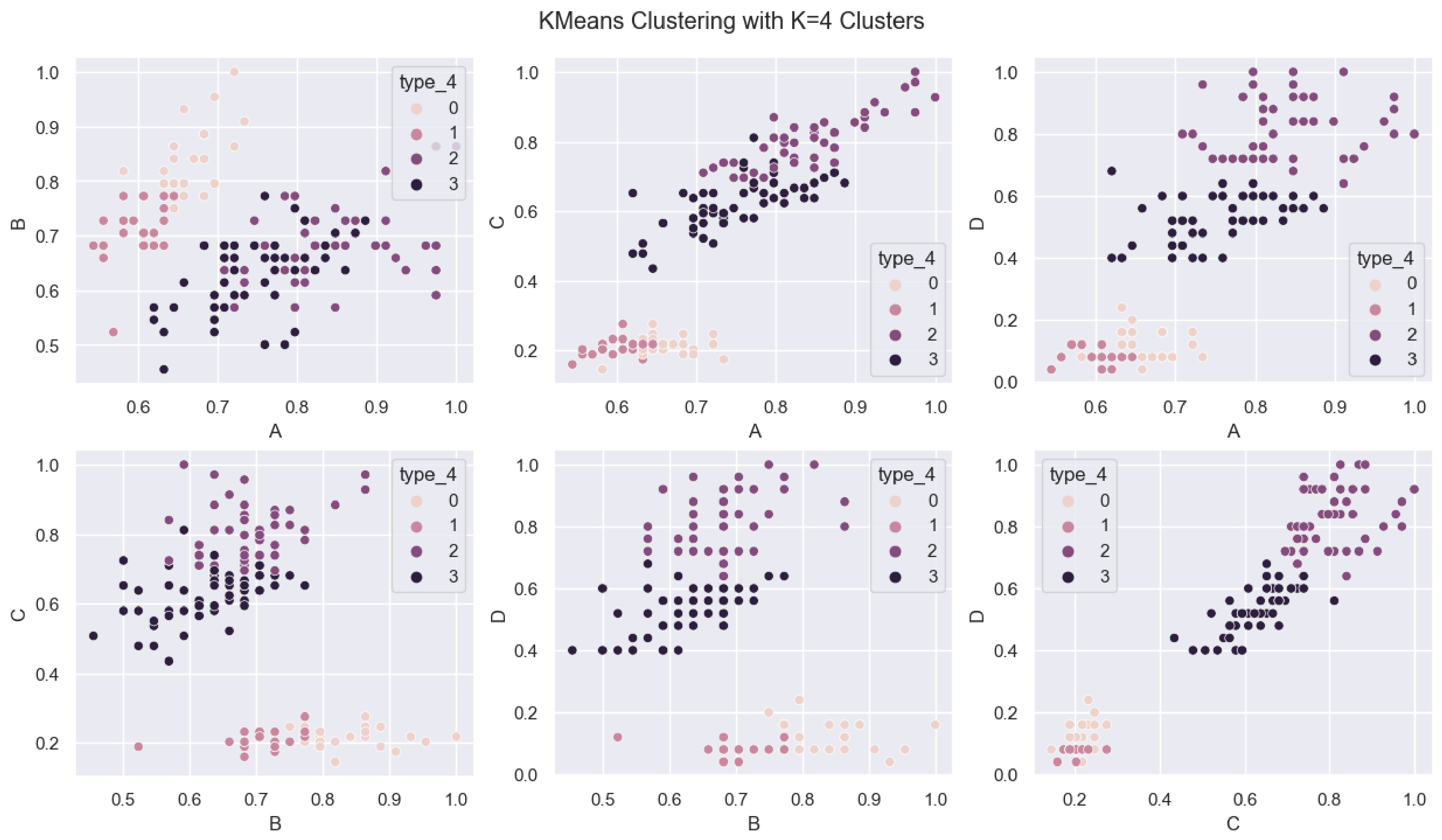
```
Out[176]: <AxesSubplot: xlabel='C', ylabel='D'>
```

KMeans Clustering with K=3 Clusters



```
In [177…  fig, axes = plt.subplots(2, 3, figsize=(15,8))
          fig.suptitle('KMeans Clustering with K=4 Clusters', y=0.93)
          sns.scatterplot(ax=axes[0,0], data = compounds, x='A', y='B', hue='type_4')
          sns.scatterplot(ax=axes[0,1], data = compounds, x='A', y='C', hue='type_4')
          sns.scatterplot(ax=axes[0,2], data = compounds, x='A', y='D', hue='type_4')
          sns.scatterplot(ax=axes[1,0], data = compounds, x='B', y='C', hue='type_4')
          sns.scatterplot(ax=axes[1,1], data = compounds, x='B', y='D', hue='type_4')
          sns.scatterplot(ax=axes[1,2], data = compounds, x='C', y='D', hue='type_4')

Out[177]: <AxesSubplot: xlabel='C', ylabel='D'>
```



(c) I ran the provided validation function on the K=3 clustering and the true label. The clustering is pretty close to the original labeling, giving the following results:

Class 0 - ether: 46 out of 50 are classified correctly

Class 1 - phenol: 50 out of 50 are classified correctly

Class 2 - amide: 48 out of 49 are classified correctly

When I visualized the results using Seaborn, I think the KMeans clustering is actually better than the true labeling, especially for correlations between `A` & `D`, `B` & `D` and `C` & `D`.

```
In [178…  def validate(y_hat,y):
              """print accuracy of prediction for each class for the compounds dataset
              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'] for compounds dataset
              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 than {len(compounds)} classes:{pred_class}'

              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]} out \
          of {np.count_nonzero(clusters[idx])} are classified correctly')
```

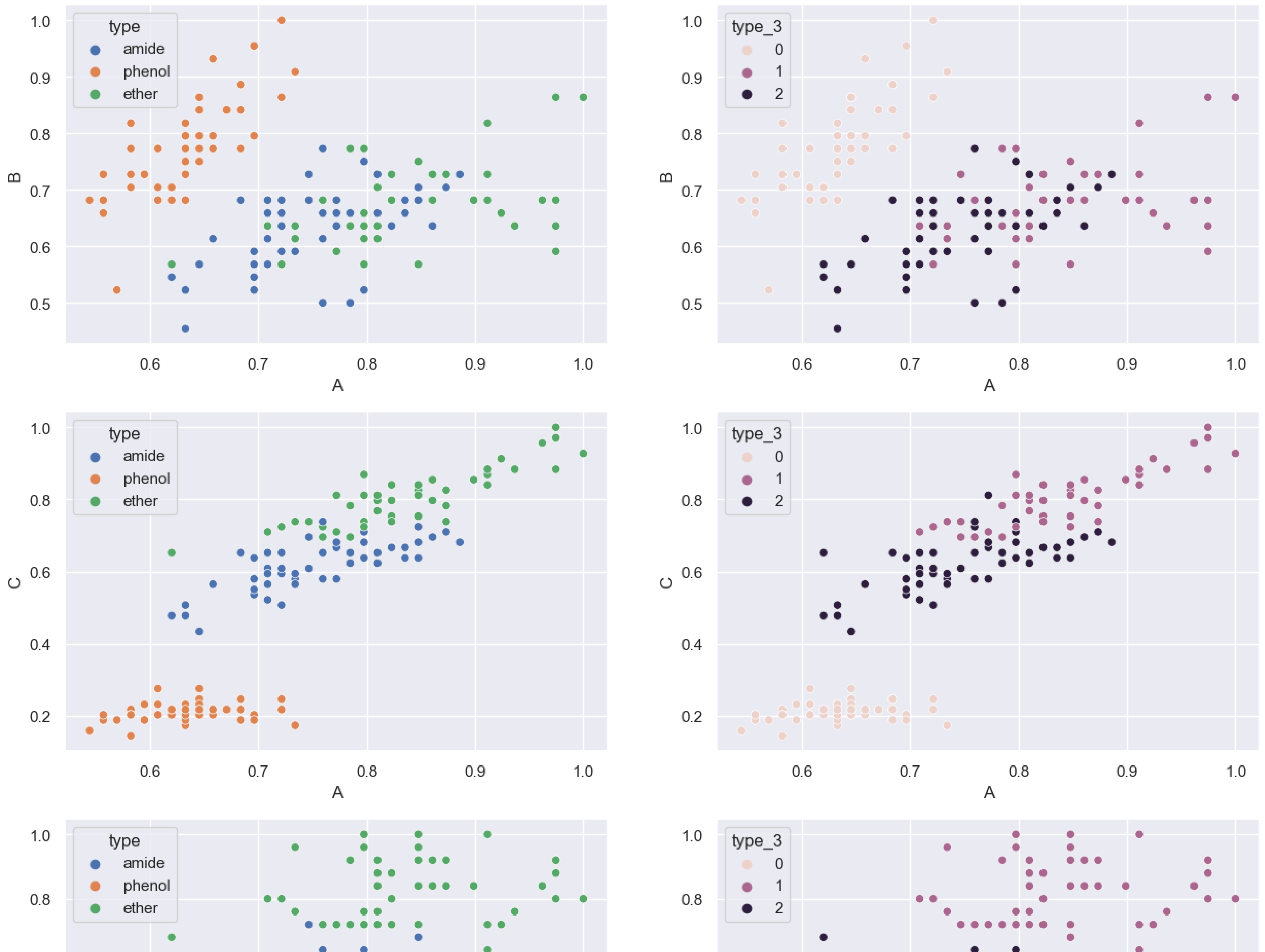
```
In [179…  validate(compounds['type_3'], compounds['type'])
```

```
Class 0 - phenol: 50 out of 50 are classified correctly
Class 1 - ether: 46 out of 50 are classified correctly
Class 2 - amide: 48 out of 49 are classified correctly
```

```
In [180…  fig, axes = plt.subplots(6, 2, figsize=(15,30))
          fig.suptitle('K=3 Cluster Comparison with True Label', y=0.93)
          sns.scatterplot(ax=axes[0,0], data = compounds, x='A', y='B', hue='type')
          sns.scatterplot(ax=axes[0,1], data = compounds, x='A', y='B', hue='type_3')
          sns.scatterplot(ax=axes[1,0], data = compounds, x='A', y='C', hue='type')
          sns.scatterplot(ax=axes[1,1], data = compounds, x='A', y='C', hue='type_3')
          sns.scatterplot(ax=axes[2,0], data = compounds, x='A', y='D', hue='type')
          sns.scatterplot(ax=axes[2,1], data = compounds, x='A', y='D', hue='type_3')
          sns.scatterplot(ax=axes[3,0], data = compounds, x='B', y='C', hue='type')
          sns.scatterplot(ax=axes[3,1], data = compounds, x='B', y='C', hue='type_3')
          sns.scatterplot(ax=axes[4,0], data = compounds, x='B', y='D', hue='type')
          sns.scatterplot(ax=axes[4,1], data = compounds, x='B', y='D', hue='type_3')
          sns.scatterplot(ax=axes[5,0], data = compounds, x='C', y='D', hue='type')
          sns.scatterplot(ax=axes[5,1], data = compounds, x='C', y='D', hue='type_3')
```

```
Out[180]: <AxesSubplot: xlabel='C', ylabel='D'>
```

K=3 Cluster Comparison with True Label



(d) I encountered bad initialization and received a lot of warning signs, including `Maximum number of iterations reached` and `warnings.warn('At least one centroid vanishes')`. I think it means that one or more of the initial K centroids are too far away from all data points, hence resulting in empty cluster(s). So in this case the algorithm just keeps running until it reaches the maximum number of iterations. But the faraway centroids will not be updated with new data points, which causes the centroids to remain in the initial distant positions.

Therefore, it is important to check whether all K clusters have at least one data point assigned to them. If not, we need to restart the clustering with a new set of randomly initialized centroids.

I found the following possible solutions: 1. KMeans++, which choose initial centroids uniformly from the dataset to make sure thei will be assigned data points. 2. Instead of K centroids, we choose 2K or 3K centroids and discard the empty centroids to keep K centroids with data points assigned.

In [181…

```
class KMeans_4():
    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 [0,1]
        """
        centroids = np.random.random((self.K, input_points.shape[1]))
        assignments = np.zeros(input_points.shape[0], dtype = np.int32)
        new_assignments = self.create_new_assignments(centroids, input_points)
        # 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 < self.maximum_iters:
            ### Compute the centroid given new assignment ###
            centroids = np.array([input_points[new_assignments == k].mean(axis=0) for \
                                  k in range(self.K)])
            assignments = new_assignments
            ### Update the assignment with current centroids ###
            new_assignments = self.create_new_assignments(centroids, input_points)
            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 point.
        """
        ###Compute the distances that stores the Eucledian distances between each datapoints and the centroid ###
        #shape (ndata,ncentroid)
        distances = np.sqrt(((data_points[:, np.newaxis, :] - centroids)**2).sum(axis = -1))
        new_assignments = np.argmin(distances, axis=-1)
        return new_assignments
```

In [182…

```
KMeans_4_1 = KMeans_4(4, maximum_iters=100)
compounds['type_4_1'] = KMeans_4_1.cluster(compounds_ndarray)
KMeans_4_2 = KMeans_4(4, maximum_iters=100)
compounds['type_4_2'] = KMeans_4_2.cluster(compounds_ndarray)
KMeans_4_3 = KMeans_4(4, maximum_iters=100)
compounds['type_4_3'] = KMeans_4_3.cluster(compounds_ndarray)
KMeans_4_4 = KMeans_4(4, maximum_iters=100)
compounds['type_4_4'] = KMeans_4_4.cluster(compounds_ndarray)
KMeans_4_5 = KMeans_4(4, maximum_iters=100)
compounds['type_4_5'] = KMeans_4_5.cluster(compounds_ndarray)
compounds.head()
```

```
Warning: Maximum number of iterations reached!
Warning: Maximum number of iterations reached!
Warning: Maximum number of iterations reached!
Warning: Maximum number of iterations reached!
Warning: Maximum number of iterations reached!
```

```
/var/folders/pk/syhjl001491bwlx124c6hv_h0000gn/T/ipykernel_17298/3366918875.py:25: RuntimeWarning: Mean of empty slice.
  centroids = np.array([input_points[new_assignments == k].mean(axis=0) for \
/opt/miniconda3/envs/qm-tools/lib/python3.10/site-packages/numpy/core/_methods.py:181: RuntimeWarning: invalid value en
countered in true_divide
  ret = um.true_divide(
/var/folders/pk/syhjl001491bwlx124c6hv_h0000gn/T/ipykernel_17298/3366918875.py:31: UserWarning: At least one centroid v
anishes
  warnings.warn('At least one centroid vanishes')
/var/folders/pk/syhjl001491bwlx124c6hv_h0000gn/T/ipykernel_17298/3366918875.py:25: RuntimeWarning: Mean of empty slice.
  centroids = np.array([input_points[new_assignments == k].mean(axis=0) for \
/opt/miniconda3/envs/qm-tools/lib/python3.10/site-packages/numpy/core/_methods.py:181: RuntimeWarning: invalid value en
countered in true_divide
  ret = um.true_divide(
/var/folders/pk/syhjl001491bwlx124c6hv_h0000gn/T/ipykernel_17298/3366918875.py:31: UserWarning: At least one centroid v
anishes
  warnings.warn('At least one centroid vanishes')
/var/folders/pk/syhjl001491bwlx124c6hv_h0000gn/T/ipykernel_17298/3366918875.py:25: RuntimeWarning: Mean of empty slice.
  centroids = np.array([input_points[new_assignments == k].mean(axis=0) for \
/opt/miniconda3/envs/qm-tools/lib/python3.10/site-packages/numpy/core/_methods.py:181: RuntimeWarning: invalid value en
countered in true_divide
  ret = um.true_divide(
/var/folders/pk/syhjl001491bwlx124c6hv_h0000gn/T/ipykernel_17298/3366918875.py:31: UserWarning: At least one centroid v
anishes
  warnings.warn('At least one centroid vanishes')
/var/folders/pk/syhjl001491bwlx124c6hv_h0000gn/T/ipykernel_17298/3366918875.py:25: RuntimeWarning: Mean of empty slice.
  centroids = np.array([input_points[new_assignments == k].mean(axis=0) for \
/opt/miniconda3/envs/qm-tools/lib/python3.10/site-packages/numpy/core/_methods.py:181: RuntimeWarning: invalid value en
countered in true_divide
  ret = um.true_divide(
/var/folders/pk/syhjl001491bwlx124c6hv_h0000gn/T/ipykernel_17298/3366918875.py:31: UserWarning: At least one centroid v
anishes
  warnings.warn('At least one centroid vanishes')
/var/folders/pk/syhjl001491bwlx124c6hv_h0000gn/T/ipykernel_17298/3366918875.py:25: RuntimeWarning: Mean of empty slice.
  centroids = np.array([input_points[new_assignments == k].mean(axis=0) for \
/opt/miniconda3/envs/qm-tools/lib/python3.10/site-packages/numpy/core/_methods.py:181: RuntimeWarning: invalid value en
countered in true_divide
  ret = um.true_divide(
/var/folders/pk/syhjl001491bwlx124c6hv_h0000gn/T/ipykernel_17298/3366918875.py:31: UserWarning: At least one centroid v
anishes
  warnings.warn('At least one centroid vanishes')
```

Out[182]:

| | A | B | C | D | type | type_2 | type_3 | type_4 | type_4_1 | type_4_2 | type_4_3 | type_4_4 | type_4_5 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 0.810127 | 0.659091 | 0.623188 | 0.52 | amide | 1 | 2 | 3 | 1 | 1 | 1 | 0 | 1 |
| **1** | 0.721519 | 1.000000 | 0.217391 | 0.16 | phenol | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 |
| **2** | 0.848101 | 0.681818 | 0.753623 | 0.92 | ether | 1 | 1 | 2 | 1 | 1 | 1 | 0 | 1 |
| **3** | 0.734177 | 0.636364 | 0.739130 | 0.96 | ether | 1 | 1 | 2 | 1 | 1 | 1 | 0 | 1 |
| **4** | 0.810127 | 0.727273 | 0.768116 | 0.92 | ether | 1 | 1 | 2 | 1 | 1 | 1 | 0 | 1 |

## 2. DBSCAN.

(a) I tried to adjust the eps and min_samples hyperparameters. Apparently there's no easy way to remove all noisy points. In the end with eps=0.9 and min_samples = 5, I acquired the following 3+1 clusters, with 3 clusters and 1 noisy cluster.

With the set hyperparameters, I have altogether 103 core points, 26 border points, and 21 noise points.

Compared to KMeans, DBSCAN seems less flexible and less effective. It results in a lot of noise points that can't be defined. With the compounds dataset, I prefer KMeans.

In [183…

```
compounds_DBSCAN = compounds.loc[:,'A':'D'].values
db = DBSCAN(eps=0.09, min_samples=5)
clustering = db.fit(compounds_DBSCAN)
# Cluster labels for each point in the dataset given to fit().  Noisy samples are given the label -1.
clustering.labels_
```

Out[183]:

```
array([ 0, -1,  1, -1,  1,  2, -1,  1,  0,  1,  0, -1, -1,  1,  2,  2,  1,
        1,  0,  2,  2,  1, -1,  1,  2,  1,  2,  2,  2,  1, -1,  2,  1,  2,
        0,  0,  2,  1,  0,  2,  0,  0, -1,  1,  2,  0,  1,  1,  0, -1, -1,
        0,  1, -1,  1,  0, -1,  0,  1,  2,  2,  0,  2,  0,  1,  2,  0,  2,
        0,  0, -1,  2,  0,  1,  0,  2,  1,  0,  0,  0,  0,  2,  2,  0,  1,
        0, -1,  0,  0,  0,  0, -1,  2,  2,  2,  2,  1,  1,  1,  2,  0,  0,
        0,  0,  0,  2,  2,  2,  2,  0, -1,  2,  2, -1,  0,  1,  2,  2,  2,
       -1,  2,  2,  2,  0, -1, -1,  1,  0,  1,  2,  1,  0,  1,  0,  1, -1,
        2,  0,  2,  0,  1,  2,  0,  2,  0,  2,  1,  0,  0,  0])
```

In [184…

```
# Indices of core samples.
clustering.core_sample_indices_
```

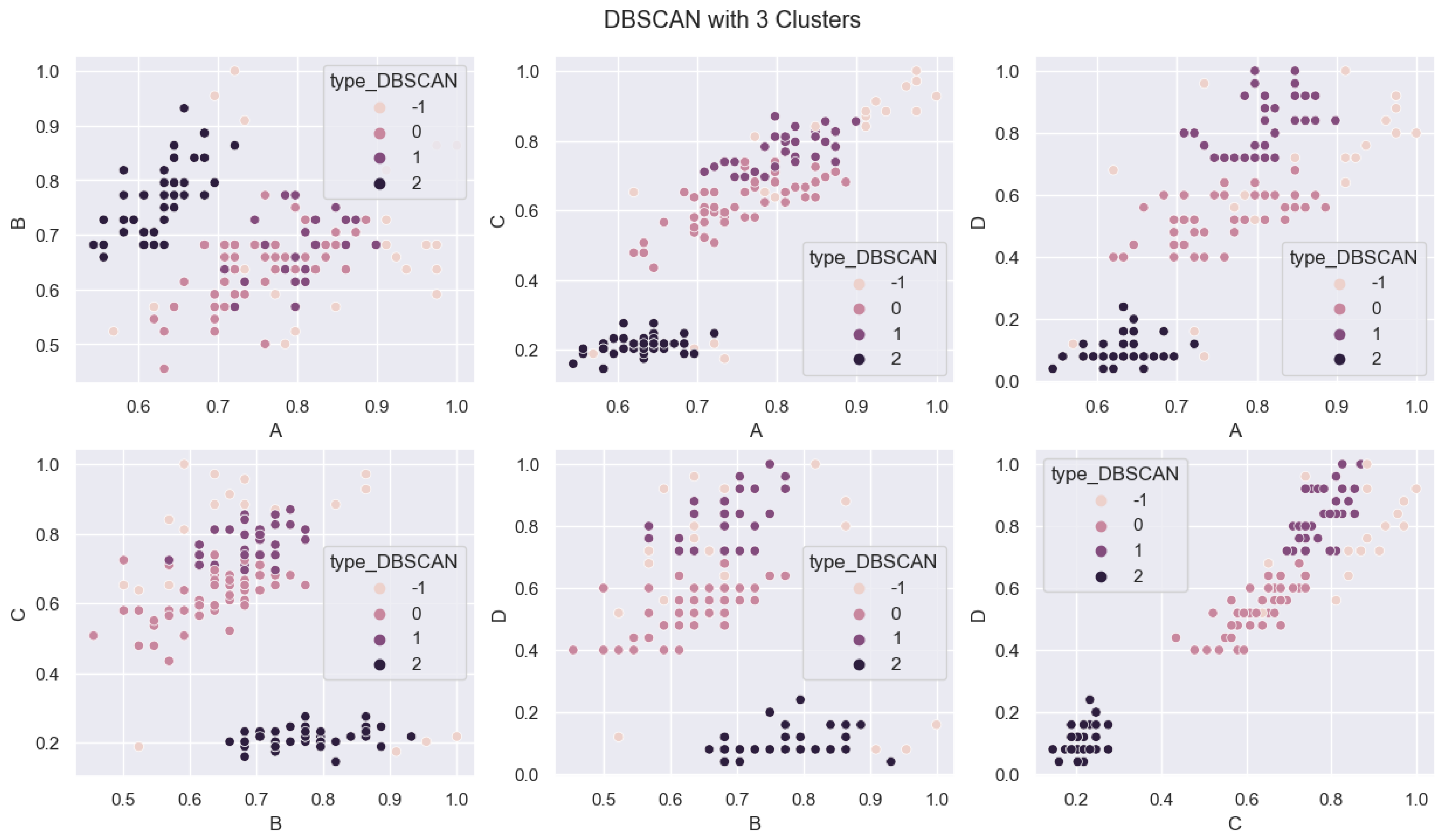
```
Out[184]: array([  0,   2,   4,   5,   7,   8,   9,  10,  13,  14,  15,  17,  18,
                  19,  20,  21,  24,  25,  26,  27,  28,  29,  31,  32,  36,  37,
                  38,  39,  40,  41,  43,  44,  45,  46,  47,  54,  55,  57,  58,
                  59,  60,  61,  62,  63,  65,  66,  67,  69,  71,  72,  73,  74,
                  75,  76,  77,  78,  79,  81,  82,  83,  84,  85,  89,  90,  92,
                  93,  99, 101, 103, 104, 105, 106, 107, 108, 109, 111, 112, 114,
                 115, 116, 117, 118, 120, 121, 122, 126, 128, 129, 130, 131, 132,
                 133, 134, 136, 137, 139, 141, 142, 143, 145, 146, 148, 149])
```

```
In [185…
# Number of core samples.
len(clustering.core_sample_indices_)
```

```
Out[185]: 103
```

```
In [186…
compounds['type_DBSCAN'] = clustering.labels_
fig, axes = plt.subplots(2, 3, figsize=(15,8))
fig.suptitle('DBSCAN with 3 Clusters', y=0.93)
sns.scatterplot(ax=axes[0,0], data = compounds, x='A', y='B', hue='type_DBSCAN')
sns.scatterplot(ax=axes[0,1], data = compounds, x='A', y='C', hue='type_DBSCAN')
sns.scatterplot(ax=axes[0,2], data = compounds, x='A', y='D', hue='type_DBSCAN')
sns.scatterplot(ax=axes[1,0], data = compounds, x='B', y='C', hue='type_DBSCAN')
sns.scatterplot(ax=axes[1,1], data = compounds, x='B', y='D', hue='type_DBSCAN')
sns.scatterplot(ax=axes[1,2], data = compounds, x='C', y='D', hue='type_DBSCAN')
```

```
Out[186]: <AxesSubplot: xlabel='C', ylabel='D'>
```

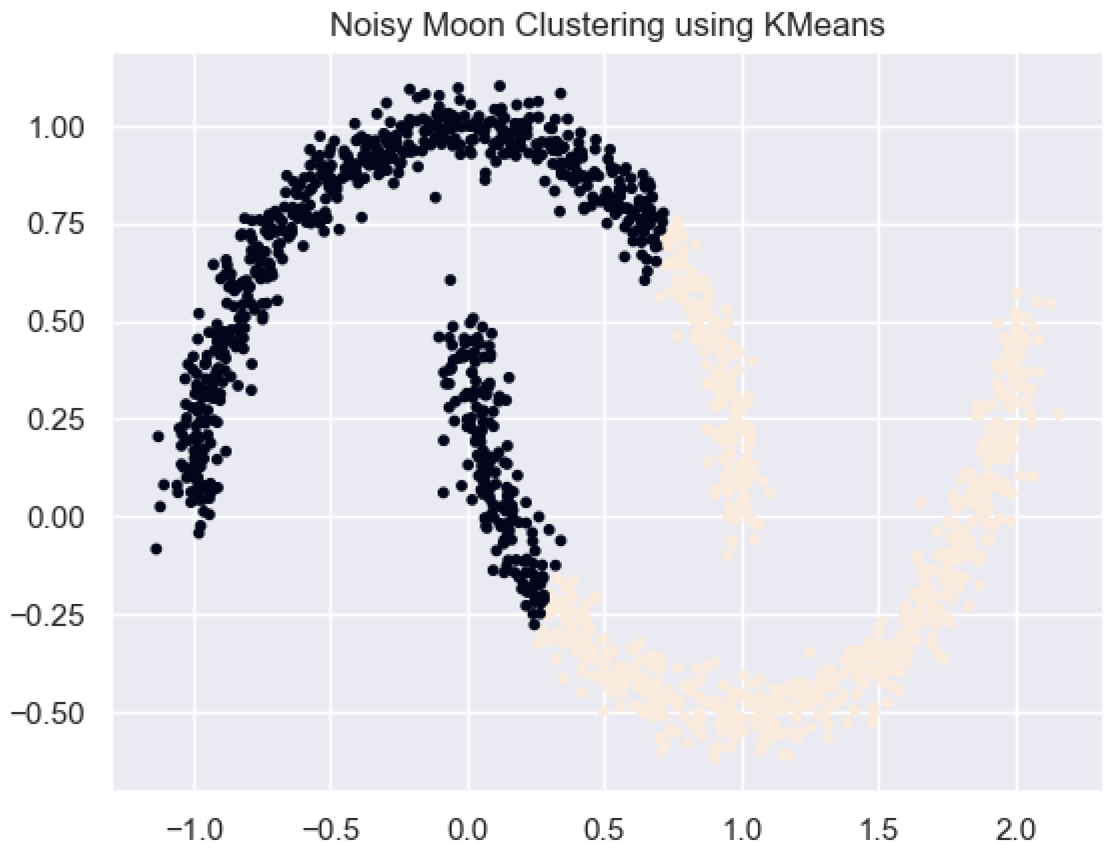


In [187…

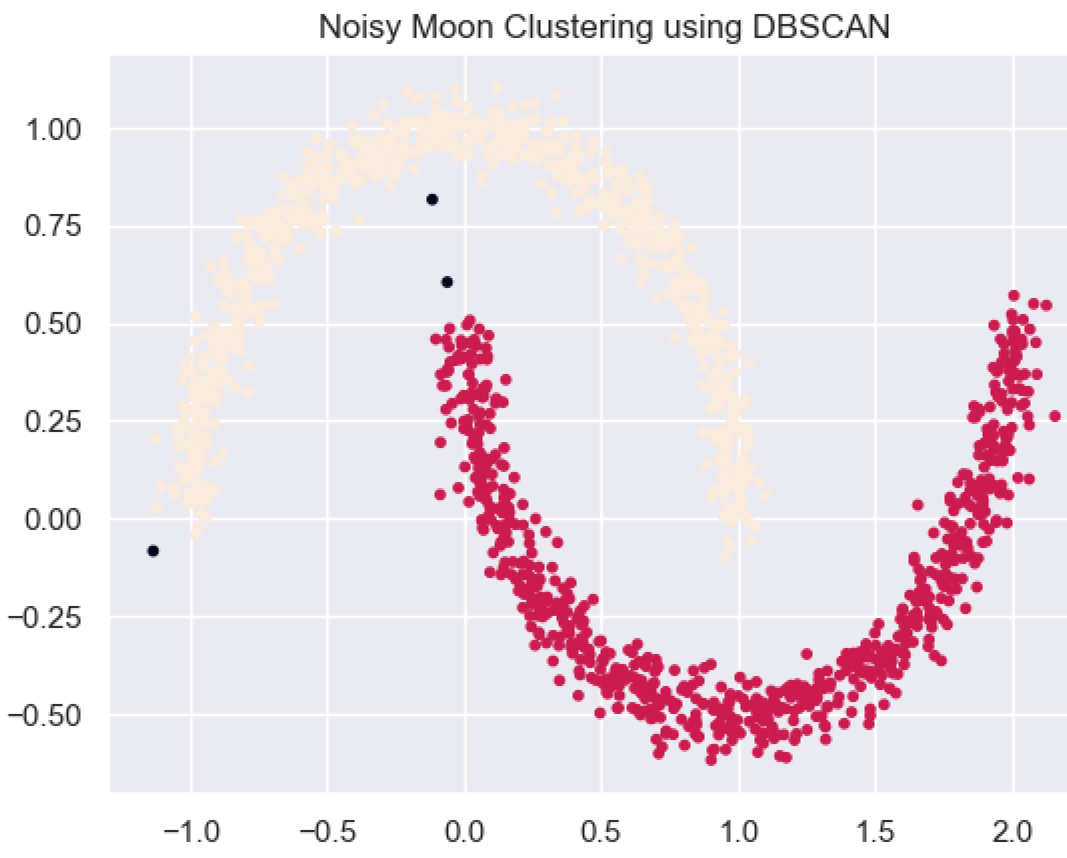
```
# Write a function to acquire core, noise and border points
core_points_mask = np.zeros_like(clustering.labels_, dtype=bool)
core_points_mask[clustering.core_sample_indices_] = True
noise_points_mask = (clustering.labels_ == -1)
border_points_mask = ~(core_points_mask | noise_points_mask)
core_points = compounds_DBSCAN[core_points_mask]
border_points = compounds_DBSCAN[border_points_mask]
noise_points = compounds_DBSCAN[noise_points_mask]
print(len(core_points), len(border_points), len(noise_points))
```

```
103 26 21
```

(b) In the case of noisy moon dataset, DBSCAN work much better than KMeans.

In [198…
```
# Clustering the noisy moon dataset using KMeans
K_moon = KMeans(2, maximum_iters=100)
moon_clustering_k = K_moon.cluster(X)
moon_clustering_k
plt.scatter(X[:, 0], X[:, 1], s=10,c=moon_clustering_k)
plt.title('Noisy Moon Clustering using KMeans')
```
Out[198]:
```
Text(0.5, 1.0, 'Noisy Moon Clustering using KMeans')
```


In [203…
```
# Clustering the noisy moon dataset using DBSCAN
db_moon = DBSCAN(eps=0.1, min_samples=10)
clustering_moon = db_moon.fit(X)
# Cluster labels for each point in the dataset given to fit().  Noisy samples are given the label -1.
clustering_moon.labels_
plt.scatter(X[:, 0], X[:, 1], s=10,c=clustering_moon.labels_)
plt.title('Noisy Moon Clustering using DBSCAN')
```
Out[203]:
```
Text(0.5, 1.0, 'Noisy Moon Clustering using DBSCAN')
```


Noisy Moon Clustering using DBSCAN
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0

```
In [ ]:
```