project3

March 26, 2021

1 Dave Miller - CSCI 347 Project 3

1.1 Part 1: The data

1. I think this data set is interesting to me because it is high dimensional, nature-oriented, and came in a nice csv with a pdf description.
2. There are 15 numerical attributes describing each sample (leaf) and a class label.
3. There are no missing values in the data set.
4. I expect clusters to be present because it would make sense that leaf measurements would group into their respective plant species. Finding clusters would be helpful because if it worked well and I wanted to make an ML model, I would not have to label any other samples in order to classify them since clustering is unsupervised. I expect to see ~30 clusters in the dataset since that is the number of species present. I think these clusters will be similar sizes since there is a pretty even distribution between classes.

[144]:
```
import pandas as pd
import numpy as np
import random
```

[145]:
```
data = pd.read_csv('datasets/leaf/leaf.csv', header=None)
data.head()
```

[145]:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	1	0.72694	1.4742	0.32396	0.98535	1.00000	0.83592	0.004657	0.003947	0.047790	0.127950	0.016108	0.005232	0.000275	1.17560
1	1	2	0.74173	1.5257	0.36116	0.98152	0.99825	0.79867	0.005242	0.005002	0.024160	0.090476	0.008119	0.002708	0.000075	0.69659
2	1	3	0.76722	1.5725	0.38998	0.97755	1.00000	0.80812	0.007457	0.010121	0.011897	0.057445	0.003289	0.000921	0.000038	0.44348
3	1	4	0.73797	1.4597	0.35376	0.97566	1.00000	0.81697	0.006877	0.008607	0.015950	0.065491	0.004271	0.001154	0.000066	0.58785
4	1	5	0.82301	1.7707	0.44462	0.97698	1.00000	0.75493	0.007428	0.010042	0.007938	0.045339	0.002051	0.000560	0.000024	0.34214

[146]:
```
# No missing / null
data.isna().sum()
```

[146]:
```
0     0
1     0
2     0
3     0
4     0
5     0
6     0
7     0
8     0
9     0
10    0
11    0
12    0
13    0
14    0
15    0
dtype: int64
```

[147]:
```
# 30 classes
np.unique(data[0])
```

[147]:
```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 22, 23,
       24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36], dtype=int64)
```

[148]:
```
# number of samples in each class
classCounts = [sum(data[0] == x) for x in np.unique(data[0])]
print(classCounts)
```

```
[12, 10, 10, 8, 12, 8, 10, 11, 14, 13, 16, 12, 13, 12, 10, 12, 11, 13, 9, 12,
11, 12, 12, 12, 11, 11, 11, 11, 11, 10]
```

[149]:
```
# drop the class label & specimen number attributes
labels = data[0]
data.drop([0, 1], axis=1, inplace=True)
data.columns
```

[149]:
```
Int64Index([2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15], dtype='int64')
```

[150]:
```
# rename axis labels (probably a better way to do this)
data.rename(columns={2:0, 3:1, 4:2, 5:3, 6:4, 7:5, 8:6, 9:7, 10:8, 11:9, 12:10,
 ↪13:11, 14:12, 15:13}, inplace=True)
```

1.2 Part 2: Functions for clustering

K-means

[151]:
```
# gives euclidean distance between 2 n-dimensional points
def dist(x1, x2):
    if len(x1) != len(x2):
```

```
        return -1

    sum = 0
    for i in range(len(x1)):
        sum += (x1[i] - x2[i])**2

    return sum**0.5
```

[152]:
```
def kMeans(data, k, epsilon):
    # init random centroids
    mins = [min(feat) for feat in data.T]
    maxes = [max(feat) for feat in data.T]
    centroids = []

    for i in range(k):
        centroids.append([])
        for j in range(len(mins)):
            centroids[i].append(random.uniform(mins[j], maxes[j]))

    iterations = 0
    while(iterations < 300):
        iterations += 1

        # cluster assignment
        clusters = [[] for _ in range(k)]
        assignments = [-1 for _ in range(len(data))]
        for i in range(len(data)):
            sample = data[i]
            dists = [dist(list(sample), cent) for cent in centroids]
            c = np.argmin(dists)
            clusters[c].append(i)
            assignments[i] = c

        # update centroids
        converged = 0
        for i in range(len(centroids)):
            newCenter = [0] * len(centroids[0])

            if len(clusters[i]) > 0:
                for sample in clusters[i]: # indices of samples
                    for z in range(len(data[sample])): # going through each␣
↪sample in cluster
                        newCenter[z] += list(data[sample])[z]

                for z in range(len(newCenter)): # averaging over # of samples␣
↪in the cluster
                    newCenter[z] /= len(clusters[i])
```

```
                converged += dist(centroids[i], newCenter)
                centroids[i] = newCenter

        if converged < epsilon:
            break

    return assignments, centroids
```

Test cases - kMeans

[153]:
```
D = np.array([[-1,2], [-2,-2], [3,2], [5,4.3], [-3,3], [-3, -1], [5,-3], [3,4],␣
↪[2.3, 6.5]])
labels, means = kMeans(D, 2, 0.000001)
print(means)
print(labels)
```

```
[[1.55, 3.6333333333333333], [0.0, -2.0]]
[0, 1, 0, 0, 0, 1, 1, 0, 0]
```

[154]:
```
D = np.array([[-1,2], [-2,-2], [3,2], [5,4.3], [-3,3], [-3, -1], [5,-3], [3,4],␣
↪[2.3, 6.5], [4, 2], [4,4], [-2.3, 1.5]])
labels, means = kMeans(D, 3, 0.000001)
print(means)
print(labels)
```

```
[[2.3, 6.5], [-2.2600000000000002, 0.7], [4.0, 2.216666666666667]]
[1, 1, 2, 2, 1, 1, 2, 2, 0, 2, 2, 1]
```

DBSCAN

[155]:
```
def getNeighbors(data, point, eps):
    neighbors = []

    for i in range(len(data)):
        if dist(data[point], data[i]) <= eps:
            neighbors.append(i)

    return neighbors
```

[156]:
```
def DBSCAN(data, epsilon, minPts):
    labels = [-1]*len(data)
    corePts = []
    borderPts = []
    noisePts = []
    c = 0

    for i in range(len(data)):
        # already labeled
        if labels[i] != -1:
```

```
            continue

        # get neighboring points
        neighbors = getNeighbors(data, i, epsilon)

        # if number of neighbors < minPts -> label as noise
        if len(neighbors) < minPts:
            labels[i] = 0
            noisePts.append(i)

        # else label it a core point and grow cluster
        else:
            c += 1
            labels[i] = c
            corePts.append(i)

            pt = 0
            while pt < len(neighbors):
                n = neighbors[pt]

                # if noise -> assign it to c
                if labels[n] == 0:
                    labels[n] = c
                    borderPts.append(n)
                    noisePts.remove(n)

                # if unassigned -> assign it to c and add its neighbors
                elif labels[n] == -1:
                    labels[n] = c

                    # get neighbors of n
                    nNeighbors = getNeighbors(data, n, epsilon)

                    if len(nNeighbors) >= minPts:
                        neighbors += nNeighbors
                        corePts.append(n)
                    else:
                        borderPts.append(n)

                pt += 1

    return labels, [corePts, borderPts, noisePts]
```

DBSCAN test cases

[157]:
```
from sklearn.datasets import make_blobs
X,y = make_blobs(n_samples=300, centers=3, random_state=35)
```

```
labels, points = DBSCAN(X, 0.5, 10)
print(labels)
```

```
[0, 1, 0, 0, 0, 3, 1, 0, 0, 2, 1, 1, 3, 0, 1, 1, 0, 2, 2, 0, 2, 2, 2, 1, 3, 0,
3, 0, 0, 0, 3, 0, 0, 1, 1, 3, 2, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 2, 3, 3, 0,
0, 0, 2, 1, 2, 0, 0, 0, 2, 1, 2, 1, 0, 3, 0, 0, 1, 1, 0, 0, 0, 2, 2, 2, 2, 0, 2,
0, 3, 0, 0, 0, 0, 1, 1, 1, 0, 3, 0, 1, 0, 2, 3, 3, 1, 1, 1, 0, 0, 0, 3, 1, 1, 0,
0, 0, 0, 1, 0, 3, 0, 3, 2, 3, 1, 0, 1, 0, 0, 2, 1, 0, 1, 2, 0, 0, 1, 0, 0, 2, 1,
2, 3, 0, 1, 2, 0, 0, 3, 0, 2, 0, 2, 2, 1, 0, 0, 0, 3, 2, 0, 3, 1, 0, 0, 1, 0, 2,
2, 3, 0, 3, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 3, 1, 0, 0, 2, 0, 3, 3,
2, 0, 1, 3, 0, 0, 3, 0, 0, 0, 0, 0, 0, 3, 1, 2, 1, 1, 0, 0, 2, 0, 0, 3, 3, 3, 3,
1, 0, 1, 0, 0, 0, 0, 0, 1, 3, 0, 2, 0, 0, 0, 0, 0, 0, 0, 2, 0, 1, 3, 0, 2, 3, 0,
1, 3, 0, 1, 0, 3, 2, 0, 2, 0, 0, 1, 0, 0, 0, 3, 1, 2, 3, 0, 3, 1, 3, 3, 0, 1, 0,
0, 2, 0, 0, 0, 0, 2, 0, 2, 0, 2, 3, 0, 3, 0, 0, 0, 0, 3, 0, 0, 0, 1, 1, 0, 1, 0,
2, 0, 0, 2]
```

[158]:
```
from sklearn.datasets import make_moons
X_moons, y = make_moons(n_samples=200, noise=.06, random_state=4)

labels, points = DBSCAN(X_moons, 0.4, 20)
print(labels)
```

```
[1, 1, 2, 2, 1, 2, 1, 2, 1, 2, 1, 2, 2, 2, 2, 2, 1, 2, 2, 1, 1, 2, 2, 1, 1, 2,
2, 1, 1, 2, 1, 1, 2, 1, 2, 2, 2, 1, 2, 1, 1, 2, 2, 2, 2, 1, 1, 2, 1, 1, 2, 2, 1,
1, 1, 2, 2, 1, 2, 1, 1, 2, 1, 2, 2, 1, 2, 1, 1, 1, 1, 2, 2, 1, 1, 2, 1, 2, 2, 2,
1, 1, 2, 1, 2, 1, 2, 2, 1, 1, 2, 1, 1, 2, 2, 2, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1,
1, 2, 2, 2, 2, 2, 1, 2, 2, 1, 2, 1, 1, 2, 2, 2, 1, 1, 1, 2, 1, 1, 1, 1, 2, 2, 2,
2, 1, 1, 2, 1, 1, 1, 2, 1, 1, 1, 1, 2, 2, 1, 1, 1, 2, 2, 1, 2, 2, 1, 1, 2, 2, 1,
2, 2, 1, 2, 2, 2, 1, 1, 2, 1, 2, 1, 2, 2, 1, 1, 2, 1, 2, 1, 1, 2, 2, 2, 1, 2, 1,
1, 2, 2, 1, 2, 1, 1, 2, 2, 2, 2, 2]
```

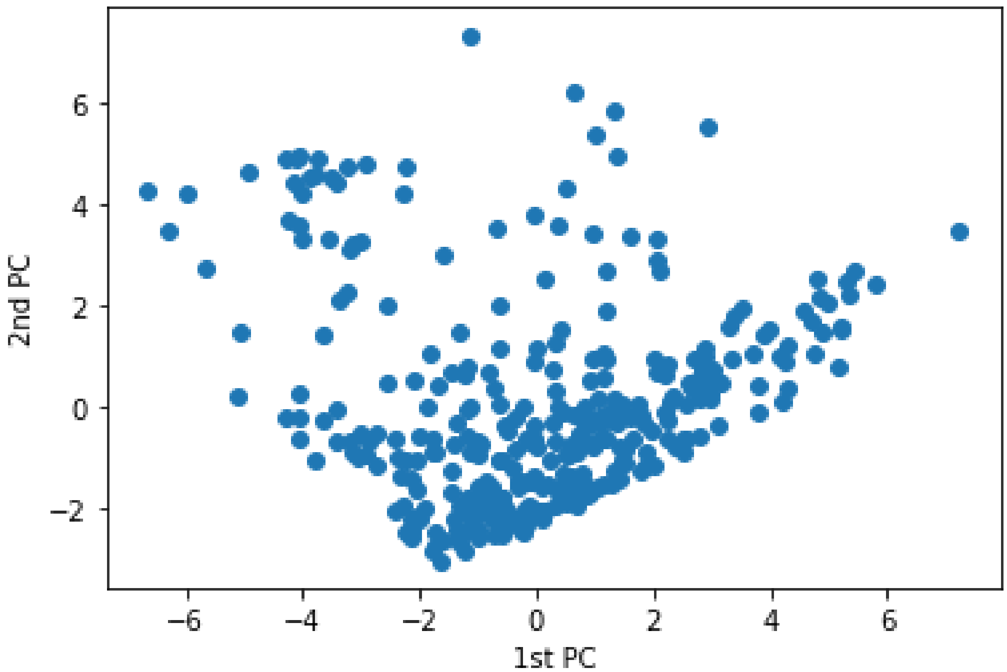
1.3 Part 3: Anaylze the data

8. It looks like there are 2ish clusters, with one cluster holding most of the points, and a lot of noise points.

[159]:
```
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
D = StandardScaler().fit_transform(data)
pca = PCA(n_components=2)
D_pca = pca.fit_transform(D)
```

[160]:
```
import matplotlib.pyplot as plt
plt.scatter(D_pca.T[0], D_pca.T[1])
plt.xlabel('1st PC')
plt.ylabel('2nd PC')
```

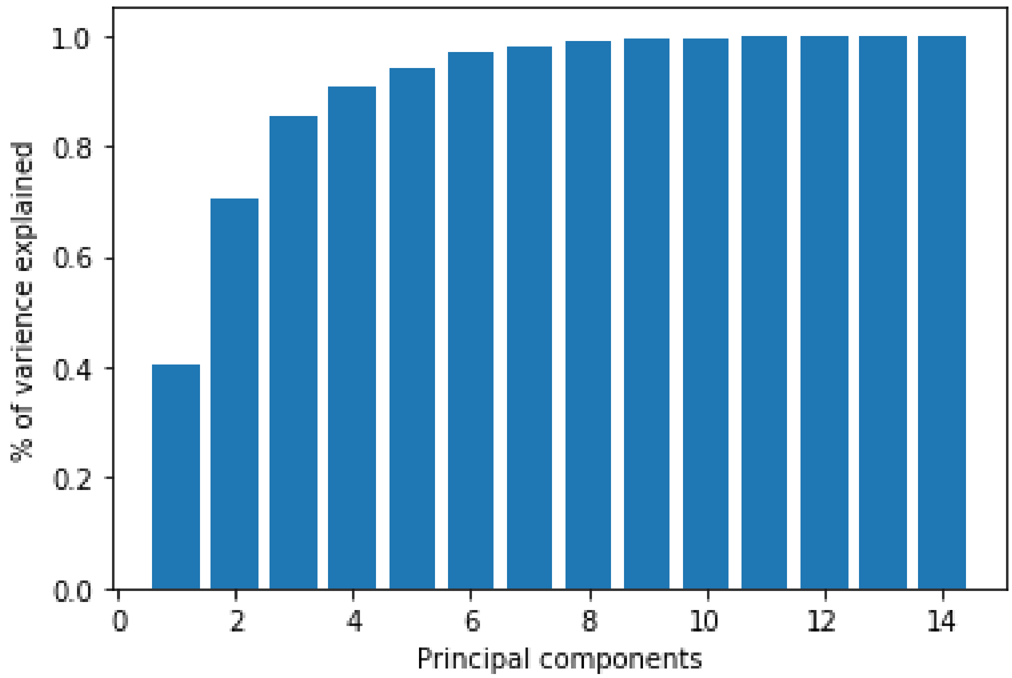
[160]: Text(0, 0.5, '2nd PC')



9. I will choose n_components = 6 since that captures ~97% of the variance

[161]:
```
pca = PCA()
D_pca_2 = pca.fit_transform(D)

var = 0
totalVar = []
for i in pca.explained_variance_ratio_:
    var += i
    totalVar.append(var)

i = [x for x in range(1, pca.n_components_ + 1)]
plt.bar(i, totalVar)
plt.xlabel('Principal components')
plt.ylabel('% of varience explained')
```

[161]: Text(0, 0.5, '% of varience explained')

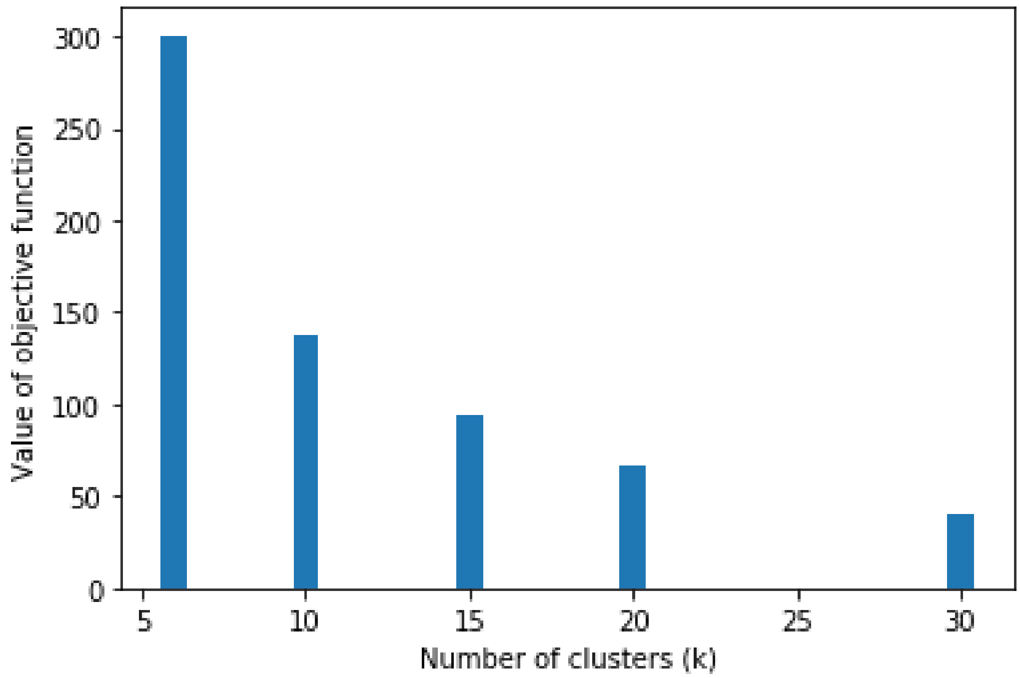


[162]:
```
pca = PCA(n_components=6)
D_pca_final = pca.fit_transform(D)
```

10. By plotting various values of k and their intertias, I found that for both the regular and PCA transformed datasets, the highest k value of 30 was the best.

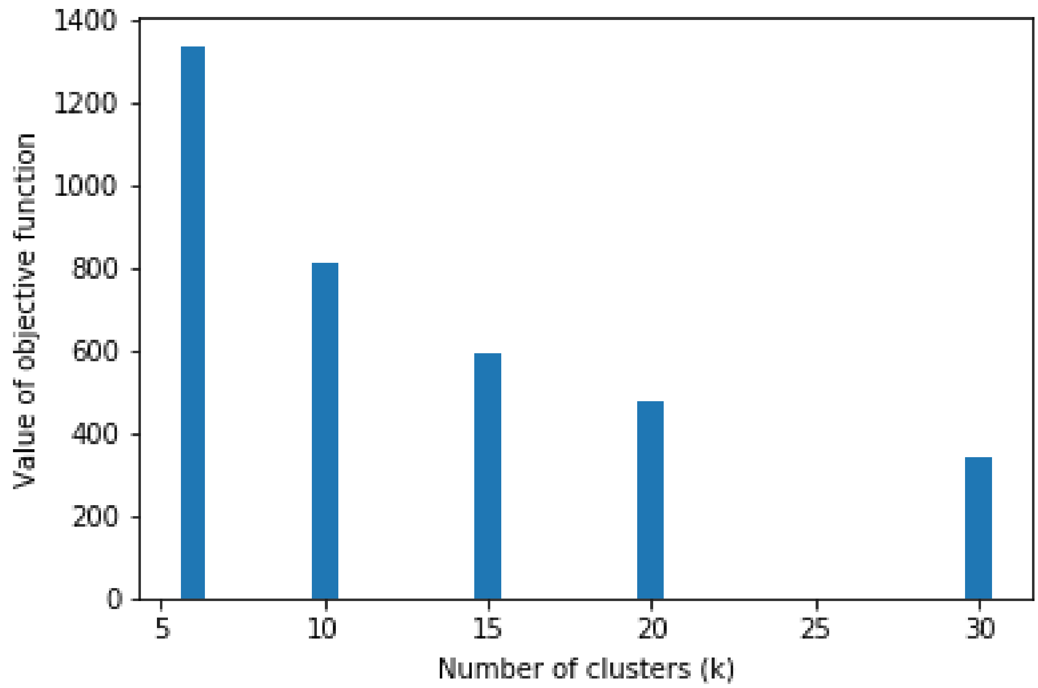
[163]:
```
from sklearn.cluster import KMeans

# Regular data set
k = [6, 10, 15, 20, 30]
intertias = []

for val in k:
    kmeans = KMeans(n_clusters=val)
    kmeans.fit_predict(data)
    intertias.append(kmeans.inertia_)

plt.bar(k, intertias)
plt.xlabel('Number of clusters (k)')
plt.ylabel('Value of objective function')
```

[163]: Text(0, 0.5, 'Value of objective function')




```
[164]: # PCA transformed data set
k = [6, 10, 15, 20, 30]
intertias = []

for val in k:
    kmeans = KMeans(n_clusters=val)
    kmeans.fit_predict(D_pca_final)
    intertias.append(kmeans.inertia_)

plt.bar(k, intertias)
plt.xlabel('Number of clusters (k)')
plt.ylabel('Value of objective function')
```

```
[164]: Text(0, 0.5, 'Value of objective function')
```

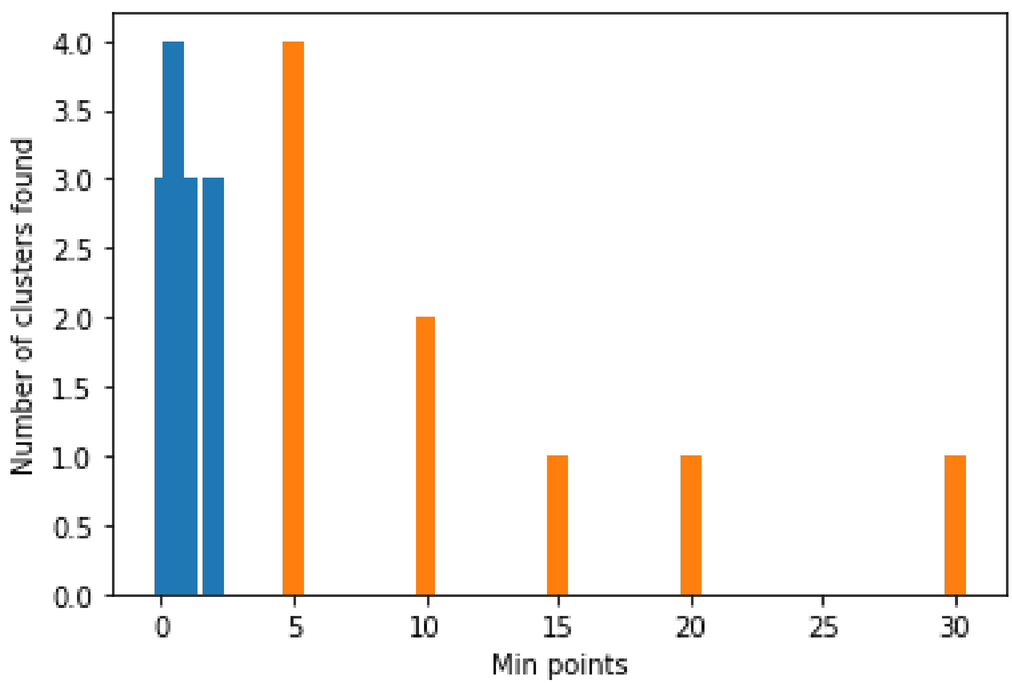


11. These plots show various epsilon values and their number of clusters found in blue and various minPts values with their number of clusters found in orange. It looks like the lower epsilons and minPts found more clusters.

[165]:
```
from sklearn.cluster import DBSCAN

# Regular data
eps = [0.1, 0.2, 0.5, 1, 2]
minPts = [5, 10, 15, 20, 30]

numClustersEps = []
numClustersMinPts = []

# keeping minPts fixed at default 5
for epsilon in eps:
    db = DBSCAN(eps=epsilon, min_samples=5)
    db.fit_predict(data)
    numClusters = len(np.unique(db.labels_)) - 1   # not counting noise points
    numClustersEps.append(numClusters)

# keeping epsilon fixed at default 0.5
for m in minPts:
    db = DBSCAN(eps=0.5, min_samples=m)
    db.fit_predict(data)
    numClusters = len(np.unique(db.labels_)) - 1
```

```
    numClustersMinPts.append(numClusters)

plt.bar(eps, numClustersEps)
plt.xlabel('Epsilon')
plt.ylabel('Number of clusters found')

plt.bar(minPts, numClustersMinPts)
plt.xlabel('Min points')
plt.ylabel('Number of clusters found')
```

[165]: Text(0, 0.5, 'Number of clusters found')



[166]:
```
# PCA transformed data
eps = [0.1, 0.2, 0.5, 1, 2]
minPts = [5, 10, 15, 20, 30]

numClustersEps = []
numClustersMinPts = []

# keeping minPts fixed at default 5
for epsilon in eps:
    db = DBSCAN(eps=epsilon, min_samples=5)
    db.fit_predict(D_pca_final)
    numClusters = len(np.unique(db.labels_)) - 1   # not counting noise points
```

```
    numClustersEps.append(numClusters)

# keeping epsilon fixed at default 0.5
for m in minPts:
    db = DBSCAN(eps=0.5, min_samples=m)
    db.fit_predict(D_pca_final)
    numClusters = len(np.unique(db.labels_)) - 1
    numClustersMinPts.append(numClusters)

plt.bar(eps, numClustersEps)
plt.xlabel('Epsilon')
plt.ylabel('Number of clusters found')

plt.bar(minPts, numClustersMinPts)
plt.xlabel('Min points')
plt.ylabel('Number of clusters found')
```

[166]: Text(0, 0.5, 'Number of clusters found')