lab12

October 6, 2023

0.1 Lab 12: Unsupervised Learning

Jack Krebsbach Math 313

0.1.1 12.5.1 Principal Components Analysis

First we explore the `USArrests` data set.

[ ]:
```
## All of our imports
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from statsmodels.datasets  import get_rdataset
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
from ISLP import load_data
```

[ ]:
```
## New imports needed
from sklearn.cluster import (KMeans,AgglomerativeClustering)
from scipy.cluster.hierarchy import  (dendrogram, cut_tree)
from ISLP.cluster import compute_linkage
```

There are 50 rows in this data set, which contain the states in alphabetical order.

[ ]:
```
USArrests = get_rdataset('USArrests').data
USArrests
```

[ ]:
```
                Murder  Assault  UrbanPop  Rape
rownames
Alabama           13.2      236        58  21.2
Alaska            10.0      263        48  44.5
Arizona            8.1      294        80  31.0
Arkansas           8.8      190        50  19.5
California         9.0      276        91  40.6
Colorado           7.9      204        78  38.7
Connecticut        3.3      110        77  11.1
Delaware           5.9      238        72  15.8
Florida           15.4      335        80  31.9
Georgia           17.4      211        60  25.8
```

```
Hawaii            5.3      46        83  20.2
Idaho             2.6     120        54  14.2
Illinois         10.4     249        83  24.0
Indiana           7.2     113        65  21.0
Iowa              2.2      56        57  11.3
Kansas            6.0     115        66  18.0
Kentucky          9.7     109        52  16.3
Louisiana        15.4     249        66  22.2
Maine             2.1      83        51   7.8
Maryland         11.3     300        67  27.8
Massachusetts     4.4     149        85  16.3
Michigan         12.1     255        74  35.1
Minnesota         2.7      72        66  14.9
Mississippi      16.1     259        44  17.1
Missouri          9.0     178        70  28.2
Montana           6.0     109        53  16.4
Nebraska          4.3     102        62  16.5
Nevada           12.2     252        81  46.0
New Hampshire     2.1      57        56   9.5
New Jersey        7.4     159        89  18.8
New Mexico       11.4     285        70  32.1
New York         11.1     254        86  26.1
North Carolina   13.0     337        45  16.1
North Dakota      0.8      45        44   7.3
Ohio              7.3     120        75  21.4
Oklahoma          6.6     151        68  20.0
Oregon            4.9     159        67  29.3
Pennsylvania      6.3     106        72  14.9
Rhode Island      3.4     174        87   8.3
South Carolina   14.4     279        48  22.5
South Dakota      3.8      86        45  12.8
Tennessee        13.2     188        59  26.9
Texas            12.7     201        80  25.5
Utah              3.2     120        80  22.9
Vermont           2.2      48        32  11.2
Virginia          8.5     156        63  20.7
Washington        4.0     145        73  26.2
West Virginia     5.7      81        39   9.3
Wisconsin         2.6      53        66  10.8
Wyoming           6.8     161        60  15.6
```

[ ]:
```
USArrests.columns
```

[ ]:
```
Index(['Murder', 'Assault', 'UrbanPop', 'Rape'], dtype='object')
```

[ ]:
```
# The variables have lots of different means.
USArrests.mean()
```

```
[ ]: Murder        7.788
     Assault     170.760
     UrbanPop     65.540
     Rape         21.232
     dtype: float64
```

```
[ ]: # We can compute statistics like variance on these features
     USArrests.var()
```

```
[ ]: Murder        18.970465
     Assault     6945.165714
     UrbanPop     209.518776
     Rape          87.729159
     dtype: float64
```

```
[ ]: # We can standardize the data.
     scaler = StandardScaler(with_std=True, with_mean=True)
     USArrests_scaled = scaler.fit_transform(USArrests)
     USArrests_scaled
```

```
[ ]: array([[ 1.25517927,  0.79078716, -0.52619514, -0.00345116],
            [ 0.51301858,  1.11805959, -1.22406668,  2.50942392],
            [ 0.07236067,  1.49381682,  1.00912225,  1.05346626],
            [ 0.23470832,  0.23321191, -1.08449238, -0.18679398],
            [ 0.28109336,  1.2756352 ,  1.77678094,  2.08881393],
            [ 0.02597562,  0.40290872,  0.86954794,  1.88390137],
            [-1.04088037, -0.73648418,  0.79976079, -1.09272319],
            [-0.43787481,  0.81502956,  0.45082502, -0.58583422],
            [ 1.76541475,  1.99078607,  1.00912225,  1.1505301 ],
            [ 2.22926518,  0.48775713, -0.38662083,  0.49265293],
            [-0.57702994, -1.51224105,  1.21848371, -0.11129987],
            [-1.20322802, -0.61527217, -0.80534376, -0.75839217],
            [ 0.60578867,  0.94836277,  1.21848371,  0.29852525],
            [-0.13637203, -0.70012057, -0.03768506, -0.0250209 ],
            [-1.29599811, -1.39102904, -0.5959823 , -1.07115345],
            [-0.41468229, -0.67587817,  0.03210209, -0.34856705],
            [ 0.44344101, -0.74860538, -0.94491807, -0.53190987],
            [ 1.76541475,  0.94836277,  0.03210209,  0.10439756],
            [-1.31919063, -1.06375661, -1.01470522, -1.44862395],
            [ 0.81452136,  1.56654403,  0.10188925,  0.70835037],
            [-0.78576263, -0.26375734,  1.35805802, -0.53190987],
            [ 1.00006153,  1.02108998,  0.59039932,  1.49564599],
            [-1.1800355 , -1.19708982,  0.03210209, -0.68289807],
            [ 1.9277624 ,  1.06957478, -1.5032153 , -0.44563089],
            [ 0.28109336,  0.0877575 ,  0.31125071,  0.75148985],
            [-0.41468229, -0.74860538, -0.87513091, -0.521125  ],
            [-0.80895515, -0.83345379, -0.24704653, -0.51034012],
            [ 1.02325405,  0.98472638,  1.0789094 ,  2.671197  ],
            [-1.31919063, -1.37890783, -0.66576945, -1.26528114],
            [-0.08998698, -0.14254532,  1.63720664, -0.26228808],
            [ 0.83771388,  1.38472601,  0.31125071,  1.17209984],
            [ 0.76813632,  1.00896878,  1.42784517,  0.52500755],
            [ 1.20879423,  2.01502847, -1.43342815, -0.55347961],
            [-1.62069341, -1.52436225, -1.5032153 , -1.50254831],
            [-0.11317951, -0.61527217,  0.66018648,  0.01811858],
            [-0.27552716, -0.23951493,  0.1716764 , -0.13286962],
            [-0.66980002, -0.14254532,  0.10188925,  0.87012344],
            [-0.34510472, -0.78496898,  0.45082502, -0.68289807],
            [-1.01768785,  0.03927269,  1.49763233, -1.39469959],
            [ 1.53348953,  1.3119988 , -1.22406668,  0.13675217],
            [-0.92491776, -1.027393  , -1.43342815, -0.90938037],
            [ 1.25517927,  0.20896951, -0.45640799,  0.61128652],
            [ 1.13921666,  0.36654512,  1.00912225,  0.46029832],
            [-1.06407289, -0.61527217,  1.00912225,  0.17989166],
            [-1.29599811, -1.48799864, -2.34066115, -1.08193832],
            [ 0.16513075, -0.17890893, -0.17725937, -0.05737552],
            [-0.87853272, -0.31224214,  0.52061217,  0.53579242],
            [-0.48425985, -1.08799901, -1.85215107, -1.28685088],
            [-1.20322802, -1.42739264,  0.03210209, -1.1250778 ],
            [-0.22914211, -0.11830292, -0.38662083, -0.60740397]])
```

Once we scale the data we can now perform `PCA` on the data.

[ ]:
```
np.std(USArrests_scaled[:, 1])
```

[ ]:
```
1.0
```

[ ]:
```
## This uses the PCA package versus by hand
pcaUS = PCA()
```

[ ]:
```
pcaUS.fit(USArrests_scaled)
```

[ ]:
```
PCA()
```

[ ]:
```
pcaUS.mean_
```

[ ]:
```
array([-7.10542736e-17,  1.38777878e-16, -4.39648318e-16,  8.59312621e-16])
```

[ ]:
```
scores = pcaUS.transform(USArrests_scaled)
```

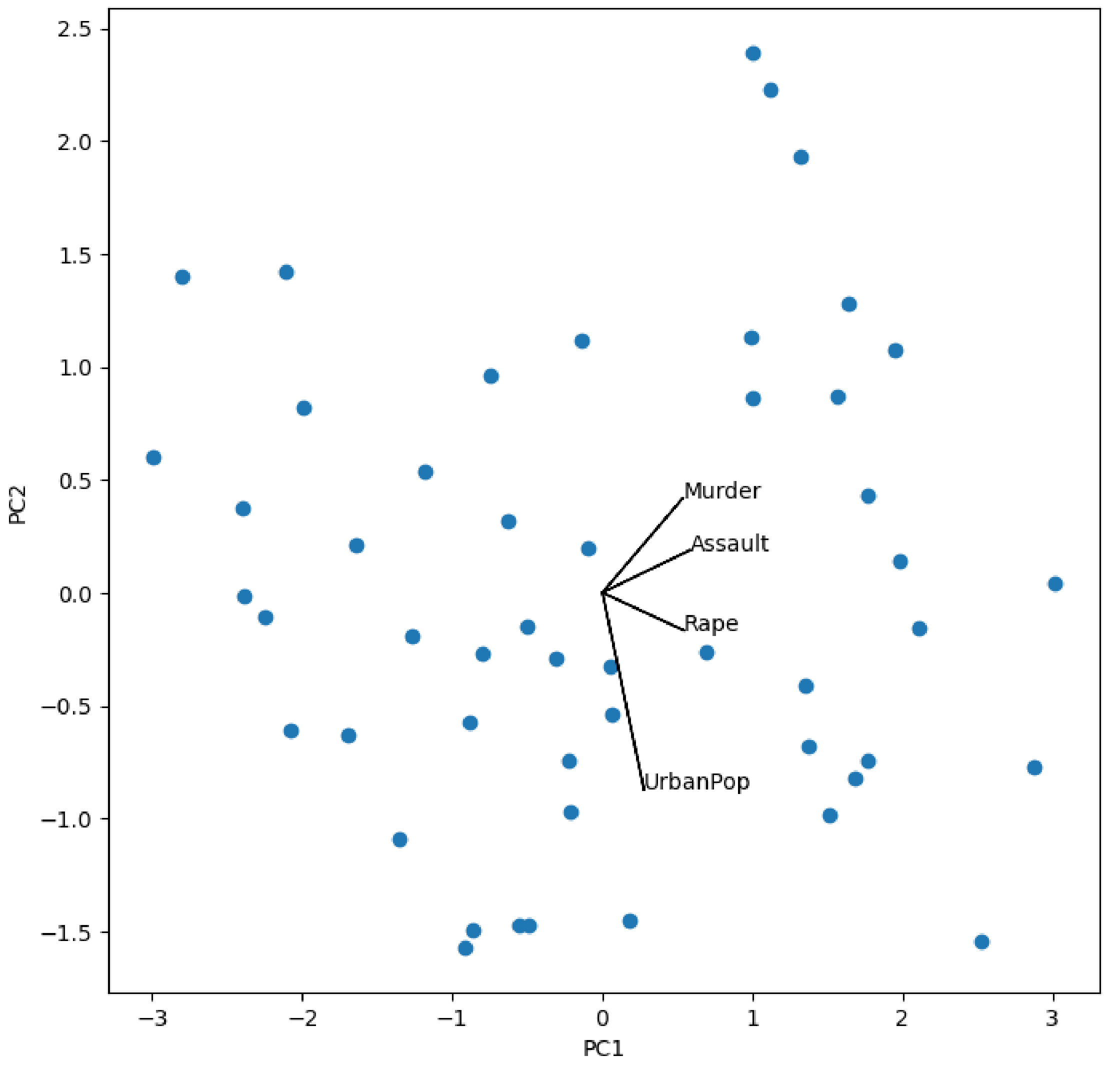
Principal Component Loadings The loadings of the principal component anlsysis can be extracted using `.components_` of the `PCA` analsysi. Each row contains the PC loading vector.

[ ]:
```
pcaUS.components_
```

```
[ ]: array([[ 0.53589947,  0.58318363,  0.27819087,  0.54343209],
            [ 0.41818087,  0.1879856 , -0.87280619, -0.16731864],
            [-0.34123273, -0.26814843, -0.37801579,  0.81777791],
            [ 0.6492278 , -0.74340748,  0.13387773,  0.08902432]])
```

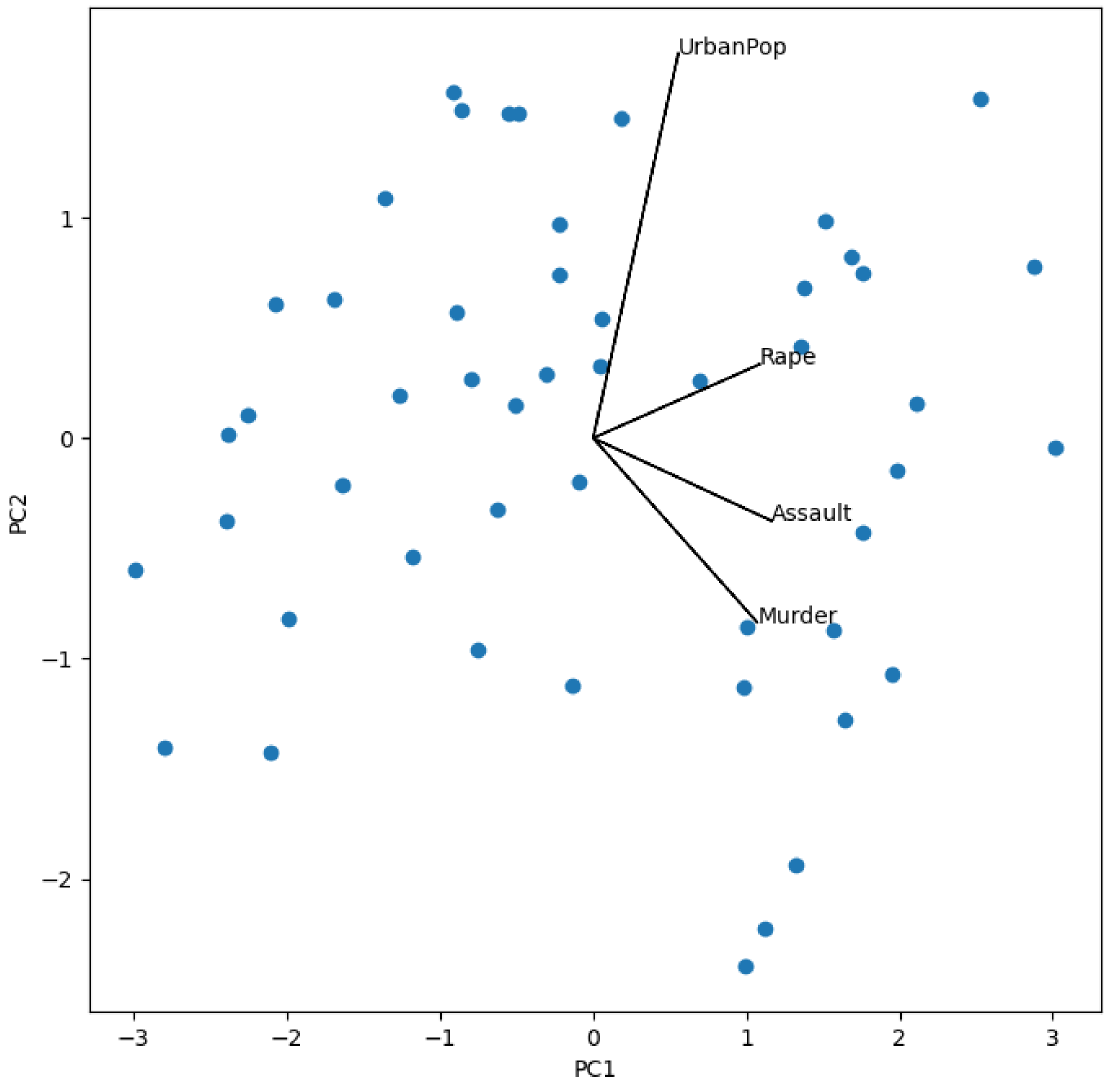
```
[ ]: # Here we make the bi-plot manually as it is not a standard import from sklearn.

     i, j = 0, 1 # which components
     fig, ax = plt.subplots(1, 1, figsize=(8, 8))
     ax.scatter(scores[:,0], scores[:,1])
     ax.set_xlabel('PC%d' % (i+1))
     ax.set_ylabel('PC%d' % (j+1))
     for k in range(pcaUS.components_.shape[1]):
         ax.arrow(0, 0, pcaUS.components_[i,k], pcaUS.components_[j,k])
         ax.text(pcaUS.components_[i,k], pcaUS.components_[j,k], USArrests.
      ↪columns[k])
```



This figure is a reflection (through the y-axis) from the book in figure 12.1. The PC are unique only up to sign change. Thus, we can replicate the figure by flipping the signs and can more clearly see the differences between loadings by increasing the length of the arrows.

```
[ ]: scale_arrow = s_ = 2
     scores[:,1] *= -1
     pcaUS.components_[1] *= -1 # flip the y-axis
     fig, ax = plt.subplots(1, 1, figsize=(8, 8))
     ax.scatter(scores[:,0], scores[:,1])
     ax.set_xlabel('PC%d' % (i+1))
     ax.set_ylabel('PC%d' % (j+1))
     for k in range(pcaUS.components_.shape[1]):
         ax.arrow(0, 0, s_*pcaUS.components_[i,k], s_*pcaUS.components_[ j,k])
         ax.text(s_*pcaUS.components_[i,k], s_*pcaUS.components_[j,k],
         USArrests.columns[k])
```



[ ]:
```
# We grab the standard deviations
scores.std(0, ddof=1)
```

[ ]:
```
array([1.5908673 , 1.00496987, 0.6031915 , 0.4206774 ])
```

What we are really after is how much variance each PC can explain, here we can find the explained variance by component. It is helpful to also find the ratio of explained variance to get a better of idea this.

[ ]:
```
pcaUS.explained_variance_
```

[ ]:
```
array([2.53085875, 1.00996444, 0.36383998, 0.17696948])
```

[ ]:
```
pcaUS.explained_variance_ratio_
```

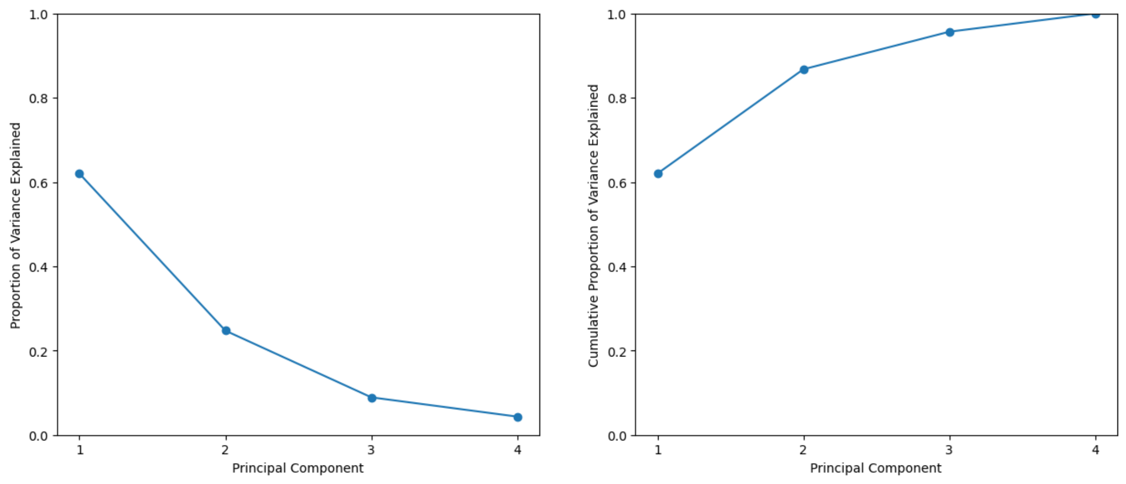
```
[ ]: array([0.62006039, 0.24744129, 0.0891408 , 0.04335752])
```

The first principal component explains 60% of the variance in the data. To visually see this we can plot it below.

```
[ ]: %%capture
fig, axes = plt.subplots(1, 2, figsize=(15, 6))
ticks = np.arange(pcaUS.n_components_)+1
ax = axes[0]
ax.plot(ticks,
pcaUS.explained_variance_ratio_ , marker='o')
ax.set_xlabel('Principal Component'); ax.set_ylabel('Proportion of Variance␣
 ↪Explained')
ax.set_ylim([0,1])
ax.set_xticks(ticks)
ax.set_xlabel('Principal Component'); ax.set_ylabel('Proportion of Variance␣
 ↪Explained')
ax.set_ylim([0,1])
ax.set_xticks(ticks)
```

```
[ ]: ax = axes[1]
ax.plot(ticks,
pcaUS.explained_variance_ratio_.cumsum(),
marker='o')
ax.set_xlabel('Principal Component')
ax.set_ylabel('Cumulative Proportion of Variance Explained')
ax.set_ylim([0, 1])
ax.set_xticks(ticks)
fig
```

[ ]:



```
[ ]: a = np.array([1,2,8,-3])
     np.cumsum(a)
```

```
[ ]: array([ 1,  3, 11,  8])
```

0.1.2 12.5.2 Matrix Completion

Our goal in this section is to re-create the analysis of the data set in section 12.3. Here we use `Singular Value Decomposition` to solve for the principal components of the data.

```
[ ]: X = USArrests_scaled
     ## Theses are the matrix that our matrix A can be decomposed into.
     U, D, V = np.linalg.svd(X, full_matrices=False)
     U.shape, D.shape, V.shape
```

```
[ ]: ((50, 4), (4,), (4, 4))
```

```
[ ]: #V is equivalent to the loading matrix from the PCA before!
     V
```

```
[ ]: array([[-0.53589947, -0.58318363, -0.27819087, -0.54343209],
            [-0.41818087, -0.1879856 ,  0.87280619,  0.16731864],
            [ 0.34123273,  0.26814843,  0.37801579, -0.81777791],
            [ 0.6492278 , -0.74340748,  0.13387773,  0.08902432]])
```

```
[ ]: pcaUS.components_
```

```
[ ]: array([[ 0.53589947,  0.58318363,  0.27819087,  0.54343209],
            [-0.41818087, -0.1879856 ,  0.87280619,  0.16731864],
            [-0.34123273, -0.26814843, -0.37801579,  0.81777791],
            [ 0.6492278 , -0.74340748,  0.13387773,  0.08902432]])
```

The matrix `U` is a standardized version of the PCA score matrix. The standardization involves scaling the columns to have sum-of-squares 1.

```
[ ]: (U * D[None,:])[:3]
```

```
[ ]: array([[-0.98556588, -1.13339238,  0.44426879,  0.15626714],
            [-1.95013775, -1.07321326, -2.04000333, -0.43858344],
            [-1.76316354,  0.74595678, -0.05478082, -0.83465292]])
```

```
[ ]: scores[:3]
```

```
[ ]: array([[ 0.98556588, -1.13339238, -0.44426879,  0.15626714],
            [ 1.95013775, -1.07321326,  2.04000333, -0.43858344],
            [ 1.76316354,  0.74595678,  0.05478082, -0.83465292]])
```

Note: this section of the lab would be able to be done by just using the `PCA()` estimator but we are interested in using `np.linalg.svd()` to help us explore how matrix completion works.

We purposely omit 50 random entries in our matrix before implementing the algorithm from 12.1 for matrix completion.

[ ]:
```
n_omit = 20
np.random.seed(15)
r_idx = np.random.choice(np.arange(X.shape[0]), n_omit ,replace=False)
c_idx = np.random.choice(np.arange(X.shape[1]), n_omit , replace=True)
Xna = X.copy()
Xna[r_idx, c_idx] = np.nan
```

This function allows us to take in a matrix and return its estimation from SVD.

[ ]:
```
def low_rank(X, M=1):
    U, D, V = np.linalg.svd(X)
    L = U[:,:M] * D[None,:M]
    return L.dot(V[:M])
```

Here we replace the missing values (the ones we purposely got rid of) with the means of the other entries in the column.

The matrix `imiss` is a logical matrix having the same dimensions of `Xna`.

Step 2A involved approximating `Xhat` with our function defined `low_rank()`. Set 2B we use `Xapp` to update the estimates for the missing data. In step 3C we finally compute the relative error.

[ ]:
```
Xhat = Xna.copy()
Xbar = np.nanmean(Xhat, axis=0)
Xhat[r_idx, c_idx] = Xbar[c_idx]
```

[ ]:
```
thresh = 1e-7
rel_err = 1
count = 0
ismiss = np.isnan(Xna)
mssold = np.mean(Xhat[~ismiss]**2)
mss0 = np.mean(Xna[~ismiss]**2)
```

This process actually only takes eight iterations as after that the relative error falls below `1e-7`.

[ ]:
```
while rel_err > thresh:
    count += 1
    # Step 2(a)
    Xapp = low_rank(Xhat, M=1)
    # Step 2(b)
    Xhat[ismiss] = Xapp[ismiss]
    # Step 2(c)
    mss = np.mean(((Xna - Xapp)[~ismiss])**2)
    rel_err = (mssold - mss) / mss0
    mssold = mss
    print("Iteration: {0}, MSS:{1:.3f}, Rel.Err {2:.2e}".format(count, mss,␣
  ↪rel_err))
```

```
Iteration: 1, MSS:0.395, Rel.Err 5.99e-01
Iteration: 2, MSS:0.382, Rel.Err 1.33e-02
Iteration: 3, MSS:0.381, Rel.Err 1.44e-03
Iteration: 4, MSS:0.381, Rel.Err 1.79e-04
Iteration: 5, MSS:0.381, Rel.Err 2.58e-05
Iteration: 6, MSS:0.381, Rel.Err 4.22e-06
Iteration: 7, MSS:0.381, Rel.Err 7.65e-07
Iteration: 8, MSS:0.381, Rel.Err 1.48e-07
Iteration: 9, MSS:0.381, Rel.Err 2.95e-08
```

```
[ ]: np.corrcoef(Xapp[ismiss], X[ismiss])[0,1]
```

```
[ ]: 0.7113567434297362
```

0.1.3 12.5.3 Clustering

K-Means Clustering We can use the estimator from the `sklearn` package to perform `K-means` clustering.

```
[ ]: np.random.seed(0);
     X = np.random.standard_normal((50,2)); X[:25,0] += 3;
     X[:25,1] -= 4;
```

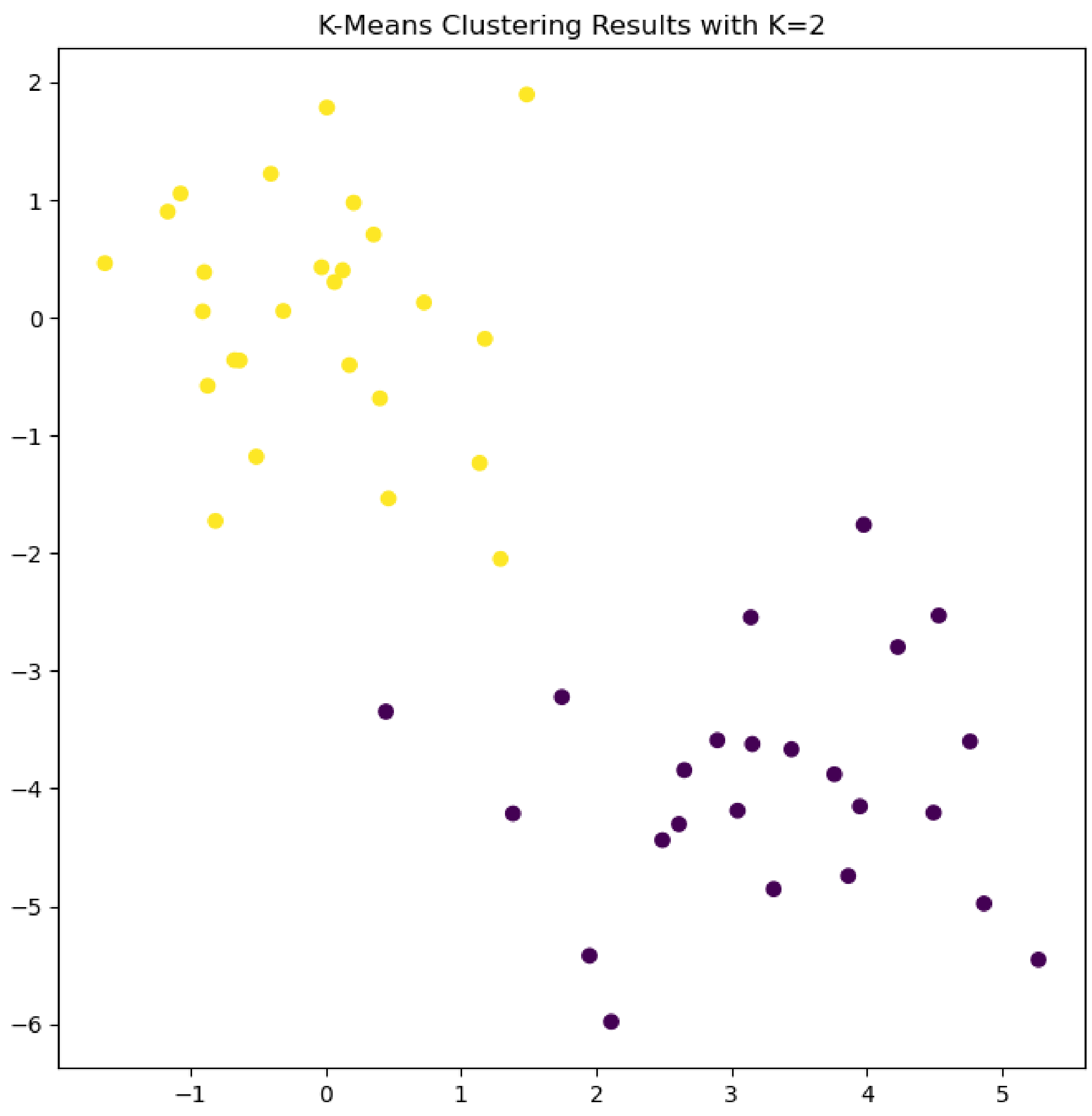
```
[ ]: # This performs k-means with 2 clusters
     kmeans = KMeans(n_clusters=2, random_state=2, n_init=20).fit(X)
     # Kmeans with 5 clusters
     kmeans_three_clsuters = KMeans(n_clusters=5, random_state=2, n_init=20).fit(X)
```

```
[ ]: # It looks like we were able to split teh data perfectly even though we did not␣
       ↪have any labels!
     kmeans.labels_
```

```
[ ]: array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,
            0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
            1, 1, 1, 1, 1, 1], dtype=int32)
```

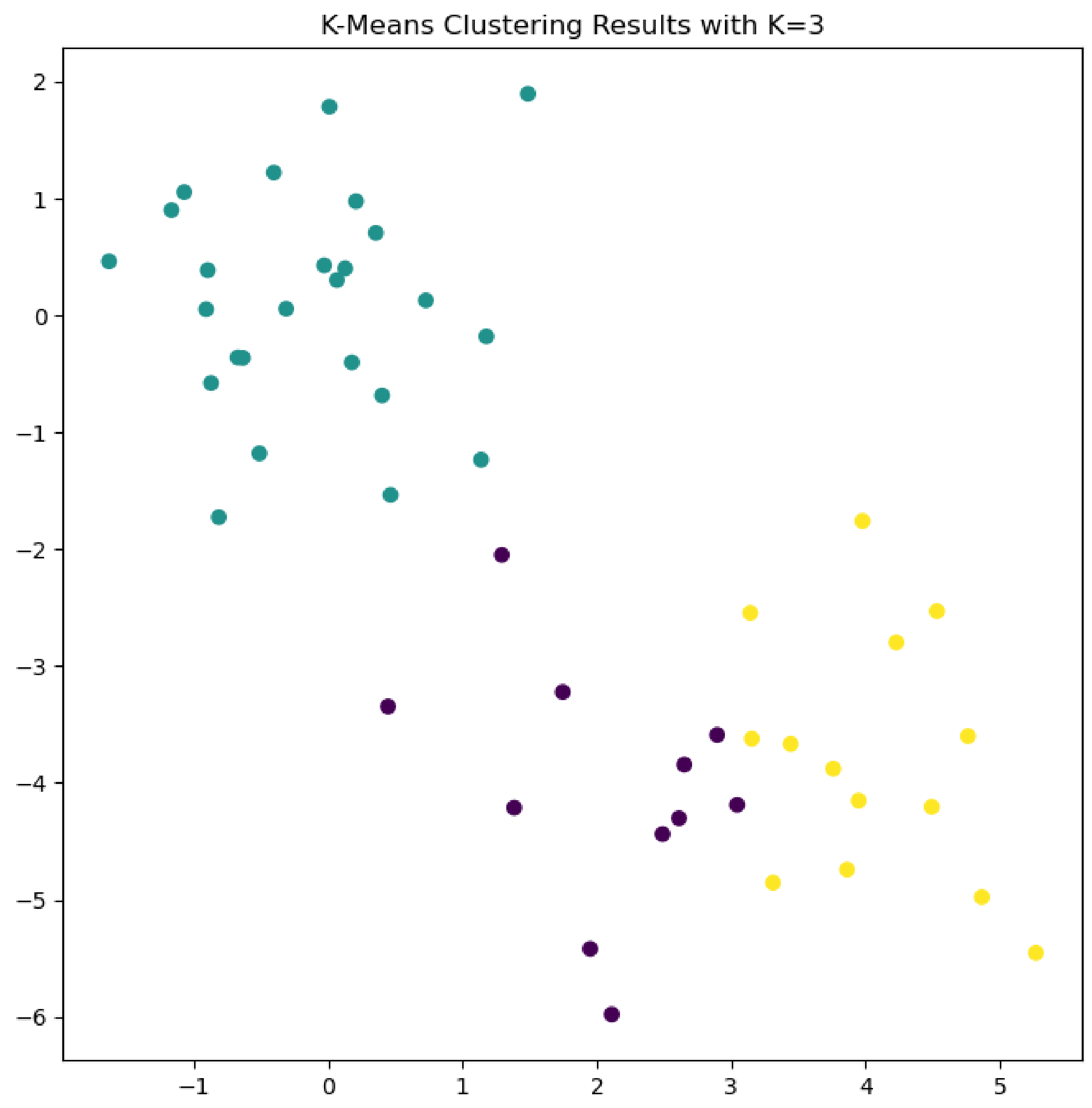
It is easy to plot the observations in the following cell because they are already two-dimensional.

```
[ ]: fig, ax = plt.subplots(1, 1, figsize=(8,8))
     ax.scatter(X[:,0], X[:,1], c=kmeans.labels_)
     ax.set_title("K-Means Clustering Results with K=2");
```



Next we try and cluster with K=3.

```
[ ]: kmeans = KMeans(n_clusters=3, random_state=3,
     n_init=20).fit(X)
     fig, ax = plt.subplots(figsize=(8,8))
     ax.scatter(X[:,0], X[:,1], c=kmeans.labels_)
     ax.set_title("K-Means Clustering Results with K=3");
```



[ ]:
```
kmeans1 = KMeans(n_clusters=3, random_state=3,
n_init=1).fit(X)
kmeans20 = KMeans(n_clusters=3, random_state=3,
n_init=20).fit(X);
kmeans1.inertia_, kmeans20.inertia_
```

[ ]:
```
(76.85131986999252, 75.06261242745384)
```

The inertia (which can be calculated with `.interti_` gives is the total within-cluster sum of squares. We seek to minimize this value with K-means clustering.

Some steps to take when we perform K-means are - Use random multiple inital cluster assignments - Use a random seed in the `random_state` argument for `Kmeans()` This is important so that you clan replicate the first step of the algorithm