Python Tutorial 12

April 13, 2022

This tutorial is for Dr. Xin Tong's DSO 530 class at the University of Southern California in spring 2022. It aims to (1) give you some supplementary code to implement *Principal Components Analysis* and *K-Means Clustering* using Python, and (2) talk about an interesting point that arises from HW2.

[1]:
```
import numpy as np
import sklearn
```

1 Unsupervised Learning

1.1 Principal Components Analysis

In this tutorial, we perform PCA on the *USArrests* data set. The rows of the data set contain the 50 states, in alphabetical order. For each of the 50 states in the United States, the data set contains the number of arrests per 100, 000 residents for each of three crimes: *Assault*, *Murder*, and *Rape*. The data set also record *UrbanPop* (the percentages of the population in each state living in urban areas).

[2]:
```
import pandas as pd

df = pd.read_csv('USArrests.csv', index_col=0)
df.head()
```

[2]:

	Murder	Assault	UrbanPop	Rape
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

[3]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 50 entries, Alabama to Wyoming
Data columns (total 4 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   Murder    50 non-null     float64
```

```
 1   Assault   50 non-null     int64
 2   UrbanPop  50 non-null     int64
 3   Rape      50 non-null     float64
dtypes: float64(2), int64(2)
memory usage: 2.0+ KB
```

We look at the column means of the data.

[4]:
```
df.mean()
```

[4]:
```
Murder        7.788
Assault     170.760
UrbanPop     65.540
Rape         21.232
dtype: float64
```

We see that the columns have vastly different means. We can also examine the variances of the four variables.

[5]:
```
df.var()
```

[5]:
```
Murder        18.970465
Assault     6945.165714
UrbanPop     209.518776
Rape          87.729159
dtype: float64
```

Not surprisingly, the variables also have vastly different variances: the *UrbanPop* variable measures the percentage of the population in each state living in an urban area, which is not a comparable number to the number of rapes in each state per 100,000 individuals. If we failed to scale the variables before performing PCA, then most of the principal components that we observed would be driven by the *Assault* variable, since it has a variance far greater than others.

Also, the means of the variables are not relevant to investigate the PC directions.

Thus, we standardize the variables to have mean zero and standard deviation one before performing PCA.

[6]:
```
from sklearn.preprocessing import StandardScaler
std = StandardScaler()
X = std.fit_transform(df)
print(X)
print(np.mean(X, axis = 0))
print(np.std(X, axis = 0))
```

```
[[ 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]]
[-7.10542736e-17  1.38777878e-16 -4.39648318e-16  8.59312621e-16]
[1. 1. 1. 1.]
```

Now we'll use the `fit()` function in `PCA()` model from `sklearn` to compute the loading vectors.

[7]:
```
from sklearn.decomposition import PCA

pca = PCA()
pca_loading_vectors = pd.DataFrame(pca.fit(X).components_.T, index=df.columns,␣
↪columns=['V1', 'V2', 'V3', 'V4'])
pca_loading_vectors
```

[7]:

	V1	V2	V3	V4
Murder	0.535899	0.418181	-0.341233	0.649228
Assault	0.583184	0.187986	-0.268148	-0.743407
UrbanPop	0.278191	-0.872806	-0.378016	0.133878
Rape	0.543432	-0.167319	0.817778	0.089024

We see that there are four distinct principal components. This is to be expected because we can compute a total of $\min(n-1, p)$ informative principal components in a data set with n observations and p variables.

Using the `fit_transform()` function, we can get the principal component scores of the original data. We'll take a look at the first few states:

[8]:
```
pca_scores = pd.DataFrame(pca.fit_transform(X), columns=['PC1', 'PC2', 'PC3',␣
↪'PC4'], index=df.index)
pca_scores.head()
```

[8]:

	PC1	PC2	PC3	PC4
Alabama	0.985566	1.133392	-0.444269	0.156267
Alaska	1.950138	1.073213	2.040003	-0.438583
Arizona	1.763164	-0.745957	0.054781	-0.834653
Arkansas	-0.141420	1.119797	0.114574	-0.182811
California	2.523980	-1.542934	0.598557	-0.341996

[9]:
```
pca_scores.shape
```

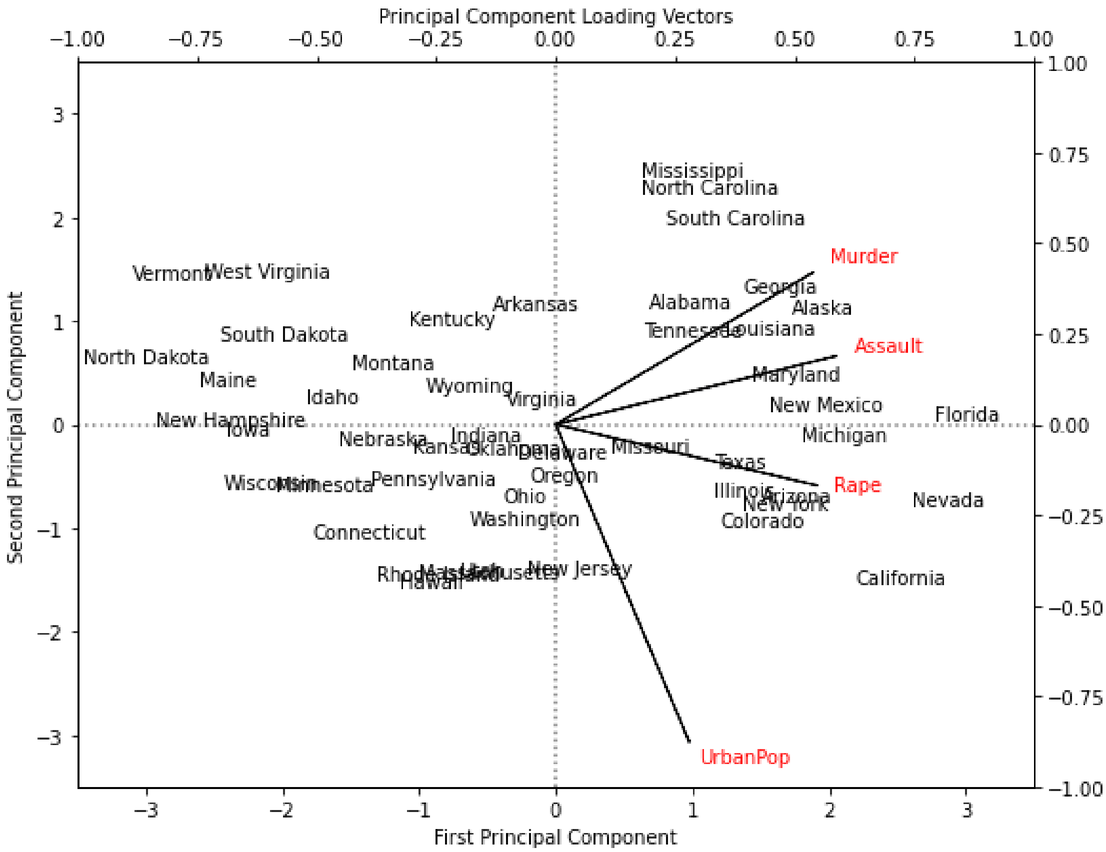
[9]:
```
(50, 4)
```

We can construct a **biplot**. (**optional, but helpful to understand**)

[10]:
```
import matplotlib.pyplot as plt

fig , ax1 = plt.subplots(figsize=(9,7))

ax1.set_xlim(-3.5,3.5)
ax1.set_ylim(-3.5,3.5)

# Plot Principal Components 1 and 2
for i in pca_scores.index:
    ax1.annotate(i, (pca_scores.PC1.loc[i], pca_scores.PC2.loc[i]), ha='center')
```

```
# Plot reference lines
ax1.hlines(0,-3.5,3.5, linestyles='dotted', colors='grey')
ax1.vlines(0,-3.5,3.5, linestyles='dotted', colors='grey')

ax1.set_xlabel('First Principal Component')
ax1.set_ylabel('Second Principal Component')

# Plot Principal Component loading vectors, using a second xy-axis.
ax2 = ax1.twinx().twiny()

ax2.set_ylim(-1,1)
ax2.set_xlim(-1,1)
ax2.set_xlabel('Principal Component Loading Vectors')

# Plot labels for vectors. Variable 'a' is a small offset parameter to separate␣
 ↪arrow tip and text.
a = 1.07
for i in pca_loading_vectors[['V1', 'V2']].index:
    ax2.annotate(i, (pca_loading_vectors.V1.loc[i]*a, pca_loading_vectors.V2.
 ↪loc[i]*a), color='red')

# Plot vectors
ax2.arrow(0,0,pca_loading_vectors.V1[0], pca_loading_vectors.V2[0])
ax2.arrow(0,0,pca_loading_vectors.V1[1], pca_loading_vectors.V2[1])
ax2.arrow(0,0,pca_loading_vectors.V1[2], pca_loading_vectors.V2[2])
ax2.arrow(0,0,pca_loading_vectors.V1[3], pca_loading_vectors.V2[3])
```

[10]: <matplotlib.patches.FancyArrow at 0x7fd3030a2a30>



The `PCA()` model also outputs the variance explained by each principal component. We can access these values in `explained_variance_`.

[11]:
```
pca.explained_variance_
```

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

We can also get the proportion of variance explained in `explained_variance_ratio_`.

[12]:
```
pca.explained_variance_ratio_
```

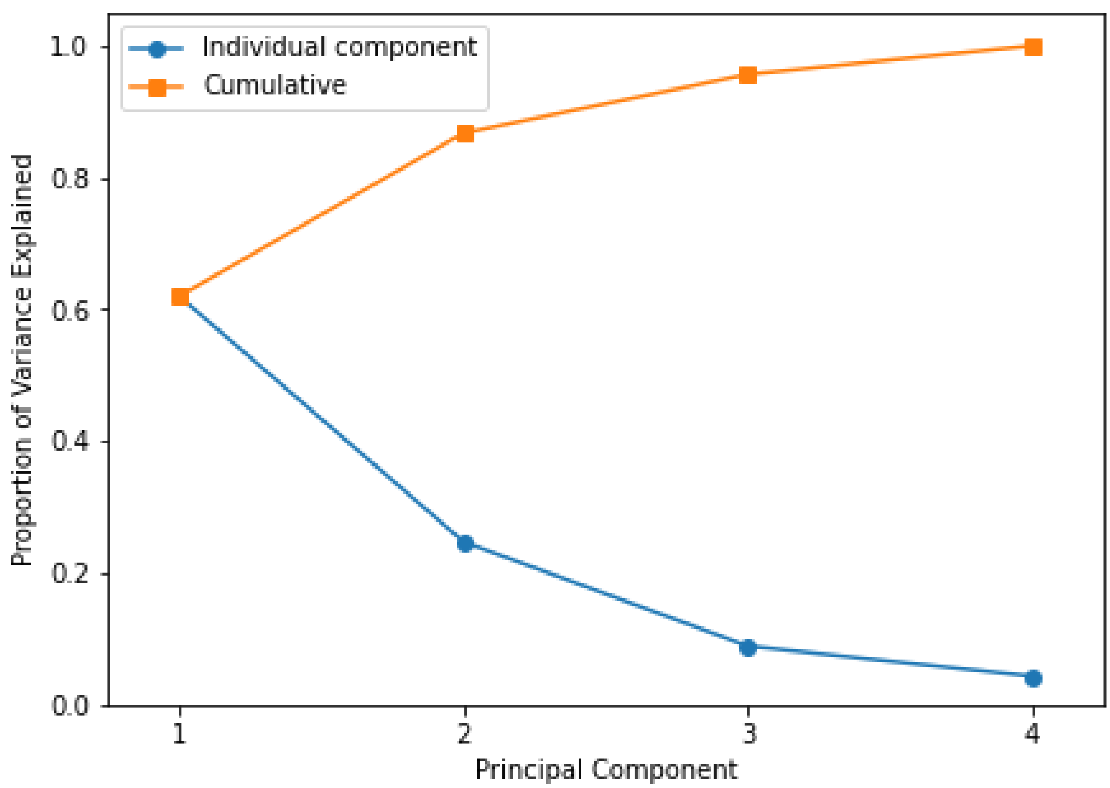
[12]: `array([0.62006039, 0.24744129, 0.0891408 , 0.04335752])`

The first principal component explains 62.0% of the variance in the data, the next principal component explains 24.7% of the variance, and so forth. We can plot the Proportion of Variance Explained (PVE) explained by each component and we can also use the function `cumsum()`, which computes the cumulative sum of the elements of a numeric vector, to plot the cumulative PVE.

[13]:
```
plt.figure(figsize=(7,5))
```

```
plt.plot([1,2,3,4], pca.explained_variance_ratio_, '-o', label='Individual␣
↪component')
plt.plot([1,2,3,4], np.cumsum(pca.explained_variance_ratio_), '-s',␣
↪label='Cumulative')

plt.ylabel('Proportion of Variance Explained')
plt.xlabel('Principal Component')
plt.xlim(0.75,4.25)
plt.ylim(0,1.05)
plt.xticks([1,2,3,4])
plt.legend(loc=2)
```

[13]: <matplotlib.legend.Legend at 0x7fd2e0a5fcd0>



1.2 K-Means Clustering

[14]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
```

The model KMeans in sklearn performs *K-means clustering* in Python. We begin with a simple simulated example in which there truly are two clusters in the data: the first 50 observations have a mean shift relative to the next 50 observations.

[15]:
```
# Generate data
np.random.seed(2)
X = np.random.standard_normal((100,2))
print(X)
```

```
[[-4.16757847e-01 -5.62668272e-02]
 [-2.13619610e+00  1.64027081e+00]
 [-1.79343559e+00 -8.41747366e-01]
 [ 5.02881417e-01 -1.24528809e+00]
 [-1.05795222e+00 -9.09007615e-01]
 [ 5.51454045e-01  2.29220801e+00]
 [ 4.15393930e-02 -1.11792545e+00]
 [ 5.39058321e-01 -5.96159700e-01]
 [-1.91304965e-02  1.17500122e+00]
 [-7.47870949e-01  9.02525097e-03]
 [-8.78107893e-01 -1.56434170e-01]
 [ 2.56570452e-01 -9.88779049e-01]
 [-3.38821966e-01 -2.36184031e-01]
 [-6.37655012e-01 -1.18761229e+00]
 [-1.42121723e+00 -1.53495196e-01]
 [-2.69056960e-01  2.23136679e+00]
 [-2.43476758e+00  1.12726505e-01]
 [ 3.70444537e-01  1.35963386e+00]
 [ 5.01857207e-01 -8.44213704e-01]
 [ 9.76147160e-06  5.42352572e-01]
 [-3.13508197e-01  7.71011738e-01]
 [-1.86809065e+00  1.73118467e+00]
 [ 1.46767801e+00 -3.35677339e-01]
 [ 6.11340780e-01  4.79705919e-02]
 [-8.29135289e-01  8.77102184e-02]
 [ 1.00036589e+00 -3.81092518e-01]
 [-3.75669423e-01 -7.44707629e-02]
 [ 4.33496330e-01  1.27837923e+00]
 [-6.34679305e-01  5.08396243e-01]
 [ 2.16116006e-01 -1.85861239e+00]
 [-4.19316482e-01 -1.32328898e-01]
 [-3.95702397e-02  3.26003433e-01]
 [-2.04032305e+00  4.62555231e-02]
 [-6.77675577e-01 -1.43943903e+00]
 [ 5.24296430e-01  7.35279576e-01]
 [-6.53250268e-01  8.42456282e-01]
 [-3.81516482e-01  6.64890091e-02]
 [-1.09873895e+00  1.58448706e+00]
 [-2.65944946e+00 -9.14526229e-02]
```

```
[ 6.95119605e-01 -2.03346655e+00]
[-1.89469265e-01 -7.72186654e-02]
[ 8.24703005e-01  1.24821292e+00]
[-4.03892269e-01 -1.38451867e+00]
[ 1.36723542e+00  1.21788563e+00]
[-4.62005348e-01  3.50888494e-01]
[ 3.81866234e-01  5.66275441e-01]
[ 2.04207979e-01  1.40669624e+00]
[-1.73795950e+00  1.04082395e+00]
[ 3.80471970e-01 -2.17135269e-01]
[ 1.17353150e+00 -2.34360319e+00]
[ 1.16152149e+00  3.86078048e-01]
[-1.13313327e+00  4.33092555e-01]
[-3.04086439e-01  2.58529487e+00]
[ 1.83533272e+00  4.40689872e-01]
[-7.19253841e-01 -5.83414595e-01]
[-3.25049628e-01 -5.60234506e-01]
[-9.02246068e-01 -5.90972275e-01]
[-2.76179492e-01 -5.16883894e-01]
[-6.98589950e-01 -9.28891925e-01]
[ 2.55043824e+00 -1.47317325e+00]
[-1.02141473e+00  4.32395701e-01]
[-3.23580070e-01  4.23824708e-01]
[ 7.99179995e-01  1.26261366e+00]
[ 7.51964849e-01 -9.93760983e-01]
[ 1.10914328e+00 -1.76491773e+00]
[-1.14421297e-01 -4.98174194e-01]
[-1.06079904e+00  5.91666521e-01]
[-1.83256574e-01  1.01985473e+00]
[-1.48246548e+00  8.46311892e-01]
[ 4.97940148e-01  1.26504175e-01]
[-1.41881055e+00 -2.51774118e-01]
[-1.54667461e+00 -2.08265194e+00]
[ 3.27974540e+00  9.70861320e-01]
[ 1.79259285e+00 -4.29013319e-01]
[ 6.96197980e-01  6.97416272e-01]
[ 6.01515814e-01  3.65949071e-03]
[-2.28247558e-01 -2.06961226e+00]
[ 6.10144086e-01  4.23496900e-01]
[ 1.11788673e+00 -2.74242089e-01]
[ 1.74181219e+00 -4.47500876e-01]
[-1.25542722e+00  9.38163671e-01]
[-4.68346260e-01 -1.25472031e+00]
[ 1.24823646e-01  7.56502143e-01]
[ 2.41439629e-01  4.97425649e-01]
[ 4.10869262e+00  8.21120877e-01]
[ 1.53176032e+00 -1.98584577e+00]
[ 3.65053516e-01  7.74082033e-01]
```

```
 [-3.64479092e-01 -8.75979478e-01]
 [ 3.96520159e-01 -3.14617436e-01]
 [-5.93755583e-01  1.14950057e+00]
 [ 1.33556617e+00  3.02629336e-01]
 [-4.54227855e-01  5.14370717e-01]
 [ 8.29458431e-01  6.30621967e-01]
 [-1.45336435e+00 -3.38017777e-01]
 [ 3.59133332e-01  6.22220414e-01]
 [ 9.60781945e-01  7.58370347e-01]
 [-1.13431848e+00 -7.07420888e-01]
 [-1.22142917e+00  1.80447664e+00]
 [ 1.80409807e-01  5.53164274e-01]
 [ 1.03302907e+00 -3.29002435e-01]]
```

[16]:

```python
X[:50,0] = X[:50,0]+3
X[:50,1] = X[:50,1]-4
print(X)
```

```
[[ 2.58324215e+00 -4.05626683e+00]
 [ 8.63803904e-01 -2.35972919e+00]
 [ 1.20656441e+00 -4.84174737e+00]
 [ 3.50288142e+00 -5.24528809e+00]
 [ 1.94204778e+00 -4.90900761e+00]
 [ 3.55145404e+00 -1.70779199e+00]
 [ 3.04153939e+00 -5.11792545e+00]
 [ 3.53905832e+00 -4.59615970e+00]
 [ 2.98086950e+00 -2.82499878e+00]
 [ 2.25212905e+00 -3.99097475e+00]
 [ 2.12189211e+00 -4.15643417e+00]
 [ 3.25657045e+00 -4.98877905e+00]
 [ 2.66117803e+00 -4.23618403e+00]
 [ 2.36234499e+00 -5.18761229e+00]
 [ 1.57878277e+00 -4.15349520e+00]
 [ 2.73094304e+00 -1.76863321e+00]
 [ 5.65232423e-01 -3.88727350e+00]
 [ 3.37044454e+00 -2.64036614e+00]
 [ 3.50185721e+00 -4.84421370e+00]
 [ 3.00000976e+00 -3.45764743e+00]
 [ 2.68649180e+00 -3.22898826e+00]
 [ 1.13190935e+00 -2.26881533e+00]
 [ 4.46767801e+00 -4.33567734e+00]
 [ 3.61134078e+00 -3.95202941e+00]
 [ 2.17086471e+00 -3.91228978e+00]
 [ 4.00036589e+00 -4.38109252e+00]
 [ 2.62433058e+00 -4.07447076e+00]
 [ 3.43349633e+00 -2.72162077e+00]
 [ 2.36532069e+00 -3.49160376e+00]
 [ 3.21611601e+00 -5.85861239e+00]
```

```
[ 2.58068352e+00 -4.13232890e+00]
[ 2.96042976e+00 -3.67399657e+00]
[ 9.59676951e-01 -3.95374448e+00]
[ 2.32232442e+00 -5.43943903e+00]
[ 3.52429643e+00 -3.26472042e+00]
[ 2.34674973e+00 -3.15754372e+00]
[ 2.61848352e+00 -3.93351099e+00]
[ 1.90126105e+00 -2.41551294e+00]
[ 3.40550544e-01 -4.09145262e+00]
[ 3.69511961e+00 -6.03346655e+00]
[ 2.81053074e+00 -4.07721867e+00]
[ 3.82470301e+00 -2.75178708e+00]
[ 2.59610773e+00 -5.38451867e+00]
[ 4.36723542e+00 -2.78211437e+00]
[ 2.53799465e+00 -3.64911151e+00]
[ 3.38186623e+00 -3.43372456e+00]
[ 3.20420798e+00 -2.59330376e+00]
[ 1.26204050e+00 -2.95917605e+00]
[ 3.38047197e+00 -4.21713527e+00]
[ 4.17353150e+00 -6.34360319e+00]
[ 1.16152149e+00  3.86078048e-01]
[-1.13313327e+00  4.33092555e-01]
[-3.04086439e-01  2.58529487e+00]
[ 1.83533272e+00  4.40689872e-01]
[-7.19253841e-01 -5.83414595e-01]
[-3.25049628e-01 -5.60234506e-01]
[-9.02246068e-01 -5.90972275e-01]
[-2.76179492e-01 -5.16883894e-01]
[-6.98589950e-01 -9.28891925e-01]
[ 2.55043824e+00 -1.47317325e+00]
[-1.02141473e+00  4.32395701e-01]
[-3.23580070e-01  4.23824708e-01]
[ 7.99179995e-01  1.26261366e+00]
[ 7.51964849e-01 -9.93760983e-01]
[ 1.10914328e+00 -1.76491773e+00]
[-1.14421297e-01 -4.98174194e-01]
[-1.06079904e+00  5.91666521e-01]
[-1.83256574e-01  1.01985473e+00]
[-1.48246548e+00  8.46311892e-01]
[ 4.97940148e-01  1.26504175e-01]
[-1.41881055e+00 -2.51774118e-01]
[-1.54667461e+00 -2.08265194e+00]
[ 3.27974540e+00  9.70861320e-01]
[ 1.79259285e+00 -4.29013319e-01]
[ 6.96197980e-01  6.97416272e-01]
[ 6.01515814e-01  3.65949071e-03]
[-2.28247558e-01 -2.06961226e+00]
[ 6.10144086e-01  4.23496900e-01]
```

```
 [ 1.11788673e+00 -2.74242089e-01]
 [ 1.74181219e+00 -4.47500876e-01]
 [-1.25542722e+00  9.38163671e-01]
 [-4.68346260e-01 -1.25472031e+00]
 [ 1.24823646e-01  7.56502143e-01]
 [ 2.41439629e-01  4.97425649e-01]
 [ 4.10869262e+00  8.21120877e-01]
 [ 1.53176032e+00 -1.98584577e+00]
 [ 3.65053516e-01  7.74082033e-01]
 [-3.64479092e-01 -8.75979478e-01]
 [ 3.96520159e-01 -3.14617436e-01]
 [-5.93755583e-01  1.14950057e+00]
 [ 1.33556617e+00  3.02629336e-01]
 [-4.54227855e-01  5.14370717e-01]
 [ 8.29458431e-01  6.30621967e-01]
 [-1.45336435e+00 -3.38017777e-01]
 [ 3.59133332e-01  6.22220414e-01]
 [ 9.60781945e-01  7.58370347e-01]
 [-1.13431848e+00 -7.07420888e-01]
 [-1.22142917e+00  1.80447664e+00]
 [ 1.80409807e-01  5.53164274e-01]
 [ 1.03302907e+00 -3.29002435e-01]]
```

Hence, the first 50 observations and the last 50 observations form two clusters.

We now perform *K-means clustering* with K=2.

[17]:
```
km1 = KMeans(n_clusters=2, random_state = 0)
km1.fit(X)
```

[17]:
```
KMeans(n_clusters=2, random_state=0)
```

The cluster assignments of the 100 observations are contained in `km.labels_`.

[18]:
```
km1.labels_
```

[18]:
```
array([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, 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, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 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, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], dtype=int32)
```

We are doing a great job here (two wrong). Note that since this is not supervised learning, if the labels are switched, it is equally good.

In this example, we knew that there really were two clusters because we generated the data. However, for real data, there might not be a "true" number of clusters. If we were to perform K-means clustering on this example with K=3, we will see the following results.

[19]:
```
km2 = KMeans(n_clusters=3, random_state = 0)
km2.fit(X)
```

[19]:
```
KMeans(n_clusters=3, random_state=0)
```

[20]:
```
km2.labels_
```

[20]:
```
array([0, 2, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 2,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 2, 1, 1, 2, 1, 1, 1, 1, 1, 2, 1, 1, 1, 2, 2, 1,
       1, 1, 1, 1, 1, 1, 2, 2, 1, 1, 1, 1, 2, 2, 1, 1, 1, 1, 2, 2, 1, 1,
       1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 2], dtype=int32)
```

[21]:
```
pd.Series(km2.labels_).value_counts()
```

[21]:
```
0    45
1    37
2    18
dtype: int64
```

We can check the centers of each cluster in `cluster_centers_`.

[22]:
```
km2.cluster_centers_
```

[22]:
```
array([[ 2.77621452, -4.11028123],
       [-0.32489076,  0.16950444],
       [ 1.8627143 , -0.84980887]])
```

We can access the sum of squared distances of instances to their closest cluster center in `inertia_`.

[23]:
```
km1.inertia_
```

[23]:
```
222.5463498742615
```

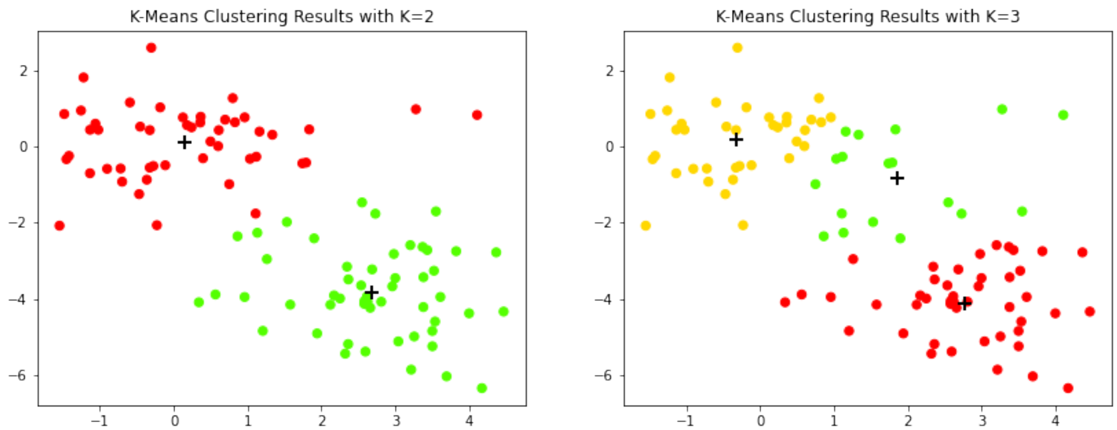
[24]:
```
km2.inertia_
```

[24]:
```
170.34499764084958
```

Now we will plot our results.

[25]:
```
fig, (ax1, ax2) = plt.subplots(1,2, figsize=(14,5))

ax1.scatter(X[:,0], X[:,1], s=40, c=km1.labels_, cmap=plt.cm.prism)
ax1.set_title('K-Means Clustering Results with K=2')
ax1.scatter(km1.cluster_centers_[:,0], km1.cluster_centers_[:,1], marker='+',␣
↪s=100, c='k', linewidth=2)

ax2.scatter(X[:,0], X[:,1], s=40, c=km2.labels_, cmap=plt.cm.prism)
```

```
ax2.set_title('K-Means Clustering Results with K=3')
ax2.scatter(km2.cluster_centers_[:,0], km2.cluster_centers_[:,1], marker='+',␣
 ↪s=100, c='k', linewidth=2);
```



Note that rescaling of features in the *k-means* algorithm might be problematic.

[26]:
```
from sklearn.preprocessing import StandardScaler
stdsc = StandardScaler()
X_std = stdsc.fit_transform(X)
km3 = KMeans(n_clusters=2, random_state = 0)
km3.fit(X_std)
km3.labels_
```

[26]:
```
array([0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 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, 0, 0, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1], dtype=int32)
```

2 About Question 2 in HW2

2.1 Random Seed and Default Value

One might have tried the following two blocks of code but wonder why the samples differ.

[27]:
```
np.random.seed(2)
color = ['red','black','blue','yellow']
for i in range(3):
    print(f'Sample {i+1}: {np.random.choice(a = color, size = 5, replace =␣
 ↪True, p = [0.25, 0.25, 0.25, 0.25])}')
```

```
Sample 1: ['black' 'red' 'blue' 'black' 'black']
Sample 2: ['black' 'red' 'blue' 'black' 'black']
Sample 3: ['blue' 'blue' 'red' 'blue' 'red']
```

[28]:
```
np.random.seed(2)
color = ['red','black','blue','yellow']
for i in range(3):
    print(f'Sample {i+1}: {np.random.choice(a = color, size = 5, replace =␣
 ↪True)}')
```

```
Sample 1: ['red' 'yellow' 'black' 'red' 'blue']
Sample 2: ['yellow' 'blue' 'yellow' 'red' 'yellow']
Sample 3: ['blue' 'black' 'yellow' 'yellow' 'black']
```

As the default option to assign equal probability to each element in the set "color", the above two blocks seem to be doing the same thing but produce different outcomes.

This has to do with how numpy handles the default values. They should be statistically identical, but it is not guaranteed that their implementation is the same. The default option may have a different implementation. And this phenomenon is not unique to this `np.random.choice` function.

2.2 Python Keyword Arguments

When we call a function that includes some specified values for its parameters, these values get assigned to the arguments according to their positions if we skip the key words.

We take Question 2 in HW2 as an example. `np.random.choice` has its default keywords' order: `numpy.random.choice(a, size=None, replace=True, p=None)`.

A standard method to call the `np.random.choice` function is using the following code.

[29]:
```
np.random.seed(2)
print(np.random.choice(a = color, size = 5, replace = True, p = [0.25, 0.25, 0.
 ↪25, 0.25]))
```

```
['black' 'red' 'blue' 'black' 'black']
```

But you can also omit the keywords to call the function like this:

[30]:
```
np.random.seed(2)
print(np.random.choice(color, 5, True, [0.25, 0.25, 0.25, 0.25]))
```

```
['black' 'red' 'blue' 'black' 'black']
```

These values get assigned to the arguments according to their positions.

The value `color` gets assigned to the argument `a` and similarly `5` to `size`, `True` to `replace`, $[0.25, 0.25, 0.25, 0.25]$ to `p`.

The following code appears in some answers to the quesion 2 of HW2.

[31]:
```
np.random.seed(2)
print(np.random.choice(color, 5, [0.25, 0.25, 0.25, 0.25]))
```

```
['red' 'yellow' 'black' 'red' 'blue']
```

Actually, the above code is not implementing what we want. $[0.25, 0.25, 0.25, 0.25]$ gets assigned to the argument `replace` and gets evaluated as `True`.

By default, an object is considered `True` unless its class defines either a `\__bool\__()` method that returns `False` or a `\__len\__()` method that returns zero, when called with the object. Here are most of the built-in objects considered `False`:

1) *constants defined to be false: None and False.*
2) *zero of any numeric type: 0, 0.0, 0j, Decimal(0), Fraction(0, 1)*
3) *empty sequences and collections: ”, (), [], {}, set(), range(0)*

You can validate it to check the following results.

[32]:
```python
np.random.seed(2)
print(np.random.choice(color, 5, [0.1, 0.1, 0.1, 0.7]))
```

```
['red' 'yellow' 'black' 'red' 'blue']
```

[33]:
```python
np.random.seed(2)
print(np.random.choice(color, 5, True))
```

```
['red' 'yellow' 'black' 'red' 'blue']
```

Therefore, a recommended practice is to keep the key words when you call the functions.

References:

https://github.com/jcrouser/islr-python/blob/master/Lab%2018%20-%20PCA%20in%20Python.ipynb

https://github.com/JWarmenhoven/ISLR-python/blob/master/Notebooks/Chapter%2010.ipynb