Python Tutorial 8

April 18, 2020

This tutorial is for Dr. Xin Tong's DSO 530 class at the University of Southern California in spring 2020. It aims to give you some supplementary code of Lecture 7 on how to implement *Decision Trees* and *Random Forest* and of Lecture 9 on how to implement *Principal Components Analysis* and *K-Means Clustering* using Python.

1 Lecture 7 Lab: Tree-Based Methods

1.1 Decision Trees

We only cover the classification tree in this tutorial. For regression trees, please check out the documentation of DecisionTreeRegressor at https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeRegressor.html.

[1]:
```
import numpy as np
import sklearn
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_breast_cancer
from sklearn.tree import DecisionTreeClassifier, export_graphviz
from sklearn.metrics import accuracy_score
print('The scikit-learn version is {}.'.format(sklearn.__version__))
```

```
The scikit-learn version is 0.22.1.
```

Note that you need to update *scikit-learng* to 0.22 or higher version.

We will use *load_breast_cancer* to import the *Breast Cancer Wisconsin (Diagnostic) Data Set* in this tutorial. The breast cancer dataset is a classic and very easy binary classification dataset.

Features are computed from a digitized image of a fine needle aspirate (FNA) of a breast mass. They describe the characteristics of the cell nuclei present in the image.

[2]:
```
data = load_breast_cancer()
data.feature_names
```

[2]:
```
array(['mean radius', 'mean texture', 'mean perimeter', 'mean area',
       'mean smoothness', 'mean compactness', 'mean concavity',
       'mean concave points', 'mean symmetry', 'mean fractal dimension',
       'radius error', 'texture error', 'perimeter error', 'area error',
       'smoothness error', 'compactness error', 'concavity error',
       'concave points error', 'symmetry error',
```

```
       'fractal dimension error', 'worst radius', 'worst texture',
       'worst perimeter', 'worst area', 'worst smoothness',
       'worst compactness', 'worst concavity', 'worst concave points',
       'worst symmetry', 'worst fractal dimension'], dtype='<U23')
```

[3]:
```
X, y = data.data, data.target
```

There are 569 instances in this dataset and each of them has 30 predictors.

[4]:
```
X.shape
```

[4]:
```
(569, 30)
```

y is a binary variable to represent two classes: *0* is *WDBC-Malignant* and *1* is *WDBC-Benign*.

[5]:
```
y[:40]
```

[5]:
```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0])
```

We use *train_test_split* function to split the dataset into training data and test data.

[6]:
```
 X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=2)
```

First, we use recursive binary splitting to grow a large tree on the training data, stopping only when each terminal node has fewer than some minimum number of observations. And we apply cost complexity pruning to this large tree to obtain a sequence of best subtrees, as a function of α.

We can use *min_samples_leaf* parameter to adjust the minimum number of observations required to be at a leaf node. The default value of *min_samples_leaf* is 1.

Please note that the parameter name *min_samples_leaf* involves an abuse of the term "sample".

[7]:
```
clf = DecisionTreeClassifier(random_state=0)
# Note that although the tree building process looks like a deterministic␣
 ↪process, inside the package,
# there is some heuristic iterative algorithm used, so setting a random_state␣
 ↪will make sure of reproducibility.

path = clf.cost_complexity_pruning_path(X_train, y_train)

ccp_alphas = path.ccp_alphas
```

ccp_alphas provides an array of alphas for subtree during pruning.

[8]:
```
print(ccp_alphas)
```

```
[0.         0.00232818 0.0068506  0.00730308 0.00985915 0.01533646
 0.02221077 0.02346023 0.02771098 0.33529903]
```

Then, we use K-fold cross-validation to choose α. Here, we use 10-fold CV to choose alpha on X_train y_train according to classification accuracy.

[9]:
```
from sklearn.model_selection import KFold
kfolds = KFold(n_splits = 10, shuffle = True, random_state = 1)# a random␣state␣
 ↪is set for reproducibility purpose

accuracies = []
for ccp_alpha in ccp_alphas:
    score_for_alpha = []
    for train_index, test_index in kfolds.split(X_train):
        clf = DecisionTreeClassifier(random_state=0, ccp_alpha=ccp_alpha)
        clf.fit(X_train[train_index], y_train[train_index])
        y_pred = clf.predict(X_train[test_index])
        score = accuracy_score(y_pred, y_train[test_index])
        score_for_alpha.append(score)
    accuracies.append(sum(score_for_alpha)/len(score_for_alpha))
```

We can get the accuracies corresponding to the *ccp_alphas* and we can pick the best α to minimize the average error.

[10]:
```
print("The accuracies: ",accuracies)
print("\nThe index corresponding to the maximum of the accuracies: ",np.
 ↪argmax(accuracies))
```

```
The accuracies:  [0.9387596899224807, 0.9387596899224807, 0.9436323366555925,
0.9436323366555925, 0.9388704318936878, 0.9225359911406423, 0.8991140642303433,
0.9014396456256921, 0.9038759689922481, 0.7767441860465116]

The index corresponding to the maximum of the accuracies:  2
```

At last, we use the selected alpha to retrain a tree on the entire X_train and y_train and evaluate on the X_test and y_test.

[11]:
```
# Use the selected alpha to retrain a tree on the entire X_train and y_train
alpha_cv = ccp_alphas[np.argmax(accuracies)]
clf_final = DecisionTreeClassifier(random_state=0, ccp_alpha=alpha_cv)
clf_final.fit(X_train, y_train)

# Evaluate on the X_test and y_test
y_pred_test = clf_final.predict(X_test)
score_test = accuracy_score(y_test, y_pred_test)
print(score_test)
```

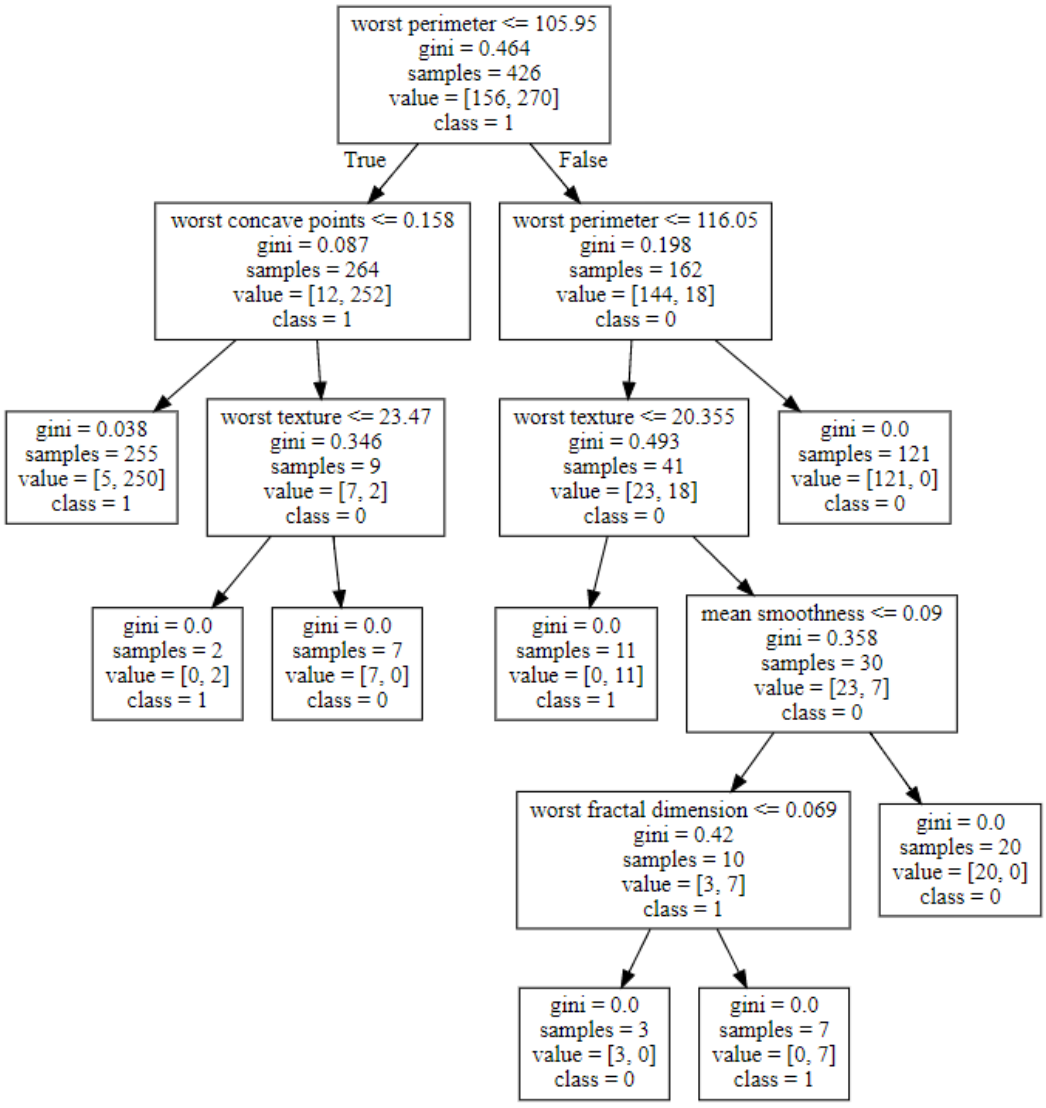
```
0.9370629370629371
```

Then we can use *graphviz* package to show the tree.

Note that you have to install the *graphviz* package according to the instructions on this website: https://graphviz.readthedocs.io/en/stable/manual.html#installation.

[12]:
```
from graphviz import Source

graph = Source(export_graphviz(clf_final,out_file=None, class_names=['0', '1'],␣
 ↪feature_names = data.feature_names))
display(graph)
```



1.2 Random Forest

Again, we only cover the *RandomForestClassifier* in this tutorial. For *RandomForestRegressor*, please check out the documentation of RandomForestRegressor at https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html.

Note that *RandomForestRegressor* in *sklearn* has a default setting of *max_features* = n_features. This is at odds with the recommendation by *ISLR*.

[13]:
```
from sklearn.ensemble import RandomForestClassifier
```

Parameters of *RandomForestClassifier*:

n_estimators (default 100) is the number of trees in the forest; *max_features* (default sqrt(n_features)) is the number of features to consider when looking for the best split.

[14]:
```
clf_rf = RandomForestClassifier(random_state=1, n_estimators = 200)
clf_rf.fit(X_train, y_train)

y_pred_rf = clf_rf.predict(X_test)
score_test_rf = accuracy_score(y_test, y_pred_rf)
print(score_test_rf)
```

```
0.951048951048951
```

We can see that if we use *random forest* and set the number of trees in the forest to 200, then we can improve the score the overall classification accuracy to 0.951 from 0.937, which we got from a single decision tree model.

2 Lecture 9 Lab: Unsupervised Learning

2.1 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.

We illustrate the use of PCA on the *USArrests* data set. 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 percent of the population in each state living in urban areas).

[15]:
```
import pandas as pd

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

[15]:

	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

[16]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 50 entries, Alabama to Wyoming
Data columns (total 4 columns):
Murder      50 non-null float64
Assault     50 non-null int64
UrbanPop    50 non-null int64
```

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

Let's start by taking a quick look at the column means of the data.

[17]:
```
df.mean()
```

[17]:
```
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.

[18]:
```
df.var()
```

[18]:
```
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.

[19]:
```
from sklearn.preprocessing import scale
X = pd.DataFrame(scale(df), index=df.index, columns=df.columns)
```

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

[20]:
```
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
```

[20]:

	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:

[21]:
```
pca_scores = pd.DataFrame(pca.fit_transform(X), columns=['PC1', 'PC2', 'PC3',␣
↪'PC4'], index=X.index)
pca_scores.head()
```

[21]:

	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

[22]:
```
pca_scores.shape
```

[22]:
```
(50, 4)
```

We can construct a **biplot** of the first two principal components using our loading vectors. (**optional, but helpful to understand**)

[23]:
```
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.
```

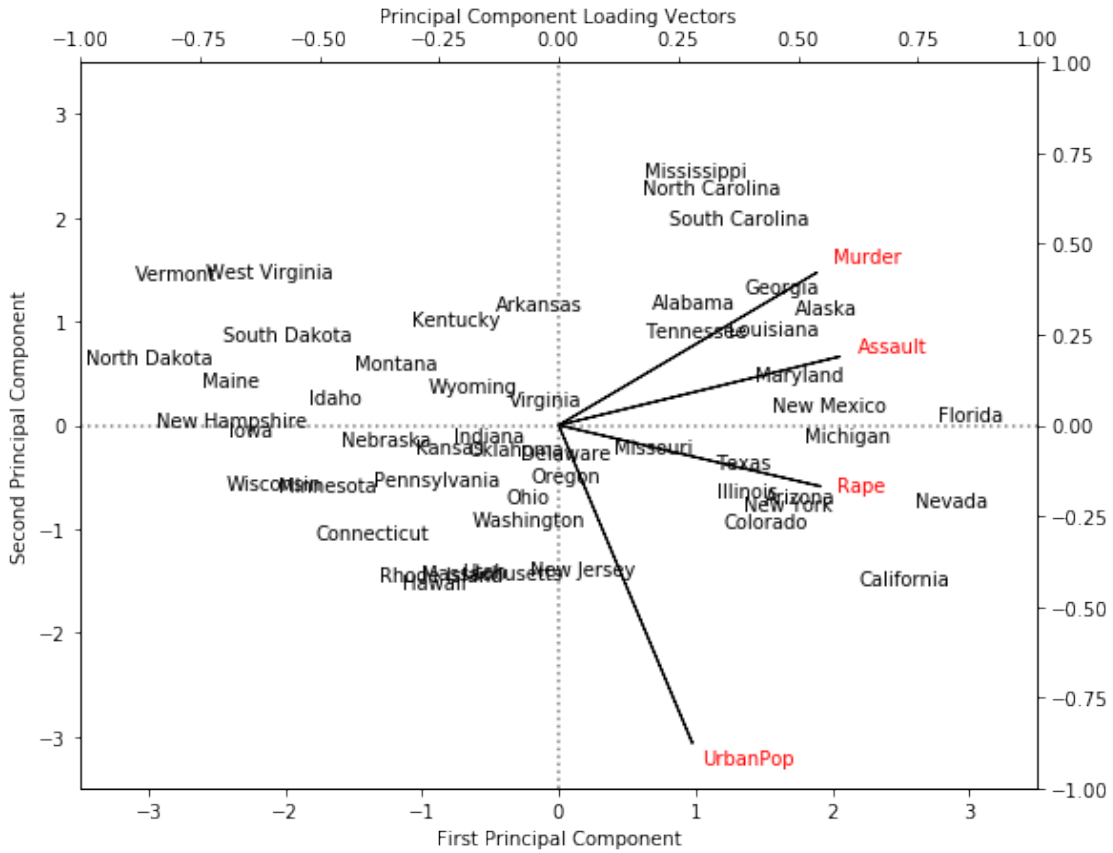
```
ax2 = ax1.twinx().twiny()

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])
```

[23]: <matplotlib.patches.FancyArrow at 0x1c9246f36c8>



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

[24]:
```
pca.explained_variance_
```

[24]:
```
array([2.53085875, 1.00996444, 0.36383998, 0.17696948])
```

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

[25]:
```
pca.explained_variance_ratio_
```

[25]:
```
array([0.62006039, 0.24744129, 0.0891408 , 0.04335752])
```

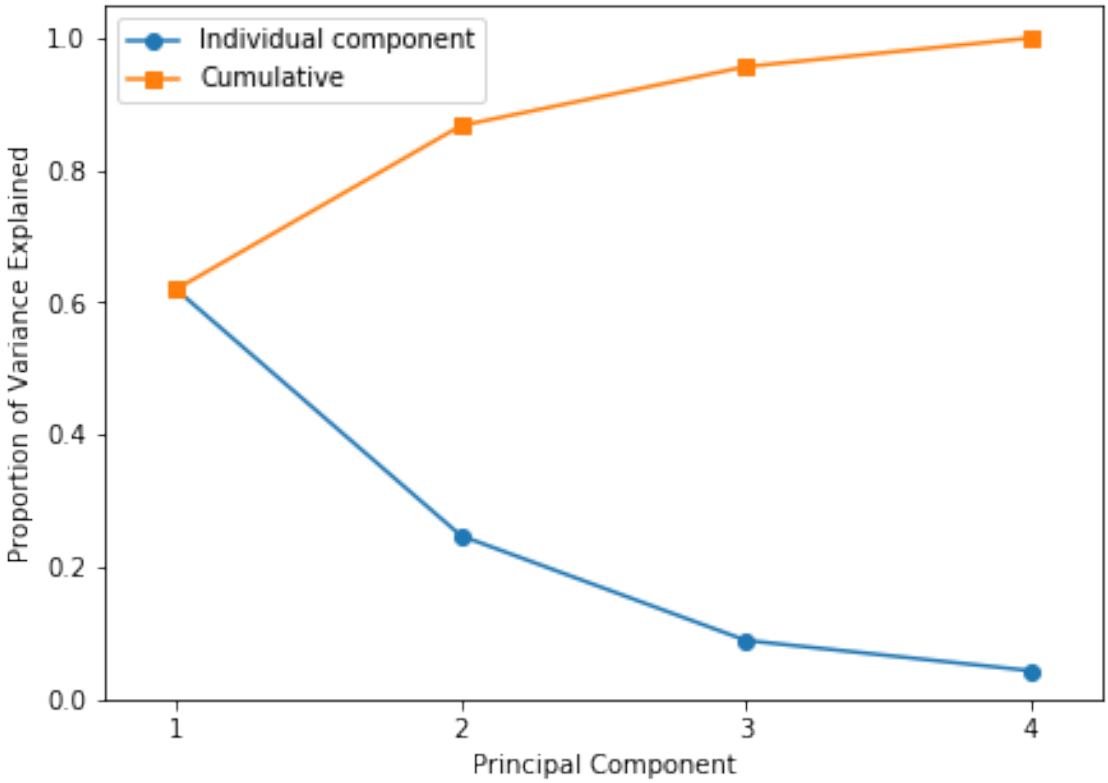
The first principal component explains 62.0% of the variance in the data, the next principal component explains 24.7% of the variance, and so forth. We can plot the Proportion of Variance Explained (PVE) explained by each component and we can also use the function *cumsum()*, which computes the cumulative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

[26]:
```
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)
```

[26]:
```
<matplotlib.legend.Legend at 0x1c922e61688>
```



2.2 K-Means Clustering

[27]:
```
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 25 observations have a mean shift relative to the next 25 observations.

[28]:
```
# Generate data
np.random.seed(2)
X = np.random.standard_normal((50,2))
X[:25,0] = X[:25,0]+3
X[:25,1] = X[:25,1]-4
```

Hence, the first 25 observations and the last 25 observations form two clusters.

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

[29]:
```
km1 = KMeans(n_clusters=2, random_state = 0)
km1.fit(X)
```

[29]:
```
KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300,
       n_clusters=2, n_init=10, n_jobs=None, precompute_distances='auto',
       random_state=0, tol=0.0001, verbose=0)
```

The cluster assignments of the 50 observations are contained in *km.labels_*.

[30]:
```
km1.labels_
```

[30]:
```
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, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 1])
```

We are doing a good job here. We only misassign one observation to the wrong cluster.

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.

[31]:
```
km2 = KMeans(n_clusters=3, random_state = 0)
km2.fit(X)
```

[31]:
```
KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300,
       n_clusters=3, n_init=10, n_jobs=None, precompute_distances='auto',
       random_state=0, tol=0.0001, verbose=0)
```

[32]:
```
km2.labels_
```

[32]:
```
array([1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 1, 1, 1, 1, 2,
       1, 1, 1, 0, 0, 0, 0, 2, 0, 0, 0, 2, 0, 0, 0, 0, 0, 2, 0, 0, 2, 0,
       0, 0, 0, 0, 0, 2])
```

[33]:
```
pd.Series(km2.labels_).value_counts()
```

[33]:
```
1    21
0    20
2     9
dtype: int64
```

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

[34]:
```
km2.cluster_centers_
```

[34]:
```
array([[-0.27876523,  0.51224152],
       [ 2.82805911, -4.11351797],
       [ 0.69945422, -2.14934345]])
```

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

[35]:
```
km1.inertia_
```

[35]:
```
99.30578397914685
```

[36]:
```
km2.inertia_
```

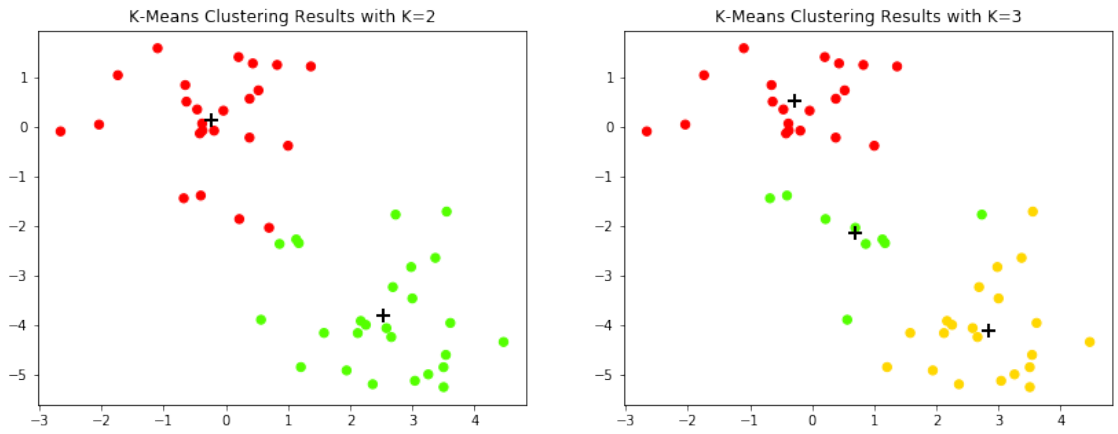
[36]:
```
68.97379200939726
```

Now we will plot our results.

[37]:
```
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);
```



References:

https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html

https://scikit-learn.org/stable/datasets/index.html#breast-cancer-dataset

https://graphviz.readthedocs.io/en/stable/manual.html#installation

https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html

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