index

April 19, 2022

1 Principal Component Analysis in scikit-learn - Lab

1.1 Introduction

Now that you've seen a brief introduction to PCA, it's time to use scikit-learn to run PCA on your own.

1.2 Objectives

In this lab you will:

- Implement PCA using the scikit-learn library
- Determine the optimal number of n components when performing PCA by observing the explained variance
- Plot the decision boundary of classification experiments to visually inspect their performance

1.3 Iris dataset

To practice PCA, you'll take a look at the iris dataset. Run the cell below to load it.

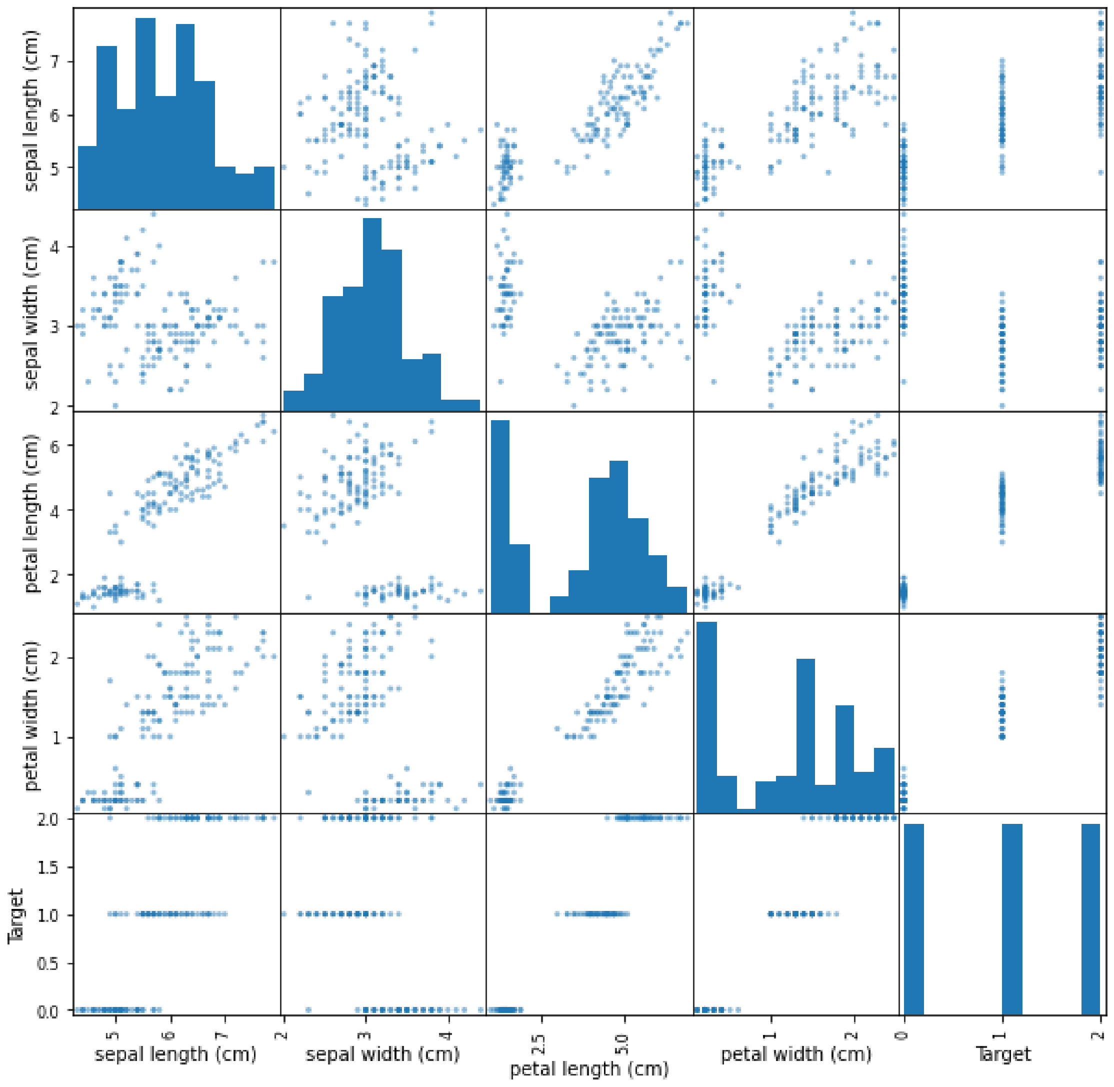
[1]:
```
from sklearn import datasets
import pandas as pd

iris = datasets.load_iris()
df = pd.DataFrame(iris.data, columns=iris.feature_names)
df['Target'] = iris.get('target')
df.head()
```

[1]:
```
   sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)  \
0                5.1               3.5                1.4               0.2
1                4.9               3.0                1.4               0.2
2                4.7               3.2                1.3               0.2
3                4.6               3.1                1.5               0.2
4                5.0               3.6                1.4               0.2

   Target
0       0
1       0
2       0
3       0
```

```
4         0
```

Before performing PCA and visualizing the principal components, it's helpful to get a little more context regarding the data that you'll be working with. Run the cell below in order to visualize the pairwise feature plots. With this, notice how the target labels are easily separable by any one of the given features.

[2]:
```
import matplotlib.pyplot as plt
%matplotlib inline

pd.plotting.scatter_matrix(df, figsize=(10,10));
```



- Assign all columns in the following `features` list to `X`
- Assign the `'Target'` column to `y`

```
[8]: # Create features and target datasets
     features = ['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)',␣
      ↪'petal width (cm)']
     X = df[features]
     y = df["Target"]
```

Standardize all the columns in `X` using `StandardScaler`.

```
[9]: # Import StandardScaler
     from sklearn.preprocessing import StandardScaler
     scale = StandardScaler()
     # Standardize the features
     X = scale.fit_transform(X)

     # Preview X
     pd.DataFrame(data=X, columns=features).head()
```

```
[9]:    sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)
     0          -0.900681          1.019004          -1.340227         -1.315444
     1          -1.143017         -0.131979          -1.340227         -1.315444
     2          -1.385353          0.328414          -1.397064         -1.315444
     3          -1.506521          0.098217          -1.283389         -1.315444
     4          -1.021849          1.249201          -1.340227         -1.315444
```

1.4 PCA Projection to 2-D Space

Now its time to perform PCA! Project the original data which is 4 dimensional into 2 dimensions. The new components are just the two main dimensions of variance present in the data.

- Initialize an instance of PCA from scikit-learn with two components
- Fit the data to the model
- Extract the first two principal components from the trained model

```
[12]: # Import PCA
      from sklearn.decomposition import PCA

      # Instantiate PCA
      pca = PCA(n_components = 2)

      # Fit PCA
      principalComponents = pca.fit_transform(X)
```

To visualize the components, it will be useful to also look at the target associated with the particular observation. As such, append the target (flower type) to the principal components in a pandas dataframe.

```
[22]: # Create a new dataset from principal components
      data = pd.DataFrame(principalComponents, columns = ["PC1", "PC2"])
```

```
data["Target"] = df["Target"]
data.head()



### G
# # Create a new dataset from principal components
# df = pd.DataFrame(data = principalComponents,
#                   columns = ['PC1', 'PC2'])

# target = pd.Series(iris['target'], name='target')

# result_df = pd.concat([df, target], axis=1)
# result_df.head(5)
```

[22]:
```
        PC1       PC2  Target
0 -2.264703  0.480027       0
1 -2.080961 -0.674134       0
2 -2.364229 -0.341908       0
3 -2.299384 -0.597395       0
4 -2.389842  0.646835       0
```

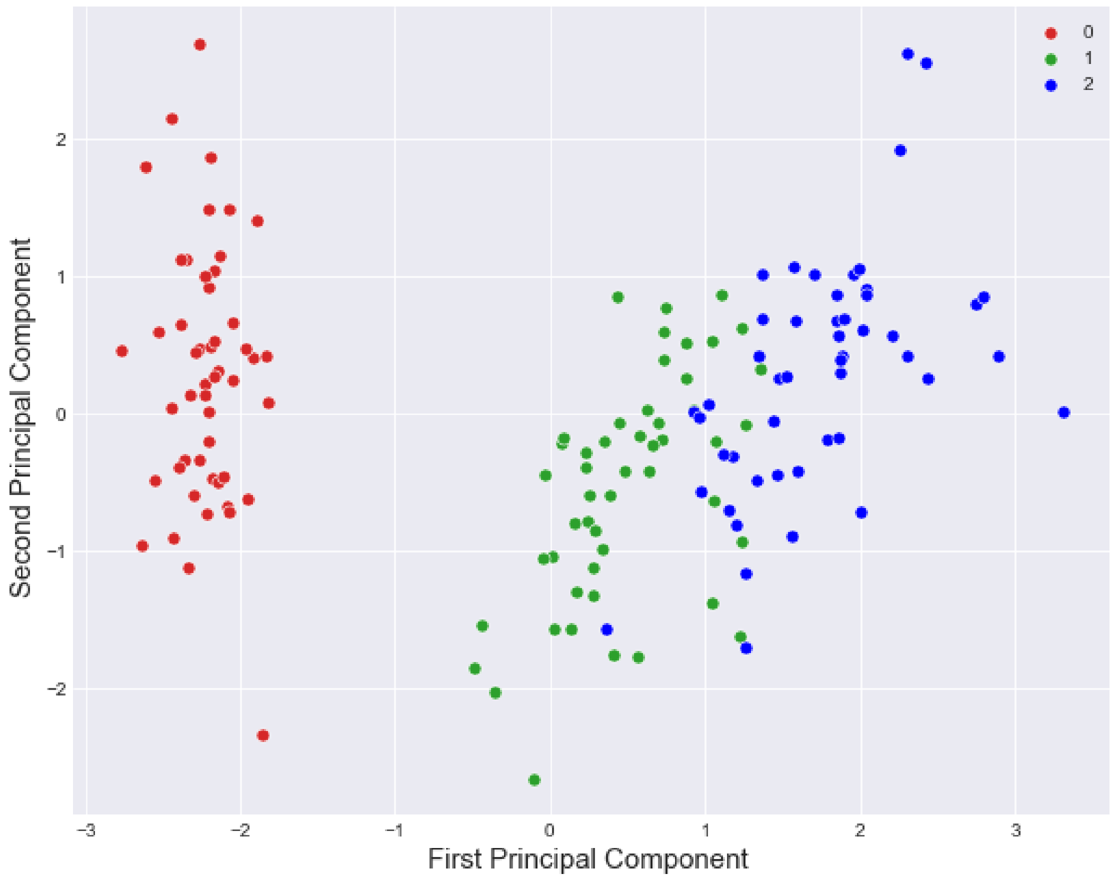
Great, you now have a set of two dimensions, reduced from four against our target variable, the flower type.

1.5 Visualize Principal Components

Using the target data, we can visualize the principal components according to the class distribution. - Create a scatter plot from principal components while color coding the examples according to what flower type each example is classified as

[36]:
```
# Principal Componets scatter plot
import seaborn as sns

plt.style.use('seaborn-dark')
fig = plt.figure(figsize = (10,8))




colors = ["tab:red", "tab:green", "blue"]
targets = data["Target"].unique()

for i, col in enumerate(list(data.columns)):
    PC1 = data.loc[data["Target"] == targets[i], "PC1"]
    PC2 = data.loc[data["Target"] == targets[i], "PC2"]
    g = sns.scatterplot(x = PC1, y = PC2,
                        color = colors[i], s = 50, label = targets[i])
    g.set_xlabel('First Principal Component ', fontsize = 15)
    g.set_ylabel('Second Principal Component ', fontsize = 15)
```

```
plt.grid()
# Your code here
```



1.6 Explained Variance

You can see above that the three classes in the dataset are fairly well separable. As such, this compressed representation of the data is probably sufficient for the classification task at hand. Compare the variance in the overall dataset to what was captured from your two primary components.

[63]:
```
# Calculate the variance explained by pricipal components
print('Variance of each component:', pca.explained_variance_ratio_)
print('\n Total Variance Explained:',
      np.round(100 * np.sum(list(pca.explained_variance_ratio_)),2))
```

```
Variance of each component: [0.72962445 0.22850762]

 Total Variance Explained: 95.81
```

As you should see, these first two principal components account for the vast majority of the overall variance in the dataset. This is indicative of the total information encapsulated in the compressed representation compared to the original encoding.

1.7 Compare Performance of a Classifier with PCA

Since the principal components explain 95% of the variance in the data, it is interesting to consider how a classifier trained on the compressed version would compare to one trained on the original dataset.

- Run a `KNeighborsClassifier` to classify the Iris dataset
- Use a train/test split of 80/20
- For the reproducibility of results, set `random_state=9` for the split
- Time the process for splitting, training and making predictions

[69]:
```
# Classification - complete Iris dataset
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
from sklearn import metrics
import time
# Your code here

X = iris.data
y = iris.target

start = time.time()
X_train, X_test, Y_train, Y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=9)
model = KNeighborsClassifier()
model.fit(X_train, Y_train)
Yhat = model.predict(X_test)
acc = metrics.accuracy_score(Yhat, Y_test)
end = time.time()

print('Accuracy:', acc)
print ('Time Taken:', end - start)
```

```
Accuracy: 1.0
Time Taken: 0.005226850509643555
```

Great, so you can see that we are able to classify the data with 100% accuracy in the given time. Remember the time taken may be different based on the load on your CPU and number of processes running on your machine.

Now repeat the above process for the dataset made from principal components:

- Run a `KNeighborsClassifier` to classify the Iris dataset with principal components
- Use a train/test split of 80/20
- For the reproducibility of results, set `random_state=9` for the split
- Time the process for splitting, training and making predictions

[70]:
```python
# Classification - reduced (PCA) Iris dataset

from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
from sklearn import metrics
import time
# Your code here

Xp = data.drop("Target", axis = 1)
yp = data["Target"]

start = time.time()
X_train, X_test, Y_train, Y_test = train_test_split(Xp, yp, test_size=0.2,␣
 ↪random_state=9)
model = KNeighborsClassifier()
model.fit(X_train, Y_train)
Yhat = model.predict(X_test)
acc = metrics.accuracy_score(Yhat, Y_test)
end = time.time()

print('Accuracy:', acc)
print ('Time Taken:', end - start)
```

```
Accuracy: 0.9666666666666667
Time Taken: 0.008942842483520508
```

Although some accuracy is lost in this representation of the data, we were able to use half of the number of features to train the model!

In more complex cases, PCA can even improve the accuracy of some machine learning tasks. In particular, PCA can be useful to reduce overfitting.

1.8 Visualize the Learned Decision Boundary

Run the cell below to visualize the decision boundary learned by the k-nearest neighbor classification model trained using the principal components of the data.

[71]:
```python
# Plot decision boundary using principal components
import numpy as np
def decision_boundary(pred_func):

    # Set the boundary
    x_min, x_max = X.iloc[:, 0].min() - 0.5, X.iloc[:, 0].max() + 0.5
    y_min, y_max = X.iloc[:, 1].min() - 0.5, X.iloc[:, 1].max() + 0.5
    h = 0.01

    # Build meshgrid
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
    Z = pred_func(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    # Plot the contour
    plt.figure(figsize=(15,10))
    plt.contourf(xx, yy, Z, cmap=plt.cm.afmhot)
    plt.scatter(X.iloc[:, 0], X.iloc[:, 1], c=y, cmap=plt.cm.Spectral,␣
 ↪marker='x')

decision_boundary(lambda x: model.predict(x))

plt.title('decision boundary');
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-71-d3255eb5c8c8> in <module>
     18     plt.scatter(X.iloc[:, 0], X.iloc[:, 1], c=y, cmap=plt.cm.Spectral,␣
 ↪marker='x')
     19
---> 20 decision_boundary(lambda x: model.predict(x))
     21
     22 plt.title('decision boundary');

<ipython-input-71-d3255eb5c8c8> in decision_boundary(pred_func)
      4
      5     # Set the boundary
----> 6     x_min, x_max = X.iloc[:, 0].min() - 0.5, X.iloc[:, 0].max() + 0.5
      7     y_min, y_max = X.iloc[:, 1].min() - 0.5, X.iloc[:, 1].max() + 0.5
      8     h = 0.01

AttributeError: 'numpy.ndarray' object has no attribute 'iloc'
```

1.9 Summary

In this lab, you applied PCA to the popular Iris dataset. You looked at the performance of a simple classifier and the impact of PCA on the accuracy of the model and the time it took to run the model. From here, you'll continue to explore PCA at more fundamental levels.