index

April 19, 2022

1 Principal Component Analysis in scikit-learn

1.1 Introduction

Now that you've seen the curse of dimensionality, it's time to take a look at a dimensionality reduction technique! This will help you overcome the challenges of the curse of dimensionality (amongst other things). Essentially, PCA, or Principal Component Analysis, attempts to capture as much information from the dataset as possible while reducing the overall number of features.

1.2 Objectives

You will be able to:

- Explain at a high level how PCA works
- Explain use cases for PCA
- Implement PCA using the scikit-learn library
- Determine the optimal number of n components when performing PCA by observing the explained variance

1.3 Generate some data

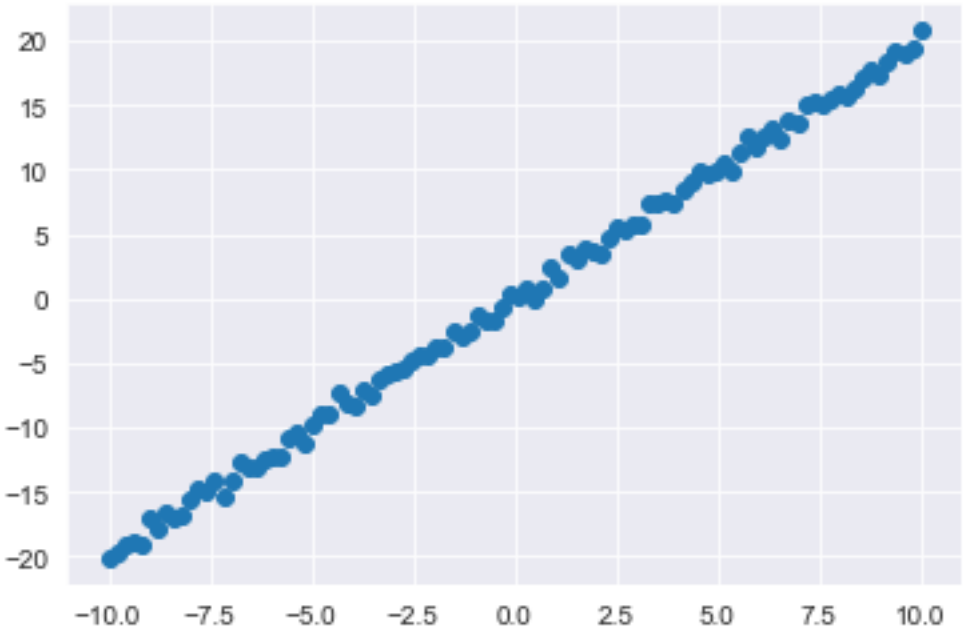
First, you need some data to perform PCA on. With that, here's a quick dataset you can generate using NumPy:

[27]:
```
import numpy as np

x1 = np.linspace(-10, 10, 100)
# A linear relationship, plus a little noise
x2 = np.array([xi*2 + np.random.normal(loc=0, scale=0.5) for xi in x1])

X = np.matrix(list(zip(x1, x2)))
```

Let's also generate a quick plot of this simple dataset to further orient ourselves:

[28]:
```
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline

sns.set_style('darkgrid')
```

```
plt.scatter(x1, x2);
```



1.4 PCA with scikit-learn

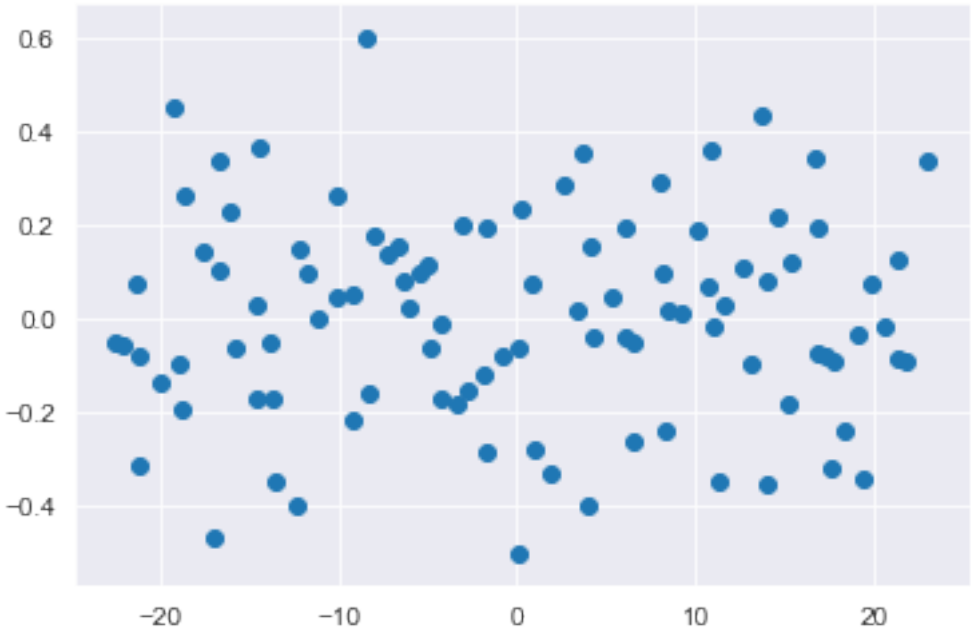
Now onto PCA. First, take a look at how simple it is to implement PCA with scikit-learn:

[29]:
```
from sklearn.decomposition import PCA

pca = PCA()
transformed = pca.fit_transform(X)
```

```
/opt/anaconda3/envs/learn-env/lib/python3.8/site-
packages/sklearn/utils/validation.py:593: FutureWarning: np.matrix usage is
deprecated in 1.0 and will raise a TypeError in 1.2. Please convert to a numpy
array with np.asarray. For more information see:
https://numpy.org/doc/stable/reference/generated/numpy.matrix.html
  warnings.warn(
```

And you can once again plot the updated dataset:

[30]:
```
plt.scatter(transformed[:,0], transformed[:,1]);
```



[31]:
```
pca.components_
```

[31]:
```
array([[-0.44628564, -0.89489057],
       [ 0.89489057, -0.44628564]])
```

[32]:
```
pca.mean_
```

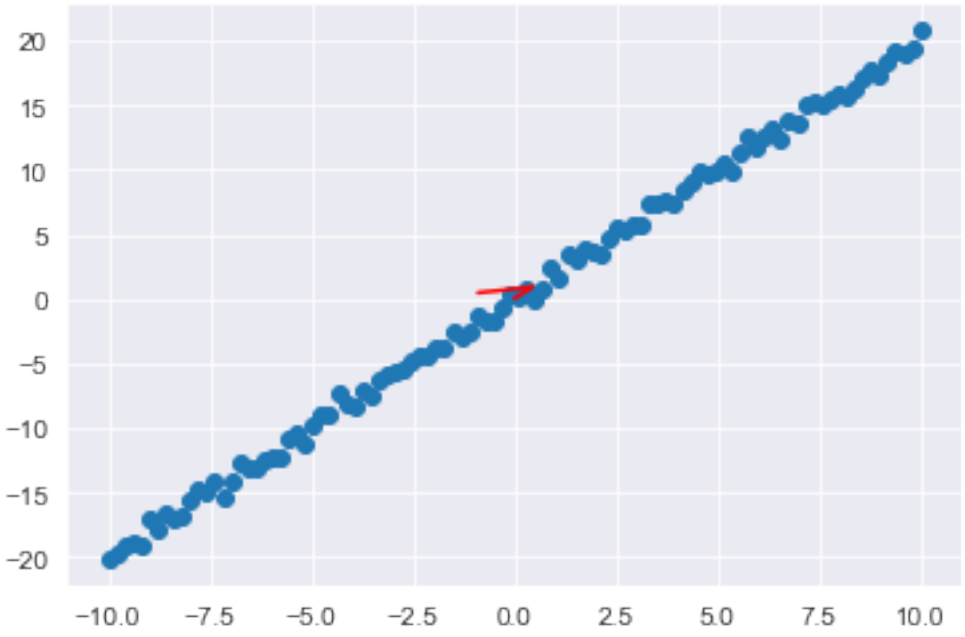
[32]:
```
array([ 7.10542736e-17, -8.04838374e-02])
```

1.5 Interpret Results

Let's take a look at what went on here. PCA transforms the dataset along principal axes. The first of these axes is designed to capture the maximum variance within the data. From here, additional axes are constructed which are orthogonal to the previous axes and continue to account for as much of the remaining variance as possible.

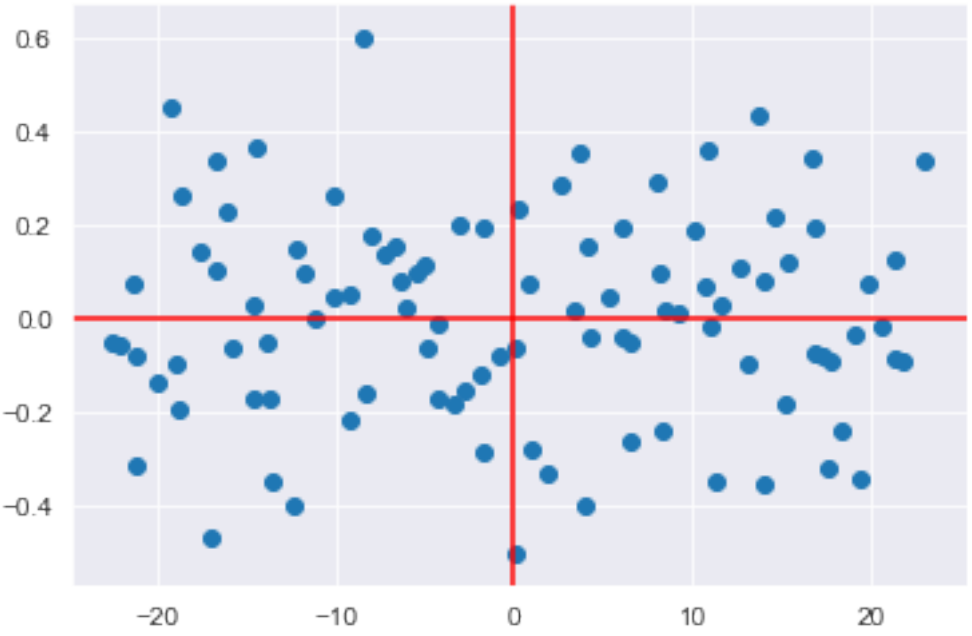
For the current 2-d case, the axes which the data were projected onto look like this:

[33]:
```
plt.scatter(x1, x2);
ax1, ay1 = pca.mean_[0], pca.mean_[1]
ax2, ay2 = pca.mean_[0] + pca.components_[0][0], pca.mean_[1] + pca.
 ↪components_[0][1]
ax3, ay3 = pca.mean_[0] + pca.components_[1][0], pca.mean_[1] + pca.
 ↪components_[1][1]
plt.plot([ax1, ax2], [ay1, ay2], color='red')
plt.plot([ax2, ax3], [ay2, ay3], color='red');
```



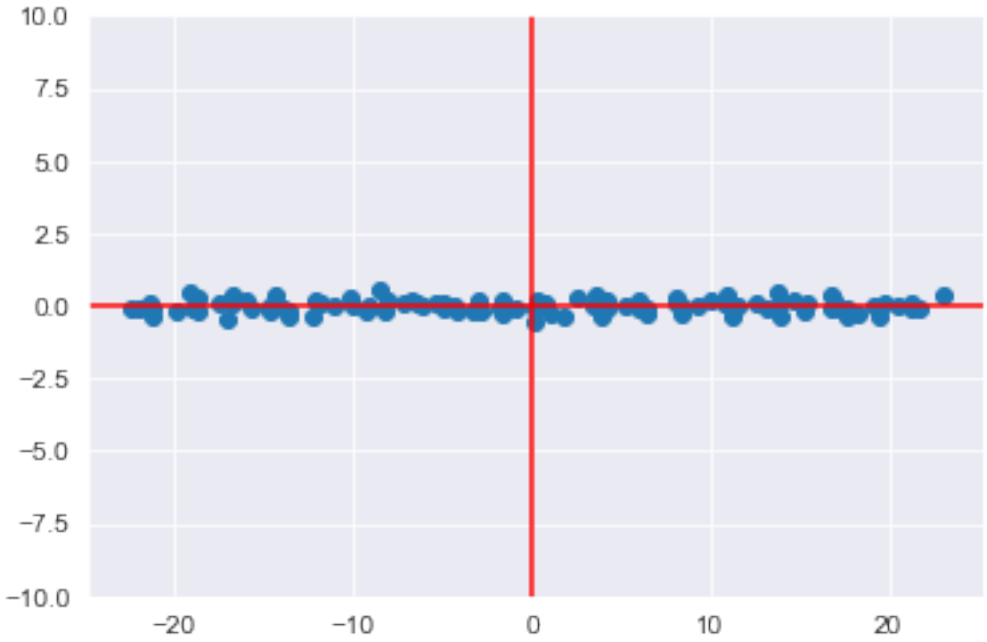
So, the updated graph you saw is the same dataset rotated onto these red axes:

[34]:
```
plt.scatter(transformed[:,0], transformed[:,1])
plt.axhline(color='red')
plt.axvline(color='red');
```



Note the small scale of the y-axis. You can also plot the transformed dataset on the new axes with a scale similar to what you saw before:

[35]:
```
plt.scatter(transformed[:,0], transformed[:,1])
plt.axhline(color='red')
plt.axvline(color='red')
plt.ylim(-10,10);
```



Again, this is the geographical interpretation of what just happened:

1.6 Determine the Explained Variance

Typically, one would use PCA to actually reduce the number of dimensions. In this case, you've simply re-re-parametrized the dataset along new axes. That said, if you look at the first of these primary axes, you can see the patterns encapsulated by the principal component. Moreover, scikit-learn also lets you quickly determine the overall variance in the dataset accounted for in each of the principal components.

[36]:
```
pca.explained_variance_ratio_
```

[36]:
```
array([9.99739242e-01, 2.60758409e-04])
```

Keep in mind that these quantities are cumulative: principal component 2 attempts to account for the variance not accounted for in the primary component. You can view the total variance using `np.cumsum()`:

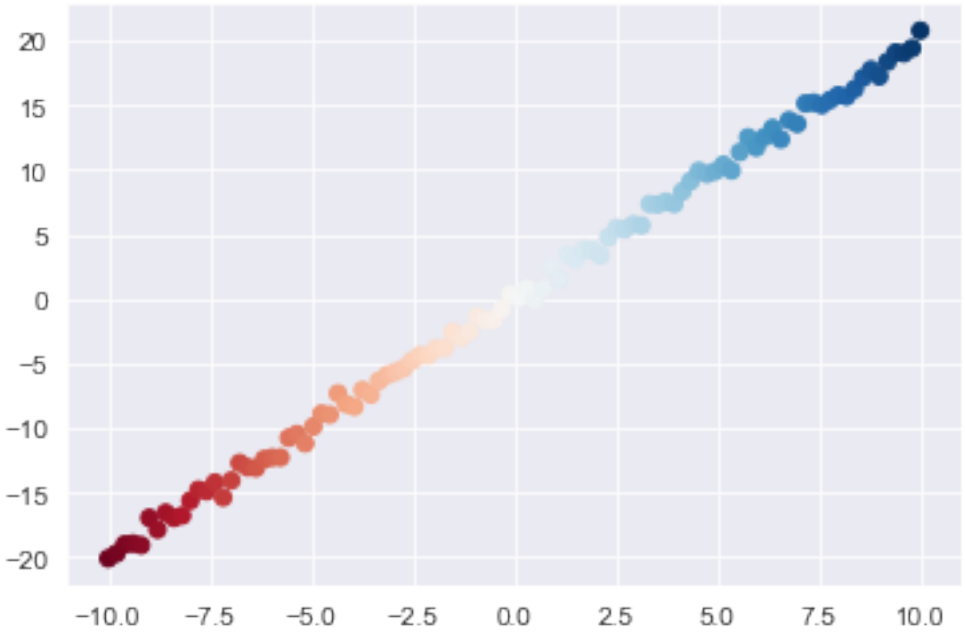
[37]:
```
np.cumsum(pca.explained_variance_ratio_)
```

[37]:
```
array([0.99973924, 1.        ])
```

1.7 Visualize the Principal Component

To help demonstrate the structure captured by the first principal component, observe the impact of coloring the dataset and then visualizing the first component.

[38]:
```
plt.scatter(x1,x2, c=sns.color_palette('RdBu', n_colors=100));
```



[39]:
```
plt.scatter(transformed[:,0], [0 for i in range(100)] ,
            c=sns.color_palette('RdBu', n_colors=100));
```

1.8 Steps for Performing PCA

The theory behind PCA rests upon many foundational concepts of linear algebra. After all, PCA is re-encoding a dataset into an alternative basis (the axes). Here are the exact steps:

1. Recenter each feature of the dataset by subtracting that feature's mean from the feature vector
2. Calculate the covariance matrix for your centered dataset
3. Calculate the eigenvectors of the covariance matrix
    1. You'll further investigate the concept of eigenvectors in the upcoming lesson
4. Project the dataset into the new feature space: Multiply the eigenvectors by the mean-centered features

You can see some of these intermediate steps from the `pca` instance object itself.

[40]:
```
# Pulling up the original feature means which were used to center the data
pca.mean_
```

[40]:
```
array([ 7.10542736e-17, -8.04838374e-02])
```

[41]:
```
# Pulling up the covariance matrix of the mean centered data
pca.get_covariance()
```

[41]:
```
array([[ 34.35023637,  68.78887942],
       [ 68.78887942, 137.98018174]])
```

[42]:
```
# Pulling up the eigenvectors of the covariance matrix
pca.components_
```

[42]:
```
array([[-0.44628564, -0.89489057],
       [ 0.89489057, -0.44628564]])
```

[70]:
```
x1_mean = np.mean(x1) * np.ones(x1.shape)
x2_mean = np.mean(x2) * np.ones(x2.shape)

x1_new = x1_mean - x1
x2_new = x2_mean - x2

X2 = np.matrix(list(zip(x1_new, x2_new)))

cov = X2.T.dot(X2)/100

print(np.mean(x1))
print(np.mean(x2))
print()
print(cov)
print()
print(pca.get_covariance() )
```

```
-7.105427357601002e-17
-0.08048383737781353

[[ 34.00673401  68.10099063]
 [ 68.10099063 136.60037992]]

[[ 34.35023637  68.78887942]
 [ 68.78887942 137.98018174]]
```

[72]:
```
evalue, evector = np.linalg.eig(cov)
print(evalue)
print()
print(evector)
print()
print(pca.components_ )
```

```
[4.44872396e-02 1.70562627e+02]

[[-0.89489057 -0.44628564]
 [ 0.44628564 -0.89489057]]

[[-0.44628564 -0.89489057]
 [ 0.89489057 -0.44628564]]
```

[82]:
```
a = np.array([[1, 0, 0],
              [0, 2, 3],
              [0, 3, 2]])
a
ev, eg = np.linalg.eig(a)
print(ev)
```

```
[ 5. -1.  1.]
```

1.9 Summary

In this lesson, you looked at implementing PCA with scikit-learn and the geometric interpretations of principal components. From here, you'll get a chance to practice implementing PCA yourself before going on to code some of the underlying components implemented by scikit-learn using NumPy.