Optional: Principal Component Analysis Deep Dive

Module 1 | Chapter 4 | Notebook 4

Sometimes data has more features than you really want. In this lesson, we'll look at how we can reduce their number while retaining the most important information. By the end you will be able to:

- Understand the principal component analysis process

PCA in Detail

In the last exercise we looked at the basic principle of how PCA works and we found that PCA recursively searches for orthogonal lines that approximate the direction of maximum variance in the data (principal components).

But how does this process work in detail and what mathematical methods are applied? First of all, we need a data set to illustrate the individual steps of PCA. Run the next cell for this.

In [1]:
```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

#read data
df = pd.read_csv('swim_records.txt', sep='\t', parse_dates=['date'])

#clean data
df['date'] = pd.to_numeric(df['date'])/10**10

#select numerical features
df = df[['date','seconds']]

df.head()
```

Out[1]:

	date	seconds
0	-193959360.0	336.48
1	-183919680.0	335.48
2	-182079360.0	329.00
3	-181690560.0	328.24
4	-181353600.0	324.24

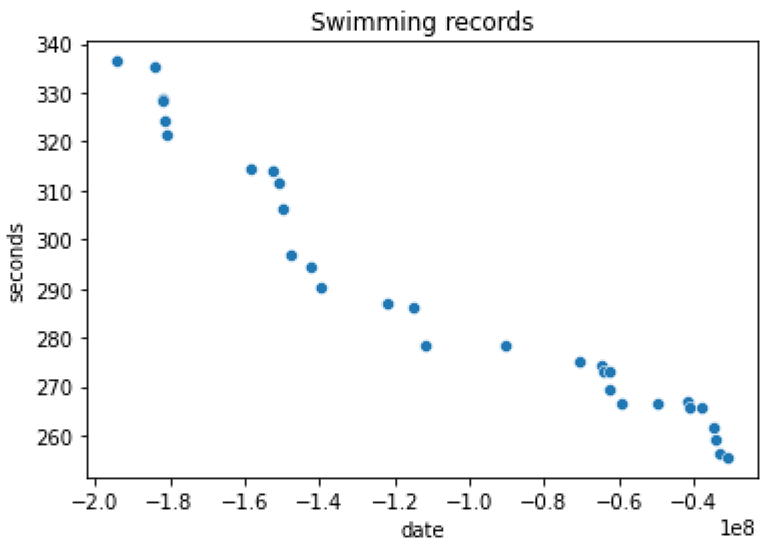
Now we should standardize the data so that the PCA features with large numerical values are considered equivalent to those with small ones. However, to understand the system behind PCA, it is more helpful to work with un-standardized data. At this point we'll settle with simply converting the `DataFrame` into a `numpy` array.

In [2]:
```
X = df.values
```

A short look into the data shows that we have two dimensions, which it only makes sense to reduce to a single dimension.

In [3]:
```
ax = sns.scatterplot(x=X[:,0], y=X[:,1])

ax.set(title='Swimming records',
       xlabel='date',
       ylabel='seconds');
```



Step 1: Finding the center of the data

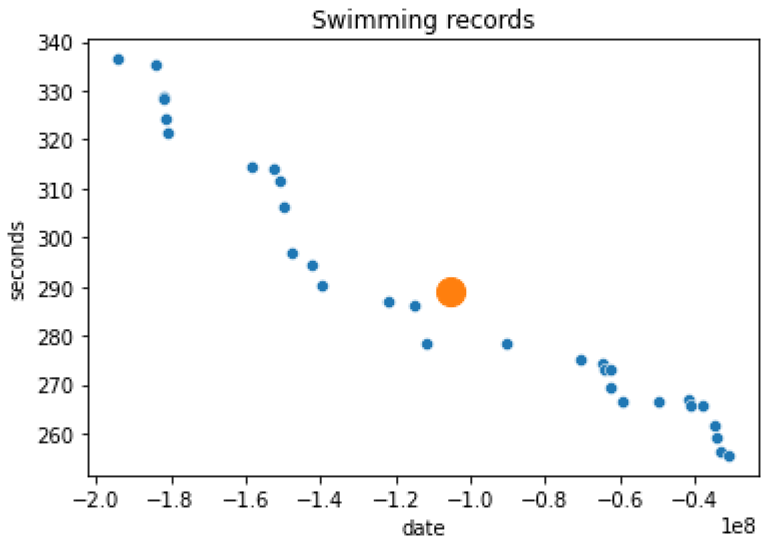
First we need the central point in the data. This will help us to move our data points to the point of origin in the next step. The central point here is nothing more than a vector containing the mean values of each feature.

In [8]:
```
means = X.mean(axis=0)
print("Center of the data: ",means)

#plot data
ax = sns.scatterplot(x=X[:,0], y=X[:,1])
#add central point
ax.scatter(means[0],
           means[1],
           s=200)

ax.set(title='Swimming records',
       xlabel='date',
       ylabel='seconds');
```

```
Center of the data:  [-1.05489662e+08  2.89072258e+02]
```

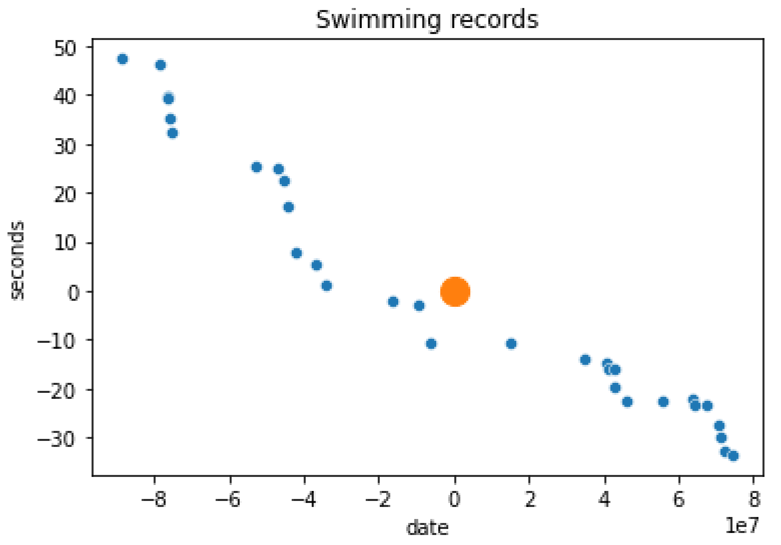


Step 2: Moving the data towards the point of origin

If we move the data so that the data center is at the point of origin (0,0), each data point is a location vector. To move the data, we subtract our `means` vector from the data points.

```
In [5]:
#plot data
X_shifted = X-means
ax = sns.scatterplot(x=X_shifted[:,0], y=X_shifted[:,1])
#add central point
ax.scatter(0, 0, s=200)

ax.set(title='Swimming records',
       xlabel='date',
       ylabel='seconds');
```



Step 3: Determining the direction of the maximum variance using SVD

As we saw in the last lesson, we now need to find a line that is able to explain as much of the variance in the data as possible. That sounds a bit like the R2 score and linear regression. Unfortunately a linear regression only works if we know which feature depends on which other feature. In our case we can assume that the swim time depends on the date the record was set and not the other way round.

In a PCA with multiple features, when we have no idea what depends on what, we have to use another technique: *Singular Value Decomposition* or SVD. Explaining SVD would go beyond the scope of this notebook. So for now let's accept the fact that SVD outputs a very special matrix for us (in our case we called it `V`), whose vectors contain similar information as weights in a linear regression. Conveniently, the vectors in this matrix are all perpendicular to each other. You can think of this matrix as a coordinate system that lies skewed in our data. The direction of the first vector in this matrix is also the direction of the highest variance in the data set.

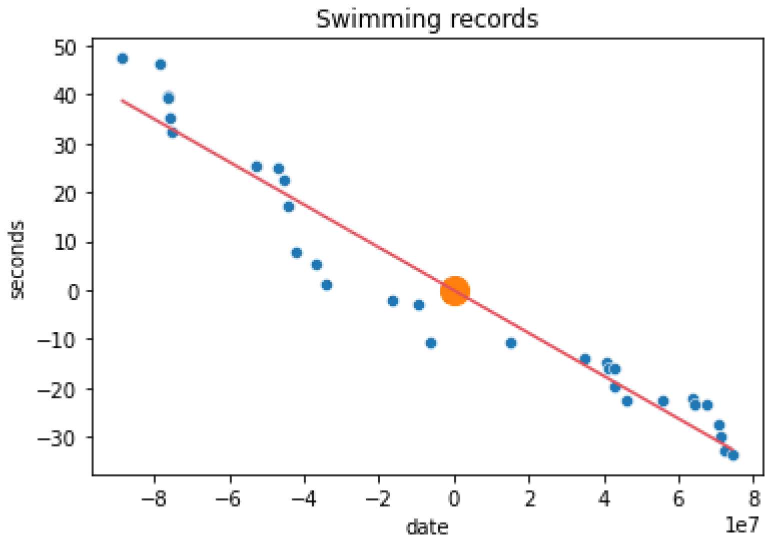
In [9]:
```
#we are only interested in the Vectors of V
U, S, V = np.linalg.svd(X_shifted, full_matrices=False)
```

In [10]:
```
ax = sns.scatterplot(x=X_shifted[:,0], y=X_shifted[:,1])
#add central point
ax.scatter(0,0,s=200)

#plot best fitting line of pc1
samples = np.linspace(start=X_shifted[:,0].min(),
                      stop=X_shifted[:,0].max(),
                      num=len(X_shifted))

preds = samples*V[0][1]
ax.plot(samples,preds, color='#e54f59')

ax.set(title='Swimming records',
       xlabel='date',
       ylabel='seconds');
```



Step 4: Generating the principal components

```
In [11]: V
```

```
Out[11]: array([[ 1.00000000e+00, -4.36965275e-07],
                [-4.36965275e-07, -1.00000000e+00]])
```

As you can see, each entry in `V` has two values - one for each feature. These are the ingredients for the "recipes" of the principal components. So the first principal component (pc1) consists of one part `'date'` and about -0.000000436 parts `'seconds'`. These vectors are referred to as **eigenvectors**. The length of these eigenvectors has to be one. Therefore we should divide them by their norm.

```
In [14]: eigenvectors = [x / np.linalg.norm(x) for x in V]
         eigenvectors
```

```
Out[14]: [array([ 1.00000000e+00, -4.36965275e-07]),
          array([-4.36965275e-07, -1.00000000e+00])]
```

Now we just have to determine how many principal components we need. In our case we want to reduce two dimensions to one dimension, so `n_components = 1`. So we only need the first entry from our list of eigenvectors.

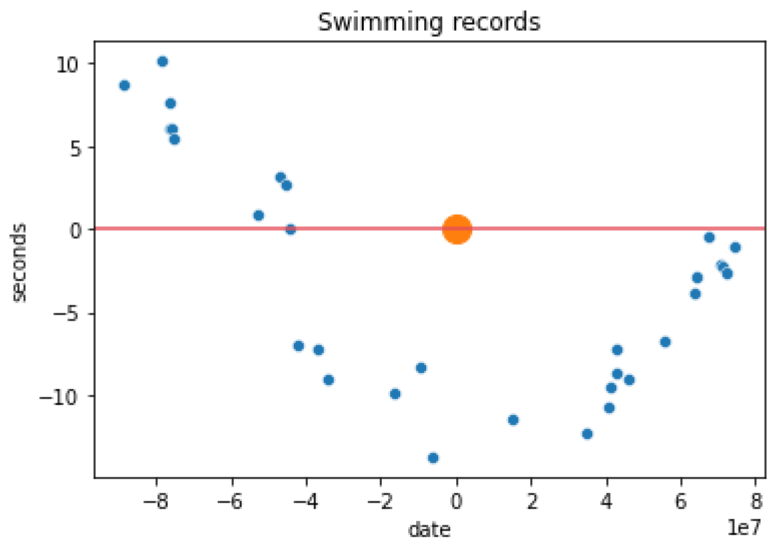
```
In [15]: n_components = 1
         components = np.array(eigenvectors[:n_components])
         components
```

```
Out[15]: array([[ 1.00000000e+00, -4.36965275e-07]])
```

Step 5: Transforming the data

In the final step we'll now use our principal components to transform the data. This simply means using our coordinate system to rotate the eigenvector of pc1...

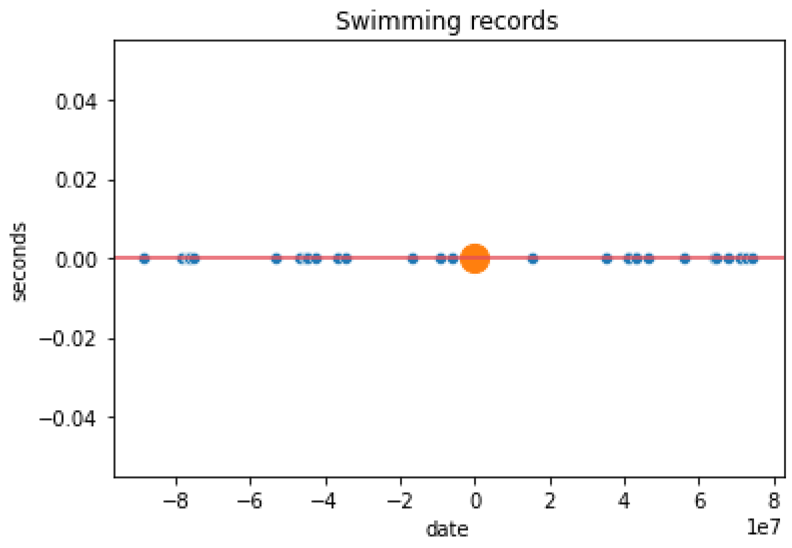
```
In [16]: ax = sns.scatterplot(x=X_shifted[:,0], y=X_shifted[:,1]-preds)
         #add central point
         ax.scatter(0,0,s=200)
         ax.axhline(0, color='#e54f59')
         ax.set(title='Swimming records',
                xlabel='date',
                ylabel='seconds');
```



… and projecting the data onto pc1. We can carry out both steps with a simple scalar product of the shifted data and transposed vector containing our principal components.

```
In [17]:
# project shifted data onto pc1 using its eigenvector
transformed = np.dot(X_shifted,components.T)

ax=sns.scatterplot(x=transformed.flatten(), y=np.zeros(len(X)))
#add central point
ax.scatter(0,0,s=200)
ax.axhline(0, color='#e54f59')
ax.set(title='Swimming records',
       xlabel='date',
       ylabel='seconds');
```



Congratulations: You have seen in detail how PCA works.

Compare the results with sklearn

You can see for yourself that the projection depicted above is correct by comparing it with the results of the PCA from `sklearn`.

```
In [18]: from sklearn.decomposition import PCA
```

```
In [19]: pca = PCA(n_components=1)
         sklearn_transformed = pca.fit_transform(df)
         sklearn_transformed
```

```
Out[19]: array([[ 88469698.0645284 ],
                [ 78430018.06452891],
                [ 76589698.06452624],
                [ 76200898.06452596],
                [ 75863938.06452425],
                [ 75224578.06452304],
                [ 53028418.06452211],
                [ 46833538.06452265],
                [ 45140098.0645216 ],
                [ 44500738.06451943],
                [ 42280258.06451555],
                [ 36716098.06451496],
                [ 34331458.06451333],
                [ 16679938.06451363],
                [  9327298.064514  ],
                [  6035458.0645109 ],
                [-15115261.93548713],
                [-35099581.93548661],
                [-40983421.93548638],
                [-41199421.93548688],
                [-42970621.93548676],
                [-43152061.9354884 ],
                [-46236541.9354893 ],
                [-55792381.93548845],
                [-63896701.93548741],
                [-64561981.93548799],
                [-67741501.93548769],
                [-70869181.93548916],
                [-71093821.93549022],
                [-72553981.93549123],
                [-74385661.93549143]])
```

Do the transformed values match?

```
In [20]: transformed
```

```
Out[20]: array([[-88469698.06452839],
               [-78430018.06452891],
               [-76589698.06452626],
               [-76200898.06452598],
               [-75863938.06452425],
               [-75224578.06452306],
               [-53028418.06452211],
               [-46833538.06452265],
               [-45140098.06452161],
               [-44500738.06451944],
               [-42280258.06451555],
               [-36716098.06451496],
               [-34331458.06451333],
               [-16679938.06451363],
               [ -9327298.064514  ],
               [ -6035458.0645109 ],
               [ 15115261.93548714],
               [ 35099581.93548661],
               [ 40983421.93548639],
               [ 41199421.93548688],
               [ 42970621.93548676],
               [ 43152061.9354884 ],
               [ 46236541.9354893 ],
               [ 55792381.93548845],
               [ 63896701.93548742],
               [ 64561981.93548799],
               [ 67741501.93548769],
               [ 70869181.93548916],
               [ 71093821.93549022],
               [ 72553981.93549123],
               [ 74385661.93549141]])
```

Except for the sign, the values are the same. Since sorting the data does not change its relative distribution, our PCA is usable. The differences in the signs are due to how `sklearn` carries things out. The principal components found are the same:

```
In [21]: print("Sklearn PCA Components:", pca.components_)
         print("DIY PCA Components:", components)
```

```
Sklearn PCA Components: [[-1.00000000e+00  4.36965275e-07]]
DIY PCA Components: [[ 1.00000000e+00 -4.36965275e-07]]
```

Congratulations: You have deepened your understanding of how principal component analysis works. You have seen that `PCA` projects the data in a way that preserves as much of the variance as possible. This is carried out by centering the data and determining the eigenvectors using SVD. Next, we will integrate dimensionality reduction into our data pipeline.

Remember:

- **Eigenvectors** in a PCA indicate how the *feature* matrix must be rotated for the direction of maximum variance to match the X-axis
- **Principal components** can be determined using *singular value decomposition* (SVD)
- The values of the eigenvectors can be found in `my_pca.components_`
  - During the data's transformation, the scalar product is formed from the feature matrix and the eigenvectors of the principal components.