PCA Recap

Module 1 | Chapter 5 | Notebook 1

In this notebook we will recap the most important things from the previous chapter. This is all repetition. If you discover a lot of new things here, we recommend that you take a look back at Chapter 4. The relevant lessons for each section are clearly marked.

Principal component analysis

In the last chapter we dealt with dimensionality reduction. Dimensionality reduction belongs to the field of unsupervised learning. So you work without a target vector. Instead, information about the structure of the data is used directly to determine whether and how the data can be transformed so that it has fewer features.

The most well-known algorithm for dimensionality reduction is *principal component analysis* or `PCA` for short. However, the basic assumption is always that there are features that contain redundant or little information about the data. The data is then projected onto this setup.

In doing so, `PCA` proceeds as follows: For each dimension that the data should have after the transformation, the algorithm searches for the direction of the greatest variance in the data. This direction is called the *principal component*. The next principal component is chosen so that it is perpendicular to the previous one as a vector, pointing in the direction of the greatest remaining variance.

In the end the data is projected onto the principal components. This is shown visually below. Each principal component is then a feature of the transformed data. The number of new features is therefore directly indicated by the number of principal components.

This procedure offers the following advantages:

- Because the principle components point in the direction of the greatest variance, as little information as possible is lost about the structure of the data.
- Because the principal components are perpendicular to one another, they are independent of one another. This means that they do not contain redundant information about the data. (Examples for perpendicular vectors in 2 dimensions are [1,0] and [0,1] or also [1,1] and [1,-1])

The best way to understand how `PCA` works is to use an example. To do this, we'll import the *swim_records.txt* file from the home folder. It contains the swimming world records for 400 meters freestyle of men from 1908 to 1960. Store the data as a `df`.

In [1]:
```
import pandas as pd  # import pandas to read in the data
df = pd.read_csv('swim_records.txt', sep='\t', parse_dates=['date'])  # use the tabulc
df.head()
```

Out[1]:

	seconds	date	country	athlete
0	336.48	1908-07-16	United Kingdom	Henry Taylor
1	335.48	1911-09-21	United Kingdom	Sydney Battersby
2	329.00	1912-04-21	Hungary	Alajos Kenyery
3	328.24	1912-06-05	Hungary	Bela Las-Torres
4	324.24	1912-07-14	Canada	George Hodgson

The following data dictionary describes the data.

Column number	Column name	Type	Description
0	`'seconds'`	continuous (`float`)	time of the world record in seconds
1	`'date'`	continuous (`datetime`)	datee the world record was set
2	`'country'`	categorical (`string`)	country where the world record was set
3	`'athlete'`	categorical (`string`)	athlete who set the world record

`PCA` was designed for continuous variables. Because of this, we'll only deal with the `'date'` and `'seconds'` columns. We'll standardize the data with `StandardScaler`. Before that, the dates are converted into numerical data. We'll store the standardized data as `arr_std`. Ignore the `DataConversionWarning`.

Important: Standardizing data is an important step. The results of `PCA` are clearly dependent on the variance of the features. If the features are on different scales, their importance is considered to be different by `PCA`. We can avoid this problem by standardizing the data.

In [2]:
```
# ignore sklearns DataConversionWarning using the module warnings
import warnings
from sklearn.exceptions import DataConversionWarning
warnings.filterwarnings(action='ignore', category=DataConversionWarning)

df.loc[:, 'date_int'] = pd.to_numeric(df.loc[:, 'date'])  # create a new column with t

from sklearn.preprocessing import StandardScaler  # import StandardScaler
scaler = StandardScaler()  # instantiate the scaler
arr_std = scaler.fit_transform(df.loc[:, ['date_int', 'seconds']])  # fit scaler and t
```

If we visualize the standardized data as a scatter plot, we quickly see that it follows a line. The greatest variance is along this line. You can see this from the fact that the data points along this direction have the largest distances between them.

 Swimming World Records Line

To make sure the data points are only described by this direction, we need `PCA` from the `sklearn.decomposition` module. `n_components` is the most important parameter when you instantiate `PCA`. It specifies how many principal components (directions) `PCA` should calculate. We only want to have one direction, so we'll choose `n_components=1`.

In [3]:
```
from sklearn.decomposition import PCA  # import PCA
pca = PCA(n_components=1)  # instantiate PCA with one component
```

`PCA` is a transformer. You can find the principal components with the `my_transformer.fit()` method. The `my_transformer.transform()` method transforms the data so that it has as many features as principal components. In our case, the transformed data only has one feature. So we'll store it as `arr_1d`.

In [4]:
```
pca.fit(arr_std)  # calculate the principal component
arr_1d = pca.transform(arr_std)  # project the data onto the principal component
```

What does that mean now that the data only has one dimension? We only have one column left in the data. These values of this column can be interpreted so that they all lie on the calculated principal component, i.e. the direction of the greatest variance. If we compare the transformed data points with the old data points, we get the following picture:

 Projection of data

The image section shows a part of the original data as blue dots. The line shows the direction of the greatest variance. The arrows show how the original data is projected onto this line. The result is the transformed data, which is displayed as orange crosses.

The transformed data points now lie exactly on the same line one after the other. Therefore they only have one dimension internally. Because the direction of the greatest variance is used in the data, the distances between the transformed data points are as large as possible. This way, `PCA` keeps the loss of information to a minimum.

In order to be able to estimate how large the loss of information is, `PCA` offers us the `pca.explained_variance_ratio_` attribute. This lists how much of the original variance can be explained by it for each principal component. That is, how much variance is restored when reconstructing the data. How much is left in this example?

In [5]:
```
pca.explained_variance_ratio_
```

Out[5]:
```
array([0.98436117])
```

With a single principal component, we can reproduce about 98% of the variance in the data. To get a feeling for what this means, we can transform the transformed data back. `PCA` offers us the `my_transformer.inverse_transform()` method to do this. We can apply this to the one-dimensional data in order to obtain it as two-dimensional data again. Afterwards we'll visualize them both.

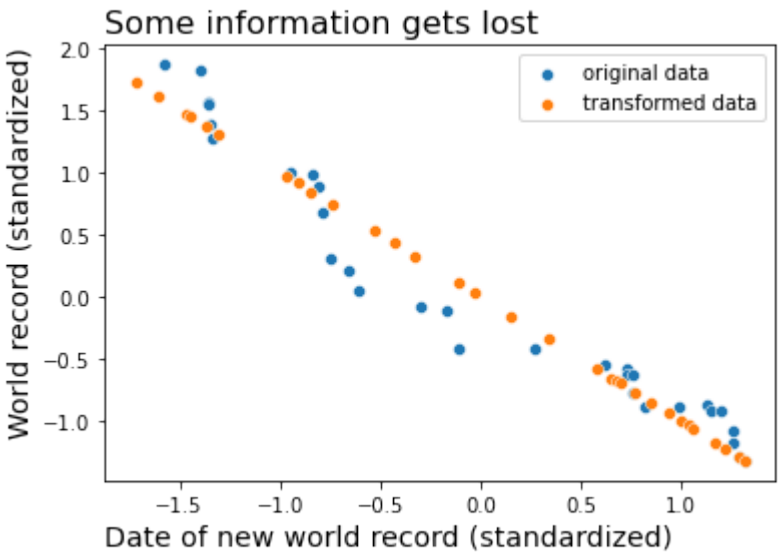
In [6]:
```
arr_reconstructed = pca.inverse_transform(arr_1d)

import seaborn as sns

ax = sns.scatterplot(x=arr_std[:, 0],
                     y=arr_std[:, 1],
                     label='original data')
sns.scatterplot(x=arr_reconstructed[:, 0],
                y=arr_reconstructed[:, 1],
                ax=ax,
                label='transformed data')

ax.set_xlabel(xlabel='Date of new world record (standardized)',
              position=[0, 0],
              horizontalalignment='left',
              size=14)
ax.set_ylabel(ylabel='World record (standardized)',
              position=[0, 1],
              horizontalalignment='right',
              size=14)
ax.set_title(label='Some information gets lost',
             loc='left',
             horizontalalignment='left',
             size=16)
```

Out[6]:
```
Text(0.0, 1.0, 'Some information gets lost')
```



The information about the position of the points perpendicular to the principal component is lost. As you can see, the reconstructed data (orange) can only be found along the principal component. However, the variance of the reconstructed data is still about 98%.

In [7]:
```
print('Variance of standardized data:', arr_std.var())  # calculate the variance of th
print('Variance of reconstructed data:', arr_reconstructed.var())  # calculate the var
```

```
Variance of standardized data: 0.9999999999999999
Variance of reconstructed data: 0.9843611656201218
```

So we lose less than 2% of the variance, but have saved 50% of the storage space. This is because the year and the swimming record show a high correlation. In this way they provide

basically the same information. The aim of dimensionality reduction is to sift out this kind of redundant information.

Dimensionality reduction can be used for the following applications:

- reducing the amount of storage space needed and increasing the speed of a model
- reducing noise in the data
- visualizing high-dimensional data
- extracting relevant features from the data

Congratulations: You've recapped what `PCA` actually is. Now we can turn to the problem of finding the optimal number of principal components.

Determining the number of principal components

With `PCA`, it's particularly interesting to find out how many principal components you need. To determine this, we recommend choosing the approach depending on what you want to use it for:

- Data visualization: `n_components=2` or `n_components=3`
- Part of a `pipeline` for a prediction model: select `n_components` as the parameter for `GridSearchCV' or 'validation_curve()`
- Data compression: use the *elbow method* for `n_components` or specify the percentage of the variance to be retained directly as a floating point number for `n_components`

Let's briefly look at the last case. For this we'll import some data with a few more features. We'll use *wine_composition.csv* from the previous chapter and save the data as `df_wines`.

In [8]:
```
df_wines = pd.read_csv('wine_composition.csv', index_col='id')  # read the compositior
df_wines.head()
```

Out[8]:

id	fixed.acidity	volatile.acidity	citric.acid	residual.sugar	chlorides	free.sulfur.dioxide	total.sulfur.dic
0	7.0	0.27	0.36	20.7	0.045	45.0	
1	6.3	0.30	0.34	1.6	0.049	14.0	
2	8.1	0.28	0.40	6.9	0.050	30.0	
3	7.2	0.23	0.32	8.5	0.058	47.0	
4	7.2	0.23	0.32	8.5	0.058	47.0	

The following data dictionary explains what the data means.

Column number	Column name	Type	Description
0	`'fixed.acidity'`	continuous (`float`)	liquid acid content.
1	`'volatile.acidity'`	continuous (`float`)	gaseous acid content, particularly relevant for the odor.
2	`'citiric.acid'`	continuous (`float`)	citric acid content. Part of the liquid acids. Affects freshness in taste.
3	`'residual.sugar'`	continuous (`float')	Natural sugar of the grapes, which is still present at the end of fermentation stage.
4	`'chlorides'`	continuous (`float`)	chloride content. Minerals, which can be dependent on the wine growing region.
5	`'free.sulfur.dioxide'`	continuous (`float`)	unbound sulfur dioxide. Perceptible by smell.
6	`'total.sulfur.dioxide'`	continuous (`float`)	total sulfur dioxide content
7	`'density'`	continuous (`float`)	density of the wine
8	`'PH'`	continuous (`float`)	pH value. Determined by the acid content.
9	`'sulphates'`	continuous (`float`)	sulphate content.
10	`'alcohol'`	continuous (`float`)	alcohol content
11	`'color'`	categorical	color of the wine. Contains only `'red'` and `'white'`
12	`'id'`	categorical (`int`)	Unique identification number of the wine.

There are not yet too many features in the `DataFrame`. In the last chapter we created new features from the existing ones by using `PolynomialFeatures` to create features of a higher degree. We'll implement this here as well and save the resulting features as `features` and standardize them.

In [9]:
```
from sklearn.preprocessing import PolynomialFeatures  # import PolynomialFeatures
poly_transformer = PolynomialFeatures(degree=2, include_bias=False)  # instantiate for
features = poly_transformer.fit_transform(df_wines.iloc[:, :-1])  # calculate new feat
features_std = scaler.fit_transform(features)  # standardize the features
```

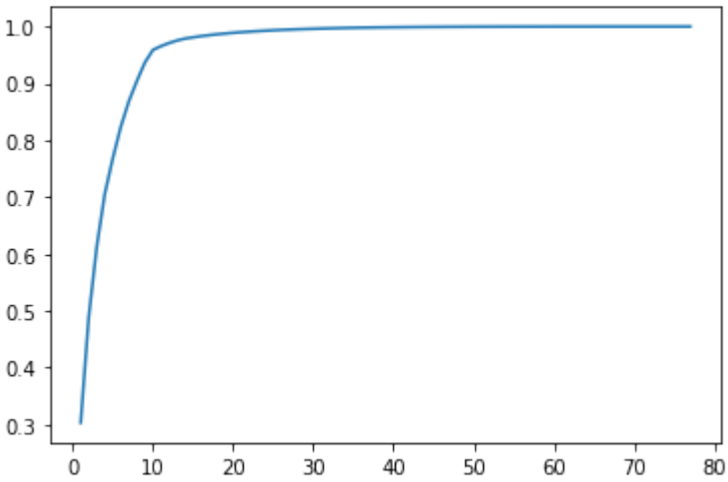
We won't pass any parameters to `PCA`, so it generates as many principal components as there are features. We can use this to generate a curve for the *elbow-method*.

In [10]:
```
model = PCA()  # instantiate PCA without parameter to get all components
model.fit(features_std)  # fit pca to the standardized data
```

Out[10]:
```
PCA()
```

We'll generate the curve by forming a cumulative sum of `pca.explained_variance_ratio_` with `np.cumsum()`. In this way we can see how much information is gained with each additional component.

In [11]:
```
import numpy as np
explained_variances = np.cumsum(model.explained_variance_ratio_)
ax=sns.lineplot(x=range(1, len(model.components_) + 1),
                y=explained_variances)
```



Here we can see a significant bend after 10 principal components. If our primary concern is to extract the most important information from the data in order to compress it, then 10 is a good value. Alternatively, we could specify the explained variance directly. How many components do we get if we want to preserve at least 95% of the variance?

In [13]:
```
model = PCA(n_components=0.98)  # instantiate PCA with float as argument for n_compone
model.fit(features_std)  # fit pca to the standardized data
len(model.components_)
```

Out[13]:
```
15
```

We can map 95% of the variance with only 10 of 77 features. This is a significant data compression.

Congratulations: You have recapped the most important things from the chapter on dimensionality reduction. `PCA` is a very useful and versatile algorithm that can be easily integrated into a data pipeline.

Remember:

- `PCA` searches for the direction (feature structure) of the greatest variance and projects the data points onto it
- `n_components` is the number of principal components to be generated
- Pass `n_components` a floating point number between 0 and 1 to specify the minimum variance you want to retain

---

Do you have any questions about this exercise? Look in the forum to see if they have already been discussed.

Found a mistake? Contact Support at support@stackfuel.com.
