

Data Compression with PCA

Module 1 | Chapter 4 | Notebook 7

You recently used principal component analysis to get fewer features to slightly improve your predictions. But it can also help you to get an impression of your data's structure. We'll take a look at that now. You will learn:

- How principal component analysis can help with visualization
 - How to find an appropriate number of principal components without using a prediction metric
 - How to transform back data after principal component analysis
-

A glance at the data

Scenario: Your company wants to digitize all its documents. This should be done in such a way that the texts are recognized as such and can be stored as metadata. To do this, your department has to create a prototype for recognizing letters in images. For this purpose, you have collected an extensive data set of letters in very different fonts. As a *proof of concept* you should first distinguish between a small selection of letters.

In this lesson, we'll look at how we can gain insight into our data's structure `PCA`. We will illustrate this using image data as an example. Pictures in particular usually have an extremely high number of features. Each pixel is often interpreted as a separate feature. With an image size of just 28 by 28 pixels, as in our example, this already results in 784 features. Most of the time, however, images are much larger than this, so you very quickly end up with millions of features.

Let's start by taking a look at our data. We have greyscale images of letters in various fonts. They are located in the database `letters.db` in your current working directory. You covered databases in *Databases (Module 0 Chapter 1)*. Import `sqlalchemy` with its conventional alias `sa` and create a connection to the database. Remember that `letters.db` is an SQLite database and that it is located in the current working directory.

```
In [1]: import sqlalchemy as sa
engine = sa.create_engine('sqlite:///letters.db')
connection = engine.connect()
```

First let's find out which tables our database contains. To do this, we'll create an *inspector* using the `sa.inspect(engine)` function. This has the `my_inspector.get_table_names()` method, which returns the names of the tables. Name the *inspector* `inspector` and print the table names.

```
In [2]: inspector = sa.inspect(engine)
inspector.get_table_names()
```

```
Out[2]: ['images', 'labels']
```

So we have two tables called `images` and `labels`. Access the two tables with `pandas` and store them in the DataFrames `df_img` and `df_labels` respectively.

```
In [3]: import pandas as pd
df_img = pd.read_sql('SELECT * FROM images', con=connection)
df_labels = pd.read_sql('SELECT * FROM labels', con=connection)
```

Now we've retrieved the data we need, you can close the connection to the database with `my_connection.close()`. Do this in the following cell.

```
In [4]: connection.close()
```

How many rows and columns does each `DataFrame` have?

```
In [5]: print('img',df_img.shape)
print('labels',df_labels.shape)
```

```
img (3745, 785)
labels (3745, 2)
```

Both DataFrames have 3745 lines. Print 5 lines from `df_img`.

```
In [6]: df_img.head()
```

```
Out[6]:
```

	index	0	1	2	3	4	5	6	7	8	...	774	775
0	0	0.0	0.0	0.0	0.0	0.000000	0.000000	0.000000	0.007843	0.000000	...	0.121569	0.007843
1	1	1.0	1.0	1.0	1.0	1.000000	1.000000	1.000000	1.000000	1.000000	...	1.000000	1.000000
2	2	0.0	0.0	0.0	0.0	0.000000	0.007843	0.007843	0.000000	0.000000	...	0.788235	0.607843
3	3	0.0	0.0	0.0	0.0	0.007843	0.007843	0.000000	0.000000	0.078431	...	0.615686	0.564706
4	4	0.0	0.0	0.0	0.0	0.011765	0.000000	0.043137	0.321569	0.486275	...	0.980392	0.937255

5 rows × 785 columns

We have a total of 785 columns in `df_img`. One of them is the `'index'` column, which is not a real feature. The others have no names and seem to contain values ranging from 0 to 1. What about `df_labels`? Print the first 5 rows.

```
In [7]: df_labels.loc[:, 'labels'].unique()
```

```
Out[7]: array(['C', 'I'], dtype=object)
```

We also have the `'index'` column in `df_labels`. We also have a column called `'labels'`. It contains the letter `'C'`.

In this data set, each row represents a picture of one letter. The values in `df_img` represent the brightness values of the pixels. The image is in greyscale, so we have no colors. `df_labels` indicates which letter the image contains.

Check if the data has missing values and if the `'index'` columns match.

Tip: If you use the method `my_mask.all()` with a Boolean mask, you get `True` if all values of the mask are `True`. Otherwise you will get `False` as a result.

```
In [8]: df_labels.isna().sum()
df_img.isna().sum()
```

```
Out[8]: index    0
0        0
1        0
2        0
3        0
..
779      0
780      0
781      0
782      0
783      0
Length: 785, dtype: int64
```

There are no missing values in either `DataFrame` and the `'index'` values match up. These are a good basis, we'll just carry out a final cleaning step.

Assign the the `'index'` column to the row names of each `DataFrame` accordingly. Delete this column afterwards. Then check whether the column has been deleted by printing the column names.

```
In [9]: df_img.index = df_img.loc[:, 'index']
df_img = df_img.drop('index', axis=1)

df_labels.index = df_labels.loc[:, 'index']
df_labels = df_labels.drop('index', axis=1)

print(df_img.columns)
print(df_labels.columns)

Index(['0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
      ...
      '774', '775', '776', '777', '778', '779', '780', '781', '782', '783'],
      dtype='object', length=784)
Index(['labels'], dtype='object')
```

Which different letters do we have in the data?

```
In [10]: df_labels.loc[:, 'labels'].unique()
```

```
Out[10]: array(['C', 'I'], dtype=object)
```

We only have the letters 'C' and 'I'. Now finally take a look at some of the images. For this, we'll import `import matplotlib.pyplot` and `import matplotlib.image`. Then we'll define an image and use `my_axis.imshow()` from `matplotlib.image` to display some of the images.

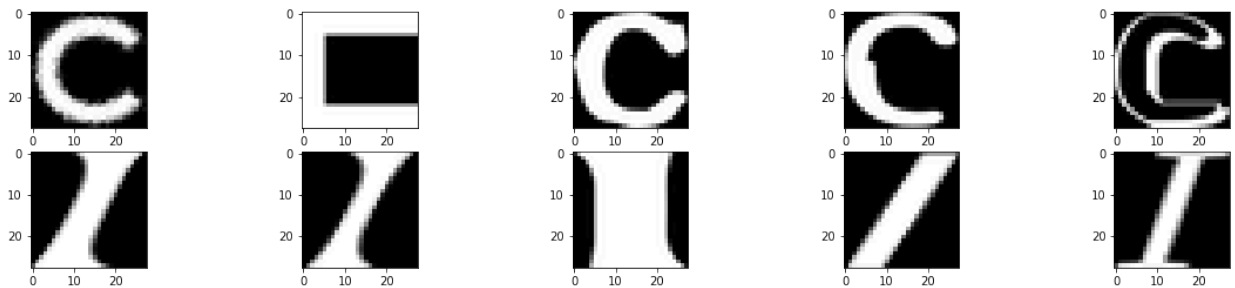
In order to be able to display the images, we have to restructure them beforehand. Because at the moment they are only stored as rows. But we need a 2-dimensional array with 28 times 28 values (pixels). We can do this with the `my_array.reshape()` method. It returns the array restructured so that the rows and columns have a given value.

Run the following cell to display a few of the pictures.

Important: The total number of values in the array must be kept the same with `my_array.reshape()`, otherwise you will get a `ValueError`. The product of the values in `my_array.shape` therefore have to remain constant.

```
In [11]: # make all imports
import matplotlib.pyplot as plt
import matplotlib.image as mpimg

# define figure with 10 axes in 2 rows
fig, axs = plt.subplots(nrows=2, ncols=5, figsize=[20,4]) # define figure with 10 axes
# plot images of C in the first row
for i in range(5):
    axs[0, i].imshow(df_img.loc[i, :].values.reshape(28,28), cmap='gray') # use cmap
# plot images of I in the second row
for i in range(1, 6): # we use .iloc and negative values. Negative 0 is also 0, so ha
    axs[1, i-1].imshow(df_img.iloc[-i, :].values.reshape(28,28), cmap='gray') # I let
```



Congratulations. You retrieved the data from the database and familiarized yourself with it. The data only contains the letters C and I. As you have just seen, they are shown in very different fonts. Now let's take a quick look at how we can use `PCA` to get a brief overview of the data.

Display high-dimensional data in two dimensions

We often want to visualize our data points to get a feeling for them. With 2 and 3 dimensions this is no problem. However, if data points have more features, this is no longer an easy task. A principal component analysis can help us with this. This allows us to reduce the data to 2

dimensions and then display it in a scatter plot. We can often already see whether there are clusters in the data. Now let's try that out with the letter data.

Follow these steps:

- Import `PCA` from `sklearn.decomposition`.
- Instantiate `PCA` with `n_components=2` and `random_state=10` as `pca`.
- Save data from `df_img` to `arr_img`.
- Transform `arr_img` so that the data in it only contains 2 dimensions. Store it as `arr_img_2d`.

How much of the variance is retained by this transformation? Each value in `pca.explained_variance_ratio_` represents the variance described by each component. The sum of these is therefore the total percentage that is retained.

```
In [12]: from sklearn.decomposition import PCA
pca = PCA(n_components=2, random_state=10)
arr_img = df_img
arr_img_2d = pca.fit_transform(arr_img)

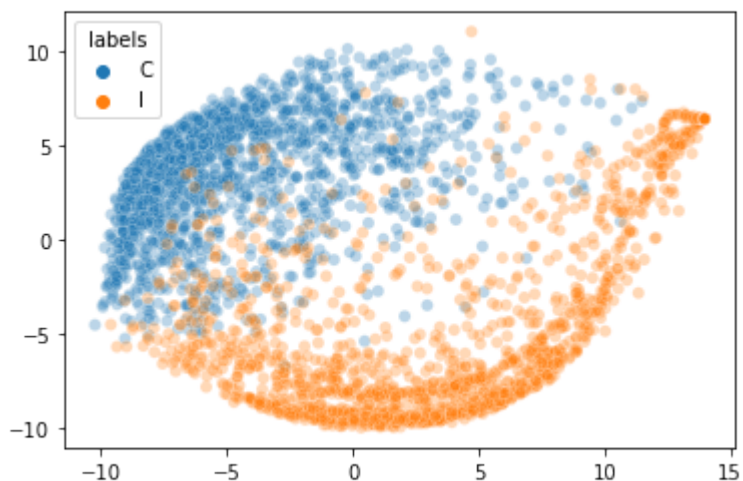
sum(pca.explained_variance_ratio_)
```

```
Out[12]: 0.5585785706927329
```


After all, about 56% of the variance is still retained. And that's just with 2 out of 784 features! Now generate a scatterplot of `arr_img_2d`. Color the data points using the labels in `df_labels` and make them a little transparent. For example, use the parameter `alpha=0.3`.

```
In [13]: import seaborn as sns
sns.scatterplot(x=arr_img_2d[:, 0], y=arr_img_2d[:, 1], hue=df_labels.loc[:, 'labels'])
```

```
Out[13]: <AxesSubplot:>
```



Our visualization looks something like this:

 scatterplot of transformed data

We can see that there are 2 different groups emerging. The *I* data points form a crescent, which is mainly located in the negative part of the y-axis. The *C* data points, on the other hand, are more likely to be found in the positive part of the y-axis and in the negative part of the x-axis. The x-axis represents the 1st principal component and the y-axis represents the 2nd principal component. What do the principal components contain?

The principal components are the directions in which the data show the greatest variance. If you look at the proportions of the original features in these directions, you can get information about the principal components. Print the first 20 values of the first principal component. These can be found in the attribute `my_pca.components_`.

```
In [14]: pca.components_[0, :20]
```

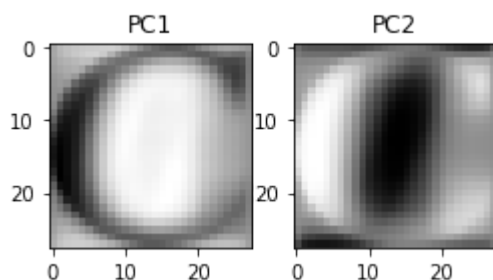
```
Out[14]: array([0.03170844, 0.03630409, 0.03861286, 0.03998485, 0.04098092,
        0.04146886, 0.04178027, 0.04228168, 0.04239474, 0.04089424,
        0.03684698, 0.03117284, 0.02527553, 0.02003297, 0.01607622,
        0.01339147, 0.01177079, 0.01177003, 0.01324852, 0.0161578 ])
```

If we multiply the values of the principal components by the respective features and sum up the products, we get the values of the transformed data. So in this case, they tell us how much each pixel contributes to the total variance in the respective component.

We can see that the first 10 pixels contribute about the same amount, since the first 10 values of the main component are about 0.04. The next 10 values are slightly lower. We can't read too much into this just yet. But since we are looking at pictures, we can also interpret the principal components as pictures. Run the following cell to display them.

```
In [15]: fig, axs = plt.subplots(ncols=2, figsize=[4,8]) # define figure with 2 axes
axs[0].imshow(pca.components_[0].reshape(28, 28), cmap='gray') # visualize the first
axs[1].imshow(pca.components_[1].reshape(28, 28), cmap='gray') # visualize the second
axs[0].set_title('PC1') # set titles for the axes
axs[1].set_title('PC2')
```

```
Out[15]: Text(0.5, 1.0, 'PC2')
```



We can see that the first principal component is the negative of a typical *C*. Where the letter is, the image is black (low values), otherwise the image is white (high values). With the first component we can measure how much a letter image resembles a typical *C*.

This is the reason that the data points with a *C* were mainly found in the negative area of our scatter plot. They usually behave in exactly the opposite way to the negative, which is PC1.

The second principal component is a mixture of the positive of a typical C and the negative of an I.

This explains why our C data points are mainly in the positive area of the y-axis. They are largely consistent with the positive of the C.

The I data points, on the other hand, are much more likely to be located in the negative part of the x-axis. They usually behave in exactly the opposite way to the negative of the I in the second principal component: They have a bright vertical line in the middle, which becomes wider at the top and bottom of the image. The first principal component shows no similarities to an I, so these data points are evenly distributed in both the positive and negative areas of the x-axis.

Congratulations: You have used `PCA` to represent high-dimensional data in 2 dimensions. And you saw that the data points divide up. This was because the principal components reproduce typical shapes of the letters. It is often a good idea to get an overview of the structure of the data by visualizing it with `PCA`.

Defining the number of principal components using the content of the information

So far, we have used regression metrics to determine the number of principal components or done this based on a visualization. However, `PCA` is an unsupervised learning method. We should therefore be able to find a good value for `n_components` even without using regression or classification. The metric which we can use to determine this is the explained variance of the components. Now let's take a look at how to do this. Instantiate `PCA` with `random_state=10` as `pca` again, but this time do not use the `n_components` parameter. Then `PCA` generates as many main components as there are features. Fit `pca` to the data in `arr_img`.

```
In [16]: pca = PCA(random_state=10)
pca.fit(arr_img)
```

```
Out[16]: PCA(random_state=10)
```

In principle, we'll now proceed in the same way as we did when we determined the number of clusters in the last chapter with the *elbow method* (see *k-Means - Determining the Number of Clusters in chapter 3*). We propose the percentage of variance based on the number of components. `pca.explained_variance_ratio_` now contains 784 values, the same number as the number of features we originally had. This is always the proportion of the total variance that can be explained by the respective component. However, the best way to do this is to use the cumulative sum of these values. For two features we can explain the proportion given by the sum of the first two values. We can calculate this cumulative sum with the `np.cumsum()` function. Pass `pca.explained_variance_ratio_` to it and store the result as `cumulative_sum_ratios`, and print the first 3 rows.

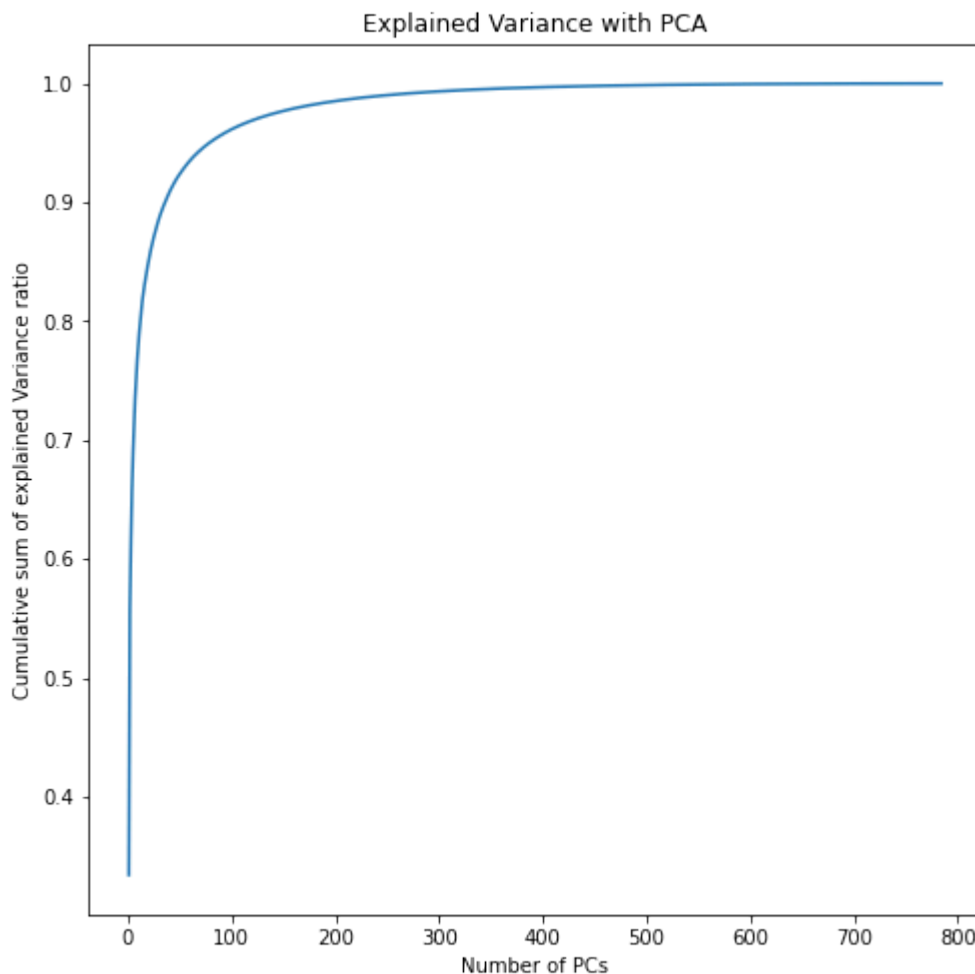
```
In [17]: import numpy as np
cumulative_sum_ratios = np.cumsum(pca.explained_variance_ratio_)
cumulative_sum_ratios[0:3]
```

```
Out[17]: array([0.33416344, 0.55857857, 0.6142112 ])
```

With 3 components we can already reproduce almost 60% of the variance.

Now create a line chart where the values of the explained variance (`cumulative_sum_ratios`) are on the y-axis with the corresponding number of components on the x-axis (they start with 1).

```
In [18]: fig, ax = plt.subplots(1, figsize=(8, 8))
ax.plot(range(1, len(cumulative_sum_ratios)+1), # we have at least 1 component
        cumulative_sum_ratios)
ax.set(title='Explained Variance with PCA',
        xlabel='Number of PCs',
        ylabel='Cumulative sum of explained Variance ratio');
```



You can see that the line starts by increasing sharply. Each component contributes to a large part of the variance. The curve then later flattens out considerably. Typically, you choose the number of components from which only little content information is added. In this case we recommend a value between 100 and 200.

But `sklearn` makes our choice even easier. We can specify values between 0 and 1 for `n_components`, and `sklearn` then chooses the number of components so that at least this portion of the variance can be reproduced by the components. Try it out. Instantiate `PCA` with the value `n_components=0.98` and fit it to `arr_img`. How many components are generated?

```
In [19]: pca = PCA(n_components=.98)
pca.fit(arr_img)
len(pca.components_)
```

Out[19]: 166

We get 166 components. This is just under a fifth of the original number of features. So we can reduce the size of the data to about 20% of its original size and lose only 2% of the total variance!

Let's look at how this affects the quality of the images. Transform the data and store it as `arr_img_pca`.

```
In [20]: arr_img_pca = pca.transform(arr_img)
```

If we now transform the data back, some of the quality should be lost. Use `pca.inverse_transform()` on `arr_img_pca`. This will transform the data back. Save the complete inverse transformation as `arr_img_reconstructed`.

```
In [21]: arr_img_reconstructed = pca.inverse_transform(arr_img_pca)
print(arr_img_pca[:4])
print(arr_img_reconstructed[:4])
```

```

[[-3.94107838e+00  3.07995723e+00 -4.92534588e+00  1.30708275e+00
  2.98850717e+00 -1.40827233e+00 -2.91407044e+00  2.53463552e+00
 -1.23253461e+00  3.28813156e+00 -7.88214663e-01 -7.48350772e-01
  1.13240810e+00 -1.43484396e+00  6.61162638e-02 -1.24162182e+00
 -7.48428969e-01 -1.60237373e+00 -5.03549554e-01 -4.74613711e-01
  1.84561406e-02  4.25375893e-01 -6.30355852e-01  1.70291665e-01
 -2.92783968e-01  1.32400906e+00 -3.78022274e-01  3.41083050e-01
  1.23449704e-01  3.29439346e-01 -4.42667841e-01  4.96937537e-01
  2.26875202e-01  5.65073645e-01  3.85588540e-01  1.52531403e-01
  1.35413121e-01 -9.17010553e-01 -5.48720949e-01 -1.27119418e-01
 -6.83400916e-01  3.46736361e-01  1.02617352e-03 -4.97374588e-02
  2.82389406e-01 -3.32955479e-02  2.02476361e-01  2.39661709e-01
  4.28471176e-01  3.69357415e-01 -6.82475615e-01 -1.94217991e-01
 -3.49646963e-01 -3.01324365e-01 -5.65208219e-02 -5.85826958e-01
 -3.89846531e-02 -1.48969187e-01  4.07415576e-03 -2.16374520e-01
 -1.09299757e-01 -2.78931767e-01 -1.05175455e-01  1.48487333e-01
 -1.83850652e-01  1.28913368e-01 -8.48873320e-02 -8.08369103e-01
 -2.22563585e-01  5.59954915e-02  1.70549427e-02  6.78277275e-02
 -8.78199490e-03 -1.29487495e-01  7.87045538e-03 -1.56156684e-01
  1.06110242e-01 -3.34885489e-01  4.22796706e-01  4.07537560e-02
  8.74248994e-02 -7.56056357e-02 -1.31120300e-01 -1.38969079e-01
  1.01321414e-01  2.76634811e-01 -8.55991516e-02 -2.23960858e-01
  5.49754704e-02  1.27114291e-01 -8.84556097e-02  1.66071447e-01
  2.04404658e-02  9.37744569e-02 -3.42351300e-01  9.50769409e-02
  2.17549351e-01 -2.91461494e-01 -1.20341426e-01  6.23723644e-02
  8.98935708e-02  2.46054542e-01  1.66061002e-02  1.97029101e-01
  2.53880663e-02 -1.55646842e-02  2.15752227e-01 -4.40120676e-01
 -4.14690550e-02 -7.06497038e-02  1.23389463e-01 -1.39853567e-02
  1.79136054e-01 -6.71792583e-02 -4.62113893e-01  1.23057667e-01
  1.20214336e-01  2.46915024e-01  1.21006981e-01  1.33006362e-01
  1.26862536e-02  3.76457012e-02  1.28276355e-01 -6.13510382e-02
 -1.28613281e-01 -1.89687536e-01 -2.01330997e-02  1.04269639e-01
 -6.28286381e-02 -1.81060896e-01  1.03244800e-02 -4.01281851e-02
 -4.87912331e-02  3.55641942e-02 -9.75803188e-02 -2.95231268e-02
 -1.29442559e-02 -2.20857468e-01  1.28145257e-01 -1.80009829e-01
 -1.00055045e-01 -1.00085427e-01  9.20205999e-02 -2.20415434e-02
  9.08253057e-03 -1.61458250e-01  5.36958560e-02 -1.01697982e-01
  1.31720856e-01  1.90568705e-02  1.90936027e-01  1.46483462e-01
 -1.04173805e-01 -5.61339524e-02  1.57934422e-02  8.78685978e-03
 -1.27804956e-01 -1.28613293e-01 -2.16674712e-01 -2.58284150e-02
 -8.09438224e-02  1.80126352e-02  1.70566957e-01  1.91094840e-01
  4.26433793e-02  8.07781015e-02]
[-1.70766081e+00  5.22477683e+00  2.98496890e+00 -5.09702492e+00
 -2.90070215e+00  4.58287327e+00 -1.63031341e+00  3.24435514e+00
  3.22140624e+00  1.27697253e+00  7.67935926e-01  2.68595791e-01
  8.55419194e-01 -3.06700290e+00  3.82338789e-01 -1.00254394e+00
 -6.14405107e-01  3.15045573e-01  2.39928671e-02 -5.04869216e-01
 -5.46598170e-01  1.15695173e+00 -2.13143574e+00  2.48930543e-01
  5.04181453e-02  3.83186034e-01 -2.84874557e-01 -6.09716170e-01
  1.48466324e-01 -3.30475005e-02 -3.53934781e-01  1.82370175e-01
  2.27449873e-01 -8.97224946e-01  7.35083391e-01 -4.94301866e-01
 -1.55457602e-01 -3.32522626e-01  3.85421884e-01  1.90511819e-01
  2.83296664e-01  3.06719181e-01  5.50425447e-01  6.31990014e-01
  5.81674325e-03  7.37931539e-01 -4.42369901e-01  1.13971590e-01
  1.13848870e-01  2.82979166e-01  1.32365950e+00  5.77663625e-01
 -2.73190481e-01 -6.76003896e-01 -2.16685449e-01 -2.37331397e-01
 -2.73756791e-01 -5.56785323e-01 -2.55241270e-01  2.11923953e-01
  1.73138389e-01 -1.60494149e-02  1.76972805e-01 -2.83192135e-01
 -4.07187460e-01  2.65502291e-01  1.44103196e-01 -6.87938651e-02
  2.98850744e-01 -3.41122418e-02  2.14885062e-01 -1.01925871e-02

```

7.06859423e-03 -3.55206840e-01 4.70184782e-02 -1.16957266e-01
-9.66012603e-02 3.04076849e-01 5.47535510e-02 -3.23461879e-01
4.03143726e-01 2.23851270e-01 2.85302282e-01 -1.42221420e-01
-2.13273441e-01 1.38047941e-01 -2.99485767e-01 3.39666387e-01
-2.63327049e-01 3.51788424e-01 -1.21805050e-01 1.02059766e-01
3.70402807e-01 6.48762156e-03 3.77861088e-01 -6.26522227e-02
-2.57339780e-02 7.08432172e-02 3.71114011e-01 3.70323102e-02
1.60498482e-01 -3.12222089e-02 1.43613673e-01 -5.61500799e-02
-1.67390374e-01 4.30263268e-02 -1.03877649e-01 -2.50203070e-01
-1.48641693e-01 9.00487273e-02 -5.88749594e-02 9.51672346e-05
1.63713323e-01 -2.56247292e-01 -8.90288115e-02 -1.07520469e-02
-1.67705210e-01 -7.31030968e-02 -1.38992391e-01 8.35605067e-02
-2.52198440e-01 9.13626778e-05 -1.80446099e-01 5.40983172e-02
1.79012923e-02 2.12107853e-01 -2.90847341e-02 -2.12636966e-01
2.95858629e-01 -2.13545277e-01 1.02794558e-01 -1.50869904e-01
-1.39526420e-01 4.49942379e-02 1.08694621e-01 -9.85759291e-02
1.58903615e-01 3.75794460e-02 -1.87608948e-01 -2.94663675e-02
1.98793529e-01 2.32778041e-01 -2.67507988e-02 1.74651479e-01
3.62020745e-02 -8.98829927e-02 -1.66734379e-01 -1.30804512e-01
2.78979852e-02 7.86771578e-02 -4.29597688e-02 -2.68534213e-02
-3.48569808e-02 -1.73529778e-01 -8.58761701e-03 8.83539716e-03
6.57172075e-03 -7.10098819e-02 -6.68740890e-02 -1.79849203e-01
-9.16307476e-03 1.18469261e-01 3.46722833e-02 1.21819267e-02
1.18412044e-01 1.07112186e-01]
[-3.76606674e+00 7.08811361e+00 -3.53285201e+00 1.27394828e-01
-9.15585528e-03 -1.87470314e+00 -3.48021599e+00 7.43694827e-01
-2.75229946e-02 6.49134979e-01 -8.36965211e-01 -8.24514796e-01
-1.38135354e+00 9.84066990e-01 5.12634695e-01 -1.30634734e+00
4.23682878e-01 -1.59366097e+00 -1.11460444e+00 1.29745016e-01
-9.16078360e-01 1.11437643e-01 3.52413685e-01 2.29586820e-01
-4.61091332e-02 2.50645134e-01 -3.01453497e-01 -1.20862957e-02
3.42640484e-02 3.37925701e-01 4.10621859e-01 -4.92649857e-01
1.76831321e-01 -1.43366166e-01 1.86348196e-01 2.10348428e-02
5.42631766e-03 -3.16137850e-01 1.52139932e-01 6.87988663e-01
-7.08443378e-02 9.99477756e-01 -2.37605309e-01 7.67923144e-01
-1.82007173e-01 -6.04462904e-01 6.00784407e-01 3.23713640e-01
1.89668125e-02 1.79124434e-01 -3.34482887e-01 3.59917138e-01
-1.80934248e-01 3.07843236e-01 1.82179324e-01 -1.22164234e-01
4.93052658e-02 1.52008984e-01 -5.51144431e-01 -4.37200293e-01
2.34421669e-01 8.46288758e-03 9.81505982e-01 -4.47510546e-01
4.51294522e-01 2.44049870e-01 -3.19253632e-01 6.04861685e-03
-4.26031896e-01 -4.47783725e-01 7.02572186e-02 -1.18421425e-01
1.60984711e-01 -2.32841926e-01 -1.81135483e-01 5.04288477e-02
-1.01755793e-01 -4.09263740e-01 9.39712601e-02 3.92622905e-01
9.13790587e-04 2.23619427e-02 -3.22342162e-01 -4.54190209e-01
2.96979137e-01 -4.84266389e-01 2.10178392e-01 -2.00520755e-01
-1.33135112e-01 9.72574374e-03 -1.04249037e-01 -3.55832539e-01
4.99115282e-02 -4.64889846e-02 -4.62294680e-01 -3.88050501e-01
1.13106043e-01 -4.66644401e-02 4.12609113e-01 -2.42017640e-01
-9.73206008e-02 -1.41634257e-01 -2.67929414e-01 8.13571856e-02
3.34511788e-01 -1.42127563e-01 9.62592636e-02 -4.18831499e-01
-3.18504426e-01 2.65569056e-01 1.30954784e-01 -1.43502065e-01
-4.24917053e-01 8.71261428e-02 2.44800004e-01 -1.67263522e-01
3.15122330e-01 -1.00038454e-01 2.02317836e-01 3.37217854e-01
-8.42525070e-02 1.80667606e-01 2.49932095e-01 8.64815042e-02
-7.67861719e-02 -5.90251954e-02 -3.91480367e-01 -1.91013078e-01
2.79633821e-01 -1.58179283e-02 -2.40443889e-01 -2.20229695e-01
4.11701669e-01 4.46406555e-01 -1.28224398e-01 -3.77525333e-02
-2.12102174e-01 2.00306424e-01 -1.82598139e-01 -7.70139439e-02
2.29073796e-01 -5.08985522e-02 3.69978357e-01 -1.71273641e-01

```

-1.29707567e-01 -1.34363195e-01 1.18551174e-01 -2.16463150e-01
8.61789475e-02 2.16174654e-01 -7.43306623e-03 1.47501378e-01
3.64577849e-01 9.90774211e-02 2.53614167e-02 -8.44870201e-02
3.14961485e-02 1.07015834e-01 -2.56340241e-02 -3.03121979e-02
1.33724337e-01 -1.64882098e-01 -2.11263661e-01 -1.59267695e-01
2.87510764e-02 1.08914403e-01]
[-5.61425731e+00 5.14690132e+00 -2.57841760e+00 -7.90267702e-01
-2.10941826e+00 -1.80861268e+00 -1.75592004e+00 -3.04447449e-01
6.05972916e-01 8.70145222e-01 2.81412420e-01 -1.22113565e+00
-1.57126639e+00 -1.12308136e+00 1.39353617e+00 -2.26106519e+00
-8.60586137e-01 1.33693639e-01 -2.84267109e-01 3.07035481e-01
8.10341987e-01 -1.50895384e-01 5.42429593e-01 1.36143530e+00
-2.72986376e-01 -5.22165012e-01 -1.41066628e+00 -4.84530082e-01
1.68732111e-01 3.44515877e-01 3.60109188e-01 6.96454372e-01
2.07129282e-01 2.86397240e-01 3.00409254e-01 4.12204069e-01
-4.58571152e-01 4.95384432e-01 -2.28765320e-01 8.41160951e-01
5.27415615e-02 7.41012398e-01 -3.90123846e-01 -4.88772548e-01
1.22061868e-01 -1.64131663e-01 -1.50495026e-01 5.67200114e-01
-5.60174889e-01 -1.17217396e+00 -9.70570717e-02 2.95628922e-01
1.70273402e-01 5.31870809e-01 1.63161013e-01 -1.11027952e+00
-8.92172823e-01 -2.76307466e-01 -2.72094837e-01 -2.02969566e-01
3.52420550e-01 6.15430745e-01 9.36281902e-02 3.35525656e-02
4.05165352e-01 -8.89515137e-01 -4.49374605e-01 1.73361868e-01
-4.79891278e-01 1.76244649e-01 -2.39483319e-01 -8.05307195e-02
4.31069153e-01 6.86347122e-01 1.92530111e-01 8.67798096e-02
-3.58629356e-02 -1.43980461e-01 -2.61686042e-02 -8.09362689e-02
4.05874486e-01 1.00813519e-01 -4.95969273e-01 -5.19148152e-01
-1.76874135e-01 -3.45182996e-01 2.57028602e-01 7.78267134e-02
-2.19541380e-03 6.36048652e-02 7.34985852e-02 -1.48315198e-03
-1.45981786e-01 -9.21359597e-02 4.25185780e-01 -8.77450774e-02
2.37343615e-01 -1.55542176e-01 -9.08396490e-02 -1.43863880e-01
-6.80880227e-01 1.92085178e-01 1.13845859e-01 -7.48472140e-02
1.84717475e-01 -2.90059091e-01 3.24141818e-01 2.93851277e-01
5.06694682e-02 -4.89499620e-01 1.30885754e-01 -5.74638274e-02
1.67772497e-01 -3.35276893e-01 1.18288382e-02 7.74541244e-03
-5.63927932e-02 -1.87744141e-01 -2.36871862e-03 -1.05972344e-01
-2.17088966e-01 3.49965282e-01 -3.48540358e-01 -4.48351815e-02
2.32698690e-02 9.87463976e-02 -7.66926428e-02 -2.60644231e-01
-4.16636973e-02 -1.35674147e-01 2.39077651e-02 -2.63232089e-01
-2.20668816e-01 -3.97226626e-01 -3.69025198e-02 2.37872627e-01
1.46396287e-01 2.78630499e-02 2.72868286e-02 1.54150259e-01
9.78390408e-02 2.97409186e-01 -1.19909877e-01 -4.62298223e-02
-1.14355214e-01 -4.61936646e-02 -1.52574891e-01 -6.42401725e-02
8.39353635e-02 1.75812966e-03 1.99914393e-01 -5.84711444e-02
3.14391874e-01 -2.42517457e-01 1.51580735e-01 -3.10124472e-02
1.11093614e-01 -1.65465238e-01 -2.79767589e-01 -5.88741721e-01
8.63709586e-02 -3.78660872e-02 -6.86545439e-02 2.89700616e-01
-7.99641577e-02 -1.27986498e-01]]
[[ 0.00774493 0.00188994 -0.00850952 ... -0.0077041 -0.02224797
-0.029094 ]
[ 1.0013181 0.99339716 0.96654934 ... 1.04247127 1.0602615
1.0204549 ]
[ 0.03420235 -0.04416801 -0.02819148 ... -0.02608369 -0.03343723
0.03950778]
[ 0.05377897 0.02709193 0.00517722 ... -0.02372381 -0.04482338
-0.03584934]]

```

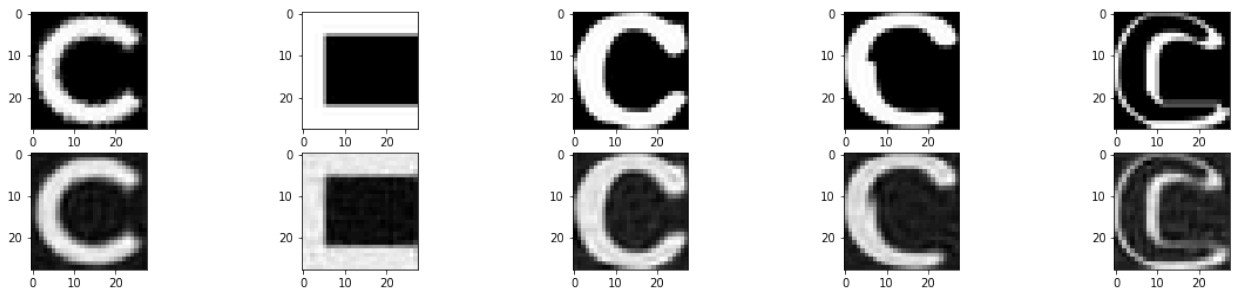
Now let's take a look at the original images and the corresponding reconstructions. Run the following cell to do this. It repeats the initial and inverse transformation of the data. It generates

an image with two rows. In the first row you can see the original letters. In the second row you can see the reconstructed images.

```
In [22]: # fit pca
pca = PCA(n_components=0.98)
arr_img_pca = pca.fit_transform(arr_img)

# reconstruct the data
arr_img_reconstructed = pca.inverse_transform(arr_img_pca) # reverse pca

# define figure with 10 axes in 2 rows
fig, axs = plt.subplots(nrows=2, ncols=5, figsize=[20, 4]) # define figure with 10 axes
# plot images of original C in the first row
for i in range(5):
    axs[0, i].imshow(df_img.loc[i, :].values.reshape(28, 28), cmap='gray') # use cmap
# plot images of reconstructed C in the second row
for i in range(5):
    axs[1, i].imshow(arr_img_reconstructed[i, :].reshape(28, 28), cmap='gray') # reconstructed
# for i in range(5):
#     axs[2, i].imshow(arr_img_pca[i, :].reshape(28, 28), cmap='gray') # Transformed a
```



We can see that the intensity of the letters has been reduced slightly. However, the letters are still clearly visible after the reconstruction.

We have seen in this lesson that the principal components contain information about the data. Despite the reduced dimensions, the original data can be reconstructed relatively well. So PCA offers us a way to extract the most important information from the features. Since the principal components are made up of the old features, it is not always obvious at first glance what they mean. It's helpful here to be clear which original features are represented how much in the respective principal component. However, it is not always necessary to interpret the components.

Congratulations: Your department manager is very happy with your data preparation. You have seen how to select the number of principal components for PCA using the explained variance. The transformed data has significantly fewer features than the original data. In the next lesson we'll look at how this can affect the quality and speed of a classification.

Remember:

- Dimensionality reduction helps to visualize high-dimensional data and can make its structure more understandable

- Pass a number between 0 and 1 to `n_components` to determine the number of components divided by the proportion of the declared variance
- A visualization of the explained variance can help you to choose `n_components`

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.
