Classifying Images

Module 1 | Chapter 4 | Notebook 8

Sometimes it's not so important whether the predictions are a bit better or worse. Instead, it's more important that it doesn't take too much time to train the algorithms. And this is often the problem, especially if you have a lot of high-dimensional data. Principal component analysis can help us here to compress the data and speed up the model training. We'll take a look at that now. You will learn:

- How you can measure how long it takes to run your code
- How principal component analysis can help you speed up model training

---

Speeding up training with `PCA`

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.

You have already prepared the data in the last lesson. Now it is time to predict the letters. Run the following cell to import and prepare the data.

In [1]:
```
# the data is in the database letters.db
import sqlalchemy as sa
engine = sa.create_engine('sqlite:///letters.db')  # connect to the database
connection = engine.connect()  # create the engine

# import pandas and read the data and the labels
import pandas as pd
df_img = pd.read_sql('SELECT * FROM images', con=connection).drop('index', axis=1)
df_labels = pd.read_sql('SELECT * FROM labels', con=connection)

connection.close() # close connection to database

# import and instantiate the scaler and pca
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
scaler = StandardScaler()
pca = PCA(n_components=0.98, random_state=10)  # keep 98 percent of the variance

arr_img_std = scaler.fit_transform(df_img)
arr_img_std_pca = pca.fit_transform(arr_img_std)
```

The number of features a dataset contains affects the duration of the training stage when making predictions. We'll try it out now. For this purpose we'll consider our letters as a classification problem. We want to know how well we can determine whether the image section is a *C* or an *I*.

In *Module 1 Chapter 2* we got to know the classification algorithm k-Nearest-Neighbors. We'll use this algorithm this time too. Import `KNeighborsClassifier` from `sklearn.neighbors`. Create an instance with the name `model` and the hyperparameter `n_neighbors=10`.

In [2]:
```
from sklearn.neighbors import KNeighborsClassifier
model = KNeighborsClassifier(n_neighbors=10)
```

To measure time, we use the *magic command* `%%timeit`. If you write the `%%timeit` command at the beginning of a cell, the code will be executed over and over again until 2 seconds have elapsed. A loop is executed several times in the background, which runs the cell code very often. The number of tests can be specified with the commands `-n` and `-r`. With `-n` you specify the repetitions within the loop. With `-r` you specify how often the loop is executed. The command then returns a summary of how long it took to run the code.

Write `%%timeit -n 10 -r 3` at the beginning of the following cell. Then fit `model` to `arr_img_std` and `df_labels`. Use the `labels` column of `df_labels` as target vector. You will get the average duration of the 30 executions as the output. This cell can take a few seconds to complete.

In [3]:
```
%%timeit -n 10 -r 3
model.fit(arr_img_std, df_labels.loc[:, 'labels'])
```

```
3.98 ms ± 41.4 µs per loop (mean ± std. dev. of 3 runs, 10 loops each)
```

Repeat this now with the difference that you use the compressed array in `arr_img_std_pca` as a feature matrix instead of `df_img`.

In [4]:
```
%timeit -n 10  -r 3 model.fit(arr_img_std_pca, df_labels.loc[:, 'labels'])
```

```
2.73 ms ± 52.5 µs per loop (mean ± std. dev. of 3 runs, 10 loops each)
```

In our case, it was 10 times faster to fit the model using the transformed data. This value may vary slightly for you.

How does the quality of the forecasts change with the use of `PCA`? To find out, we'll create two pipelines. One with `PCA` and one without. Import `Pipeline` from `sklearn.pipeline`. Create two instances. Name the first one `pipe`, containing two steps, standardization and k-nearest neighbors. Name the second instance `pipe_pca`. This should also contain the `PCA` step between the two steps. Use the variables `scaler`, `pca` and `model` we already created based on the pattern `scaler` - `pca` -> `model`.

In [5]:
```
from sklearn.pipeline import Pipeline

# without pca
```

```
pipe = Pipeline([('std', scaler),
                 ('knn', model)])

# with pca
pipe_pca = Pipeline([('std', scaler),
                     ('pca', pca),
                     ('knn', model)])
```

For which number of neighbors do we get the best predictions? This could change with the use of `PCA`.

Therefore, import `GridSearchCV` from `sklearn.model_selection` to answer this question. You already learned about `GridSearchCV` in *Grid Search (Module 1 Chapter 2)*. `GridSearchCV` uses cross-validation to search for the best hyperparameter from the given search space. Create a dictionary and call it `search_space`. This dictionary consists of the name of the hyperparameter and a list of numbers used for it. When naming the hyperparameter, it's important to specify the name of the `Pipeline` step followed by two underscores. So it should have the *key* `'knn__n_neighbors'` if you named the last step `knn`. The dictionary should contain `[1, 2, 3, 4, 5, 10]` as the *value*.

In [6]:
```
from sklearn.model_selection import GridSearchCV
search_space = {'knn__n_neighbors': [1, 2, 3, 4, 5, 10]}
```

Another parameter we need to specify for `GridSearchCV` is `cv`. Up to now we have always assigned an integer to `cv`. It indicates how many steps the cross-validation consists of. With our data, however, it is now the case that first all the *C* letters are in the data and then all the *I* letters.

As a result, `GridSearchCV` may only train on *C* and validate with *I*. This can lead to poor results. So we should mix the labels first, but unfortunately `GridSearchCV` itself has no parameter for this. But we can give the parameter `cv` the cross-validation generator `KFold`, which we learned about in *Evaluating Models with Cross Validation* (*Module 1 Chapter 2*), which can mix the values.

So import `KFold` from `sklearn.model_selection` and instantiate it with the parameters `n_splits=3`, `shuffle=True` und `random_state=10` under the name `kf`.

In [7]:
```
from sklearn.model_selection import KFold
kf = KFold(n_splits=3, shuffle=True, random_state=10)
```

Now instantiate `GridSearchCV` under the name `neighbor_search`. Pass the following parameters:

- `estimator=pipe`
- `param_grid=search_space`
- `scoring='f1'`
- `cv=kf`

```
In [8]: neighbor_search = GridSearchCV(estimator=pipe,                  # estimator
                     param_grid=search_space,                         # the grid of the grid s
                     scoring='f1',                                    # which measure to optim
                     cv=kf,                                           # number of folds during
                     n_jobs=-1 )                                      # number of CPU cores to
```

Our labels in `'df_labels.loc[:, 'labels']'` are given as letters. `GridSearchCV` cannot handle these. Thankfully, `sklearn` provides us with the transformer `sklearn.preprocessing.LabelEncoder`. You can read more about it (in the `sklearn` documentation). When you call the `my_labelencoder.fit_transform()` function, it replaces the individual values with ascending numbers starting from 0. So in our case '*C*' becomes 0 and '*I*' becomes a 1. Note that this encoder is only applicable to target labels. Instantiate `LabelEncoder` as `label` and store the return values of the `fit_transform` method in the variable `target`. Print `target` and see for yourself what `LabelEncoder` has done.

```
In [9]: from sklearn.preprocessing import LabelEncoder

        label = LabelEncoder()
        target = label.fit_transform(df_labels.loc[:, 'labels'])
        print(target)
```

```
[0 0 0 ... 1 1 1]
```

Now fit `neighbor_search` to `df_img` and use `target` as the target vector. Then print the best F1 value and the best parameters. Fitting the model this time will take a relatively long time, so this might be a good time to have a coffee break.

```
In [10]: neighbor_search.fit(df_img, target)
         print(neighbor_search.best_score_)
         print(neighbor_search.best_params_)
```

```
0.9716419001280641
{'knn__n_neighbors': 3}
```

With this simple model we've already achieved a very good *f1 score* of about 0.972 on the original data. The best number of neighbors is 3.

What about for the transformed data? To do this, we'll create the variable `neighbor_search_pca`, which uses `pipe_pca`. Run the following cell to do this.

```
In [11]: neighbor_search_pca = GridSearchCV(estimator=pipe_pca,
                                            param_grid=search_space,
                                            scoring='f1',
                                            cv=kf,
                                            n_jobs=-1,
                                            verbose=4)
```

Now repeat the fitting and printing of the best result and best parameters. But replace `neighbor_search` with `neighbor_search_pca`. This will take a while again.

```
In [ ]: neighbor_search_pca.fit(df_img, target)
        print(neighbor_search_pca.best_score_)
        print(neighbor_search_pca.best_params_)
```

```
Fitting 3 folds for each of 6 candidates, totalling 18 fits
```

For the transformed data we get a very slightly higher *f1 score* of about 0.973. The number of neighbors is still 3. So the quality of our forecast has not suffered from the fact that we now only have 169 features instead of 784 (see last exercise). This is ideal, as the memory requirements have been reduced and the performance increased at the same time.

Congratulations: Your boss is very satisfied with the prediction quality of your relatively simple model. They're glad you're keeping track of your training time too. So the proof of concept is a complete success!

Remember:

- The fewer features the data has, the faster you can train your model and make predictions
- Measure the execution time of a cell with `%%timeit`
