Principal Component Analysis as Part of the Data Pipeline

Module 1 | Chapter 4 | Notebook 5

In the last exercise, you encountered the principal component analysis. Now we'll use it as part of our data pipeline to reduce the features in our analysis of the wines. You will:

- Use the principal component analysis as part of a data pipeline
- Find the best number of principal components for the predictions

---

Determine number of principal components

Scenario: 1001Wines is an online retailer that sells wines through its website. Recently, 1001Wines took over a start-up company that specializes in producing wines synthetically. They want to use this expertise to create a separate product range. The customers of 1001Wines leave ratings for the products. This data will be used to predict how well new wines will be accepted by customers in the future.

Since you already created a good product recommendation algorithm for 1001Wines (see *Module 0, Chapter 2*), they came back to you for more help. They want you to analyze the composition of the wines in view of their ratings. They provided you with the respective wine characteristics in the file *wine_composition.csv*. The corresponding ratings are in the file *clustering_data-2.csv*. Both files are already in the current working directory.

So far you have looked at the wine data and made predictions. By creating features of a higher degree you slightly improved the result of the regression. We want to follow up on this. Run the following cell to do this. This is how you import the data and save it in the DataFrames `df_composition` and `df_rating`.

In [1]:
```
import pandas as pd
df_composition = pd.read_csv('wine_composition.csv')  # import the composition of the
df_rating = pd.read_csv('wine_rating.csv')  # import the ratings for the wines

df_composition.head()
```

Out[1]:

	fixed.acidity	volatile.acidity	citric.acid	residual.sugar	chlorides	free.sulfur.dioxide	total.sulfur.dio
0	7.0	0.27	0.36	20.7	0.045	45.0	1
1	6.3	0.30	0.34	1.6	0.049	14.0	1
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	1
4	7.2	0.23	0.32	8.5	0.058	47.0	1

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.

The ratings are currently on a 10-level scale. However, our data only contains the values 3 to 9. Although these are ordinal categories, we will use a regression to predict the values. Now we will run the linear regression again and see what average absolute error we get with it (as we did

in the lesson *Feature Engineering with Polynomials (Module 1 Chapter 4)*). Run the following cell to do this.

In [2]:
```
# import and create the regression model
from sklearn.linear_model import LinearRegression
model = LinearRegression()

target = df_rating.loc[:, 'rating']  # do not use id for target variable
features = df_composition.iloc[:, :-2]  # only use numeric values as features

# use cross-validation to evalute the model
from sklearn.model_selection import cross_val_score
cv_results = cross_val_score(estimator=model, X=features, y=target, cv=5, scoring='neg
print('negative mean absolute error:', cv_results.mean())
```

```
negative mean absolute error: -0.58104275094183
```

The error of the simple linear model is about 0.581. So we are about half a step off in the rating prediction. Let's compare this again with the polynomial model. Carry out the following steps:

- Import `PolynomialFeatures` from `sklearn.preprocessing`. Instantiate the *transformer* with the parameters `degree=2` and `include_bias=false` with the name `poly_transformer`.
- Import `Pipeline` from the `sklearn.pipeline` module. Create a `Pipeline` containing `poly_transformer` and `model`. For example, you can use the names `'poly'` and `'reg'` as names within the `Pipeline`. You can store the `Pipeline` itself in the variable `pipeline`.
- Use `cross_val_score`, to perform a 5-fold cross validation with `pipeline`. Use `'neg_mean_absolute_error'` for the evaluation. Then print the average of the cross-validation results.

In [3]:
```
from sklearn.preprocessing import PolynomialFeatures
poly_transformer = PolynomialFeatures(degree=2, include_bias=False)

from sklearn.pipeline import Pipeline
pipeline = Pipeline([('poly', poly_transformer),
                     ('reg', model)])

cv_results = cross_val_score(estimator=pipeline,
                             X=features,
                             y=target,
                             cv=5,
                             scoring='neg_mean_absolute_error'
                            )
print('mean absolute error:', cv_results.mean())
```

Out[3]:
```
-0.5771691803265556
```

The result is slightly better and is around 0.577. But now we have relatively many features. How many are there exactly? Print the number of columns of the feature matrix after transformation with `poly_transformer`.

In [4]:
```
featuresTransformed =  poly_transformer.fit_transform(features)
featuresTransformed.shape
```

Out[4]:
```
(6497, 77)
```

We now have 77 features. At the beginning we only had 11 different numerical columns. So we've added 66 features. These 66 additional features have slightly improved our prediction, but not significantly. So we can assume that many of the features have no added value for us. Either the rating is not particularly dependent on them, or some of them are highly correlated, meaning that they don't provide any new information.

If we reduce the dimensionality of the data, we can remove the unnecessary features and suppress the noise, for example. So let's see if we can improve our prediction. But how many features do we want in the end? When is the prediction the best? This number is a hyperparameter of our entire model. You saw how to optimally select the value for this hyperparameter in the lesson *Grid Search (Module 1 Chapter 2)*. Here we will use the `validation_curve()` function to visualize this.

First import `StandardScaler` from `sklearn.preprocessing` and `PCA` from `sklearn.decomposition`. Instantiate them and overwrite `pipeline` with a `Pipeline` containing the two new steps. Pay attention to the order the steps come in. First the new features must be created, then they should be standardized. Only then should the dimensionality reduction happen before the regression is performed.

Tip: You do not need to define hyperparameters when instantiating `PCA`. We will do this later with `validation_curve()`. Within the `Pipeline` you could use the names `'scale'` and `'pca'` for `StandardScaler()` and `PCA()`.

In [5]:
```
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA

pipeline = Pipeline([('poly', poly_transformer),
                     ('scale', StandardScaler()),
                     ('pca', PCA()),
                     ('reg', model)])
```

After we have created the corresponding `Pipeline`, we can now evaluate individual hyper parameters with `validation_curve`. It automatically separates between training data, which our `Pipeline` is trained with, and validation data, which it generates predictions for. It gives us the results for both training and validation. So we can use them to see if we end up overfitting with too many features and what the optimal number of features for us to generate is.

Import `validation_curve` from `sklearn.model_selection`. It has the following parameters, the values we need are in brackets:

```
validation_curve(estimator=model, # the model to be used (`pipe`)
                 X=DataFrame,     # the feature matrix (`features`)
                 y=DataFrame,     # the target vector (`target`)
                 param_name=str,  # the hyperparameter to be varied within
```

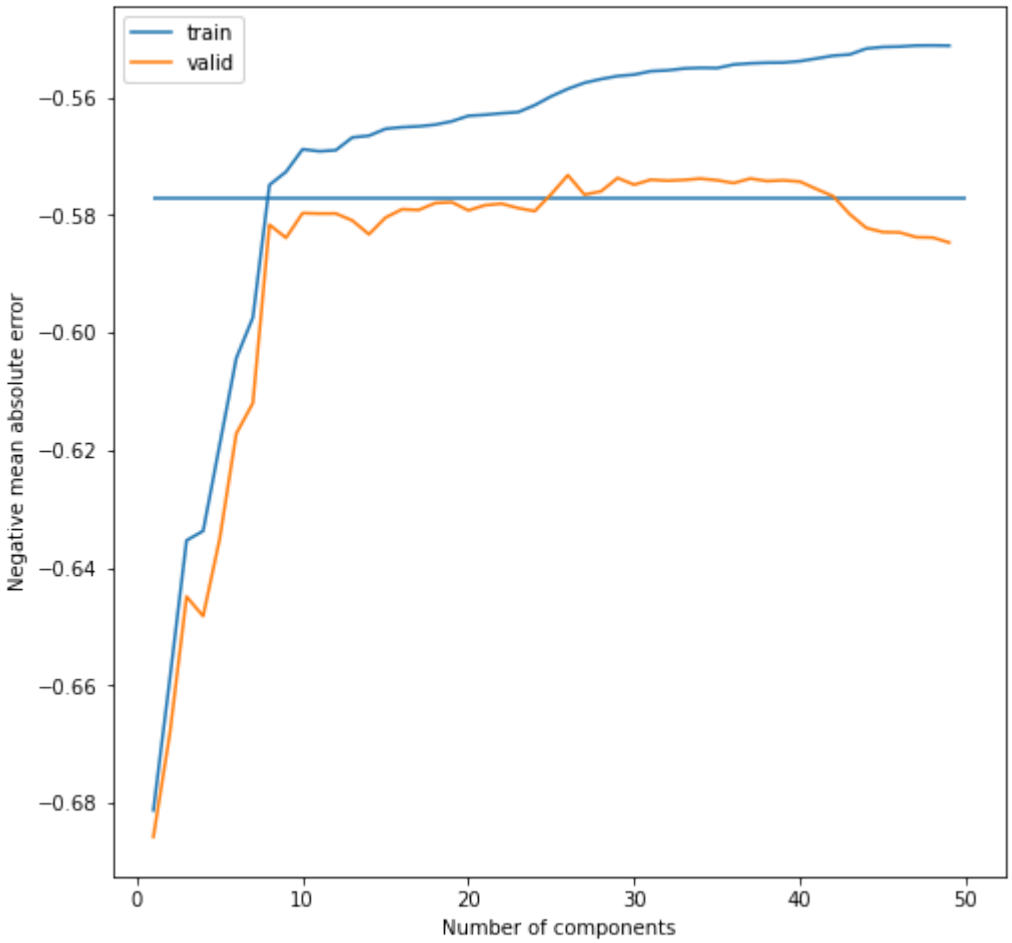
```
a part of the pipeline (`'pca__n_components'`)
                param_range=list,# the numbers to be used for the
parameter (`range(1,50)`)
                cv=int           # the number of cross validation folds
(`5`)
                scoring=str      # the evaluation metric
(`'neg_mean_absolute_error'`)
```

Run `validation_curve` with the appropriate parameters and store the result as `train_scores, valid_scores`. It may take a few minutes to run the cell. You can speed up the process by ending the kernels of any other DataLab tabs you have open and closing them.

In [6]:
```
from sklearn.model_selection import validation_curve

train_scores, valid_scores = validation_curve(estimator=pipeline,  # estimator (pipeli
                                              X=features,  # features matrix
                                              y=target,  # target vector
                                              param_name='pca__n_components',  # defir
                                              param_range=range(1,50),  # test these k
                                              cv=5,  # 5-fold cross-validation
                                              scoring='neg_mean_absolute_error')  # us
```

Now we have five scores for training and validation data for each number of principal components from 1 to 49. We are interested in the mean values of these 5 scores. Generate the mean values of `train_scores` and `valid_scores` with `np.mean()`. Remember to set the `axis` parameter correctly. Then create a visualization displaying the mean values of these two arrays as lines. The x-axis should contain the number of components. Insert a horizontal line at the value `-0.57717`. This is the mean value of the prediction without dimensionality reduction. The line will be helpful for us to compare.

In [7]:
```
import numpy as np
train_scores_mean = np.mean(train_scores, axis=1)
valid_scores_mean = np.mean(valid_scores, axis=1)

import matplotlib.pyplot as plt

fig, ax = plt.subplots(figsize=(8,8))
ax.plot(range(1,50), train_scores_mean, label='train')
ax.plot(range(1,50), valid_scores_mean, label='valid')
ax.hlines(y=-0.57717, xmin=1, xmax=50)
ax.set_xlabel('Number of components')
ax.set_ylabel('Negative mean absolute error')
ax.legend();
```



The line should look something like this:

You can see that the validation curve looks how you would expect. The more principal components we choose, the better the model will be in terms of training data. For the validation data, the error increases again slightly as the number of features increases. So the lower line goes down slightly towards the right. When the curve of the validation data is above the horizontal line, we have achieve an improvement with the model. Again, this is only a relatively small step. But at least we are improving. What is the best average absolute error we get now? For which number of components does this value occur?

In [8]:
```
print('best score:', max(valid_scores_mean))
print('position:', valid_scores_mean.tolist().index(max(valid_scores_mean)))
```

```
best score: -0.5732419895359975
position: 25
```

We get the best error value of about `0.573` for 26 principal components. So we have therefore reduced the number of features from 77 back down to 26. As a result, some of the noise seems to have disappeared and the model has improved slightly.

Congratulations: You have created a data pipeline which not only creates and standardizes features but also reduces the dimensionality and trains a regression model. You were able to use

`validation_curve` to determine the best number of principal components for `PCA`. You improved the prediction error a small amount.

Interpreting the principal components

A PCA usually helps us to avoid collinearity or simply to reduce the dimensions in our data. Unfortunately, the resulting models are not always as easy to interpret, as we first have to find out how our features form the principal components.

Now that we know that our model gives the best results with a PCA with 26 components, we should first build a pipeline so that we can analyze our model. Execute the next code cell for this.

In [9]:
```
pipeline = Pipeline([('poly', poly_transformer),
                     ('scale', StandardScaler()),
                     ('pca', PCA(n_components=26)),
                     ('reg', model)])

pipeline.fit(features,target)
```

Out[9]:
```
Pipeline(steps=[('poly', PolynomialFeatures(include_bias=False)),
                ('scale', StandardScaler()), ('pca', PCA(n_components=26)),
                ('reg', LinearRegression())])
```

Now we have a trained model. The PCA that we'll look at now is part of the pipeline. To access the methods and attributes of the PCA object, you can use `my_pipeline.named_steps['my_stepName']`. Using this syntax, you can access all steps of the pipeline and use them just as if you had defined them outside the pipeline. Start by storing the PCA from our pipeline in the variable `pca`.

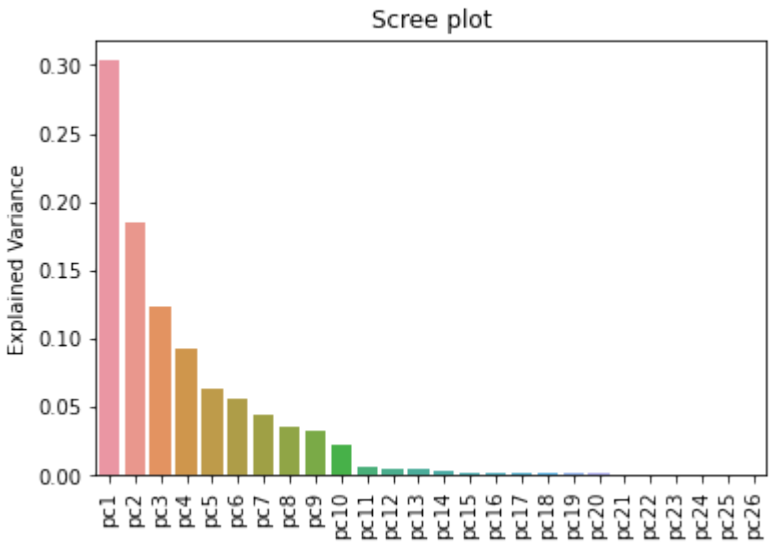
In [10]:
```
pca = pipeline.named_steps['pca']
```

Next we will see how well our 26 principal components explain the variance in the data. The technical name for this is a *Scree plot*. To create this kind of visualization, you need the attribute `my_pca.explained_variance_ratio_`, which contains the explained variances for each principal component. It is best to display the values in a column chart. Also give the columns meaningful names such as `'pc#'`

Tip: Try creating the names for the columns in a loop and storing them in the variable `pc_names`

In [11]:
```
import seaborn as sns

#create names
pc_names = ['pc{}'.format(i+1) for i in range(pca.n_components)]

#plot scree plot
ax = sns.barplot(x=pc_names,
                 y=pca.explained_variance_ratio_)
#style plot
```

```
ax.set(title="Scree plot",
       ylabel="Explained Variance")
plt.xticks(rotation=90);
```



After some tweaking, our *scree plot* looks like this:

The screeplot shows us that the first 10 principal components explain much more variance than pc11 to pc26. In particular, pc1 and pc2 together explain almost 50% of the variance. It is therefore worthwhile finding out which features make up these principal components. In the optional lesson *Optional: Principal Component Analysis Deep Dive (Module 1 Chapter 4)* you learnt that we can imagine the eigenvectors of the principal components as recipes. Mix x parts from one feature and y parts from another feature and get a principal component. Let's take a look at these recipes for `pca`. You can find them in the attribute `my_pca.components_`, which are called eigenvectors. Store the eigenvectors of `pca` as a `DataFrame` and use `pc_names` as the index.

In [12]:
```
eigenvectors = pd.DataFrame(pca.components_, index=pc_names)
```

As with all transformations with `sklearn`, the names of the features have been lost. Fortunately, `PolynomialFeatures` has a `get_feature_names` method that allows us to create the names of polynomial features when we pass a *list* with the original names of the features to the method.

First save the `PolynomialFeatures` object from our pipeline under the name `poly` using `Pipeline.named_steps[]`. Then use it to create `poly_feature_names` - a `list` containing the names of polynomial features.

Then you can use `poly_feature_names` to override the column names of `eigenvectors`.

In [14]:
```
PolynomialFeatures = pipeline.named_steps['poly']
poly_feature_names = PolynomialFeatures.get_feature_names(input_features=features.colu
```

```
eigenvectors.columns = poly_feature_names
eigenvectors.head()
```

Out[14]:

	fixed.acidity	volatile.acidity	citric.acid	residual.sugar	chlorides	free.sulfur.dioxide	total.sulfur.c
pc1	-0.069972	-0.110622	0.056803	0.148785	-0.076028	0.167887	0.
pc2	0.153235	0.079925	0.109280	0.080190	0.188528	0.008078	-0.
pc3	0.110272	-0.185304	0.256142	-0.082003	-0.055867	-0.020213	-0.
pc4	-0.096511	0.004370	-0.026403	-0.176756	0.097823	0.163961	0.
pc5	-0.066535	-0.114387	0.000238	-0.054139	0.201046	-0.040767	0.

5 rows × 77 columns

Now plot the composition of `'pc1'` using a bar chart.

In [ ]:
```
plt.figure(figsize=(15,5))
ax = eigenvectors.loc['pc1',:].sort_values(ascending=False).plot.bar()
ax.set(title="Eigenvector for pc1",
       ylabel="Importance in pc1");
```

After a little tweaking your visualization should look something like this:

For the *feature importance* it is irrelevant whether the numerical value is positive or negative. We wanted to apply the same code again to the same `DataFrame` but with absolute numbers and sort the columns by size. Then we can read the most important features on the far left. Create a corresponding visualizazion that gives you the top 10 features.

In [ ]:
```
plt.figure(figsize=(15,5))
ax = abs(eigenvectors).loc['pc1',:].sort_values(ascending=False).head(10).plot.bar()
ax.set(title="Eigenvector for pc1",
       ylabel="Importance in pc1");
```

For us, the three most important features of `'pc1'` are now clearly visible, they are:

1. total.sulfur.dioxide density
2. total.sulfur.dioxide
3. total.sulfur.dioxide

All three features have something to do with the sulfur dioxide content in wine. So this is one of the factors that 1001Wines should do to optimize the production of its synthetic wines.

Congratulations: 1001Wines thanks you for developing a model for predicting the ratings of wines according to their chemical composition so quickly. They are very satisfied with it and will use it to assess the quality of synthetically produced wines.

Remember:

- Validation curves can help you get a feel for the number of principal components to improve predictions
- First create the polynomial features, then standardize them and then you can use `'PCA'`
- Use `my_pca.components_` to evaluate the principal components. Here you can find the corresponding parts of the features that make up the principal 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.

The data was published here first: P. Cortez, A. Cerdeira, F. Almeida, T. Matos and J. Reis. Modeling wine preferences by data mining from physicochemical properties. In Decision Support Systems, Elsevier, 47(4):547-553, 2009.
