

[Home](#) ▶ [Regression](#) ▶ [Partial Least Squares Regression](#) ▶
Partial Least Squares Regression in Python

Partial Least Squares Regression in Python

📅 06/14/2018

Hi everyone, and thanks for stopping by. Today we are going to present a worked example of Partial Least Squares Regression in Python on real world NIR data.

PLS, acronym of [Partial Least Squares](#), is a widespread regression technique used to analyse near-infrared spectroscopy data. If you know a bit about NIR spectroscopy, you sure know very well that **NIR is a**

DO YOU LIKE



?

SIGN UP FOR OUR FREE
NEWSLETTER AND GET MORE
CONTENT DIRECTLY TO YOUR
INBOX.


SUPPORT NIRPY RESEARCH

secondary method and NIR data needs to be calibrated against primary reference data of the parameter one seeks to measure. This calibration must be done the first time only. Once the calibration is done, and is robust, one can go ahead and use NIR data to predict values of the parameter of interest.

In previous posts we discussed qualitative analysis of NIR data by Principal Component Analysis (PCA), and how one can make a step further and build a regression model using Principal Component Regression (PCR). You can check out some of our related posts here.

- [PCA classification of NIR spectra](#)
- [Variable selection with PLS](#)
- [Principal Component Regression](#)

IMP



PCR is quite simply a regression model built using a number of principal components derived using PCA. In our [last post on PCR](#), we discussed how PCR is a nice and simple technique, but limited by the fact that it does not take into account anything other than the regression data. That is, our primary reference data are not considered when building a PCR model. That is obviously not optimal, and PLS is a way to fix that.

In this post I am going to show you how to build a simple regression model using PLS in Python. This is the overview of what we are going to do.



Considering supporting us on [Patreon](#), to keep this blog and our [GitHub](#) content always free for everyone. Supporters have access to additional material and participate to our patron-only [Discord](#) community.

● [Become a patron](#)

KEEP IN TOUCH



Search

- [Classification](#)
- [Classification metrics](#)
- [Data Correction and Normalisation](#)
- [Data Operations and Plotting](#)

1. Mathematical introduction on the difference between PCR and PLS regression (for the bravest)
2. Present the basic code for PLS
3. Discuss the data we want to analyse and the pre-processing required.
We are going to use NIR spectra of fresh peach fruits having an associated value of Brix (same as for the [PCR post](#)). That is the quantity we want to calibrate for.
4. We will build our model using a cross-validation approach

Difference between PCR and PLS regression

Before working on some code, let's very briefly discuss the mathematical difference between PCR and PLS. I decided to include this description because it may be of interest for some of our readers, however this is not required to understand the code. Feel free to skip this section altogether if you're not feeling like dealing with math right now. I won't hold it against you.

Still here? OK here we go.

Both PLS and PCR perform multiple linear regression, that is they build a linear model, $Y = XB + E$. Using a common language in statistics, X is the predictor and Y is the response. In NIR analysis, X is the set of

- [Linear Discriminant Analysis](#)
- [Logistic Regression](#)
- [Multivariate Curve Resolution](#)
- [Neural Networks](#)
- [Outliers Detection](#)
- [Partial Least Squares Regression](#)
- [Perceptron](#)
- [Plots and Charts](#)
- [PLS Discriminant Analysis](#)
- [Principal Components Analysis](#)
- [Principal Components Regression](#)
- [Regression](#)
- [Regression metrics](#)
- [Regression Model Validation](#)
- [Ridge Regression](#)
- [Use Cases](#)
- [Variable Selection](#)

spectra, Y is the quantity – or quantities- we want to calibrate for (in our case the brix values). Finally E is an error.

As we discussed in the PCR post, the matrix X contains highly correlated data and this correlation (unrelated to brix) may obscure the variations we want to measure, that is the variations of the brix content. Both PCR and PLS will get rid of the correlation.

In PCR (if you're tuning in now, that is Principal Component Regression) the set of measurements X is transformed into an equivalent set $X' = XW$ by a linear transformation W , such that all the new 'spectra' (which are the principal components) are linearly independent. In statistics X' is called the factor scores. The linear transformation in PCR is such that it minimises the covariance between the different rows of X' . That means this process only uses the spectral data, not the response values.

This is the key difference between PCR and PLS regression. PLS is based on finding a similar linear transformation, but accomplishes the same task by maximising the covariance between Y and X' . In other words, PLS takes into account both spectra and response values and in doing so will improve on some of the limitations on PCR. For these reasons PLS is one of the staples of modern chemometrics.

I'm not sure if that makes any sense to you, but that was my best shot at explaining the difference without writing down too many equations.

PLS Python code

OK, here's the basic code to run PLS in cross-validation, based on Python 3.5.2.

```
1 from sklearn.cross_decomposition import PLSRegression
2 from sklearn.metrics import mean_squared_error, r2_score
3 from sklearn.model_selection import cross_val_predict
4
5 # Define PLS object
6 pls = PLSRegression(n_components=5)
7
8 # Fit
9 pls.fit(X, Y)
10
11 # Cross-validation
12 y_cv = cross_val_predict(pls, X, y, cv=10)
13
14 # Calculate scores
15 score = r2_score(y, y_cv)
16 mse = mean_squared_error(y, y_cv)
```

As you can see, `sklearn` has already got a PLS package, so we go ahead and use it without reinventing the wheel. So, first we define the number of components we want to keep in our PLS regression. I decided to keep 5 components for the sake of this example, but later will use that as a free parameter to be optimised. Once the PLS object is defined, we fit the regression to the data `X` (the predictor) and `y` (the known response). The third step is to use the model we just built to run a cross-validation experiment using 10 folds cross-validation. When we do not have a large number of spectra, cross-validation is a good way to test the predictive ability of our calibration.

To check how good our calibration is, we measure the usual metrics (see PCR post for more details on it). We'll evaluate these metrics by comparing the result of the cross-validation `y_cv` with the known responses. To optimise the parameters of our PLS regression (for instance pre-processing steps and number of components) we'll just track those metrics, most typically the MSE.

One more thing. In the actual code the various `X`, `y`, etc. are `numpy` arrays read from a spreadsheet. For that you'll probably need to import `numpy` (of course), `pandas` and a bunch of other libraries which we will see below.

This is the basic block of PLS regression in Python. You can take this snippet and use it in your code, provided that you have defined the arrays in the right way. Now it's time for us to take a look at the data import and pre-processing.

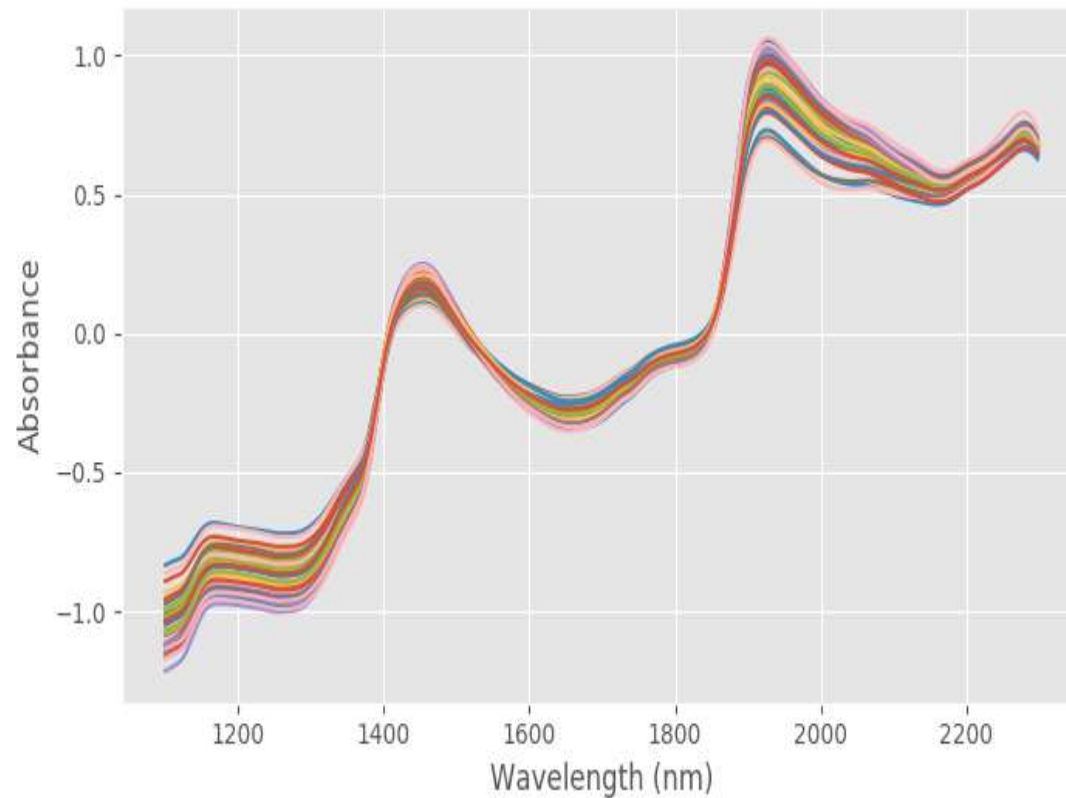
NIR data import and pre-processing

Here's the complete list of imports

```
1 from sys import stdout
2
3 import numpy as np
4 import pandas as pd
5 import matplotlib.pyplot as plt
6
7 from scipy.signal import savgol_filter
8
9 from sklearn.cross_decomposition import PLSRegression
10 from sklearn.model_selection import cross_val_predict
11 from sklearn.metrics import mean_squared_error, r2_score
```

Next let's import the data, which is kept into a csv file. The data is composed of 50 NIR spectra of fresh peaches. Each spectrum has a corresponding brix value (the response) associated with it. Finally, each spectrum is taken over 600 wavelength points, from 1100 nm to 2300 nm in steps of 2 nm. The data is available for download at our [Github repository](#).

```
1 data = pd.read_csv('./data/peach_spectra+brixvalues.csv')
2 # Get reference values
3 y = data['Brix'].values
4 # Get spectra
5 X = data.drop(['Brix'], axis=1).values
6 # Get wavelengths
7 wl = np.arange(1100,2300,2)
```

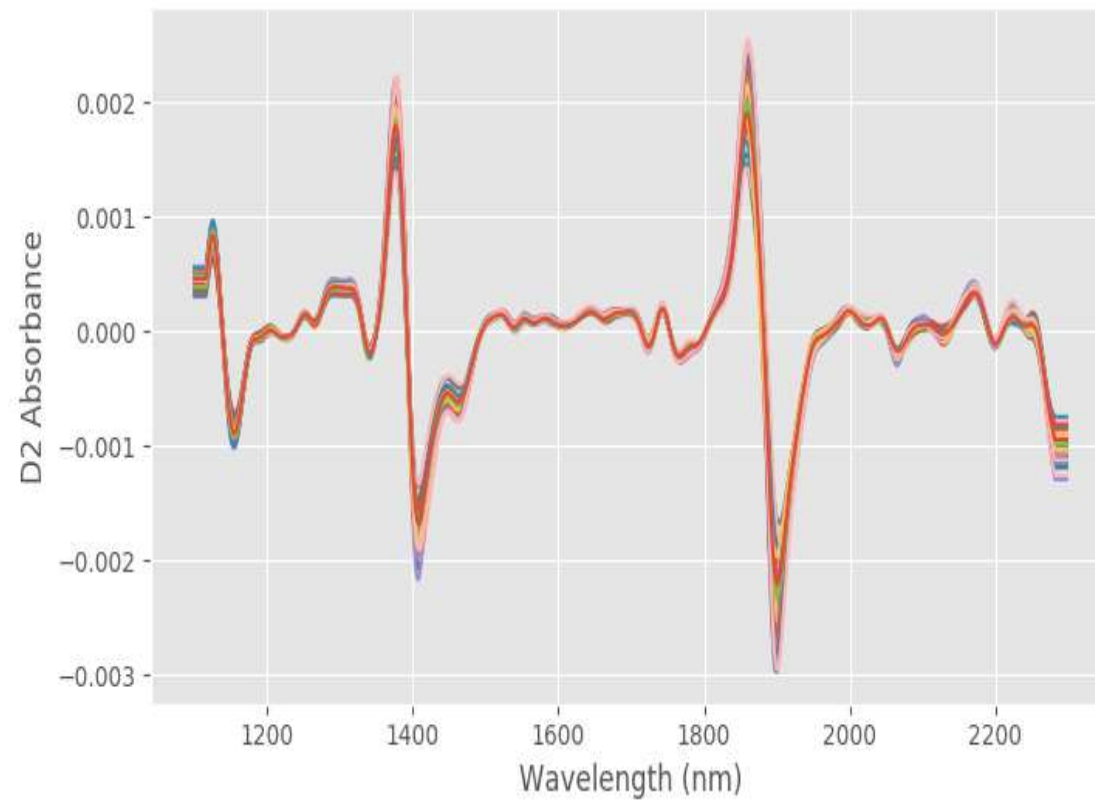


If required, data can be easily sorted by PCA (we've seen some examples of that) and corrected with [multiplicative scatter correction](#), however a simple yet effective way to get rid of baseline and linear variations is to perform a second derivative on the data. Let's do that and check the result.

```
1 # Calculate second derivative
2 X2 = savgol_filter(X, 17, polyorder = 2, deriv=2)
3
4 # Plot second derivative
5 plt.figure(figsize=(8,4.5))
6
```



```
7 with plt.style.context('ggplot'):  
8     plt.plot(wl, X2.T)  
9     plt.xlabel('Wavelength (nm)')  
10    plt.ylabel('D2 Absorbance')  
    plt.show()
```



The offset is gone and the data look more bunched together.

Partial Least Squares Regression

Now it's time to get to the optimisation of the PLS regression. As anticipated above, we want to run a PLS regression with a variable number of components and test its performance in cross-validation. In practice we want to find the number of components that minimises the MSE. Let's write a function for it

```
1 def optimise_pls_cv(X, y, n_comp, plot_components=True):
2
3     '''Run PLS including a variable number of components, up to n_comp,
4     and calculate MSE '''
5
6     mse = []
7     component = np.arange(1, n_comp)
8
9     for i in component:
10         pls = PLSRegression(n_components=i)
11
12         # Cross-validation
13         y_cv = cross_val_predict(pls, X, y, cv=10)
14
15         mse.append(mean_squared_error(y, y_cv))
16
17         comp = 100*(i+1)/n_comp
18         # Trick to update status on the same line
19         stdout.write("\r%d%% completed" % comp)
20         stdout.flush()
21     stdout.write("\n")
22
23     # Calculate and print the position of minimum in MSE
24     msemin = np.argmin(mse)
25     print("Suggested number of components: ", msemin+1)
26     stdout.write("\n")
27
28     if plot_components is True:
29         with plt.style.context('ggplot'):
30             plt.plot(component, np.array(mse), '-v', color='blue', mfc='blue')
31             plt.plot(component[msemin], np.array(mse)[msemin], 'P', ms=10, mfc='r')
32             plt.xlabel('Number of PLS components')
33             plt.ylabel('MSE')
34
```

```

35         plt.title('PLS')
36         plt.xlim(left=-1)
37
38     plt.show()
39
40     # Define PLS object with optimal number of components
41     pls_opt = PLSRegression(n_components=msemin+1)
42
43     # Fit to the entire dataset
44     pls_opt.fit(X, y)
45     y_c = pls_opt.predict(X)
46
47     # Cross-validation
48     y_cv = cross_val_predict(pls_opt, X, y, cv=10)
49
50     # Calculate scores for calibration and cross-validation
51     score_c = r2_score(y, y_c)
52     score_cv = r2_score(y, y_cv)
53
54     # Calculate mean squared error for calibration and cross validation
55     mse_c = mean_squared_error(y, y_c)
56     mse_cv = mean_squared_error(y, y_cv)
57
58     print('R2 calib: %5.3f' % score_c)
59     print('R2 CV: %5.3f' % score_cv)
60     print('MSE calib: %5.3f' % mse_c)
61     print('MSE CV: %5.3f' % mse_cv)
62
63     # Plot regression and figures of merit
64     rangey = max(y) - min(y)
65     rangex = max(y_c) - min(y_c)
66
67     # Fit a line to the CV vs response
68     z = np.polyfit(y, y_c, 1)
69     with plt.style.context(('ggplot')):
70         fig, ax = plt.subplots(figsize=(9, 5))
71         ax.scatter(y_c, y, c='red', edgecolors='k')
72         #Plot the best fit line
73         ax.plot(np.polyval(z,y), y, c='blue', linewidth=1)
74         #Plot the ideal 1:1 line
75         ax.plot(y, y, color='green', linewidth=1)
76         plt.title('$R^2$ (CV): '+str(score_cv))
77         plt.xlabel('Predicted $^{\\circ}$Brix')

```

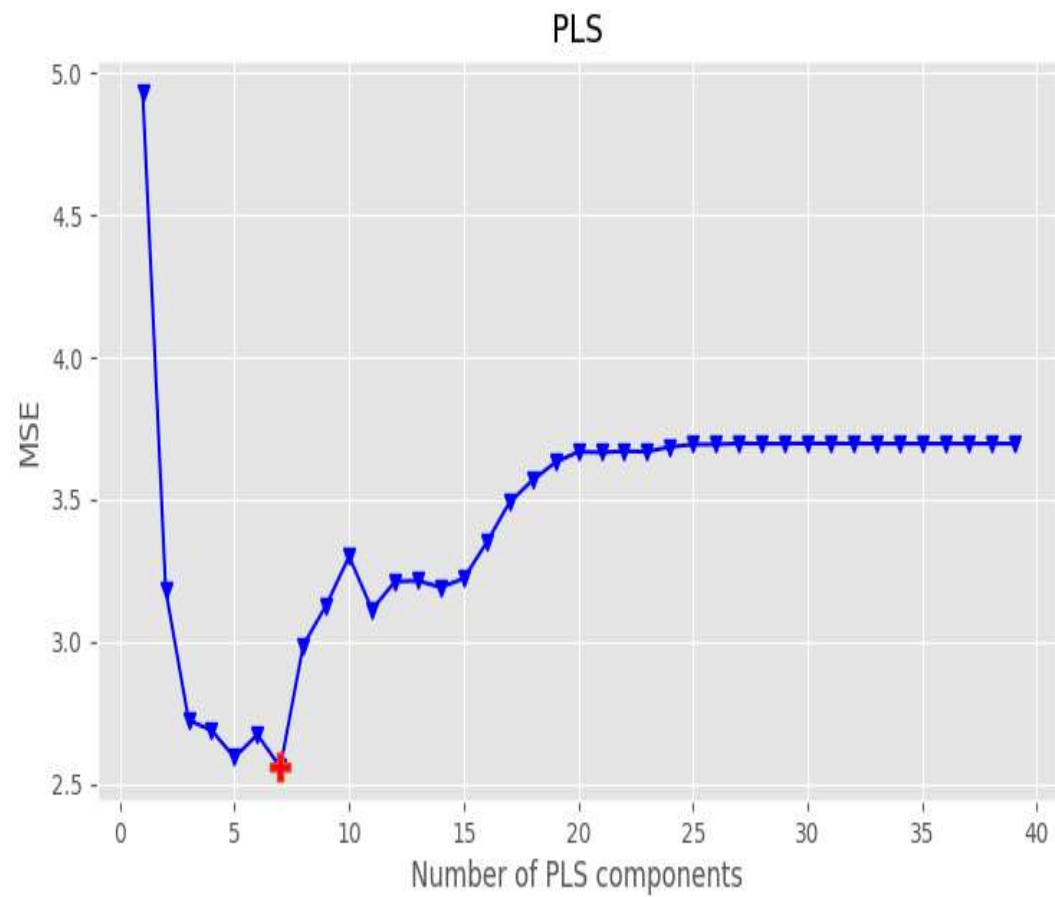
```
78     plt.ylabel('Measured  $\hat{\circ}$ Brix')
79
80     plt.show()
81
    return
```

This function first runs a loop over the number of PLS components and calculates the MSE of prediction. Secondly, it finds the number of components that minimises the MSE and uses that value to run a PLS again. A bunch of metrics is calculated and printed the second time around.

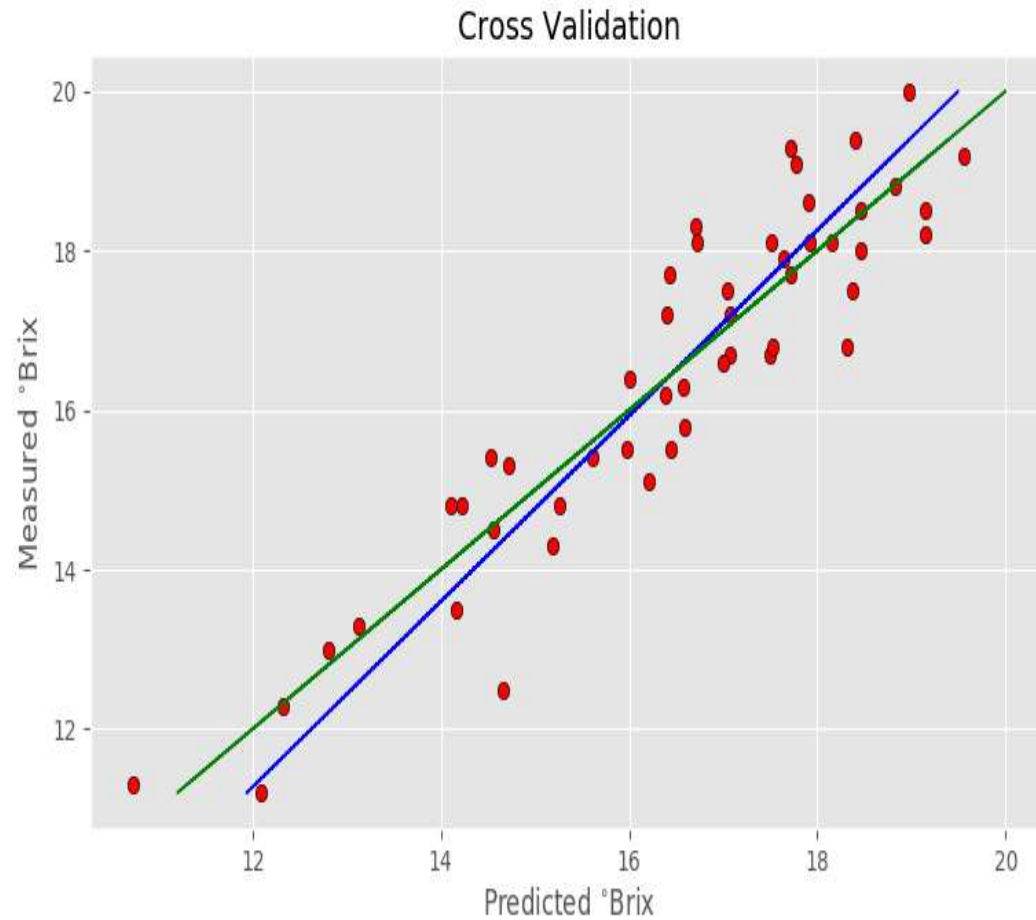
Let's run this function by setting the maximum number of components to 40.

```
1 optimise_pls_cv(X2,y, 40, plot_components=True)
```

The first plot that'll come up is the MSE as a function of the number of components. The suggested number of components that minimises the MSE is highlighted on the plot.



The second plot is the actual regression figure, including the metrics for the prediction.



At the same time, the following information is going to be printed on screen.

```
1 Suggested number of components: 7
2
3 R2 calib: 0.859
4 R2 CV: 0.451
5 MSE calib: 0.657
6 MSE CV: 2.558
```

The model seems to work well on calibration data, but not quite as well on the validation sets. This is a classic example of what in machine learning is called overfitting. In our next post on [variable selection with PLS](#) we shall see how we can improve on this result by preselecting the wavelength bands of our spectra.

Well, we reached the end of this introductory post on PLS regression using Python. I hope you enjoyed reading it and I'll see you next time. Thanks again for reading!



 Tweet

 Share

 Pocket

About The Author

Daniel Pelliccia