A variable selection method for PLS in Python

07/04/2018

Welcome to our new technical tutorial on Python chemometrics; today we will be discussing a variable selection method for PLS in Python.

In other posts we've covered Principal Component Regression (PCR) and the basics of Partial Least Squares (PLS) regression. At the heart of all regression methods is the idea of correlating measured spectra of a

substance to be analysed, with known reference values of the same substance. In this way one can build a calibration model that 'translates' the spectrum into the value of interest. A robust calibration will then enable to obtain the value (or values) of interest straight form the spectra.

- Principal Component Regression
- Partial Least Squares regression
- Outliers detection with PLS

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 nice and simple, 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.

For an introduction to PLS take a look at this post. There we run through a tutorial for a simple PLS optimisation in cross-validation. The quality of the calibration model is evaluated using the coefficient of determination R^2 and the mean squared error (MSE).

The typical cross-validation procedure is to divide the set of data into a few groups, leave one of the group out and fit a PLS model on the remaining groups. The model is then used to predict the values of the left out group. This basic process is repeated so that all samples have been

predicted once. The metrics are then averaged to produce cross-validation scores.

We have seen how a simple model might fail to have sufficient predictive power (the cross-validation metrics are not optimal). In this post we will discuss what variable selection is, and how it can be used to improve the predictive power of the calibration. Hope you'll enjoy it.

Variable selection in chemometrics

The idea behind variable selection in chemometrics is that when it comes to spectral measurements not all wavelengths are created equals. In visible and NIR spectroscopy especially, it is often hard to predict in advance which wavelength bands will contain most of the signal related to the analyte we want to measure. So, as a first shot, we measure the whole range allowed by our instrument, and then figure out later which bands are more relevant for our calibration.

To say the same thing in a bit more quantitative way, we want to check which wavelength bands lead to a better quality model. Actually, in practice, we check which bands give a worse quality model, so we can get rid of them.

This seems logical enough, but I've deliberately left a fundamental piece of the puzzle out. How do we even start breaking our spectrum into

bands and searching for the worst performing ones?

For this example we'll take a simple approach: we'll filter out wavelength bands based on the strength of the related regression coefficients. Let's understand what that means.

Feature selection by filtering

Feature selection by filtering is one of the simplest ways of performing wavelength selection. For an overview of the methods that are currently used, check out this excellent review paper by T. Mehmood *et al.*: A review of variable selection methods in Partial Least Squares Regression.

The idea behind this method is very simple, and can be summarised in the following:

1. Optimise the PLS regression using the full spectrum, for instance using cross-validation or prediction data to quantify its quality.
2. Extract the regression coefficients form the best model. Each regression coefficient uniquely associates each wavelength with the response. A low absolute value of the regression coefficient means that specific wavelength has a low correlation with the quantity of interest.
3. Discard the lowest correlation wavelengths according to some rule. These wavelengths are the ones that typically worsen the quality of the calibration model, therefore by discarding them effectively we

expect to improve the metrics associated with our prediction or cross-validation.

One way to discard the lowest correlation wavelengths is to set a threshold and get rid of all wavelengths whose regression coefficients (in absolute value) fall below that threshold. However this method is very sensitive to the choice of threshold, which tends to be a subjective choice, or requires a trial-and-error approach.

Another approach, that is much less subjective, is to discard one wavelength at a time (the one with the lowest absolute value of the associated regression coefficient) and rebuild the calibration model. By choosing the MSE (means square error) of prediction or cross-validation as metric, the procedure is iterated until the MSE decreases. At some point, removing wavelengths will produce a worse calibration, and that is the stopping criterion for the optimisation algorithm.

Alternatively, one could simply remove a fixed number of wavelengths iteratively, and then check for which number of removed wavelengths the MSE is minimised. Either way we have a method that does not depend on the subjective choice of the threshold and can be applied without changes regardless of the data set being analysed.

One caveat of this method, at least in its simple implementation is that it may get a bit slow for large datasets.

Extracting PLS regression coefficients: Python code

We start, as usual, with the list of import you'll need to run this example

```
from sys import stdout

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

from scipy.signal import savgol_filter

from sklearn.cross_decomposition import PLSRegression
from sklearn.model_selection import cross_val_predict
from sklearn.metrics import mean_squared_error, r2_score
```

Next, we import the data, calibration and calculate the derivatives. The data is available for download at our Github repository.

```
# Read data
data = pd.read_csv('./data/peach_spectra+brixvalues.csv')
X = data.values[:,1:]
y = data['Brix']

# Define wavelength range
wl = np.arange(1100,2300,2)

# Calculate derivatives
X1 = savgol_filter(X, 11, polyorder = 2, deriv=1)
X2 = savgol_filter(X, 13, polyorder = 2,deriv=2)
```

Our aim for this tutorial is to define a function that will simultaneously optimise the number of PLS components and the sequence of

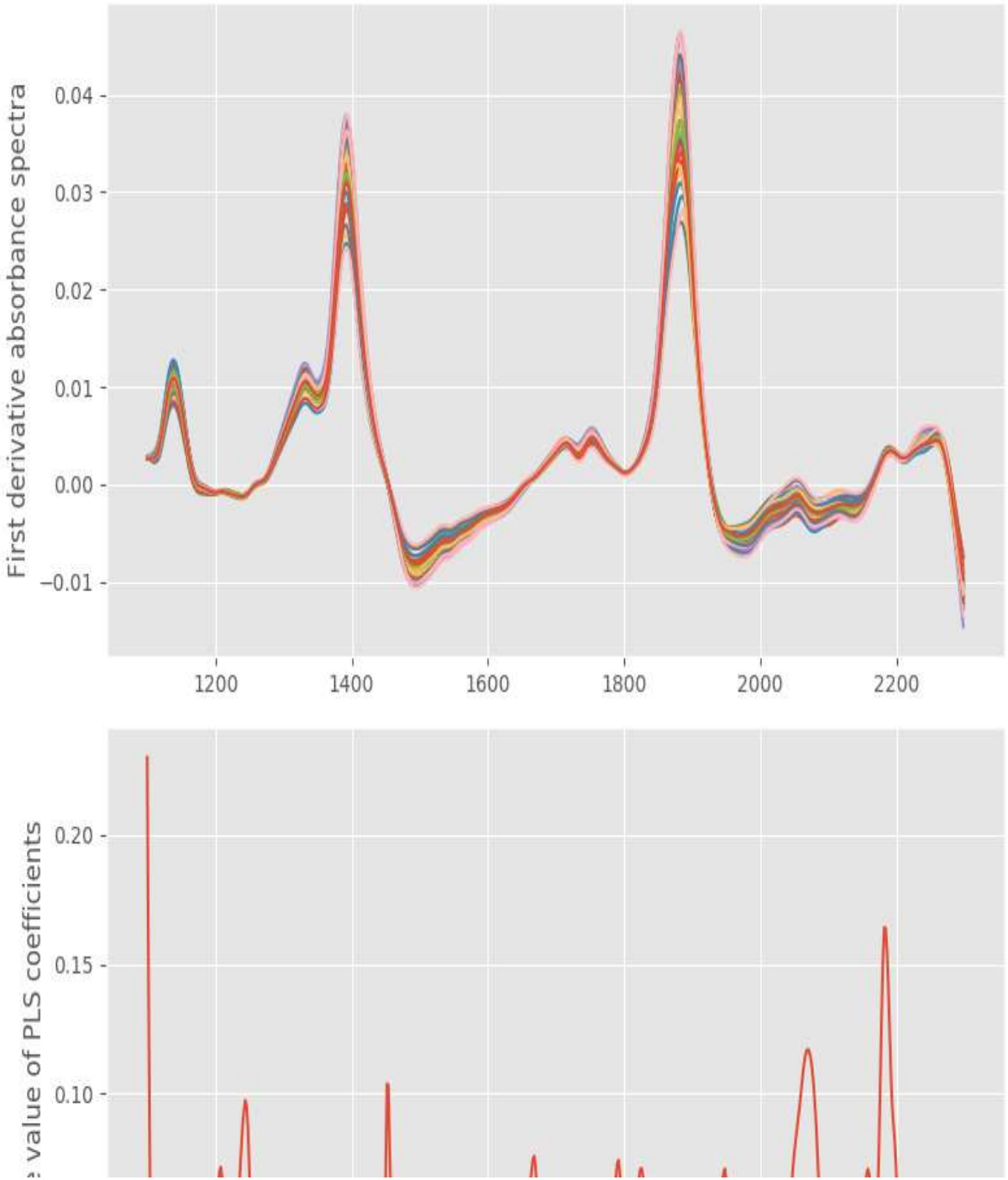
wavelengths to minimise the MSE in cross-validation (take a look at our previous post on PLS regression if you want to brush up on the basic PLS code and the terminology). Before doing that however, let's look at the basic feature-selection idea: the absolute value of the PLS coefficients.

```
# Define the PLS regression object
pls = PLSRegression(n_components=8)
# Fit data
pls.fit(X1, y)

# Plot spectra
plt.figure(figsize=(8,9))
with plt.style.context(('ggplot')):
    ax1 = plt.subplot(211)
    plt.plot(wl, X1.T)
    plt.ylabel('First derivative absorbance spectra')

    ax2 = plt.subplot(212, sharex=ax1)
    plt.plot(wl, np.abs(pls.coef_[:,0]))
    plt.xlabel('Wavelength (nm)')
    plt.ylabel('Absolute value of PLS coefficients')

    plt.show()
```

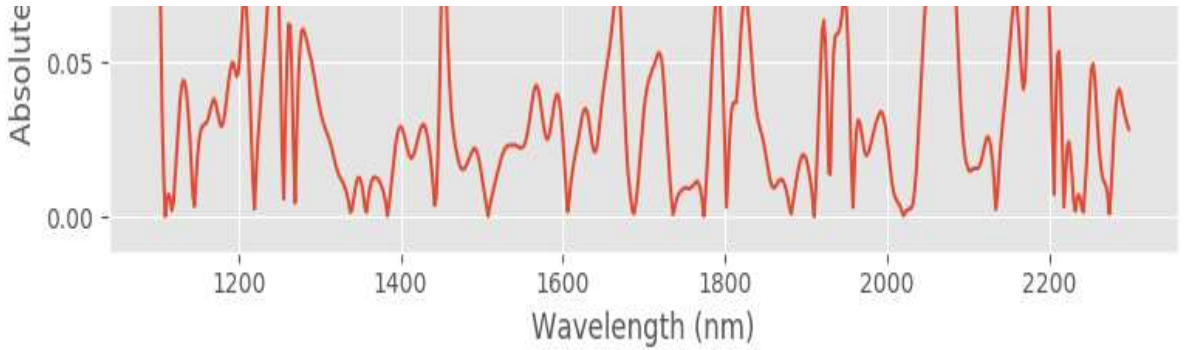
What we have done is a basic PLS regression with 8 components and used it to fit the first derivative data. The PLS coefficients of this fit can be accessed by `pls.coef_[:,0]`, that is the first (and in this case only) column of the `pls.coef_` object. It is worth reiterating that **these are the regression coefficients that quantify the strength of the association between each wavelength and the response**. We are interested only in the absolute value of these coefficients, as large positive and large negative coefficients are equally important for our regression, denoting

large positive or large negative correlation with the response respectively. In other words we want to identify and discard the regression coefficients that are close to zero: they are the ones with poor correlation with the response.

The association between each coefficient and the spectral feature can be visualised using the the second part of the code above, producing this plot.


First derivative absorbance spectra
0.04
0.03
0.02
0.01
0.00
−0.01
1200
1400
1600
1800
2000
2200

value of PLS coefficients
0.20
0.15
0.10



As you can see, a number of wavelengths are associated with fairly small regression coefficients. We want to use this basic idea to identify and discard small coefficients iteratively, starting from the smallest, and optimise the PLS regression at the same time.

NIR variable selection for PLS regression

We are almost ready to write down the code for the complete function we want to use. One last thing to point out is the trick we'll use to discard small PLS regression coefficients.

```
# Get the list of indices that sorts the PLS coefficients in ascending order
# of the absolute value
sorted_ind = np.argsort(np.abs(pls.coef_[:,0]))

# Sort spectra according to ascending absolute value of PLS coefficients
Xc = X1[:,sorted_ind]
```

What we have done is to extract the sequence of indices corresponding to sorting the absolute value of the PLS regression coefficients in ascending order, and use that sequence to sort the spectra accordingly. Note that sorting the spectral component according to the strength of the associated PLS coefficients has no influence whatsoever on the PLS regression, but it enables us to easily discard one component at a time.

OK, we are now ready to reveal the entire optimisation function.

```
def pls_variable_selection(X, y, max_comp):

    # Define MSE array to be populated
    mse = np.zeros((max_comp,X.shape[1]))

    # Loop over the number of PLS components
    for i in range(max_comp):

        # Regression with specified number of components, using full spectrum
        pls1 = PLSRegression(n_components=i+1)
        pls1.fit(X, y)

        # Indices of sort spectra according to ascending absolute value of PLS co
        sorted_ind = np.argsort(np.abs(pls1.coef_[:,0]))

        # Sort spectra accordingly
        Xc = X[:,sorted_ind]

        # Discard one wavelength at a time of the sorted spectra,
        # regress, and calculate the MSE cross-validation
        for j in range(Xc.shape[1]-(i+1)):

            pls2 = PLSRegression(n_components=i+1)

```

```
            pls2.fit(Xc[:, j:], y)

            y_cv = cross_val_predict(pls2, Xc[:, j:], y, cv=5)

            mse[i,j] = mean_squared_error(y, y_cv)

        comp = 100*(i+1)/(max_comp)
        stdout.write("\r%d%% completed" % comp)
        stdout.flush()
    stdout.write("\n")

    # # Calculate and print the position of minimum in MSE
    mseminx,mseminy = np.where(mse==np.min(mse[np.nonzero(mse)]))

    print("Optimised number of PLS components: ", mseminx[0]+1)
    print("Wavelengths to be discarded ",mseminy[0])
    print('Optimised MSEP ', mse[mseminx,mseminy][0])
    stdout.write("\n")
    # plt.imshow(mse, interpolation=None)
    # plt.show()


    # Calculate PLS with optimal components and export values
    pls = PLSRegression(n_components=mseminx[0]+1)
    pls.fit(X, y)

    sorted_ind = np.argsort(np.abs(pls.coef_[:,0]))

    Xc = X[:,sorted_ind]

    return(Xc[:,mseminy[0]:],mseminx[0]+1,mseminy[0], sorted_ind)
```

This function works by running a PLS regression with a given number of components (up to a specified maximum), then filtering out one regression coefficient at a time up to the maximum number allowed. All data are stored in the 2D array `mse`. At the end of the double loop we search for the global minimum of mse excluding the zeros. That will give

us the number of components and the wavelength bands that corresponds to the optimal cross-validation MSE.

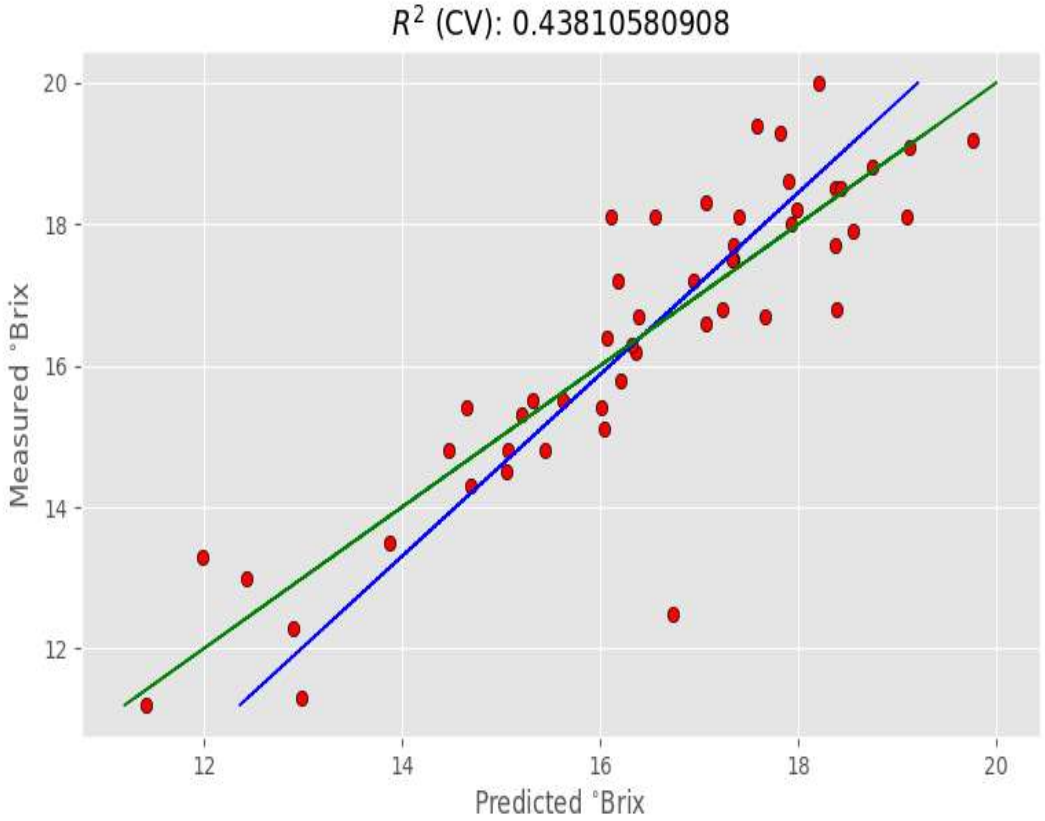
To give you an idea of the huge improvement we can get with this approach, let's compare the result obtained using the full spectrum with the one optimised by variable selection. We start by defining a function to perform simple PLS with a fixed number of components, and calculate the metrics in cross-validation.

```
def simple_pls_cv(X, y, n_comp):

    # Run PLS with suggested number of components
    pls = PLSRegression(n_components=n_comp)
    pls.fit(X, y)
    y_c = pls.predict(X)

    # Cross-validation
    y_cv = cross_val_predict(pls, X, y, cv=10)

    # Calculate scores for calibration and cross-validation
    score_c = r2_score(y, y_c)
    score_cv = r2_score(y, y_cv)

    # Calculate mean square error for calibration and cross validation
    mse_c = mean_squared_error(y, y_c)
    mse_cv = mean_squared_error(y, y_cv)

    print('R2 calib: %5.3f'  % score_c)
    print('R2 CV: %5.3f'  % score_cv)
    print('MSE calib: %5.3f' % mse_c)
    print('MSE CV: %5.3f' % mse_cv)

    # Plot regression

    z = np.polyfit(y, y_cv, 1)
    with plt.style.context(('ggplot')):
        fig, ax = plt.subplots(figsize=(9, 5))
        ax.scatter(y_cv, y, c='red', edgecolors='k')
        ax.plot(z[1]+z[0]*y, y, c='blue', linewidth=1)
        ax.plot(y, y, color='green', linewidth=1)
        plt.title('$R^{2}$ (CV): '+str(score_cv))
```

```
        plt.xlabel('Predicted $^{\circ}$Brix')
        plt.ylabel('Measured $^{\circ}$Brix')

        plt.show()
```

This is the output for 8 components

```
R2 calib: 0.778
R2 CV: 0.438
MSE calib: 1.035
MSE CV: 2.618
```



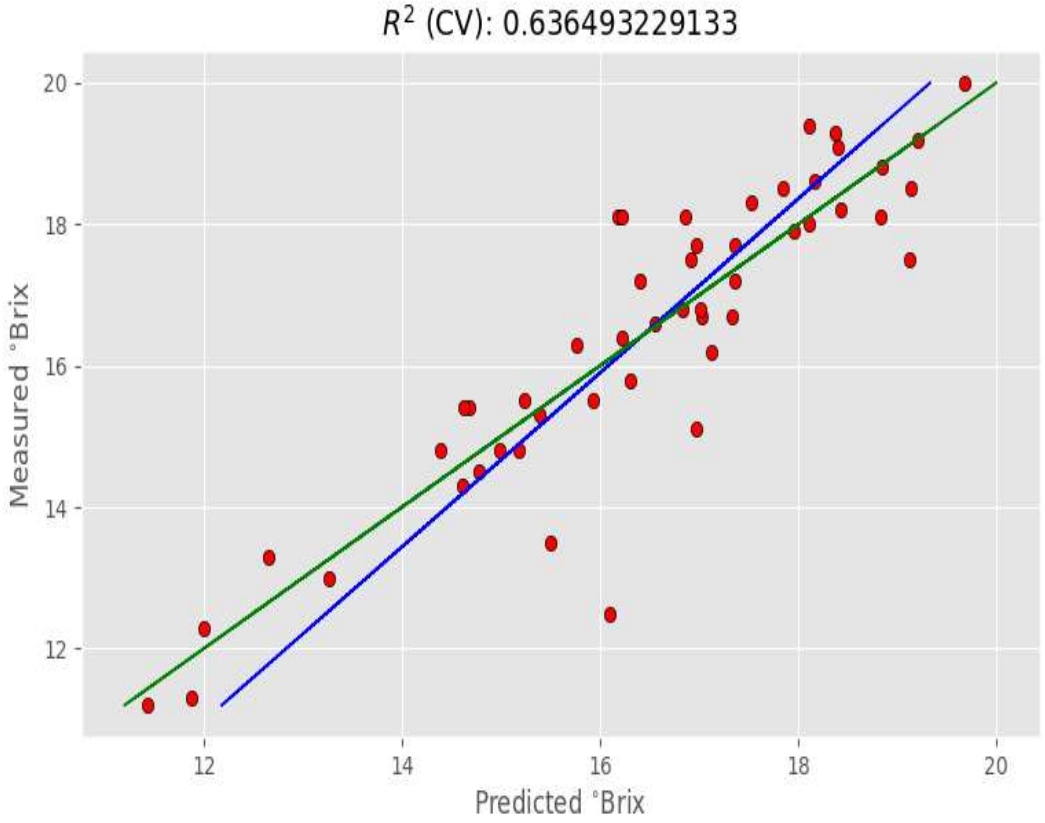
Now we run the optimisation process

```
opt_Xc, ncomp, wav, sorted_ind = pls_variable_selection(X1, y, 15)
simple_pls_cv(opt_Xc, y, ncomp)
```

And this is the results

```
Optimised number of PLS components: 9
Wavelengths to be discarded 452
Optimised MSE 1.74293961079

R2 calib: 0.813

```

```
R2 CV: 0.636
MSE calib: 0.873
MSE CV: 1.694
```

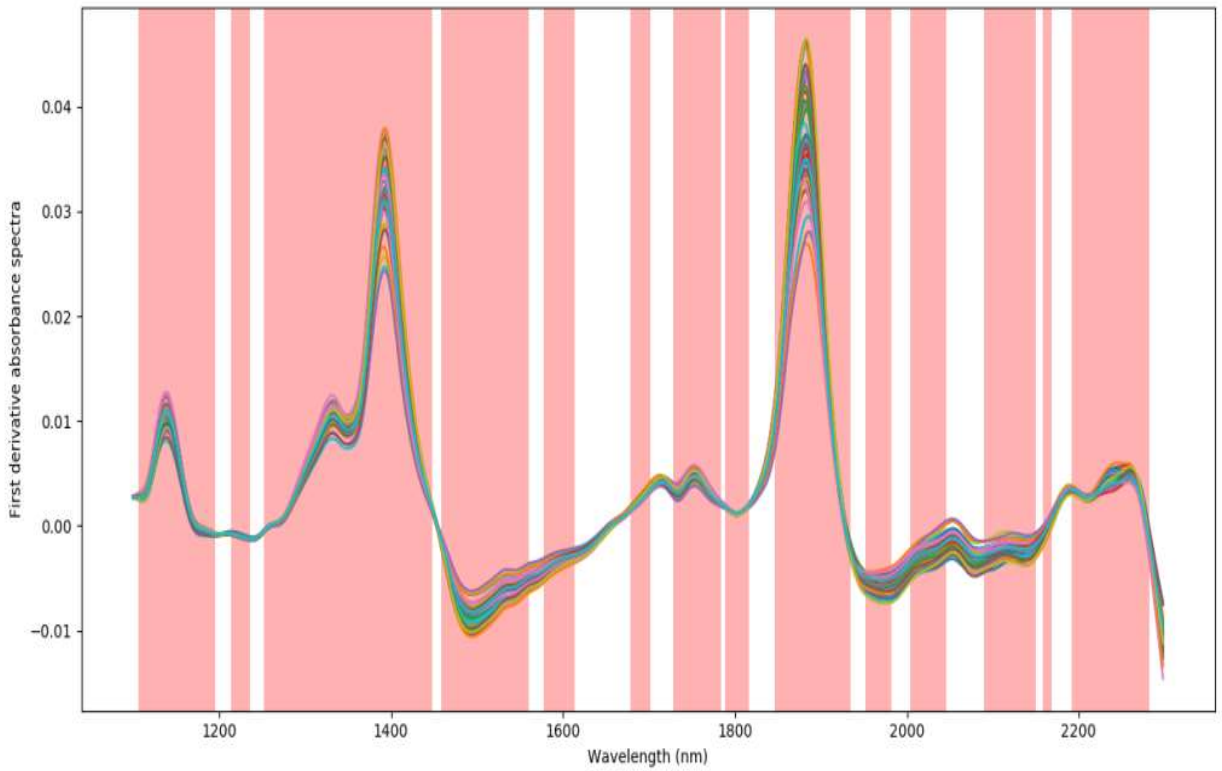


The R^2 score jumped from 0.44 to 0.64, while the MSE cross-validation decreased from 2.6 to 1.7! That amounted to discarding 452 wavelengths out of 600, that is finding the truly significant bands for Brix calibration.

As a sanity check, we can plot the discarded bands on top of the absorbance spectra. The code is as follows.

```
# Get a boolean array according to the indices that are being discarded
ix = np.in1d(wl.ravel(), wl[sorted_ind][:wav])

import matplotlib.collections as collections

# Plot spectra with superimpose selected bands
fig, ax = plt.subplots(figsize=(8,9))
with plt.style.context(('ggplot')):
    ax.plot(wl, X1.T)
    plt.ylabel('First derivative absorbance spectra')
    plt.xlabel('Wavelength (nm)')

collection = collections.BrokenBarHCollection.span_where(
    wl, ymin=-1, ymax=1, where=ix == True, facecolor='red', alpha=0.3)
ax.add_collection(collection)

plt.show()
```

This is the plot of the first derivative spectra with overlapped the discarded bands in red. Note how the main water peaks are actually discarded.



Thanks for reading this tutorial. I hope you enjoyed it and until next time!