Analysis of sugar content in apples through near infrared spectroscopy

Yunqi WEI

March 26, 2019

In this notebook, we will show how to predict the sugar content for apples through the reflection in near infarred spectroscopy (NIR).

In [1]:
```
%config InlineBackend.figure_format ='retina'
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import StratifiedKFold
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import GradientBoostingClassifier
rng = np.random.RandomState(123)

from sys import stdout
from scipy.signal import savgol_filter
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
from sklearn.cross_decomposition import PLSRegression
from sklearn.model_selection import cross_val_predict
from sklearn import linear_model
from sklearn import model_selection
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.model_selection import GridSearchCV
plt.style.use('ggplot')
```

In [2]:
```
def plot_spectrum(x, y, xlabel='wavelength (nm)', ylabel='spectra'):
    ''' Plot the spectra '''
    plt.figure(figsize=(8,6))
    plt.plot(x, np.transpose(y), alpha=0.6)
    plt.xlabel(xlabel, fontsize=14)
    plt.ylabel(ylabel, fontsize=14)
    plt.xticks(fontsize=14)
```

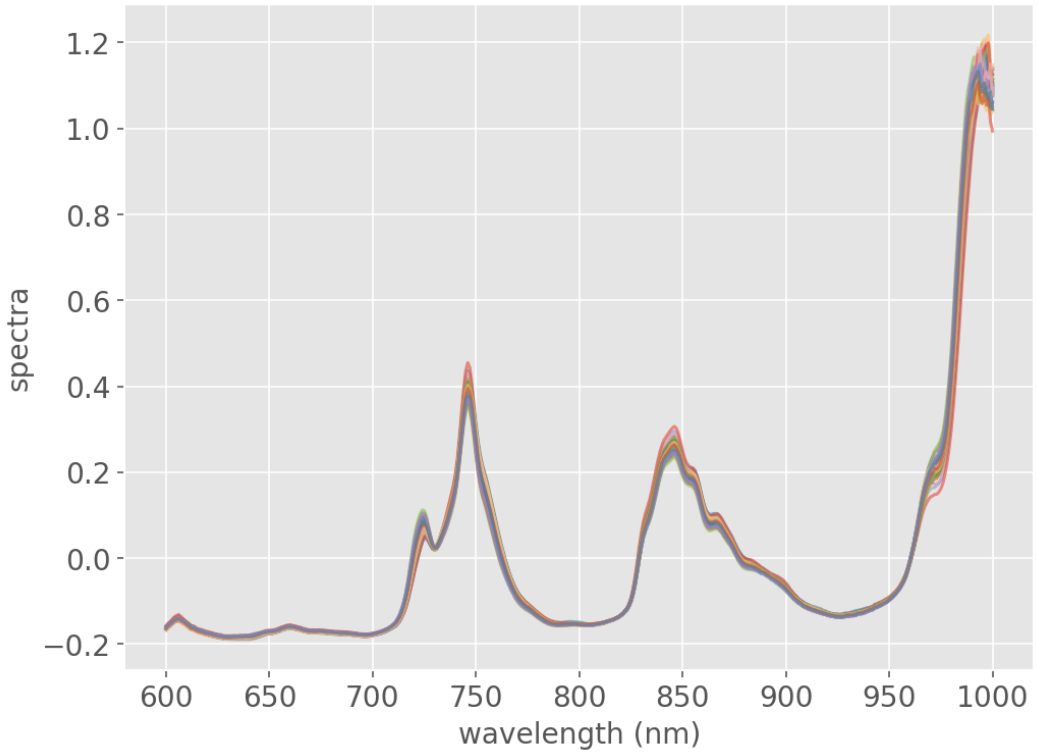
```
        plt.yticks(fontsize=14)
        plt.show()
```

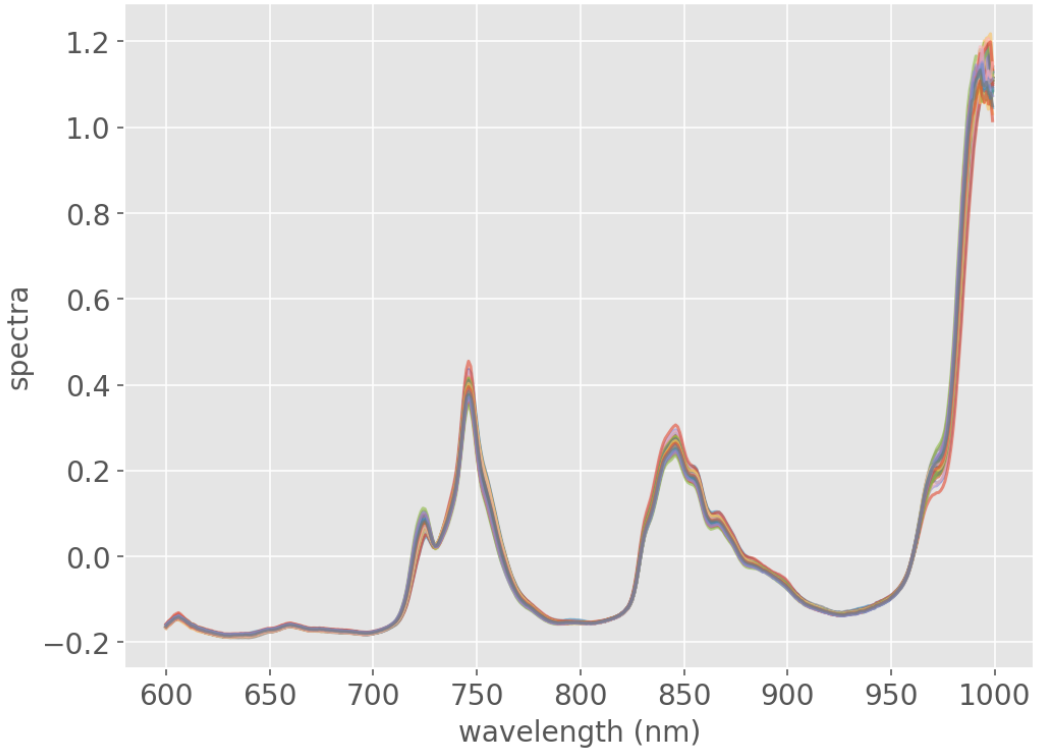
1 Read data

We first read the spectra from the csv files

```
In [3]: # data = pd.read_csv("apple_x.csv", header=None).values
        # data = np.transpose(data)
        # wavelength = data[0,:]
        # data = data[1:]
        # label = pd.read_csv("apple_y.csv", header=None).values
        # label = label[:,0]

        NIR = np.array(pd.read_table('https://raw.githubusercontent.com/khliland/hoggormExamples
        octane = np.array(pd.read_table('https://raw.githubusercontent.com/khliland/hoggormExamp
        data = NIR
        label = octane[:,0]
        sample_count, spectrum_count = data.shape
        print('sample_count = ', sample_count)
        print('spectrum_count =', spectrum_count)
        wavelength = np.linspace(600, 1000, num=spectrum_count)

        # Centering the raw data
        mean_col = np.mean(data, axis=1, keepdims=True)
        std_col = np.std(data, axis=1, keepdims=True)
        data = data - mean_col
        plot_spectrum(wavelength, data)

        ## Truncate the spectra within the certern range
        truncation_start = 0
        truncation_end = -1
        wavelength_truncate = wavelength[truncation_start:truncation_end]
        data_truncate = data[:,truncation_start:truncation_end]
        plot_spectrum(wavelength_truncate, data_truncate)
```

```
sample_count =  60
spectrum_count = 401
```


spectra
1.2
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
600
650
700
750
800
850
900
950
1000
wavelength (nm)


spectra
1.2
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
600
650
700
750
800
850
900
950
1000
wavelength (nm)

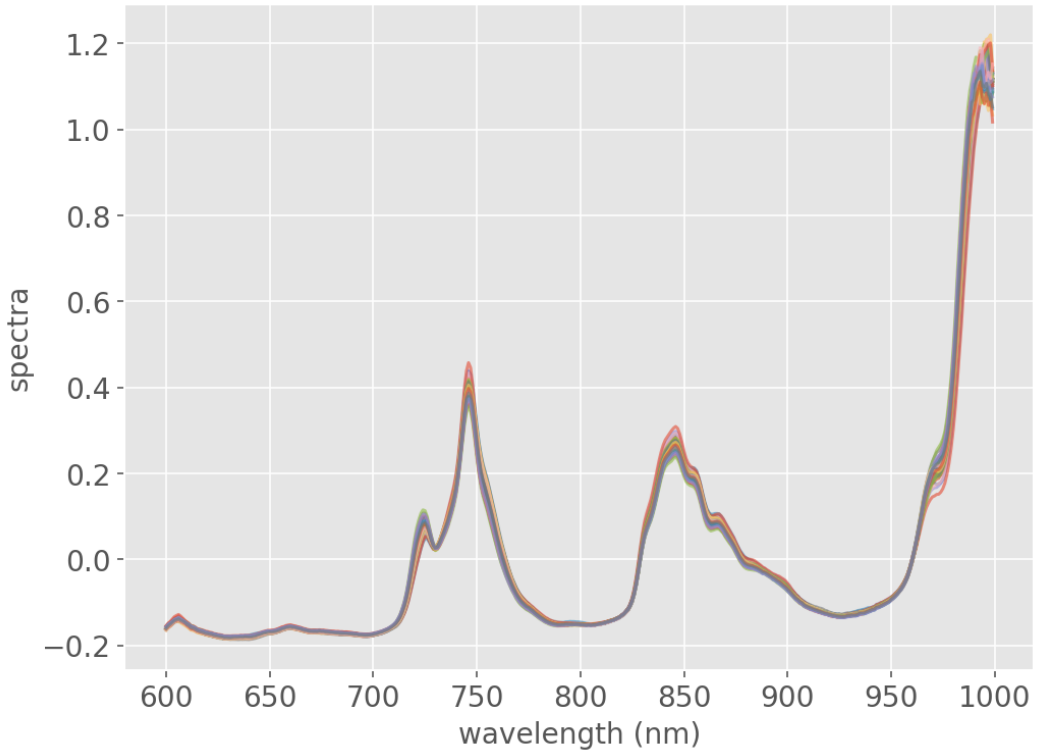
1.1 Preprocessing with the multiplicative scatter correction (MSC)

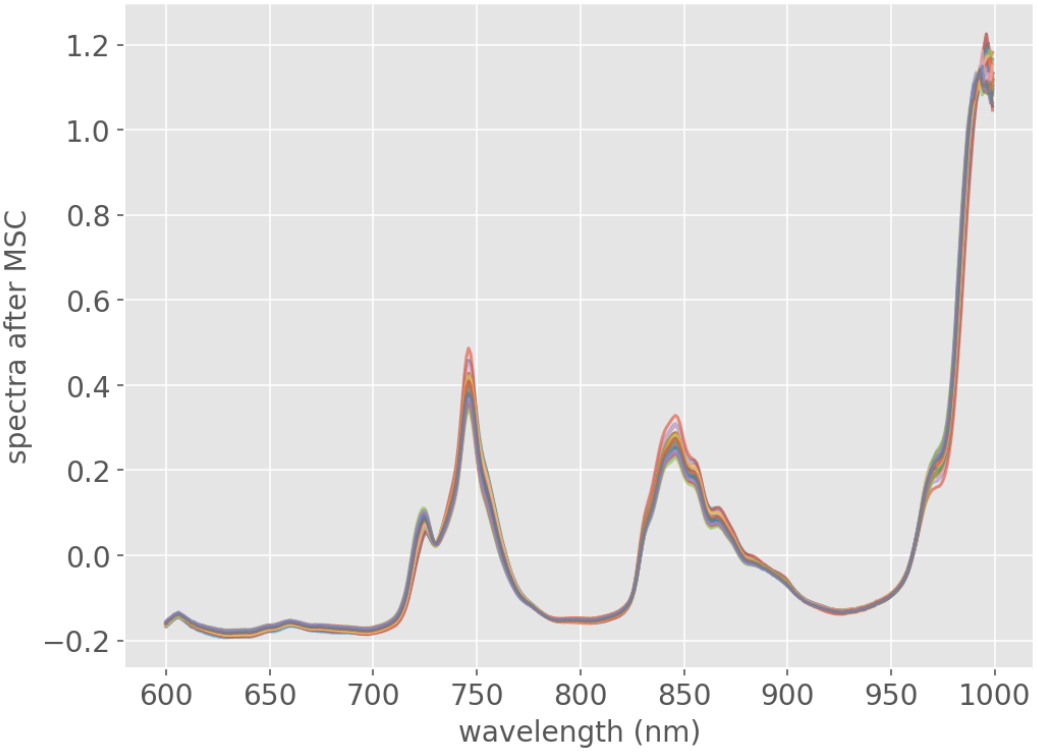
We can model the noises with the following addtive multiplicative model: $X_i \approx a_i + b_i X_{mean}$, where X_{mean} is the average spectrum which we assume to be free of scattering effect. Then we can obtain the corrected spectrum as $X_{msc} = (X_i - a_i)/b_i$.

After the MSC, we see the spectra are better aligned.

```
In [4]: def msc(input_data, reference=None):
            ''' Perform Multiplicative scatter correction'''

            # mean centre correction
            for i in range(input_data.shape[0]):
                input_data[i,:] -= input_data[i,:].mean()

            # Get the reference spectrum. If not given, estimate it from the mean
            if reference is None:
                # Calculate mean
                ref = np.mean(input_data, axis=0)
            else:
                ref = reference

            # Define a new array and populate it with the corrected data
            data_msc = np.zeros_like(input_data)
            for i in range(input_data.shape[0]):
                # Run regression
                fit = np.polyfit(ref, input_data[i,:], 1, full=True)
                # Apply correction
                data_msc[i,:] = (input_data[i,:] - fit[0][1]) / fit[0][0]

            return (data_msc, ref)
```

```
In [5]: data_truncate_msc = msc(data_truncate)[0]
        plot_spectrum(wavelength_truncate, data_truncate)
        plot_spectrum(wavelength_truncate, data_truncate_msc, ylabel='spectra after MSC')
```


spectra
1.2
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
600
650
700
750
800
850
900
950
1000
wavelength (nm)


spectra after MSC
1.2
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
600
650
700
750
800
850
900
950
1000
wavelength (nm)

1.2 Split the dataset into training and validation set

```
In [6]: train_count = 40
        valid_count = 20
        x_train = data_truncate[:train_count, :]
        x_valid = data_truncate[train_count:, :]
        y_train = label[:train_count]
        y_valid = label[train_count:]
```

2 PLS

We perform the multivariate analysis with the partial least square regression. The following equation is used for the prediction expression for y (sugar content) in terms of the explanatory varialbes x_p. $\hat{y} = \beta_0 + \beta_1 x_1 + ... + \beta_p x_p = Xc$.

The prediction error is given as: $e = y - \hat{y}$

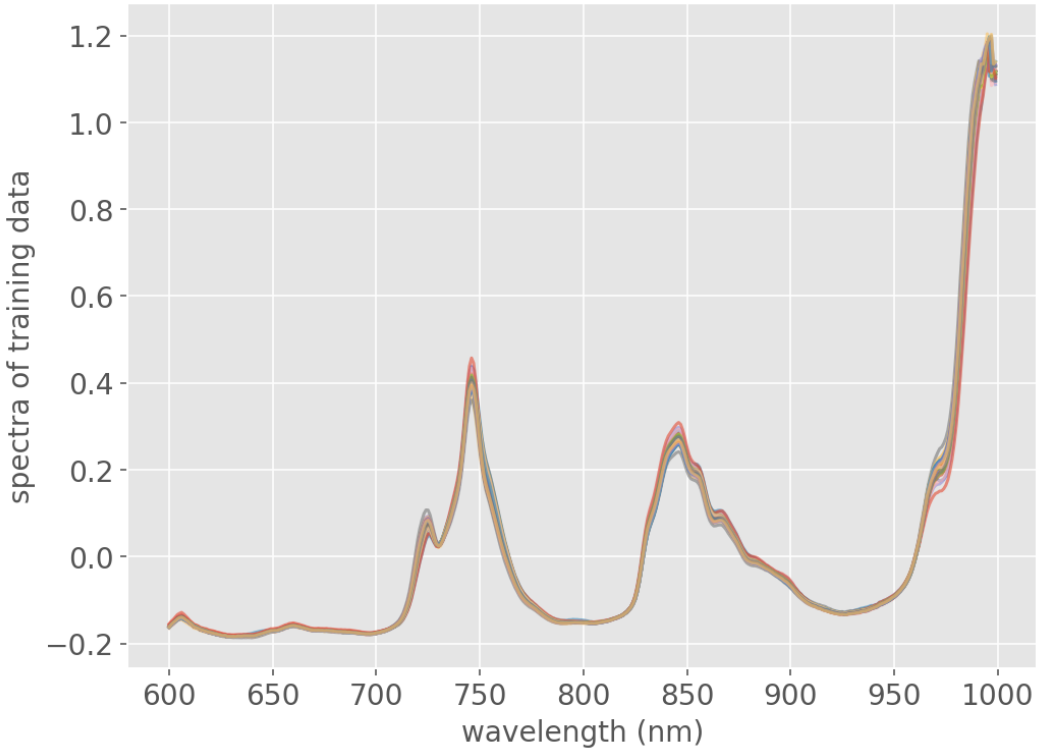
We hope to minimize the squared error: $Q = ||e||^2$ with respect to c, and the mimiization can be done by solving the partial least square equation.

Here, y is the sugar content, x is the reflection spectrum, and p is the number of wavelengths. When two wavelengths are chosen, the sugar calibration equation become: $y = \beta_0 + \beta_1 x_1 + \beta_2 x_2$

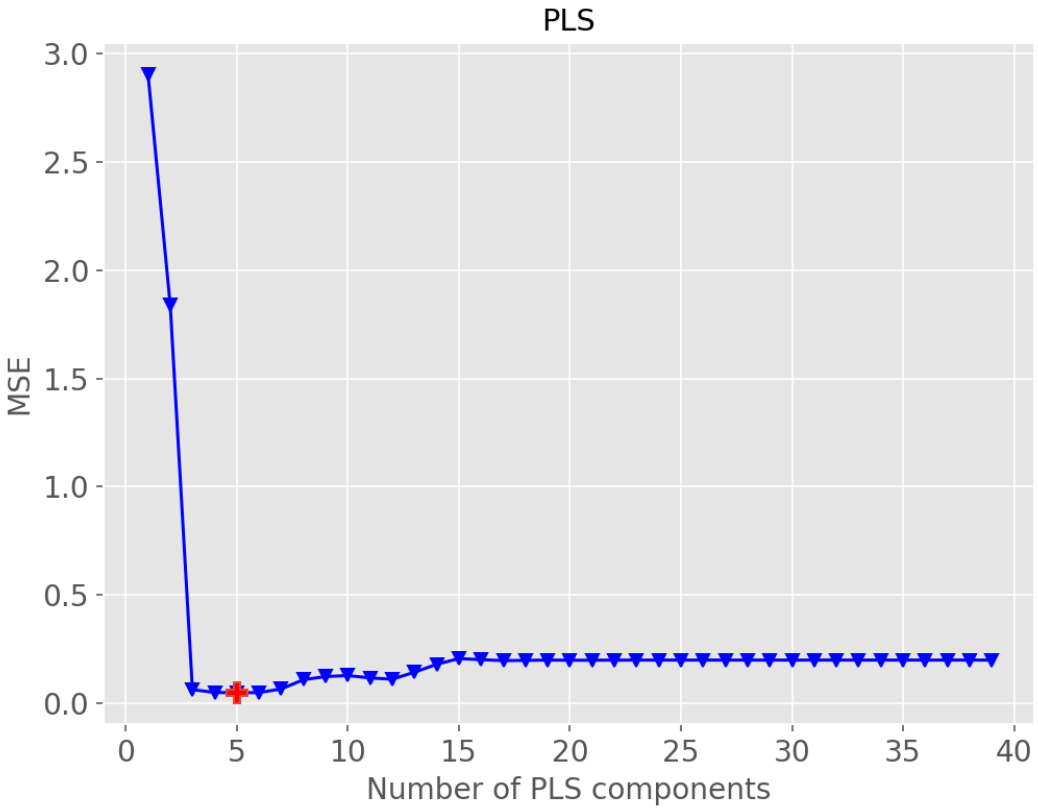
```
In [7]: def pls(X_calib, Y_calib, X_valid, Y_valid, specify_component=0, plot_components=False):
            ''' Performe the partial least square regression '''
            if specify_component == 0:
                #Run PLS including a variable number of components, up to 40,  and calculate MSE
                mse = []
                component = np.arange(1, 40)
                for i in component:
                    pls = PLSRegression(n_components=i)
                    # Fit
                    pls.fit(X_calib, Y_calib)
                    # Prediction
                    Y_pred = pls.predict(X_valid)

                    mse_p = mean_squared_error(Y_valid, Y_pred)
                    mse.append(mse_p)

                stdout.write("\n")

                # Calculate and print the position of minimum in MSE
                msemin = np.argmin(mse)
                print("Suggested number of components: ", msemin+1)
                stdout.write("\n")

                if plot_components is True:
                    with plt.style.context(('ggplot')):
```

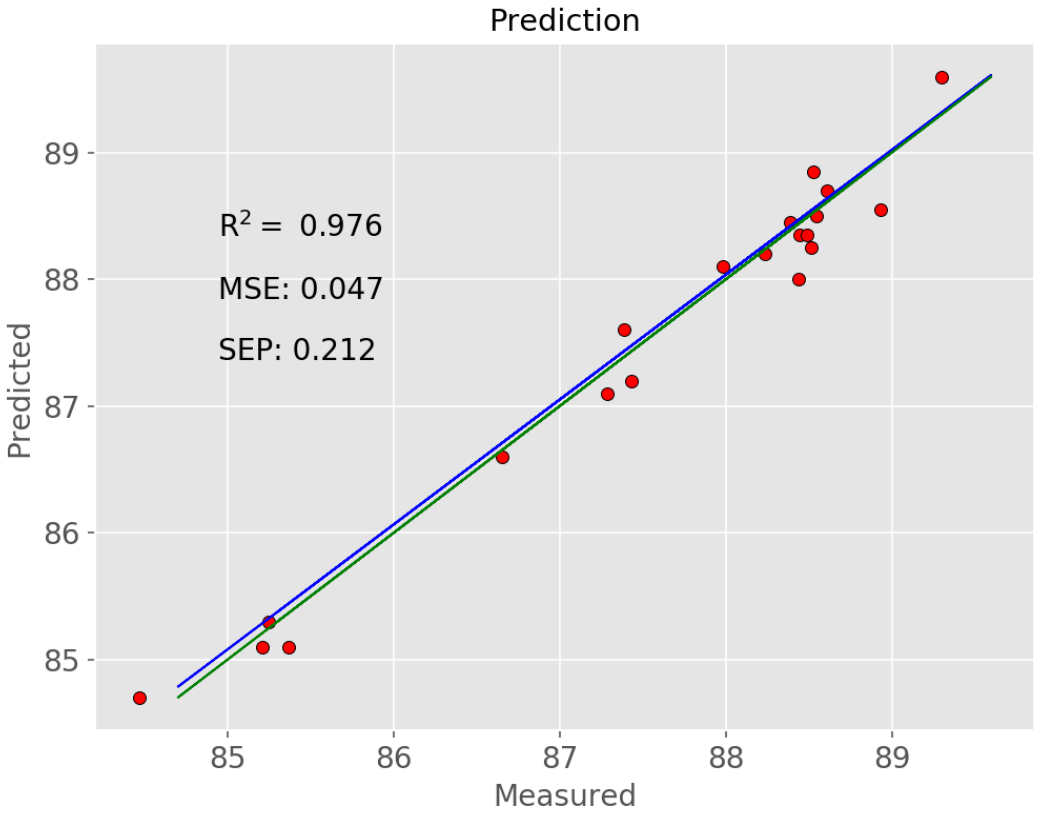
```
            plt.figure(figsize=(8,6))
            plt.plot(component, np.array(mse), '-v', color = 'blue', mfc='blue')
            plt.plot(component[msemin], np.array(mse)[msemin], 'P', ms=10, mfc='red'
            plt.xlabel('Number of PLS components', fontsize=14)
            plt.ylabel('MSE', fontsize=14)
            plt.xticks(fontsize=14)
            plt.yticks(fontsize=14)
            plt.title('PLS')
            plt.xlim(xmin=-1)

        plt.show()
    pls = PLSRegression(n_components = msemin+1)

else:
    pls = PLSRegression(n_components = specify_component)

# fit the pls and do the prediction
pls.fit(X_calib, Y_calib)
Y_pred = pls.predict(X_valid)

# Calculate and print scores
score_p = r2_score(Y_valid, Y_pred) # correlation coefficient
mse_p = mean_squared_error(Y_valid, Y_pred) # mean square error
sep = np.std(Y_pred[:,0]-Y_valid) # standard error of prediction
bias = np.mean(Y_pred[:,0]-Y_valid) # bias

# Plot regression and figures of merit
rangex = max(Y_valid) - min(Y_valid)
rangey = max(Y_pred) - min(Y_pred)

z = np.polyfit(Y_valid, Y_pred, 1)
with plt.style.context(('ggplot')):
    fig, ax = plt.subplots(figsize=(8, 6))
    ax.scatter(Y_pred, Y_valid, color='red', edgecolors='k')
    ax.plot(Y_valid, z[1]+z[0]*Y_valid, color='blue', linewidth=1)
    ax.plot(Y_valid, Y_valid, color='green', linewidth=1)
    plt.xlabel('Measured', fontsize=14)
    plt.ylabel('Predicted', fontsize=14)
    plt.xticks(fontsize=14)
    plt.yticks(fontsize=14)
    plt.title('Prediction')

    # Print the scores on the plot
    plt.text(min(Y_valid)+0.05*rangex, min(Y_pred)+0.8*rangey, 'R$^{2}=$ %5.3f'  % s
    plt.text(min(Y_valid)+0.05*rangex, min(Y_pred)+0.7*rangey, 'MSE: %5.3f' % mse_p,
    plt.text(min(Y_valid)+0.05*rangex, min(Y_pred)+0.6*rangey, 'SEP: %5.3f' % sep, f
    plt.show()
```

In [8]:
```
plot_spectrum(wavelength_truncate, x_train, ylabel='spectra of training data')
pls(x_train, y_train, x_valid, y_valid, plot_components=True)
```




```
C:\Users\Bo\Anaconda3\lib\site-packages\sklearn\cross_decomposition\pls_.py:287: UserWarning: Y
  warnings.warn('Y residual constant at iteration %s' % k)


Suggested number of components:  5
```


PLS
MSE
Number of PLS components
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0
5
10
15
20
25
30
35
40



3 Principle componnet regression (PCR)

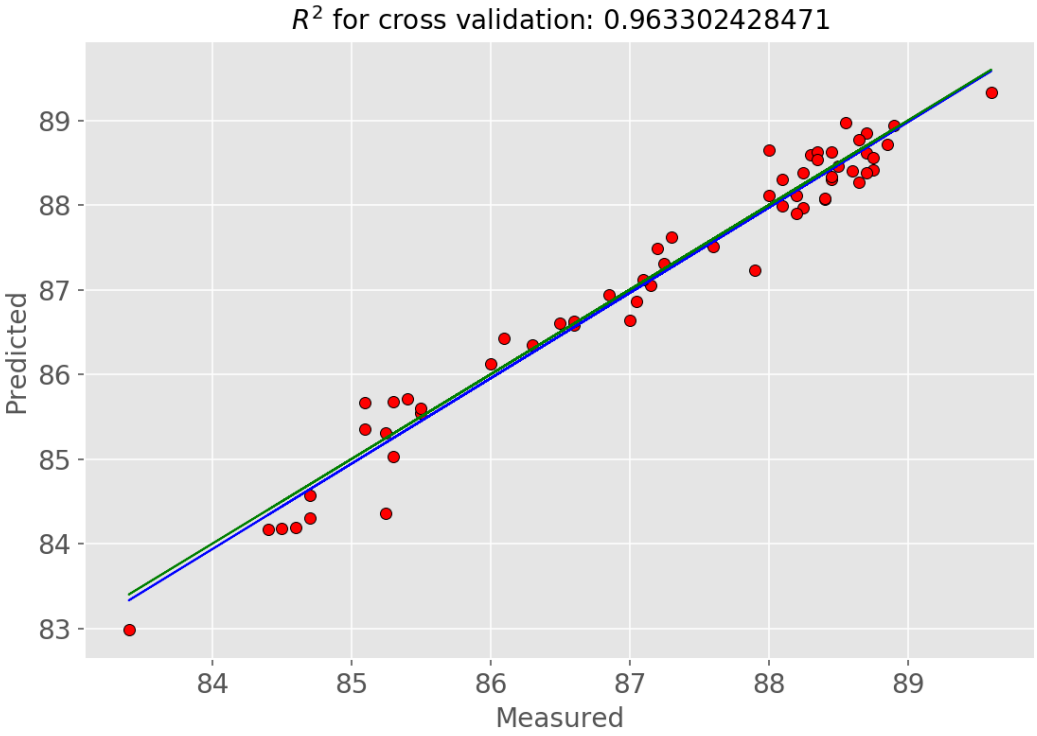
Another method is to first reduce the dimension of the spectrum through pincipal component analysis, and then perform the linear regression according to the compressed wavelengths. Since the spectrum data is compressed without knowing the labeling, the performance of PCR is usually not as good as PLS.

```
In [9]: def pcr(X,y,pc):

            ''' PCA '''
            pca = PCA()
            Xstd = StandardScaler().fit_transform(X[:,:])
            Xreg = pca.fit_transform(Xstd)[:,:pc]

            ''' linear regression'''
            regr = linear_model.LinearRegression()
            regr.fit(Xreg, y)
            # Calibration and cross validation
            y_c = regr.predict(Xreg)
            y_cv = cross_val_predict(regr, Xreg, 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)

        return(y_cv, score_c, score_cv, mse_c, mse_cv)
```

```
In [10]: predicted, r2r, r2cv, mser, mscv = pcr(data_truncate, label, pc=20)

         # Regression plot
         z = np.polyfit(label, predicted, 1)
         with plt.style.context(('ggplot')):
             fig, ax = plt.subplots(figsize=(9, 6))
             ax.scatter(label, predicted, c='red', edgecolors='k')
             ax.plot(label, z[1]+z[0]*label, c='blue', linewidth=1)
             ax.plot(label, label, color='green', linewidth=1)
             plt.title('$R^{2}$ for cross validation: '+str(r2cv), fontsize=14)
             plt.xlabel('Measured', fontsize=14)
             plt.ylabel('Predicted', fontsize=14)
             plt.xticks(fontsize=14)
             plt.yticks(fontsize=14)
             plt.show()
```

```
Out[10]: <matplotlib.collections.PathCollection at 0x28c21bf4748>

Out[10]: [<matplotlib.lines.Line2D at 0x28c21bf4c18>]

Out[10]: [<matplotlib.lines.Line2D at 0x28c21bfa128>]

Out[10]: Text(0.5,1,'$R^{2}$ for cross validation: 0.963302428471')

Out[10]: Text(0.5,0,'Measured')

Out[10]: Text(0,0.5,'Predicted')

Out[10]: (array([ 83.,  84.,  85.,  86.,  87.,  88.,  89.,  90.]),
          <a list of 8 Text xticklabel objects>)

Out[10]: (array([ 82.,  83.,  84.,  85.,  86.,  87.,  88.,  89.,  90.]),
          <a list of 9 Text yticklabel objects>)
```


R^2 for cross validation: 0.963302428471
Predicted
Measured
83
84
85
86
87
88
89