How do I perform Partial Least Squares in Python using a step-by-step approach?

Authored by

stats writer

April 22, 2024

Partial Least Squares (PLS) is a statistical method used for regression analysis, dimensionality reduction, and prediction modeling. It is commonly used in data analysis and machine learning to handle high-dimensional data with collinearity or multicollinearity. In Python, PLS can be performed using a step-by-step approach to accurately analyze and interpret data.

The first step is to import necessary libraries such as numpy, pandas, and sklearn. Next, the dataset should be loaded and preprocessed to remove any missing values or outliers. Then, the data can be split into training and testing sets.

After that, the PLS model can be instantiated and fitted to the training data. This involves selecting the number of components to use and setting other parameters such as the method for calculating the components. The model can then be used to make predictions on the test data.

To evaluate the performance of the PLS model, metrics such as mean squared error and R-squared can be calculated. These can be compared to the metrics of other models to determine the effectiveness of PLS.

Finally, the PLS model can be used for feature selection and dimensionality reduction, by analyzing the importance of each component in predicting the target variable. This step is important for interpreting the results and understanding the underlying relationships in the data.

In summary, performing Partial Least Squares in Python using a step-by-step approach involves importing libraries, preprocessing data, fitting the model, evaluating performance, and interpreting the results. With this approach, PLS can be effectively utilized for various data analysis and modeling tasks.

Partial Least Squares in Python (Step-by-Step)

One of the most common problems that you'll encounter in machine learning is multicollinearity. This occurs when two or more predictor variables in a dataset are highly correlated.

When this occurs, a model may be able to fit a training dataset well but it may perform poorly on a new dataset

it has never seen because it **overfits** the training set.

One way to get around this problem is to use a method known as **partial least squares**, which works as follows:

Standardize both the predictor and response variables.Calculate *M* linear combinations (called "PLS components") of the original *p* predictor variables that explain a significant amount of variation in both the response variable and the predictor variables.Use the method of least squares to fit a linear regression model using the PLS components as predictors.Use **k-fold cross-validation** to find the optimal number of PLS components to keep in the model.

This tutorial provides a step-by-step example of how to perform partial least squares in Python.

Step 1: Import Necessary Packages

First, we'll import the necessary packages to perform partial least squares in Python:

```
import numpy as np
import pandas as pd
import matplotlib.pyplotas plt
```

```
from sklearn.preprocessingimport scale
from sklearn import model_selection
from sklearn.model_selectionimport RepeatedKFold
from sklearn.model_selectionimport train_test_split
from sklearn.cross_decompositionimport PLSRegressionfromsklearn.metricsimportmean_squared_error
```

Step 2: Load the Data

For this example, we'll use a dataset called mtcars, which contains information about 33 different cars. We'll use hp as the response variable and the following variables as the predictors:

mpgdispdratwtqsec

The following code shows how to load and view this dataset:

```
#define URL where data is located
url = "https://raw.githubusercontent.com/Statology/Python-Guides/main/mtcars.csv"

#read in data
```

```
data_full = pd.read_csv(url)

#select subset of data
data = data_full]

#view first six rows of data
data

mpg disp drat wt qsec hp
0 21.0 160.0 3.90 2.620 16.46 110
1 21.0 160.0 3.90 2.875 17.02 110
2 22.8 108.0 3.85 2.320 18.61 93
3 21.4 258.0 3.08 3.215 19.44 110
4 18.7 360.0 3.15 3.440 17.02 175
5 18.1 225.0 2.76 3.460 20.22 105
```

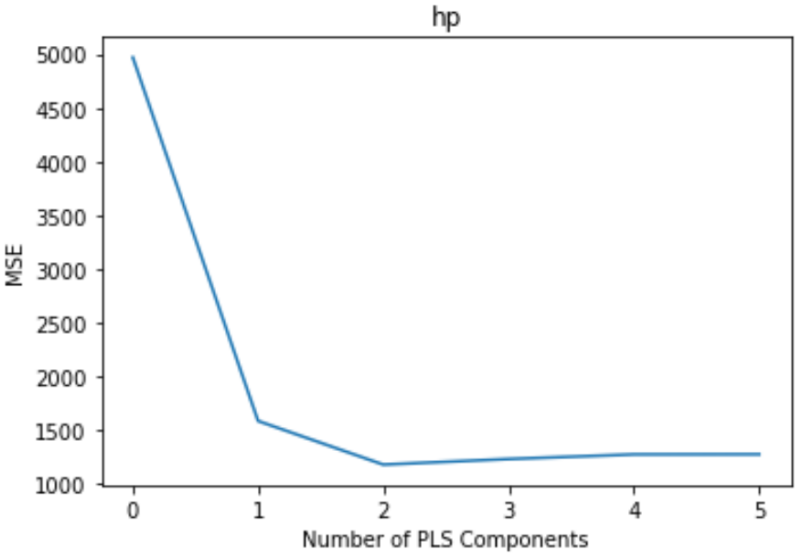
Step 3: Fit the Partial Least Squares Model

The following code shows how to fit the PLS model to this data.

Note that cv = RepeatedKFold() tells Python to use k-fold cross-validation to evaluate the performance of the model. For this example we choose k = 10 folds, repeated 3 times.

```
#define predictor and response variables
X = data]
y = data]

#define cross-validation method
cv = RepeatedKFold(n_splits=10, n_repeats=3, random_state=1)

mse =
n = len(X)

# Calculate MSE with only the intercept
score = -1*model_selection.cross_val_score(PLSRegression(n_components=1), np.ones((n,1)), y, cv=cv, scoring='neg_mean_squared_error').mean()
mse.append(score)

# Calculate MSE using cross-validation, adding one component at a time
for i in np.arange(1, 6):
pls = PLSRegression(n_components=i)
score = -1*model_selection.cross_val_score(pls, scale(X), y, cv=cv,
scoring='neg_mean_squared_error').mean()
```

```
mse.append(score)

#plot test MSE vs. number of components
plt.plot(mse)
plt.xlabel('Number of PLS Components')
plt.ylabel('MSE')
plt.title('hp')
```



The plot displays the number of PLS components along the x-axis and the test MSE (mean squared error) along the y-axis.

From the plot we can see that the test MSE decreases by adding in two PLS components, yet it begins to increase as we add more than two PLS components.

Thus, the optimal model includes just the first two PLS components.

Step 4: Use the Final Model to Make Predictions

We can use the final PLS model with two PLS components to make predictions on new observations.

The following code shows how to split the original

dataset into a training and testing set and use the PLS model with two PLS components to make predictions on the testing set.

```
#split the dataset into training (70%) and testing (30%) sets
X_train,X_test,y_train,y_test = train_test_split(X,y,test_size=0.3,random_state=0)

#calculate RMSE
pls = PLSRegression(n_components=2)
pls.fit(scale(X_train), y_train)np.sqrt(mean_squared_error(y_test, pls.predict(scale(X_test))))
29.9094
```

We can see that the test RMSE turns out to be 29.9094. This is the average deviation between the predicted value for *hp* and the observed value for *hp* for the observations in the testing set.

The complete Python code use in this example can be found here.