A Practical Guide to Partial Least Squares Regression in Python: Addressing Multicollinearity

Authored by

Mohammed looti

November 6, 2025

One of the most persistent challenges encountered in statistical modeling and machine learning is the issue of multicollinearity. This problematic scenario arises when two or more predictor variables within a dataset exhibit a high degree of correlation.

The presence of **multicollinearity** can severely undermine the stability and interpretability of standard linear regression models. While a model might appear to achieve a remarkably good fit on the training data, this high correlation frequently leads to unstable coefficients and overfitting, resulting in poor generalization and highly inaccurate predictions when applied to new, unseen datasets.

Understanding Partial Least Squares (PLS) Methodology

To effectively mitigate the detrimental effects of multicollinearity, practitioners often turn to specialized dimension reduction techniques. Partial Least Squares (PLS) Regression is a powerful, multivariate method designed explicitly for predictive modeling in the presence of highly correlated predictors. Unlike Principal Component Regression (PCR), PLS strategically focuses on maximizing the covariance between the independent (predictor) variables and the dependent (response) variables.

The PLS algorithm operates through a series of sequential steps to extract latent variables, often referred to as PLS components. This methodology ensures that the derived components explain the maximum shared variation in both the input features (X) and the target variable (Y), thereby summarizing the predictive information efficiently.

The core steps involved in the robust PLS methodology are summarized below:

Standardize both the **predictor variables** (X) and the **response variable** (Y) to ensure equal scaling and influence across all features.
Calculate *M* linear combinations (the "PLS components") from the original *p* predictor variables. These components are meticulously constructed to maximize the explanation of variance in both the response and predictor sets simultaneously.
Fit a standard linear regression model utilizing the newly derived PLS components as the independent predictors for the response variable.
Employ robust evaluation methods, such as k-fold cross-validation, to precisely determine the optimal number of PLS components necessary for the final, best-performing predictive model.

This tutorial provides a comprehensive, step-by-step implementation guide demonstrating how to utilize **Partial Least Squares Regression** efficiently within the Python ecosystem, leveraging modules from the popular scikit-learn library.

Step 1: Setting Up the Python Environment and Dependencies

Before commencing the analysis, it is mandatory to configure the Python environment by importing all required libraries and dependencies. Our approach relies heavily on the established data science ecosystem, utilizing packages for numerical computation, efficient data structuring, professional visualization, and, crucially, the specialized machine learning algorithms provided by Scikit-learn.

The following code block initiates the environment setup, importing fundamental packages. This includes **NumPy** for high-performance array operations, **Pandas** for efficient data management, **Matplotlib** for creating informative plots, and specific modules from Scikit-learn essential for data preprocessing (scaling), robust model evaluation (cross-validation), and the core PLS implementation itself (`PLSRegression` from `cross_decomposition`).

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import scale
from sklearn import model_selection
from sklearn.model_selection import RepeatedKFold
from sklearn.model_selection import train_test_split
from sklearn.cross_decomposition import PLSRegression
from sklearn.metrics import mean_squared_error
```

Step 2: Preparing and Loading the Sample Data

For demonstration purposes, we will utilize the well-known **mtcars** dataset, which contains performance metrics for 33 different automobile models. Our primary objective is to predict the vehicle's horsepower (`hp`), which will serve as our **response variable** (Y) throughout the subsequent analysis.

The selection and definition of **predictor variables** (X) are foundational to any successful regression analysis. We have chosen five specific features from the dataset, which are often correlated, making them ideal candidates for a Partial Least Squares modeling approach:

mpg (Miles per Gallon)
disp (Displacement)
drat (Rear Axle Ratio)
wt (Weight)
qsec (1/4 Mile Time)

The following Python code snippet executes the data retrieval process, loading the dataset directly from a public URL, isolating the necessary subset of columns, and confirming successful data integration by displaying the initial six rows of the resultant DataFrame:

#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: Determining the Optimal Number of PLS Components

A critical stage in implementing PLS regression involves the selection of the ideal number of components (latent variables). Using too few components risks model underfitting, while incorporating too many components risks including noise, which can lead to overfitting. To ensure model robustness, we rely on cross-validation to assess performance across a range of component counts.

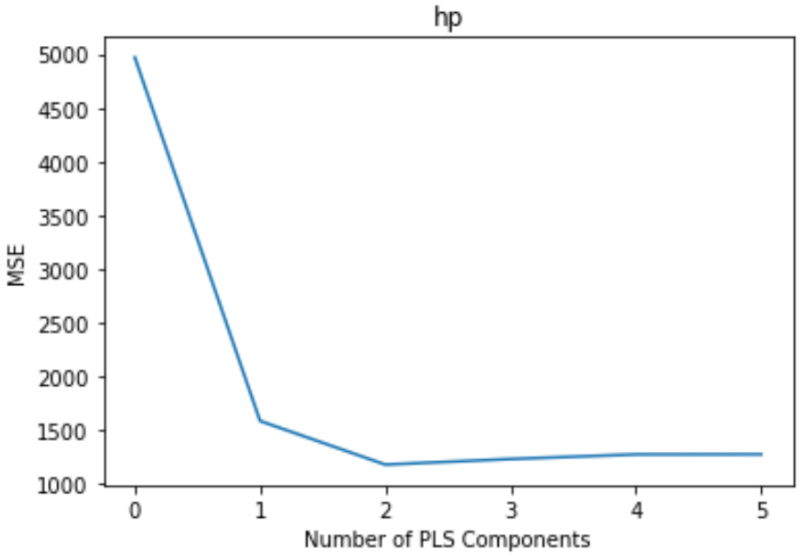
For this analysis, we implement **Repeated K-Fold Cross-Validation**. We set `cv = RepeatedKFold()` with k = 10 folds, repeated 3 times, which provides a highly stable and reliable estimate of the model's true predictive power. The guiding performance metric used to determine component selection is the minimization of the Mean Squared Error (MSE).

The following script systematically calculates the cross-validated MSE for models ranging from 1 to 5 components (corresponding to the maximum number of original predictors). Note that the predictors (`X`) are scaled before fitting the PLS model, a necessary preprocessing step for this technique.

```
#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')
```

Analyzing the Component Selection Plot

The resulting plot visualizes the relationship between the number of PLS components included (X-axis) and the cross-validated Mean Squared Error (MSE) (Y-axis). This visualization is essential for identifying the point of diminishing returns and optimal model complexity.



Upon careful inspection of the graph, we observe that the test MSE drops significantly when moving from one to two PLS components. However, the inclusion of more than two components (three, four, or five) causes the test MSE to stabilize or slightly increase. This pattern strongly indicates that these additional components primarily capture noise rather than meaningful predictive variation.

Consequently, based on the robust cross-validation evidence, the **optimal model complexity** for this dataset is achieved by retaining precisely the first two PLS components.

Step 4: Evaluating the Final PLS Regression Model

Once the optimal number of components (two, in this case) has been determined, the final stage involves rigorously evaluating the model's true predictive capability. This critical step aligns with standard machine learning practice, where model generalization is tested exclusively on a reserved subset of data known as the testing set.

We begin this evaluation by partitioning the complete dataset into two distinct parts: a **training set** (70% of the data, used to calibrate the model parameters) and a **testing set** (30% of the data, reserved for final, unbiased performance assessment). The PLS model is then instantiated using the optimal `n_components=2` and fitted exclusively on the scaled training data.

Finally, we calculate the Root Mean Squared Error (RMSE) on the testing set. RMSE provides a highly interpretable metric, representing the average magnitude of the errors in predicting the `hp` variable in its original units. The calculation is shown below:

#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

The resultant test RMSE is precisely calculated as **29.9094**. This value indicates that, on average, our optimized PLS model predicts the horsepower (`hp`) for a vehicle in the testing set with an error margin of approximately 29.91 units. This comprehensive process successfully concludes the implementation and evaluation of **Partial Least Squares Regression** in Python.

The complete, executable Python script utilized in this demonstration is available for review and download here.