How to do Ridge Regression in Python (Step-by-Step)

Authored by

stats writer

December 18, 2025

Building a robust predictive model often involves navigating complex statistical challenges, chief among which is multicollinearity--a condition where predictor variables are highly correlated with each other. When standard Least Squares Regression methods are applied to such data, the resulting coefficient estimates can become unstable and highly sensitive to small changes in the data. To mitigate this critical issue and enhance model generalization, regularization techniques are employed.

This tutorial provides a comprehensive, step-by-step guide on implementing Ridge regression in Python using the powerful capabilities of the Scikit-learn library. Ridge regression, a form of L2 regularization, introduces a penalty term to the optimization function, effectively shrinking coefficient estimates toward zero. This controlled shrinkage stabilizes the model, improves the prediction accuracy, and remains particularly effective in scenarios involving numerous highly correlated predictors.

We will walk through the entire workflow, starting from importing essential Python packages like pandas and numpy, handling data preparation, fitting the specialized Ridge model from Scikit-learn, optimizing the regularization parameter, and finally, utilizing the resultant model to generate robust predictions for new observations.

The Core Concept: Understanding Ridge Regression

At its heart, Ridge regression serves as an essential tool when the assumptions of classical linear regression are violated by the presence of strong correlations among independent variables. Unlike Ordinary Least Squares (OLS), which focuses solely on minimizing the error between predicted and actual values, Ridge regression introduces a controlled bias to the coefficients to achieve a significant reduction in model variance. This crucial trade-off between bias and variance is fundamental to achieving improved generalization performance on unseen test data, a cornerstone of successful machine learning applications.

The mechanism by which Ridge regression achieves this stability is through the addition of a shrinkage penalty. While OLS aims to find coefficient estimates ($beta_j$) that minimize the Residual Sum of Squares (RSS), Ridge regression modifies this objective function. The RSS represents the total squared difference between the actual response values (y_i) and the predicted response values ($hat{y}_i$), reflecting the inherent error of the model.

The mathematical formulation for OLS minimization is expressed as:

RSS = Σ(yi - ?i)2

Where the terms in the OLS calculation are defined as follows:

Σ: The summation operator, indicating the sum over all observations.

yi: The actual observed response value for the ith observation.
?i: The predicted response value derived from the fitted multiple linear regression model.

Conversely, the Ridge regression objective function includes an additional penalty term, focusing on minimizing the sum of the RSS and the L2 penalty:

RSS + λΣβ**j2**

In this modified equation, the index j spans from 1 to p predictor variables, and $lambda$ (lambda) is the non-negative regularization parameter ($lambda ge 0$). This second term, $lambdaSigmabeta_j^2$, is known as the L2 penalty. Choosing the appropriate value for $lambda$ is critical; it controls the strength of the coefficient shrinkage and is typically selected via cross-validation to yield the lowest possible test Mean Squared Error (MSE).

Setting Up the Environment and Dependencies

The initial stage of any Python machine learning project requires setting up the computational environment by importing the necessary libraries. For Ridge regression, we rely heavily on the ecosystem provided by Scikit-learn for modeling and cross-validation utilities, alongside pandas for efficient data handling and numpy for mathematical operations, particularly defining numerical ranges.

We specifically import the `Ridge` and `RidgeCV` classes from `sklearn.linear_model`. While `Ridge` is used for fitting the model with a predefined penalty strength ($lambda$ or alpha), `RidgeCV` (Ridge Cross-Validation) is essential for automatically searching for the optimal alpha value across a specified range using built-in cross-validation. The `RepeatedKFold` function from `sklearn.model_selection` allows us to define a robust resampling strategy for this optimization process, ensuring the selected alpha is generalizable.

Execute the following code snippet to load all required dependencies into your Python workspace, preparing the necessary infrastructure for data manipulation and statistical modeling:

```
import pandas as pd
from numpy import arange
from sklearn.linear_model import Ridge
from sklearn.linear_model import RidgeCV
from sklearn.model_selection import RepeatedKFold
```

Data Acquisition and Variable Selection

For practical demonstration, we utilize the classic **mtcars** dataset, which contains metrics for 33

different automobile models. This dataset is excellent for regression exercises as it naturally exhibits some internal correlations among features, making it a suitable test case for dealing with multicollinearity. Before fitting any model, it is crucial to clearly define the independent (predictor) variables and the dependent (response) variable.

In this specific implementation of Ridge regression, our goal is to predict the vehicle's **hp** (horsepower) using a subset of related characteristics. Therefore, **hp** is designated as our response variable. The predictor variables selected for modeling are chosen based on their potential influence on horsepower:

mpg: Miles per gallon (fuel efficiency)
wt: Weight (in 1000 lbs)
drat: Rear axle ratio
qsec: Quarter mile time

The code below loads the data directly from a specified GitHub URL using pandas' `read_csv` function. It then isolates the relevant columns--the chosen predictors and the response variable--into a new DataFrame ready for analysis. Reviewing the initial rows of the data ensures that the dataset has been loaded correctly and confirms the structure and type of the input features before moving on to the modeling phase.

#define URL where data is located
url = "https://raw.githubusercontent.com/arabpsychology/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 wt drat qsec hp
0 21.0 2.620 3.90 16.46 110
1 21.0 2.875 3.90 17.02 110
2 22.8 2.320 3.85 18.61 93
3 21.4 3.215 3.08 19.44 110
4 18.7 3.440 3.15 17.02 175
5 18.1 3.460 2.76 20.22 105

Note on Preprocessing: Feature Scaling

Although the provided code example skips explicit scaling steps for simplicity, it is imperative to understand that Ridge regression, like all L1 and L2 regularization methods, is highly sensitive to the scale of the predictor variables. The penalty term ($lambdaSigmabeta_j^2$) penalizes the magnitude of the coefficients. If predictors are measured on vastly different scales (e.g., car weight in tons vs. engine displacement in cubic inches), the coefficients associated with features having naturally larger values will appear smaller in magnitude to compensate, leading to an unfair and disproportionate application of the penalty across features.

Therefore, in virtually every real-world application, a crucial preprocessing step would involve standardizing or normalizing the independent variables (X) before fitting the Ridge model. Standardization ensures that all features contribute equally to the distance calculation in the penalty term, typically resulting in a more robust and statistically sound model. Failure to scale the data correctly can lead to a suboptimal alpha selection and less effective shrinkage.

Configuring and Fitting the Optimal Ridge Model

The most critical aspect of applying Ridge regression is the determination of the optimal regularization parameter, $lambda$. In Scikit-learn, this parameter is commonly referred to as **alpha**. A small alpha value results in coefficients closer to OLS estimates, minimizing bias but risking higher variance. Conversely, a large alpha value leads to excessive shrinkage, minimizing variance but risking high bias (underfitting). To identify the perfect balance, we employ the `RidgeCV()` function, which efficiently integrates model fitting and parameter selection through comprehensive cross-validation.

First, we must formally define our feature matrix **X** (predictors) and our target vector **y** (response) from the loaded DataFrame. We then define a robust cross-validation scheme using `RepeatedKFold()`. In this example, we specify `n_splits=10` (10-fold CV) and `n_repeats=3`, meaning the entire 10-fold procedure will be executed three times. This repetition averages the results across different randomizations, ensuring the alpha selection is reliable and minimizes dependency on the particular train/test splits. We also set a `random_state` for exact reproducibility.

Crucially, we must specify the range of alpha values to test. The default alpha values tested by `RidgeCV()` are often too broad for fine-tuning. We use `numpy.arange(0, 1, 0.01)` to create a finely spaced grid of potential alphas between 0 and 1, testing 100 different values. We set the scoring metric to `'neg_mean_absolute_error'` (Negative Mean Absolute Error), instructing the function to select the alpha that maximizes this score, which corresponds to the minimum absolute error. After fitting the model to the training data, the best determined alpha value is automatically

stored in the `model.alpha_` attribute.

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

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

#define model
model = RidgeCV(alphas=arange(0, 1, 0.01), cv=cv, scoring='neg_mean_absolute_error')

#fit model
model.fit(X, y)

#display lambda that produced the lowest test MSE
print(model.alpha_)

0.99

Upon executing the cross-validation procedure across the defined range of alpha values, the optimal regularization strength that minimized the test Mean Squared Error (or maximized the negative MAE) was determined to be **0.99**. This optimal alpha value is now incorporated into the final fitted Ridge model, ensuring maximum predictive performance and stability for future predictions.

Utilizing the Model for Real-World Predictions

Once the Scikit-learn `RidgeCV` model has been fitted and the optimal alpha (0.99 in our case) has been selected, the model is finalized and ready for inference. The final step involves using the model to forecast the dependent variable (horsepower, *hp*) for new, unseen data points, simulating a real-world application where predictions are required from incoming data.

We define a new hypothetical observation based on specific attributes that fall within the expected range of the training data. This new car is characterized by: 24 mpg, a weight (wt) of 2.5, a rear axle ratio (drat) of 3.5, and a quarter mile time (qsec) of 18.5. It is important to remember that if the training data had been standardized (as recommended in the preprocessing note), this new observation would also need to be standardized using the exact same scaling parameters before being passed to the model's `.predict()` method to ensure consistent calculation of the coefficients.

The `.predict()` method accepts the new feature vector and outputs the estimated value of the

response variable. The resulting array provides the model's statistically informed forecast for the car's horsepower, demonstrating the practical utility of the fitted Ridge regression model in making stable and reliable predictions despite potential multicollinearity in the original feature set.

#define new observation
new =

#predict hp value using ridge regression model
model.predict()

array()

Based on the distinct input features provided for the new car, the finalized Ridge regression model predicts an estimated horsepower value of approximately **104.164**. This concludes the process of setting up, optimizing, and deploying a Ridge regression model in Python. The full source code for this demonstration is available for review here.