How can I implement Multivariate Adaptive Regression Splines in Python?

Authored by

stats writer

April 22, 2024

Multivariate Adaptive Regression Splines (MARS) is a powerful statistical tool used for modeling complex relationships between multiple variables. To implement MARS in Python, one can use the py-earth library which provides a user-friendly interface for creating and fitting MARS models. This library also offers various customization options such as specifying the maximum number of basis functions, controlling the trade-off between model complexity and accuracy, and handling missing values in the data. Additionally, the scikit-learn library also offers an implementation of MARS through the Earth module. By using these libraries, one can easily incorporate MARS into their data analysis and prediction workflows in Python.

Multivariate Adaptive Regression Splines in Python

Multivariate adaptive regression splines (MARS) can be used to model nonlinear relationships between a set of predictor variables and a response variable.

This method works as follows:

1. Divide a dataset into *k* pieces.

2. Fit a regression model to each piece.

3. Use k-fold cross-validation to choose a value for *k*.

This tutorial provides a step-by-step example of how to fit a MARS model to a dataset in Python.

Step 1: Import Necessary Packages

To fit a MARS model in Python, we'll use the Earth() function from sklearn-contrib-py-earth. We'll start by

installing this package:

pip install sklearn-contrib-py-earth

Next, we'll install a few other necessary packages:

```
import pandas as pd
from numpy import mean
from sklearn.model_selectionimport cross_val_score
from sklearn.model_selectionimport RepeatedKFold
from sklearn.datasetsimport make_regression
from pyearth import Earth
```

Step 2: Create a Dataset

For this example we'll use the make_regression() function to create a fake dataset with 5,000 observations and 15 predictor variables:

```
#create fake regression data
X, y = make_regression(n_samples=5000,
n_features=15, n_informative=10,
noise=0.5, random_state=5)
```

Step 3: Build & Optimize the MARS Model

Next, we'll use the Earth() function to build a MARS model and the RepeatedKFold() function to perform k-fold cross-validation to evaluate the model performance.

For this example we'll perform 10-fold cross-validation, repeated 3 times.

```
#define the model
model = Earth()

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

#evaluate model performance
scores = cross_val_score(model, X, y, scoring='neg_mean_absolute_error',
cv=cv, n_jobs=-1)

#print results
mean(scores)
```

-1.745345918289

From the output we can see that the mean absolute error (ignore the negative sign) for this type of model is 1.7453.

In practice we can fit a variety of different models to a given dataset (like Ridge, Lasso, Multiple Linear Regression, Partial Least Squares, Polynomial Regression, etc.) and compare the mean absolute error among all models to determine the one that produces the lowest MAE.

Note that we could also use other metrics to measure error such as adjusted R-squared or mean squared error.

You can find the complete Python code used in this example **here**.

From the output we can see that the mean absolute error (ignore the negative sign) for this type of model is 1.7453.

In practice we can fit a variety of different models to a given dataset (like Ridge, Lasso, Multiple Linear Regression, Partial Least Squares, Polynomial

Regression, etc.) and compare the mean absolute error among all models to determine the one that produces the lowest MAE.

Note that we could also use other metrics to measure error such as adjusted R-squared or mean squared error.

You can find the complete Python code used in this example here.