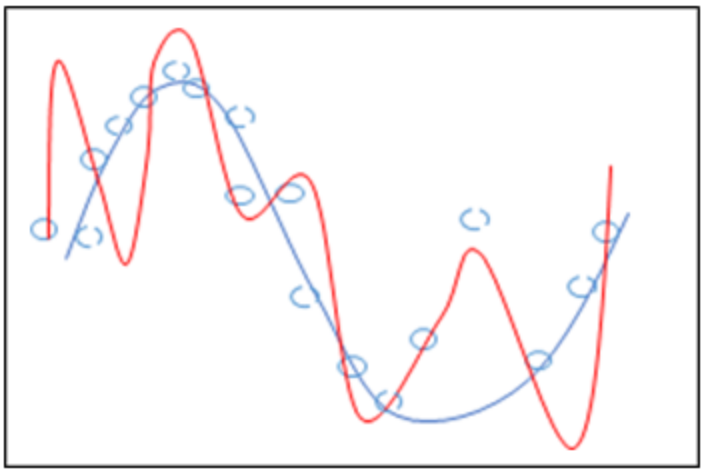
Robert John Follow
Jul 12 · 3 min read

Regularization of Linear Models with SKLearn



An overfit model

Linear models are usually a good starting point for training a model. However, a lot of datasets do not exhibit linear relationships between the independent and the dependent variables. As a result, it is frequently necessary to create a polynomial model. However, these models are usually prone to overfitting. One method of reducing overfitting in polynomial models is through the use of regularization.

Let's start by building a baseline model to determine the required improvement. We will make use of the popular Boston Housing dataset which is available on Kaggle here.

Let's import the necessary libraries and load up our training dataset.

```
#imports
import numpy as np
import pandas as pd
import math

from sklearn.model_selection import train_test_split

from sklearn.linear_model import LinearRegression
from sklearn.linear_model import Ridge
from sklearn.linear_model import Lasso
from sklearn.linear_model import ElasticNet
from sklearn.metrics import mean_squared_error

from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

import matplotlib.pyplot as plt
import seaborn as sns

```

Let's split our data into a training set and a validation set. We will hold out 30% of the data for validation. We will use a random state to make our experiment reproducible.

```
#create our X and y
X = train_df.drop('medv', axis=1)
y = train_df['medv']

```

Let's establish a baseline by training a linear regression model.

```
lr_model = LinearRegression()
lr_model.fit(X_train, y_train)

print('Training score: {}'.format(lr_model.score(X_train,
print('Test score: {}'.format(lr_model.score(X_test, y_tes

y_pred = lr_model.predict(X_test)
mse = mean_squared_error(y_test, y_pred)
```

The model above should give us a training accuracy and a test accuracy of about 72%. We should also get an RMSE of about 4.587. The next models we train should outperform this model with higher accuracy scores and a lower RMSE.

We need to engineer new features. Specifically, we need to create polynomial features by taking our individual features and raising them to a chosen power. Thankfully, scikit-learn has an implementation for this and we don't need to do it manually.

Something else we would like to do is standardize our data. This scales our data down to a range between 0 and 1. This serves the purpose of letting us work with reasonable numbers when we raise to a power.

Finally, because we need to carry out the same operations on our training, validation, and test sets, we will introduce a pipeline. This will let us pipe our process so the same steps get carried out repeatedly.

To summarize, we will scale our data, then create polynomial features, and then train a linear regression model.

```
steps = [
    ('scalar', StandardScaler()),
    ('poly', PolynomialFeatures(degree=2)),
    ('model', LinearRegression())
]

pipeline = Pipeline(steps)

pipeline.fit(X_train, y_train)
```

After running our code, we will get a training accuracy of about 94.75%, and a test accuracy of 46.76%. This is a sign of overfitting. It's normally not a desirable feature, but that is exactly what we were hoping for.

We will now apply regularization to our new data.

l2 Regularization or Ridge Regression

To understand Ridge Regression, we need to remind ourselves of what happens during gradient descent, when our model coefficients are trained. During training, our initial weights are updated according to a gradient update rule using a learning rate and a gradient. Ridge regression adds a penalty to the update, and as a result shrinks the size of our weights. This is implemented in scikit-learn as a class called Ridge.

We will create a new pipeline, this time using Ridge. We will specify our regularization strength by passing in a parameter, alpha. This can be really small, like 0.1, or as large as you would want it to be. The larger the value of alpha, the less variance your model will exhibit.

```
steps = [
    ('scalar', StandardScaler()),
    ('poly', PolynomialFeatures(degree=2)),
    ('model', Ridge(alpha=10, fit_intercept=True))
]

ridge_pipe = Pipeline(steps)
ridge_pipe.fit(X_train, y_train)
```

By executing the code, we should have a training accuracy of about 91.8%, and a test accuracy of about 82.87%. That is an improvement on our baseline linear regression model.

Let's try something else.

l1 Regularization or Lasso Regression

By creating a polynomial model, we created additional features. The question we need to ask ourselves is which of our features are relevant to our model, and which are not.

l1 regularization tries to answer this question by driving the values of certain coefficients down to 0. This eliminates the least important features in our model. We will create a pipeline similar to the one above, but using Lasso. You can play around with the value of alpha, which can range from 0.1 to 1.

```
steps = [
    ('scalar', StandardScaler()),
    ('poly', PolynomialFeatures(degree=2)),
    ('model', Lasso(alpha=0.3, fit_intercept=True))
]

lasso_pipe = Pipeline(steps)

lasso_pipe.fit(X_train, y_train)
```

The code above should give us a training accuracy of 84.8%, and a test accuracy of 83%. This is an even better model than the one we trained earlier.

At this point, you can evaluate your model by finding the RMSE. Don't forget to read the documentation for everything we used.

I hope you found this tutorial useful. Until next time.