Ridge and Lasso Regression

Introduction

At this point, you've seen a number of criteria and algorithms for fitting regression models to data. You've seen the simple linear regression using ordinary least squares, and its more general regression of polynomial functions. You've also seen how we can overfit models to data using polynomials and interactions. With all of that, you began to explore other tools to analyze this general problem of overfitting versus underfitting, all this using training and test splits, bias and variance, and cross validation.

Now you're going to take a look at another way to tune the models you create. These methods all modify the mean squared error function that you are optimizing against. The modifications will add a penalty for large coefficient weights in the resulting model.

Objectives

You will be able to:

- Define lasso regression
- Define ridge regression
- Describe why standardization is necessary before ridge and lasso regression
- Compare and contrast lasso, ridge, and non-regularized regression
- Use lasso and ridge regression with scikit-learn

Linear Regression Cost Function

From an earlier lesson, you know that when solving for a linear regression, you can express the cost function as

$$\text{cost\_function} = \sum_{i=1}^{n}(y_i - \hat{y})^2 = \sum_{i=1}^{n}(y_i - (mx_i + b))^2$$

This is the expression for simple linear regression (for 1 predictor x). If you have multiple predictors, you would have something that looks like:

$$\text{cost\_function} = \sum_{i=1}^{n}(y_i - \hat{y})^2 = \sum_{i=1}^{n}(y_i - \sum_{j=1}^{k}(m_j x_{ij}) - b)^2$$

where k is the number of predictors.

Penalized Estimation

You've seen that **when the number of predictors increases, your model complexity increases, with a higher chance of overfitting as a result**. We've previously seen fairly ad-hoc variable selection methods (such as forward/backward selection), to simply select a few variables from a longer list of variables as predictors.

Now, instead of completely "deleting" certain predictors from a model (which is equal to setting coefficients equal to zero), wouldn't it be interesting to just reduce the values of the coefficients to make them less sensitive to noise in the data? ***Penalized estimation*** operates in a way where parameter shrinkage effects are used to make some or all of the coefficients smaller in magnitude (closer to zero). Some of the penalties have the property of performing both variable selection (setting some coefficients exactly equal to zero) and shrinking the other coefficients.

- They reduce model complexity
- They may prevent overfitting
- Some of them may perform variable selection at the same time (when coefficients are set to 0)
- They can be used to counter multicollinearity

Penalized estimation is a commonly used ***regularization*** technique. **Regularization is a general term used when one tries to battle overfitting.**

Two commonly-used models that use penalized estimation are ***ridge regression*** and ***lasso regression***. While linear regression has been in use since the 1800s, ridge regression was proposed in 1970 (journal article here (https://web.archive.org/web/20180413003503id_/http://math.arizona.edu:80/~hzhang/math574m/Read/Ridge.pdf)) and lasso regression in 1996 (journal article here (https://biostat.app.vumc.org/wiki/pub/Main/MSJournalClub/lasso.pdf)).

Ridge Regression

In ridge regression, the cost function is changed by adding a penalty term to the square of the magnitude of the coefficients.

$$n \quad n \quad k \quad p$$

$$\text{cost\_function\_ridge} = \sum_{i=1}(y_i - \hat{y})^2 = \sum_{i=1}(y_i - \sum_{j=1}(m_j x_{ij}) - b)^2 + \lambda \sum_{j=1} m_j^2$$

If you have two predictors the full equation would look like this (notice that there is a penalty term λm^2 for each predictor in the model - in this case, two) :

$$\text{cost\_function\_ridge} = \sum_{i=1}^{n}(y_i - \hat{y})^2 =$$

$$\sum_{i=1}^{n}(y_i - ((m_1 x_{1i}) - b)^2 + \lambda m_1^2 + (m_2 x_{2i}) - b)^2 + \lambda m_2^2)$$

Remember that you want to minimize your cost function, so by adding the penalty term λ, ridge regression puts a constraint on the coefficients m. This means that large coefficients penalize the optimization function. That's why **ridge regression leads to a shrinkage of the coefficients** and helps to reduce model complexity and multicollinearity.

λ is a ***hyperparameter***, which means you have to specify the value for lambda when creating the model. (This differs from a regular model parameter, which is determined by the fitting algorithm.) For a small lambda, the outcome of your ridge regression will resemble a linear regression model. For large lambda, penalization will increase and more parameters will shrink.

Ridge regression is often also referred to as **L2 norm regularization**. This is the same L2 norm from linear algebra used in the concepts of variance, MSE, and Euclidean distance.

Lasso regression

Lasso regression is very similar to ridge regression, except that the magnitude of the coefficients are not squared in the penalty term. So, while ridge regression keeps the sum of the squared regression coefficients (except for the intercept) bounded, the lasso method bounds the sum of the absolute values.

The resulting cost function looks like this:

$$\text{cost\_function\_lasso} = \sum_{i=1}^{n}(y_i - \hat{y})^2 = \sum_{i=1}^{n}(y_i - \sum_{j=1}^{k}(m_j x_{ij}) - b)^2 + \lambda \sum_{j=1}^{p} |m_j|$$

If you have two predictors the full equation would look like this (notice that there is a penalty term $\lambda |m|$ for each predictor in the model - in this case, two):

$$\text{cost\_function\_lasso} = \sum_{i=1}^{n}(y_i - \hat{y})^2 =$$

$$\sum_{i=1}^{n}(y_i - ((m_1 x_{1i}) - b)^2 + \lambda |m_1|) + ((m_2 x_{2i}) - b)^2 + \lambda |m_2|)$$

The name "lasso" comes from "Least Absolute Shrinkage and Selection Operator".

While it may look similar to the definition of the ridge estimator, the effect of the absolute values is that some coefficients might be set exactly equal to zero, while other coefficients are shrunk towards zero. Hence the lasso method is attractive because it performs estimation *and* selection simultaneously. This is especially valuable for variable selection when the number of predictors is very high.

Lasso regression is often also referred to as **L1 norm regularization**. This is the same L1 norm from linear algebra used in the concepts of absolute deviation, MAE, and Manhattan distance.

Standardization Before Regularization

An important step before using either lasso or ridge regularization is to **first standardize your data such that it is all on the same scale**.

Regularization is based on the concept of penalizing larger coefficients, so if you have features that are on different scales, some will get unfairly penalized.

Below, you can see that we are using a `MinMaxScaler` to standardize our data to the same scale.

Examples Using Our `auto-mpg` Data

Regularized Linear Regression vs. Linear Regression

Let's transform our continuous predictors in `auto-mpg` and see how they perform as predictors in a ridge versus lasso regression.

We import the dataset, separate the target and predictors, and then split the data into training and test sets:

In [1]:
```
import pandas as pd
from sklearn.linear_model import Lasso, Ridge, LinearRegression
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler, PolynomialFeatures

data = pd.read_csv('auto-mpg.csv')

y = data[['mpg']]
X = data.drop(['mpg', 'car name', 'origin'], axis=1)

# Perform test train split
X_train , X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=12)
```

After splitting the data into training and test sets, we use the `MixMaxScaler()` to fit and transform `X_train` and transform `X_test`.

> NOTE: You want to **fit and transform** only the **training** data because in a real-world setting, you only have access to this data. You can then use the same scalar object to **transform** the **test** data. If you first transform the data and then split into training and test sets this leads to **data leakage**.

In [2]:
```
scale = MinMaxScaler()
X_train_transformed = scale.fit_transform(X_train)
X_test_transformed = scale.transform(X_test)
```

We will now fit the ridge, lasso, and linear regression models to the transformed training data. Notice that the ridge and lasso models have the parameter `alpha`, which is scikit-learn's version of λ in the regularization cost functions.

In [3]:
```
# Build a ridge, lasso and regular linear regression model
# Note that in scikit-learn, the regularization parameter is denoted by alpha (and not lambda)
ridge = Ridge(alpha=0.5)
ridge.fit(X_train_transformed, y_train)

lasso = Lasso(alpha=0.5)
lasso.fit(X_train_transformed, y_train)

lin = LinearRegression()
lin.fit(X_train_transformed, y_train)
```

Out[3]:
```
LinearRegression()
```

Next, let's generate predictions for both the training and test sets:

In [4]:
```
# Generate preditions for training and test sets
y_h_ridge_train = ridge.predict(X_train_transformed)
y_h_ridge_test = ridge.predict(X_test_transformed)

y_h_lasso_train = lasso.predict(X_train_transformed)
y_h_lasso_test = lasso.predict(X_test_transformed)

y_h_lin_train = lin.predict(X_train_transformed)
y_h_lin_test = lin.predict(X_test_transformed)
```

Look at the MSE for training and test sets for each of the three models:

In [5]:
```
print('Train Error Ridge Model', mean_squared_error(y_train, y_h_ridge_train))
print('Test Error Ridge Model', mean_squared_error(y_test, y_h_ridge_test))
print('\n')

print('Train Error Lasso Model', mean_squared_error(y_train, y_h_lasso_train))
print('Test Error Lasso Model', mean_squared_error(y_test, y_h_lasso_test))
print('\n')

print('Train Error Unpenalized Linear Model', mean_squared_error(y_train, y_h_lin_train))
print('Test Error Unpenalized Linear Model', mean_squared_error(y_test, y_h_lin_test))
```

```
Train Error Ridge Model 9.798079515529826
Test Error Ridge Model 17.523692433834444


Train Error Lasso Model 16.244450797081786
Test Error Lasso Model 30.034636315030966


Train Error Unpenalized Linear Model 9.700888480581277
Test Error Unpenalized Linear Model 16.74802531396471
```

We note that ridge is clearly better than lasso here, but that the unpenalized model performs best here. This makes sense because a linear regression model with these features is probably not overfitting, so adding regularization just contributes to underfitting.

Let's see how including ridge and lasso changed our parameter estimates.

In [6]:
```
print('Ridge parameter coefficients:', ridge.coef_)
print('Lasso parameter coefficients:', lasso.coef_)
print('Linear model parameter coefficients:', lin.coef_)
```

```
Ridge parameter coefficients: [[ -2.06904445  -2.88593443  -1.81801505 -15.23785349  -1.45594148
    8.1440177 ]]
Lasso parameter coefficients: [-9.09743525 -0.         -0.         -4.02703963  0.          3.92348219]
Linear model parameter coefficients: [[ -1.33790698  -1.05300843  -0.08661412 -19.26724989  -0.37043697
    8.56051229]]
```

Did you notice that lasso shrank a few parameters to 0? The ridge regression mostly affected the fourth parameter (estimated to be -19.26 for the linear regression model).

Regularized Polynomial Regression vs. Polynomial Regression

Now let's compare this to a model built using `PolynomialFeatures`, which has more complexity than an ordinary multiple regression.

In [7]:
```
# Prepare data
poly = PolynomialFeatures(degree=6)
X_train_poly = poly.fit_transform(X_train)
X_test_poly = poly.transform(X_test)

X_train_transformed = scale.fit_transform(X_train_poly)
X_test_transformed = scale.transform(X_test_poly)

# Fit models
ridge.fit(X_train_transformed, y_train)
lasso.fit(X_train_transformed, y_train)
lin.fit(X_train_transformed, y_train)

# Generate predictions
y_h_ridge_train = ridge.predict(X_train_transformed)
y_h_ridge_test = ridge.predict(X_test_transformed)
y_h_lasso_train = lasso.predict(X_train_transformed)
y_h_lasso_test = lasso.predict(X_test_transformed)
y_h_lin_train = lin.predict(X_train_transformed)
y_h_lin_test = lin.predict(X_test_transformed)

# Display results
print('Train Error Polynomial Ridge Model', mean_squared_error(y_train, y_h_ridge_train))
print('Test Error Polynomial Ridge Model', mean_squared_error(y_test, y_h_ridge_test))
print('\n')
print('Train Error Polynomial Lasso Model', mean_squared_error(y_train, y_h_lasso_train))
print('Test Error Polynomial Lasso Model', mean_squared_error(y_test, y_h_lasso_test))
print('\n')
print('Train Error Unpenalized Polynomial Model', mean_squared_error(y_train, y_h_lin_train))
print('Test Error Unpenalized Polynomial Model', mean_squared_error(y_test, y_h_lin_test))
print('\n')
print('Polynomial Ridge Parameter Coefficients:', len(ridge.coef_[ridge.coef_ != 0]),
      'non-zero coefficient(s) and', len(ridge.coef_[ridge.coef_ == 0]), 'zeroed-out coefficient(s)')
print('Polynomial Lasso Parameter Coefficients:',  len(lasso.coef_[lasso.coef_ != 0]),
      'non-zero coefficient(s) and', len(lasso.coef_[lasso.coef_ == 0]), 'zeroed-out coefficient(s)')
print('Polynomial Model Parameter Coefficients:',  len(lin.coef_[lin.coef_ != 0]),
      'non-zero coefficient(s) and', len(lin.coef_[lin.coef_ == 0]), 'zeroed-out coefficient(s)')
```

```
Train Error Polynomial Ridge Model 5.498365263214828
Test Error Polynomial Ridge Model 10.705099905648876


Train Error Polynomial Lasso Model 16.429632826093172
Test Error Polynomial Lasso Model 30.384937999587347


Train Error Unpenalized Polynomial Model 2.4132749692874228e-18
Test Error Unpenalized Polynomial Model 184148.0307538523


Polynomial Ridge Parameter Coefficients: 923 non-zero coefficient(s) and 1 zeroed-out coefficient(s)
Polynomial Lasso Parameter Coefficients: 3 non-zero coefficient(s) and 921 zeroed-out coefficient(s)
Polynomial Model Parameter Coefficients: 924 non-zero coefficient(s) and 0 zeroed-out coefficient(s)
```

In this case, the unpenalized model was overfitting. Therefore when ridge and lasso regression were applied, this reduced overfitting and made the overall model fit better. Note that the best model we have seen so far is the polynomial + ridge model, which seems to have the best balance of bias and variance.

If we were to continue tweaking our models, we might want to reduce the `alpha` (λ) for the lasso model, because it seems to be underfitting compared to the ridge model. Reducing `alpha` would reduce the strength of the regularization, allowing for more non-zero coefficients.

Additional Reading

Full code examples for ridge and lasso regression, advantages and disadvantages, and how to code ridge and lasso in Python can be found here (https://www.analyticsvidhya.com/blog/2016/01/complete-tutorial-ridge-lasso-regression-python/).

Make sure to have a look at the scikit-learn documentation for `Ridge` (https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Ridge.html) and `Lasso` (https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Lasso.html).

Summary

Great! You now know how to perform lasso and ridge regression. Let's move on to the lab so you can use these!