CHAPTER 13

Linear Regression

13.0 Introduction

Linear regression is one of the simplest supervised learning algorithms in our toolkit. If you have ever taken an introductory statistics course in college, likely the final topic you covered was linear regression. In fact, it is so simple that it is sometimes not considered machine learning at all! Whatever you believe, the fact is that linear regression—and its extensions—continues to be a common and useful method of making predictions when the target vector is a quantitative value (e.g., home price, age). In this chapter we will cover a variety of linear regression methods (and some extensions) for creating well-performing prediction models.

13.1 Fitting a Line

Problem

You want to train a model that represents a linear relationship between the feature and target vector.

Solution

Use a linear regression (in scikit-learn, `LinearRegression`):

```
# Load libraries
from sklearn.linear_model import LinearRegression
from sklearn.datasets import load_boston

# Load data with only two features
boston = load_boston()
features = boston.data[:,0:2]
target = boston.target
```

```
# Create linear regression
regression = LinearRegression()

# Fit the linear regression
model = regression.fit(features, target)
```

Discussion

Linear regression assumes that the relationship between the features and the target vector is approximately linear. That is, the *effect* (also called *coefficient*, *weight*, or *parameter*) of the features on the target vector is constant. In our solution, for the sake of explanation we have trained our model using only two features. This means our linear model will be:

$$\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 x_1 + \hat{\beta}_2 x_2 + \epsilon$$

where ŷ is our target, x_i is the data for a single feature, $\hat{\beta}_1$ and $\hat{\beta}_2$ are the coefficients identified by fitting the model, and ϵ is the error. After we have fit our model, we can view the value of each parameter. For example, $\hat{\beta}_0$, also called the *bias* or *intercept*, can be viewed using `intercept_`:

```
# View the intercept
model.intercept_
```

```
22.46681692105723
```

And $\hat{\beta}_1$ and $\hat{\beta}_2$ are shown using `coef_`:

```
# View the feature coefficients
model.coef_
```

```
array([-0.34977589,  0.11642402])
```

In our dataset, the target value is the median value of a Boston home (in the 1970s) in thousands of dollars. Therefore the price of the first home in the dataset is:

```
# First value in the target vector multiplied by 1000
target[0]*1000
```

```
24000.0
```

Using the `predict` method, we can predict a value for that house:

```
# Predict the target value of the first observation, multiplied by 1000
model.predict(features)[0]*1000
```

```
24560.23872370844
```

Not bad! Our model was only off by $560.24!

The major advantage of linear regression is its interpretability, in large part because the coefficients of the model are the effect of a one-unit change on the target vector. For example, the first feature in our solution is the number of crimes per resident. Our model's coefficient of this feature was ~–0.35, meaning that if we multiply this coefficient by 1,000 (since the target vector is the house price in thousands of dollars), we have the change in house price for each additional one crime per capita:

```
# First coefficient multiplied by 1000
model.coef_[0]*1000
```

```
-349.77588707748947
```

This says that every single crime per capita will decrease the price of the house by approximately $350!

13.2 Handling Interactive Effects

Problem

You have a feature whose effect on the target variable depends on another feature.

Solution

Create an interaction term to capture that dependence using scikit-learn's `PolynomialFeatures`:

```
# Load libraries
from sklearn.linear_model import LinearRegression
from sklearn.datasets import load_boston
from sklearn.preprocessing import PolynomialFeatures

# Load data with only two features
boston = load_boston()
features = boston.data[:,0:2]
target = boston.target

# Create interaction term
interaction = PolynomialFeatures(
    degree=3, include_bias=False, interaction_only=True)
features_interaction = interaction.fit_transform(features)

# Create linear regression
regression = LinearRegression()

# Fit the linear regression
model = regression.fit(features_interaction, target)
```

Discussion

Sometimes a feature's effect on our target variable is at least partially dependent on another feature. For example, imagine a simple coffee-based example where we have two binary features—the presence of sugar (`sugar`) and whether or not we have stirred (`stirred`)—and we want to predict if the coffee tastes sweet. Just putting sugar in the coffee (`sugar=1, stirred=0`) won't make the coffee taste sweet (all the sugar is at the bottom!) and just stirring the coffee without adding sugar (`sugar=0, stirred=1`) won't make it sweet either. Instead it is the interaction of putting sugar in the coffee *and* stirring the coffee (`sugar=1, stirred=1`) that will make a coffee taste sweet. The effects of `sugar` and `stir` on sweetness are dependent on each other. In this case we say there is an interaction effect between the features `sugar` and `stirred`.

We can account for interaction effects by including a new feature comprising the product of corresponding values from the interacting features:

$$\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 x_1 + \hat{\beta}_2 x_2 + \hat{\beta}_3 x_1 x_2 + \epsilon$$

where x_1 and x_2 are the values of the `sugar` and `stirred`, respectively, and $x_1 x_2$ represents the interaction between the two.

In our solution, we used a dataset containing only two features. Here is the first observation's values for each of those features:

```
# View the feature values for first observation
features[0]
```

```
array([  6.32000000e-03,   1.80000000e+01])
```

To create an interaction term, we simply multiply those two values together for every observation:

```
# Import library
import numpy as np

# For each observation, multiply the values of the first and second feature
interaction_term = np.multiply(features[:, 0], features[:, 1])
```

We can then view the interaction term for the first observation:

```
# View interaction term for first observation
interaction_term[0]
```

```
0.11376
```

However, while often we will have a substantive reason for believing there is an interaction between two features, sometimes we will not. In those cases it can be useful to use scikit-learn's `PolynomialFeatures` to create interaction terms for all combina-

tions of features. We can then use model selection strategies to identify the combination of features and interaction terms that produce the best model.

To create interaction terms using `PolynomialFeatures`, there are three important parameters we must set. Most important, `interaction_only=True` tells `PolynomialFeatures` to only return interaction terms (and not polynomial features, which we will discuss in Recipe 13.3). By default, `PolynomialFeatures` will add a feature containing ones called a bias. We can prevent that with `include_bias=False`. Finally, the `degree` parameter determines the maximum number of features to create interaction terms from (in case we wanted to create an interaction term that is the combination of three features). We can see the output of `PolynomialFeatures` from our solution by checking to see if the first observation's feature values and interaction term value match our manually calculated version:

```
# View the values of the first observation
features_interaction[0]
```

```
array([  6.32000000e-03,   1.80000000e+01,   1.13760000e-01])
```

13.3 Fitting a Nonlinear Relationship

Problem

You want to model a nonlinear relationship.

Solution

Create a polynomial regression by including polynomial features in a linear regression model:

```
# Load library
from sklearn.linear_model import LinearRegression
from sklearn.datasets import load_boston
from sklearn.preprocessing import PolynomialFeatures

# Load data with one feature
boston = load_boston()
features = boston.data[:,0:1]
target = boston.target

# Create polynomial features x^2 and x^3
polynomial = PolynomialFeatures(degree=3, include_bias=False)
features_polynomial = polynomial.fit_transform(features)

# Create linear regression
regression = LinearRegression()

# Fit the linear regression
model = regression.fit(features_polynomial, target)
```

Discussion

So far we have only discussed modeling linear relationships. An example of a linear relationship would be the number of stories a building has and the building's height. In linear regression, we assume the effect of number of stories and building height is approximately constant, meaning a 20-story building will be roughly twice as high as a 10-story building, which will be roughly twice as high as a 5-story building. Many relationships of interest, however, are not strictly linear.

Often we want to model a non-linear relationship—for example, the relationship between the number of hours a student studies and the score she gets on the test. Intuitively, we can imagine there is a big difference in test scores between students who study for one hour compared to students who did not study at all. However, there is a much smaller difference in test scores between a student who studied for 99 hours and a student who studied for 100 hours. The effect one hour of studying has on a student's test score decreases as the number of hours increases.

Polynomial regression is an extension of linear regression to allow us to model non-linear relationships. To create a polynomial regression, convert the linear function we used in Recipe 13.1:

$$\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 x_1 + \epsilon$$

into a polynomial function by adding polynomial features:

$$\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 x_1 + \hat{\beta}_2 x_1^2 + \ldots + \hat{\beta}_d x_i^d + \epsilon$$

where d is the degree of the polynomial. How are we able to use a linear regression for a nonlinear function? The answer is that we do not change how the linear regression fits the model, but rather only add polynomial features. That is, the linear regression does not "know" that the x^2 is a quadratic transformation of x. It just considers it one more variable.

A more practical description might be in order. To model nonlinear relationships, we can create new features that raise an existing feature, x, up to some power: x^2, x^3, and so on. The more of these new features we add, the more flexible the "line" created by our model. To make this more explicit, imagine we want to create a polynomial to the third degree. For the sake of simplicity, we will focus on only one observation (the first observation in the dataset), x_0:

```
# View first observation
features[0]
```

```
array([ 0.00632])
```

To create a polynomial feature, we would raise the first observation's value to the second degree, x_1^2:

```
# View first observation raised to the second power, x^2
features[0]**2
```

```
array([  3.99424000e-05])
```

This would be our new feature. We would then also raise the first observation's value to the third degree, x_1^3:

```
# View first observation raised to the third power, x^3
features[0]**3
```

```
array([  2.52435968e-07])
```

By including all three features (x, x^2, and x^3) in our feature matrix and then running a linear regression, we have conducted a polynomial regression:

```
# View the first observation's values for x, x^2, and x^3
features_polynomial[0]
```

```
array([  6.32000000e-03,   3.99424000e-05,   2.52435968e-07])
```

`PolynomialFeatures` has two important parameters. First, `degree` determines the maximum number of degrees for the polynomial features. For example, `degree=3` will generate x^2 and x^3. Finally, by default `PolynomialFeatures` includes a feature containing only ones (called a bias). We can remove that by setting `include_bias=False`.

13.4 Reducing Variance with Regularization

Problem

You want to reduce the variance of your linear regression model.

Solution

Use a learning algorithm that includes a *shrinkage penalty* (also called *regularization*) like ridge regression and lasso regression:

```
# Load libraries
from sklearn.linear_model import Ridge
from sklearn.datasets import load_boston
from sklearn.preprocessing import StandardScaler

# Load data
boston = load_boston()
features = boston.data
target = boston.target

# Standardize features
scaler = StandardScaler()
```

```
features_standardized = scaler.fit_transform(features)

# Create ridge regression with an alpha value
regression = Ridge(alpha=0.5)

# Fit the linear regression
model = regression.fit(features_standardized, target)
```

Discussion

In standard linear regression the model trains to minimize the sum of squared error between the true (y_i) and prediction, ($\hat{y}_i$) target values, or residual sum of squares (RSS):

$$RSS = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

Regularized regression learners are similar, except they attempt to minimize RSS *and* some penalty for the total size of the coefficient values, called a shrinkage penalty because it attempts to "shrink" the model. There are two common types of regularized learners for linear regression: ridge regression and the lasso. The only formal difference is the type of shrinkage penalty used. In ridge regression, the shrinkage penalty is a tuning hyperparameter multiplied by the squared sum of all coefficients:

$$\text{RSS} + \alpha \sum_{j=1}^{p} \hat{\beta}_j^2$$

where $\hat{\beta}_j$ is the coefficient of the jth of p features and α is a hyperparameter (discussed next). The lasso is similar, except the shrinkage penalty is a tuning hyperparameter multiplied by the sum of the absolute value of all coefficients:

$$\frac{1}{2n}\text{RSS} + \alpha \sum_{j=1}^{p} \left|\hat{\beta}_j\right|$$

where n is the number of observations. So which one should we use? As a very general rule of thumb, ridge regression often produces slightly better predictions than lasso, but lasso (for reasons we will discuss in Recipe 13.5) produces more interpretable models. If we want a balance between ridge and lasso's penalty functions we can use elastic net, which is simply a regression model with both penalties included. Regardless of which one we use, both ridge and lasso regressions can penalize large or complex models by including coefficient values in the loss function we are trying to minimize.

The hyperparameter, α, lets us control how much we penalize the coefficients, with higher values of α creating simpler models. The ideal value of α should be tuned like any other hyperparameter. In scikit-learn, α is set using the `alpha` parameter.

scikit-learn includes a `RidgeCV` method that allows us to select the ideal value for α:

```
# Load library
from sklearn.linear_model import RidgeCV

# Create ridge regression with three alpha values
regr_cv = RidgeCV(alphas=[0.1, 1.0, 10.0])

# Fit the linear regression
model_cv = regr_cv.fit(features_standardized, target)

# View coefficients
model_cv.coef_
```

```
array([-0.91215884,  1.0658758 ,  0.11942614,  0.68558782, -2.03231631,
        2.67922108,  0.01477326, -3.0777265 ,  2.58814315, -2.00973173,
       -2.05390717,  0.85614763, -3.73565106])
```

We can then easily view the best model's α value:

```
# View alpha
model_cv.alpha_
```

```
1.0
```

One final note: because in linear regression the value of the coefficients is partially determined by the scale of the feature, and in regularized models all coefficients are summed together, we must make sure to standardize the feature prior to training.

13.5 Reducing Features with Lasso Regression

Problem

You want to simplify your linear regression model by reducing the number of features.

Solution

Use a lasso regression:

```
# Load library
from sklearn.linear_model import Lasso
from sklearn.datasets import load_boston
from sklearn.preprocessing import StandardScaler

# Load data
boston = load_boston()
```

```
features = boston.data
target = boston.target

# Standardize features
scaler = StandardScaler()
features_standardized = scaler.fit_transform(features)

# Create lasso regression with alpha value
regression = Lasso(alpha=0.5)

# Fit the linear regression
model = regression.fit(features_standardized, target)
```

Discussion

One interesting characteristic of lasso regression's penalty is that it can shrink the coefficients of a model to zero, effectively reducing the number of features in the model. For example, in our solution we set `alpha` to `0.5` and we can see that many of the coefficients are 0, meaning their corresponding features are not used in the model:

```
# View coefficients
model.coef_
```

```
array([-0.10697735,  0.        , -0.        ,  0.39739898, -0.        ,
        2.97332316, -0.        , -0.16937793, -0.        , -0.        ,
       -1.59957374,  0.54571511, -3.66888402])
```

However, if we increase α to a much higher value, we see that literally none of the features are being used:

```
# Create lasso regression with a high alpha
regression_a10 = Lasso(alpha=10)
model_a10 = regression_a10.fit(features_standardized, target)
model_a10.coef_
```

```
array([-0.,  0., -0.,  0., -0.,  0., -0.,  0., -0., -0., -0.,  0., -0.])
```

The practical benefit of this effect is that it means that we could include 100 features in our feature matrix and then, through adjusting lasso's α hyperparameter, produce a model that uses only 10 (for instance) of the most important features. This lets us reduce variance while improving the interpretability of our model (since fewer features is easier to explain).