Linear Regression

Linear Regression Basics

Linear regression is a predictive modeling technique for predicting a numeric response variable based on one or more explanatory variables. The term "regression" in predictive modeling generally refers to any modeling task that involves predicting a real number (as opposed classification, which involves predicting a category or class.). The term "linear" in the name linear regression refers to the fact that the method models data with linear combination of the explanatory variables. A linear combination is an expression where one or more variables are scaled by a constant factor and added together. In the case of linear regression with a single explanatory variable, the linear combination used in linear regression can be expressed as:

$$\text{response} = \text{intercept} + \text{constant} * \text{explanatory}$$

The right side if the equation defines a line with a certain y-intercept and slope times the explanatory variable. In other words, linear regression in its most basic form fits a straight line to the response variable. The model is designed to fit a line that minimizes the squared differences (also called errors or residuals.). We won't go into all the math behind how the model actually minimizes the squared errors, but the end result is a line intended to give the "best fit" to the data. Since linear regression fits data with a line, it is most effective in cases where the response and explanatory variable have a linear relationship.

Let's revisit the mtcars data set and use linear regression to predict vehicle gas mileage based on vehicle weight. First, let's load some libraries and look at a scatterplot of weight and mpg to get a sense of the shape of the data:

```
%matplotlib inline
import numpy as np
import pandas as pd
from ggplot import mtcars
import matplotlib
import matplotlib.pyplot as plt
import scipy.stats as stats
matplotlib.style.use('ggplot')
mtcars.plot(kind="scatter",
           x="wt",
           y="mpg",
           figsize=(9,9),
           color="black")
<matplotlib.axes._subplots.AxesSubplot at 0xb257198>
```

The scatterplot shows a roughly linear relationship between weight and mpg, suggesting a linear regression model might work well.

Python's scikit-learn library contains a wide range of functions for predictive modeling. Let's load its linear regression training function and fit a line to the mtcars data:

```
from sklearn import linear_model
# Initialize model
regression_model = linear_model.LinearRegression()

# Train the model using the mtcars data
regression_model.fit(X = pd.DataFrame(mtcars["wt"]),
                     y = mtcars["mpg"])

# Check trained model y-intercept
print(regression_model.intercept_)

# Check trained model coefficients
print(regression_model.coef_)
```

```
37.2851261673
[-5.34447157]
```

The output above shows the model intercept and coefficients used to create the best fit line. In this case the y-intercept term is set to 37.2851 and the coefficient for the weight variable is -5.3445. In other words, the model fit the line mpg = 37.2851 - 5.3445*wt.

We can get a sense of how much of the variance in the response variable is explained by the model using the model.score() function:

```
regression_model.score(X = pd.DataFrame(mtcars["wt"]),
                       y = mtcars["mpg"])
```

```
0.75283279365826461
```

The output of the score function for linear regression is "R-squared", a value that ranges from 0 to 1 which describes the proportion of variance in the response variable that is explained by the model. In this case, car weight explains roughly 75% of the variance in mpg.

The R-squared measure is based on the residuals: differences between what the model predicts for each data point and the actual value of each data point. We can extract the model's residuals by making a prediction with the model on the data and then subtracting the actual value from each prediction:

```
train_prediction = regression_model.predict(X = pd.DataFrame(mtcars["wt"]))

# Actual - prediction = residuals
residuals = mtcars["mpg"] - train_prediction

residuals.describe()
```

```
count    3.200000e+01
mean    -5.107026e-15
std      2.996352e+00
min     -4.543151e+00
25%     -2.364709e+00
50%     -1.251956e-01
75%      1.409561e+00
max      6.872711e+00
```

```
Name: mpg, dtype: float64
```

R-squared is calculated as 1 - (SSResiduals/SSTotal) were SSResiduals is the sum of the squares of the model residuals and SSTotal is the sum of the squares of the difference between each data point and the mean of the data. We could calculate R-squared by hand like this:

```
SSResiduals = (residuals**2).sum()

SSTotal = ((mtcars["mpg"] - mtcars["mpg"].mean())**2).sum()

# R-squared
1 - (SSResiduals/SSTotal)
```

```
0.75283279365826461
```

Now that we have a linear model, let's plot the line it fits on our scatterplot to get a sense of how well it fits the data:

```
mtcars.plot(kind="scatter",
           x="wt",
           y="mpg",
           figsize=(9,9),
           color="black",
           xlim = (0,7))

# Plot regression line
plt.plot(mtcars["wt"],      # Explanitory variable
         train_prediction,  # Predicted values
         color="blue")
```

```
 [<matplotlib.lines.Line2D at 0xb9d67b8>]
```

The regression line looks like a reasonable fit and it follows our intuition: as car weight increases we would expect fuel economy to decline.

Outliers can have a large influence on linear regression models: since regression deals with minimizing squared residuals, large residuals have a disproportionately large influence on the model. Plotting the result helps us detect influential outliers. In this case there does not appear to be any influential outliers. Let's add an outlier--a super heavy fuel efficient car--and plot a new regression model:

```
mtcars_subset = mtcars[["mpg","wt"]]

super_car = pd.DataFrame({"mpg":50,"wt":10}, index=["super"])

new_cars = mtcars_subset.append(super_car)

# Initialize model
regression_model = linear_model.LinearRegression()

# Train the model using the new_cars data
regression_model.fit(X = pd.DataFrame(new_cars["wt"]),
                      y = new_cars["mpg"])

train_prediction2 = regression_model.predict(X = pd.DataFrame(new_cars["wt"
]))

# Plot the new model
new_cars.plot(kind="scatter",
           x="wt",
           y="mpg",
           figsize=(9,9),
           color="black", xlim=(1,11), ylim=(10,52))

# Plot regression line
plt.plot(new_cars["wt"],      # Explanatory variable
         train_prediction2,   # Predicted values
         color="blue")
 [<matplotlib.lines.Line2D at 0xb9d6748>]
```

Although this is an extreme, contrived case, the plot above illustrates how much influence a single outlier can have on a linear regression model.

In a well-behaved linear regression model, we'd like the residuals to be roughly normally distributed. That is, we'd like a roughly even spread of error above and below the regression line. We can investigate the normality of residuals with a Q-Q (quantile-quantile) plot. Make a qqplot by passing the residuals to the stats.probplot() function in the scipy.stats library:

```
plt.figure(figsize=(9,9))

stats.probplot(residuals, dist="norm", plot=plt)
 ((array([-2.02511189, -1.62590278, -1.38593914, -1.20666642, -1.05953591,
         -0.93235918, -0.81872017, -0.71478609, -0.6180591 , -0.52680137,
         -0.43973827, -0.35589149, -0.27447843, -0.19484777, -0.11643566,
         -0.03873405,  0.03873405,  0.11643566,  0.19484777,  0.27447843,
          0.35589149,  0.43973827,  0.52680137,  0.6180591 ,  0.71478609,
          0.81872017,  0.93235918,  1.05953591,  1.20666642,  1.38593914,
          1.62590278,  2.02511189]),
  array([-4.54315128, -3.90536265, -3.72686632, -3.46235533, -3.20536265,
         -2.97258623, -2.78093991, -2.61100374, -2.28261065, -2.08595212,
         -1.88302362, -1.10014396, -1.0274952 , -0.9197704 , -0.69325453,
         -0.20014396, -0.0502472 ,  0.152043  ,  0.29985604,  0.35642633,
          0.86687313,  1.17334959,  1.20105932,  1.29734994,  1.74619542,
          2.10328764,  2.34995929,  2.46436703,  4.16373815,  5.98107439,
          6.42197917,  6.87271129])),
 (3.032779748945897, -4.8544962703334722e-15, 0.9756674451791317 3))
```

When residuals are normally distributed, they tend to lie along the straight line on the Q-Q plot. In this case residuals appear to follow a slightly non-linear pattern: the residuals are bowed a bit away from the normality line on each end. This is an indication that simple straight line might not be sufficient to fully describe the relationship between weight and mpg.

After making model predictions, it is useful to have some sort of metric to evaluate oh well the model performed. Adjusted R-squared is one useful measure, but it only applies to the regression model itself: we'd like some universal evaluation metric that lets us compare the performance of different types of models. Root mean squared error (RMSE) is a common evaluation metric for predictions involving real numbers. Root mean squared error is square root of the average of the squared error (residuals.). If you recall, we wrote a function to calculate RMSE way back in lesson lesson 12:

```
def rmse(predicted, targets):
    """
    Computes root mean squared error of two numpy ndarrays

    Args:
        predicted: an ndarray of predictions
        targets: an ndarray of target values

    Returns:
        The root mean squared error as a float
    """
    return (np.sqrt(np.mean((targets-predicted)**2)))

rmse(train_prediction, mtcars["mpg"])
2.9491626859550282
```

Instead of defining your own RMSE function, you can use the scikit-learn library's mean squared error function and take the square root of the result:

```
from sklearn.metrics import mean_squared_error

RMSE = mean_squared_error(train_prediction, mtcars["mpg"])**0.5

RMSE
2.9491626859550282
```