Regression Model Validation - Lab

Introduction

In this lab, you'll be able to validate your model using train-test-split.

Objectives

You will be able to:

- Calculate the mean squared error (MSE) as a measure of predictive performance
- Validate the model using the test data

Let's use our Boston Housing Data again!

This time, let's only include the variables that were previously selected using recursive feature elimination. We included the code to preprocess below.

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
from sklearn.datasets import load_boston

boston = load_boston()

boston_features = pd.DataFrame(boston.data, columns = boston.feature_names)
b = boston_features["B"]
logdis = np.log(boston_features["DIS"])
loglstat = np.log(boston_features["LSTAT"])

# minmax scaling
boston_features["B"] = (b-min(b))/(max(b)-min(b))
boston_features["DIS"] = (logdis-min(logdis))/(max(logdis)-min(logdis))

#standardization
boston_features["LSTAT"] = (loglstat-np.mean(loglstat))/np.sqrt(np.var(loglstat))
```

In [2]:
```
X = boston_features[['CHAS', 'RM', 'DIS', 'B', 'LSTAT']]
y = pd.DataFrame(boston.target,columns = ["target"])
```

Perform a train-test-split

In [3]:
```
from sklearn.model_selection import train_test_split
```

In [4]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y)
```

In [5]:
```
#A brief preview of our train test split
print(len(X_train), len(X_test), len(y_train), len(y_test))
```

```
379 127 379 127
```

Apply your model to the train set

Importing and initializing the model class

In [6]:
```
from sklearn.linear_model import LinearRegression
linreg = LinearRegression()
```

Fitting the model to the train data

In [7]:
```
linreg.fit(X_train, y_train)
```

Out[7]:
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1, normalize=False)
```

Calculating predictions on the train set, and on the test set

In [8]:
```
y_hat_train = linreg.predict(X_train)
y_hat_test = linreg.predict(X_test)
```

Calculating your residuals

In [9]:
```
train_residuals = y_hat_train - y_train
test_residuals = y_hat_test - y_test
```

Calculating the Mean Squared Error

A good way to compare overall performance is to compare the mean squarred error for the predicted values on the train and test sets.

In [10]:
```
from sklearn.metrics import mean_squared_error
```

In [11]:
```
train_mse = mean_squared_error(y_train, y_hat_train)
test_mse = mean_squared_error(y_test, y_hat_test)
print('Train Mean Squarred Error:', train_mse)
print('Test Mean Squarred Error:', test_mse)
```

```
Train Mean Squarred Error: 21.620204537961026
Test Mean Squarred Error: 22.547316698156916
```

If your test error is substantially worse then our train error, this is a sign that our model doesn't generalize well to future cases.

One simple way to demonstrate overfitting and underfitting is to alter the size of our train test split. By default, scikit learn's built in method allocates 25% of the data to the test set and 75% to the training set. Fitting a model on only 10% of the data is apt to lead to underfitting, while training a model on 99% of the data is apt to lead to overfitting.

Evaluating the effect of train-test split size

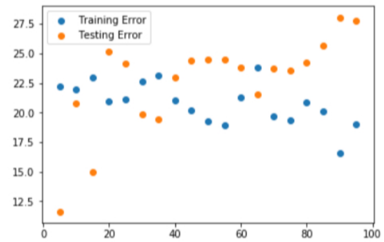
Iterate over a range of train-test split sizes from .5 to .95. For each of these, generate a new train/test split sample. Fit a model to the training sample and calculate both the training error and the test error (mse) for each of these splits. Plot these two curves (train error vs. training size and test error vs. training size) on a graph.

In [12]:

```
import random
random.seed(11)

train_err = []
test_err = []
t_sizes = list(range(5,100,5))
for t_size in t_sizes:
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=t_size/100)
    linreg.fit(X_train, y_train)
    y_hat_train = linreg.predict(X_train)
    y_hat_test = linreg.predict(X_test)
    train_err.append(mean_squared_error(y_train, y_hat_train))
    test_err.append(mean_squared_error(y_test, y_hat_test))
plt.scatter(t_sizes, train_err, label='Training Error')
plt.scatter(t_sizes, test_err, label='Testing Error')
plt.legend()
```

Out[12]: <matplotlib.legend.Legend at 0x1a24d6cef0>



Evaluating the effect of train-test split size: extension

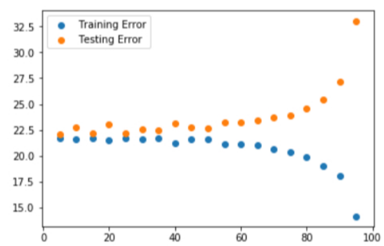
Repeat the previous example, but for each train-test split size, generate 100 iterations of models/errors and save the average train/test error. This will help account for any particularly good/bad models that might have resulted from poor/good splits in the data.

In [13]:

```
random.seed(8)

train_err = []
test_err = []
t_sizes = list(range(5,100,5))
for t_size in t_sizes:
    temp_train_err = []
    temp_test_err = []
    for i in range(100):
        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=t_size/100)
        linreg.fit(X_train, y_train)
        y_hat_train = linreg.predict(X_train)
        y_hat_test = linreg.predict(X_test)
        temp_train_err.append(mean_squared_error(y_train, y_hat_train))
        temp_test_err.append(mean_squared_error(y_test, y_hat_test))
    train_err.append(np.mean(temp_train_err))
    test_err.append(np.mean(temp_test_err))
plt.scatter(t_sizes, train_err, label='Training Error')
plt.scatter(t_sizes, test_err, label='Testing Error')
plt.legend()
```

Out[13]: <matplotlib.legend.Legend at 0x1a26e93438>



What's happening here? evaluate your result!

Summary

Congratulations! You now practiced your knowledge on MSE and on using train-test-split.