model_evaluation_and_refinement

December 20, 2018

Link
Data Analysis with Python

1 Module 5: Model Evaluation and Refinement

We have built models and made predictions of vehicle prices. Now we will determine how accurate these predictions are.

2 Table of contents

```
In [1]: import pandas as pd
        import numpy as np

        # Import clean data
        path = path='https://ibm.box.com/shared/static/q6iiqb1pd7wo8r3q28jvgsrprzezjqk3.csv'
        print('Data is read!')
        df = pd.read_csv(path)
```

```
Data is read!
```

```
In [2]: df.head()
```

```
Out[2]:    Unnamed: 0  symboling  normalized-losses         make aspiration  \
        0           0          3                122  alfa-romero        std
        1           1          3                122  alfa-romero        std
        2           2          1                122  alfa-romero        std
        3           3          2                164         audi        std
        4           4          2                164         audi        std

          num-of-doors   body-style drive-wheels engine-location  wheel-base ...   \
        0          two  convertible          rwd           front        88.6 ...
```

```
1          two  convertible          rwd          front        88.6 ...
2          two    hatchback          rwd          front        94.5 ...
3         four        sedan          fwd          front        99.8 ...
4         four        sedan          4wd          front        99.4 ...

   compression-ratio  horsepower  peak-rpm  city-mpg highway-mpg    price  \
0                9.0       111.0    5000.0        21           27  13495.0
1                9.0       111.0    5000.0        21           27  16500.0
2                9.0       154.0    5000.0        19           26  16500.0
3               10.0       102.0    5500.0        24           30  13950.0
4                8.0       115.0    5500.0        18           22  17450.0

   city-L/100km horsepower-binned  diesel  gas
0     11.190476            Medium       0    1
1     11.190476            Medium       0    1
2     12.368421            Medium       0    1
3      9.791667            Medium       0    1
4     13.055556            Medium       0    1

[5 rows x 30 columns]
```

First let's only use numeric data:

```
In [3]: df=df._get_numeric_data()
```

```
In [4]: df.head()
```

```
Out[4]:    Unnamed: 0  symboling  normalized-losses  wheel-base    length     width  \
        0           0          3                122        88.6  0.811148  0.890278
        1           1          3                122        88.6  0.811148  0.890278
        2           2          1                122        94.5  0.822681  0.909722
        3           3          2                164        99.8  0.848630  0.919444
        4           4          2                164        99.4  0.848630  0.922222

           height  curb-weight  engine-size  bore  stroke  compression-ratio  \
        0    48.8         2548          130  3.47    2.68                9.0
        1    48.8         2548          130  3.47    2.68                9.0
        2    52.4         2823          152  2.68    3.47                9.0
        3    54.3         2337          109  3.19    3.40               10.0
        4    54.3         2824          136  3.19    3.40                8.0

           horsepower  peak-rpm  city-mpg  highway-mpg    price  city-L/100km  diesel  \
        0       111.0    5000.0        21           27  13495.0     11.190476       0
        1       111.0    5000.0        21           27  16500.0     11.190476       0
        2       154.0    5000.0        19           26  16500.0     12.368421       0
        3       102.0    5500.0        24           30  13950.0      9.791667       0
        4       115.0    5500.0        18           22  17450.0     13.055556       0

           gas
```

```
0    1
1    1
2    1
3    1
4    1
```

Libraries for plotting:

```
In [5]: from IPython.display import display
        from IPython.html import widgets
        from IPython.display import display
        from ipywidgets import interact, interactive, fixed, interact_manual
        print("done")
```

```
done
```

```
/home/jupyterlab/conda/lib/python3.6/site-packages/IPython/html.py:14: ShimWarning: The `IPython
  "`IPython.html.widgets` has moved to `ipywidgets`.", ShimWarning)
```

2.1 Functions for plotting

```
In [6]: def DistributionPlot(RedFunction,BlueFunction,RedName,BlueName,Title ):
            width = 12
            height = 10
            plt.figure(figsize=(width, height))

            ax1 = sns.distplot(RedFunction, hist=False, color="r", label=RedName)
            ax2 = sns.distplot(BlueFunction, hist=False, color="b", label=BlueName, ax=ax1)

            plt.title(Title)
            plt.xlabel('Price (in dollars)')
            plt.ylabel('Proportion of Cars')

            plt.show()
            plt.close()
```

```
In [7]: def PollyPlot(xtrain,xtest,y_train,y_test,lr,poly_transform):
            width = 12
            height = 10
            plt.figure(figsize=(width, height))


            #training data
            #testing data
            # lr:  linear regression object
            #poly_transform:  polynomial transformation object
```

```
        xmax=max([xtrain.values.max(),xtest.values.max()])

        xmin=min([xtrain.values.min(),xtest.values.min()])

        x=np.arange(xmin,xmax,0.1)


        plt.plot(xtrain,y_train,'ro',label='Training Data')
        plt.plot(xtest,y_test,'go',label='Test Data')
        plt.plot(x,lr.predict(poly_transform.fit_transform(x.reshape(-1,1))),label='Predicte
        plt.ylim([-10000,60000])
        plt.ylabel('Price')
        plt.legend()
```

3 Part 1: Training and Testing

An important step in testing your model is to split your data into training and testing data. We will place the target data **price** in a separate dataframe **y**:

```
In [8]: y_data=df['price']
```

Drop price data in x data:

```
In [9]: x_data=df.drop('price',axis=1)
```

Now we randomly split our data into training and testing data using the function **train_test_split**:

```
In [10]: from sklearn.model_selection import train_test_split


         x_train, x_test, y_train, y_test = train_test_split(x_data, y_data, test_size=0.15, ran


         print("number of test samples :", x_test.shape[0])
         print("number of training samples:",x_train.shape[0])

number of test samples : 31
number of training samples: 170
```

The **test_size** parameter sets the proportion of data that is split into the testing set. In the above, the testing set is set to 10% of the total dataset.

Question #1:

Use the function "train_test_split" to split up the data set such that 40% of the data samples will be utilized for testing, and set the parameter "random_state" equal to zero. The output of the function should be the following: "x_train_1" , "x_test_1", "y_train_1" and "y_test_1":

```
In [11]: x_train_1, x_test_1, y_train_1, y_test_1 = train_test_split(x_data, y_data, test_size=0


         print("number of test samples :", x_test_1.shape[0])
         print("number of training samples:",x_train_1.shape[0])
```

```
number of test samples : 81
number of training samples: 120
```

Click here for the solution

```
x_train1, x_test1, y_train1, y_test1 = train_test_split(x_data, y_data, test_size=0.4, random_st
```

Let's import **LinearRegression** from the module **linear_model**:

```
In [12]: from sklearn.linear_model import LinearRegression
```

We create a Linear Regression object:

```
In [13]: lre=LinearRegression()
```

We fit the model using the feature 'horsepower':

```
In [14]: lre.fit(x_train[['horsepower']],y_train)
```

```
Out[14]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,
                  normalize=False)
```

Let's Calculate the R^2 on the test data:

```
In [15]: lre.score(x_test[['horsepower']],y_test)
```

```
Out[15]: 0.707688374146705
```

We can see the R^2 is much smaller using the test data:

```
In [16]: lre.score(x_train[['horsepower']],y_train)
```

```
Out[16]: 0.6449517437659684
```

Question #2:
Find the R^2 on the test data using 90% of the data for training data:

```
In [17]: #train_test_split at 90% test data
         x_train_1, x_test_1, y_train_1, y_test_1 = train_test_split(x_data, y_data, test_size=0

         #Fit model using horsepower
         lre.fit(x_train_1[['horsepower']],y_train_1)

         #Get the R^2 for test data
         lre.score(x_test_1[['horsepower']],y_test_1)
```

```
Out[17]: 0.6559543699796797
```

Click here for the solution

```
x_train1, x_test1, y_train1, y_test1 = train_test_split(x_data, y_data, test_size=0.9, random_st
lre.fit(x_train1[['horsepower']],y_train1)
lre.score(x_test1[['horsepower']],y_test1)
```

Sometimes you do not have sufficient testing data. As such, you may want to perform Cross-validation. Let's go over several methods that you can use for Cross-validation.

3.1 Cross-validation Score

Let's import **model_selection** from the module **cross_val_scor**:

```
In [18]: from sklearn.model_selection import cross_val_score
         print("done")

done
```

We input the object, the feature in this case ' horsepower', the target data (y_data). The parameter 'cv' determines the number of folds; in this case 4:

```
In [19]: Rcross=cross_val_score(lre,x_data[['horsepower']], y_data,cv=4)
```

The default scoring is R^2; each element in the array has the average R^2 value in the fold:

```
In [20]: Rcross

Out[20]: array([0.7746232 , 0.51716687, 0.74785353, 0.04839605])
```

We can calculate the average and standard deviation of our estimate:

```
In [21]: print("The mean of the folds are", Rcross.mean(),"and the standard deviation is" ,Rcros

The mean of the folds are 0.522009915042119 and the standard deviation is 0.2911839444756029
```

We can use negative squared error as a score by setting the parameter 'scoring' metric to 'neg_mean_squared_error':

```
In [22]: -1*cross_val_score(lre,x_data[['horsepower']], y_data,cv=4,scoring='neg_mean_squared_er

Out[22]: array([20254142.84026704, 43745493.26505169, 12539630.34014931,
                17561927.72247591])
```

Question #3:
Calculate the average R^2 using two folds, find the average R^2 for the second fold utilizing the horsepower as a feature :

```
In [23]: #Calculate the average R^2 using two folds (cv=2) utilizing the horsepower as a feature
         Rc=cross_val_score(lre,x_data[['horsepower']], y_data,cv=2)

         #find the average R^2 for the second fold, Rc[1]
         Rc[1]
```

```
Out[23]: 0.443196127755029
```

Click here for the solution

```
Rc=cross_val_score(lre,x_data[['horsepower']], y_data,cv=2)
Rc[1]
```

You can also use the function 'cross_val_predict' to predict the output. The function splits up the data into the specified number of folds, using one fold to get a prediction while the rest of the folds are used as test data. First import the function:

```
In [24]: from sklearn.model_selection import cross_val_predict
```

We input the object, the feature in this case **'horsepower'** , the target data **y_data**. The parameter 'cv' determines the number of folds, in this case 4. We can produce an output:

```
In [25]: yhat=cross_val_predict(lre,x_data[['horsepower']], y_data,cv=4)
         yhat[0:5]
```

```
Out[25]: array([14141.63807508, 14141.63807508, 20814.29423473, 12745.03562306,
                14762.35027598])
```

4 Part 2: Overfitting, Underfitting and Model Selection

It turns out that the test data sometimes referred to as the out of sample data is a much better measure of how well your model performs in the real world. One reason for this is overfitting; let's go over some examples. It turns out these differences are more apparent in Multiple Linear Regression and Polynomial Regression so we will explore overfitting in that context.

Let's create Multiple linear regression objects and train the model using **'horsepower'**, **'curb-weight'**, **'engine-size'** and **'highway-mpg'** as features:

```
In [26]: lr=LinearRegression()
         lr.fit(x_train[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']],y_train)
```

```
Out[26]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,
                  normalize=False)
```

Prediction using training data:

```
In [27]: yhat_train=lr.predict(x_train[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg
         yhat_train[0:5]
```

```
Out[27]: array([11927.70699817, 11236.71672034,  6436.91775515, 21890.22064982,
                16667.18254832])
```

Prediction using test data:

```
In [28]: yhat_test=lr.predict(x_test[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']
         yhat_test[0:5]

Out[28]: array([11349.16502418,  5914.48335385, 11243.76325987,  6662.03197043,
                15555.76936275])
```

Let's perform some model evaluation using our training and testing data separately. First we import the seaborn and matplotlibb library for plotting:

```
In [29]: import matplotlib.pyplot as plt
         %matplotlib inline
         import seaborn as sns
```

```
        ---------------------------------------------------------------------------

        ModuleNotFoundError                       Traceback (most recent call last)

        <ipython-input-29-e4cdd369eeb7> in <module>
          1 import matplotlib.pyplot as plt
          2 get_ipython().run_line_magic('matplotlib', 'inline')
    ----> 3 import seaborn as sns


        ModuleNotFoundError: No module named 'seaborn'
```

Let's examine the distribution of the predicted values of the training data:

```
In [ ]: Title='Distribution  Plot of  Predicted Value Using Training Data vs Training Data Distr
        DistributionPlot(y_train,yhat_train,"Actual Values (Train)","Predicted Values (Train)",T
```

Figure 1: Plot of predicted values using the training data compared to the training data.

So far the model seems to be doing well in learning from the training dataset. But what happens when the model encounters new data from the testing dataset? When the model generates new values from the test data, we see the distribution of the predicted values is much different from the actual target values.

```
In [ ]: Title='Distribution  Plot of  Predicted Value Using Test Data vs Data Distribution of Te
        DistributionPlot(y_test,yhat_test,"Actual Values (Test)","Predicted Values (Test)",Title
```

Figure 2: Plot of predicted value compared to the actual value using the test data.

Comparing Figure 1 and Figure 2, it is evident that the distribution of the test data in Figure 1 is much better at fitting the data. This difference in Figure 2 is apparent where the ranges are from 5000 to 15 000. This is where the distribution shape is exceptionally different. Let's see if polynomial regression also exhibits a drop in the prediction accuracy when analysing the test dataset.

```
In [ ]: from sklearn.preprocessing import PolynomialFeatures
        print("done")
```

Overfitting Overfitting occurs when the model fits the noise, not the underlying process. Therefore when testing your model using the test-set, your model does not perform as well as it is modelling noise, not the underlying process that generated the relationship. Let's create a degree 5 polynomial model.

Let's use 55 percent of the data for testing and the rest for training:

```
In [ ]: x_train, x_test, y_train, y_test = train_test_split(x_data, y_data, test_size=0.45, rand
        print("done")
```

We will perform a degree 5 polynomial transformation on the feature **'horse power'**:

```
In [ ]: pr=PolynomialFeatures(degree=5)
        x_train_pr=pr.fit_transform(x_train[['horsepower']])
        x_test_pr=pr.fit_transform(x_test[['horsepower']])
        pr
```

Now let's create a linear regression model "poly" and train it:

```
In [ ]: poly=LinearRegression()
        poly.fit(x_train_pr,y_train)
```

We can see the output of our model using the method "predict", then assign the values to "yhat":

```
In [ ]: yhat=poly.predict(x_test_pr )
        yhat[0:5]
```

Let's take the first five predicted values and compare it to the actual targets:

```
In [ ]: print("Predicted values:", yhat[0:4])
        print("True values:",y_test[0:4].values)
```

We will use the function "PollyPlot" that we defined at the beginning of the lab to display the training data, testing data, and the predicted function:

```
In [ ]: PollyPlot(x_train[['horsepower']],x_test[['horsepower']],y_train,y_test,poly,pr)
```

Figure 4: A polynomial regression model. Red dots represent training data, green dots represent test data, and the blue line represents the model prediction.

We see that the estimated function appears to track the data but at around 200 horsepower, the function begins to diverge from the data points.

R^2 of the training data:

```
In [ ]: poly.score(x_train_pr, y_train)
```

R^2 of the test data:

```
In [ ]: poly.score(x_test_pr, y_test)
```

We see the R^2 for the training data is 0.5567 while the R^2 on the test data was -29.87. The lower the R^2, the worse the model; a Negative R^2 is a sign of overfitting.

Let's see how the R^2 changes on the test data for different order polynomials and plot the results:

```
In [ ]: Rsqu_test=[]

        order=[1,2,3,4]
        for n in order:
            pr=PolynomialFeatures(degree=n)

            x_train_pr=pr.fit_transform(x_train[['horsepower']])

            x_test_pr=pr.fit_transform(x_test[['horsepower']])

            lr.fit(x_train_pr,y_train)

            Rsqu_test.append(lr.score(x_test_pr,y_test))

        plt.plot(order,Rsqu_test)
        plt.xlabel('order')
        plt.ylabel('R^2')
        plt.title('R^2 Using Test Data')
        plt.text(3, 0.75, 'Maximum R^2 ')
```

We see the R^2 gradually increases until an order three polynomial is used. Then the R^2 dramatically decreases at four.

The following function will be used in the next section. Please run the cell.

```
In [ ]: def f(order,test_data):
            x_train, x_test, y_train, y_test = train_test_split(x_data, y_data, test_size=test_d
            pr=PolynomialFeatures(degree=order)
            x_train_pr=pr.fit_transform(x_train[['horsepower']])
            x_test_pr=pr.fit_transform(x_test[['horsepower']])
            poly=LinearRegression()
            poly.fit(x_train_pr,y_train)
            PollyPlot(x_train[['horsepower']],x_test[['horsepower']],y_train,y_test,poly,pr)
```

The following interface allows you to experiment with different polynomial orders and different amounts of data.

```
In [ ]: interact(f, order=(0,6,1),test_data=(0.05,0.95,0.05))
```

Question #4(a):

We can perform polynomial transformations with more than one feature. Create a "PolynomialFeatures" object "pr1" of degree two:

Click here for the solution

```
pr1=PolynomialFeatures(degree=2)
```

Question #4(b):

Transform the training and testing samples for the features 'horsepower', 'curb-weight', 'engine-size' and 'highway-mpg'. Hint: use the method "fit_transform":

Click here for the solution

```
x_train_pr1=pr.fit_transform(x_train[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']
x_test_pr1=pr.fit_transform(x_test[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']])
```

Question #4(c):

How many dimensions does the new feature have? Hint: use the attribute "shape":

Click here for the solution

```
There are now 15 features: x_train_pr1.shape
```

Question #4(d):

Create a linear regression model "poly1" and train the object using the method "fit" using the polynomial features:

Click here for the solution

```
poly1=linear_model.LinearRegression().fit(x_train_pr1,y_train)
```

Question #4e):

Use the method "predict" to predict an output on the polynomial features, then use the function "DistributionPlot" to display the distribution of the predicted output vs the test data:

Click here for the solution

```
yhat_test1=poly1.predict(x_train_pr1)
Title='Distribution  Plot of  Predicted Value Using Test Data vs Data Distribution of Test Data'
DistributionPlot(y_test,yhat_test1,"Actual Values (Test)","Predicted Values (Test)",Title)
```

Question #4(f):

Use the distribution plot to determine the two regions were the predicted prices are less accurate than the actual prices:

Click here for the solution

```
The predicted value is lower than actual value for cars where the price  $ 10,000 range. Convers
```

4.1 Part 3: Ridge Regression

In this section, we will review Ridge Regression. We will see how the parameter Alfa changes the model. Just a note here, our test data will be used as validation data.

Let's perform a degree two polynomial transformation on our data:

```
In [ ]: pr=PolynomialFeatures(degree=2)
        x_train_pr=pr.fit_transform(x_train[['horsepower', 'curb-weight', 'engine-size', 'highwa
        x_test_pr=pr.fit_transform(x_test[['horsepower', 'curb-weight', 'engine-size', 'highway-
```

Let's import **Ridge** from the module **linear models**:

```
In [ ]: from sklearn.linear_model import Ridge
```

Let's create a Ridge regression object, setting the regularization parameter to 0.1:

```
In [ ]: RigeModel=Ridge(alpha=0.1)
```

Like regular regression, you can fit the model using the method **fit**:

```
In [ ]: RigeModel.fit(x_train_pr,y_train)
```

Similarly, you can obtain a prediction:

```
In [ ]: yhat=RigeModel.predict(x_test_pr)
```

Let's compare the first five predicted samples to our test set:

```
In [ ]: print('predicted:', yhat[0:4])
        print('test set :', y_test[0:4].values)
```

We select the value of Alfa that minimizes the test error. For example, we can use a for loop:

```
In [ ]: Rsqu_test=[]
        Rsqu_train=[]
        dummy1=[]
        ALFA=5000*np.array(range(0,10000))
        for alfa in ALFA:
            RigeModel=Ridge(alpha=alfa)
            RigeModel.fit(x_train_pr,y_train)
            Rsqu_test.append(RigeModel.score(x_test_pr,y_test))
            Rsqu_train.append(RigeModel.score(x_train_pr,y_train))
```

We can plot out the value of Rˆ2 for different Alphas:

```
In [ ]: width = 12
        height = 10
        plt.figure(figsize=(width, height))

        plt.plot(ALFA,Rsqu_test,label='validation data  ')
        plt.plot(ALFA,Rsqu_train,'r',label='training Data ')
        plt.xlabel('alpha')
        plt.ylabel('R^2')
        plt.legend()
```

Figure 6: The blue line represents the Rˆ2 of the test data, and the red line represents the Rˆ2 of the training data. The x-axis represents the different values of Alfa.

The red line in Figure 6 represents the Rˆ2 of the test data; as Alpha increases the Rˆ2 decreases. Therefore, as Alfa increases, the model performs worse on the test data. The blue line represents the Rˆ2 on the validation data, as the value for Alfa increases the Rˆ2 decreases.

Question #5:

Perform Ridge regression and calculate the Rˆ2 using the polynomial features. Use the training data to train the model and test data to test the model. The parameter alpha should be set to 10:

```
In [ ]: RigeModel=Ridge(alpha=10)
        RigeModel.fit(x_train_pr,y_train)
        RigeModel.score(x_test_pr, y_test)
```

Click here for the solution

```
RigeModel=Ridge(alpha=0)
RigeModel.fit(x_train_pr,y_train)
RigeModel.score(x_test_pr, y_test)
```

4.2 Part 4: Grid Search

The term Alfa is a hyperparameter. Sklearn has the class **GridSearchCV** to make the process of finding the best hyperparameter simpler.

Let's import **GridSearchCV** from the module **model_selection**:

```
In [ ]: from sklearn.model_selection import GridSearchCV
        print("done")
```

We create a dictionary of parameter values:

```
In [ ]: parameters1= [{'alpha': [0.001,0.1,1, 10, 100, 1000,10000,100000,100000]}]
        parameters1
```

Create a ridge regions object:

```
In [ ]: RR=Ridge()
        RR
```

Create a ridge grid search object:

```
In [ ]: Grid1 = GridSearchCV(RR, parameters1,cv=4)
```

Fit the model:

```
In [ ]: Grid1.fit(x_data[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']],y_data)
```

The object finds the best parameter values on the validation data. We can obtain the estimator with the best parameters and assign it to the variable BestRR as follows:

```
In [ ]: BestRR=Grid1.best_estimator_
        BestRR
```

We now test our model on the test data:

```
In [ ]: BestRR.score(x_test[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']],y_test)
```

Question #6:

Perform a grid search for the alpha parameter and the normalization parameter, then find the best values of the parameters:

```
In [ ]: parameters2= [{'alpha': [0.001,0.1,1, 10, 100, 1000,10000,100000,100000],'normalize':[Tr

        Grid2 = GridSearchCV(Ridge(), parameters2,cv=4)

        Grid2.fit(x_data[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']],y_data)

        Grid2.best_estimator_
```

Click here for the solution

```
parameters2= [{'alpha': [0.001,0.1,1, 10, 100, 1000,10000,100000,100000],'normalize':[True,False
Grid2 = GridSearchCV(Ridge(), parameters2,cv=4)
Grid2.fit(x_data[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']],y_data)
Grid2.best_estimator
```

5 About the Authors:

This notebook written Joseph Santarcangelo PhD

Link