Regression Model Validation

Introduction

Previously you've briefly touched upon model evaluation when using a multiple linear regression model for prediction. In this section you'll learn why it's important to split your data in a train and a test set if you want to do proper performance evaluation.

Objectives ¶

You will be able to:

- Test a trained model with unseen data and calculate the RMSE
- Split the data in Pandas and Scikit-Learn using train-test-split
- Understand the rationale behind a train-test-split
- Understand the need for validation testing in predictive analysis

The need for train-test-split

Making predictions and evaluation

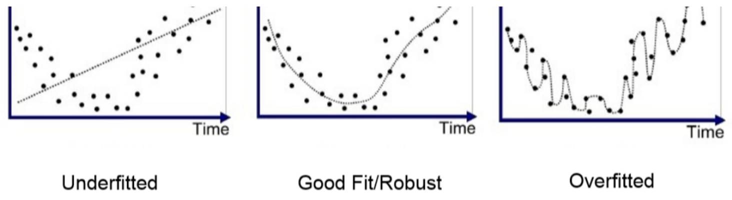
So far we've simply been fitting models to data, and evaluated our models calculating the errors between our $\hat{y}$ and our actual targets y, while these targets y contributed in fitting the model.

The reason why we built the model in the first place, however, is because we want to predict the outcome for observations that are not necessarily in our data set now. Eg: we want to predict mile per gallon for a new car that isn't part of our data set, or for a new house in Boston.

In order to get a good sense of how well your model will be doing on new instances, you'll have to perform a so-called "train-test-split". What you'll be doing here, is take a sample of the data that serves as input to "train" our model - fit a linear regression and compute the parameter estimates for our variables, and calculate how well our predictive performance is doing comparing the actual targets y and the fitted $\hat{y}$ obtained by our model.

Underfitting and overfitting

Another reason to use train-test-split is because of common problem which doesn't only affect linear model, but nearly all (other) machine learning algorithms: overfitting and underfitting. An overfit model is not generalizable and will not hold to future cases. An underfit model does not make full use of the information available and produces weaker predictions then is feasible. The following image gives a nice, more general demonstration:



How to evaluate?

It is pretty straightforward that, to evaluate our model, you'll want to compare your predicted values, $\hat{y}$ with the actual value, y. The difference between the two values is referred to as the residuals. When using a train test split, you'll compare your residuals for both test set and training set:

$r_{i,train} = y_{i,train} - \hat{y}_{i,train}$

$r_{i,test} = y_{i,test} - \hat{y}_{i,test}$

To get a summarized measure over all the instances in the test set and training set, a popular metric is the (Root) Mean Squared Error:

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

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

Again, you can compute these for both the traing and the test set. A big difference in value between the test and training set (R)MSE is an indication of overfitting.

Applying this to our auto-mpg data

Let's copy our pre-processed auto-mpg data again

In [30]:
```
import pandas as pd
import numpy as np
data = pd.read_csv("auto-mpg.csv")
data['horsepower'].astype(str).astype(int)

acc = data["acceleration"]
logdisp = np.log(data["displacement"])
loghorse = np.log(data["horsepower"])
logweight= np.log(data["weight"])

scaled_acc = (acc-min(acc))/(max(acc)-min(acc))
scaled_disp = (logdisp-np.mean(logdisp))/np.sqrt(np.var(logdisp))
scaled_horse = (loghorse-np.mean(loghorse))/(max(loghorse)-min(log
scaled_weight= (logweight-np.mean(logweight))/np.sqrt(np.var(logwe

data_fin = pd.DataFrame([])
data_fin["acc"]= scaled_acc
data_fin["disp"]= scaled_disp
data_fin["horse"] = scaled_horse
data_fin["weight"] = scaled_weight
cyl_dummies = pd.get_dummies(data["cylinders"], prefix="cyl")
yr_dummies = pd.get_dummies(data["model year"], prefix="yr")
orig_dummies = pd.get_dummies(data["origin"], prefix="orig")
mpg = data["mpg"]
data_fin = pd.concat([mpg, data_fin, cyl_dummies, yr_dummies, orig
```

In [31]:
```
data = pd.concat([mpg, scaled_acc, scaled_weight, orig_dummies], a
y = data[["mpg"]]
X = data.drop(["mpg"], axis=1)
```

Scikit-learn has a very useful function `train-test-split`. The optional argument `test_size` makes it possible to choose the size of he test set and the training set.

```
In [32]: from sklearn.model_selection import train_test_split
         X_train, X_test, y_train, y_test = train_test_split(X, y, test_siz
```

```
In [33]: print(len(X_train), len(X_test), len(y_train), len(y_test))
```

```
313 79 313 79
```

```
In [34]: from sklearn.linear_model import LinearRegression
         linreg = LinearRegression()
         linreg.fit(X_train, y_train)

         y_hat_train = linreg.predict(X_train)
         y_hat_test = linreg.predict(X_test)
```

Look at the residuals and the

```
In [35]: train_residuals = y_hat_train - y_train
         test_residuals = y_hat_test - y_test
```

```
In [36]: mse_train = np.sum((y_train-y_hat_train)**2)/len(y_train)
         mse_test =np.sum((y_test-y_hat_test)**2)/len(y_test)
         print('Train Mean Squarred Error:', mse_train)
         print('Test Mean Squarred Error:', mse_test)
```

```
Train Mean Squarred Error: mpg    16.223245
dtype: float64
Test Mean Squarred Error: mpg    18.395887
dtype: float64
```

Sklearn also has a very function `mean_squared_error`.

```
In [37]: from sklearn.metrics import mean_squared_error

         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: 16.22324478035774
Test Mean Squarred Error: 18.395887468660337
```

Great, there does not seem to be a big difference between the train and test MSE!

Additional resources

Great! By now, you have a lot of ingredients to build a pretty good (multiple) linear regression model. We'll add one more concept in the next section: the idea of cross-validation. But first, we strongly recommend to have a look at this blogpost to get a refresher on a lot of the concepts learned!

Summary

Great! Now let's put this into practice on our Boston Housing Data!