index

January 26, 2022

1 Regression Model Validation

1.1 Introduction

Previously you've briefly touched upon model evaluation when using a multiple linear regression model for prediction. In this lesson 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.

1.2 Objectives

You will: * Compare training and testing errors to determine if model is over or underfitting

1.3 The need for train-test split

1.3.1 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 dataset now; e.g: we want to predict miles per gallon for a new car that isn't part of our dataset, 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.

1.3.2 Underfitting and overfitting

Another reason to use train-test-split is because of a common problem which doesn't only affect linear models, 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 than is feasible. The following image gives a nice, more general demonstration:

1.3.3 Mechanics of train-test split

When performing a train-test-split, it is important that the data is **randomly** split. At some point, you will encounter datasets that have certain characteristics that are only present in certain segments of the data. For example, if you were looking at sales data for a website, you might expect the data to look different on days that promotional deals were held versus days that deals were not held. If we don't randomly split the data, there is a chance we might overfit to the characteristics of certain segments of data.

Another thing to consider is just how big each training and testing set should be. There is no hard and fast rule for deciding the correct size, but the range of training set is usually anywhere from 66% - 80% (and testing set between 33% and 20%). Some types of machine learning models need a substantial amount of data to train on, and as such, the training sets should be larger. Some models with many different tuning parameters will need to be validated with larger sets (the test size should be larger) to determine what the optimal parameters should be. When in doubt, just stick with training set sizes around 70% and test set sizes around 30%.

1.4 How to evaluate?

It is pretty straightforward that, to evaluate the 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:

$$r_{i,train} = y_{i,train} - \hat{y}_{i,train} \tag{1}$$

$$r_{i,test} = y_{i,test} - \hat{y}_{i,test} \tag{2}$$

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:

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2} \tag{3}$$

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{4}$$

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.

1.5 Applying this to our auto-mpg data

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

[3]:
```python
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'])
```

```python
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(loghorse))
scaled_weight = (logweight-np.mean(logweight))/np.sqrt(np.var(logweight))

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', drop_first=True)
yr_dummies = pd.get_dummies(data['model year'], prefix='yr', drop_first=True)
orig_dummies = pd.get_dummies(data['origin'], prefix='orig', drop_first=True)
mpg = data['mpg']
data_fin = pd.concat([mpg, data_fin, cyl_dummies, yr_dummies, orig_dummies],
                     axis=1)
```

[4]:
```python
data = pd.concat([mpg, scaled_acc, scaled_weight, orig_dummies], axis=1)
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 the test set and the training set. Since the observations are randomly assigned to the training and test splits each time you run `train_test_split()`, you can also pass an optional `random_state` argument to obtain reproducible results. This will ensure your training and test sets are always the same.

[5]:
```python
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
                                                    random_state=42)
```

[6]:
```python
print(len(X_train), len(X_test), len(y_train), len(y_test))
```

```
313 79 313 79
```

[7]:
```python
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 calculate the MSE for training and test sets:

[8]:
```python
train_residuals = y_hat_train - y_train
test_residuals = y_hat_test - y_test
```

[9]:
```python
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    16.790262
dtype: float64
Test Mean Squarred Error: mpg    16.500021
dtype: float64
```

You can also do this directly using sklearn's `mean_squared_error()` function:

[10]:
```python
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: 16.79026189951861
Test Mean Squarred Error: 16.500020627881508
```

Great, there does not seem to be a big difference between the train and test MSE! Interestingly, the test set error is smaller than the training set error. This is fairly rare but does occasionally happen.

1.6 Additional resources

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

1.7 Summary

In this lesson, you learned the importance of the train-test split approach and were introduced to one of the most popular metrics for evaluating regression models, (R)MSE. You also saw how to use the `train_test_split()` function from `sklearn` to split your data into training and test sets.