Regression Model Validation

Introduction

Previously you've evaluated a multiple linear regression model by calculating metrics based on the fit of the training data. 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 evaluate a model used for prediction.

Objectives

You will be able to:

- Perform a train-test split
- Prepare training and testing data for modeling
- Compare training and testing errors to determine if model is over or underfitting

Model Evaluation

Recall some ways that we can evaluate linear regression models.

Residuals

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**:

$r_i = y_i - \hat{y}_i$

To get a summarized measure over all the instances, 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 = $\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$

Larger (R)MSE indicates a *worse* model fit.

The Need for Train-Test Split

Making Predictions and Evaluation

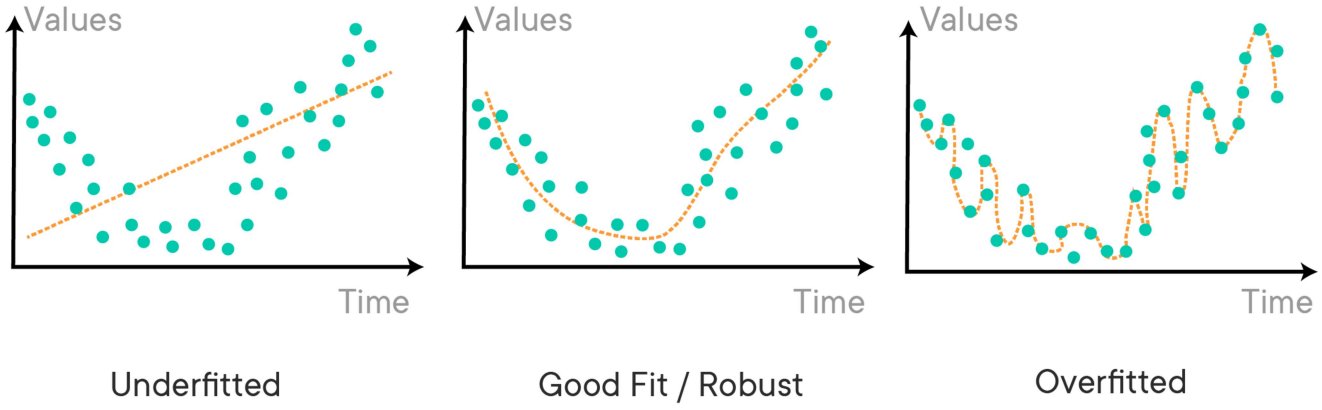
So far we've simply been fitting models to data, and evaluated our models calculating the errors between our $\hat{y}$ and our actual targets y, while these targets y contributed in fitting the model.

Let's say 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 predict the price for a new house in Ames.

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 taking 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 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:



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%.

Train-Test Split with Scikit-Learn

You could write your own pandas code to shuffle and split your data, but we'll use the convenient `train_test_split` function from scikit-learn instead. We'll also use the Auto MPG dataset.

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

data = pd.read_csv('auto-mpg.csv')
data
```

Out[1]:

	mpg	cylinders	displacement	horsepower	weight	acceleration	model year	origin	car name
0	18.0	8	307.0	130	3504	12.0	70	1	chevrolet chevelle malibu
1	15.0	8	350.0	165	3693	11.5	70	1	buick skylark 320
2	18.0	8	318.0	150	3436	11.0	70	1	plymouth satellite
3	16.0	8	304.0	150	3433	12.0	70	1	amc rebel sst
4	17.0	8	302.0	140	3449	10.5	70	1	ford torino
...	...	...	...	...	...	...	...	...	...
387	27.0	4	140.0	86	2790	15.6	82	1	ford mustang gl
388	44.0	4	97.0	52	2130	24.6	82	2	vw pickup
389	32.0	4	135.0	84	2295	11.6	82	1	dodge rampage
390	28.0	4	120.0	79	2625	18.6	82	1	ford ranger
391	31.0	4	119.0	82	2720	19.4	82	1	chevy s-10

392 rows × 9 columns

The `train_test_split` function (documentation here (https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html)) takes in a series of array-like variables, as well as some optional arguments. It returns multiple arrays.

For example, this would be a valid way to use `train_test_split`:

In [2]:
```
from sklearn.model_selection import train_test_split

train, test = train_test_split(data)
```

In [3]:
```
train
```

Out[3]:

	mpg	cylinders	displacement	horsepower	weight	acceleration	model year	origin	car name
245	39.4	4	85.0	70	2070	18.6	78	3	datsun b210 gx
101	26.0	4	97.0	46	1950	21.0	73	2	volkswagen super beetle
198	18.0	6	250.0	78	3574	21.0	76	1	ford granada ghia
263	17.5	8	318.0	140	4080	13.7	78	1	dodge magnum xe
12	15.0	8	400.0	150	3761	9.5	70	1	chevrolet monte carlo
...	...	...	...	...	...	...	...	...	...
129	32.0	4	71.0	65	1836	21.0	74	3	toyota corolla 1200
132	16.0	6	258.0	110	3632	18.0	74	1	amc matador
360	20.2	6	200.0	88	3060	17.1	81	1	ford granada gl
7	14.0	8	440.0	215	4312	8.5	70	1	plymouth fury iii
87	14.0	8	302.0	137	4042	14.5	73	1	ford gran torino

294 rows × 9 columns

In [4]:
```
test
```

Out[4]:

	mpg	cylinders	displacement	horsepower	weight	acceleration	model year	origin	car name
298	23.9	8	260.0	90	3420	22.2	79	1	oldsmobile cutlass salon brougham
63	15.0	8	318.0	150	4135	13.5	72	1	plymouth fury iii
106	18.0	6	232.0	100	2789	15.0	73	1	amc gremlin
0	18.0	8	307.0	130	3504	12.0	70	1	chevrolet chevelle malibu
337	23.5	6	173.0	110	2725	12.6	81	1	chevrolet citation
...	...	...	...	...	...	...	...	...	...
320	46.6	4	86.0	65	2110	17.9	80	3	mazda glc
216	36.0	4	79.0	58	1825	18.6	77	2	renault 5 gtl
199	18.5	6	250.0	110	3645	16.2	76	1	pontiac ventura sj
152	15.0	6	250.0	72	3432	21.0	75	1	mercury monarch
67	13.0	8	350.0	155	4502	13.5	72	1	buick lesabre custom

98 rows × 9 columns

In this case, the DataFrame `data` was split into two DataFrames called `train` and `test`. `train` has 294 values (75% of the full dataset) and `test` has 98 values (25% of the full dataset). Note the randomized order of the index values on the left.

However you can also pass multiple array-like variables into `train_test_split` at once. For each variable that you pass in, you will get a train and a test copy back out.

Most commonly in this curriculum these are the inputs and outputs:

Inputs

- `X`
- `y`

Outputs

- `X_train`
- `X_test`
- `y_train`
- `y_test`

In [5]:
```
y = data[['mpg']]
X = data.drop(['mpg', 'car name'], axis=1)

X_train, X_test, y_train, y_test = train_test_split(X, y)
```

In [6]:
```
X_train
```

Out[6]:

	cylinders	displacement	horsepower	weight	acceleration	model year	origin
244	4	78.0	52	1985	19.4	78	3
188	8	351.0	152	4215	12.8	76	1
195	4	90.0	70	1937	14.2	76	2
382	4	156.0	92	2585	14.5	82	1
254	6	225.0	100	3430	17.2	78	1
...	...	...	...	...	...	...	...
208	6	156.0	108	2930	15.5	76	3
282	6	225.0	110	3360	16.6	79	1
211	8	350.0	145	4055	12.0	76	1
379	4	91.0	67	1995	16.2	82	3
112	6	155.0	107	2472	14.0	73	1

294 rows × 7 columns

In [7]:
```
y_train
```

Out[7]:

	mpg
244	32.8
188	14.5
195	29.0
382	26.0
254	20.5
...	...
208	19.0
282	20.6
211	13.0
379	38.0
112	21.0

294 rows × 1 columns

We can view the lengths of the results like this:

In [8]:
```
print(len(X_train), len(X_test), len(y_train), len(y_test))
```

```
294 98 294 98
```

However it is not recommended to pass in just the data to be split. This is because the randomization of the split means that you will get different results for `X_train` etc. every time you run the code. **For reproducibility, it is always recommended that you specify a `random_state`**, such as in this example:

In [9]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)
```

Another optional argument is `test_size`, which makes it possible to choose the size of the test set and the training set instead of using the default 75% train/25% test proportions.

In [10]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Note that the lengths of the resulting datasets will be different:

In [11]:
```
print(len(X_train), len(X_test), len(y_train), len(y_test))
```

```
313 79 313 79
```

Preparing Data for Modeling

When using a train-test split, data preparation should happen *after* the split. This is to avoid ***data leakage***. The general idea is that the treatment of the test data should be as similar as possible to how genuinely unknown data should be treated. And genuinely unknown data would not have been there at the time of fitting the scikit-learn transformers, just like it would not have been there at the time of fitting the model!

In some cases you will see all of the data being prepared together for expediency, but the best practice is to prepare it separately.

Log Transformation

In [12]:
```
from sklearn.preprocessing import FunctionTransformer
import numpy as np

# Instantiate a custom transformer for log transformation
log_transformer = FunctionTransformer(np.log, validate=True)

# Columns to be log transformed
log_columns = ['displacement', 'horsepower', 'weight']

# New names for columns after transformation
new_log_columns = ['log_disp', 'log_hp', 'log_wt']

# Log transform the training columns and convert them into a DataFrame
X_train_log = pd.DataFrame(log_transformer.fit_transform(X_train[log_columns]),
                           columns=new_log_columns, index=X_train.index)

# Replace training columns with transformed versions
X_train = pd.concat([X_train.drop(log_columns, axis=1), X_train_log], axis=1)
X_train
```

Out[12]:

	cylinders	acceleration	model year	origin	log_disp	log_hp	log_wt
258	6	18.7	78	1	5.416100	4.700480	8.194229
182	4	14.9	76	1	4.941642	4.521789	7.852439
172	6	14.5	75	1	5.141664	4.574711	8.001020
63	8	13.5	72	1	5.762051	5.010635	8.327243
340	4	16.4	81	1	4.454347	4.158883	7.536364
...	...	...	...	...	...	...	...
71	8	12.5	72	1	5.717028	5.010635	8.266678
106	6	15.0	73	1	5.446737	4.605170	7.933438
270	4	17.6	78	1	5.017280	4.442651	7.956827
348	4	20.7	81	1	4.584967	4.174387	7.774856
102	8	14.0	73	1	5.991465	5.010635	8.516593

313 rows × 7 columns

In [13]:
```
# Log transform the test columns and convert them into a DataFrame
X_test_log = pd.DataFrame(log_transformer.transform(X_test[log_columns]),
                          columns=new_log_columns, index=X_test.index)

# Replace testing columns with transformed versions
X_test = pd.concat([X_test.drop(log_columns, axis=1), X_test_log], axis=1)
X_test
```

Out[13]:

	cylinders	acceleration	model year	origin	log_disp	log_hp	log_wt
78	4	18.0	72	2	4.564348	4.234107	7.691200
274	4	15.7	78	2	4.795791	4.744932	7.935587
246	4	16.4	78	3	4.510860	4.094345	7.495542
55	4	20.5	71	1	4.510860	4.248495	7.578145
387	4	15.6	82	1	4.941642	4.454347	7.933797
...	...	...	...	...	...	...	...
361	6	16.6	81	1	5.416100	4.442651	8.150468
82	4	15.0	72	1	4.584967	4.382027	7.679714
114	8	13.0	73	1	5.857933	4.976734	8.314342
3	8	12.0	70	1	5.717028	5.010635	8.141190
18	4	14.5	70	3	4.574711	4.477337	7.663877

79 rows × 7 columns

One-Hot Encoding

In [14]:
```
from sklearn.preprocessing import OneHotEncoder

# Instantiate OneHotEncoder
ohe = OneHotEncoder(handle_unknown='ignore', sparse=False)

# Categorical columns
cat_columns = ['cylinders', 'model year', 'origin']

# Fit encoder on training set
ohe.fit(X_train[cat_columns])

# Get new column names
new_cat_columns = ohe.get_feature_names(input_features=cat_columns)

# Transform training set
X_train_ohe = pd.DataFrame(ohe.fit_transform(X_train[cat_columns]),
                           columns=new_cat_columns, index=X_train.index)

# Replace training columns with transformed versions
X_train = pd.concat([X_train.drop(cat_columns, axis=1), X_train_ohe], axis=1)
X_train
```

Out[14]:

	acceleration	log_disp	log_hp	log_wt	cylinders_3	cylinders_4	cylinders_5	cylinders_6	cylinders_8	model year_70	...	model year_76	model year_77	model year_78	model year_79
258	18.7	5.416100	4.700480	8.194229	0.0	0.0	0.0	1.0	0.0	0.0	...	0.0	0.0	1.0	0.0
182	14.9	4.941642	4.521789	7.852439	0.0	1.0	0.0	0.0	0.0	0.0	...	1.0	0.0	0.0	0.0
172	14.5	5.141664	4.574711	8.001020	0.0	0.0	0.0	1.0	0.0	0.0	...	0.0	0.0	0.0	0.0
63	13.5	5.762051	5.010635	8.327243	0.0	0.0	0.0	0.0	1.0	0.0	...	0.0	0.0	0.0	0.0
340	16.4	4.454347	4.158883	7.536364	0.0	1.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
71	12.5	5.717028	5.010635	8.266678	0.0	0.0	0.0	0.0	1.0	0.0	...	0.0	0.0	0.0	0.0
106	15.0	5.446737	4.605170	7.933438	0.0	0.0	0.0	1.0	0.0	0.0	...	0.0	0.0	0.0	0.0
270	17.6	5.017280	4.442651	7.956827	0.0	1.0	0.0	0.0	0.0	0.0	...	0.0	0.0	1.0	0.0
348	20.7	4.584967	4.174387	7.774856	0.0	1.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0
102	14.0	5.991465	5.010635	8.516593	0.0	0.0	0.0	0.0	1.0	0.0	...	0.0	0.0	0.0	0.0

313 rows × 25 columns

In [15]:
```
# Transform testing set
X_test_ohe = pd.DataFrame(ohe.transform(X_test[cat_columns]),
                          columns=new_cat_columns, index=X_test.index)

# Replace testing columns with transformed versions
X_test = pd.concat([X_test.drop(cat_columns, axis=1), X_test_ohe], axis=1)
X_test
```

Out[15]:

	acceleration	log_disp	log_hp	log_wt	cylinders_3	cylinders_4	cylinders_5	cylinders_6	cylinders_8	model year_70	...	model year_76	model year_77	model year_78	model year_79
78	18.0	4.564348	4.234107	7.691200	0.0	1.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0
274	15.7	4.795791	4.744932	7.935587	0.0	1.0	0.0	0.0	0.0	0.0	...	0.0	0.0	1.0	0.0
246	16.4	4.510860	4.094345	7.495542	0.0	1.0	0.0	0.0	0.0	0.0	...	0.0	0.0	1.0	0.0
55	20.5	4.510860	4.248495	7.578145	0.0	1.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0
387	15.6	4.941642	4.454347	7.933797	0.0	1.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
361	16.6	5.416100	4.442651	8.150468	0.0	0.0	0.0	1.0	0.0	0.0	...	0.0	0.0	0.0	0.0
82	15.0	4.584967	4.382027	7.679714	0.0	1.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0
114	13.0	5.857933	4.976734	8.314342	0.0	0.0	0.0	0.0	1.0	0.0	...	0.0	0.0	0.0	0.0
3	12.0	5.717028	5.010635	8.141190	0.0	0.0	0.0	0.0	1.0	1.0	...	0.0	0.0	0.0	0.0
18	14.5	4.574711	4.477337	7.663877	0.0	1.0	0.0	0.0	0.0	1.0	...	0.0	0.0	0.0	0.0

79 rows × 25 columns

Building, Evaluating, and Validating a Model

Great, now that you have preprocessed all the columns, you can fit a linear regression model:

In [16]:
```
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:

In [17]:
```
train_residuals = y_hat_train - y_train
test_residuals = y_hat_test - y_test
```

In [18]:
```
mse_train = float(np.sum((y_train - y_hat_train)**2)/len(y_train))
mse_test = float(np.sum((y_test - y_hat_test)**2)/len(y_test))
print('Train Mean Squared Error:', mse_train)
print('Test Mean Squared Error:', mse_test)
```

```
Train Mean Squared Error: 6.689589737676635
Test Mean Squared Error: 5.8967566665747615
```

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

In [19]:
```
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 Squared Error:', train_mse)
print('Test Mean Squared Error:', test_mse)
```

```
Train Mean Squared Error: 6.689589737676635
Test Mean Squared Error: 5.8967566665747615
```

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.

In other words, our validation process has indicated that we are **not** overfitting. In fact, we may be *underfitting* because linear regression is not a very complex model.

Overfitting with a Different Model

Just for the sake of example, here is a model that is overfit to the data. Don't worry about the model algorithm being shown! Instead, just look at the MSE for the train vs. test set, using the same preprocessed data:

In [20]:
```
from sklearn.tree import DecisionTreeRegressor

other_model = DecisionTreeRegressor(random_state=42)
other_model.fit(X_train, y_train)

other_train_mse = mean_squared_error(y_train, other_model.predict(X_train))
other_test_mse = mean_squared_error(y_test, other_model.predict(X_test))
print('Train Mean Squared Error:', other_train_mse)
print('Test Mean Squared Error:', other_test_mse)
```

```
Train Mean Squared Error: 0.0
Test Mean Squared Error: 14.577215189873417
```

This model initially seems great...0 MSE for the training data! But then you see that it is performing much worse than our linear regression model on the test data. This model **is** overfitting.

Additional Resources

This blog post (https://towardsdatascience.com/linear-regression-in-python-9a1f5f000606) shows a walkthrough of the key steps for model validation with train-test split and scikit-learn.

Summary

In this lesson, you learned the importance of the train-test split approach and used 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, and then evaluated whether models were overfitting using metrics on those training and test sets.