

Week 5 - Model Evaluation and Refinement

Learning Objectives

- Describe data model refinement techniques
- Explain overfitting, underfitting and model selection
- Apply ridge regression to regularize and reduce the standard errors to avoid overfitting a regression model
- Apply grid search techniques to Python data

1. Model Evaluation and Refinement

Model Evaluation



- In-sample evaluation tells us how well our model will fit the data used to train it
- Problem?
 - It does not tell us how well the trained model can be used to predict new data
- Solution?
 - In- sample data or training data
 - Out-of-sample evaluation or test set

Model evaluation tells us how our model performs in the real world. In the previous module, we talked about in-sample evaluation. In-sample evaluation tells us how well our model fits the data already given to train it. It does not give us an estimate of how well the train model can predict new data. The solution is to split our data up, use the in-sample data or training data to train the model. The rest of the data, called Test Data, is used as out-of-sample data. This data is then used to approximate, how the model performs in the real world.

Separating data into training and testing sets is an important part of model evaluation. We use the test data to get an idea how our model will perform in the real world. When we split a dataset, usually the larger portion of data is used for training and a smaller part is used for testing. For example, we can use 70 percent of the data for training. We then use 30 percent for testing. We use training set to build a model and discover predictive relationships. We then use a testing set to evaluate model performance. When we have completed testing our model, we should use all the data to train the model.

Training/Testing Sets

Data: 

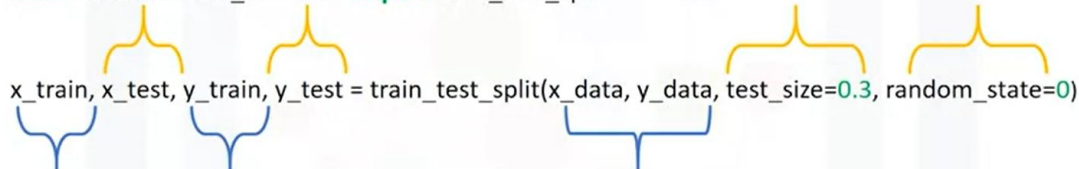
- Split dataset into:
 - Training set (70%), 
 - Testing set (30%) 
- Build and train the model with a training set
- Use testing set to assess the performance of a predictive model
- When we have completed testing our model we should use all the data to train the model to get the best performance

Function `train_test_split()`

- Split data into random train and test subsets

```
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=0.3, random_state=0)
```



- **x_data**: features or independent variables
- **y_data**: dataset target: `df['price']`
- **x_train, y_train**: parts of available data as training set
- **x_test, y_test**: parts of available data as testing set
- **test_size**: percentage of the data for testing (here 30%)
- **random_state**: number generator used for random sampling

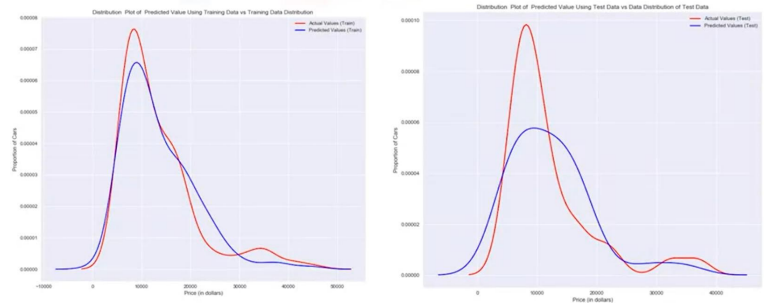
A popular function, in the scikit-learn package for splitting datasets, is the ***train_test_split function***. This function randomly splits a dataset into training and testing subsets. From the example code snippet, this method is imported from `sklearn.model_selection`. The input parameters `y_data` is the target variable. In the car appraisal example, it would be the price. `x_data`, the list of predictive variables. In this case, it would be all the other variables in the car dataset that we are using to try to predict the price. The output is an array. `x_train` and `y_train` the subsets for training. `x_test` and `y_test` the subsets for testing. In this case, the test size is a percentage of the data for the testing set. Here, it is 30 percent. **The random state is a random seed for random data set splitting.**

总结:

Generalization Performance

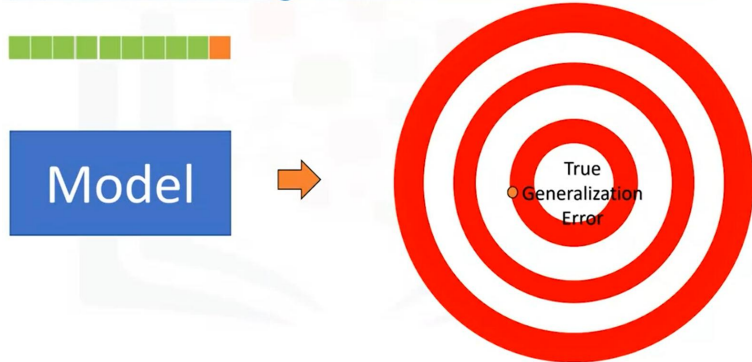
- Generalization error is measure of how well our data does at predicting previously unseen data
- The error we obtain using our testing data is an approximation of this error

Generalization Error



Generalization error is a measure of how well our data does at predicting previously unseen data. The error we obtain using our testing data is an approximation of this error. This figure shows the distribution of the actual values in red compared to the predicted values from a linear regression in blue. We see the distributions are somewhat similar. If we generate the same plot using the test data, we see the distributions are relatively different. The difference is due to a generalization error and represents what we see in the real world.

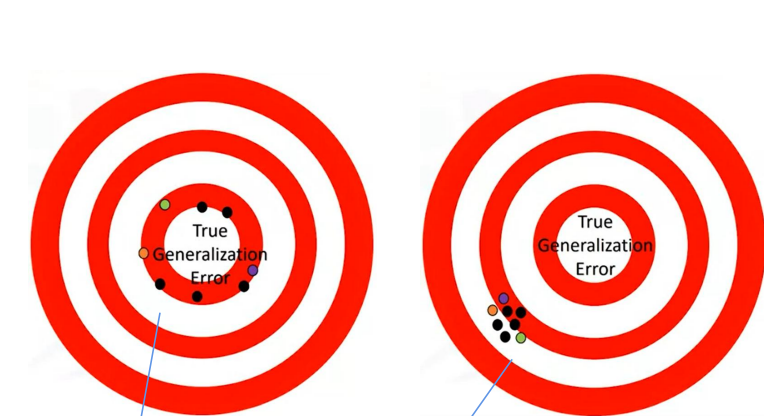
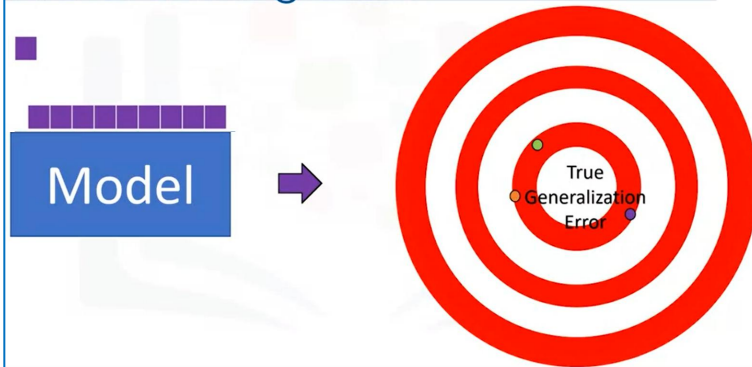
Lots of Training Data



Lots of Training Data



Lots of Training Data



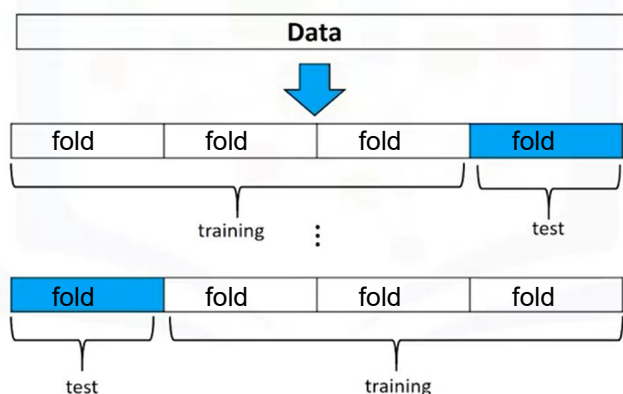
Using a lot of data for training, gives us an accurate means of determining how well our model will perform in the real world. But the precision of the performance will be low. Let's clarify this with an example. The center of this bull's eye represents the correct generalization error.

- Let's say we take a random sample of the data using 90 percent of the data for training and 10 percent for testing. The first time we experiment, we get a good estimate of the training data.
- If we experiment again training the model with a different combination of samples, we also get a good result. But, the results will be different relative to the first time we run the experiment.
- Repeating the experiment again with a different combination of training and testing samples, the results are relatively close to the generalization error, but distinct from each other.
- Repeating the process, we get good approximation of the generalization error, but the precision is poor i.e. all the results are extremely different from one another. (这里的 precision 相当于 各个结果的集中程度)
- If we use fewer data points to train the model and more to test the model, the accuracy of the generalization performance will be less, but the model will have good precision. The figure above demonstrates this. All our error estimates are relatively close together, but they are further away from the true generalization performance.

To overcome this problem, we use **cross-validation**. One of the most common out of sample evaluation metrics is cross-validation.

Cross Validation

- Most common out-of-sample evaluation metrics
- More effective use of data (each observation is used for both training and testing)



Function cross_val_score()

```
from sklearn.model_selection import cross_val_score
```

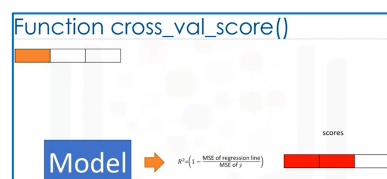
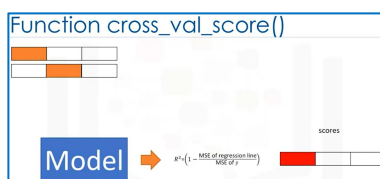
```
scores = cross_val_score(lr, x_data, y_data, cv=3)
```

np.mean(scores)

... One of the most common out of sample evaluation metrics is **cross-validation**. In this method, the dataset is split into **K equal groups**. Each group is referred to as a **fold**. For example, **four folds**. Some of the folds can be used as a training set which we use to train the model and the remaining parts are used as a test set, which we use to test the model. For example, we can use three folds for training, then use one fold for testing. **This is repeated until each partition is used for both training and testing. At the end, we use the average results as the estimate of out-of-sample error. The evaluation metric depends on the model, for example, the r squared.**

The simplest way to apply cross-validation is to call the **cross_val_score** function, which performs multiple out-of-sample evaluations. This method is imported from sklearn's **model_selection** package. We then use the function **cross_val_score**. The first input parameters, the type of model we are using to do the cross-validation. In this example, we initialize a linear regression model or object **lr** which we passed the **cross_val_score** function. **The other parameters are x_data, the predictive variable data, and y_data, the target variable data.** We can manage the number of partitions with the **cv** parameter. Here, **cv** equals 3, which means the data set is split into three equal partitions. **The function returns an array of scores, one for each partition that was chosen as the testing set. We can average the result together to estimate out of sample r squared using the np.mean() function.**

Function cross_val_score()



Model

$$R^2 = \left(1 - \frac{\text{MSE of regression line}}{\text{MSE of } y} \right)$$



Let's see an animation, let's see the result of the score array in the last slide.

First, we split the data into three folds. We use two folds for training, the remaining fold for testing. The model will produce an output. We will use the output to calculate a score. In the case of the r squared i.e. coefficient of determination, we will store that value in an array. We will repeat the process using two folds for training and one fold for testing, save the score, then use a different combination for training and the remaining fold for testing. We store the final result.

Function cross_val_predict()

- It returns the prediction that was obtained for each element when it was in the test set
- Has a similar interface to cross_val_score()

```
from sklearn.model_selection import cross_val_predict
```

```
yhat= cross_val_predict (lr2e, x_data, y_data, cv=3)
```



Function cross_val_predict()

x ₁	x ₂	x ₃	x ₄	x ₅	x ₆	x ₇	x ₈	x ₉
x ₁	x ₂	x ₃						
			x ₄	x ₅	x ₆			
						x ₇	x ₈	x ₉

Model

x ₁	x ₂	x ₃	x ₄	x ₅	x ₆	x ₇	x ₈	x ₉
x ₁	x ₂	x ₃						
			x ₄	x ₅	x ₆			

Model

y ₇	y ₈	y ₉
----------------	----------------	----------------

x ₁	x ₂	x ₃	x ₄	x ₅	x ₆	x ₇	x ₈	x ₉
----------------	----------------	----------------	----------------	----------------	----------------	----------------	----------------	----------------

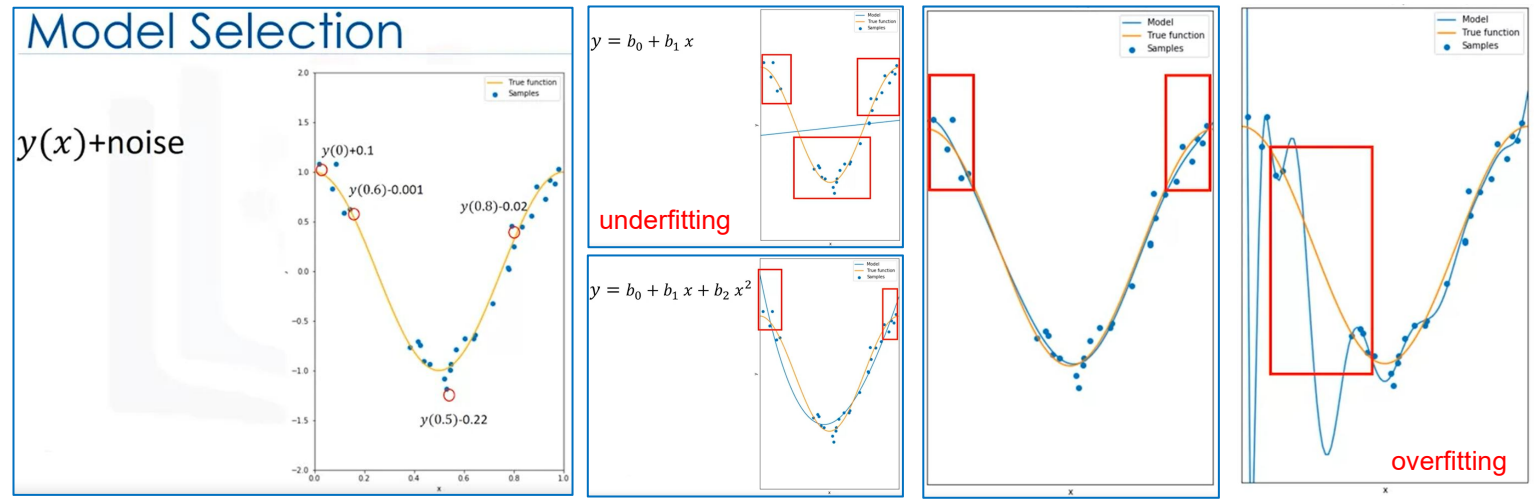
Model

y ₁	y ₂	y ₃
y ₄	y ₅	y ₆
y ₇	y ₈	y ₉

The cross_val_score function returns a score value to tell us the cross-validation result. What if we want a little more information? What if we want to know the actual predicted values supplied by our model before the r squared values are calculated? To do this, we use the **cross_val_predict** function. The input parameters are exactly the same as the cross_val_score function, but the output is a prediction. Let's illustrate the process.

First, we split the data into three folds. We use two folds for training, the remaining fold for testing. The model will produce an output, and we will store it in an array. We will repeat the process using two folds for training, one for testing. The model produces an output again. Finally, we use the last two folds for training. Then we use the testing data. This final testing fold produces an output. **These predictions are stored in an array.**

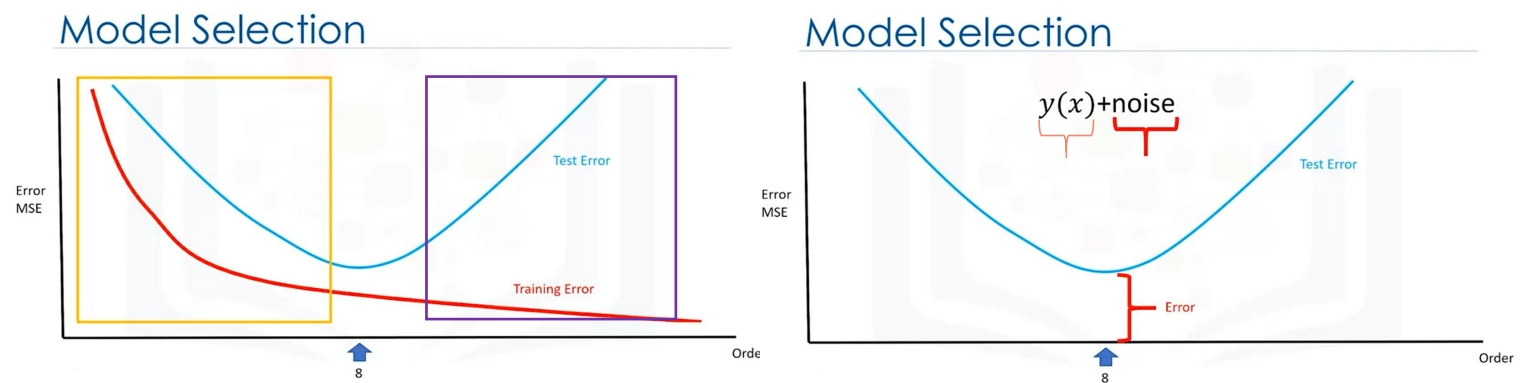
2. Overfitting, Underfitting and Model Selection



If you recall, in the last module, we discussed polynomial regression. In this section, we will discuss how to pick the best polynomial order and problems that arise when selecting the wrong order polynomial.

Consider the following function, we assume the training points come from a polynomial function plus some noise. The goal of Model Selection is to determine the order of the polynomial to provide the best estimate of the function $y(x)$.

- If we try and fit the function with a linear function, the line is not complex enough to fit the data. As a result, there are many errors. This is called **underfitting**, where the model is too simple to fit the data.
- If we increase the order of the polynomial, the model fits better, but the model is still not flexible enough and exhibits underfitting.
- This is an example of the 8th order polynomial used to fit the data. We see the model does well at fitting the data and estimating the function even at the inflection points.
- Increasing it to a 16th order polynomial, the model does extremely well at tracking the training point but performs poorly at estimating the function. This is especially apparent where there is little training data. The estimated function oscillates not tracking the function. This is called **overfitting**, where the model is too flexible and fits the noise rather than the function.

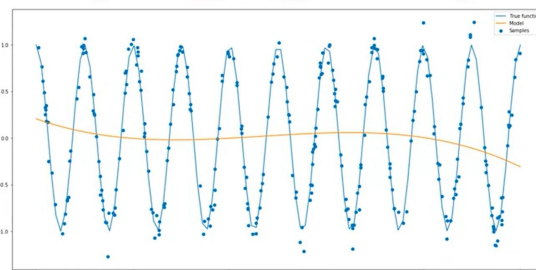


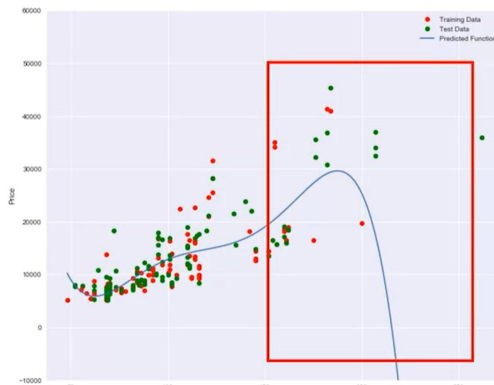
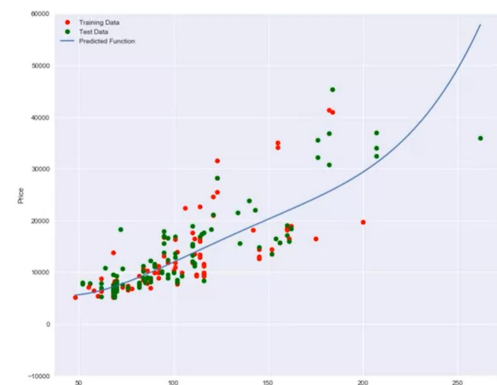
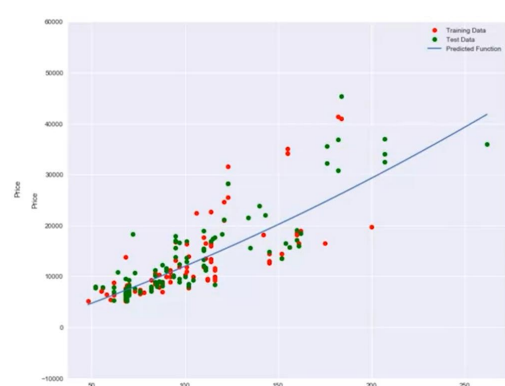
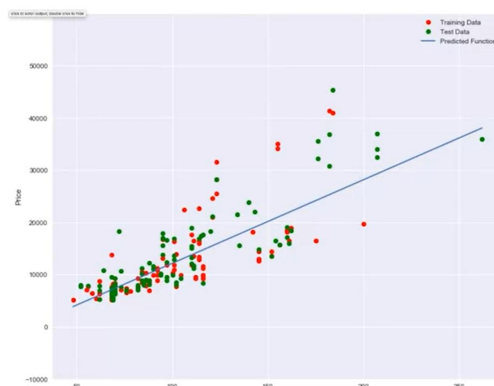
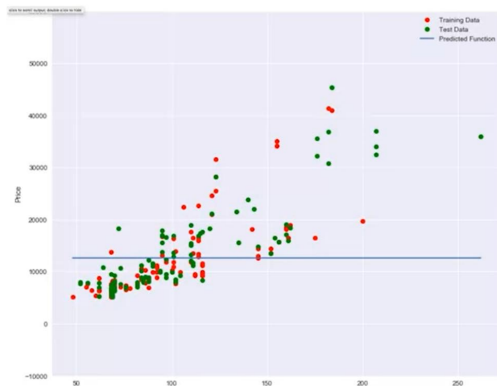
Let's look at a plot of the mean square error for the training and testing set of different order polynomials. The horizontal axis represents the order of the polynomial. The vertical axis is the mean square error. The training error decreases with the order of the polynomial. The test error is a better means of estimating the error of a polynomial. The error decreases 'till the best order of the polynomial is determined. Then the error begins to increase. We select the order that minimizes the test error. In this case, it was eight. Anything on the left would be considered underfitting. Anything on the right is overfitting.

If we select the best order of the polynomial, we will still have some errors.

- If you recall the original expression for the training points we see a **noise term**. This term is one reason for the error. This is because the noise is random, and we can't predict it. This is sometimes referred to as an **irreducible error**.
- There are other sources of errors as well. For example, our polynomial assumption may be wrong. Our sample points may have come from a different function. For example, in this plot, the data is generated from a sine wave. The polynomial function does not do a good job at fitting the sine wave. For real data, the model may be too difficult to fit or we may not have the correct type of data to estimate the function. (见下图)

Model Selection





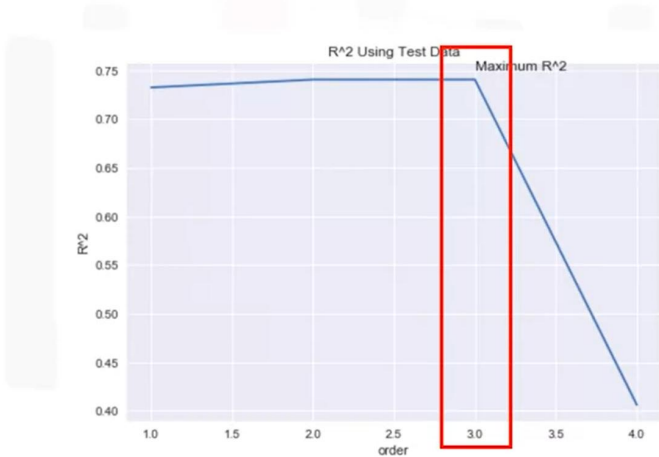
Let's try different order polynomials on the real data using horsepower. The red points represent the training data. The green points represent the test data.

1. If we just use the mean of the data, our model does not perform well.
2. A linear function does fit the data better.
3. A second order model looks similar to the linear function.
4. A third order function also appears to increase, like the previous two orders.
5. Here, we see a fourth order polynomial. At around 200 horsepower, the predicted price suddenly decreases. This seems erroneous.

Let's use R-squared to see if our assumption is correct. The following is a plot of the R-squared value. The horizontal axis represents the order polynomial models. The closer the R-squared is to one, the more accurate the model is. Here, we see the R-squared is optimal when the order of the polynomial is three. The R-squared drastically decreases when the order is increased to four, validating our initial assumption.

We can calculate different R-squared values as follows. First, we create an empty list to store the values. We create a list containing different polynomial orders. We then iterate through the list using a loop. We create a polynomial feature object with the order of the polynomial as a parameter. We transform the training and test data into a polynomial using the fit transform method. We fit the regression model using the transform data. We then calculate the R-squared using the test data and store it in the array.

Model Selection



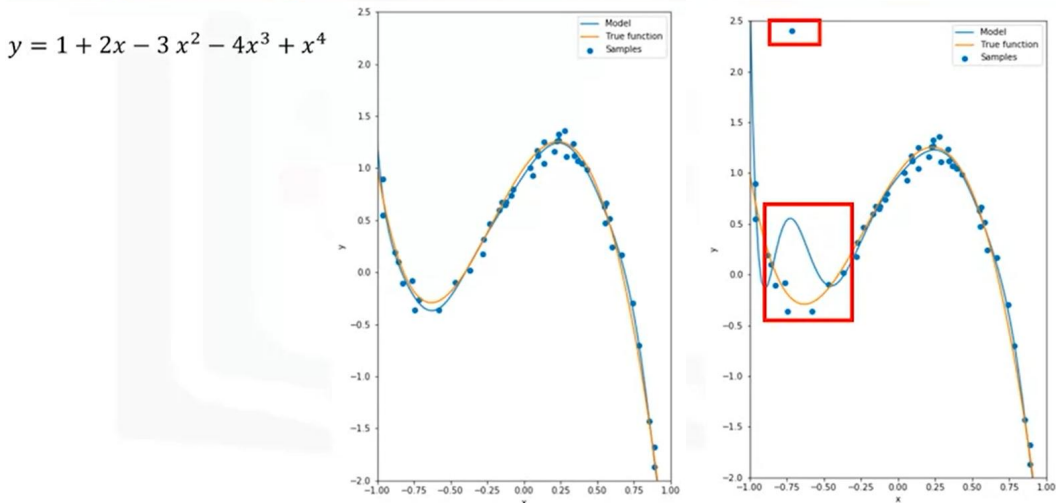
```
Rsqu_test=[]
order=[1,2,3,4]
for n in order:
    pr=PolynomialFeatures(degree=n)
    x_train_pr=pr.fit_transform(x_train[['horsepower']])
    x_test_pr=pr.fit_transform(x_test[['horsepower']])
    lr.fit(x_train_pr,y_train)
    Rsqu_test.append(lr.score(x_test_pr,y_test))
```

3. Ridge Regression

Ridge Regression Introduction

Ridge regression is a regression that is employed in a Multiple regression model when Multicollinearity occurs. Multicollinearity is when there is a strong relationship among the independent variables. Ridge regression is very common with polynomial regression. Next we will show how Ridge regression is used to regularize and reduce the standard errors to avoid over-fitting a regression model.

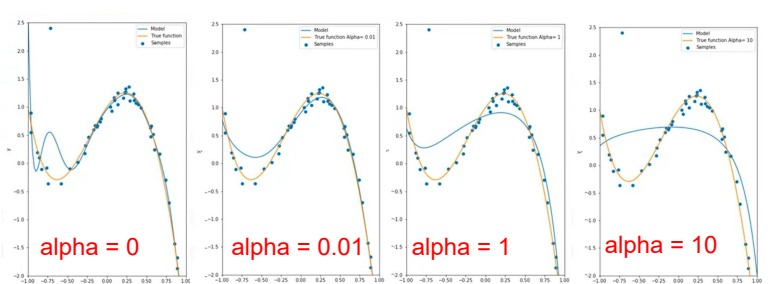
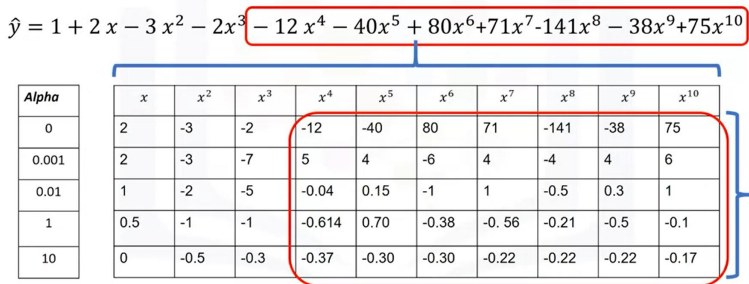
Ridge Regression



In this video, we'll discuss **ridge regression**. **Ridge regression prevents overfitting**. We will focus on polynomial regression for visualization, but overfitting is also a big problem when you have multiple independent variables, or features. Consider.

The following fourth order polynomial in orange. The blue points are generated from this function. We can use a tenth order polynomial to fit the data. The estimated function in blue does a good job at approximating the true function. In many cases real data has outliers. For example, this point shown here does not appear to come from the function in orange. If we use a tenth order polynomial function to fit the data, the estimated function in blue is incorrect, and is not a good estimate of the actual function in orange.

Ridge Regression



If we examine the expression for the estimated function, we see the estimated polynomial coefficients have a very large magnitude. This is especially evident for the higher order polynomials. **Ridge regression controls the magnitude of these polynomial coefficients by introducing the parameter alpha. Alpha is a parameter we select before fitting or training the model.** Each row in the following table represents an increasing value of alpha. Let's see how different values of alpha change the model. This table represents the polynomial coefficients for different values of alpha. The column corresponds to the different polynomial coefficients, and the rows correspond to the different values of alpha. As alpha increases, the parameters get smaller. This is most evident for the higher order polynomial features. But **Alpha must be selected carefully. If alpha is too large, the coefficients will approach zero and underfit the data.**

If alpha is zero, the overfitting is evident. For alpha equal to 0.001, the overfitting begins to subside. For Alpha equal to 0.01, the estimated function tracks the actual function. When alpha equals one, we see the first signs of underfitting. The estimated function does not have enough flexibility. At alpha equals to 10, we see extreme underfitting. It does not even track the two points.

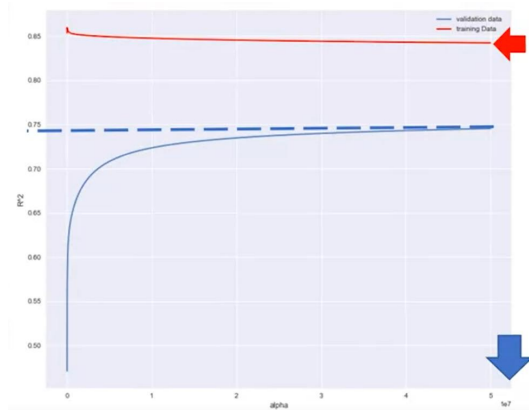
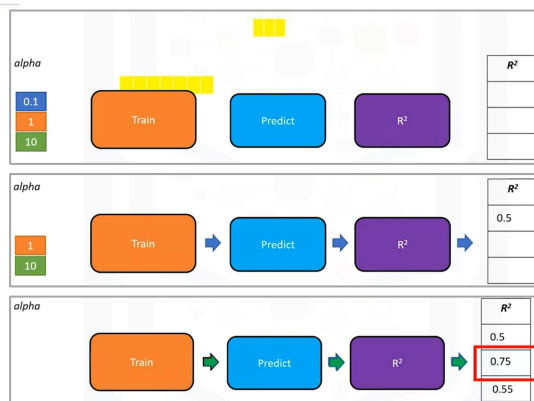
Ridge Regression

```
from sklearn.linear_model import Ridge
```

```
RidgeModel=Ridge(alpha=0.1)
```

```
RidgeModel.fit(X,y)
```

```
Yhat=RidgeModel.predict(X)
```



In order to select alpha, we use cross validation. To make a prediction using ridge regression, import Ridge from sklearn.linear_model. Create a ridge object using the constructor. The parameter alpha is one of the arguments of the constructor. We train the model using the fit method. To make a prediction, we use the predict method.

In order to determine the parameter alpha, we use some data for training. We use a second set called **validation data**. This is similar to test data, but it is used to select parameters like alpha.

- We start with a small value of alpha. We train the model, make a prediction using the validation data, then calculate the R-squared and store the values.
- Repeat the value for a larger value of alpha. We train the model again, make a prediction using the validation data, then calculate the R-squared and store the values of R-squared.
- We repeat the process for a different alpha value, training the model, and making a prediction.

We select the value of alpha that maximizes the R-squared. Note that we can use other metrics to select the value of alpha, like mean squared error.

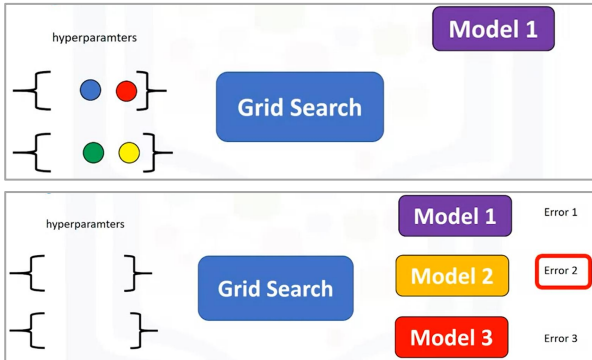
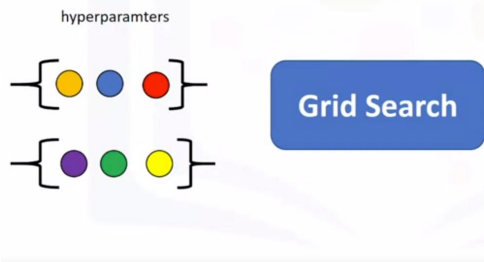
The overfitting problem is even worse if we have lots of features. The following plot shows the different values of R-squared on the vertical axis. The horizontal axis represents different values for alpha. We use several features from our used car data set and a second order polynomial function. The training data is in red, and validation data is in blue. We see as the value for alpha increases, the value of R-squared increases and converges at approximately 0.75. In this case, we select the maximum value of alpha because running the experiment for higher values of alpha have little impact. Conversely, as alpha increases, the R-squared on the test data decreases. This is because the term alpha prevents overfitting. This may improve the results in the unseen data, but the model has worse performance on the test data.

See the lab on how to generate this plot.

4. Grid Search

Hyperparameters

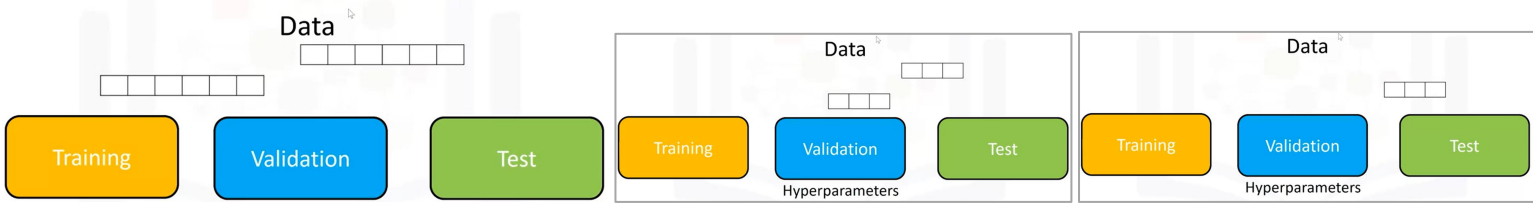
- In the last section, the term alpha in Ridge regression is called a hyperparameter
- Scikit-learn has a means of automatically iterating over these hyperparameters using cross-validation called Grid Search



Grid Search allows us to scan through multiple free parameters with few lines of code. Parameters like the alpha term discussed in the previous video are not part of the fitting or training process. These values are called **hyperparameters**. Scikit-learn has a means of automatically iterating over these hyperparameters using **cross-validation**. This method is called **Grid Search**. **Grid Search takes the model or objects you would like to train and different values of the hyperparameters. It then calculates the mean square error or R-squared for various hyperparameter values, allowing you to choose the best values.**

Let the small circles represent different hyperparameters. We start off with one value for hyperparameters and train the model. We use different hyperparameters to train the model. We continue the process until we have exhausted the different free parameter values. Each model produces an error. We select the hyperparameter that minimizes the error.

Grid Search



To select the hyperparameter, we split dataset into three parts:

- **training set**
- **validation set**
- **test set**

We train the model for different hyperparameters. We use the R-squared or mean square error for each model. **We select the hyperparameter that minimizes the mean squared error or maximizes the R-squared on the validation set.** We finally test our model performance using the test data.

What data do we use to pick the best hyperparameter

☐ Training data

☒ Validation data

☐ Test data

✓ 正确
correct

sklearn.linear_model.Ridge

```
class sklearn.linear_model.Ridge(alpha=1.0, fit_intercept=True, normalize=False, copy_X=True, max_iter=None, tol=0.001, solver='auto', random_state=None):
```

Linear least squares with l2 regularization.

This model solves a regression model where the loss function is the linear least squares function and regularization is given by the l2-norm. Also known as Ridge Regression or Tikhonov regularization. This estimator has built-in support for multi-variate regression (i.e., when y is a 2d-array of shape [n_samples, n_targets]).

Read more in the [User Guide](#).

Parameters

- alpha**: (float, array-like), shape (n_targets)
- fit_intercept**: boolean
- normalize**: boolean, optional, default False
- copy_X**: boolean, optional, default True

This is the scikit-learn web page, where the object constructor parameters are given. It should be noted that the attributes of an object are also called parameters. We will not make the distinction even though some of the options are not hyperparameters per se. In this module, we will focus on the hyperparameter alpha and the normalization parameter.

总结:

Grid Search

```
parameters = [{ 'alpha': [1, 10, 100, 1000] } ]
```

Alpha	1	10	100	1000
-------	---	----	-----	------

Ridge()

Grid Search

Ridge()

Scoring
Number
of Folds

Grid Search CV

Alpha	1	10	100	1000
-------	---	----	-----	------

Grid Search CV

Alpha	1	10	100	1000
R ²	0.74	0.35	0.073	0.008

The value of your Grid Search is a Python list that contains a Python dictionary. The key is the name of the free parameter. The value of the dictionary is the different values of the free parameter. This can be viewed as a table with various free parameter values. We also have the object or model. The Grid Search takes on the scoring method, in this case, R-squared. The number of folds, the model or object, and the free parameter values. Some of the outputs include the different scores for different free parameter values. In this case, the R-squared along with a free parameter values that have the best score.

```
from sklearn.linear_model import Ridge
from sklearn.model_selection import GridSearchCV

parameters1 = [{'alpha': [0.001, 0.1, 1, 10, 100, 1000, 10000, 100000, 1000000]}]

RR = Ridge()

Grid1 = GridSearchCV(RR, parameters1, cv=4)

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

Grid1.best_estimator_

scores = Grid1.cv_results_
scores['mean_test_score']

array([ 0.66549413, 0.66554568, 0.66602936, 0.66896822, 0.67334636, 0.65781884, 0.65781884])
```

First, we import the libraries we need, including GridSearchCV, the dictionary of parameter values.

We create a ridge regression object or model.

We then create a GridSearchCV object. The inputs are the ridge regression object, the parameter values, and the number of folds.

We will use R-squared. This is the default scoring method.

We fit the object. We can find the best values for the free parameters using the attribute best estimator.

We can also get information like the mean score on the validation data using the attribute CV result.

Grid Search

Ridge()

Grid Search CV

Alpha	1	10	100	1000
True	0.69	0.32	0.17	0.17
False	0.67	0.66	0.66	0.64

Alpha	1	10	100	1000
Normalize	True	True	True	True
	False	False	False	False

```
parameters = [{ 'alpha': [1, 10, 100, 1000], 'normalize': [True, False] } ]
```

```
from sklearn.linear_model import Ridge
from sklearn.model_selection import GridSearchCV

parameters2 = [{'alpha': [0.001, 0.1, 1, 10, 100], 'normalize': [True, False]}]

RR = Ridge()

Grid1 = GridSearchCV(RR, parameters2, cv=4)

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

Grid1.best_estimator_

scores = Grid1.cv_results_
```

What are the advantages of Grid Search is how quickly we can test multiple parameters.

For example, ridge regression has the option to normalize the data. To see how to standardize, see module four. The term alpha is the first element in the dictionary. The second element is the normalized option. The key is the name of the parameter. The value is the different options. In this case because we can either normalize the data or not. The values are True or False respectively. The dictionary is a table or grid that contains two different values. As before, we need the ridge regression object or model. The procedure is similar except that we have a table or grid of different parameter values. The output is the score for all the different combinations of parameter values.

The code is also similar. The dictionary contains the different free parameter values. We can find the best value for the free parameters. The resulting scores of the different free parameters are stored in this dictionary, Grid1.cv_results_. We can print out the score for the different free parameter values. The parameter values are stored as shown here. See the course labs for more examples.

```
for param, mean_val, mean_test in zip(scores['params'], scores['mean_test_score'], scores['mean_train_score']):
    print(param, "R2 on test data:", mean_val, "R2 on train data:", mean_test)
```

```
{ 'alpha': 0.001, 'normalize': True } R2 on test data: 0.66605547293 R2 on train data: 0.814001968709
{ 'alpha': 0.001, 'normalize': False } R2 on test data: 0.665488366584 R2 on train data: 0.814002698797
{ 'alpha': 0.1, 'normalize': True } R2 on test data: 0.694175625356 R2 on train data: 0.810546768311
{ 'alpha': 0.1, 'normalize': False } R2 on test data: 0.665488937796 R2 on train data: 0.814002698794
{ 'alpha': 1, 'normalize': True } R2 on test data: 0.690486934584 R2 on train data: 0.749104440368
{ 'alpha': 1, 'normalize': False } R2 on test data: 0.665494127178 R2 on train data: 0.814002698472
{ 'alpha': 10, 'normalize': True } R2 on test data: 0.321376875232 R2 on train data: 0.341856042902
{ 'alpha': 10, 'normalize': False } R2 on test data: 0.665545680812 R2 on train data: 0.8140026666
{ 'alpha': 100, 'normalize': True } R2 on test data: 0.0170551710263 R2 on train data: 0.0496044796826
{ 'alpha': 100, 'normalize': False } R2 on test data: 0.666029359996 R2 on train data: 0.813999791851
{ 'alpha': 1000, 'normalize': True } R2 on test data: -0.0301961745066 R2 on train data: 0.005184451599
{ 'alpha': 1000, 'normalize': False } R2 on test data: 0.668968215369 R2 on train data: 0.813870488264
{ 'alpha': 10000, 'normalize': True } R2 on test data: -0.0351687400461 R2 on train data: 0.000520784757979
{ 'alpha': 10000, 'normalize': False } R2 on test data: 0.673346359342 R2 on train data: 0.812583743226
{ 'alpha': 100000, 'normalize': True } R2 on test data: -0.0356685844558 R2 on train data: 5.2101975528e-05
{ 'alpha': 100000, 'normalize': False } R2 on test data: 0.657818838432 R2 on train data: 0.789541446486
{ 'alpha': 1000000, 'normalize': True } R2 on test data: -0.0356685844558 R2 on train data: 5.2101975528e-05
{ 'alpha': 1000000, 'normalize': False } R2 on test data: 0.657818838432 R2 on train data: 0.789541446486
```

Lesson Summary

In this lesson, you have learned how to:

Identify over-fitting and under-fitting in a predictive model: Overfitting occurs when a function is too closely fit to the training data points and captures the noise of the data. Underfitting refers to a model that can't model the training data or capture the trend of the data.

Apply Ridge Regression to linear regression models: Ridge regression is a regression that is employed in a Multiple regression model when Multicollinearity occurs.

Tune hyper-parameters of an estimator using Grid search: Grid search is a time-efficient tuning technique that exhaustively computes the optimum values of hyperparameters performed on specific parameter values of estimators.