Module 8: Feature Engineering and Overfitting

Quick Reference Guide

Learning Outcomes:

1. Generate nonlinear features for use by a linear model
2. Describe the difference between inference and prediction
3. Describe the relationship between training error and complexity
4. Evaluate the sensitivity of a model
5. Use a validation set for model selection to avoid overfitting
6. Use a test set for final model evaluation
7. Create and evaluate a linear regression model with nonlinear features using scikit-learn

Parabolic Model Fitting

In a linear model, predictions are a linear combination of the available features. For example, if there are three features, the model uses the following equation to predict the output:

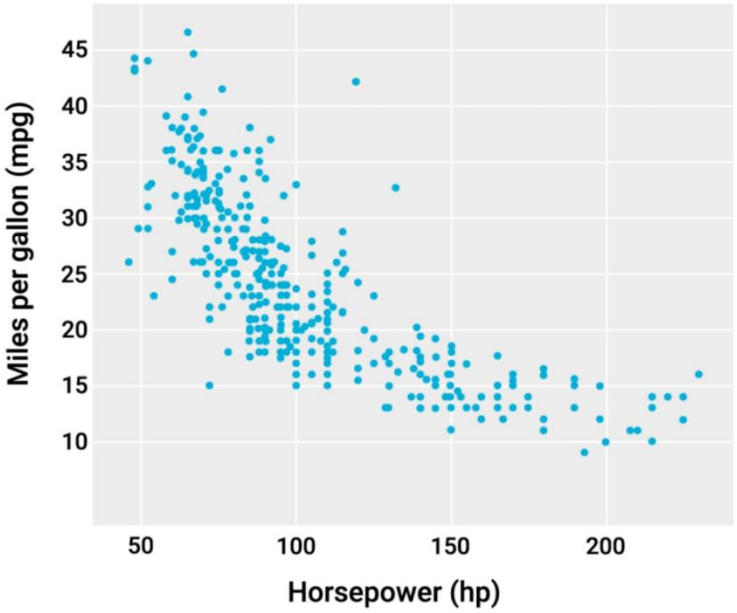
$$\hat{y} = \theta_1\phi_2 + \theta_2\phi_2 + \theta_3\phi_3$$

More generally, the output of the model can be expressed as follows, where d is the number of features and parameters, and α is an extra y-intercept term:

$$\hat{y} = \sum_{j=1}^{d} \theta_j\phi_j + \alpha$$

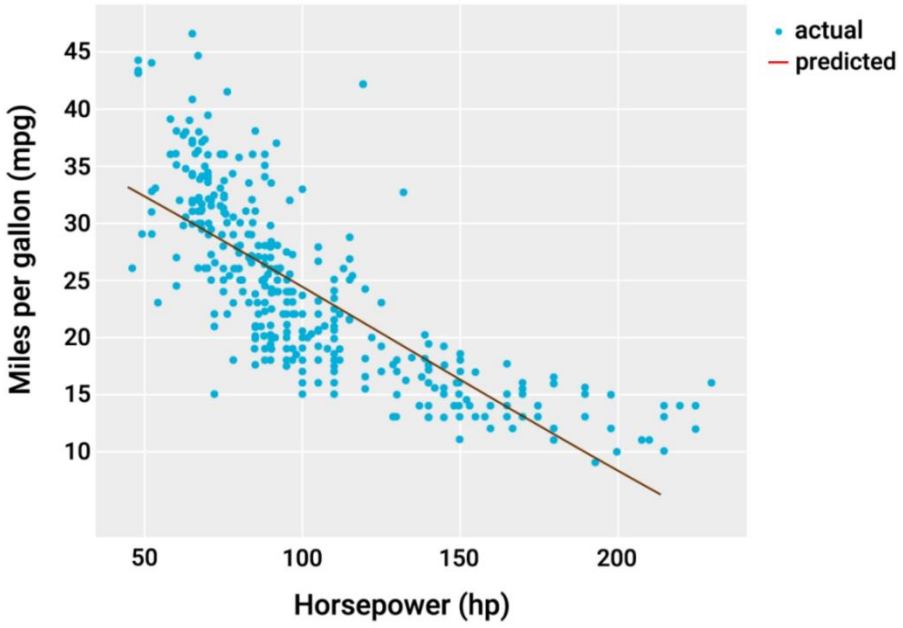
A challenge for linear models

This scatter plot shows the fuel efficiency and engine power of many different models of vehicle. The x-axis measures total engine power in horsepower, and the y-axis measures fuel efficiency in miles per gallon.



The data shows a strong inverse correlation. As engine power increases, the mileage per gallon decreases.

Suppose you want to build a model to relate miles per gallon to horsepower. Simple linear regression yields the model you see here, with a mean squared error (MSE) of 23.94.

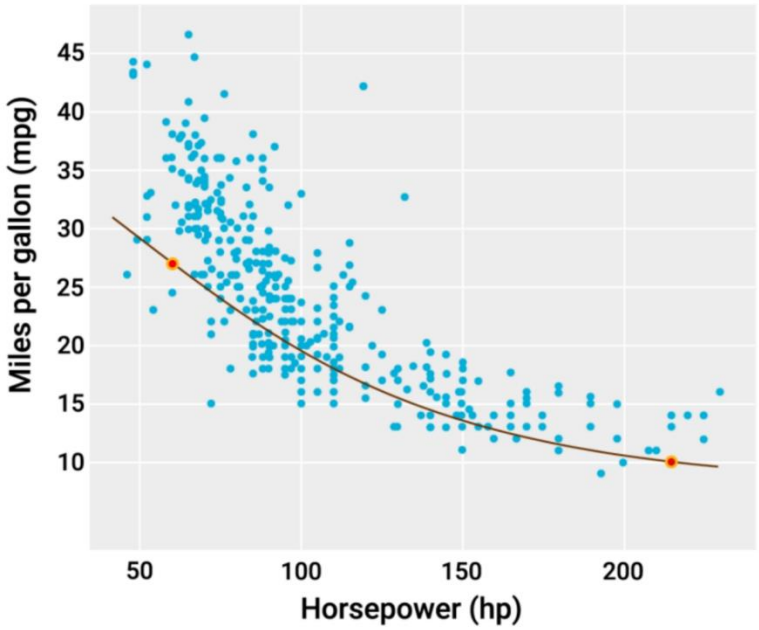


This model fails to capture the curved, nonlinear shape of the relationship between fuel efficiency and engine power.

One solution is to fit a parabola. You can roughly see that the bottom of the parabola is around x equals 250, y equals 10. And another point on the parabola is around x equals 75, y equals 27.5. Having the coordinates of the vertex and any other point on the parabola is enough to derive its equation:

$$y = \frac{(x - 250)^2}{1{,}750} + 10$$

When plotted, the curve of this equation aligns more closely with the data.



It is instructive to also consider what this equation looks like when expanded out:

$$y = \frac{x^2}{1{,}750} + \frac{2}{7}x + 45.7$$

This yields an MSE of 21, which is much better than 23.94 using simple linear regression.

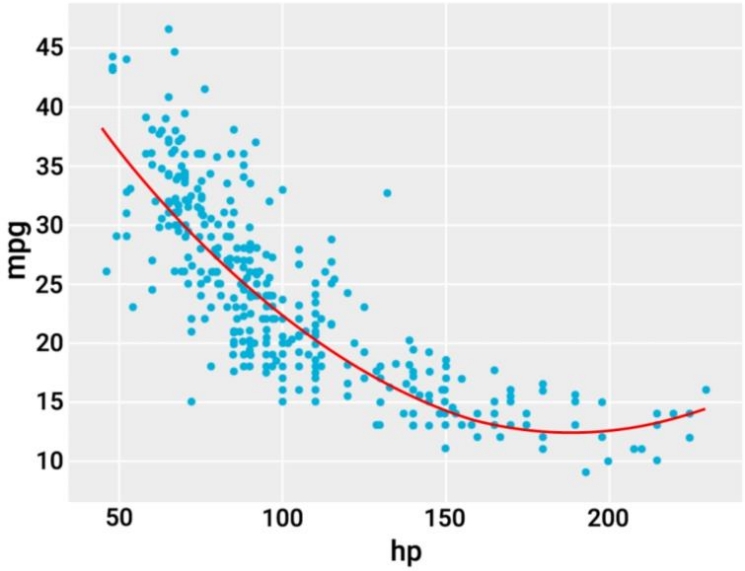
Nonlinear Features

A simple linear regression model is insufficient for producing mean squared error (MSE) predictions because it lacks a squared term. There is no such thing as squared regression in scikit-learn. However, there is a simple way to get nonlinear behavior using the same linear regression library.

To do this, add a square of the x-axis to your model. For example, in this dataframe containing horsepower and miles per gallon values, you add an additional horsepower squared feature called hp2.

Vehicle Data			
	hp	hp2	mpg
0	130.0	16,900.0	18.0
1	165.0	27,225.0	15.0
2	150.0	22,500.0	18.0
...	...	...	...

Next, you use the linear regression model to fit the vehicle data using both the hp and hp2 features. Since the mathematical equations are of the same form, this produces a parabola like the one fitted manually earlier.



The nice thing about this is that it uses the same code as for linear regression, except for the inclusion of the hp2 column in the dataframe.

```
from sklearn.linear_model import LinearRegression
model = LinearRegression()
model.fit(vehicle_data[['hp', 'hp2']], vehicle_data['mpg'])
```

Comparing the model created manually (by eyeballing the data) versus scikit-learn's model, you see they are very similar.

The manually created model:

$$y = \frac{(x - 250)^2}{1{,}750} + 10$$

$$y = \frac{x^2}{1{,}750} + \frac{2}{7}x + 45.7$$

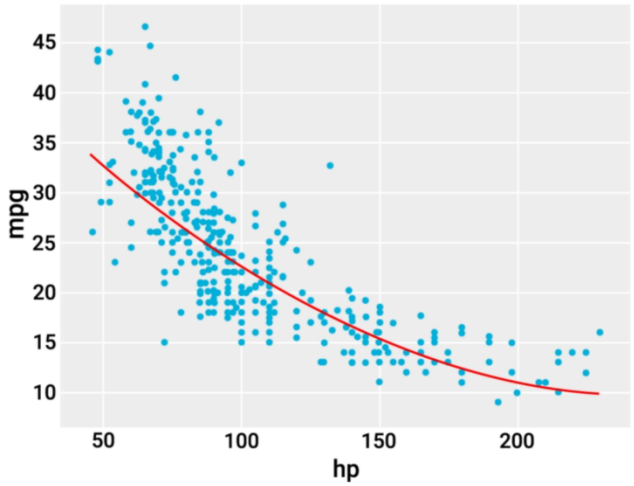
The scikit-learn model:

$$y = \frac{(x - 189.424)^2}{812.68} + 12.746$$

$$y = \frac{x^2}{812.68} + 0.466x + 56.9$$

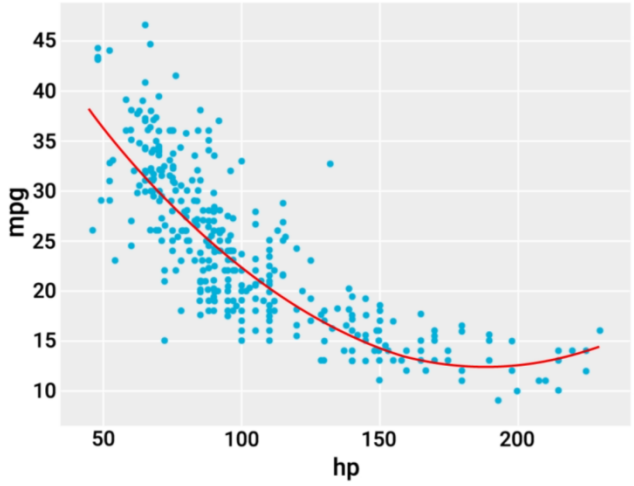
Both are parabolas with an output $\hat{y}$ equal to ϕ_1 times horsepower, plus ϕ_2 times horsepower squared, plus an intercept term. However, their ϕ values are different. For example, in the manually created model, ϕ_2, the coefficient for horsepower squared, is 1 over 1,750. And in the scikit-learn model, ϕ_2 is 1 over 812.68.

This difference in ϕ values can be observed visually.



Manually created model

MSE = 21



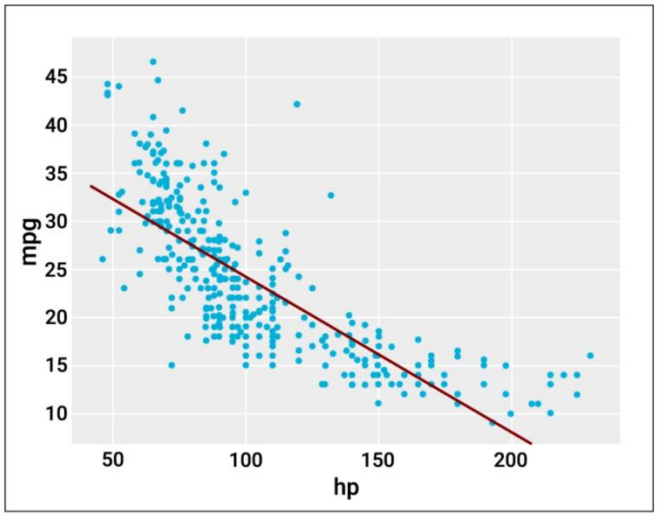
Sklearn model

MSE = 18.98

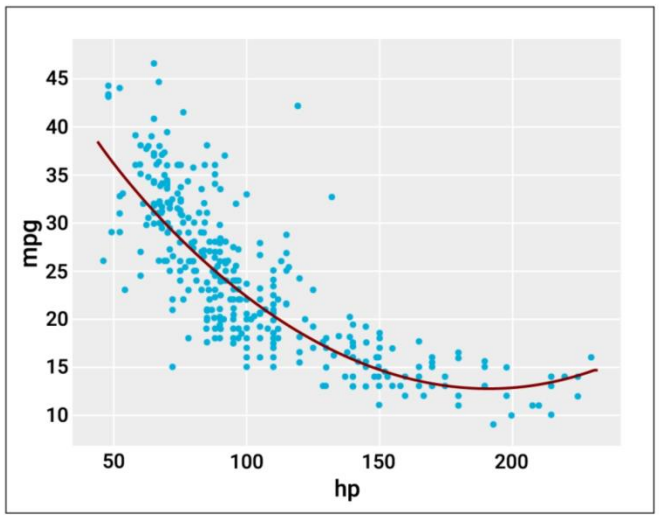
The manually created model has a mean squared error of 21. The scikit-learn parabola is steeper and better optimized for performance. It finds the absolute best parameters and yields an MSE of 18.98.

Prediction vs. Inference

Consider the two linear regression models built using scikit-learn.



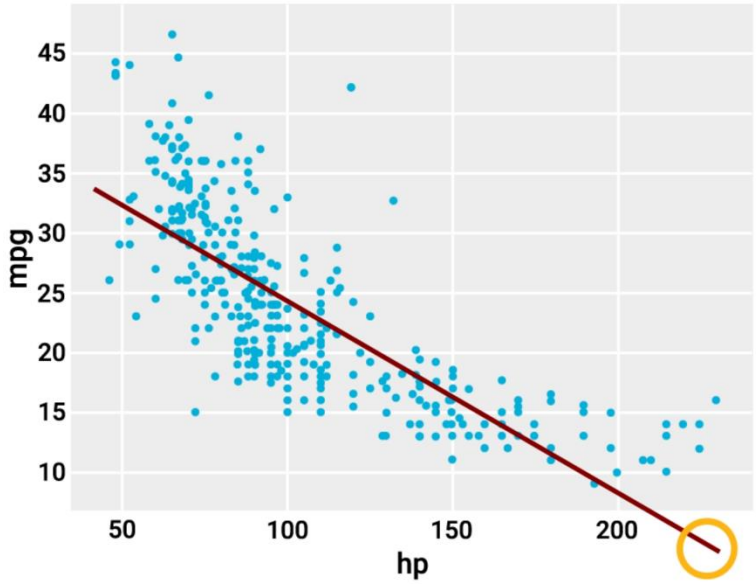
Model trained using only hp



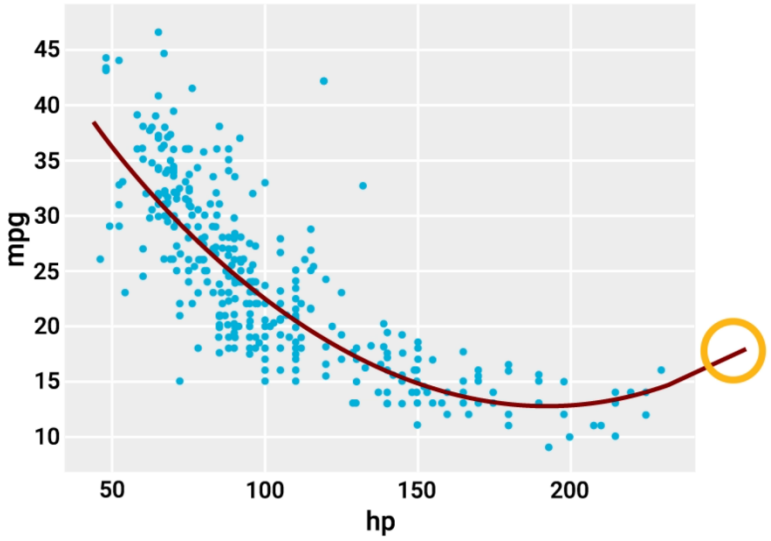
Model trained using hp and hp²

The only difference between the two is the dataframe provided for training. For the left model, the data included one feature for prediction, horsepower. For the right model, the data included two features, horsepower and horsepower squared.

Both models only make sense in the range of data that they have seen. Going outside of this range yields strange predictions. For example, the left model predicts negative fuel efficiency for engines with higher horsepower.



And the right model predicts that, as vehicles exceed 200hp, fuel efficiency will start to rapidly increase.



However, this model is still useful so long as it sticks to the original range in which it was trained.

This leads to two distinct goals when creating models: inference and prediction.

With **prediction**, the model predicts accurate outcomes from real-world data. For example, a model predicts an accurate fuel efficiency for a new vehicle with a 160-horsepower engine.

And with the goal of **inference**, the model helps you understand the true relationship. For example, it explains why fuel efficiency drops quickly as horsepower increases, before leveling off at some relatively constant value. Inference is a much harder task.

This program focuses on prediction. So if your model makes nonsensical predictions for data outside of the range in which it was trained, do not worry. This is perfectly normal.

Scikit-Learn Transformers

Linear models cannot generate higher-degree polynomial features during the fitting process. Therefore, including them in advance gives them more expressive power.

As an alternative to manually creating these features, scikit-learn provides special **transformers** that take a set of existing features as input and then output new features.

In this exercise, the goal is to create a dataframe with hp, hp2, and hp3 columns, without creating each column manually.

Vehicle Data		
hp	**hp2**	**hp3**
130.0	16,900.0	2,197,000.0
165.0	27,225.0	4,492,125.0
150.0	22,500.0	3,375,000.0
...	...	...

To do this, you use the PolynomialFeatures transformer:

poly_transform = PolynomialFeatures(degree = 3)
poly_transform.fit_transform(vehicle_data[["hp"]])

The constructor for **PolynomialFeatures()** takes the **degree** for the new transformer, which is three in this example. The **fit_transform()** method

takes the original hp values from the vehicle_data dataframe and automatically generates all the features of degree three.

The output is a large two-dimensional NumPy array, with each row corresponding to a sample from the original input dataframe.

```
array([[1.000000e+00, 1.300000e+02, 1.690000e+04, 2.197000e+06],
       [1.000000e+00, 1.650000e+02, 2.722500e+04, 4.492125e+06],
       [1.000000e+00, 1.500000e+02, 2.250000e+04, 3.375000e+06],
       ...,
       [1.000000e+00, 8.400000e+01, 7.056000e+03, 5.927040e+05],
       [1.000000e+00, 7.900000e+01, 6.241000e+03, 4.930390e+05],
       [1.000000e+00, 8.200000e+01, 6.724000e+03, 5.513680e+05]])
```

The NumPy array must be converted into a dataframe by calling **pd.DataFrame()** on the output of the **fit_transform()** method.

poly_transform = PolynomialFeatures(degree = 3)
p.DataFrame(poly_transform.fit_transform(vehicle_data[["hp"]]))

At this stage, the output contains a trivial 'bias' feature, labeled 0, which is always equal to one.

Vehicle Data			
0	**1**	**2**	**3**
1.0	130.0	16,900.0	2,197,000.0
1.0	165.0	27,225.0	4,492,125.0
1.0	150.0	22,500.0	3,375,000.0
...	...	...	...

To remove this bias feature, you set **include_bias** to False:

```
poly_transform = PolynomialFeatures(degree = 3, include_bias = False)
p.DataFrame(poly_transform.fit_transform(vehicle_data[["hp"]]))
```

The output of the **fit_transform()** function no longer has this bias column.

Vehicle Data		
0	**1**	**2**
130.0	16,900.0	2,197,000.0
165.0	27,225.0	4,492,125.0
150.0	22,500.0	3,375,000.0
...	...	...

Finally, the columns need to be renamed hp, hp^2, and hp^3. This is done using **get_feature_names_out()**.

```
poly_transform = PolynomialFeatures(degree = 3, include_bias = False)
p.DataFrame(poly_transform.fit_transform(vehicle_data[["hp"]]),
                columns = poly_transform.get_feature_names_out())
```

Providing these names to the dataframe constructor using the **columns** keyword, you get a nice dataframe with useful column names:

Vehicle Data		
hp	**hp^2**	**hp^3**
130.0	16,900.0	2,197,000.0
165.0	27,225.0	4,492,125.0
150.0	22,500.0	3,375,000.0
...	...	...

Scikit-Learn Pipelines

Now that you have higher-degree features, you can fit the model as usual. However, it cannot be used to make predictions yet.

For example, suppose you want to make a prediction for a 100hp vehicle. Entering the code **cu_model.predict([[100]])** returns an error because **LinearRegression()** is expecting three features as input, not one.

While it is possible to work around this issue by manually creating new features, or calling **fit_transform()** to create new features, a much better approach is to use a scikit-learn pipeline. A scikit-learn pipeline creates a list of data processing tasks to be completed sequentially.

This code example shows a two-stage pipeline:

```
from sklearn.pipeline import Pipeline
pipelined_model = Pipeline([
    ('josh_transform', PolynomialFeatures(degree = 3, include_bias = False)),
    ('josh_regression', LinearRegression())
```

])
pipelined_model.fit(vehicle_data[["hp"]], vehicle_data["mpg"])

It contains a **Pipeline** object with two arbitrarily named stages. The first stage, 'josh_transform,' is a degree three polynomial transformer. The second stage, 'josh_regression,' is a standard linear regression model. Using this pipeline, the horsepower data is first transformed using the PolynomialFeatures transformer, and then passed to the next stage, the LinearRegression object.

A pipeline model:

- Uses code that is simpler to read
- Can be applied to new data directly
- Can be used for both fitting and predicting
- Avoids the need to explicitly create a separate dataframe of new features
- Avoids the need to keep track of separate transformer and regression objects

The downside of working with a pipeline model is that it requires more syntax for some tasks. For example, if you want to know the coefficients for the linear model once it has been fit, you must first retrieve the associated pipeline stage. In this code line, the **named_steps** attribute of the Pipeline object is required to access the linear regression model:

pipelined_model.named_steps['josh_regression']

After retrieving the relevant pipeline stage, you can use it as normal. For example, you can ask for the coefficients by requesting the **.coef_** attribute.

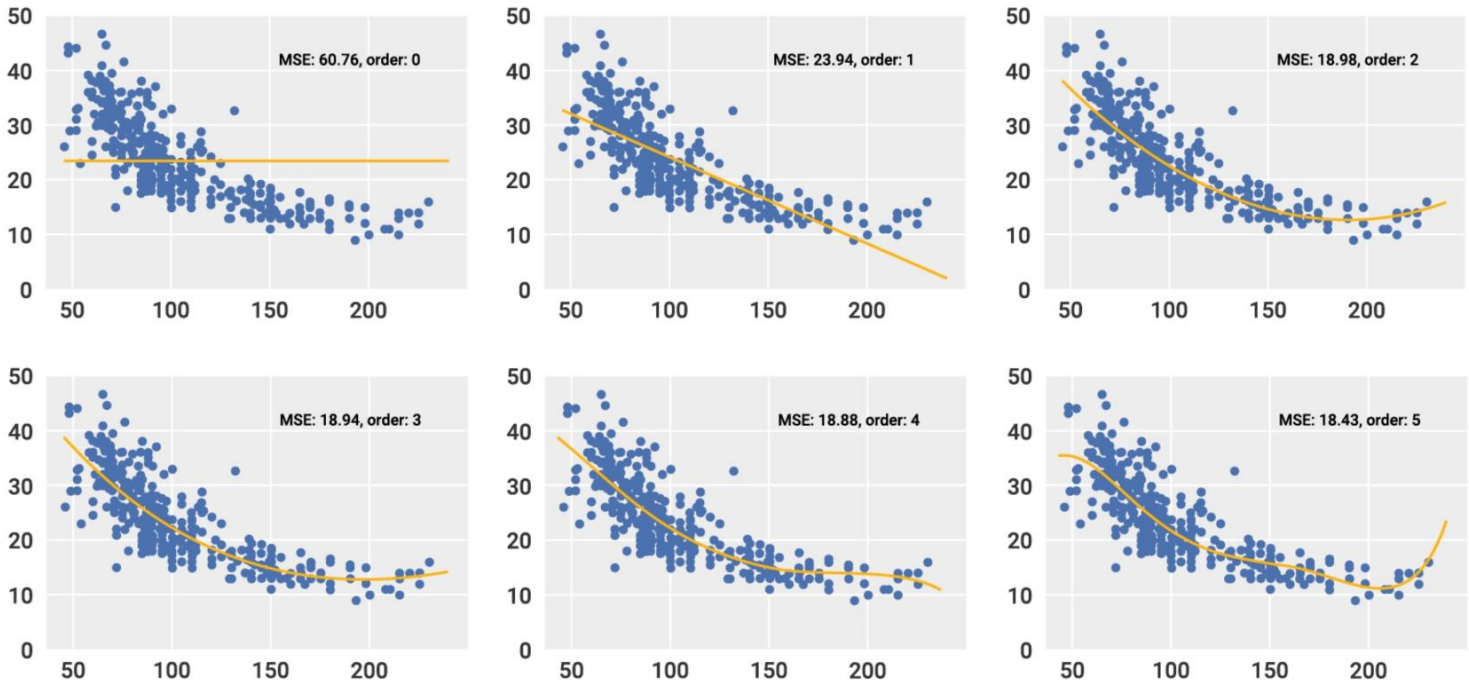
In practice, you will always use the entire pipeline when calling **predict()**:

pipelined_model.predict([[100]])

Order 0 through 6 Models on Vehicle Data

Now that you have pipelines, you can explore higher-degree models. To increase the degree of your model, simply set the **degree** parameter for **PolynomialFeatures()** to the value of your choosing.

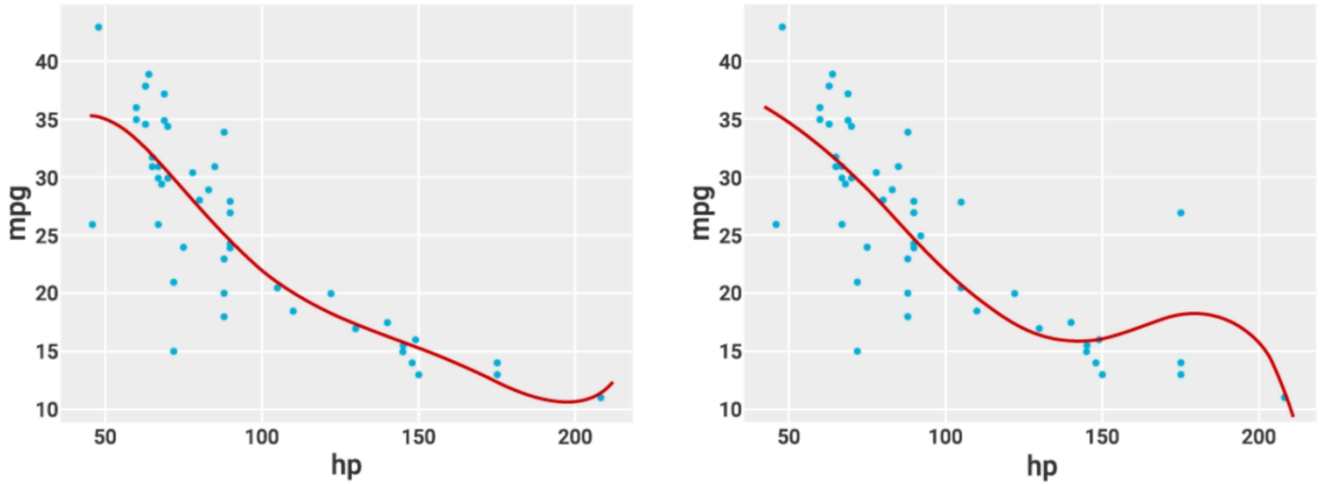
As model complexity increases, the error on the training data, or training error, decreases. For example, the mean squared error (MSE) in these six model plots decreases from 60.76 in the degree 0 model, to 18.43 in the degree 5 model.



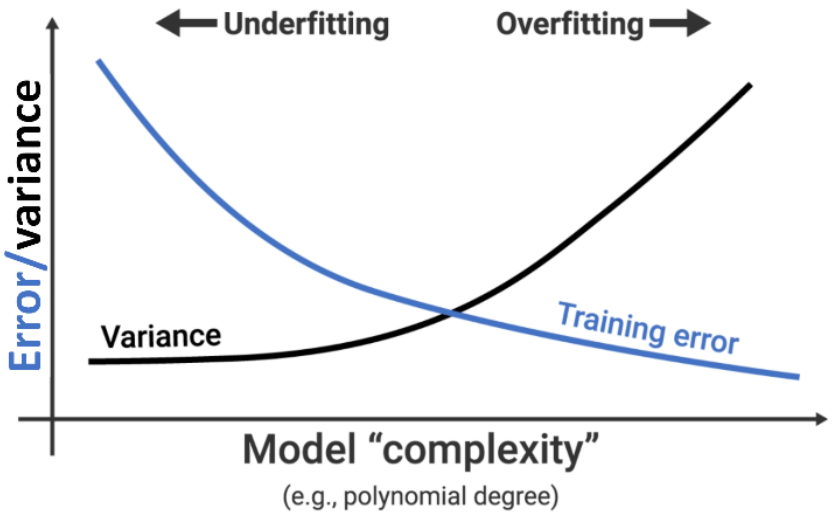
Increasing model complexity may seem to be a strictly better proposition, but it has a very important downside. The fit curve grows increasingly erratic and **sensitive** to the data as the additional polynomial degrees give

more expressive power to the linear model. In machine learning, this sensitivity to data is known as **variance**.

To better visualize this sensitivity, observe two degree-six models fit to very similar datasets.



Although these two datasets differ in only a single datapoint, the shape of the model curve is quite different. A relatively minor perturbation to the input data results in a significant change in prediction. As a result, this model is said to have high variance. This is important because anytime you collect real-world data, you get some degree of random noise.



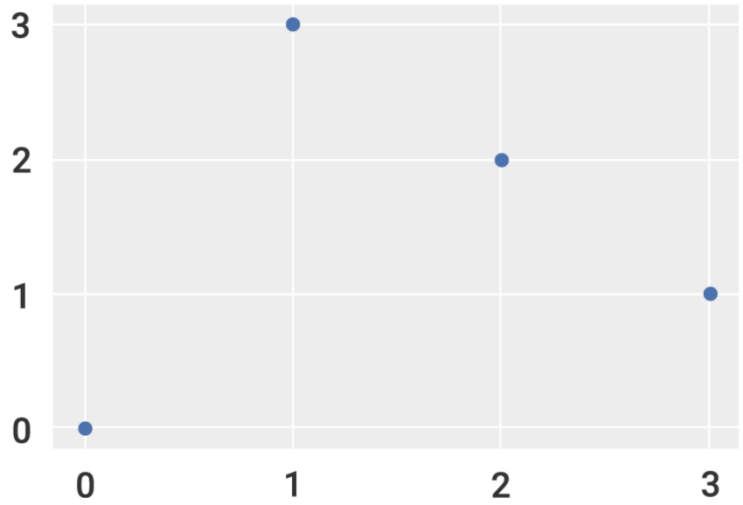
The above diagram shows the relationship between training error, variance, and model complexity.

While increased model complexity decreases the training error, it also increases the model's sensitivity to all the tiny details of the data. Being overly sensitive to the data is also known as **overfitting**.

The Dangers of Overfitting

Given N datapoints, you can always fit a model of polynomial degree N-1 that goes through all data points with zero mean squared error (MSE).

For example, consider a plot with four data points, 0,0; 1.3; 2,2; and 3,1.



Since there are four datapoints, a degree-three model can perfectly fit this data. In other words, there are θ_1, θ_2, θ_3, and α values for which $\theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \alpha$ goes through all four datapoints with zero MSE.

To do this, you enter the four datapoints into the model equation.

For example, substituting datapoint 3,1 where x = 1 and y = 3 you get the following equation:

$$\theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \alpha = y$$
$$\theta_1(1) + \theta_2(1)^2 + \theta_3(1)^3 + \alpha = (3)$$
$$\theta_1 + \theta_2 + \theta_3 + \alpha = 3$$

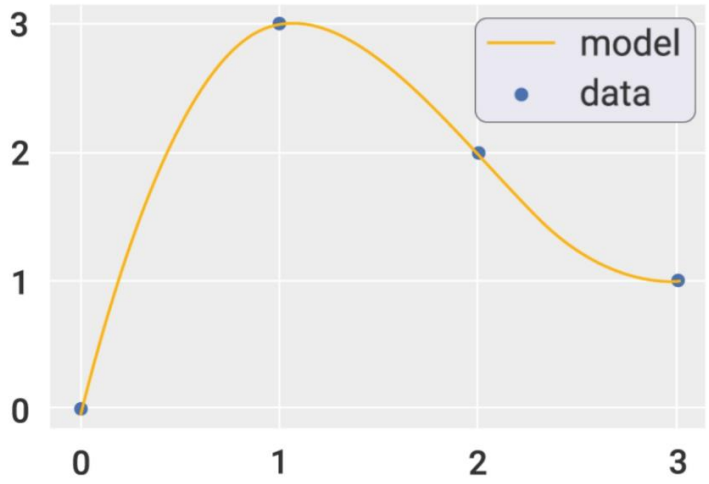
Substituting all four datapoints yields a system of four linear equations:

$$\alpha = 0$$
$$\theta_1 + \theta_2 + \theta_3 + \alpha = 3$$
$$2\theta_1 + 4\theta_2 + 8\theta_3 + \alpha = 2$$
$$3 + 9\theta_2 + 27\theta_3 + \alpha = 1$$

Using these equations, you can solve for θ_1, θ_2, θ_3, and α:

$$\theta_1 = 19/3$$
$$\theta_2 = -4$$
$$\theta_3 = 2/3$$
$$\alpha = 0$$

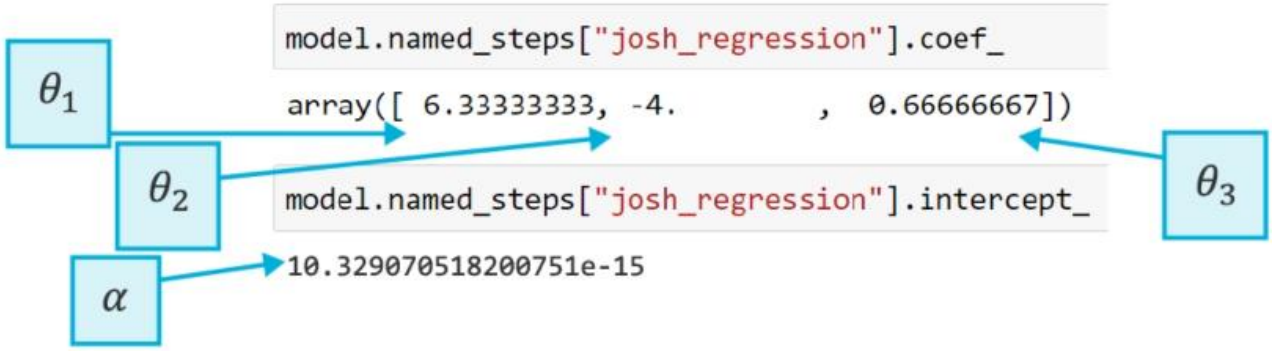
Plotting the curve with these inputs results in a perfect model of the data:



This can also be done in scikit-learn. If **arbitrary_data** is stored as a four-row, two-column DataFrame, you can call **fit()** using the pipeline shown.

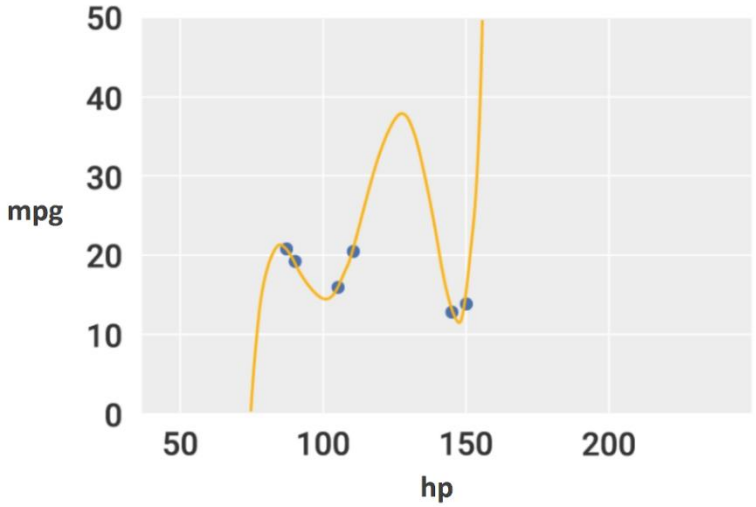
```
model = Pipeline([
    ('josh_transform', PolynomialFeatures(degree = 3, include_bias = False)),
    ('josh_regression', LinearRegression())
])
model.fit(arbitrary_data[["x"]], arbitrary_data["y"])
```

Requesting the coefficients, θ_1, θ_2, θ_3, and intercept, α, from that scikit-learn model with this code, outputs the same values. For example, θ_1 is 6.333… or $19/3$.



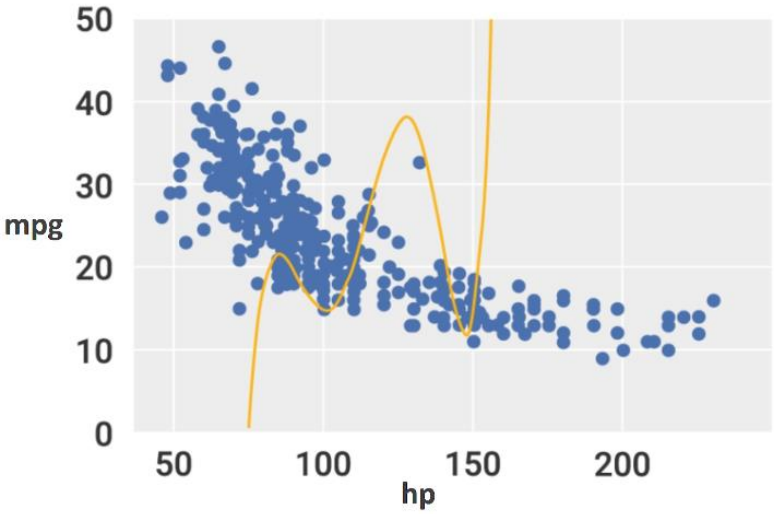
In theory, you can plot a perfect model for 100 data points, using a 100-parameter model. It will have zero MSE but would be totally useless for real-world data. This problem is called **overfitting**. It happens when a model simply memorizes existing data and cannot handle new situations.

For example, consider a degree-five model of any six vehicles from the vehicle_data dataframe.



No matter which six vehicles are picked, the model provides a perfect fit. However, it makes nonsensical predictions on other parts of the curve. For example, the miles per gallon increases rapidly from around 150hp.

If this perfectly fit model is overlayed on the full dataset, it has enormous error on a bigger sample of real-world data.



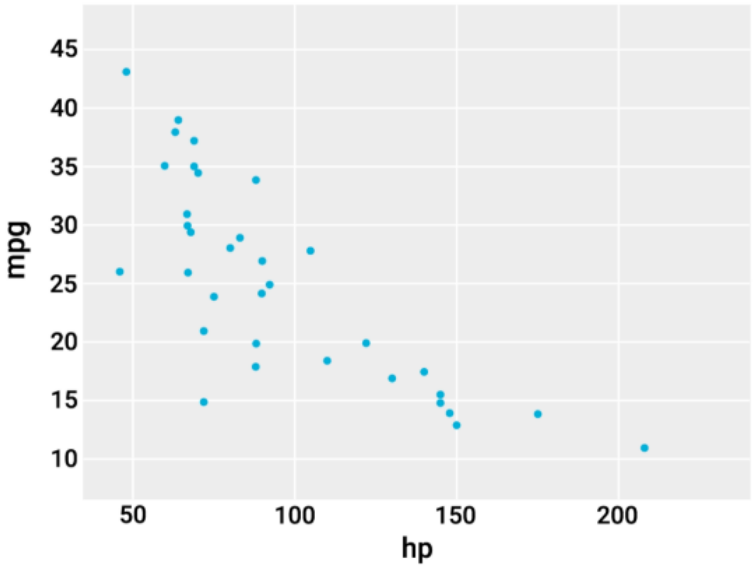
Using Feature Data to Detect Overfitting

A popular technique for detecting overfitting is known as **simple cross-validation**. To illustrate, the following exercise is restricted to 35 datapoints taken from the vehicle_data dataframe.

The code for collecting this data sample is:

vehicle_data_sample_35 = vehicle_data.sample(35)

The 35 datapoints are shown in this scatter plot:



The **get_MSE_for_degree_k_model()** function fits a pipeline model of degree k, computes the mean squared error (MSE), and then returns the MSE value.

```
def get_MSE_for_degree_k_model(k):
    pipelined_model = Pipeline([
        ('josh_transform', PolynomialFeatures(degree = k)),
        ('josh_regression', LinearRegression(fit_intercept = True))
    ])
    pipelined_model.fit(vehicle_data_sample_35[["hp"]], vehicle_data_sample_35["mpg"])
    return mean_squared_error(pipelined_model.predict(vehicle_data_sample_35[["hp"]]), vehicle_data_sample_35["mpg"])
```

This code outputs the results of **get_MSE_for_degree_k_model** for each k:

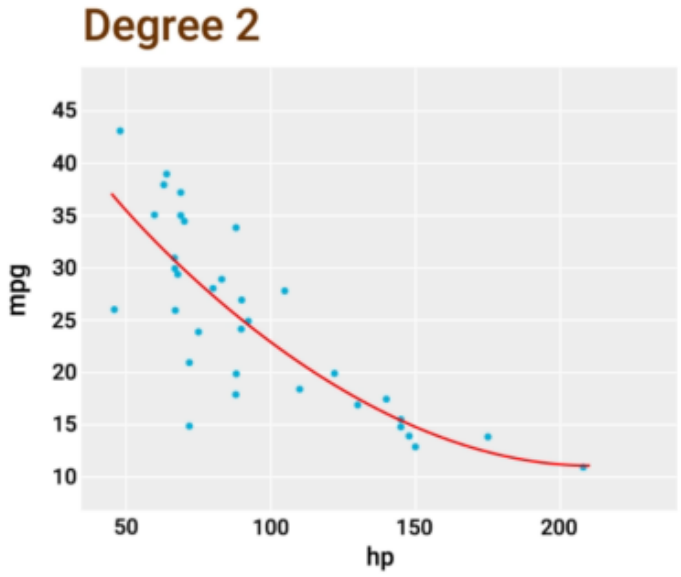
```
ks = np.array(range(0, 7))
MSEs = [get_MSE_for_degree_k_model(k) for k in ks]
MSEs_and_k = pd.DataFrame({"k": ks, "MSE" : MSEs})
```

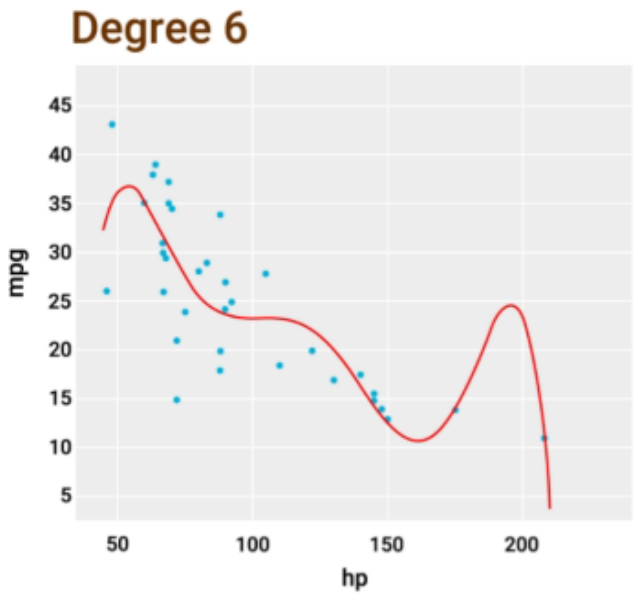
The MSE values for degree k are shown here in table form:

Degree k MSE	
k	**MSE**
0	72.091396
1	28.002727
2	25.835769
3	25.831592
4	25.763052
5	25.609403
6	23.269001

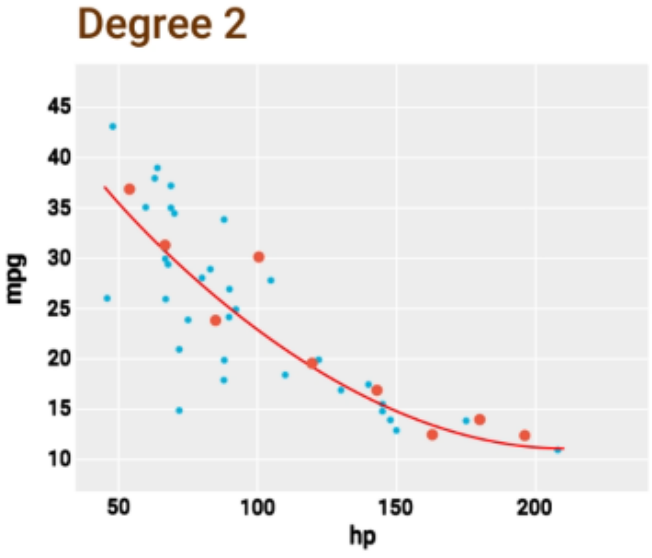
For example, a degree k equals 0 model has a training error of 72 on the original 35-point dataset. The table shows that MSE decreases monotonically as k increases.

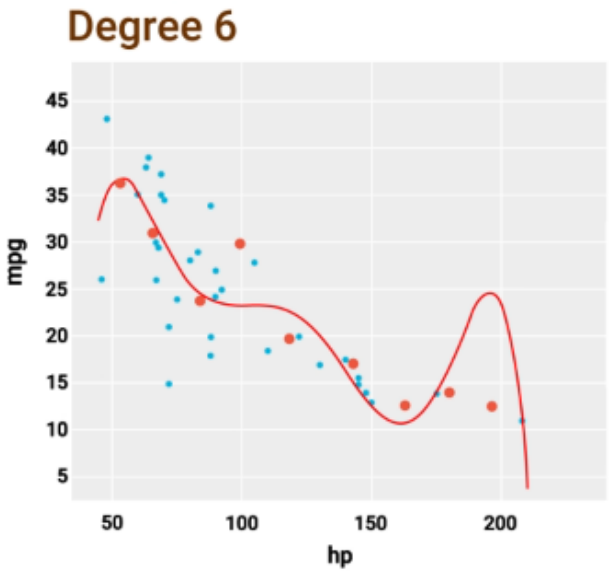
This can also be shown visually by plotting each model over the dataset. For example, the degree-two model is a parabola, while the degree-six model has an erratic shape due to overfitting.





If nine new datapoints from the original vehicle dataset are overlaid on the existing models, it is evident how well the models perform on new, real-world observations. The new datapoints are shown below in orange.





For example, the degree-six model performs poorly on the new datapoint around x equals 190. In other words, the error on that point is high.

To compare the models more thoroughly, you can compute their MSEs on just those nine new points, without making any changes to their parameters. The MSE for each model is provided in this table:

Degree k MSE	
k	**MSE**
0	69.198210
1	31.189267
2	27.287612
3	29.127612
4	34.198272
5	37.182632
6	53.128712

Based on this, the degree-two model performs best on the new data, even though its training error was much higher than those of more complex models.

There are three steps in this procedure to detect overfitting:

- Collect data
- Fit multiple models
- Evaluate models using new data

The downside to this approach is having to wait for new data after fitting the model. In some contexts, acquiring this additional data is very expensive, time consuming, or simply impossible.

Simple Cross-Validation

Rather than waiting for future data to estimate the generalizability of a model, cross-validation allows you to do so with data that you already have.

In **simple cross-validation**, the data is split into two random, non-overlapping sets. The first set is the training set, which is used to train the models. The second set is the development set, which is used to compare models after they are trained. The development set is also known as a validation set, or cross-validation set.

Whatever technique you use to generate a training and development set, it is extremely important to first randomize the order of observations.

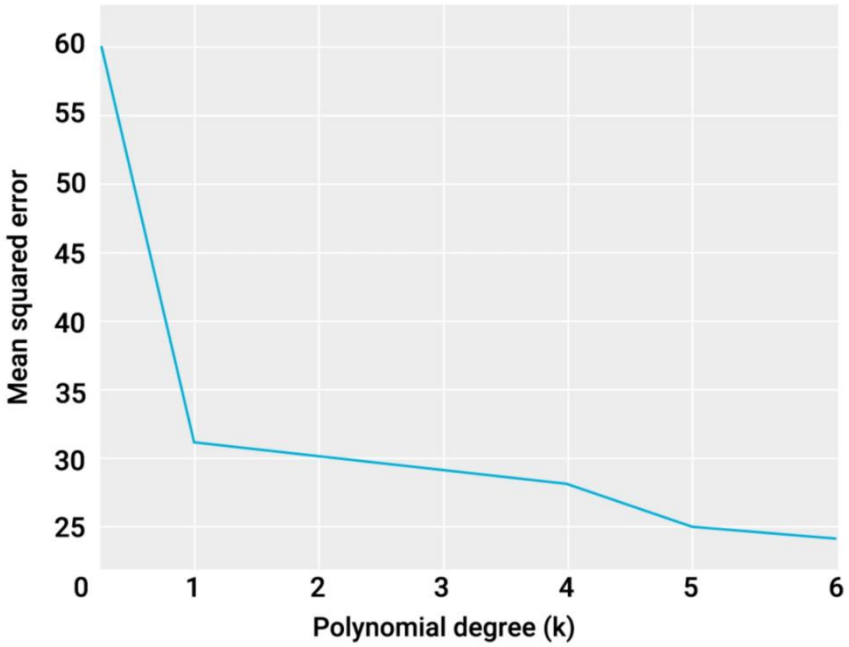
Here, the data is shuffled by calling **sklearn.utils import shuffle**.

```
from sklearn.utils import shuffle
training_set, dev_set = np.split(shuffle(vehicle_data_sample_35), [25])
```

Next, the NumPy **split()** function splits the 35 data samples into two pieces. The second argument, **[25]**, tells NumPy that the first split should be of length 25. This first split is assigned to a variable called **training_set**. The remaining ten rows are returned as the second split, which is assigned to a variable called **dev_set**.

So in this instance, you train your models on 25 datapoints and then you re-evaluate their performance using the ten development set points.

Once the models are trained using the first split, you can plot the training error as a function of model degree, k.

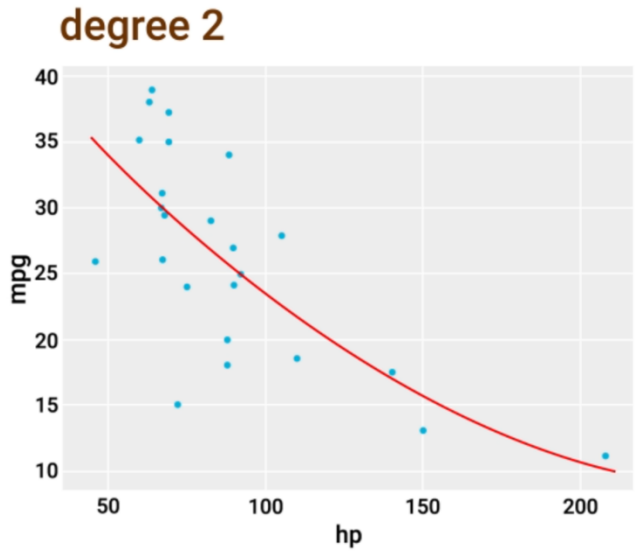


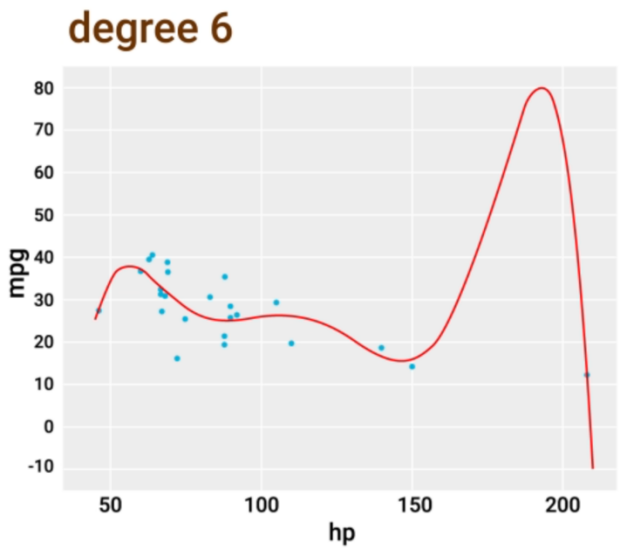
The MSE for each model is provided in this table:

Degree k MSE	
k	**MSE**
0	60.235744
1	30.756678
2	29.875269
3	29.180868
4	28.214850
5	25.290990
6	23.679651

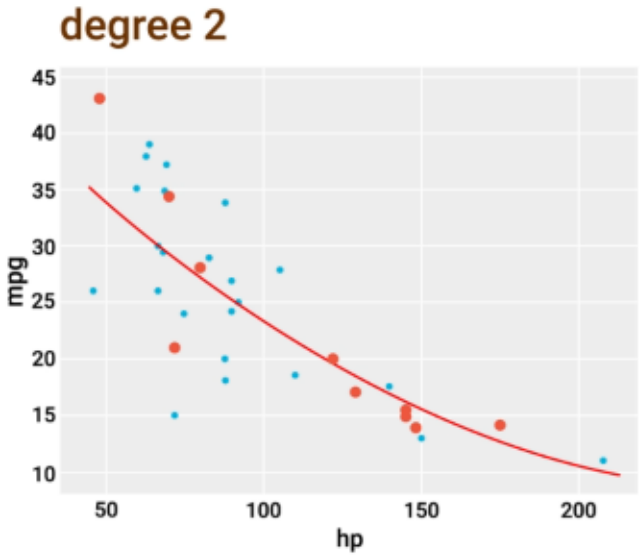
As expected, the training error decreases monotonically. If the different

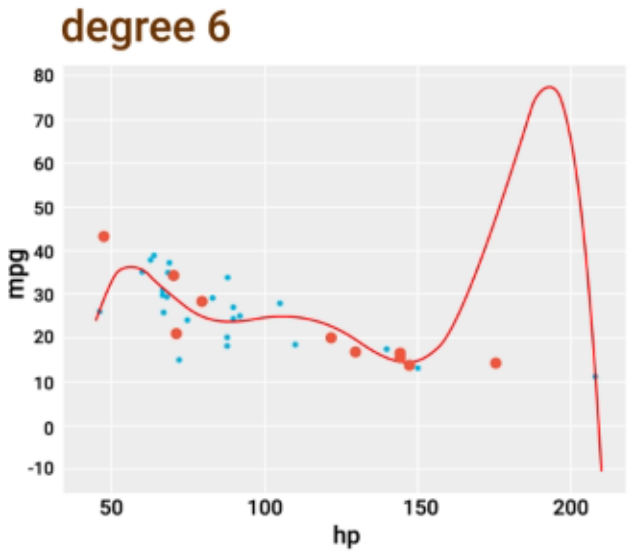
models are plotted on top of the training data, you can see that the degree-six model has high variance and seems overfit.





To determine which model is best, you compute the validation error for each model on the ten datapoints in the validation set. These datapoints are shown (below) in orange.



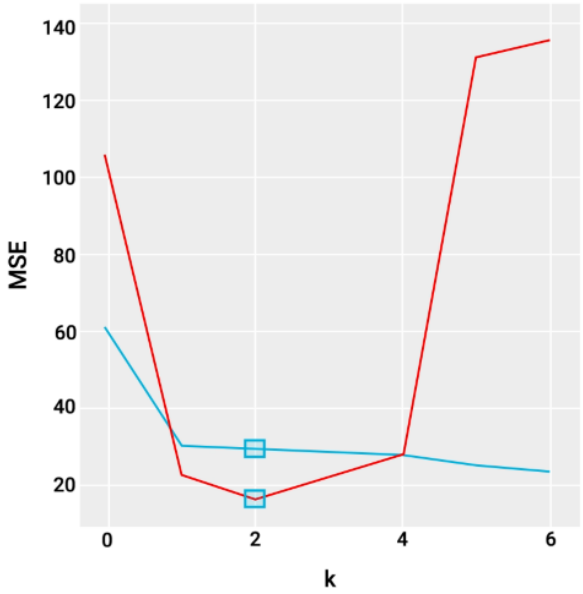


The validation and training error can be shown side-by-side for each model in this table:

Degree k MSE		
k	**Training MSE**	**Validation MSE**
0	60.235744	106.925296
1	30.756678	22.363676
2	29.875269	17.331880
3	29.180868	21.889257
4	28.214850	27.340989
5	25.290990	130.597678
6	23.679651	135.787493

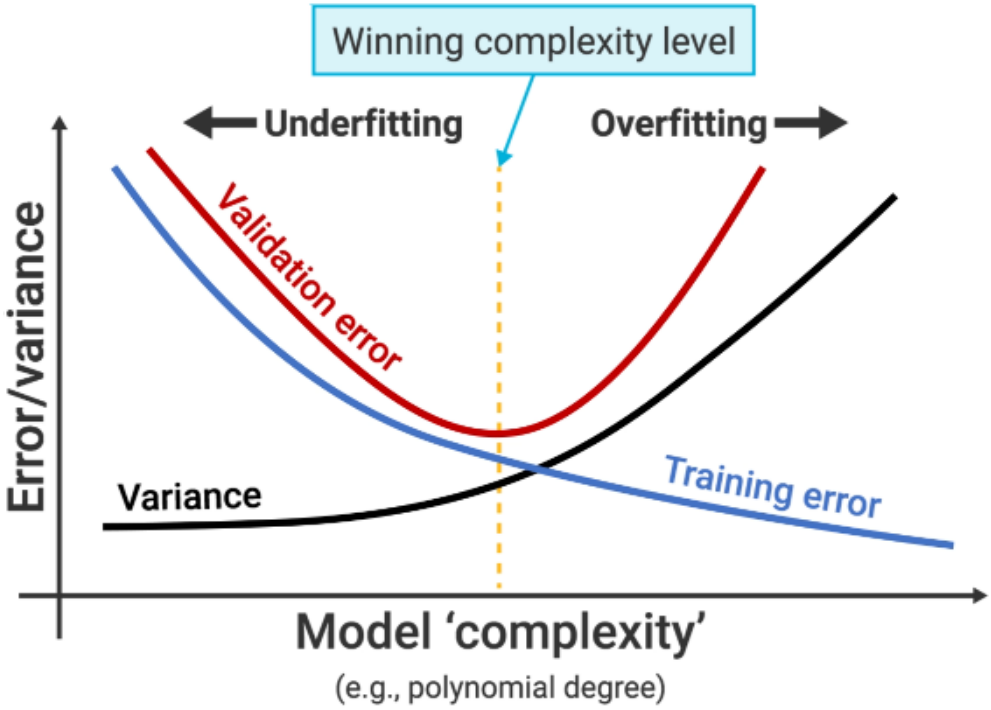
Although the validation error initially decreases as the model grows more complex, it then starts to rapidly increase from degree three and up. The degree-six model has the best training error but the worst validation error.

This plot shows the relationship between the validation and training error for each polynomial degree:



Simple cross-validation indicates that the degree-two model is the best choice since it has the lowest validation error.

This diagram shows an idealized version of the relationship between validation and training error.



As the model complexity grows, the training error decreases. By contrast, the curve of the validation error is more of a bowl shape. It decreases initially, hits some bottom point, and then increases back up as the model begins to abuse its surfeit of expressive power and starts overfitting. This increase in validation error is caused by the increased model variance. The model with the winning complexity level is positioned at the bottom of the validation error curve.

In a machine learning model, a **hyperparameter** is a value that controls the learning process itself. Each model has a hyperparameter, called **degree**, that controls how many polynomial features are generated. The training set is used to select parameters, and the development set is used to select hyperparameters.

Test Sets

A development set is used to evaluate and compare how different models perform on new, real-world data. However, while validation error does provide a reasonable measure of a model's performance, it is slightly biased. It is worth noting that this process of comparing two models is also subject to random error. For this reason, a third set, called a **test set**, is used to give a more accurate and unbiased assessment of a model.

To incorporate this final trial, the data is split into three sets at the very beginning:

- Training
- Development
- Test

It is important to make sure that the test set is only used at the very end, and not for training or development.

For example, suppose you want to build a model to predict the sales price of a diamond based on attributes, such as size and opacity. First, you shuffle the diamond data:

diamond_data = shuffle(diamond_data)
diamond_data.head()

Next, you split the data into three sets. This function splits the original 2,000 observations into 1,500 for training, 300 for cross-validation, and 200 for testing.

diamond_training_data, diamond_validation_data, diamond_test_data = np.split(diamond_data, [1500, 1800])

The 1,500 points are used to pick parameters for the model. The 300 points are used to pick between different models, including any hyperparameters. And finally, after picking the best model at the very end, you compute the mean squared error (MSE) on the test set exactly once, and report that value.

The syntax for creating the model pipeline is:

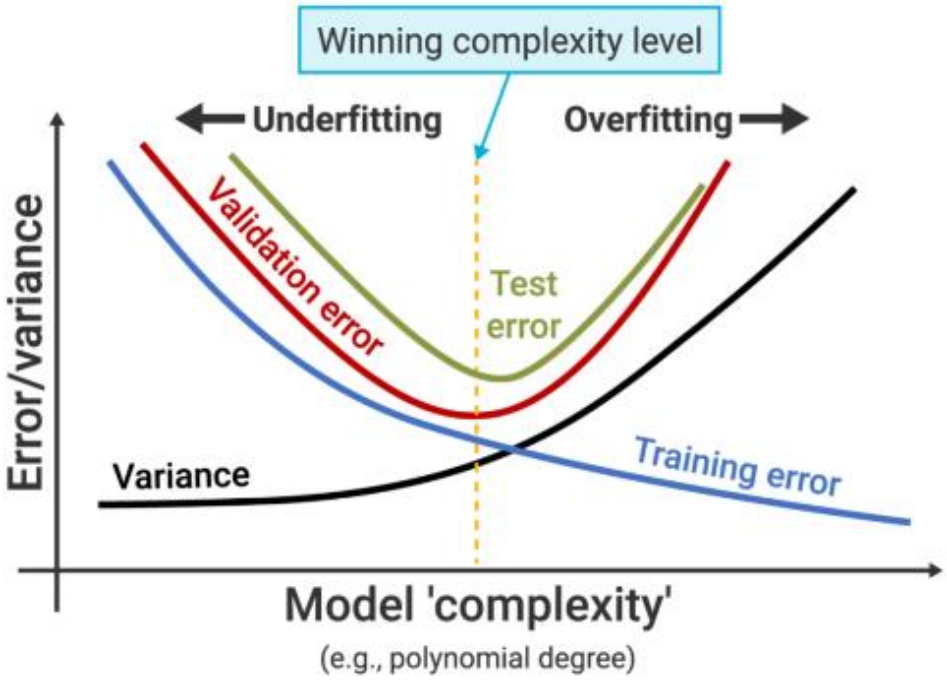
```
diamond_poly_model = Pipeline([
    ('poly', PolynomialFeatures(degree=2)),
    ('model', LinearRegression(fit_intercept=False))
])
diamond_poly_model.fit(numeric_diamond_training_data, diamond_training_data[["price"]])
```

You then compute the error on the test set using the following syntax:

mean_squared_error(diamond_poly_model.predict(numeric_diamond_test_data), diamond_test_data[["price"]])
3335120.7722796244

In this example, the test error is roughly 3.3 million. In other words, for an arbitrarily selected diamond, from the same distribution as the test set, the average MSE in diamond price will be around $3.3 million squared.

Note that the test error follows a very similar curve to the validation error.



While the entire validation error curve is often computed to pick the lowest point, this should never be done for the test error. Instead, you should pick a model and only use the test set once. The test error is the intersection of the vertical dashed line and the secret test error curve.