From:
DS100 UC Berkeley

`sklearn` and Feature Engineering

Introduction to Data Science
Spring 1403

Yadollah Yaghoobzadeh

Agenda

- Sklearn
- Feature Engineering
- One-Hot Encoding
- Polynomial Features
- Complexity and Overfitting

Sklearn

- Sklearn
- Feature Engineering
- One-Hot Encoding
- Polynomial Features
- Complexity and Overfitting

`sklearn`: a Standard Library for Model Creation

So far, we have been doing the "heavy lifting" of model creation ourselves – via calculus, ordinary least squares, or gradient descent

In research and industry, it is more common to rely on data science libraries for creating and training models. In this course, we will use Scikit-Learn, commonly called `sklearn`


```
import sklearn
my_model = linear_model.LinearRegression()
my_model.fit(X, y)
my_model.predict(X)
```

`sklearn`: a Standard Library for Model Creation

`sklearn` uses an object-oriented programming paradigm. Different types of models are defined as their own classes. To use a model, we initialize an instance of the model class.

The `sklearn` Workflow

At a high level, there are three steps to creating an **`sklearn`** model:

1 Initialize a new model instance
Make a "copy" of the model template

2 Fit the model to the training data
Save the optimal model parameters

3 Use fitted model to make predictions
Fitted model outputs predictions for y

The `sklearn` Workflow

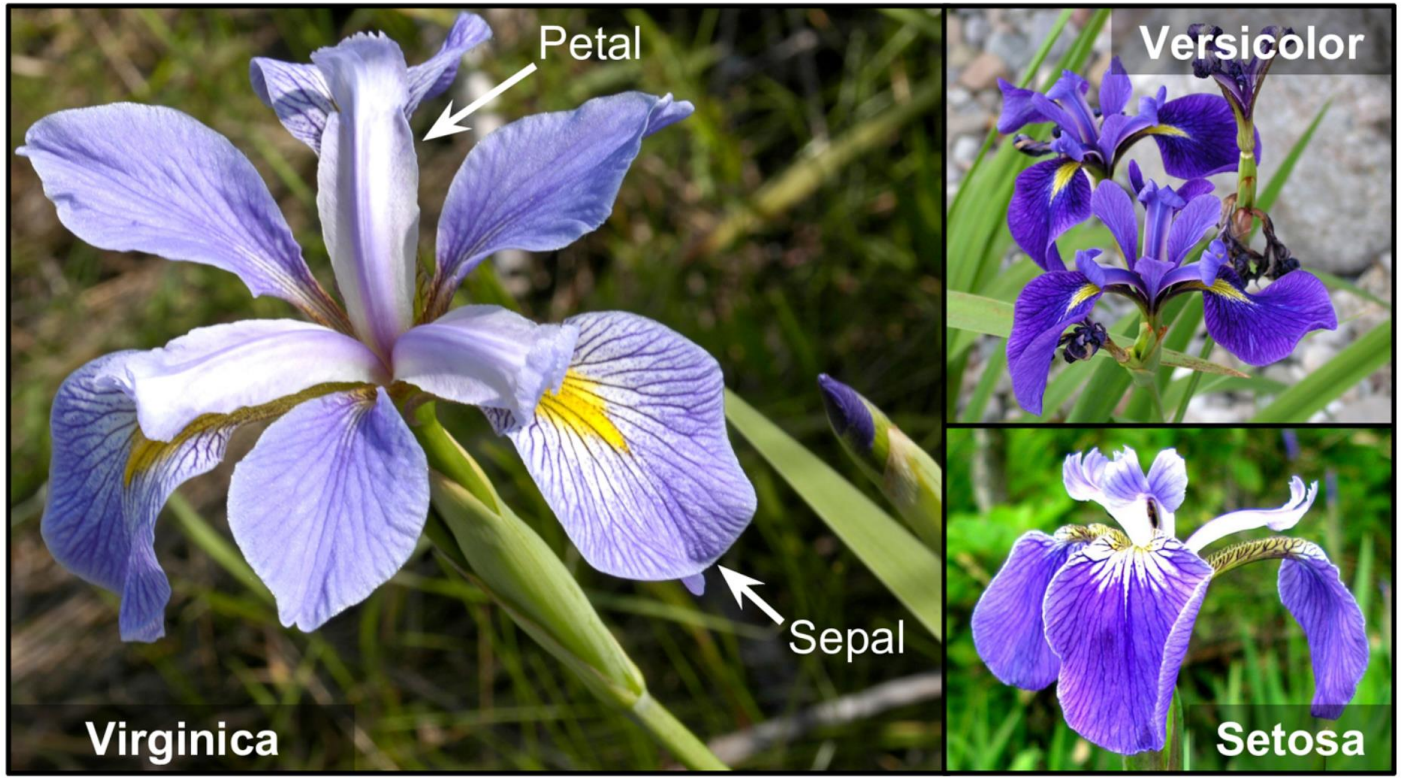
At a high level, there are three steps to creating an **`sklearn`** model:

1	Initialize a new model instance *Make a "copy" of the model template*	`my_model = lm.LinearRegression()`
2	Fit the model to the training data *Save the optimal model parameters*	`my_model.fit(X, y)`
3	Use fitted model to make predictions *Fitted model makes predictions for y*	`my_model.predict(X)`

To extract the fitted parameters: **`my_model.coef_`** and **`my_model.intercept_`**

Example for logistic regression: Iris Flower Dataset

- The dataset that contains the sepal and petal length and width of 150 iris flowers of three different species: *Iris setosa*, *Iris versicolor*, and *Iris virginica*.



From **Hands-on Machine Learning** book

Load the Dataset

```
from sklearn.datasets import load_iris

iris = load_iris(as_frame=True)
list(iris)
```

```
['data',
 'target',
 'frame',
 'target_names',
 'DESCR',
 'feature_names',
 'filename',
 'data_module']
```

```
iris.data.head(3)
```

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)
0	5.1	3.5	1.4	0.2
1	4.9	3.0	1.4	0.2
2	4.7	3.2	1.3	0.2

```
iris.target.head(3)  # note that the instances are not shuffled
```

```
0    0
1    0
2    0
Name: target, dtype: int32
```

```
iris.target_names
```

```
array(['setosa', 'versicolor', 'virginica'], dtype='<U10')
```

Build a Simple Classifier

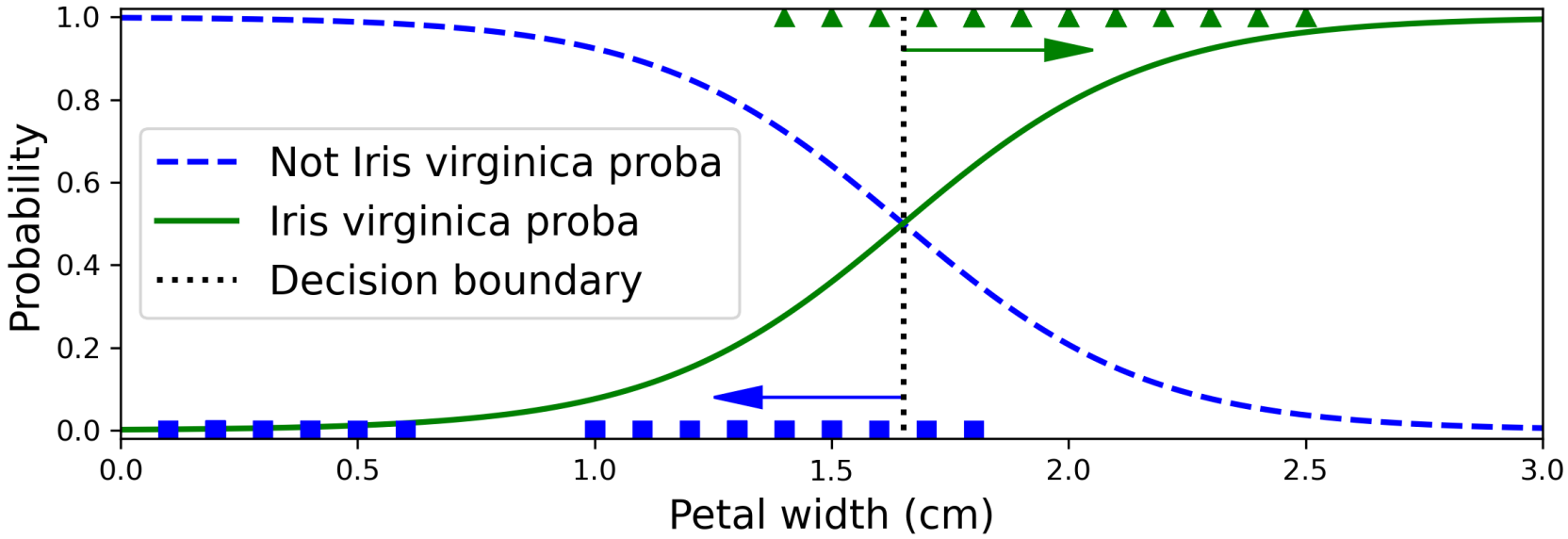
- Build a classifier to detect the *Iris virginica* type based only on the *petal width* feature:

```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split

X = iris.data[["petal width (cm)"]].values
y = iris.target_names[iris.target] == 'virginica'
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)

log_reg = LogisticRegression(random_state=42)
log_reg.fit(X_train, y_train)
```

Decision Boundaries

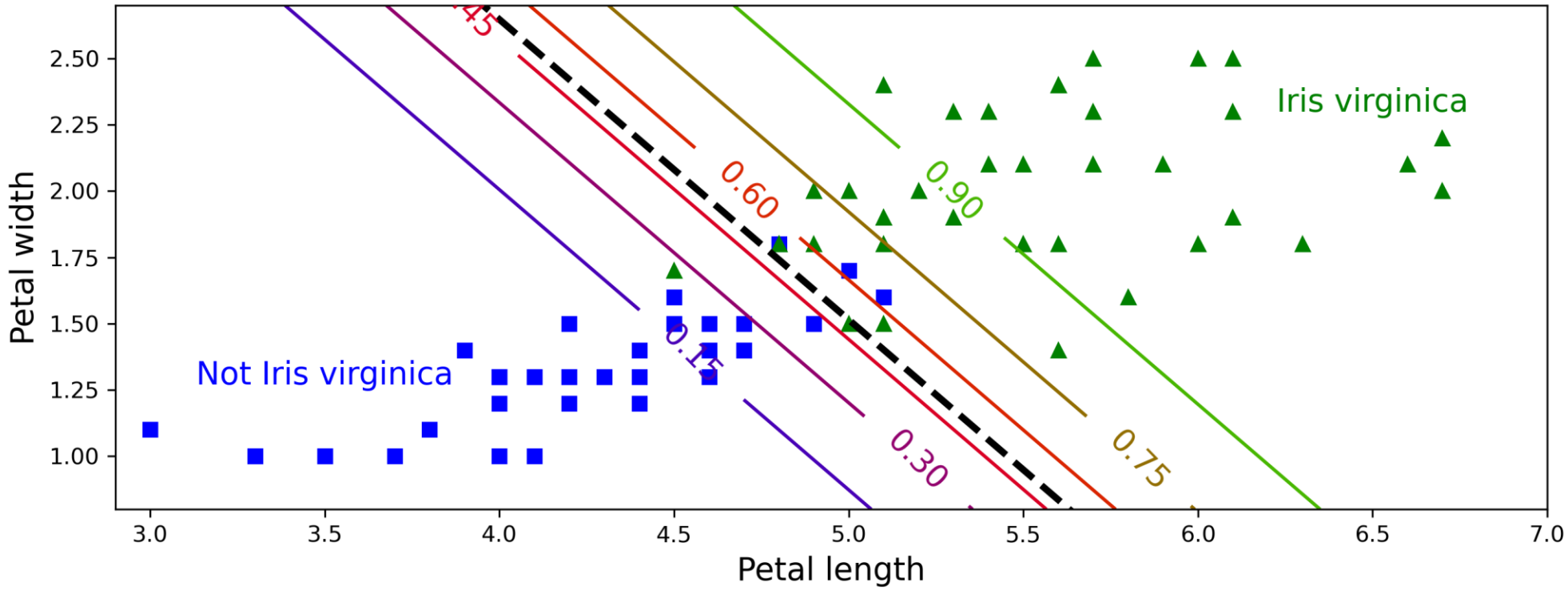
- The model's estimated probabilities for flowers with petal widths varying from 0 cm to 3 cm:

```
X_new = np.linspace(0, 3, 1000).reshape(-1, 1)  # reshape to get a column vector
y_proba = log_reg.predict_proba(X_new)
decision_boundary = X_new[y_proba[:, 1] >= 0.5][0, 0]
```



Decision Boundaries

- A logistic regression model to detect the *Iris virginica* type based on two features: *petal length* and *width*.



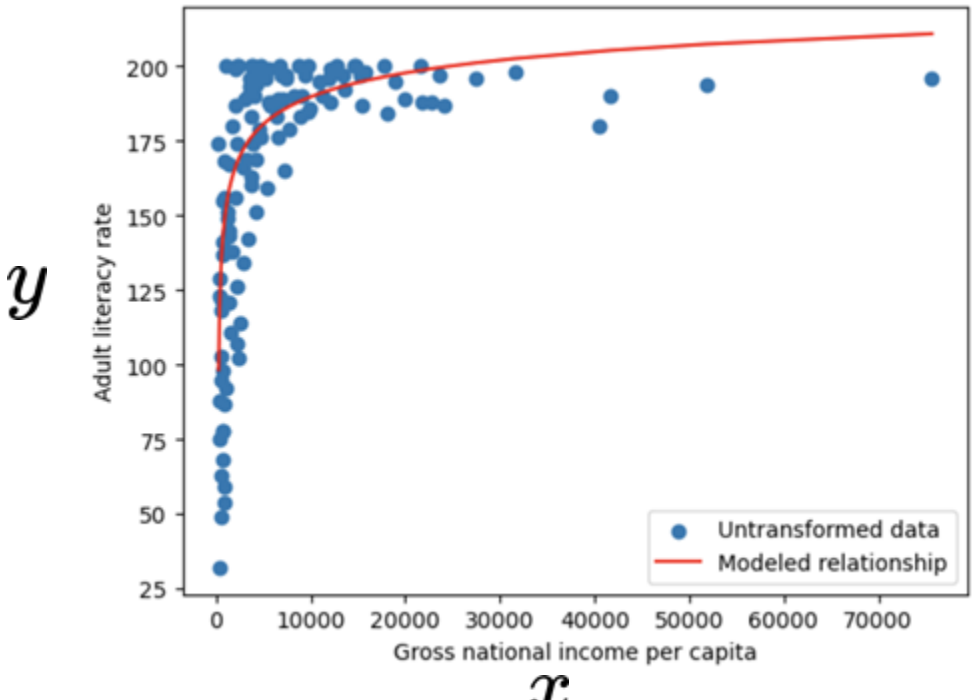
Feature Engineering

- sklearn
- **Feature Engineering**
- One-Hot Encoding
- Polynomial Features
- Complexity and Overfitting

Transforming Features

Two observations:

- Applying a transformation could help **linearize** a dataset
- In our work on modeling, we saw that linear **modeling works best** when our dataset has linear relationships



$$y^4 = m(\log x) + b$$

Putting ideas together:

Feature engineering = transforming features to improve model performance

Feature Engineering

Feature engineering is the process of transforming raw features into more informative features for use in modeling

Allows us to:

- Capture domain knowledge
- Express non-linear relationships using linear models
- Use non-numeric features in models

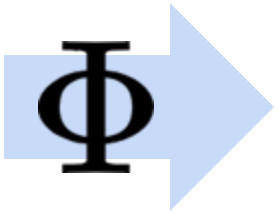
Feature Functions

A **feature function** describes the transformations we apply to raw features in the dataset to create transformed features. Often, the dimension of the *featurized* dataset increases.

Example: a feature function that adds a squared feature to the design matrix

	hp	mpg
0	130.00	18.00
1	165.00	15.00
2	150.00	18.00
...	...	...
395	84.00	32.00
396	79.00	28.00
397	82.00	31.00

392 rows × 2 columns



	hp	hp^2	mpg
0	130.00	16900.00	18.00
1	165.00	27225.00	15.00
2	150.00	22500.00	18.00
...	...	...	...
395	84.00	7056.00	32.00
396	79.00	6241.00	28.00
397	82.00	6724.00	31.00

392 rows × 3 columns

Dataset of raw features:

$$\mathbb{X} \in \mathbb{R}^{n \times p}$$

After applying the feature function Φ

$$\Phi(\mathbb{X}) \in \mathbb{R}^{n \times p'}$$

Feature Functions

A **feature function** describes the transformations we apply to raw features in the dataset to create transformed features. Often, the dimension of the *featurized* dataset increases.

Linear models trained on transformed data are sometimes written using the symbol Φ instead of X:

$$\hat{y} = \theta_0 + \theta_1 x + \theta_2 x^2$$
$$\hat{\mathbb{Y}} = \mathbb{X}\theta$$

$$\hat{y} = \theta_0 + \theta_1 \phi_1 + \theta_2 \phi_2$$
$$\hat{\mathbb{Y}} = \Phi\theta$$

Shorthand for “the design matrix after feature engineering”

One-Hot Encoding

- sklearn
- Feature Engineering
- **One-Hot Encoding**
- Polynomial Features
- Complexity and Overfitting

Regression Using Non-Numeric Features

Think back to the `tips` dataset we used when first exploring regression

	total_bill	size	day
0	16.99	2	Sun
1	10.34	3	Sun
2	21.01	3	Sun
3	23.68	2	Sun
4	24.59	4	Sun

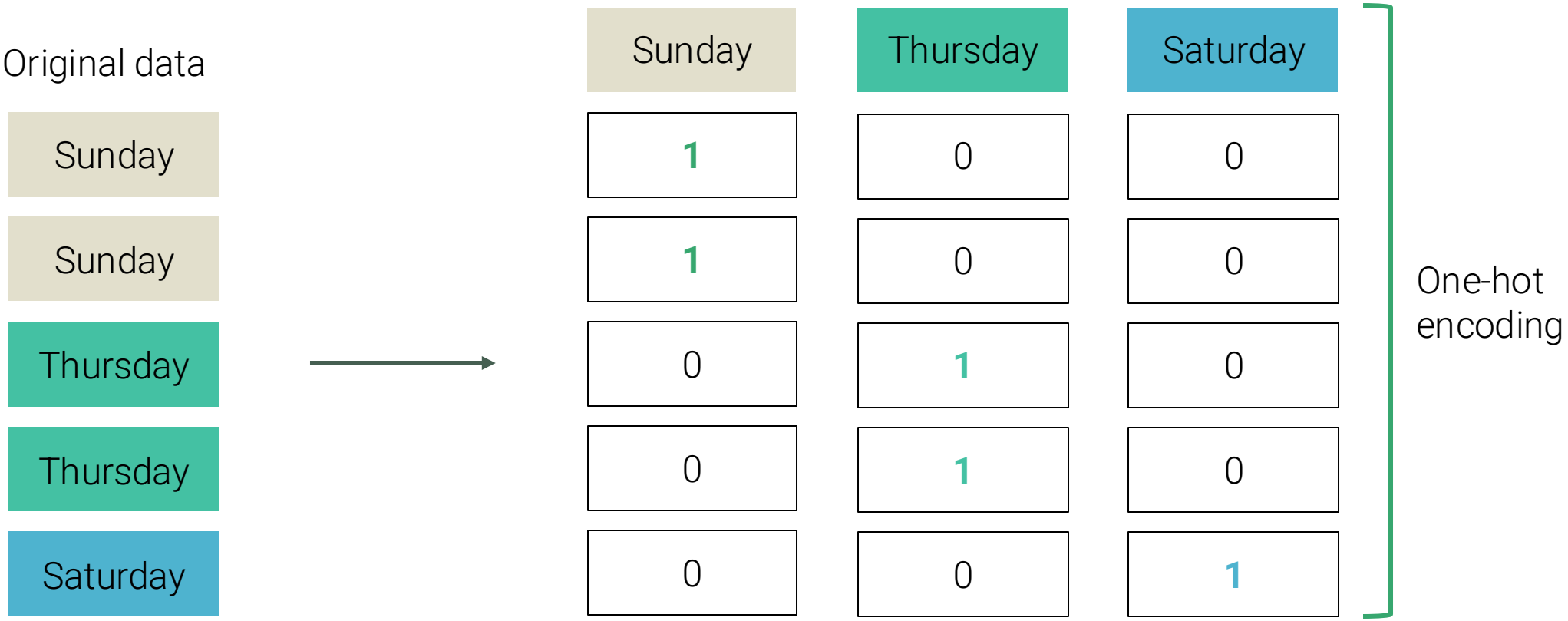
Before, we were limited to only using numeric features in a model – `total_bill` and `size`

By performing feature engineering, we can incorporate *non-numeric* features like the day of the week

One-hot Encoding

One-hot encoding is a feature engineering technique to transform non-numeric data into numeric features for modeling

- Each category of a categorical variable gets its own feature
  - Value = 1 if a row belongs to the category
  - Value = 0 otherwise



Regression Using the One-Hot Encoding

The one-hot encoded features can then be used in the design matrix to train a model

	total_bill	size	day_Fri	day_Sat	day_Sun	day_Thur
0	16.99	2	0.0	0.0	1.0	0.0
1	10.34	3	0.0	0.0	1.0	0.0
2	21.01	3	0.0	0.0	1.0	0.0
3	23.68	2	0.0	0.0	1.0	0.0
4	24.59	4	0.0	0.0	1.0	0.0

Raw features

One-hot encoded features

$$\hat{y} = \theta_1(\text{total\_bill}) + \theta_2(\text{size}) + \theta_3(\text{day\_Fri}) + \theta_4(\text{day\_Sat}) + \theta_5(\text{day\_Sun}) + \theta_6(\text{day\_Thur})$$

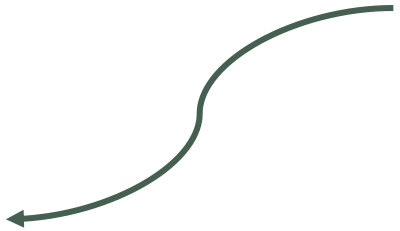
In shorthand: $\hat{y} = \theta_1\phi_1 + \theta_2\phi_2 + \theta_3\phi_3 + \theta_4\phi_4 + \theta_5\phi_5 + \theta_6\phi_6$

Regression Using the One-Hot Encoding

Using `sklearn` to fit the new model:

$$\hat{y} = \theta_1(\text{total\_bill}) + \theta_2(\text{size}) + \theta_3(\text{day\_Fri}) + \theta_4(\text{day\_Sat}) + \theta_5(\text{day\_Sun}) + \theta_6(\text{day\_Thur})$$

Feature	Model Coefficient
total_bill	0.092994
size	0.187132
day_Fri	0.745787
day_Sat	0.621129
day_Sun	0.732289
day_Thur	0.668294



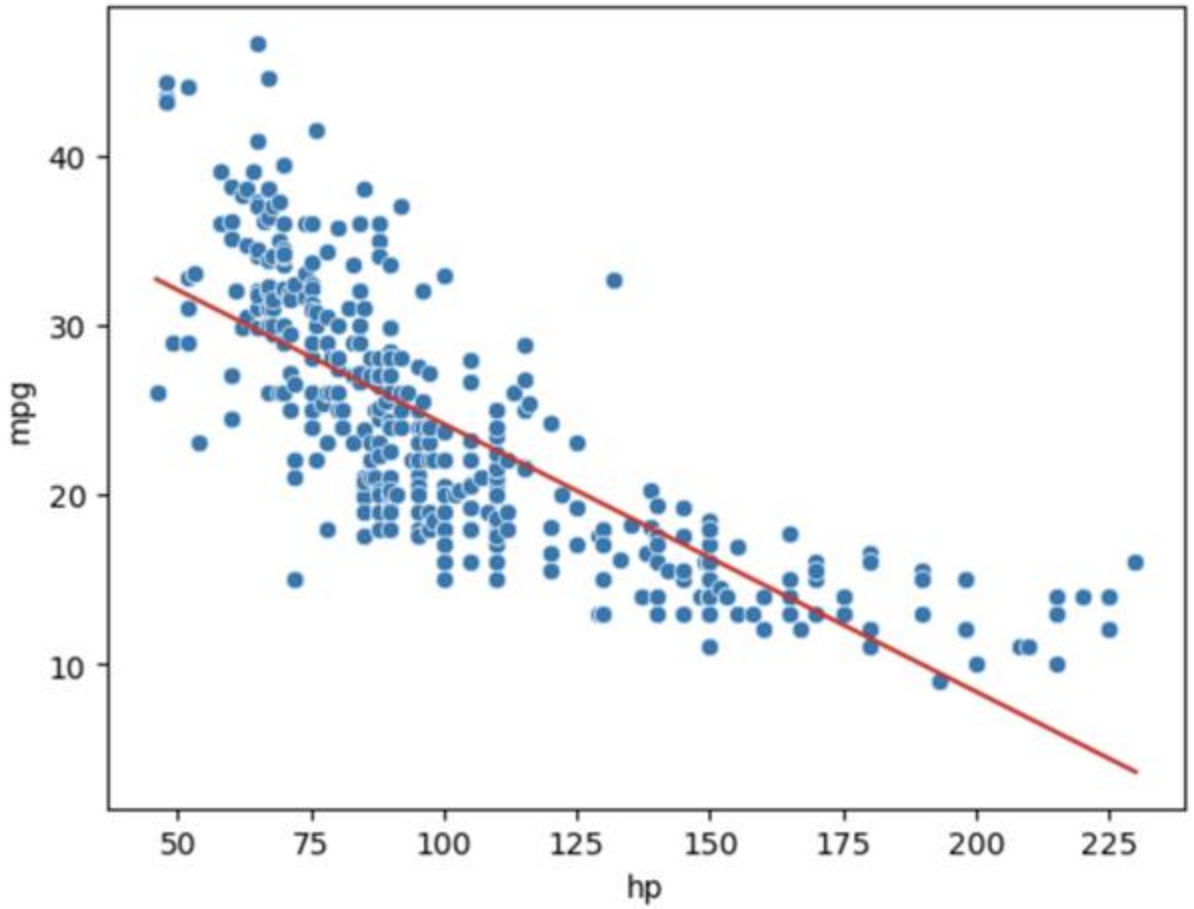
Interpretation: how much the fact that it is Friday impacts the predicted tip

Polynomial Features

- sklearn
- Feature Engineering
- One-Hot Encoding
- **Polynomial Features**
- Complexity and Overfitting

Accounting for Curvature

We've seen a few cases now where models with linear features have performed poorly on datasets with a clear non-linear curve.



$$\hat{y} = \theta_0 + \theta_1(\text{hp})$$

MSE: 23.94

When our model uses only a single linear feature (**hp**), it cannot capture non-linearity in the relationship

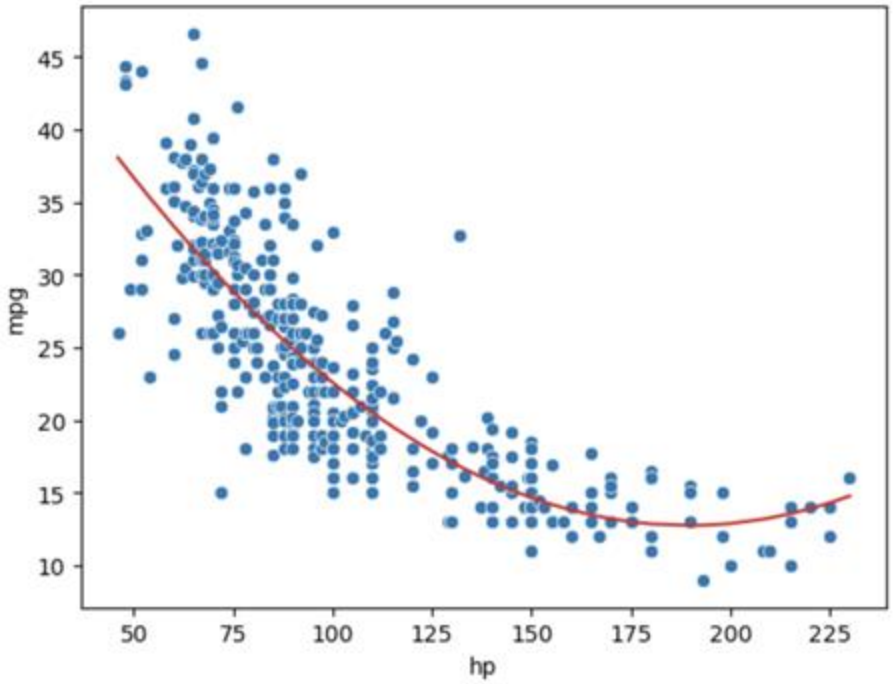
Solution: incorporate a non-linear feature!

Polynomial Features

We create a new feature: the square of the **hp**

$$\hat{y} = \theta_0 + \theta_1(\text{hp}) + \theta_2(\text{hp}^2)$$

This is still a **linear model**. Even though there are non-linear *features*, the model is linear with respect to θ



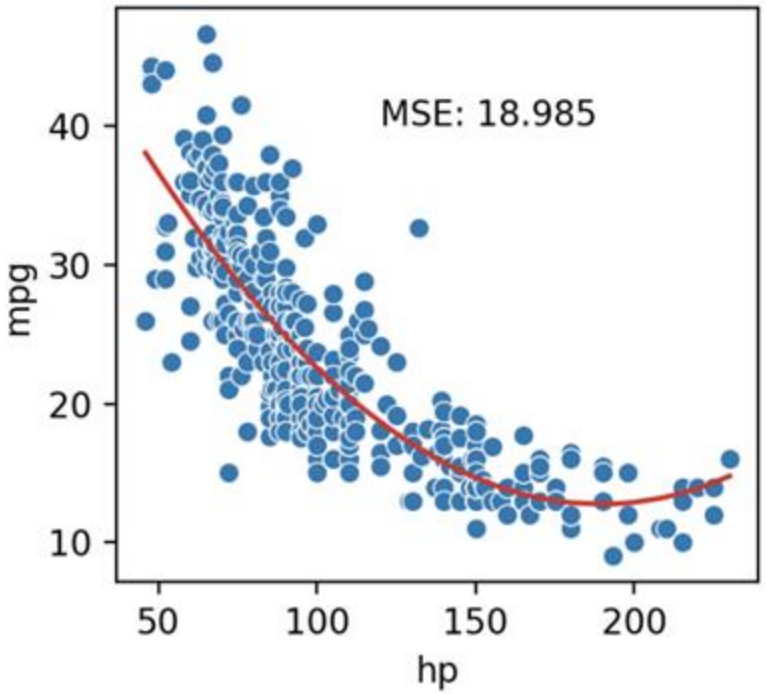
Degree of model: 2
MSE: 18.98

Looking a lot better: our predictions capture the curvature of the data.

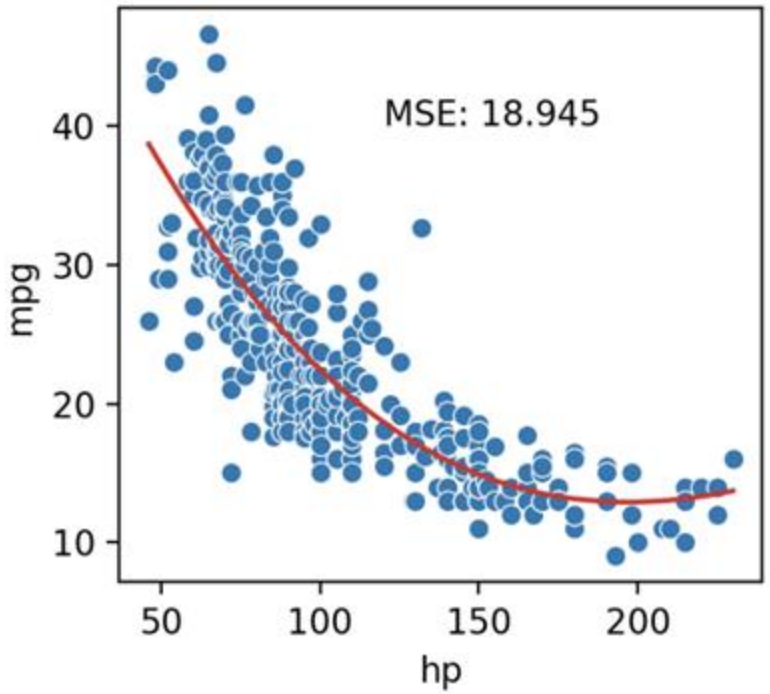
Polynomial Features

What if we add more polynomial features?

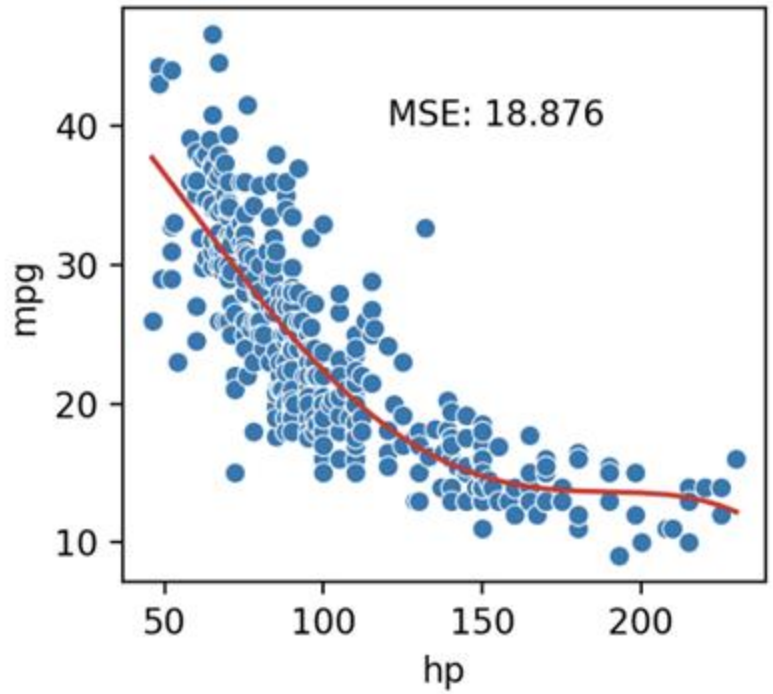
$\hat{y} = \theta_0 + \theta_1(\text{hp}) + \theta_2(\text{hp}^2)$



$\hat{y} = \theta_0 + \theta_1(\text{hp}) + \theta_2(\text{hp}^2) + \theta_3(\text{hp}^3)$



$\hat{y} = \theta_0 + \theta_1(\text{hp}) + \theta_2(\text{hp}^2) + \theta_3(\text{hp}^3) + \theta_4(\text{hp}^4)$

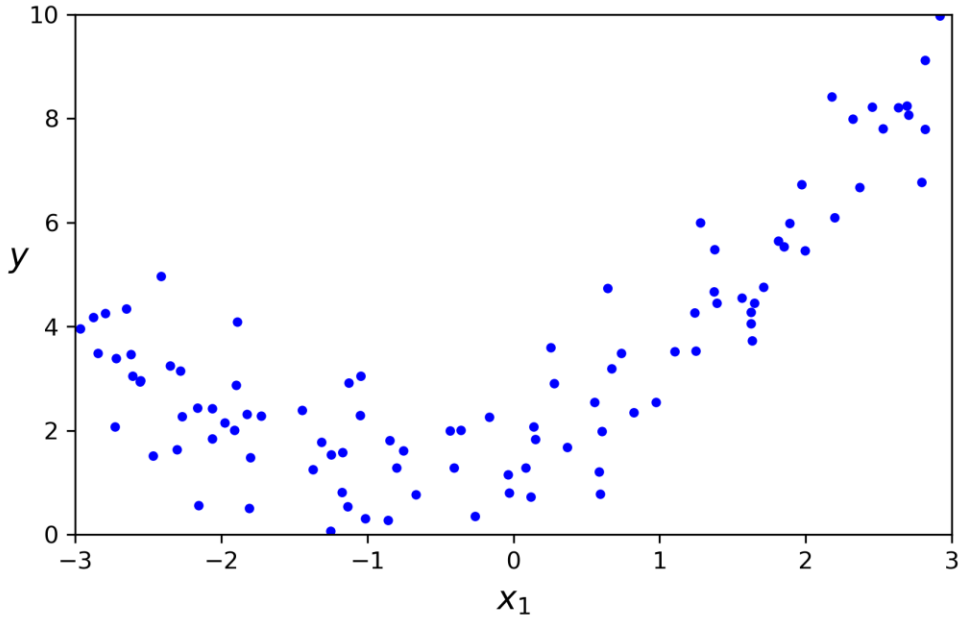


MSE continues to decrease with each additional polynomial term

Polynomial Regression in sklearn

- *Polynomial regression*: add powers of each feature as new features, then train a linear model on this extended set of features.
- *Example:* generate nonlinear data, based on a quadratic equation:

```
np.random.seed(42)
m = 100
X = 6 * np.random.rand(m, 1) - 3
y = 0.5 * X ** 2 + X + 2 + np.random.randn(m, 1)
```



Polynomial Features in sklearn

- Use ScikitLearn's `PolynomialFeatures` class to transform our training data, adding the square of each feature in the training set:

```
from sklearn.preprocessing import PolynomialFeatures

poly_features = PolynomialFeatures(degree=2, include_bias=False)
X_poly = poly_features.fit_transform(X)
X[0]
```

```
array([-0.75275929])
```

```
X_poly[0]
```

```
array([-0.75275929,  0.56664654])
```

- Fit a `LinearRegression` model to this extended training data:

```
lin_reg = LinearRegression()
lin_reg.fit(X_poly, y)
lin_reg.intercept_, lin_reg.coef_
```

```
(array([1.78134581]), array([[0.93366893, 0.56456263]]))
```

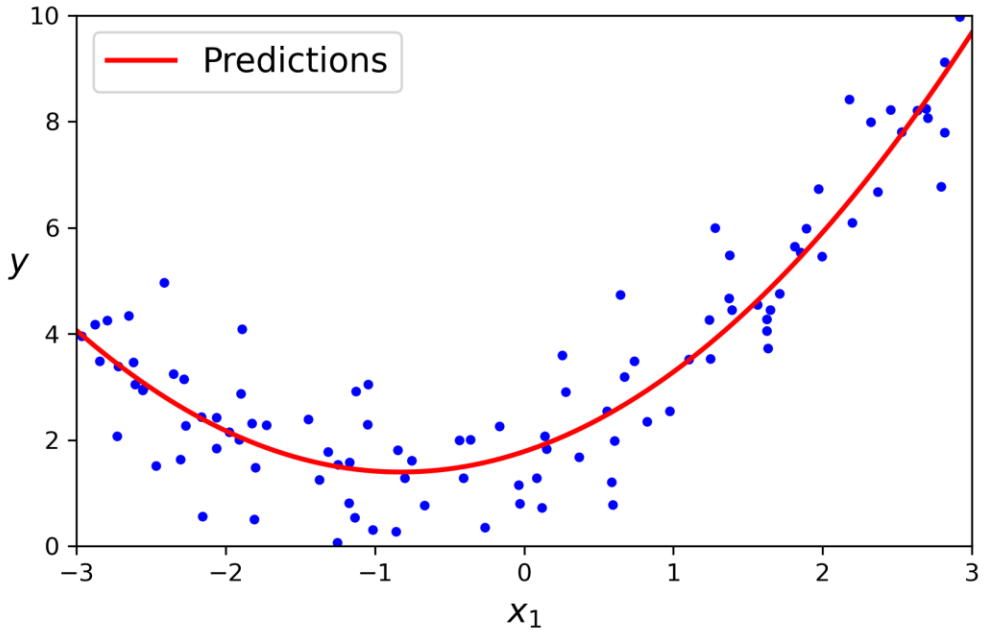
Polynomial Regression

- Fit a `LinearRegression` model to this extended training data:

```
lin_reg = LinearRegression()
lin_reg.fit(X_poly, y)
lin_reg.intercept_, lin_reg.coef_
```

```
(array([1.78134581]), array([[0.93366893, 0.56456263]]))
```

- The model estimates $\hat{y} = 0.56x_1^2 + 0.93x_1 + 1.78$ when in fact the original function was $y = 0.5x_1^2 + 1.0x_1 + 2.0 +$ Gaussian noise.



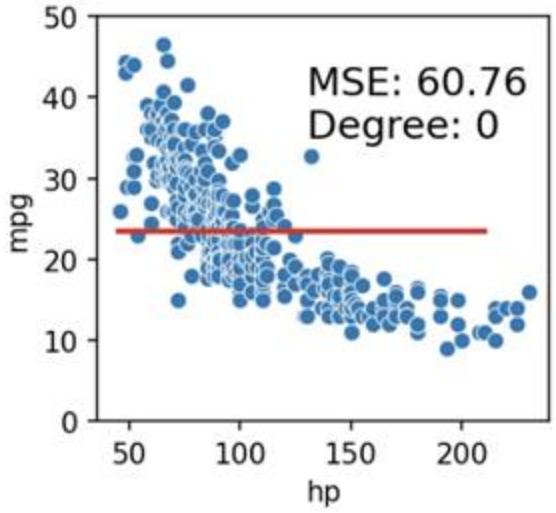
Relationship Between Features

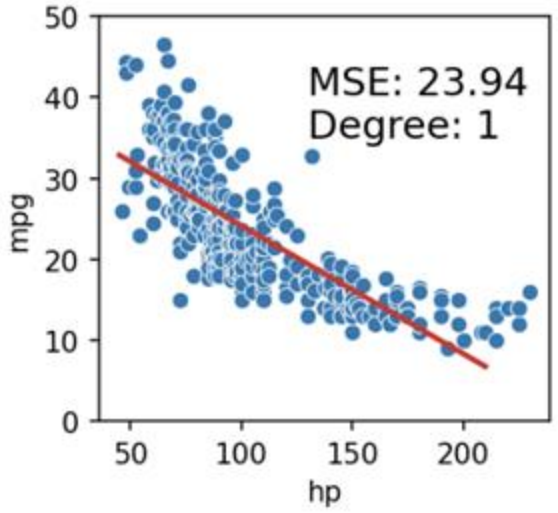
- When there are multiple features, polynomial regression is capable of finding relationships between features.
- The reason is `PolynomialFeatures` also adds all combinations of features up to the given degree.
- Example. if there are two features a and b, `PolynomialFeatures` with degree = 3 would add features $a^2, b^2, a^3, b^3, ab\ , a^2b\ , ab^2$.

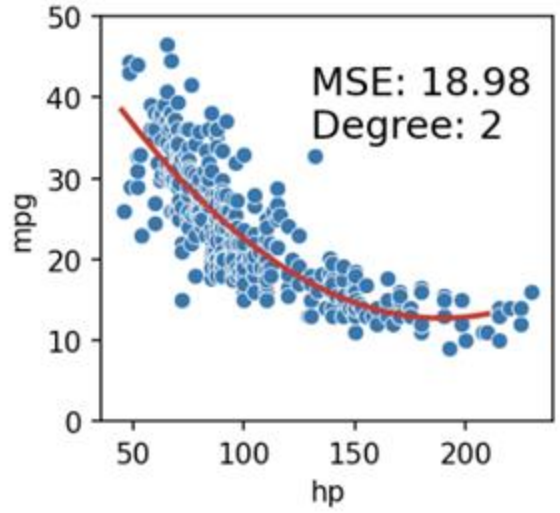
Polynomial Features

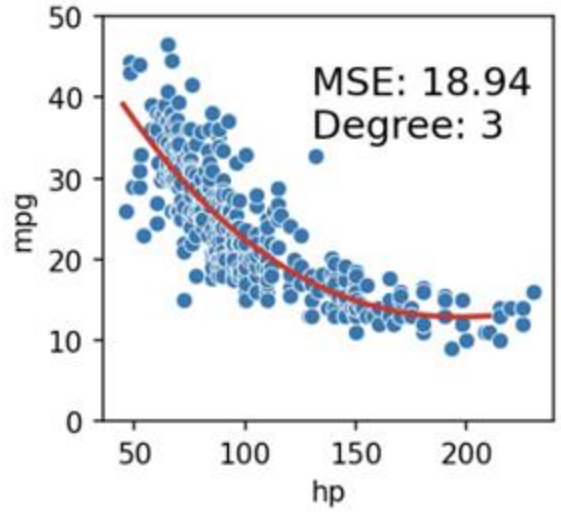
- sklearn
- Feature Engineering
- One-Hot Encoding
- Polynomial Features
- **Complexity and Overfitting**

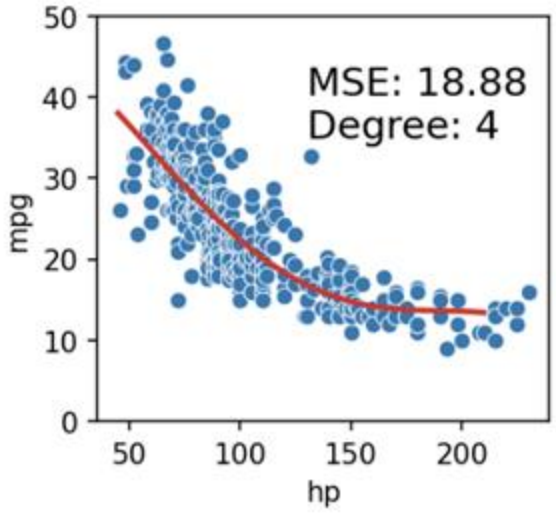
How Far Can We Take This?

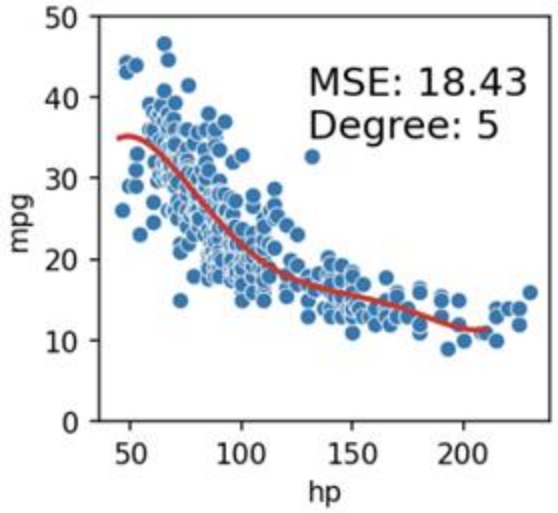


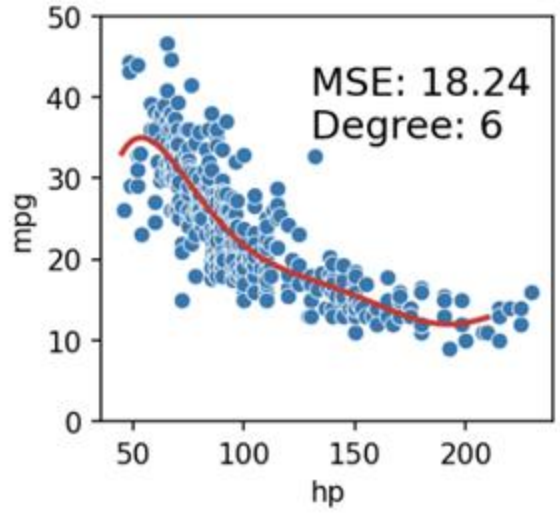


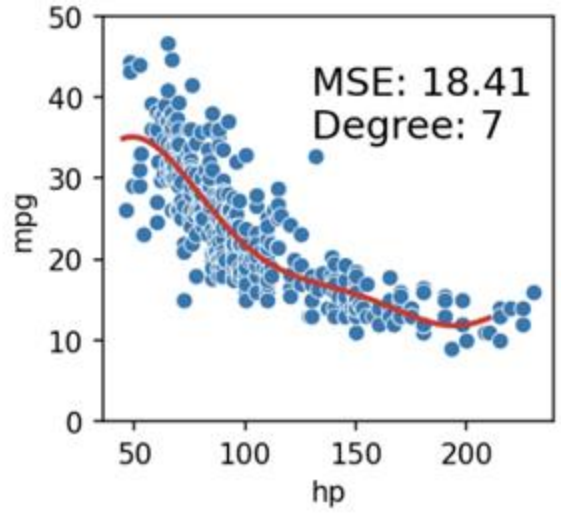












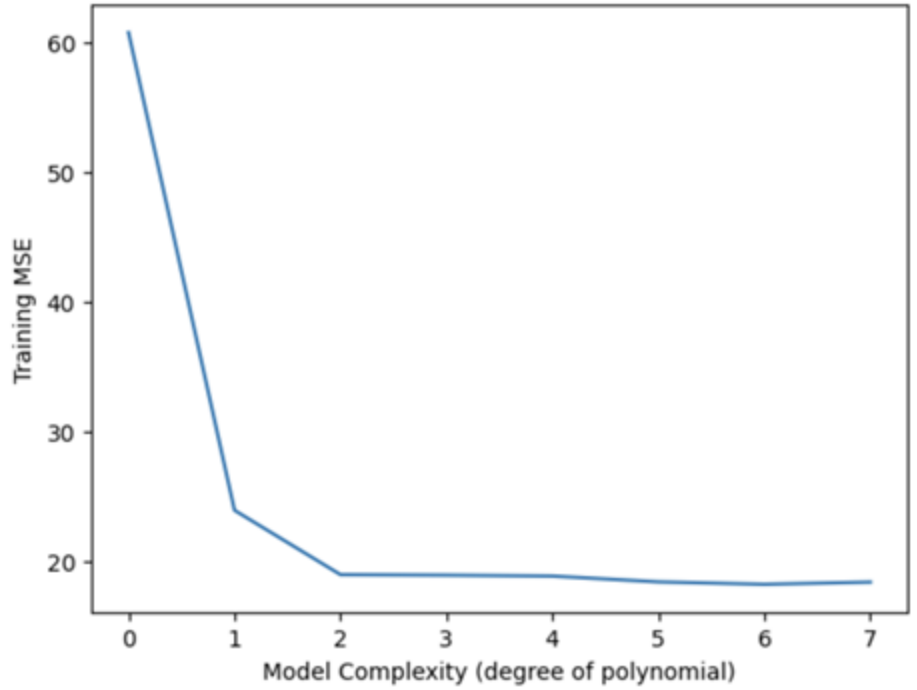
Which higher-order polynomial model do you think fits best?

Model Complexity

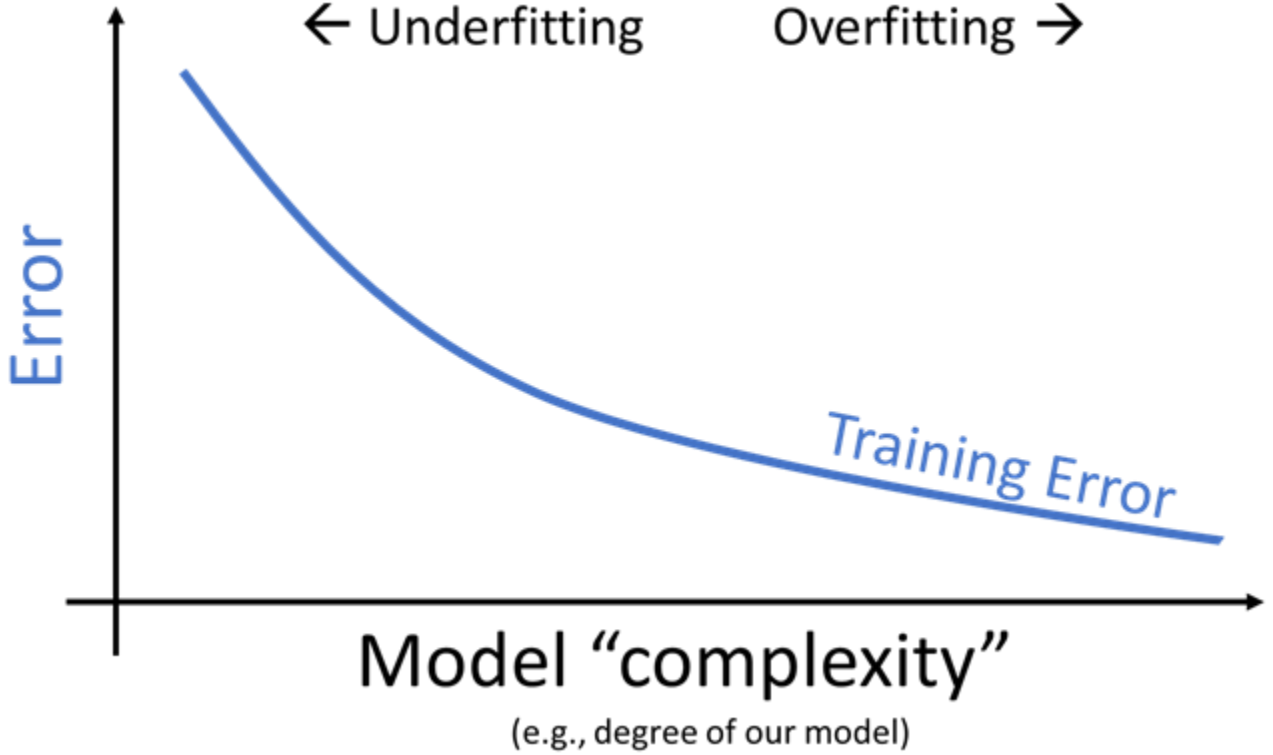
As we continue to add more and more polynomial features, the MSE continues to decrease

Equivalently: as the **model complexity** increases, its *training error* decreases

Our experiment using `vehicles`



General trend for an arbitrary dataset

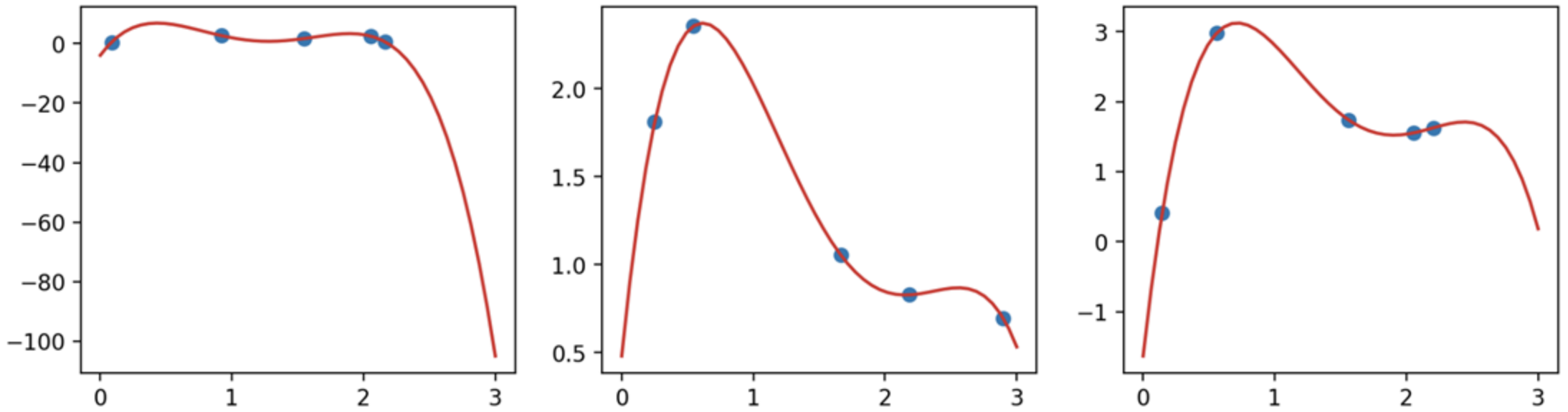


Seems like a good deal?

An Extreme Example: Perfect Polynomial Fits

Math fact: given N non-overlapping data points, we can always find a polynomial of degree N-1 that goes through all those points.

For example, there always exists a degree-4 polynomial curve that can perfectly model a dataset of 5 datapoints



Model Performance on Unseen Data

Our `vehicle` models from before considered a somewhat artificial scenario – we trained the models on the *entire* dataset, then evaluated their ability to make predictions on this same dataset

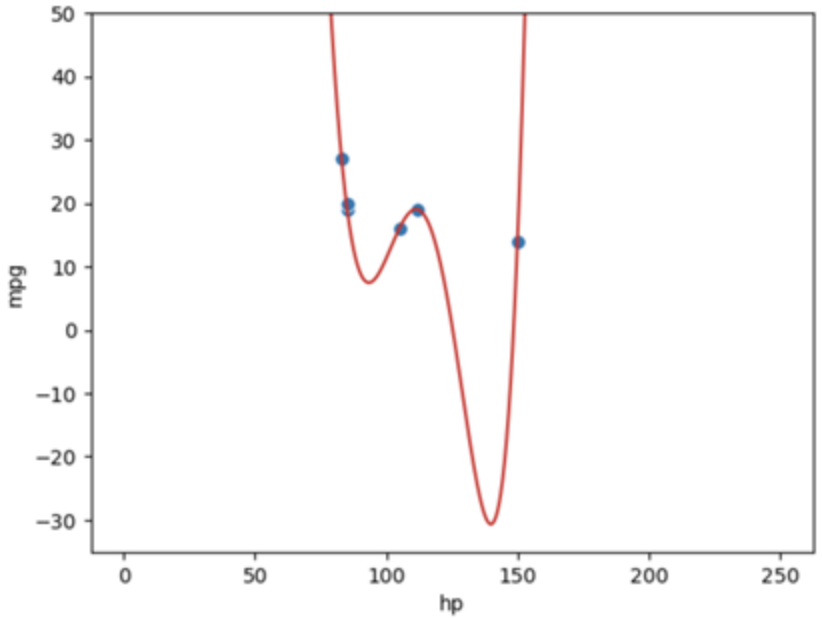
More realistic situation: we train the model on a *sample* from the population, then use it to make predictions on data it didn't encounter during training

Model Performance on Unseen Data

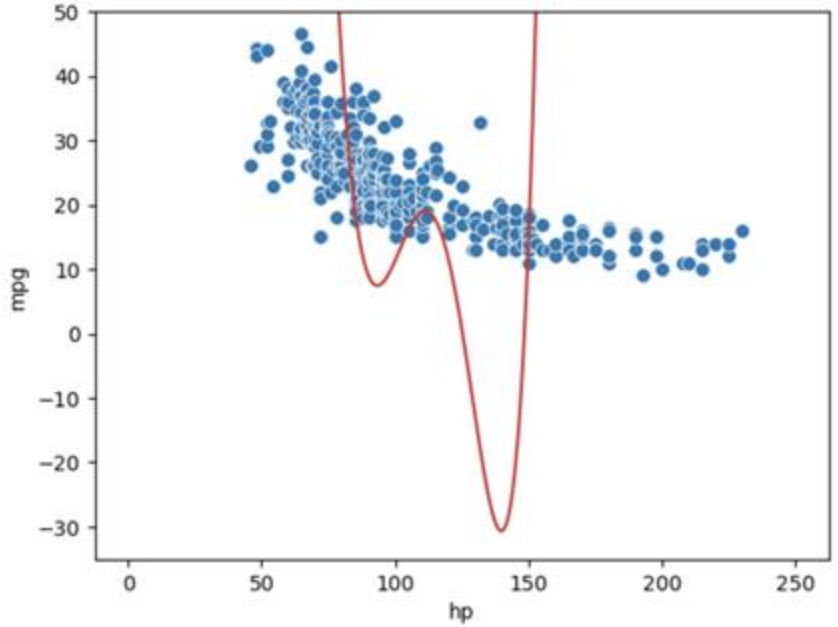
New (more realistic) example:

- We are given a training dataset of just 6 datapoints
- We want to train a model to then make predictions on a *different* set of points

We may be tempted to make a highly complex model (eg degree 5)



Complex model makes perfect predictions on the training data…



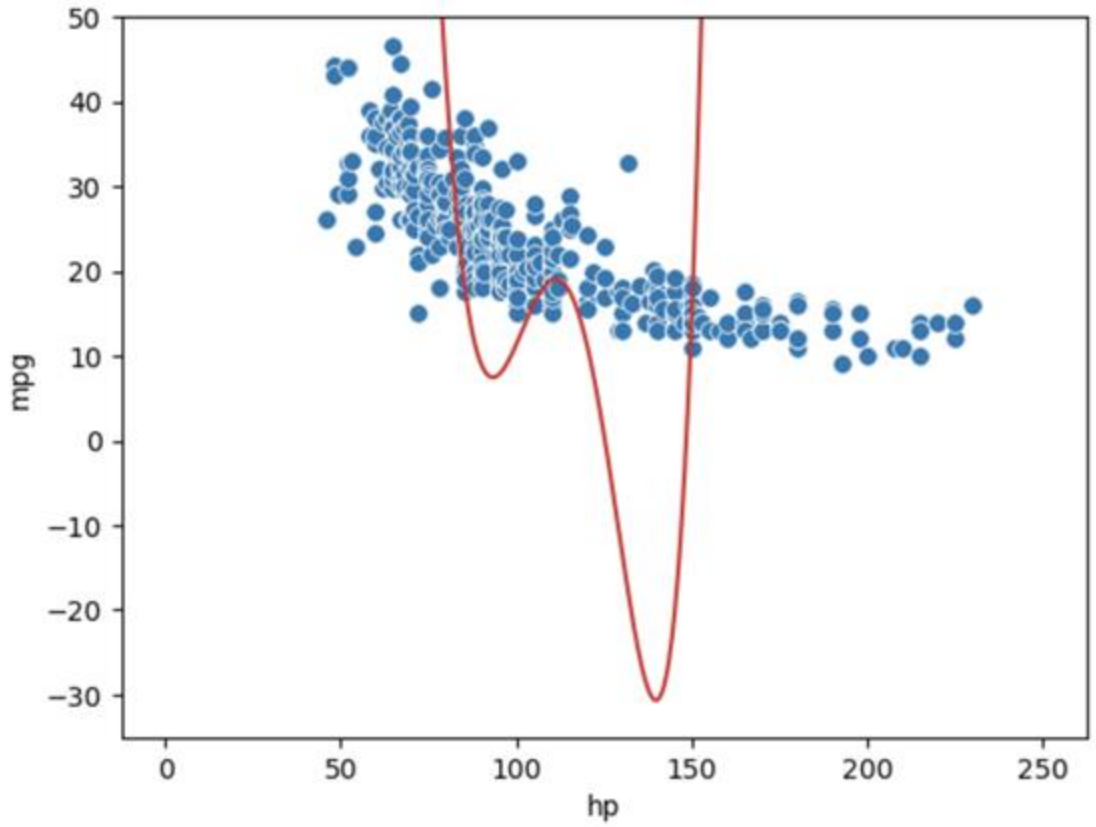
…but performs *horribly* on the rest of the population!

Model Performance on Unseen Data

What went wrong?

- The complex model **overfit** to the training data – it essentially "memorized" these 6 training points
- The overfitted model does not **generalize** well to data it did not encounter during training

This is a problem: we want models that are generalizable to "unseen" data

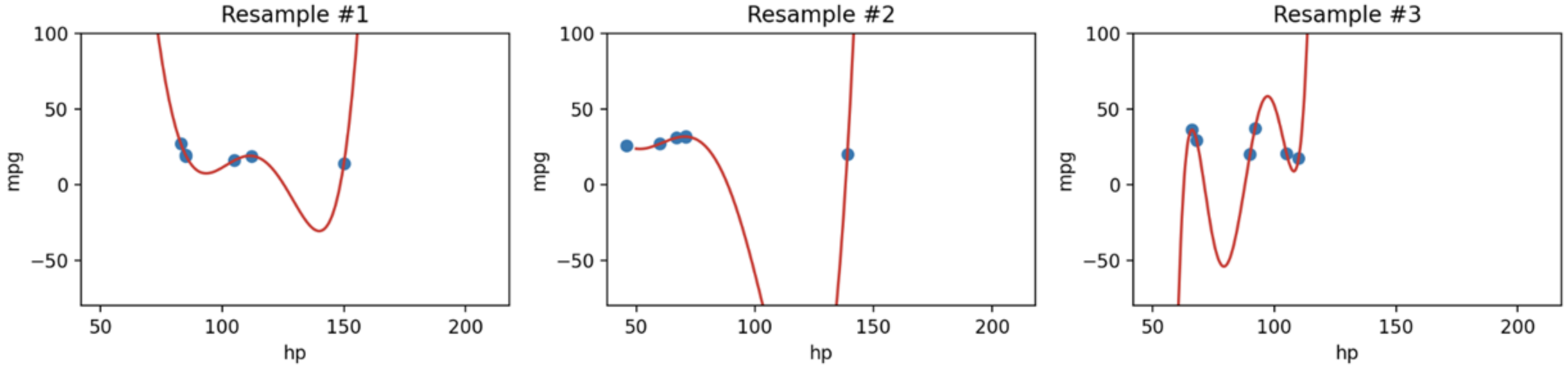


Model Variance

Complex models are sensitive to the specific dataset used to train them – they have high **variance**, because they will *vary* depending on what datapoints are used for training them

`vehicles`

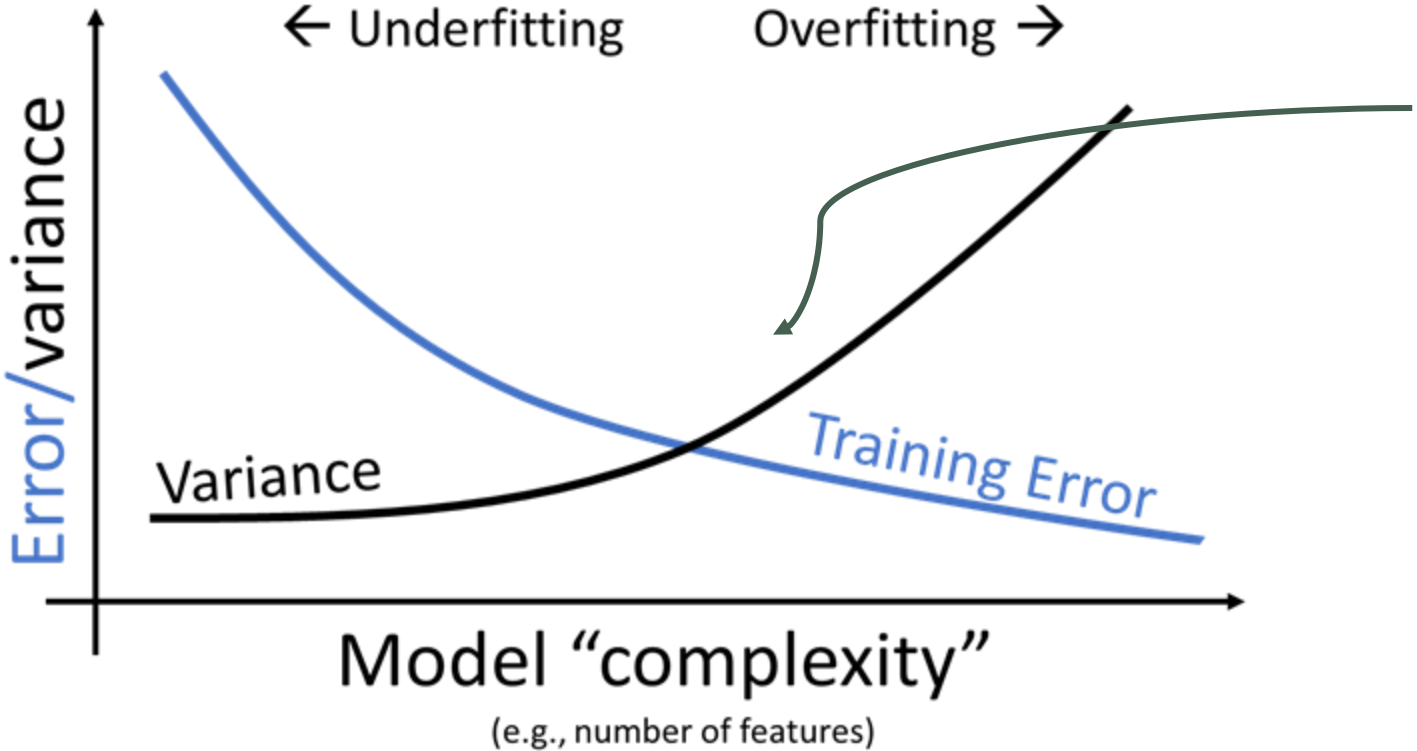
Our degree-5 model varies erratically when we fit it to different samples of



Error, Variance, and Complexity

We face a dilemma:

- We know that we can **decrease training error** by increasing model complexity
- However, models that are *too* complex start to overfit and do not generalize well – their **high variance** means they can't be reapplied to new datasets



Our goal: find this "sweet spot"