Linear Regression

Some slides were retrieved from this link and
Some images were retrieved from Aurélien Géron - Hands-on Machine Learning with Scikit-Learn and TensorFlow (2017, O'Reilly)

Modeling bivariate data as a function + noise

Ingredients

- Bivariate data $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$.
- Model: $y_i = f(x_i) + E_i$

  where $f(x)$ is some function, E_i random error.
- Total squared error: $\displaystyle\sum_{i=1}^{n} E_i^2 = \sum_{i=1}^{n} (y_i - f(x_i))^2$

Model allows us to predict the value of y for any given value of x.

- x is called the independent or predictor variable.
- y is the dependent or response variable.

Examples of $f(x)$

lines	$y = ax + b + E$
polynomials	$y = ax^2 + bx + c + E$
other:	$y = a/x + b + E$
other:	$y = a \sin(x) + b + E$

Simple linear regression: finding the best fitting line

- Bivariate data $(x_1, y_1), \ldots, (x_n, y_n)$.
- Simple linear regression: fit a line to the data

$$y_i = ax_i + b + E_i, \quad \text{where} \quad E_i \sim \mathrm{N}(0, \sigma^2)$$

  and where σ is a fixed value, the same for all data points.

- Total squared error: $\displaystyle\sum_{i=1}^{n} E_i^2 = \sum_{i=1}^{n} (y_i - ax_i - b)^2$
- Goal: Find the values of a and b that give the 'best fitting line'.
- Best fit: (least squares)
  The values of a and b that minimize the total squared error.

Linear Regression: finding the best fitting polynomial

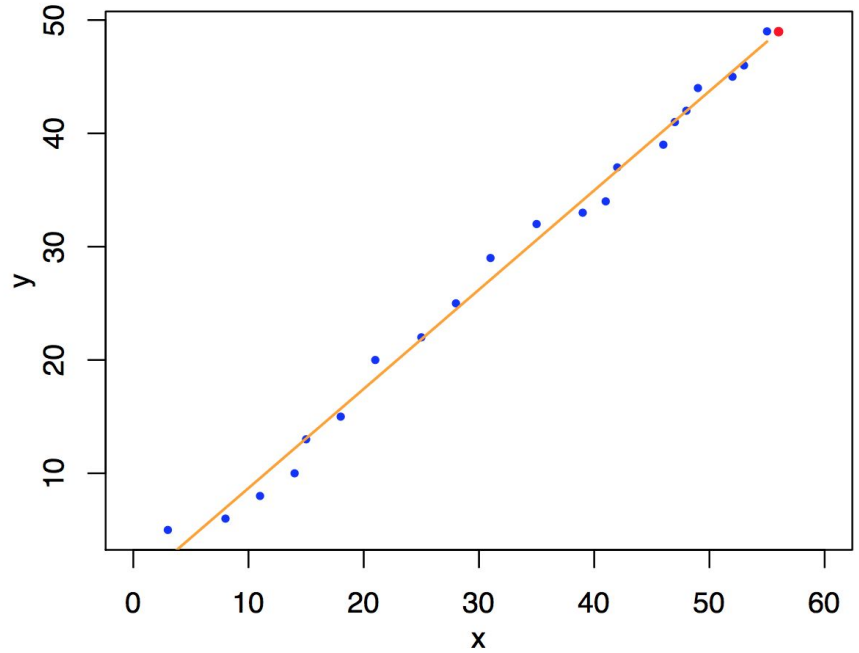
- Bivariate data: $(x_1, y_1), \ldots, (x_n, y_n)$.
- Linear regression: fit a parabola to the data

$$y_i = ax_i^2 + bx_i + c + E_i, \quad \text{where} \quad E_i \sim \mathrm{N}(0, \sigma^2)$$

  and where σ is a fixed value, the same for all data points.
- Total squared error: $\displaystyle\sum_{i=1}^{n} E_i^2 = \sum_{i=1}^{n} (y_i - ax_i^2 - bx_i - c)^2$.
- Goal:
  Find the values of a, b, c that give the 'best fitting parabola'.
- Best fit: (least squares)
  The values of a, b, c that minimize the total squared error.

  Can also fit higher order polynomials.

Stamps



Stamp cost (cents) vs. time (years since 1960)
(Red dot = 49 cents is predicted cost in 2016.)

(Actual cost of a stamp dropped from 49 to 47 cents on 4/8/16.)

What is linear about linear regression?

Linear in the parameters $a, b, \ldots$.

$$y = ax + b.$$

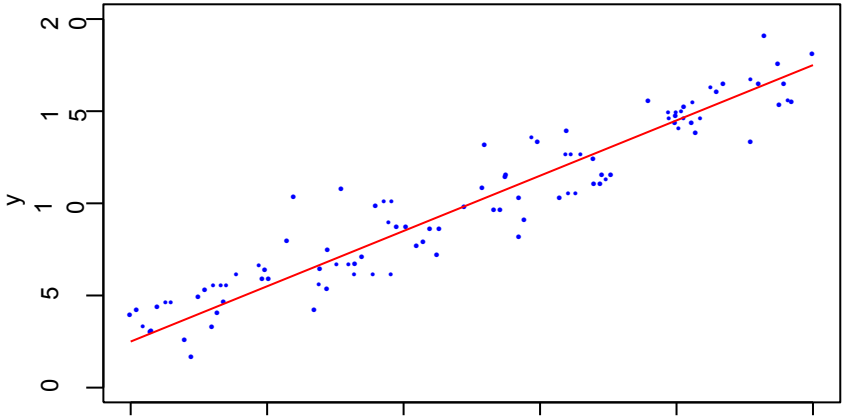
$$y = ax^2 + bx + c.$$

It is **not** because the curve being fit has to be a straight line –although this is the simplest and most common case.

Fitting a line is called simple linear regression.

Homoscedastic

BIG ASSUMPTIONS: the E_i are independent with the same variance σ^2.



Multiple Linear Regression

Simple linear regression : a single independent variable is used to predict the value of a dependent variable.

Equation : **y=A+BX**

Multiple linear regression : two or more independent variables are used to predict the value of a dependent variable. The difference between the two is the number of independent variables.

Equation : **y=A+BX1+CX2+DX3**

Multiple Linear Regression

More generally, a linear model makes a prediction by simply computing a weighted sum of the input features, plus a constant called the *bias term* (also called the *intercept term*), as shown in Equation 4-1.

Equation 4-1. Linear Regression model prediction

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

- $\hat{y}$ is the predicted value.
- n is the number of features.
- x_i is the i^{th} feature value.
- θ_j is the j^{th} model parameter (including the bias term θ_0 and the feature weights $\theta_1, \theta_2, \cdots, \theta_n$).

Multiple Linear Regression

This can be written much more concisely using a vectorized form, as shown in Equation 4-2.

Equation 4-2. Linear Regression model prediction (vectorized form)

$$\hat{y} = h_{\theta}(\mathbf{x}) = \theta^{T} \cdot \mathbf{x}$$

- θ is the model's *parameter vector*, containing the bias term θ_0 and the feature weights θ_1 to θ_n.
- θ^T is the transpose of θ (a row vector instead of a column vector).
- $\mathbf{x}$ is the instance's *feature vector*, containing x_0 to x_n, with x_0 always equal to 1.
- $\theta^T \cdot \mathbf{x}$ is the dot product of θ^T and $\mathbf{x}$.
- h_θ is the hypothesis function, using the model parameters θ.

How to Train the Data

We need to find θ which minimizes the cost function (RMSE or MSE).

$$\mathrm{MSE}(\mathbf{X}, h_\theta) = \frac{1}{m} \sum_{i=1}^{m} \left(\theta^T \cdot \mathbf{x}^{(i)} - y^{(i)} \right)^2$$

How to Find Values of θ

1. Using Normal Equation (Enclosed Form)

2. Using Gradient Descent

1) Using Normal Equation

To find the value of θ that minimizes the cost function, there is a *closed-form solution*—in other words, a mathematical equation that gives the result directly. This is called the *Normal Equation* (Equation 4-4).[2]

Equation 4-4. Normal Equation

$$\hat{\theta} = \left(\mathbf{X}^T \cdot \mathbf{X}\right)^{-1} \cdot \mathbf{X}^T \cdot \mathbf{y}$$

- $\hat{\theta}$ is the value of θ that minimizes the cost function.
- $\mathbf{y}$ is the vector of target values containing $y^{(1)}$ to $y^{(m)}$.

1) Using Normal Equation

Computational Complexity

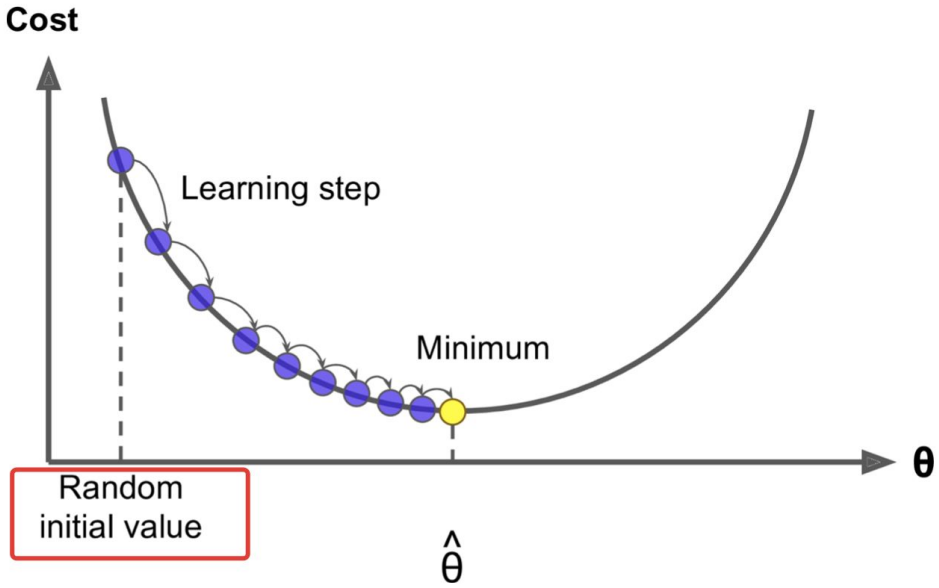
The Normal Equation computes the inverse of $\mathbf{X}^T \cdot \mathbf{X}$, which is an $n \times n$ matrix (where n is the number of features). The *computational complexity* of inverting such a matrix is typically about $O(n^{2.4})$ to $O(n^3)$ (depending on the implementation). In other words, if you double the number of features, you multiply the computation time by roughly $2^{2.4} = 5.3$ to $2^3 = 8$.

The Normal Equation gets very slow when the number of features grows large (e.g., 100,000).

On the positive side, this equation is linear with regards to the number of instances in the training set (it is $O(m)$), so it handles large training sets efficiently, provided they can fit in memory.

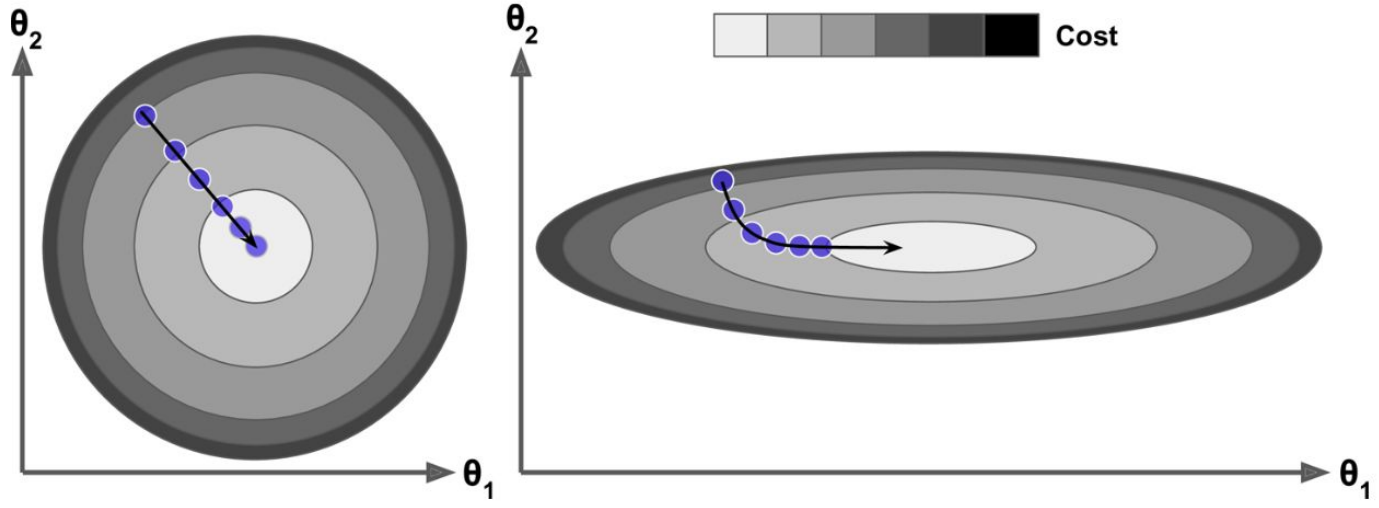
2) Gradient Descent

Gradient Descent is a very generic optimization algorithm capable of finding optimal solutions to a wide range of problems. The general idea of Gradient Descent is to tweak parameters iteratively in order to minimize a cost function.



2) Gradient Descent

Gradient Descent with and without feature scaling



When using Gradient Descent, you should ensure that all features have a similar scale (e.g., using Scikit-Learn's `StandardScaler` class), or else it will take much longer to converge.

2) Gradient Descent

There are three versions of Gradient Descent:

A. Batch Gradient Descent

B. Stochastic Gradient Descent

C. Mini-batch Gradient Descent

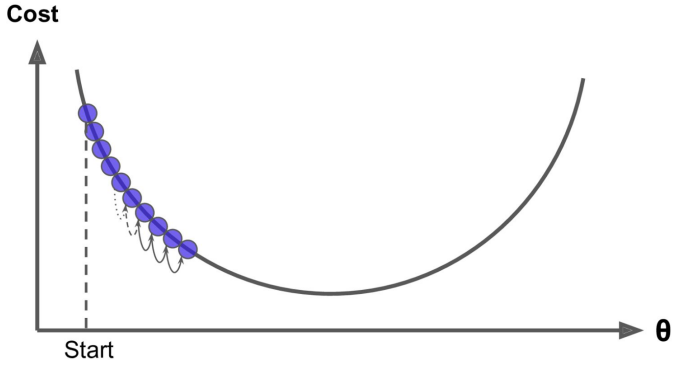
A) Batch Gradient Descent

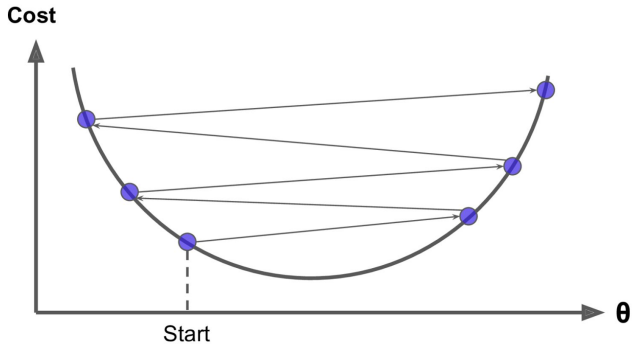
$$\nabla_\theta \mathrm{MSE}(\theta) = \begin{pmatrix} \frac{\partial}{\partial \theta_0}\mathrm{MSE}(\theta) \\ \frac{\partial}{\partial \theta_1}\mathrm{MSE}(\theta) \\ \vdots \\ \frac{\partial}{\partial \theta_n}\mathrm{MSE}(\theta) \end{pmatrix} = \frac{2}{m}\mathbf{X}^T \cdot (\mathbf{X} \cdot \theta - \mathbf{y})$$

$$\theta^{(\text{next step})} = \theta - \eta \nabla_\theta \mathrm{MSE}(\theta)$$

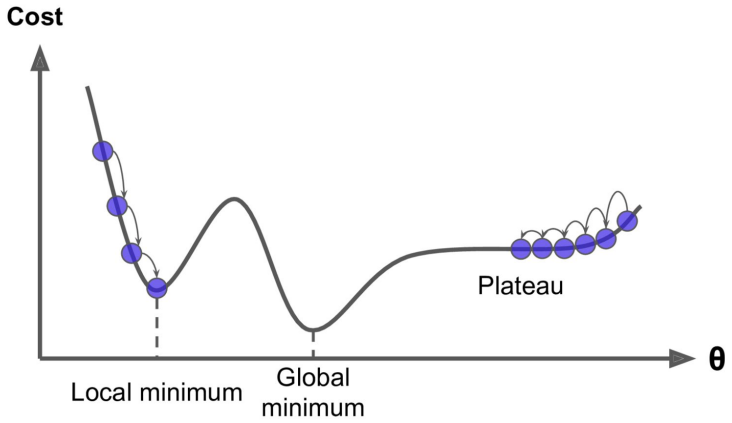
- Notice that this formula involves calculations over the full training set X, at each Gradient Descent step! As a result it is terribly slow on very large training sets.
- However, Gradient Descent scales well with the number of features; training a Linear Regression model when there are hundreds of thousands of features is much faster using Gradient Descent than using the Normal Equation.
- “Eta” is the “learning rate”

A) Batch Gradient Descent

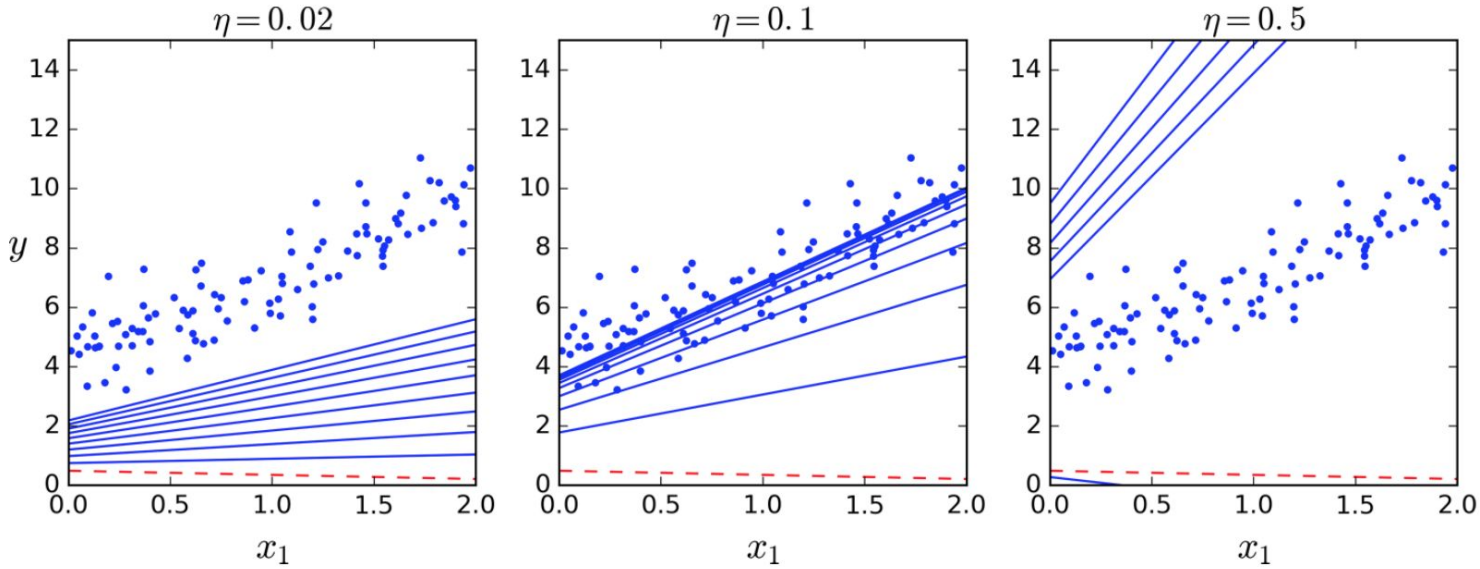




When learning rate is too small and too large



A) Batch Gradient Descent



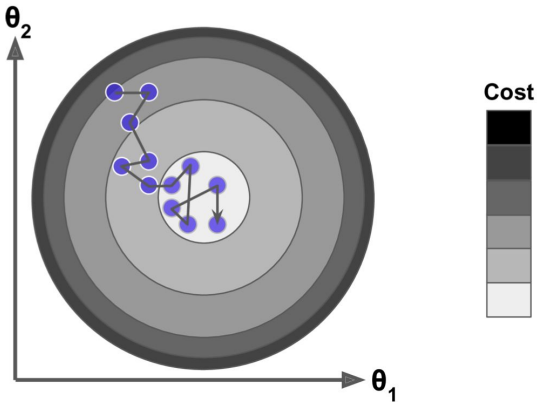
To find a good learning rate, you can use grid search. However, you may want to limit the number of iterations so that grid search can eliminate models that take too long to converge.

B) Stochastic Gradient Descent

- Stochastic Gradient Descent just picks a random instance in the training set at every step and computes the gradients based only on that single instance.

- Obviously this makes the algorithm much faster since it has very little data to manipulate at every iteration. It also makes it possible to train on huge training sets, since only one instance needs to be in memory at each iteration.

- On the other hand, due to its stochastic (i.e., random) nature, this algorithm is much less regular than Batch Gradient Descent: instead of gently decreasing until it reaches the minimum, the cost function will bounce up and down, decreasing only on average.



B) Stochastic Gradient Descent

- When the cost function is very irregular Stochastic Gradient Descent has a better chance of finding the global minimum than Batch Gradient Descent does.

- Therefore randomness is good to escape from local optima, but bad because it means that the algorithm can never settle at the minimum.

- One solution to this dilemma is to gradually reduce the learning rate. The steps start out large (which helps make quick progress and escape local minima), then get smaller and smaller, allowing the algorithm to settle at the global minimum. This process is called simulated annealing.

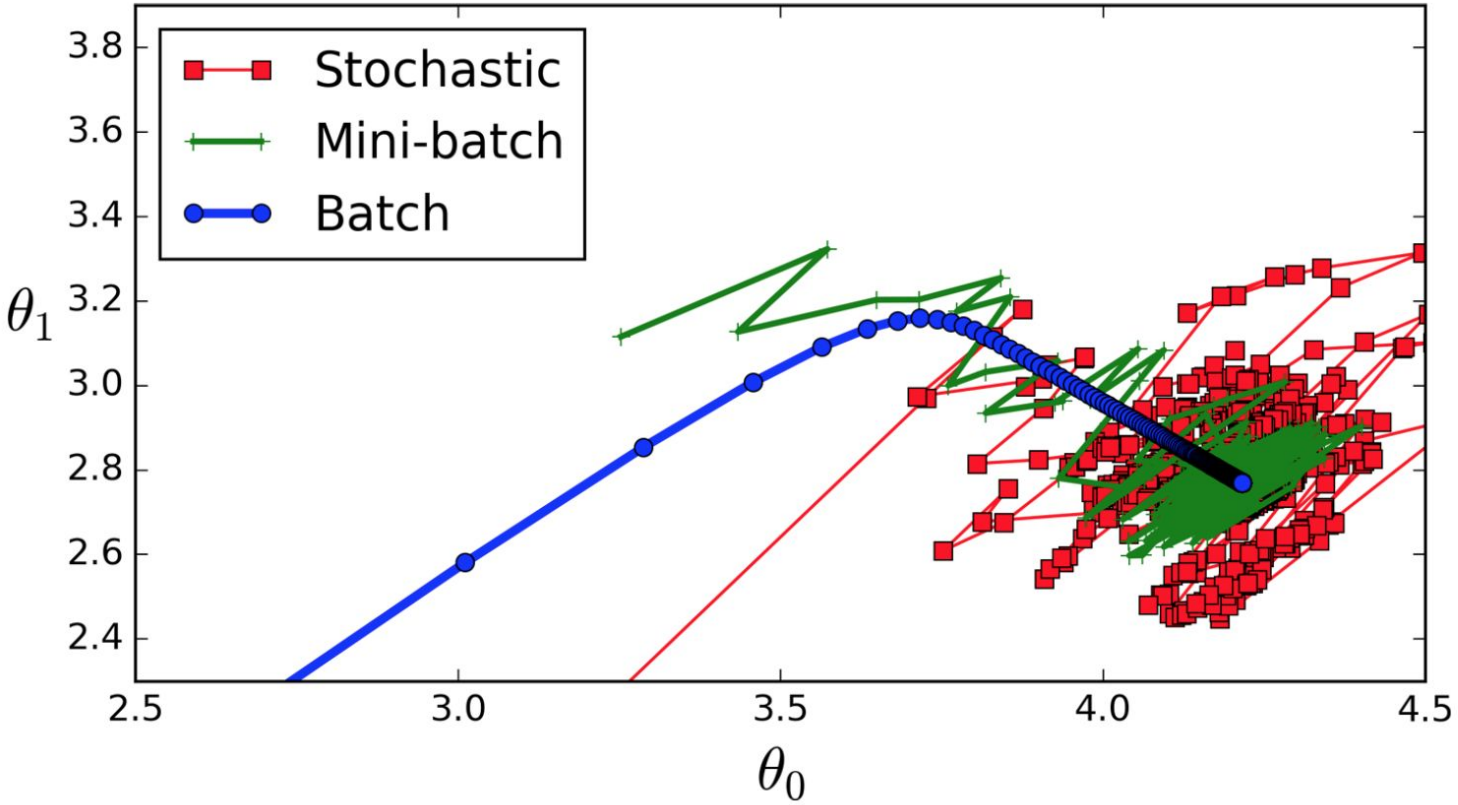
C) Mini-batch Gradient Descent

- Instead of computing the gradients based on the full training set (as in Batch GD) or based on just one instance (as in Stochastic GD), Mini-batch GD computes the gradients on small random sets of instances called mini- batches.

- The main advantage of Mini-batch GD over Stochastic GD is that you can get a performance boost from hardware optimization of matrix operations, especially when using GPUs.

- The algorithm's progress in parameter space is less erratic than with SGD, especially with fairly large mini-batches. As a result, Mini-batch GD will end up walking around a bit closer to the minimum than SGD. But, on the other hand, it may be harder for it to escape from local minima.

C) Mini-batch Gradient Descent

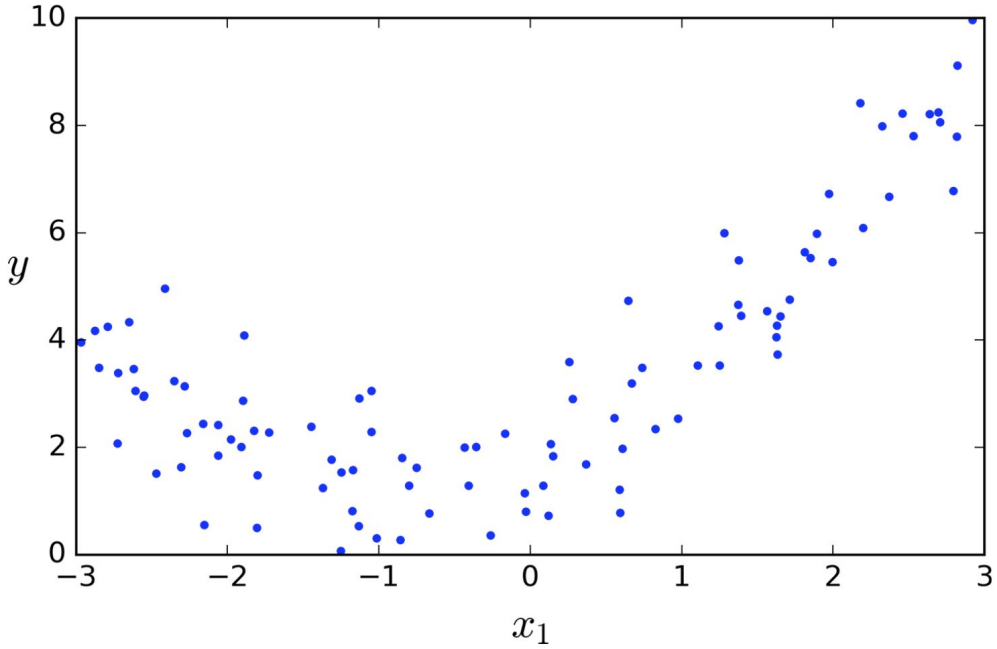


C) Mini-batch Gradient Descent

Algorithm	Large *m*	Out-of-core support	Large *n*	Hyperparams	Scaling required	Scikit-Learn
Normal Equation	Fast	No	Slow	0	No	`LinearRegression`
Batch GD	Slow	No	Fast	2	Yes	n/a
Stochastic GD	Fast	Yes	Fast	≥2	Yes	`SGDRegressor`
Mini-batch GD	Fast	Yes	Fast	≥2	Yes	n/a

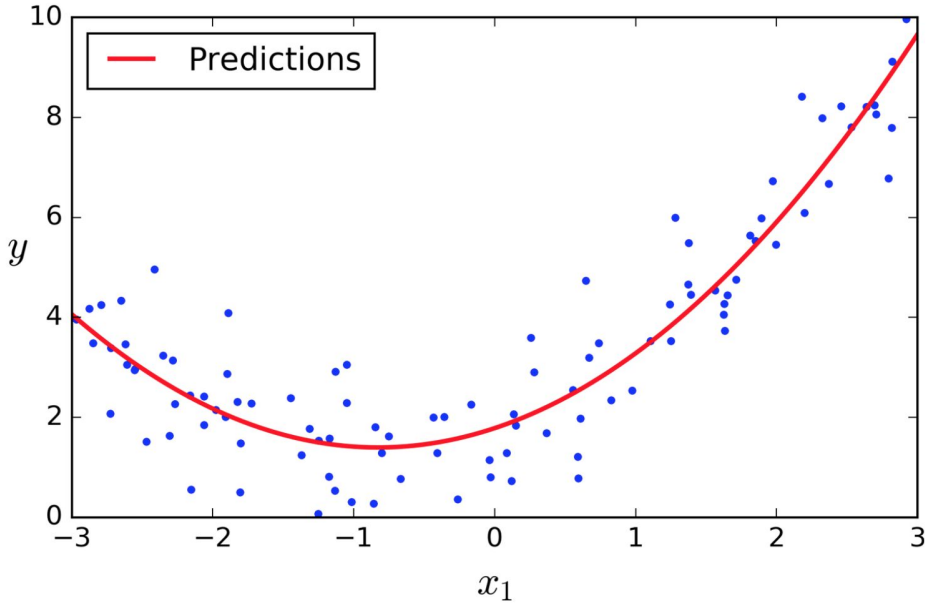
Polynomial Regression

What if your data is actually more complex than a simple straight line?



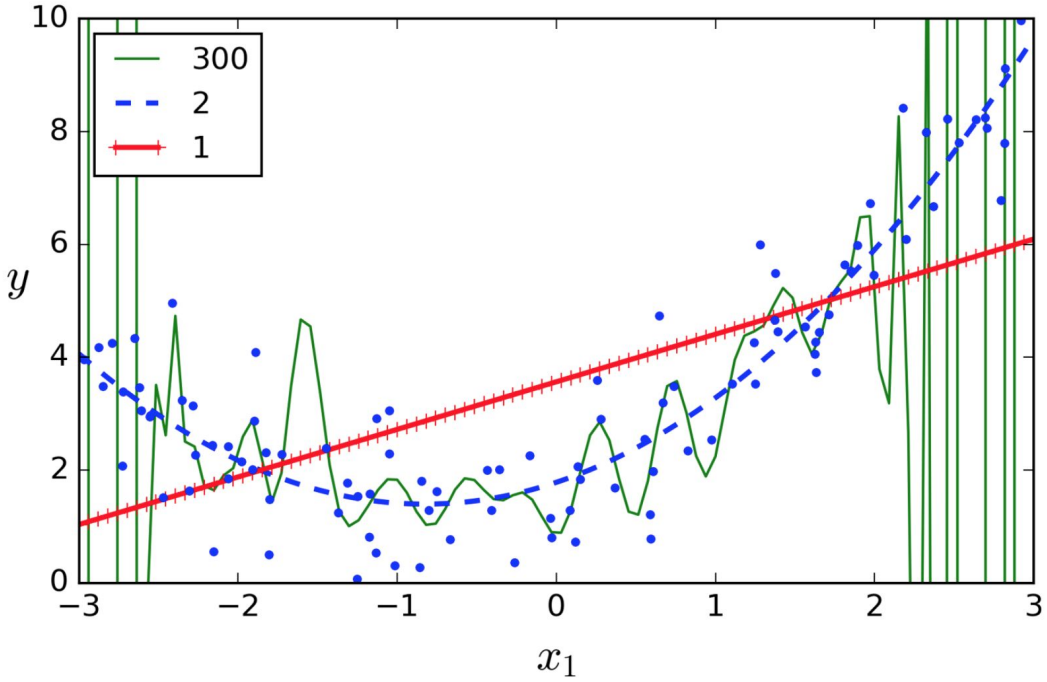
Polynomial Regression

We can actually use a linear model to fit nonlinear data. A simple way to do this is to add powers of each feature as new features, then train a linear model on this extended set of features. This technique is called Polynomial Regression.



Polynomial Regression

If you perform high-degree Polynomial Regression, you will likely fit the training data much better than with plain Linear Regression.



Polynomial Regression

How can we decide how complex your model should be? How can we tell that your model is overfitting or underfitting the data?

1. Cross-Validation
2. **Learning Curves**

Let’s see some examples