Linear regression

CSCI-P 556

ZORAN TIGANJ

Deeper dive into ML techniques

- So far we have treated Machine Learning models and their training algorithms mostly like black boxes.
- As of today, we will start taking a deeper dive into how popular machine learning techniques actually work.
- Having a good understanding of how things work can help you quickly home in on the appropriate model, the right training algorithm to use, and a good set of hyperparameters for your task.
- Understanding what's under the hood will also help you de bug issues and perform error analysis more efficiently.

Today: linear regression

- Based on chapter 4 of the *Hands On ML* textbook.
- We will discuss two ways to train linear regression models:
  - Using a "closed-form" equation that directly computes the model parameters that best fit the model to the training set
  - Using an iterative optimization approach called Gradient Descent (GD) that gradually tweaks the model parameters to minimize the cost function over the training set, eventually converging to the same set of parameters as the first method.
    - much faster because it has very little data to manipulate at every iteration.
    - the cost function will bounce up and down, decreasing only on average.

Linear regression

Linear regression model

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

- $\hat{y}$ 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 $\theta_1, \theta_2, \cdots, \theta_n$).

$$life\_satisfaction = \theta_0 + \theta_1 \times GDP\_per\_capita$$

Linear regression

Linear regression model:

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

More concise form of linear regression model:

$$\hat{y} = \boldsymbol{\theta}^\intercal \mathbf{x}$$

- $\boldsymbol{\theta}$ is the model's *parameter vector*, containing the bias term θ_0 and the feature weights θ_1 to θ_n.
- $\mathbf{x}$ is the instance's *feature vector*, containing x_0 to x_n, with x_0 always equal to 1.
- $\boldsymbol{\theta}^\intercal \mathbf{x}$ is the dot product of the vectors $\boldsymbol{\theta}$ and $\mathbf{x}$, which is of course equal to $\theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$.

Note: $\boldsymbol{\theta}$ and $\mathbf{x}$ are both column vectors (in ML vectors are commonly represented as column vectors, in other words vectors are represented as 2D arrays with a single column).

Training

- We first need a measure of how well (or poorly) the model fits the training data.
- For example, we can use RMSE.
- In practice, it is simpler to minimize the mean squared error (MSE) than the RMSE, and it leads to the same result (because the value that minimizes a function also minimizes its square root).
- The MSE of a Linear Regression hypothesis $h_{\boldsymbol{\theta}}$ on a training set **X** is

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

- $h_{\boldsymbol{\theta}}$ instead of just h to make it clear that the model is parametrized by the vector **θ**.

Training

- 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.

$$\widehat{\boldsymbol{\theta}} = (\mathbf{X}^\intercal \mathbf{X})^{-1}\, \mathbf{X}^\intercal\, \mathbf{y}$$

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

Deriving the normal equation

First, we use linear algebra to reorganize the loss function

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

$$X\theta - y = \begin{bmatrix} \theta_0 + \theta_1 x_1^{(1)} - y^{(1)} \\ \theta_0 + \theta_1 x_1^{(2)} - y^{(2)} \\ \vdots \\ \theta_0 + \theta_1 x_1^{(m)} - y^{(m)} \end{bmatrix}$$

$$MSE(\boldsymbol{\Theta}) = \frac{1}{m}\left(\mathbf{X}\Theta - \mathbf{y}\right)^T\left(\mathbf{X}\Theta - \mathbf{y}\right)$$

$$MSE(\boldsymbol{\Theta}) = \frac{1}{m}\left((\mathbf{X}\Theta)^T - \mathbf{y}^T\right)\left(\mathbf{X}\Theta - \mathbf{y}\right)$$

$$MSE(\boldsymbol{\Theta}) = \frac{1}{m}\left((\mathbf{X}\Theta)^T(\mathbf{X}\Theta) - (\mathbf{X}\Theta)^T\mathbf{y} - \mathbf{y}^T(\mathbf{X}\Theta) + \mathbf{y}^T\mathbf{y}\right)$$

$$MSE(\boldsymbol{\Theta}) = \frac{1}{m}\left(\Theta^T\mathbf{X}^T\mathbf{X}\Theta - 2(\mathbf{X}\Theta)^T\mathbf{y} + \mathbf{y}^T\mathbf{y}\right)$$

Deriving the normal equation

$$MSE(\Theta) = \frac{1}{m}\left(\Theta^T \mathbf{X}^T \mathbf{X}\Theta - 2(\mathbf{X}\Theta)^T \mathbf{y} + \mathbf{y}^T \mathbf{y}\right)$$

Note: we ignore 1/m

Compute gradient w.r.t. Θ, equate with 0 and solve for Θ.

$$\nabla_{\Theta} MSE(\Theta) = 0$$

$$\nabla_{\boldsymbol{\theta}} \operatorname{MSE}(\boldsymbol{\theta}) = \begin{pmatrix} \frac{\partial}{\partial \theta_0} \operatorname{MSE}(\boldsymbol{\theta}) \\ \frac{\partial}{\partial \theta_1} \operatorname{MSE}(\boldsymbol{\theta}) \\ \vdots \\ \frac{\partial}{\partial \theta_n} \operatorname{MSE}(\boldsymbol{\theta}) \end{pmatrix}$$

y	$\frac{\partial y}{\partial X}$
AX	A^T
X^TA	A
X^TX	2X
X^TAX	$AX + A^TX$

Deriving the normal equation

$$MSE(\Theta) = \frac{1}{m}\left(\Theta^T \mathbf{X}^T \mathbf{X}\Theta - 2(\mathbf{X}\Theta)^T \mathbf{y} + \mathbf{y}^T \mathbf{y}\right)$$

Note: we ignore 1/m

Compute gradient w.r.t. Θ, equate with 0 and solve for Θ.

$$\nabla_{\Theta} MSE(\Theta) = 0$$

$$2\mathbf{X}^T \mathbf{X}\Theta - 2\mathbf{X}^T \mathbf{y} = 0$$

$$\mathbf{X}^T \mathbf{X}\Theta = \mathbf{X}^T \mathbf{y}$$

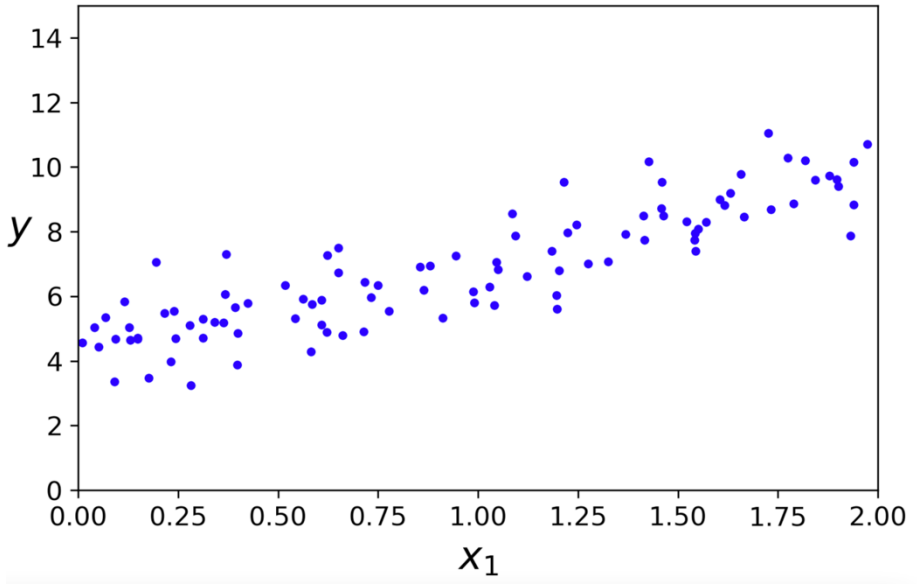
$$\Theta = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}$$

y	$\frac{\partial y}{\partial X}$
AX	A^T
X^TA	A
X^TX	2X
X^TAX	$AX + A^TX$

Linear regression

- Let's generate some linear-looking data to test this equation on

```
import numpy as np

X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)
```



The Normal Equation

- Now let's compute $\hat{\theta}$ using the Normal Equation.

```
X_b = np.c_[np.ones((100, 1)), X]  # add x0 = 1 to each instance
theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)
```

```
>>> theta_best
array([[4.21509616],
       [2.77011339]])
```

The function that we used to generate the data is y = 4 + 3x + Gaussian noise. Close enough, but the noise made it impossible to recover the exact parameters of the original function.

The Normal Equation

- Now we can make predictions using $\hat{\theta}$

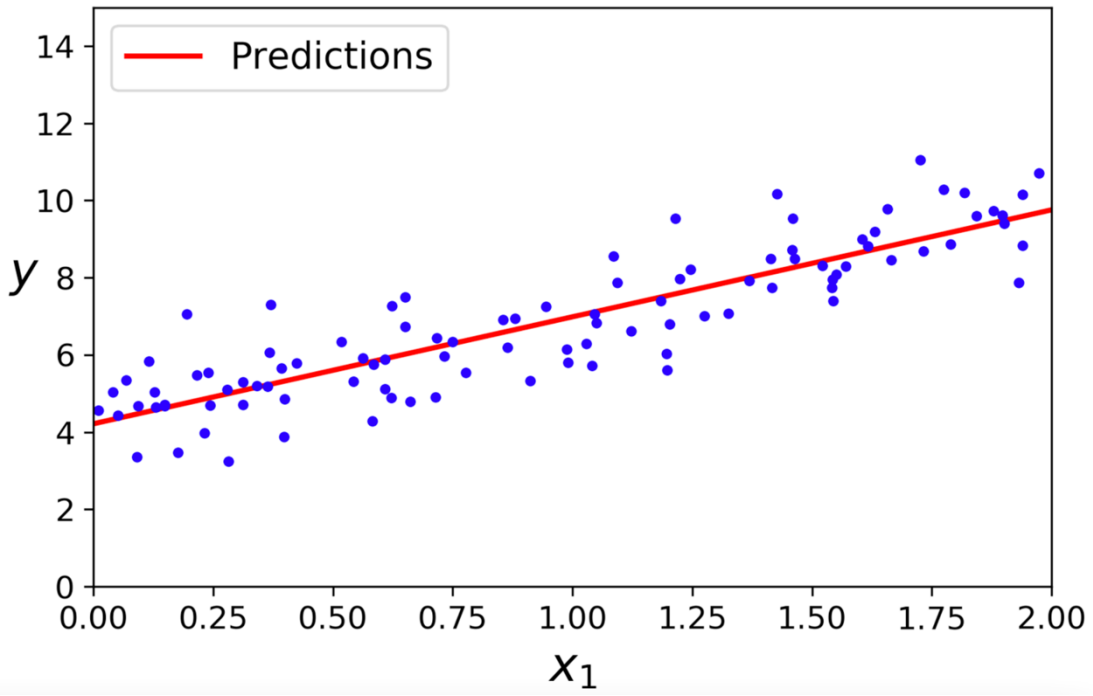
```
>>> X_new = np.array([[0], [2]])
>>> X_new_b = np.c_[np.ones((2, 1)), X_new] # add x0 = 1 to each instance
>>> y_predict = X_new_b.dot(theta_best)
>>> y_predict
array([[4.21509616],
       [9.75532293]])
```

y = 4 + 3x + Gaussian noise.

The Normal Equation

- Let's plot this model's predictions

```
plt.plot(X_new, y_predict, "r-")
plt.plot(X, y, "b.")
plt.axis([0, 2, 0, 15])
plt.show()
```



Linear regression using Scikit-Learn

```
>>> from sklearn.linear_model import LinearRegression
>>> lin_reg = LinearRegression()
>>> lin_reg.fit(X, y)
>>> lin_reg.intercept_, lin_reg.coef_
(array([4.21509616]), array([[2.77011339]]))
>>> lin_reg.predict(X_new)
array([[4.21509616],
       [9.75532293]])
```

This function computes $\widehat{\boldsymbol{\theta}} = \mathbf{X}^{+}\mathbf{y}$, where $\mathbf{X}^{+}$ is the *pseudoinverse* of $\mathbf{X}$

This approach is more efficient than computing the Normal Equation, plus it handles edge cases nicely: indeed, the Normal Equation may not work if the matrix $\mathbf{X}^T\mathbf{X}$ is not invertible (i.e., singular), such as:

- if m < n (where n is the number of features) or
- if some features are redundant, but the pseudoinverse is always defined.

Computational Complexity of The Normal Equation

- The Normal Equation computes the inverse of $\mathbf{X}^T\mathbf{X}$, which is an $(n + 1) \times (n + 1)$ 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 SVD approach used by Scikit-Learn's Linear Regression class is about $O(n^2)$. If you double the number of features, you multiply the computation time by roughly 4.

Computational Complexity of The Normal Equation

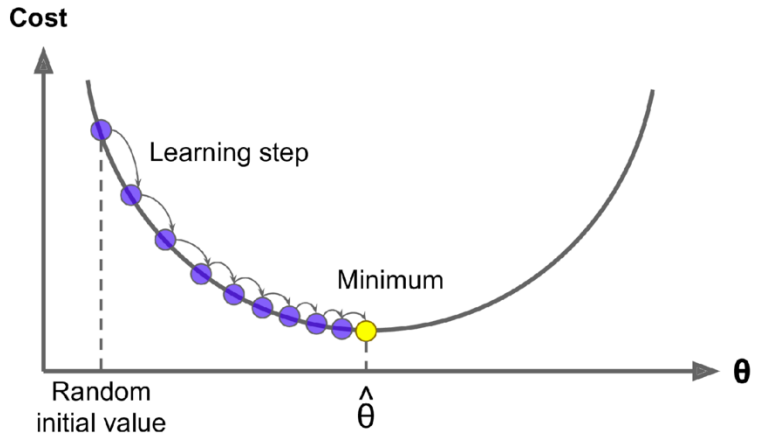
- Both the Normal Equation and the SVD approach get very slow when the number of features grows large (e.g., 100,000).
- On the positive side, both are linear with regard to the number of instances in the training set (they are O(m)), so they handle large training sets efficiently, provided they can fit in memory.

Gradient Descent

- Now we will look at a very different way to train a Linear Regression model, which is better suited for cases where there are a large number of features or too many training instances to fit in memory.

Gradient Descent

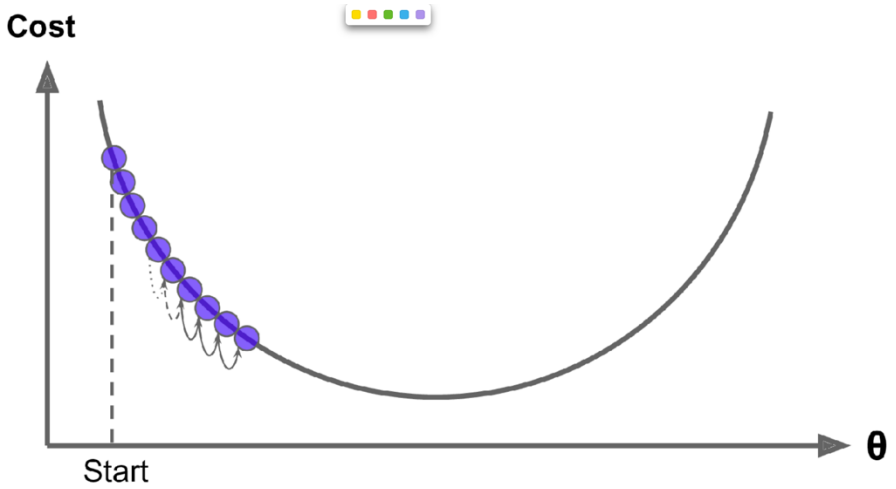
- Gradient Descent is a generic optimization algorithm capable of finding optimal solutions to a wide range of problems.
- The general idea of Gradient Descent is to tweak parameters iteratively in order to minimize a cost function.
- Suppose you are lost in the mountains in a dense fog, and you can only feel the slope of the ground be low your feet. A good strategy to get to the bottom of the valley quickly is to go downhill in the direction of the steepest slope.
- This is exactly what Gradient Descent does: it measures the local gradient of the error function with regard to the parameter vector θ, and it goes in the direction of descending gradient.



In this depiction of Gradient Descent, the model parameters are initialized randomly and get tweaked repeatedly to minimize the cost function; the learning step size is proportional to the slope of the cost function, so the steps gradually get smaller as the parameters approach the minimum

Learning rate in Gradient Descent

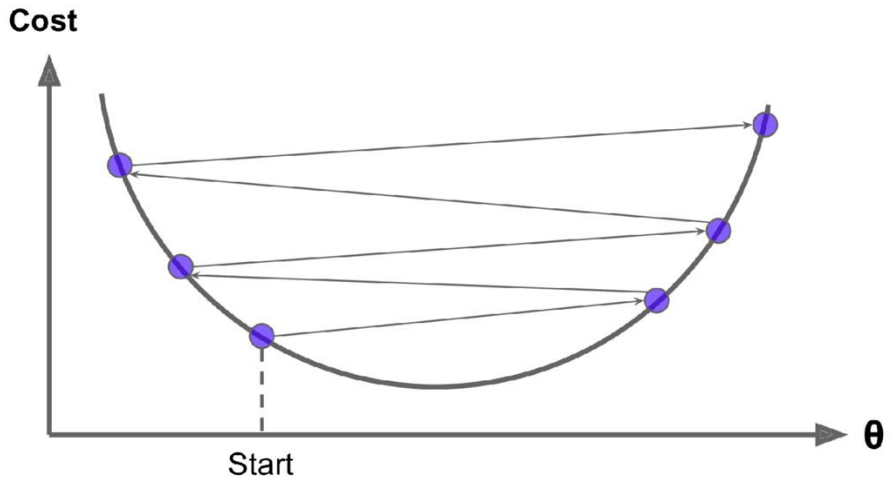
- An important parameter in Gradient Descent is the size of the steps, determined by t he learning rate hyperparameter. If the learning rate is too small, then the algorithm will have to go through many iterations to converge, which will take a long time.



The learning rate is too small

Learning rate in Gradient Descent

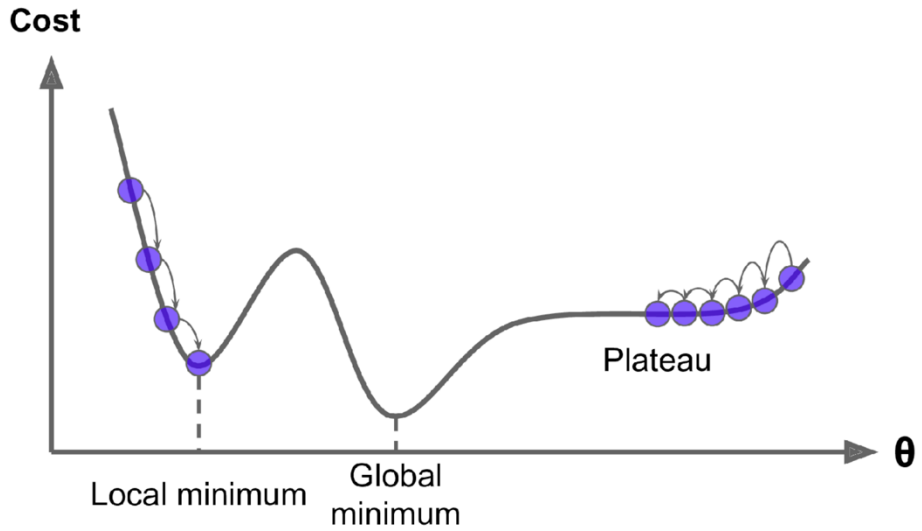
- On the other hand, if the learning rate is too high, you might jump across the valley and end up on the other side, possibly even higher up than you were before. This might make the algorithm diverge, with larger and larger values, failing to find a good solution.



The learning rate is too large

Challenges in Gradient Descent

- Finally, not all cost functions look like nice, regular bowls. There may be holes, ridges, plateaus, and all sorts of irregular terrains, making convergence to the minimum difficult.



- If the random initialization starts the algorithm on the left, then it will converge to a local minimum, which is not as good as the global minimum.
- If it starts on the right, then it will take a very long time to cross the plateau.

Gradient descent for MSE cost function and Linear Regression

- Fortunately, the MSE cost function for a Linear Regression model happens to be a convex function, which means that if you pick any two points on the curve, the line segment joining them never crosses the curve.
- This implies that there are no local minima, just one global minimum and Gradient Descent is guaranteed to approach arbitrarily close the global minimum (if you wait long enough and if the learning rate is not too high).

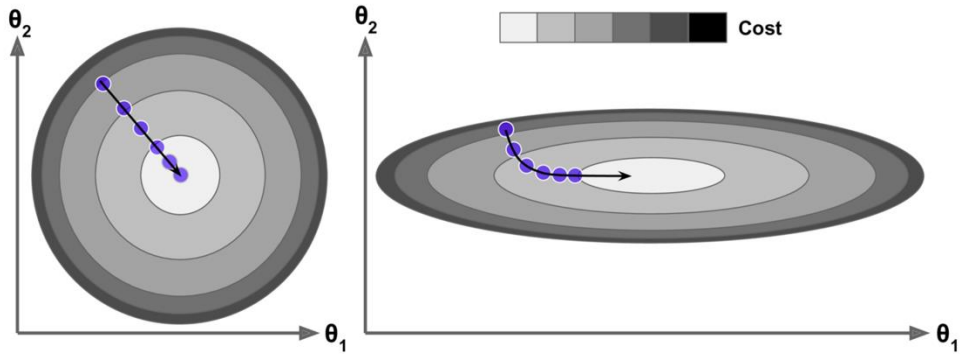


Figure 4-7. Gradient Descent with (left) and without (right) feature scaling

You should ensure that all features have a similar scale or else it will take much longer to converge.

Batch Gradient Descent

- To implement Gradient Descent, you need to compute the gradient of the cost function with regard to each model parameter θ_j.
- In other words, you need to calculate how much the cost function will change if you change θ_j just a little bit.

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

$$\frac{\partial}{\partial\theta_j}\mathrm{MSE}\left(\boldsymbol{\theta}\right) = \frac{2}{m}\sum_{i=1}^{m}\left(\boldsymbol{\theta}^{\mathsf{T}}\mathbf{x}^{(i)} - y^{(i)}\right)\, x_j^{(i)}$$

Batch Gradient Descent

- Instead of computing these partial derivatives individually, we can use gradient vector to compute them in one step:

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

- Notice that this formula involves calculations over the full training set X, at each Gradient Descent step! This is why the algorithm is called Batch Gradient Descent: it uses the whole batch of training data at every 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 or SVD decomposition.

Batch Gradient Descent

- Once you have the gradient vector, which points uphill, just go in the opposite direction to go downhill.

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

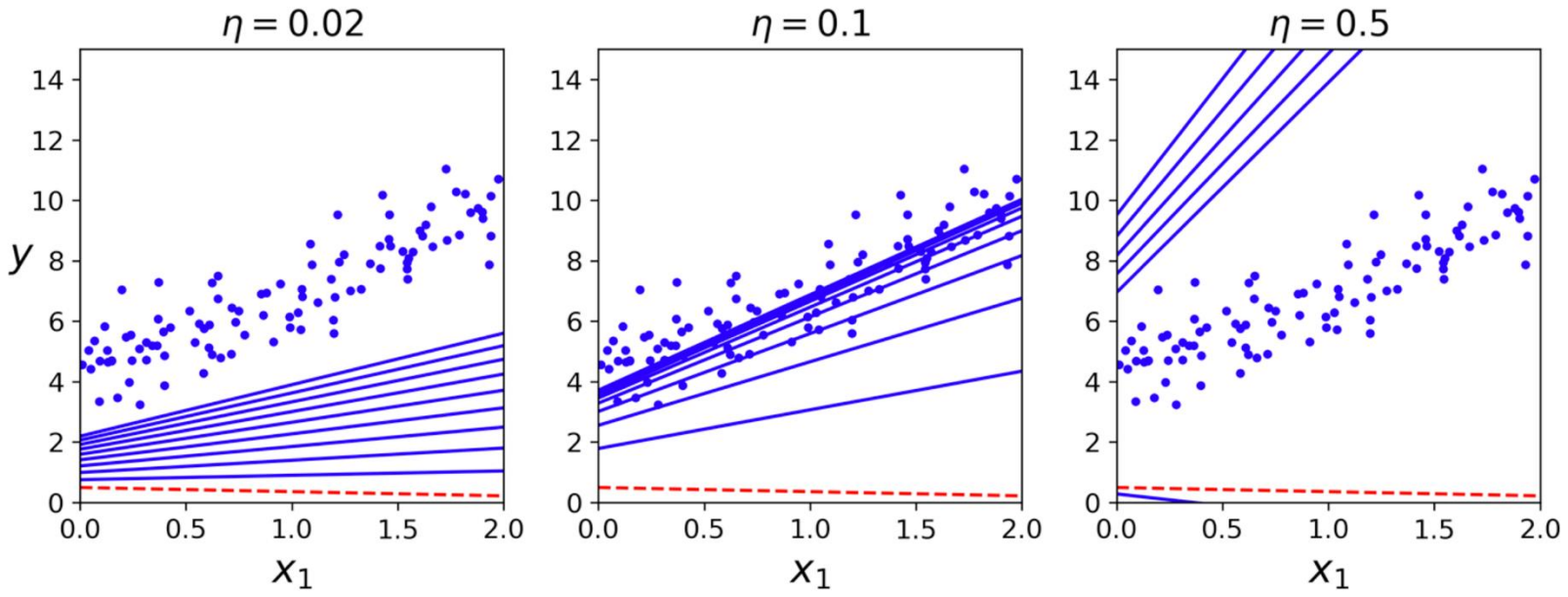
η is the learning rate

```
eta = 0.1  # learning rate
n_iterations = 1000
m = 100

theta = np.random.randn(2,1)  # random initialization

for iteration in range(n_iterations):
    gradients = 2/m * X_b.T.dot(X_b.dot(theta) - y)
    theta = theta - eta * gradients
```

```
>>> theta
array([[4.21509616],
       [2.77011339]])
```

Batch Gradient Descent

- First 10 steps of gradient descent using various learning rates



When to stop? When the gradient vector becomes tiny

Stochastic Gradient Descent

- The main problem with *Batch Gradient Descent* is the fact that it uses the whole training set to compute the gradients at every step, which makes it very slow when the training set is large.
- At the opposite extreme, *Stochastic Gradient Descent* picks a random instance in the training set at every step and computes the gradients based only on that single instance.

Stochastic Gradient Descent

- When the cost function is very irregular, this can actually help the algorithm jump out of local minima, so Stochastic Gradient Descent has a better chance of finding the global minimum than Batch Gradient Descent does.

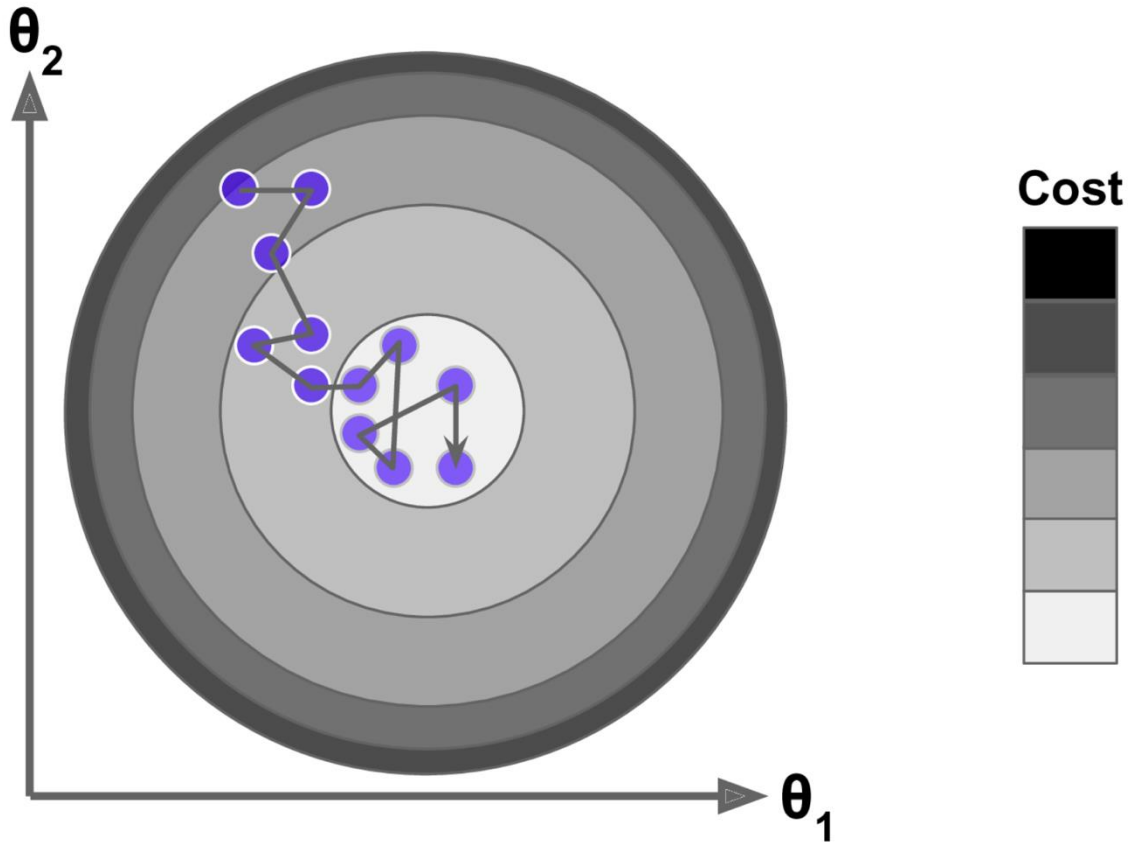


Figure 4-9. With Stochastic Gradient Descent, each training step is much faster but also much more stochastic than when using Batch Gradient Descent

Stochastic Gradient Descent

- Therefore, randomness is good to escape from local optima, but bad because it means that the algorithm can never settle at the minimum.
- The 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 (*simulated annealing*).
- The function that determines the learning rate at each iteration is called the *learning schedule*.

Stochastic Gradient Descent

```
n_epochs = 50
t0, t1 = 5, 50  # learning schedule hyperparameters

def learning_schedule(t):
    return t0 / (t + t1)

theta = np.random.randn(2,1)  # random initialization

for epoch in range(n_epochs):
    for i in range(m):
        random_index = np.random.randint(m)
        xi = X_b[random_index:random_index+1]
        yi = y[random_index:random_index+1]
        gradients = 2 * xi.T.dot(xi.dot(theta) - yi)
        eta = learning_schedule(epoch * m + i)
        theta = theta - eta * gradients
```

```
>>> theta
array([[4.21076011],
       [2.74856079]])
```

Stochastic Gradient Descent

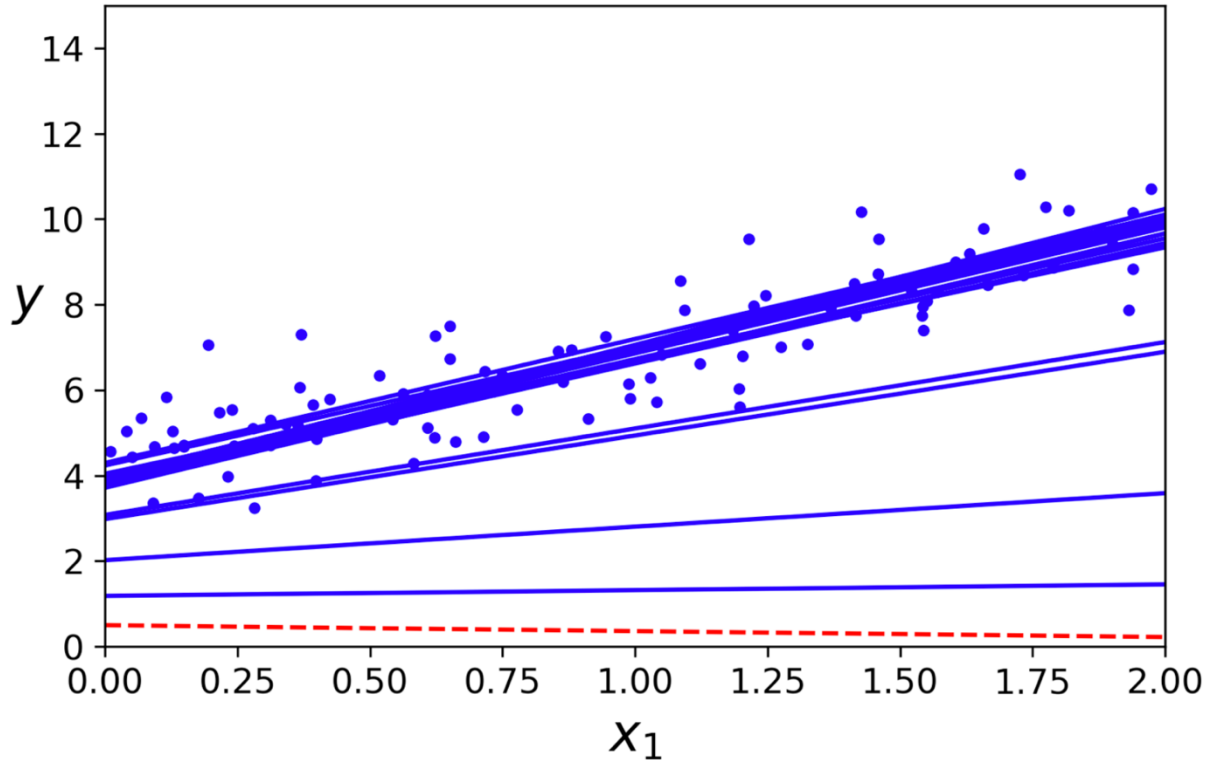


Figure 4-10. The first 20 steps of Stochastic Gradient Descent

Stochastic Gradient Descent

- When using Stochastic Gradient Descent, the training instances must be independent and identically distributed (IID) to ensure that the parameters get pulled toward the global optimum, on average.
- A simple way to ensure this is to shuffle the instances during training (e.g., pick each instance randomly, or shuffle the training set at the beginning of each epoch).
- If you do not shuffle the instances—for example, if the instances are sorted by label—then SGD will start by optimizing for one label, then the next, and so on, and it will not settle close to the global minimum.

Stochastic Gradient Descent

```
from sklearn.linear_model import SGDRegressor
sgd_reg = SGDRegressor(max_iter=1000, tol=1e-3, penalty=None, eta0=0.1)
sgd_reg.fit(X, y.ravel())
```

```
>>> sgd_reg.intercept_, sgd_reg.coef_
(array([4.24365286]), array([2.8250878]))
```

Mini-batch Gradient Descent

- At each step, 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 Minibatch GD over Stochastic GD is that you can get a performance boost from hardware optimization of matrix operations, especially when using GPUs.

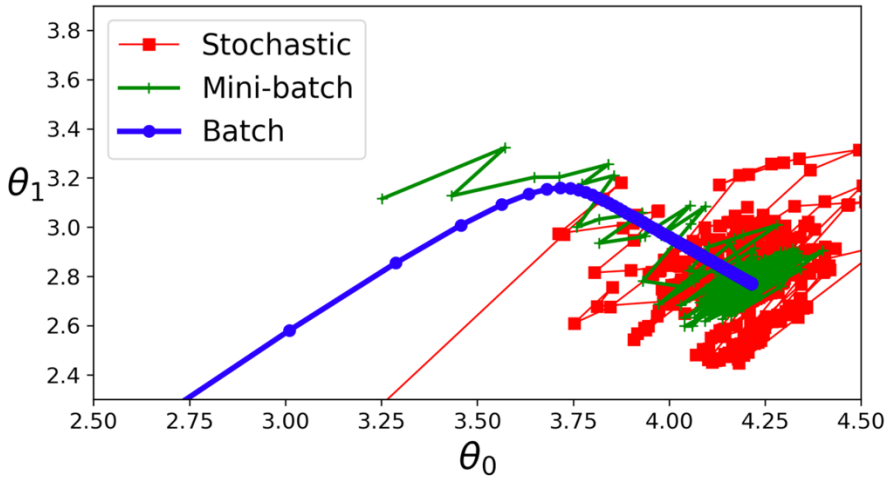


Figure 4-11. Gradient Descent paths in parameter space

Comparison

Table 4-1. Comparison of algorithms for Linear Regression

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

Next time

- Regularization and polynomial regression, from Chapter 4 from *Hands on machine learning textbook*