Gradient Descent

Linear regression using the Normal Equation

In [1]:
```
import numpy as np

# to make this notebook's output stable across runs
np.random.seed(42)

# Let’s generate some linear-looking data
X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1) # y= 4 + 3 X1 + Gaussian Noise
```

In [2]:
```
# To plot pretty figures
%matplotlib inline
import matplotlib.pyplot as plt

plt.plot(X, y, "b.")
plt.xlabel("$x_1$", fontsize=18)
plt.ylabel("$y$", rotation=90, fontsize=18)
plt.axis([0, 2, 0, 15])            # X-axis: 0:2, y-axis: 0:15
plt.show()
```

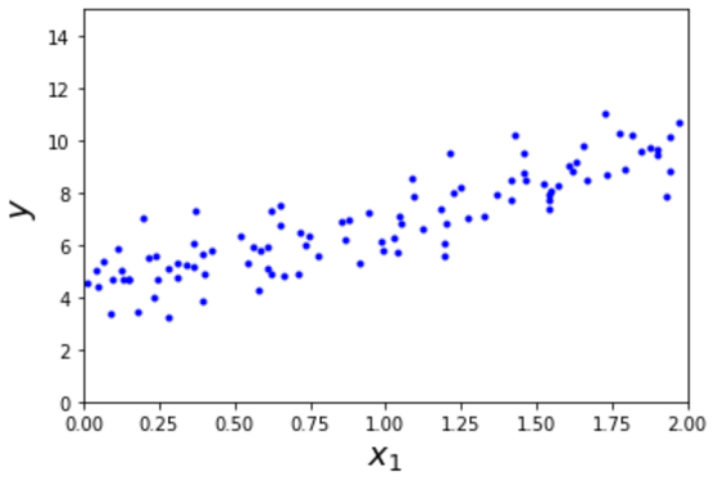


Figure 1: Randomly generated linear dataset.

Now let’s compute `Beta` using the Normal Equation. We will use the `inv()` function from NumPy’s linear algebra module (`np.linalg`) to compute the inverse of a matrix, and the `dot()` method for matrix multiplication:

In [3]:
```
X_b = np.c_[np.ones((100, 1)), X]  # add x0 = 1 to each instance
Beta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)
```

In [4]:
```
Beta_best
```

Out[4]:
```
array([[4.21509616],
       [2.77011339]])
```

We would have hoped for $Beta_0$ = 4 and $Beta_1$ = 3 instead of $Beta_0$ = 4.215 and $Beta_1$ = 2.770. Close enough, but the noise made it impossible to recover the exact parameters of the original function.

Now we can make predictions using `Beta_best` :

```
In [5]: X_new = np.array([[0], [2]])
        X_new
```

```
Out[5]: array([[0],
               [2]])
```

```
In [6]: X_new_b = np.c_[np.ones((2, 1)), X_new]  # add x0 = 1 to each instance
        X_new_b
```

```
Out[6]: array([[1., 0.],
               [1., 2.]])
```

```
In [7]: y_predict = X_new_b.dot(Beta_best)
        y_predict
```

```
Out[7]: array([[4.21509616],
               [9.75532293]])
```

Let's plot this model's predictions (Figure 2):

```
In [8]: plt.plot(X_new, y_predict, "r-", linewidth=2, label="Predictions")
        plt.plot(X, y, "b.")
        plt.xlabel("$x_1$", fontsize=18)
        plt.ylabel("$y$", rotation=90, fontsize=18)
        plt.legend(loc="upper right", fontsize=14)
        plt.axis([0, 2, 0, 15])
        plt.show()
```

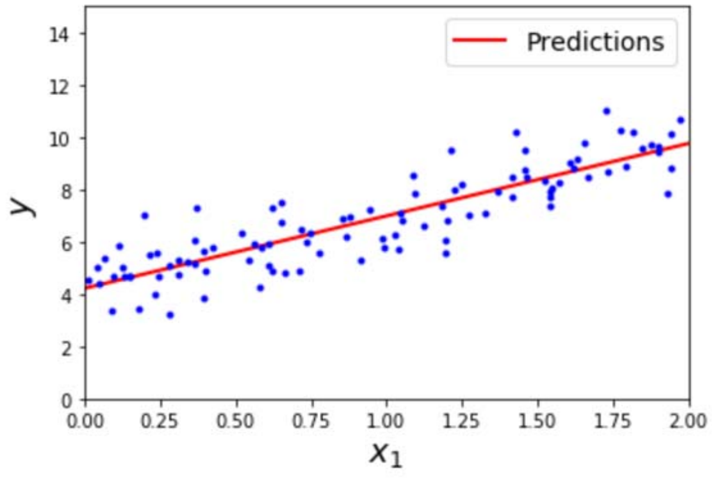


Figure 2: Linear Regression model predictions.

Performing Linear Regression using Scikit-Learn is shown below:

```
In [9]: from sklearn.linear_model import LinearRegression

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

```
Out[9]: (array([4.21509616]), array([[2.77011339]]))
```

```
In [10]: lin_reg.predict(X_new)
```

```
Out[10]: array([[4.21509616],
                [9.75532293]])
```

Linear regression using batch gradient descent

Let's look at an implementation of the Gradient Descent algorithm:

In [11]:
```
alpha = 0.1  # learning rate
n_iterations = 1000
N = 100    # Sample size

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

Out[11]:
```
array([[0.01300189],
       [1.45353408]])
```

In [12]:
```
for iteration in range(n_iterations):
    gradients = 2/N * X_b.T.dot(X_b.dot(Beta) - y)
    Beta = Beta - alpha * gradients
    print(Beta)
```

```
[[1.10103284]
 [2.5689581 ]]
[[1.76167724]
 [3.20430202]]
[[2.17070217]
 [3.55850018]]
[[2.43130725]
 [3.74830359]]
[[2.60409456]
 [3.84222284]]
[[2.72466079]
 [3.88045243]]
[[2.81392385]
 [3.88656613]]
[[2.88418448]
 [3.87438216]]
[[2.94268445]
 [3.85199292]]
[[2.99369522]
 [3.82413043]]
[[3.039744  ]
 [3.79355481]]
[[3.08233345]
 [3.76186927]]
[[3.12236419]
 [3.72999852]]
[[3.16038278]
 [3.69846951]]
[[3.19672739]
 [3.66757628]]
[[3.23161323]
 [3.63747665]]
[[3.26518282]
 [3.608249  ]]
[[3.29753539]
 [3.57992555]]
[[3.32874431]
 [3.5525119 ]]
[[3.3588672 ]
 [3.52599845]]
[[3.38795195]
 [3.50036716]]
[[3.41604028]
 [3.47559544]]
[[3.44316983]
 [3.4516584 ]]
[[3.46937536]
 [3.42853024]]
[[3.49468955]
 [3.406185  ]]
[[3.51914342]
 [3.38459697]]
[[3.54276663]
 [3.36374097]]
[[3.56558763]
 [3.34359244]]
[[3.58763381]
 [3.32412755]]
[[3.60893156]
 [3.3053232 ]]
[[3.62950634]
 [3.28715702]]
[[3.64938272]
 [3.2696074 ]]
[[3.66858442]
 [3.25265343]]
[[3.68713436]
 [3.23627489]]
[[3.70505465]
 [3.22045227]]
[[3.72236668]
 [3.20516671]]
[[3.7390911 ]
```

```
In [13]: Beta
```

```
Out[13]: array([[4.21509616],
                [2.77011339]])
```

```
In [14]: X_new_b.dot(Beta)
```

```
Out[14]: array([[4.21509616],
                [9.75532293]])
```

Note: The results of the Gradient Descent approach are precisely what the Normal Equation found. However, ***this was possible because the problem with a one-dimensional linear equation is very simple to solve, and we were even lucky with the learning rate***.

Let us try with a different learning rate alpha? Figure 3 shows the first 10 steps of Gradient Descent using three different learning rates (the dashed line represents the starting point).

```
In [15]: Beta_path_bgd = []

         def plot_gradient_descent(Beta, alpha, Beta_path=None):
             N = len(X_b)
             plt.plot(X, y, "b.")
             n_iterations = 1000
             for iteration in range(n_iterations):
                 if iteration < 10:
                     y_predict = X_new_b.dot(Beta)
                     style = "b-" if iteration > 0 else "r--"
                     plt.plot(X_new, y_predict, style)
                 gradients = 2/N * X_b.T.dot(X_b.dot(Beta) - y)
                 Beta = Beta - alpha * gradients
                 if Beta_path is not None:
                     Beta_path.append(Beta)
             plt.xlabel("$x_1$", fontsize=18)
             plt.axis([0, 2, 0, 15])
             plt.title(r"$\alpha = {}$".format(alpha), fontsize=16)
```

```
In [16]: np.random.seed(42)
         Beta = np.random.randn(2,1)  # random initialization

         plt.figure(figsize=(10,4))
         plt.subplot(131); plot_gradient_descent(Beta, alpha=0.02)
         plt.ylabel("$y$", rotation=90, fontsize=18)
         plt.subplot(132); plot_gradient_descent(Beta, alpha=0.1, Beta_path=Beta_path_bgd)
         plt.subplot(133); plot_gradient_descent(Beta, alpha=0.5)

         plt.show()
```

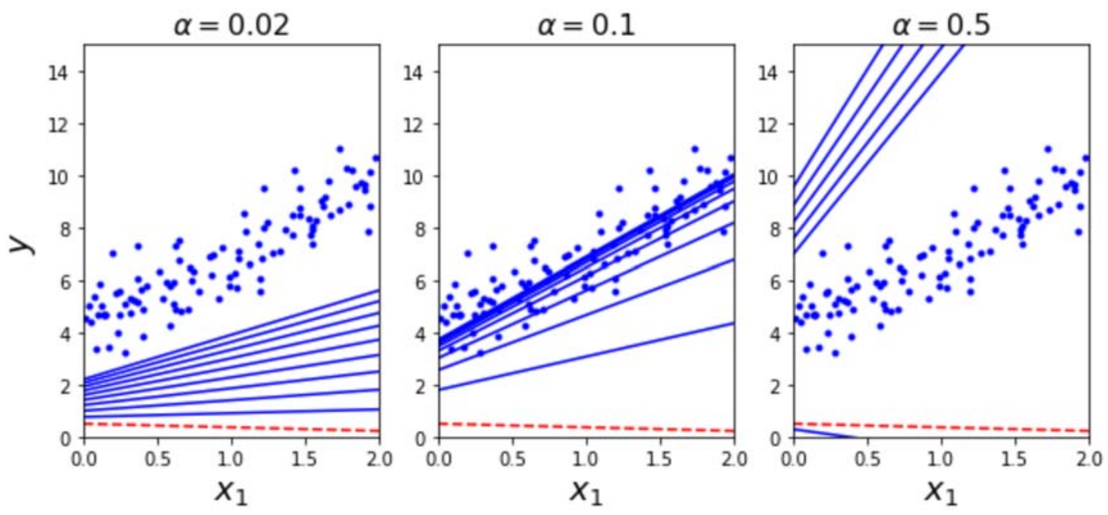


Figure 3: Gradient Descent with various learning rates.

From **Figure 3**:

- On the left, the learning rate is too low: the algorithm will eventually reach the solution, but it will take a long time.
- In the middle, the learning rate looks pretty good: in just a few iterations, it has already converged to the solution.
- On the right, the learning rate is too high: **the algorithm diverges, jumping all over the place and actually getting further and further away from the solution at every step**.

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.

How to set the number of iterations?:

The number of iteration can be set by trial and error:

- If it is too low, you will still be far away from the optimal solution when the algorithm stops.
- but if it is too high, you will waste time while the model parameters do not change anymore.

A simple solution is to set a very large number of iterations but to interrupt the algorithm when the gradient vector becomes tiny—that is, when its norm becomes smaller than a tiny number ϵ (called the tolerance)—because this happens when Gradient Descent has (almost) reached the minimum.

Stochastic Gradient Descent

In [17]:
```
Beta_path_sgd = []

N = len(X_b)
np.random.seed(42)
```

The code below implements Stochastic Gradient Descent (SGD) using a simple learning schedule:

In [18]:
```
n_epochs = 50
t0, t1 = 5, 50  # learning schedule hyperparameters

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

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

Out[18]:
```
array([[ 0.49671415],
       [-0.1382643 ]])
```

In [19]:
```python
for epoch in range(n_epochs):
    for i in range(N):
        if epoch == 0 and i < 20:
            y_predict = X_new_b.dot(Beta)
            style = "b-" if i > 0 else "r--"
            plt.plot(X_new, y_predict, style)
        random_index = np.random.randint(N)
        xi = X_b[random_index:random_index+1]
        yi = y[random_index:random_index+1]
        gradients = 2 * xi.T.dot(xi.dot(Beta) - yi)
        alpha = learning_schedule(epoch * N + i)
        # print(alpha)
        Beta = Beta - alpha * gradients
        Beta_path_sgd.append(Beta)

plt.plot(X, y, "b.")
plt.xlabel("$x_1$", fontsize=18)
plt.ylabel("$y$", rotation=90, fontsize=18)
plt.axis([0, 2, 0, 15])
plt.show()
```

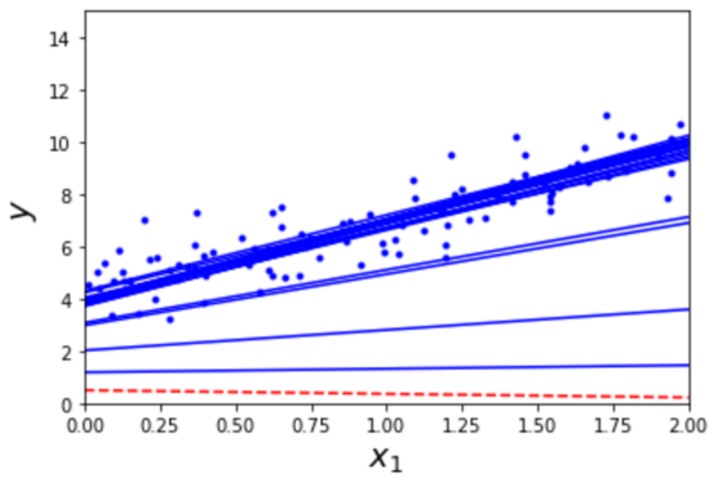


Figure 4: shows the first 20 steps of training (notice how irregular the steps are).

- By convention we iterate by rounds of N iterations; each round is called an **epoch**.
- While the Batch Gradient Descent code iterated 1,000 times through the whole training set (i.e., N samples per epoch), Stochastic Gradient Descent (SGD) code goes through the training set only 50 times (i.e., 50 epochs) and reaches a pretty good solution.

In [20]:
```python
Beta
```

Out[20]:
```
array([[4.21076011],
       [2.74856079]])
```

To perform Linear Regression using Stochastic GD with Scikit-Learn, you can use the `SGDRegressor` class, which defaults to optimizing the squared error cost function. The following code runs for maximum 1,000 epochs or until the loss drops by less than 0.001 during one epoch (`max_iter=1000` , `tol=1e-3`). It starts with a learning rate of 0.1 (`eta0=0.1`), using the default learning schedule (different from the preceding one). Lastly, it does not use any regularization (`penalty=None` ; about which we will study in the next chapter):

In [21]:
```python
from sklearn.linear_model import SGDRegressor

sgd_reg = SGDRegressor(max_iter=1000, tol=1e-3, penalty=None, eta0=0.1, random_state=42)
sgd_reg.fit(X, y.ravel())
```

Out[21]:
```
SGDRegressor(alpha=0.0001, average=False, early_stopping=False, epsilon=0.1,
             eta0=0.1, fit_intercept=True, l1_ratio=0.15,
             learning_rate='invscaling', loss='squared_loss', max_iter=1000,
             n_iter_no_change=5, penalty=None, power_t=0.25, random_state=42,
             shuffle=True, tol=0.001, validation_fraction=0.1, verbose=0,
             warm_start=False)
```

In [22]:
```
sgd_reg.intercept_, sgd_reg.coef_
```
Out[22]:
```
(array([4.24365286]), array([2.8250878]))
```

Again, the solution is quite close to the one returned by the Normal Equation.

Mini-batch gradient descent

- For Mini-batch GD, 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 Mini-batch GD over Stochastic GD is that you can get a performance boost from hardware optimization of matrix operations, especially when using GPUs.
- Thus, Mini-batch GD will end up walking around a bit closer to the minimum than Stochastic GD—but it may be harder for it to escape from local minima.

In [23]:
```
Beta_path_mgd = []

n_iterations = 50
minibatch_size = 20

np.random.seed(42)
Beta = np.random.randn(2,1)  # random initialization

t0, t1 = 200, 1000
def learning_schedule(t):
    return t0 / (t + t1)

t = 0
for epoch in range(n_iterations):
    shuffled_indices = np.random.permutation(N)
    X_b_shuffled = X_b[shuffled_indices]
    y_shuffled = y[shuffled_indices]
    for i in range(0, N, minibatch_size):
        t += 1
        xi = X_b_shuffled[i:i+minibatch_size]
        yi = y_shuffled[i:i+minibatch_size]
        gradients = 2/minibatch_size * xi.T.dot(xi.dot(Beta) - yi)
        alpha = learning_schedule(t)
        Beta = Beta - alpha * gradients
        Beta_path_mgd.append(Beta)
```

In [24]:
```
Beta
```
Out[24]:
```
array([[4.25214635],
       [2.7896408 ]])
```

Let us draw the paths taken by the three Gradient Descent algorithms in parameter space during training:

In [25]:
```
Beta_path_bgd = np.array(Beta_path_bgd)
Beta_path_sgd = np.array(Beta_path_sgd)
Beta_path_mgd = np.array(Beta_path_mgd)
```

In [26]:
```python
plt.figure(figsize=(7,4))
plt.plot(Beta_path_sgd[:, 0], Beta_path_sgd[:, 1], "r-s", linewidth=1, label="Stochastic")
plt.plot(Beta_path_mgd[:, 0], Beta_path_mgd[:, 1], "g-+", linewidth=2, label="Mini-batch")
plt.plot(Beta_path_bgd[:, 0], Beta_path_bgd[:, 1], "b-o", linewidth=3, label="Batch")
plt.legend(loc="upper left", fontsize=16)
plt.xlabel(r"$\beta_0$", fontsize=20)
plt.ylabel(r"$\beta_1$   ", fontsize=20, rotation=0)
plt.axis([2.5, 4.5, 2.3, 3.9])
plt.show()
```

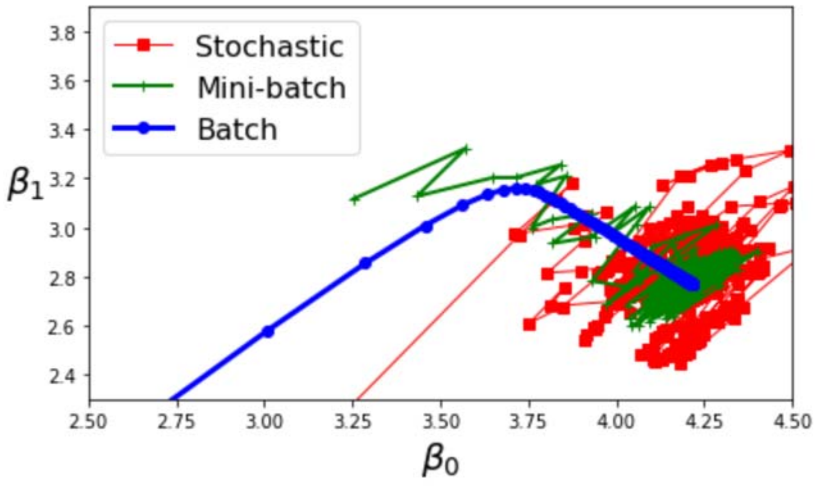
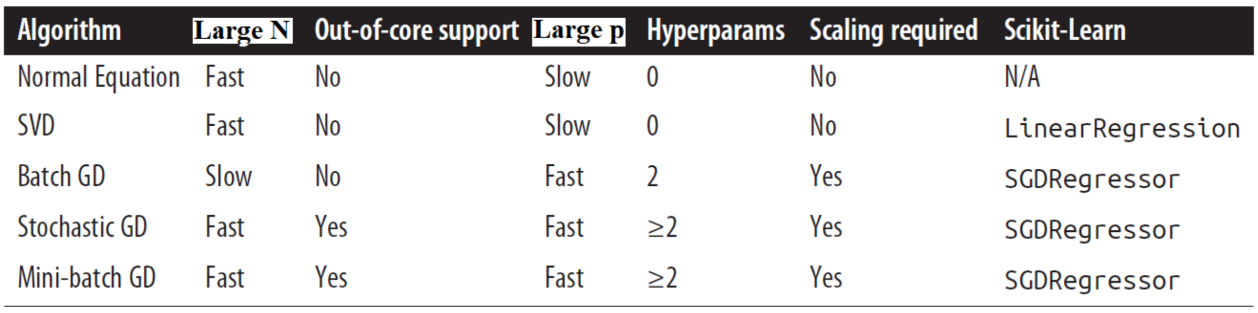


Figure 5: Gradient Descent paths in parameter space.

Figure 5 shows the paths taken by the three Gradient Descent algorithms in parameter space during training. They all end up near the minimum, but Batch GD's path actually stops at the minimum, while both Stochastic GD and Mini-batch GD continue to walk around. However, don't forget that Batch GD takes a lot of time to take each step, and Stochastic GD and Mini-batch GD would also reach the minimum if you used a good learning schedule.

We compare the algorithms we've discussed so far for Linear Regression (recall that N is the number of training instances and p is the number of features) in Table 2.1 below:

Table 2.1: Comparison of algorithms for Linear Regression.



Algorithm	Large N	Out-of-core support	Large p	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`