Lecture Two: Supervised Learning - Regression

COMP3032 Machine Learning

Linear regression

- Model based learning
- Linear equations and cost functions
- Use scikit-learn to build a regression model
- Linear regression in general form

Optimisation

- Mean Square Error (MSE)
- An example
- Gradient descent
- Batch gradient descent
- Stochastic Gradient Descent
- Mini-batch Gradient Descent

Model based learning: An example

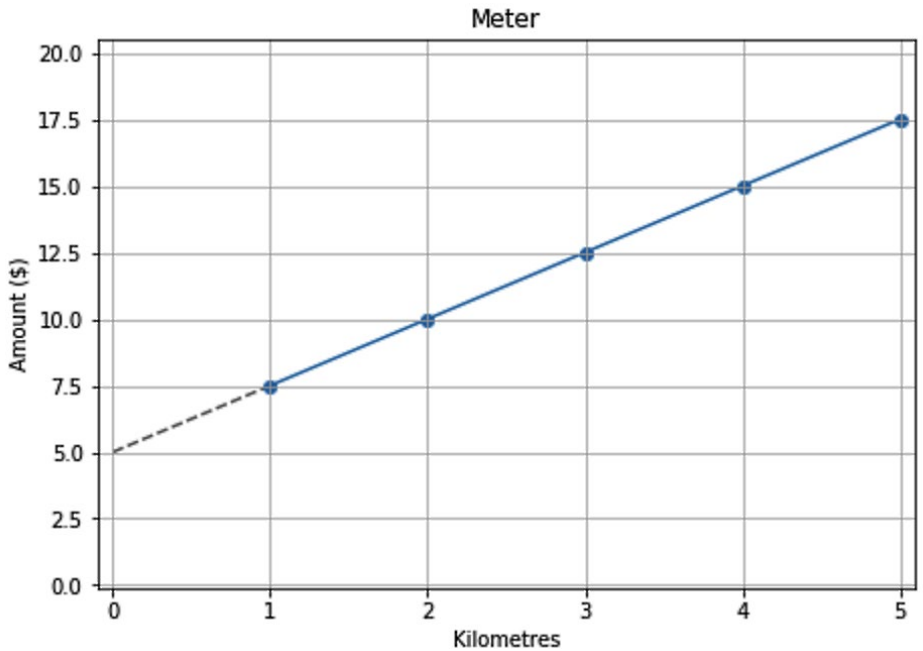
Build a model from the training examples and then use that model to make predictions.

- Five passengers have taken a taxi trip
- The record shows the distance each taxi covered in kilometers and the fair displayed on its meter at the end of each trip

Kilometres	Meter
1	7.5
2	10.0
3	12.5
4	15.0
5	17.5

Taxi trip example

- Taxi meters usually start with a certain amount, then add a fixed charge for each kilometer traveled
- Can model the meter using the following equation
- $Meter = \theta_0 + \theta_1 Distance$
- $\theta_0 = 5, \theta_1 = 2.5$

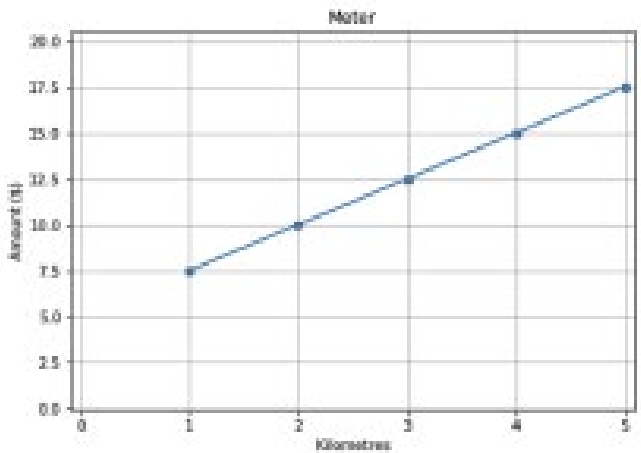


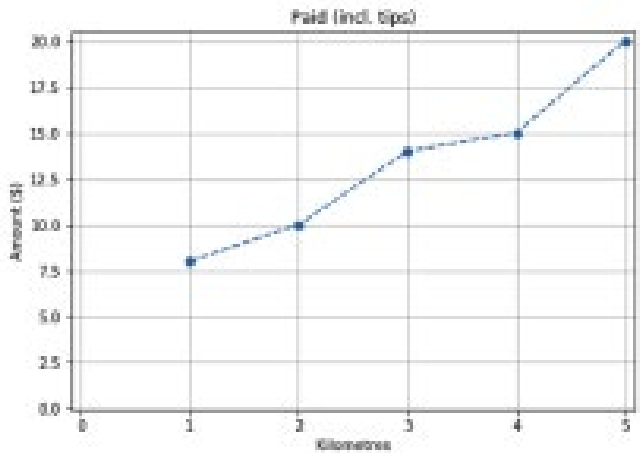
Taxi trip example-continued

- The real payment usually includes a tip as shown in the table
- The relationship between the distance traveled and the amount paid is no longer linear
- There is still a line that can somewhat approximate this relationship
- We will use a linear regression algorithm to find this approximation

Taxi trip example-continued

Kilometres	Meter	Paid (incl. tips)
1	7.5	8
2	10.0	10
3	12.5	14
4	15.0	15
5	17.5	20





Linear equations

- $Amount\_paid = \theta_0 + \theta_1 Distance$
- Two parameters: θ_0 and θ_1
- By tweaking these two parameters, you can represent any linear function

Performance measure

- Need to define the two parameters: θ_0 and θ_1
- Which values will make your model perform best?
- Need to define a performance measure
- Utility function (or fitness function): measures how good your model is
- Cost function: measures how bad it is

Cost functions

Linear Regression problems:

- People typically use a cost function
- Measures the distance between the linear model's predictions and the given answers
- The objective is to minimize this distance

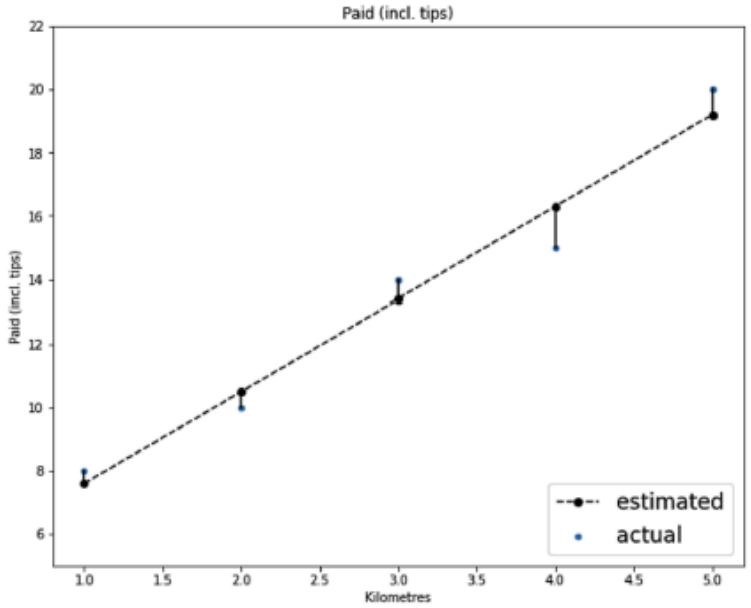
Linear regression algorithm:

- Feed it your training examples
- Finds the parameters that make the linear model fit best to your data
- For our example, the optimal parameter values are: $\theta_0 = 4.7$ and $\theta_1 = 2.9$

The linear model that fits the training data best

The linear regression algorithm tries to find a line

- where the Mean of the Squared Errors (MSE)between the estimated points on the line and the actual points is minimal
- In the graph, the objective is to find a dotted line that makes the average squared lengths of the vertical lines minimal



Estimating the amount paid

Use scikit-learn to build a regression model to estimate the amount paid to the taxi driver:

```
from sklearn.linear_model import LinearRegression

# Initialize and train the model
reg = LinearRegression()
reg.fit(df_taxi[['Kilometres']],
                  df_taxi['Paid (incl. tips)'])

# Make predictions
df_taxi['Paid (Predicted)'] =
                  reg.predict(df_taxi[['Kilometres']])
```

Linear equation

- Once a linear model is trained, you can get its intercept and coefficients
- Using the *intercept_* and *coef_* parameters
- Use the following code fragment to create the linear equations of the estimated line

```
print(
    'Amount Paid = {:.1f} + {:.1f} * Distance'.format(
        reg.intercept_, reg.coef_[0]
    )
)
```

The output:

```
Amount Paid=4.7+2.9 * Distance
```

Use the model to make predictions

- How much a person is expected to pay to a taxi driver?
- Find the distance travelled, say 6 kilometers
- Apply the model to make a prediction
- $Amount\_paid$
  $= \theta_0 + \theta_1 Distance$
  $= 4.7 + 2.9 * 6$
  $= 22.1$

In summary

- Understand the problem
- Collect and study the data
- Select a model
- Select a performance measure, say a cost function
- Trained the model on the training data
  - The learning algorithm searches for the model parameter values that minimize a cost function
- Applying the model to make predictions on new cases

Linear regression in general form

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).

- $\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + ... + \theta_n x_n$
- $\hat{y}$ is the predicted value
- n is the number of features
- x_i is the *ith* feature value
- θ_j is the *jth* model parameter
- θ_0: term
- θ_1, ... θ_n: feature weight

The vector form

- $\hat{y} = h_\theta(\mathbf{x}) = \theta \cdot \mathbf{x}$
- θ is the model's parameter vector
- $\mathbf{x}$ is the instance's feature vector, $x_0 = 1$
- $\theta \cdot \mathbf{x}$ is the dot product of the vectors θ and $\mathbf{x}$
- h_θ is the hypothesis function, using the model parameters θ

Mean Square Error (MSE)

- Training a model:the process of finding its parameters so that the model best fits the training set
- Popular Performance measure: Mean Square Error (MSE)
- $MSE(X, h_\theta) = \frac{1}{m} \sum_{i=1}^{m} (\theta^T x^{(i)} - y^{(i)})^2$

The Normal Equation

To find the value of θ that minimizes the cost function $MSE(\theta)$, there is a mathematical equation that gives the result directly. It is known as the normal equation:

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

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

An example

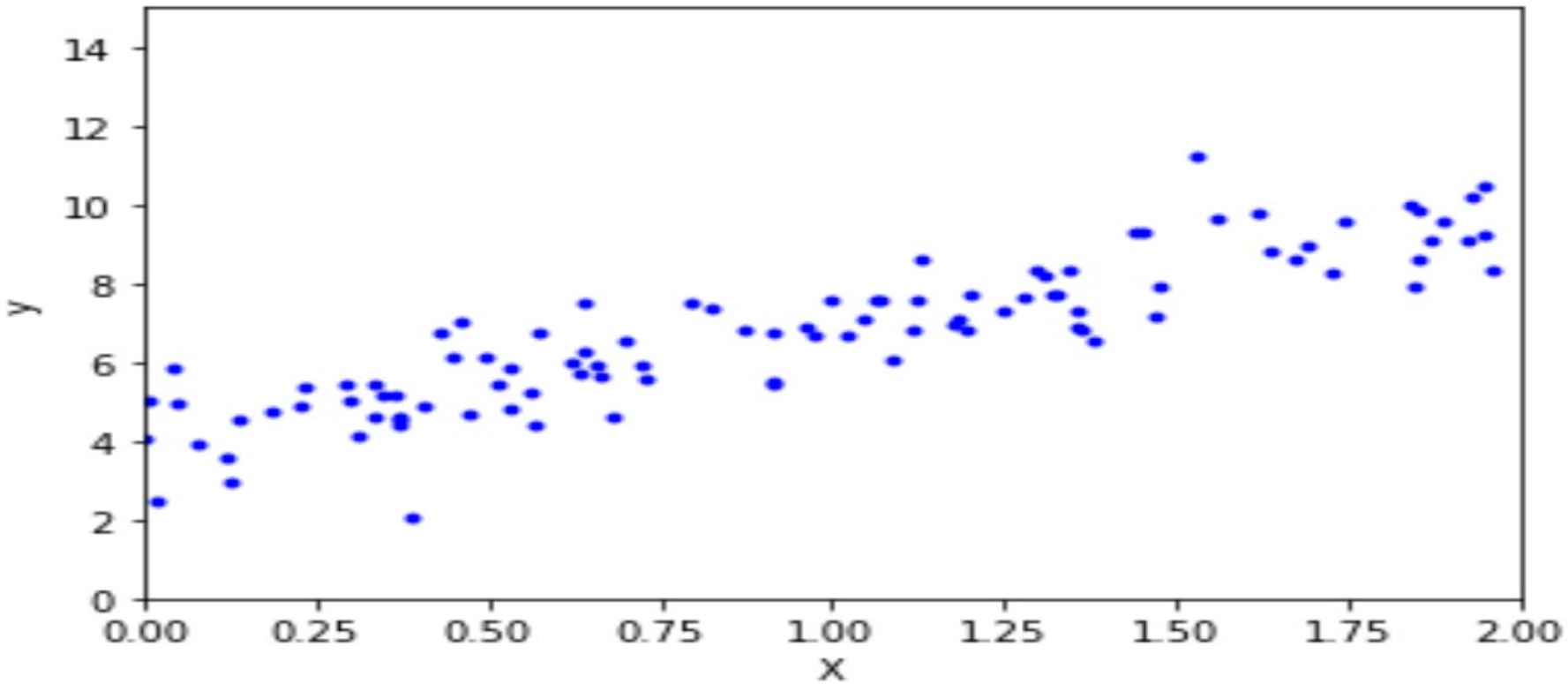
Generate some linear-looking data to test the equation in scikit-learn, using a random plot.

```
import numpy as np

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

Then plot using matplotlib.pyplot:

```
import matplotlib.pyplot as plt

plt.ylabel('y')
plt.xlabel('X')
plt.plot(X, y, 'bo')
```

The plot



An example continued

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([[4.1285339],[2.97567591]])
```

- *np.linalg* module: NumPy's linear algebra module
- *inv*() function: Compute the inverse of a matrix
- *dot*() function: matrix multiplication
- Close to original 4 and 3 enough, but the noise made it impossible to recover the exact parameters of the original function

An example continued

Now make predictions using $\hat{\theta}$

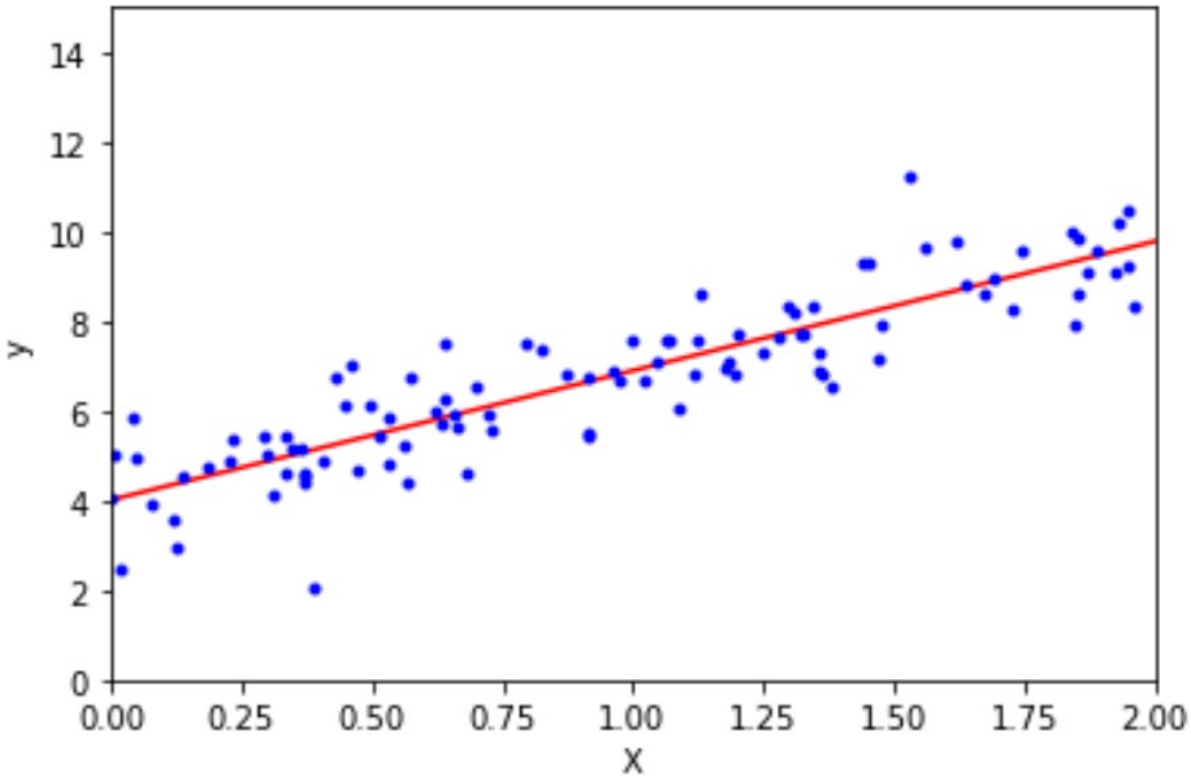
```
X_new = np.array([[0], [2]])
X_new_b = np.c_[np.ones((2, 1)), X_new]
y_predict = X_new_b.dot(theta_best)
print(y_predict)
```

```
array([[4.1285339],[10.14603678]])
```

Plot the model's predictions:

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

The plot



scikit-learn's LinearRegression

Using scikit-learn's LinearRegression:

```
from sklearn.linear model import LinearRegression
lin_reg = LinearRegression()
lin_reg.fit(X, y)
print(lin_reg.intercept_, lin_reg.coef_)
print(lin_reg.predict(X_new))
```

Gradient descent

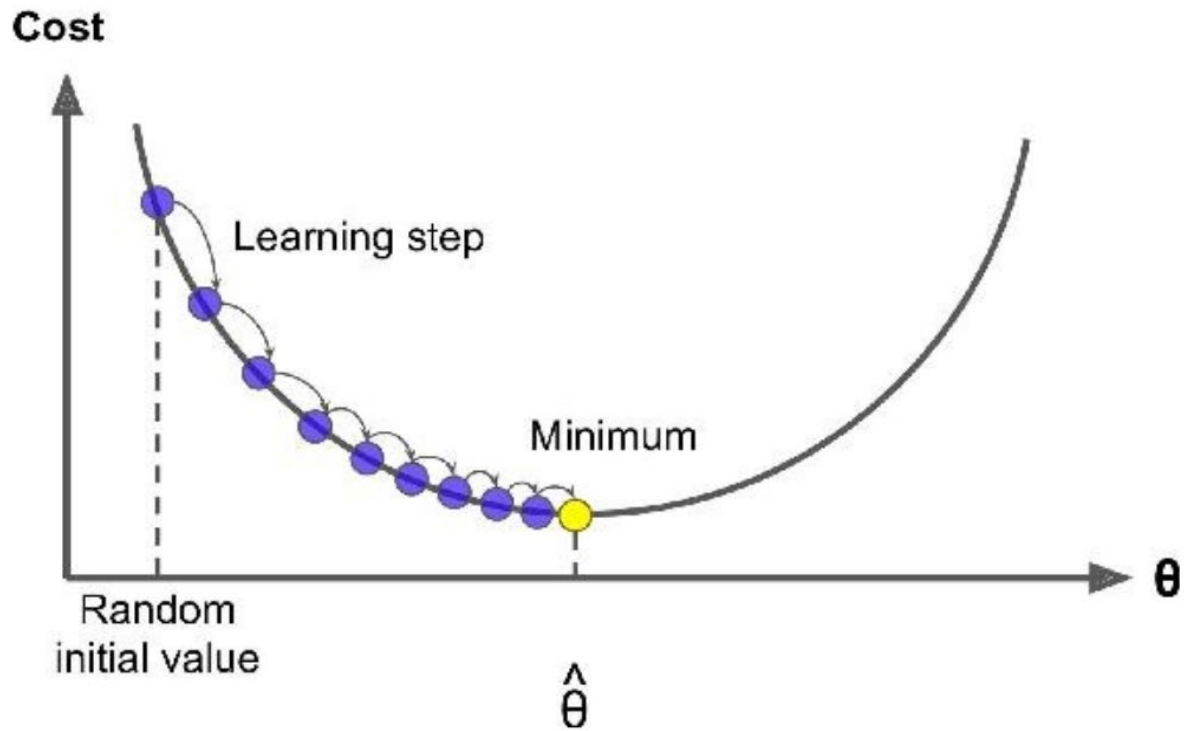
A generic optimization algorithm capable of finding optimal solutions to a wide range of problems.
Basic ideas:

- Iteratively tweak parameters until an optimal solution is found
- What strategies used for "tweaking" of the parameters?
- Determine the gradient of the error function and then follow along the descending gradient
- When the gradient becomes zero, then we have a minimum;

More Formally:

- all the parameters in θ are initialised (random initialisation)
- Improve the estimates by iterative incremental steps
- Steps are taken towards the descending slope

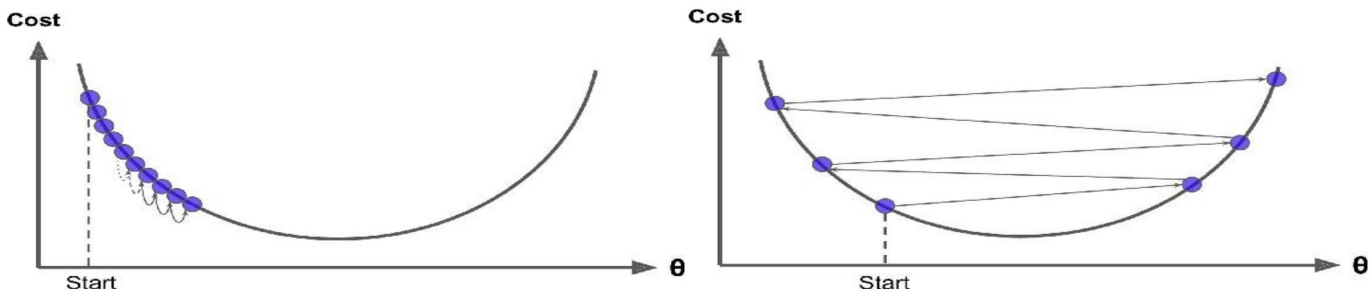
The plot



Learning rate

A critical parameter in gradient descent (GD) is the so-called learning rate:

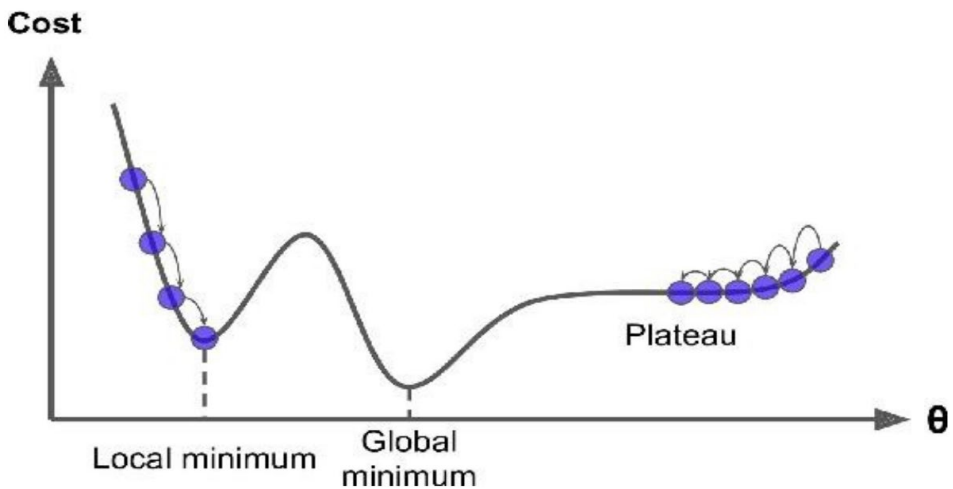
- too small takes more iterations to converge to minimum
- too large learning make it diverge, missing the minimum



learning rate

not all cost functions have contours that are ‘nice’ convenient bowls

- It may have holes, ridges and plateaus, i.e., all sorts of irregularities
- makes convergence to the minimum difficult



learning rate

Fortunately, the *MSE* for linear regression is a nice bowl shape

- it is actually a convex function
- every line segment made by two-points in the function's contour lies on or above the contour
- local minimums are also global minimums
- GD is guaranteed to approach towards the global minimum, provided the learning rate is not too high!

The partial derivative

- Gradient descent (GD) requires knowing the gradient of the cost functions at a "point" for each parameter θ_j
- Knowing the rate-of-change of the cost function as we change θ_j a little bit
- this is known as the partial derivative
- Intuitively, the partial derivative with respect to θ_j gives us the slope at point $\Theta = (\theta_0, \theta_1, \theta_2, ..., \theta_m)$ in the "hyperspace" along the θ_j axis

The partial derivative

The partial derivative of the cost function $MSE(\theta)$ with respect to parameter θ_j, denoted: $\frac{\partial}{\partial \theta_j}\left(MSE(\theta)\right)$:

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

Batch gradient descent

- Instead of computing these partial derivatives individually, you can compute them all in one go
- $\nabla_\theta MSE(\theta) = \begin{pmatrix} \frac{\partial}{\partial \theta_0}(MSE(\theta)) \\ \frac{\partial}{\partial \theta_1}(MSE(\theta)) \\ \frac{\partial}{\partial \theta_2}(MSE(\theta)) \\ \vdots \\ \frac{\partial}{\partial \theta_n}(MSE(\theta)) \end{pmatrix} = \frac{2}{m} X^T \cdot (X \cdot \theta - y)$
- This formula calculates over the full training set X at once, i.e., the full batch.
- The vector of all the partial derivatives $\nabla_\theta MSE(\theta)$ is called the gradient vector

Batch gradient descent

- The gradient vector points uphill, so move in the opposite direction to go downhill
- Subtracting $\nabla_\theta MSE(\theta)$ from θ: $\theta^{(nextstep)} = \theta - \eta \nabla_\theta MSE(\theta)$
- learning rate η: determine the size of the downhill step

An implementation using numpy

```
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
print(theta)

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

Stochastic Gradient Descent

- A drawback of gradient descent is that the whole training set is used
- The size of the training set greatly influences the algorithm's performance: much slower when the training set is very large
- Stochastic gradient descent (SGD) is at the opposite extreme
- At each step, SGD just chooses a random instance in the training set
- Only the gradient for that particular instance is used in the calculation
- Makes the algorithm much faster on large data sets
- Drawbacks: when the cost functions is irregular, the algorithm can escape local minima

An implementation of SGD using python

```
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

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

Using scikit-learn's SGDRegressor class

```
from sklearn.linear model import SGDRegressor
sgd_reg = SGDRegressor(max iter=50,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

- Between Batch and stochastic GD
- At each step, the gradient vector can be computed from a random subset of the training set instead of the whole batch or a single instance
- The (usually small) random subset instances are called mini-batch
- Can take advantage of optimisation on matrix operations from the hardware, e.g., GPUs