Optimization Using Gradient Descent

Dr. Ram Prasad K
VisionCog R&D

ram.krish@visioncog.com
https://www.visioncog.com

Machine Learning

Machine Learning:

An algorithmic way of *making sense (learning) from* ***data***.

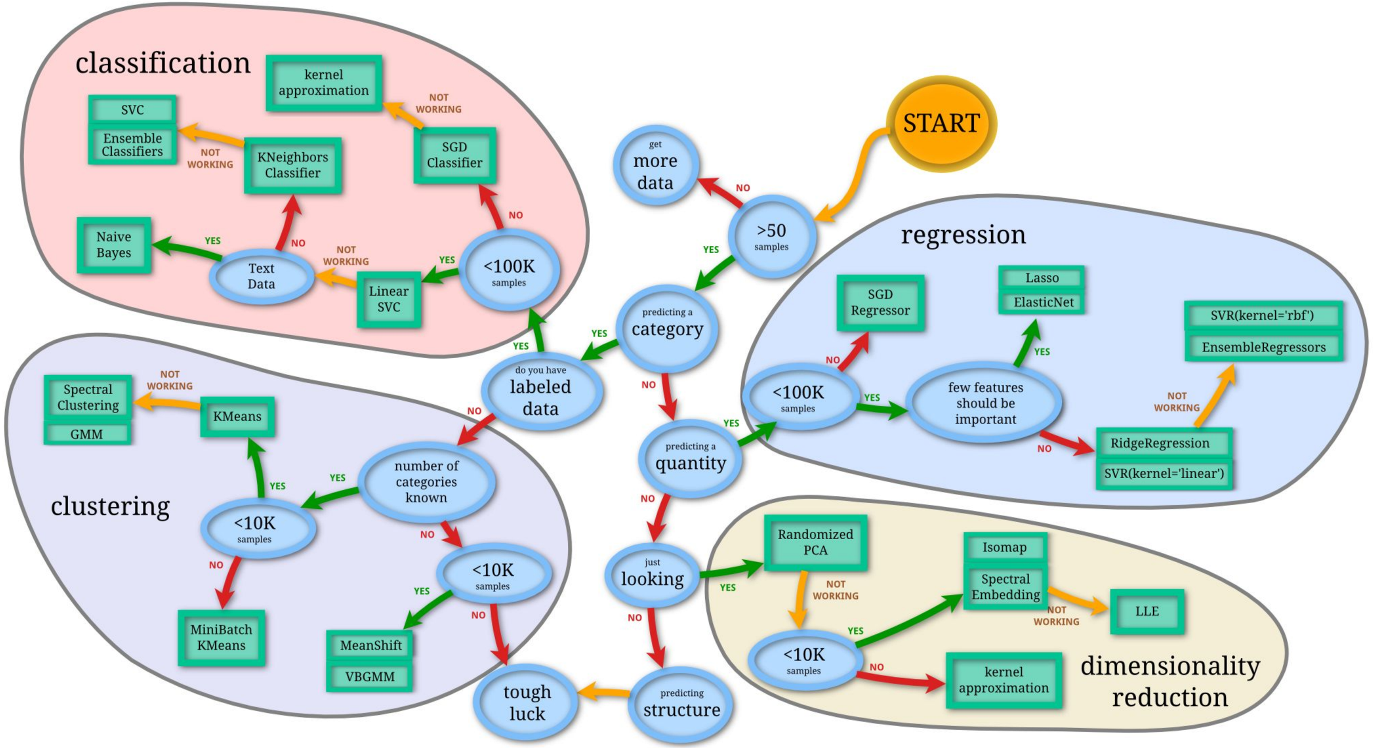
Applications:

- Spam filters (***Classification***)
- Predict height based on weight and age (***Regression***)
- Online recommendation systems (***Clustering***)
- Visualizing multidimensional data (***Dimensionality reduction***)

Scikit Learn

- Machine Learning library - in **Python**
- Simple and efficient tools for data analysis
- Built on NumPy, SciPy, and matplotlib
- API is remarkably well designed

Machine Learning



Linear Regression

Dependent and Independent variable

Expression	Independent	Dependent
$y = 3 + 2x$	x	y
$y = x^2 - 2x$	x	y
$z = 5x^2 + 8y^3$	x, y	z

Regression:

Modeling a relationship between ***dependent*** and ***independent*** variables for ***prediction***.

Linear Regression

Simple Linear Regression or ***Univariate Linear Regression***.

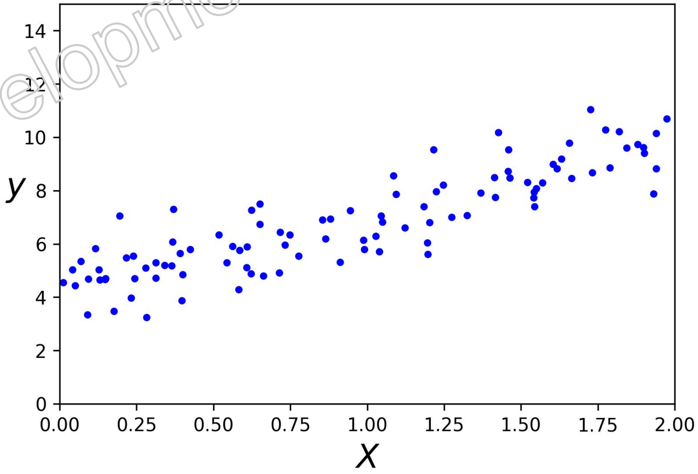
Only one independent variable

Multiple Linear Regression or ***Multivariate Linear Regression***.

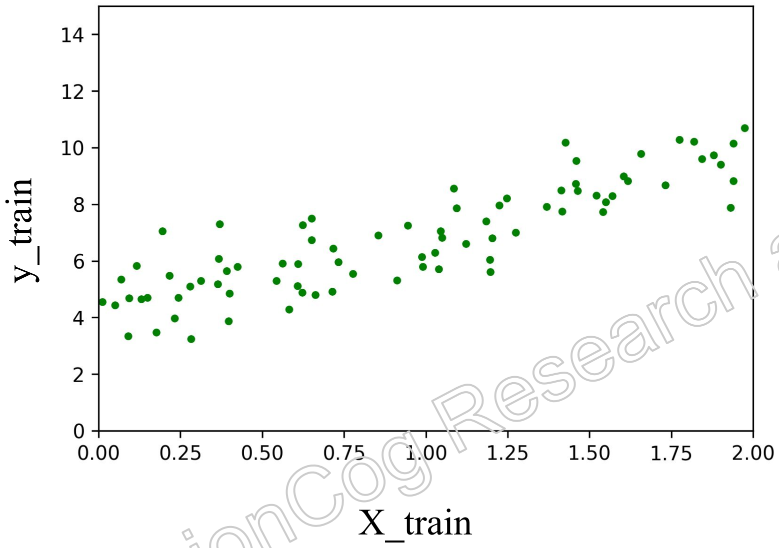
More than one independent variable

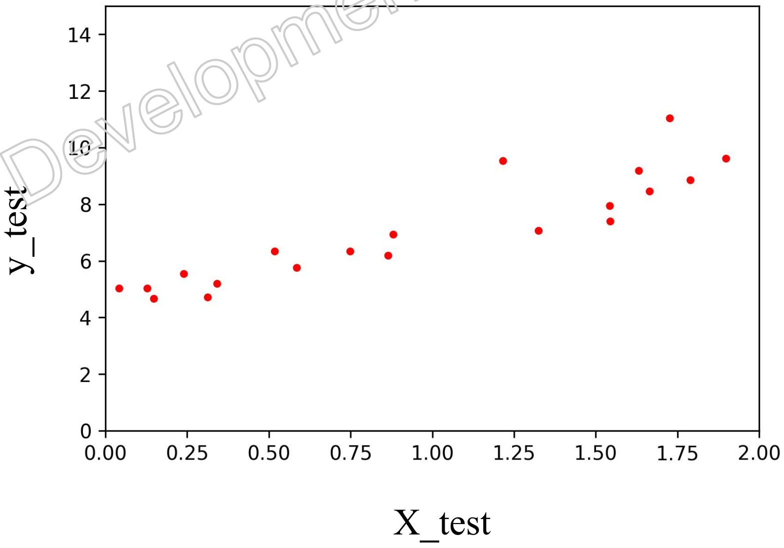
Linear Regression

```python
import numpy as np
np.random.seed(42)

X = 2 * np.random.rand(100, 1)

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


from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size = 0.20, random_state = 42)
```

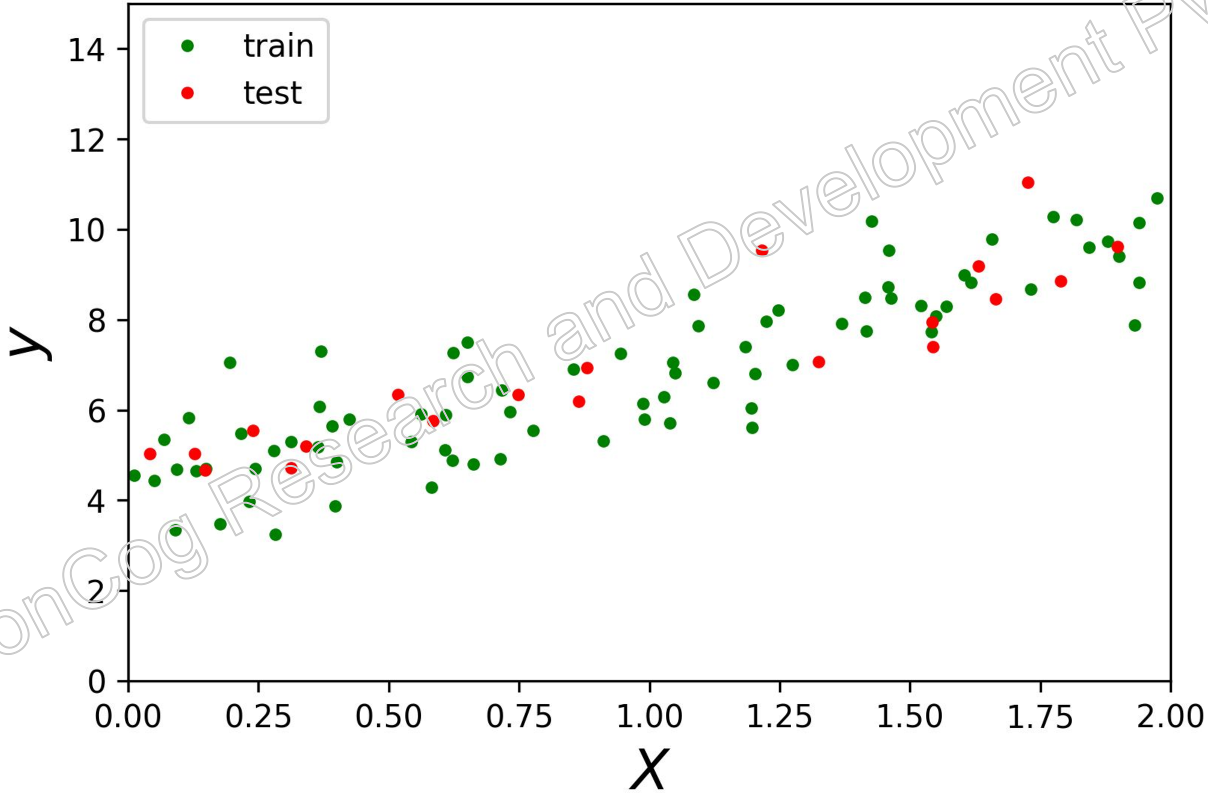


Linear Regression

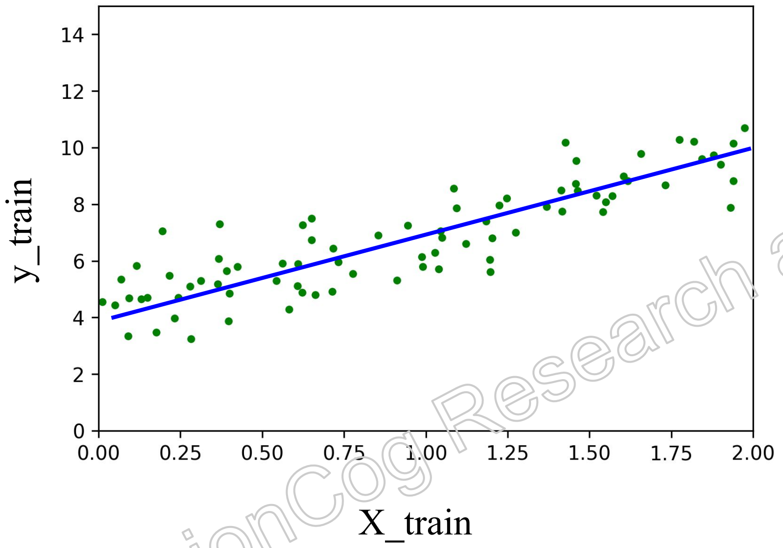


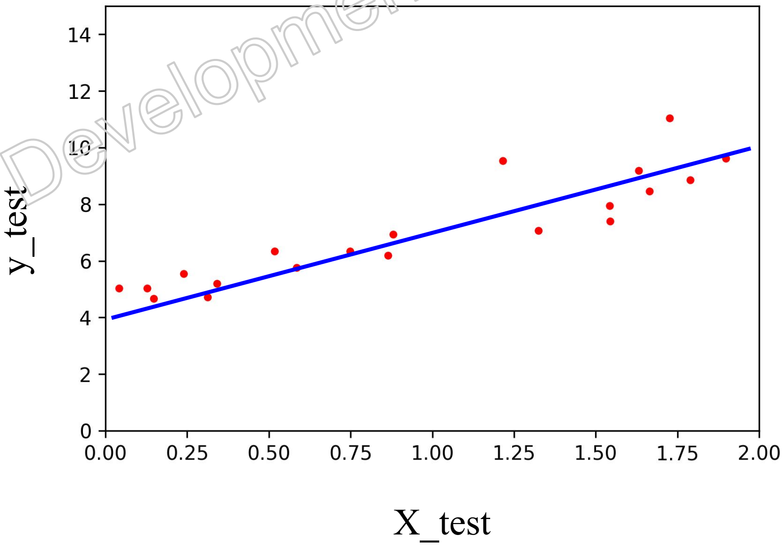


Linear Regression



Linear Regression





Linear Regression

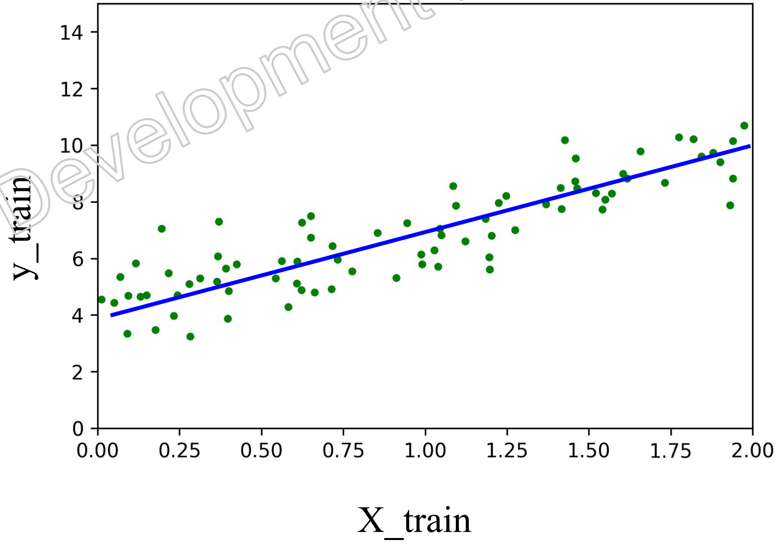
Mathematical model for Simple Linear Regress

model parameters

$$\hat{y} = \theta_0 + \theta_1 x$$

intercept

coefficient



The line models the relationship between cake independent and dependent variable.

Linear Regression

General/Multiple Linear Regression

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

$\hat{y}$ 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

θ_0 is the intercept (also called ***bias*** term)

Linear Regression

Vectorized general form

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

θ is the models ***parameter*** vector

θ_0 is the *bias/intercept*

$\theta_1, \theta_2, \ldots, \theta_n$ are ***coefficients*** or feature weights.

$\mathbf{x}$ is the ***feature*** vector x_0 to x_n with x_0 always 1

$\theta^T \cdot \mathbf{x}$ is the dot product of θ^T and $\mathbf{x}$

h_θ is the ***hypothesis*** function using model parameters θ

Linear Regression

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

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

cost function

Normal Equation

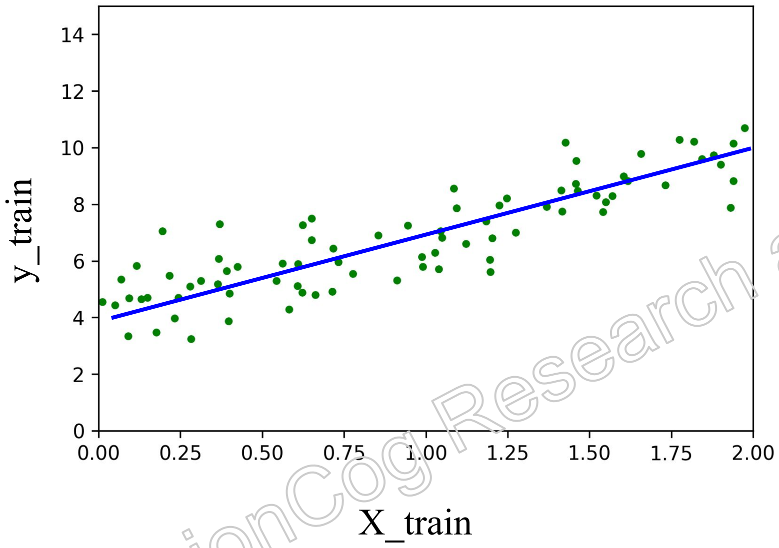
To find the parameters, we have a closed-form solution:

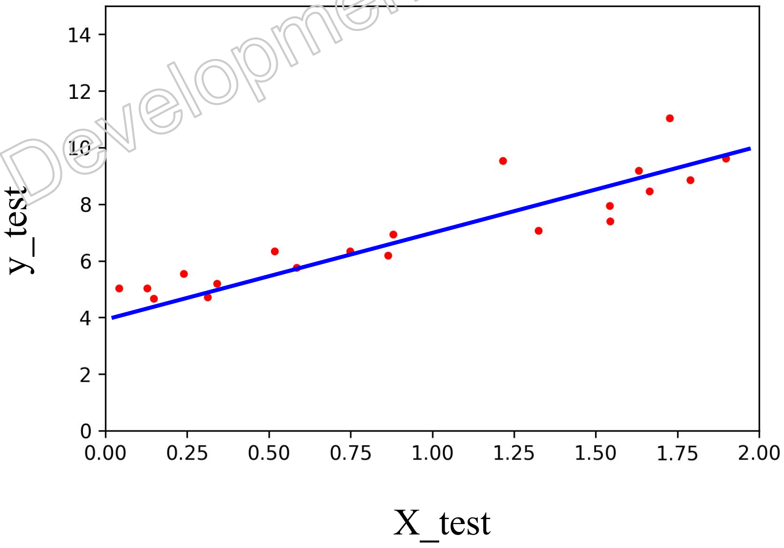
$$\hat{\theta} = (\mathbf{X}^T \cdot \mathbf{X})^{-1} \cdot \mathbf{X}^T \cdot \mathbf{y}$$

$\hat{\theta}$ is the value of θ that minimizes the cost function (least squares)

$\mathbf{y}$ is the vector of target values

Linear Regression





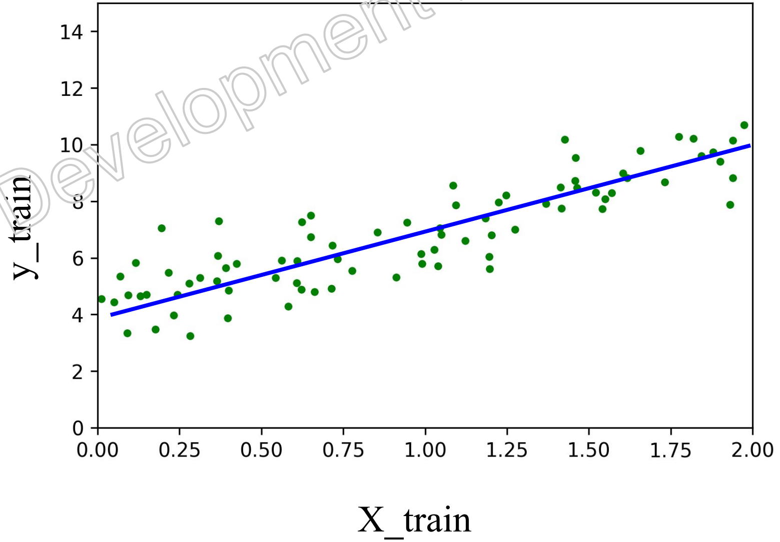
Linear Regression

Mathematical model for Simple Linear Regress

model
parameters

$$\hat{y} = \theta_0 + \theta_1 x$$

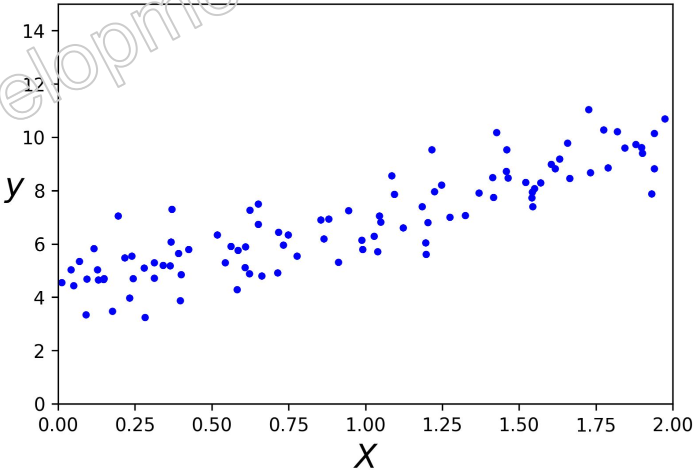
intercept coefficient



The line models the relationship between cake independent and dependent variable.

Linear Regression

```python
import numpy as np
np.random.seed(42)

X = 2 * np.random.rand(100, 1)

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


from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size = 0.20, random_state = 42)
```



Linear Regression

```
X_train_b = np.concatenate([np.ones((80, 1)), X_train], axis=1)

from numpy.linalg import inv

THETA_NE = inv(X_train_b.T.dot(X_train_b)).dot(X_train_b.T).dot(y_train)

print(THETA_NE)

# [[4.14291332]
#  [2.79932366]]
```

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

Linear Regression

```python
from sklearn.linear_model import LinearRegression

model = LinearRegression()

model.fit(X_train, y_train)

print(model.intercept_)
print(model.coef_)

# [4.14291332]
# [[2.79932366]]

score = model.score(X_test, y_test)
print(score)

# 0.8072059636181392
```

```python
print(THETA_NE)

# [[4.14291332]
#  [2.79932366]]
```

Gradient Descent

Normal Equation - Analytical solution

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

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

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

This operation involves ***matrix inversion*** which is a costly operation.

Complexity of matrix inversion is $O(n^{2.4})$ to $O(n^3)$.

When number of ***feature*** increases (more than 1 lakh), this technique will become slow.

If the number of ***training*** instances does not fit in the memory, then also this is an issue.

GRADIENT DESCENT

Derivative

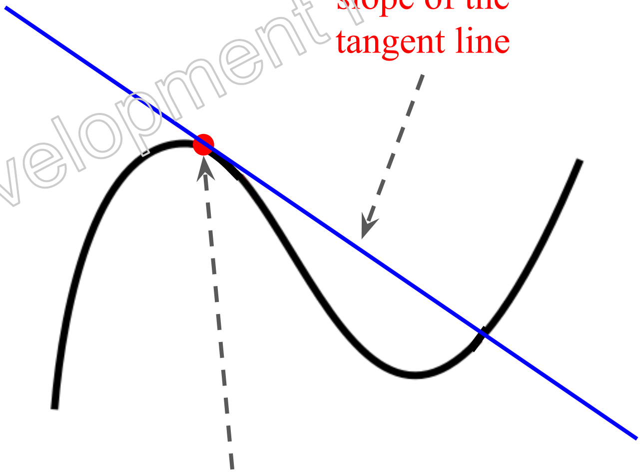
$y = f(x)$ functions of single variable

$\left.\begin{array}{c} f'(x) \\ \\ \dfrac{dy}{dx} \end{array}\right\} \Rightarrow$ rate at which value of ***y*** changes w.r.t change of variable ***x***

$$f(x + \epsilon) \approx f(x) + \epsilon f'(x)$$

Relationship capturing how small change in input influences the output

slope of the tangent line



derivative of function marked at point

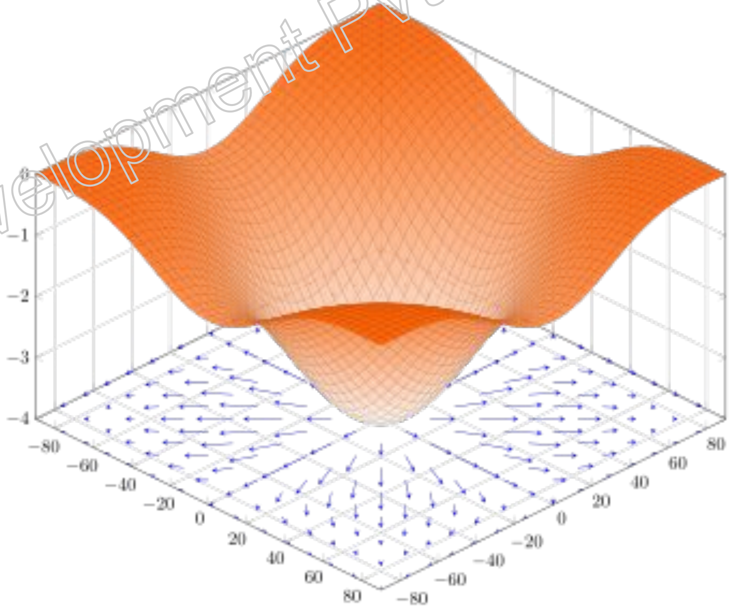
Gradient Descent

Gradients

Multivariable generalization of derivative

$$f(x_1, x_2, x_3)$$

$$\nabla f = \left[\frac{\partial f(x_1, x_2, x_3)}{\partial x_1}, \frac{\partial f(x_1, x_2, x_3)}{\partial x_2}, \frac{\partial f(x_1, x_2, x_3)}{\partial x_3} \right]$$



Gradient Descent

Jacobian

$$f : \mathbb{R}^n \rightarrow \mathbb{R}^m$$

$$[x_1, x_2, \ldots, x_n] \rightarrow [f_1, f_2, \ldots, f_m]$$

$$J = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_n} \end{pmatrix}$$

Gradient Descent

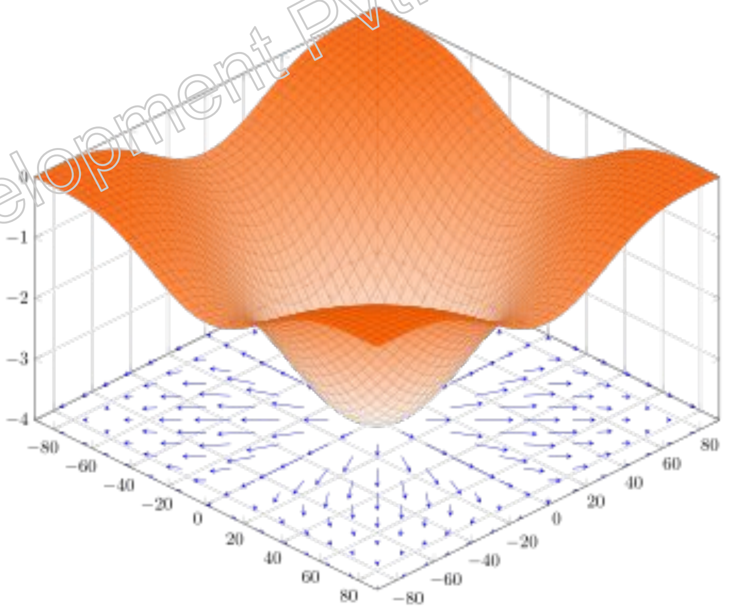
Hessian

$$f(x_1, x_2, x_3)$$

$$\nabla f = \left[\frac{\partial f(x_1,x_2,x_3)}{\partial x_1}, \frac{\partial f(x_1,x_2,x_3)}{\partial x_2}, \frac{\partial f(x_1,x_2,x_3)}{\partial x_3}\right]$$

$$f(x_1, x_2, x_3, \ldots, x_n)$$

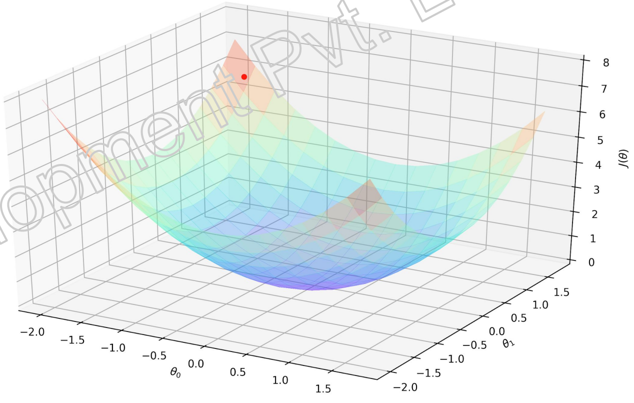
$$H = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{pmatrix}$$

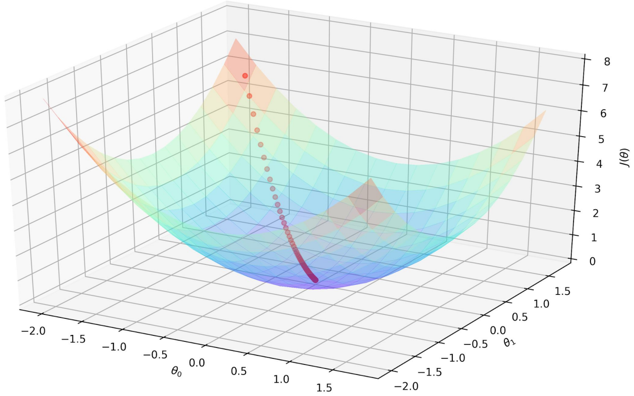


Gradient descent

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

- Initialize the parameters ***randomly***.
- Calculate the ***error*** using cost function.
- Make small change to parameter (***learning rate***).
- Again calculate ***error***.
- Repeat until error converges to a ***minimum***.





Gradient Descent

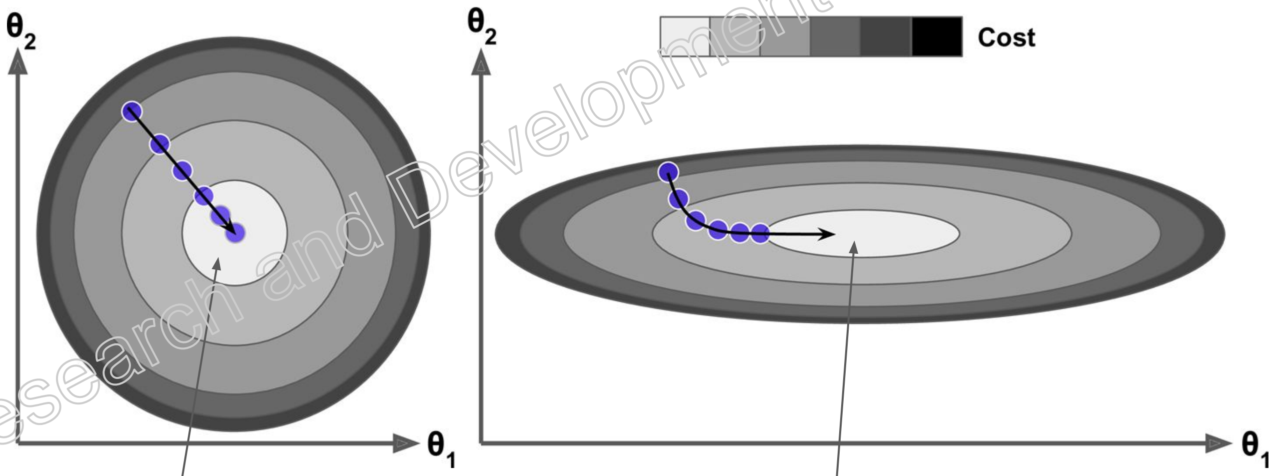
Feature scaling

Normalization

$$X' = \frac{X - X_{min}}{X_{max} - X_{min}}$$

Standardization

$$X' = \frac{X - \mu}{\sigma}$$



With feature scaling, optimization is faster

Slower optimization without feature scaling

Gradient Descent

Batch Gradient Descent

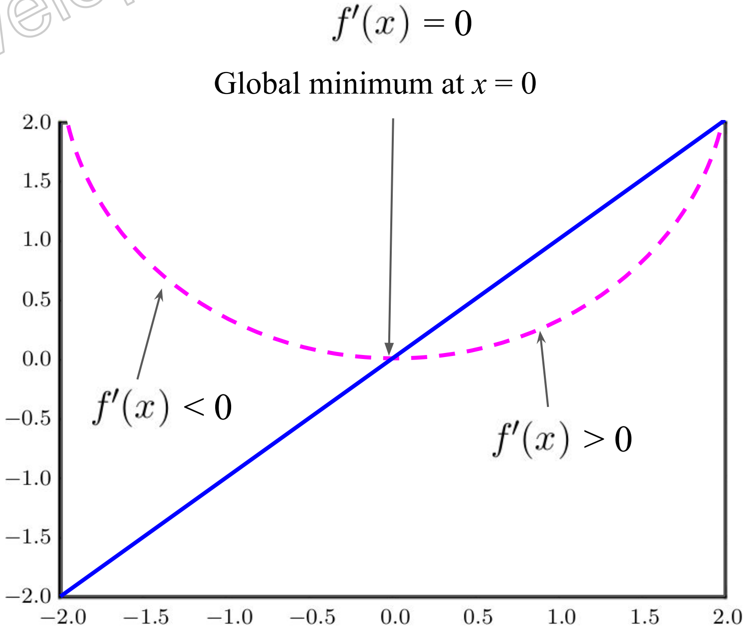
Uses the whole training set for parameter optimization.

Gradient descent usually faster than Normal Equation method for large number of features.

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

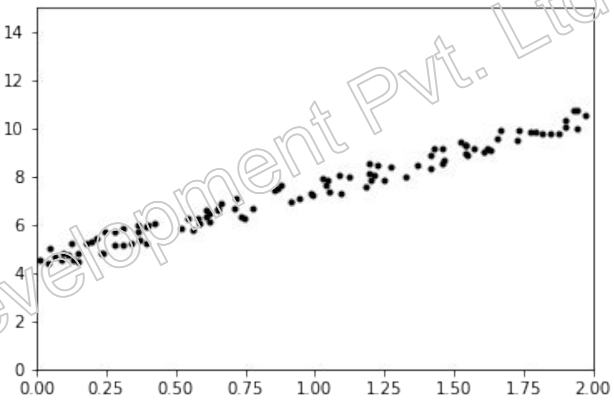
$$\nabla_{\boldsymbol{\theta}} \text{MSE}(\boldsymbol{\theta}) = \begin{pmatrix} \frac{\partial}{\partial \theta_0} \text{MSE}(\boldsymbol{\theta}) \\ \frac{\partial}{\partial \theta_1} \text{MSE}(\boldsymbol{\theta}) \\ \vdots \\ \frac{\partial}{\partial \theta_n} \text{MSE}(\boldsymbol{\theta}) \end{pmatrix}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

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



Gradient Descent

```
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(42)
X = 2 * np.random.rand(100,1)
y = 4 + 3 * X + np.random.rand(100,1)

X_b = np.concatenate([np.ones((100,1)), X], axis = 1)
```




```
from numpy.linalg import inv
from numpy import dot, transpose

THETA_NE = dot(inv(dot(transpose(X_b), X_b)), dot(transpose(X_b), y))
```

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

```
array([[4.51359766],
       [2.98323418]])
```

Gradient Descent

```python
from numpy.linalg import inv
from numpy import dot, transpose

THETA_NE = dot(inv(dot(transpose(X_b), X_b)), dot(transpose(X_b), y))
```

```
array([[4.51359766],
       [2.98323418]])
```

```python
from sklearn.linear_model import LinearRegression

model = LinearRegression()
model.fit(X, y)

model.intercept_, model.coef_
```

```
(array([4.51359766]), array([[2.98323418]]))
```

Batch Gradient Descent

```python
eta = 0.1        # Learning rate
n_itr = 1000
m = 100          # Number of samples

# Random initialization of parameters
theta = np.random.rand(2,1)

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

theta
```

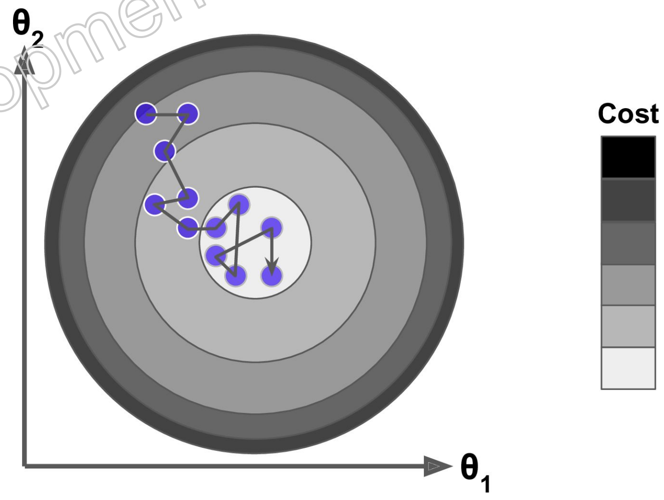
$$\frac{2}{m}\mathbf{X}^T \cdot (\mathbf{X} \cdot \theta - y)$$

```
array([[4.51359766],
       [2.98323418]])
```

Gradient Descent

Stochastic Gradient Descent

- Picks one instance at a time randomly.
- Faster and helps in situations with huge training sets (in billions).
- Randomness helps to escape from local minima.



- For some irregular cost function, it might keep jumping and never settle at minimum.
- Use simulated annealing to solve this issue.
  - Started with larger learning rate and then gradually decrease.

Gradient Descent

Stochastic Gradient Descent

```
from sklearn.linear_model import SGDRegressor

sgdRegressor = SGDRegressor(n_iter=75, penalty=None, eta0=0.1
sgdRegressor.fit(X,y)
sgdRegressor.intercept_, sgdRegressor.coef_
```

```
(array([4.50569579]), array([2.977436]))
```

```
n_epochs = 75
t0, t1 = 5, 50 # learnig schedule hyperparameters
m = 100 # number of samples

# For simulated annealing
def learning_schedule(t):
    return t0/(t+t1)


theta = np.random.rand(2,1)
```

```
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)
```

SGD

```
[[4.51266446]
 [2.98215661]]
```

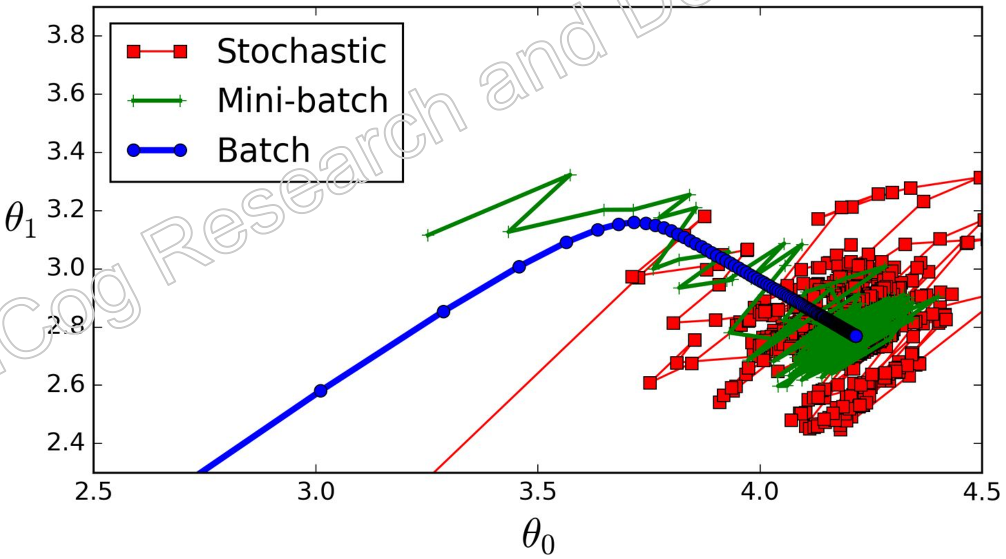
BGD

```
array([[4.51359766],
       [2.98323418]])
```

Gradient Descent

Mini-batch Gradient Descent

- In-between Batch (whole set) and Gradient descent (one sample).
- Works with small random set of training data called ***mini-batches***.



Gradient Descent

Mini-batch Gradient Descent

```
n_iterations = 50
minibatch_size = 20

np.random.seed(42)
theta = np.random.randn(2,1)

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

SGD

```
[[4.51266446]
 [2.98215661]]
```

BGD

```
array([[4.51359766],
       [2.98323418]])
```

```
t = 0
for epoch in range(n_iterations):
    shuffled_indices = np.random.permutation(m)

    X_b_shuffled = X_b[shuffled_indices]
    y_shuffled = y[shuffled_indices]

    for i in range(0, m, 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(theta) - yi)

        eta = learning_schedule(t)
        theta = theta - eta * gradients

print(theta)
```

M-BGD

```
[[4.52651397]
 [2.99723869]]
```