Machine Learning

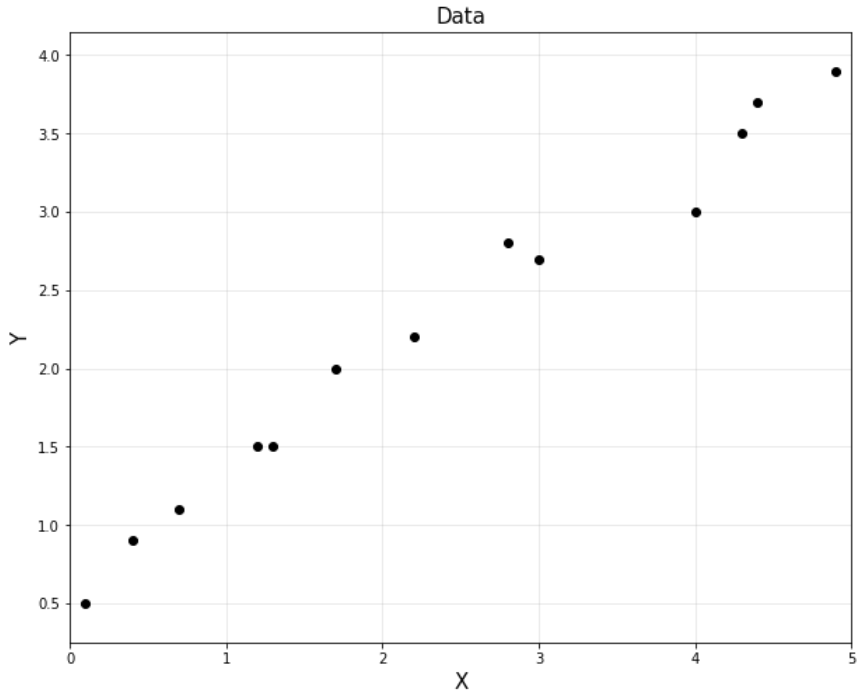
Prof. Seungchul Lee

Industrial AI Lab.

Regression

Assumption: Linear Model

$\hat{y}_i = f(x_i; \theta)$ in general

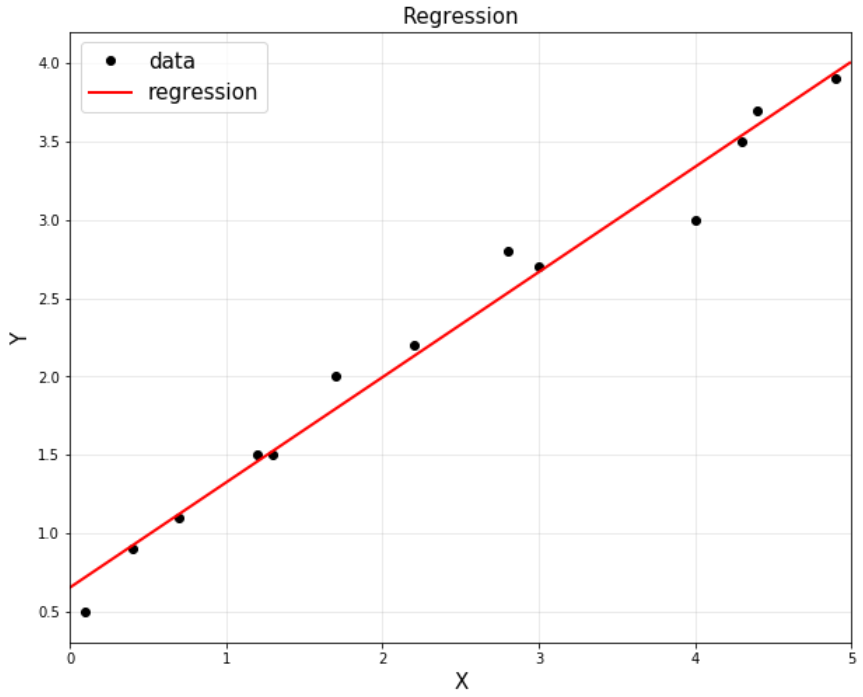


- In many cases, a linear model is used to predict y_i

$$\hat{y}_i = \theta_1 x_i + \theta_2$$

Assumption: Linear Model

$$\hat{y}_i = f(x_i; \theta) \text{ in general}$$



- In many cases, a linear model is used to predict y_i

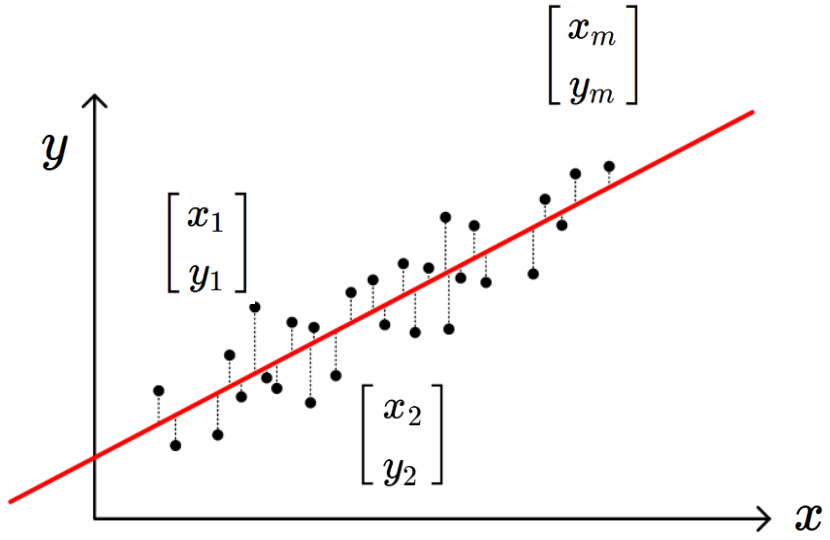
$$\hat{y}_i = \theta_1 x_i + \theta_2$$

Linear Regression

- $\hat{y}_i = f(x_i, \theta)$ in general
- In many cases, a linear model is assumed to predict y_i

$$\text{Given} \begin{cases} x_i : \text{inputs} \\ y_i : \text{outputs} \end{cases}, \text{ Find } \theta_0 \text{ and } \theta_1$$

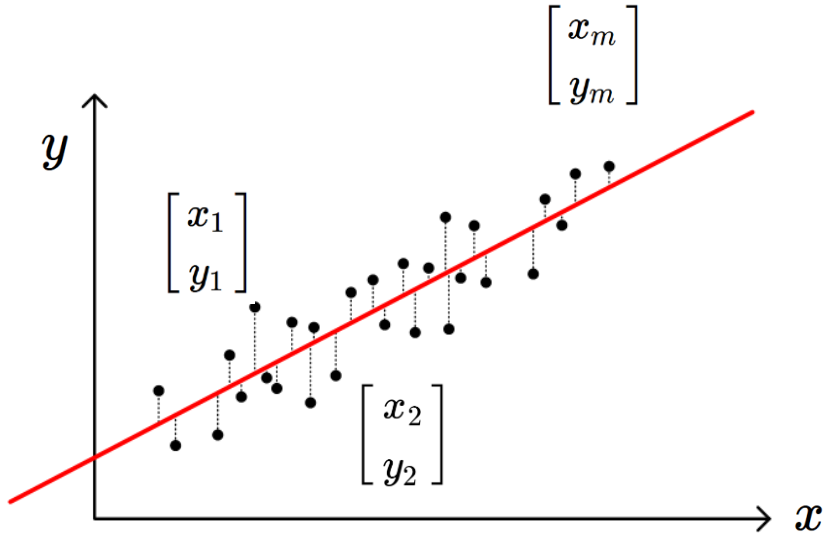
$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix}, \qquad y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} \approx \hat{y}_i = \theta_0 + \theta_1 x_i$$



- $\hat{y}_i$: predicted output
- $\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix}$: model parameters

Linear Regression as Optimization

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix}, \qquad y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} \approx \hat{y}_i = \theta_0 + \theta_1 x_i$$



- How to find model parameters $\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix}$
- Optimization problem

$$\hat{y}_i = \theta_0 + \theta_1 x_i \qquad \text{such that} \qquad \min_{\theta_0, \theta_1} \sum_{i=1}^{m} (\hat{y}_i - y_i)^2$$

Re-cast Problem as Least Squares

- For convenience, we define a function that maps inputs to feature vectors, ϕ

$$\hat{y}_i = \theta_0 + x_i\theta_1 = 1 \cdot \theta_0 + x_i\theta_1$$

$$= \begin{bmatrix} 1 & x_i \end{bmatrix} \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix}$$

$$= \begin{bmatrix} 1 \\ x_i \end{bmatrix}^T \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix}$$

$$= \phi^T(x_i)\theta$$

$$\text{feature vector } \phi(x_i) = \begin{bmatrix} 1 \\ x_i \end{bmatrix}$$

$$\Phi = \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \\ 1 & x_m \end{bmatrix} = \begin{bmatrix} \phi^T(x_1) \\ \phi^T(x_2) \\ \vdots \\ \phi^T(x_m) \end{bmatrix} \quad \Longrightarrow \quad \hat{y} = \begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \vdots \\ \hat{y}_m \end{bmatrix} = \Phi\theta$$

Optimization

$$\min_{\theta_0,\theta_1} \sum_{i=1}^{m} (\hat{y}_i - y_i)^2 = \min_{\theta} \|\Phi\theta - y\|_2^2 \qquad \left(\text{same as } \min_{x} \|Ax - b\|_2^2\right)$$

$$\text{solution } \theta^* = (\Phi^T\Phi)^{-1}\Phi^T y$$

- Scalar Objective: $J = \|Ax - y\|^2$

$$\begin{aligned} J(x) &= (Ax - y)^T(Ax - y) \\ &= \left(x^T A^T - y^T\right)(Ax - y) \\ &= x^T A^T Ax - x^T A^T y - y^T Ax + y^T y \end{aligned}$$

$$\begin{aligned} \frac{\partial J}{\partial x} &= A^T Ax + \left(A^T A\right)^T x - A^T y - \left(y^T A\right)^T \\ &= 2A^T Ax - 2A^T y = 0 \end{aligned}$$

$$\implies \left(A^T A\right) x = A^T y$$

$$\therefore \quad x^* = \left(A^T A\right)^{-1} A^T y$$

y	$\frac{\partial y}{\partial x}$
Ax	A^T
$x^T A$	A
$x^T x$	$2x$
$x^T Ax$	$Ax + A^T x$

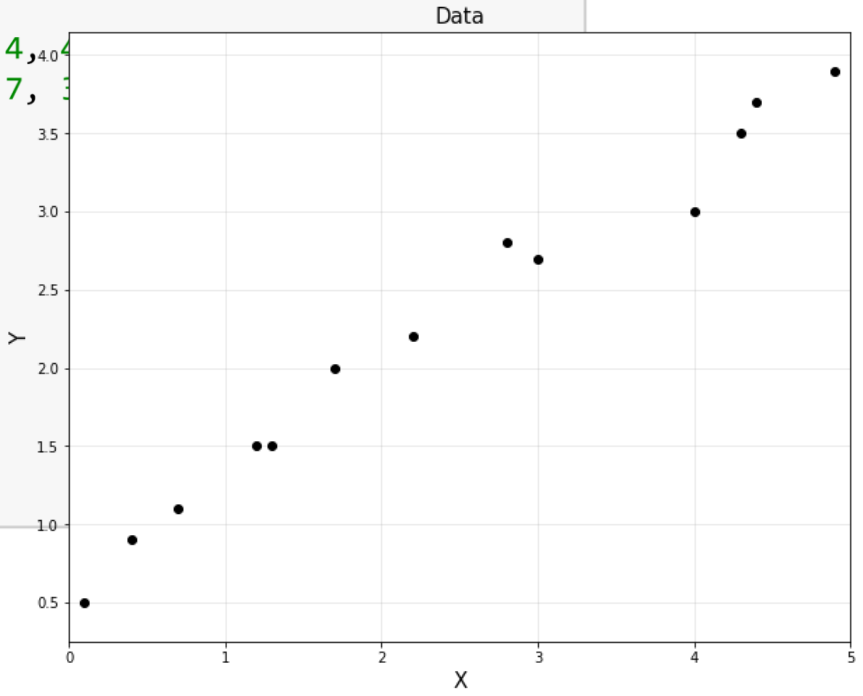
Solve using Linear Algebra

- known as *least square*

$$\theta = (A^T A)^{-1} A^T y$$

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

```
# data points in column vector [input, output]
x = np.array([0.1, 0.4, 0.7, 1.2, 1.3, 1.7, 2.2, 2.8, 3.0, 4.0, 4.3, 4.4,
y = np.array([0.5, 0.9, 1.1, 1.5, 1.5, 2.0, 2.2, 2.8, 2.7, 3.0, 3.5, 3.7,

plt.figure(figsize=(10,8))
plt.plot(x,y,'ko')
plt.title('Data', fontsize=15)
plt.xlabel('X', fontsize=15)
plt.ylabel('Y', fontsize=15)
plt.axis('equal')
plt.grid(alpha=0.3)
plt.xlim([0, 5])
plt.show()
```

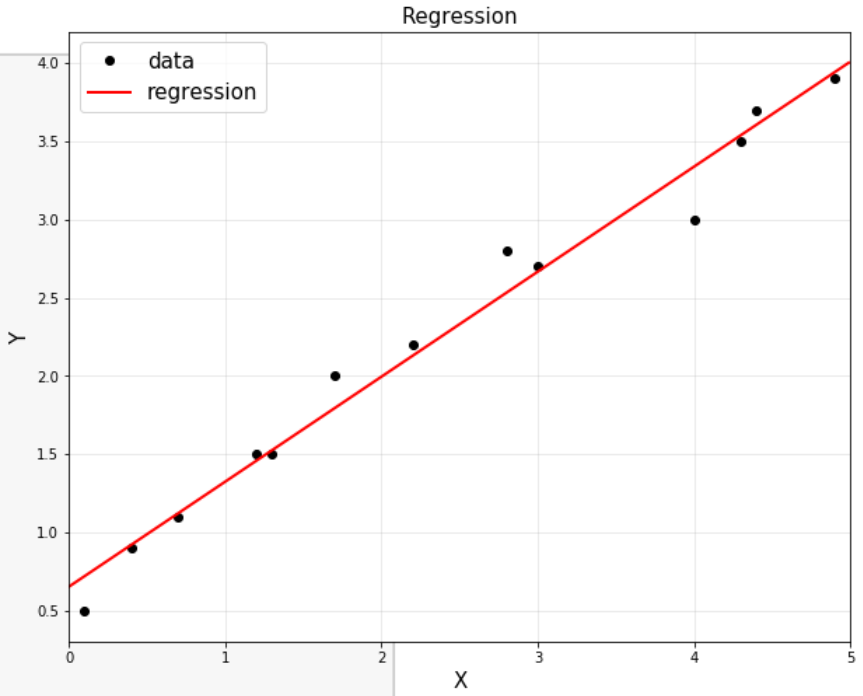


Solve using Linear Algebra

```
m = y.shape[0]
#A = np.hstack([x, np.ones([m, 1])])
A = np.hstack([x**0, x])
A = np.asmatrix(A)

theta = (A.T*A).I*A.T*y

print('theta:\n', theta)
```

```
theta:
 [[0.65306531]
 [0.67129519]]
```

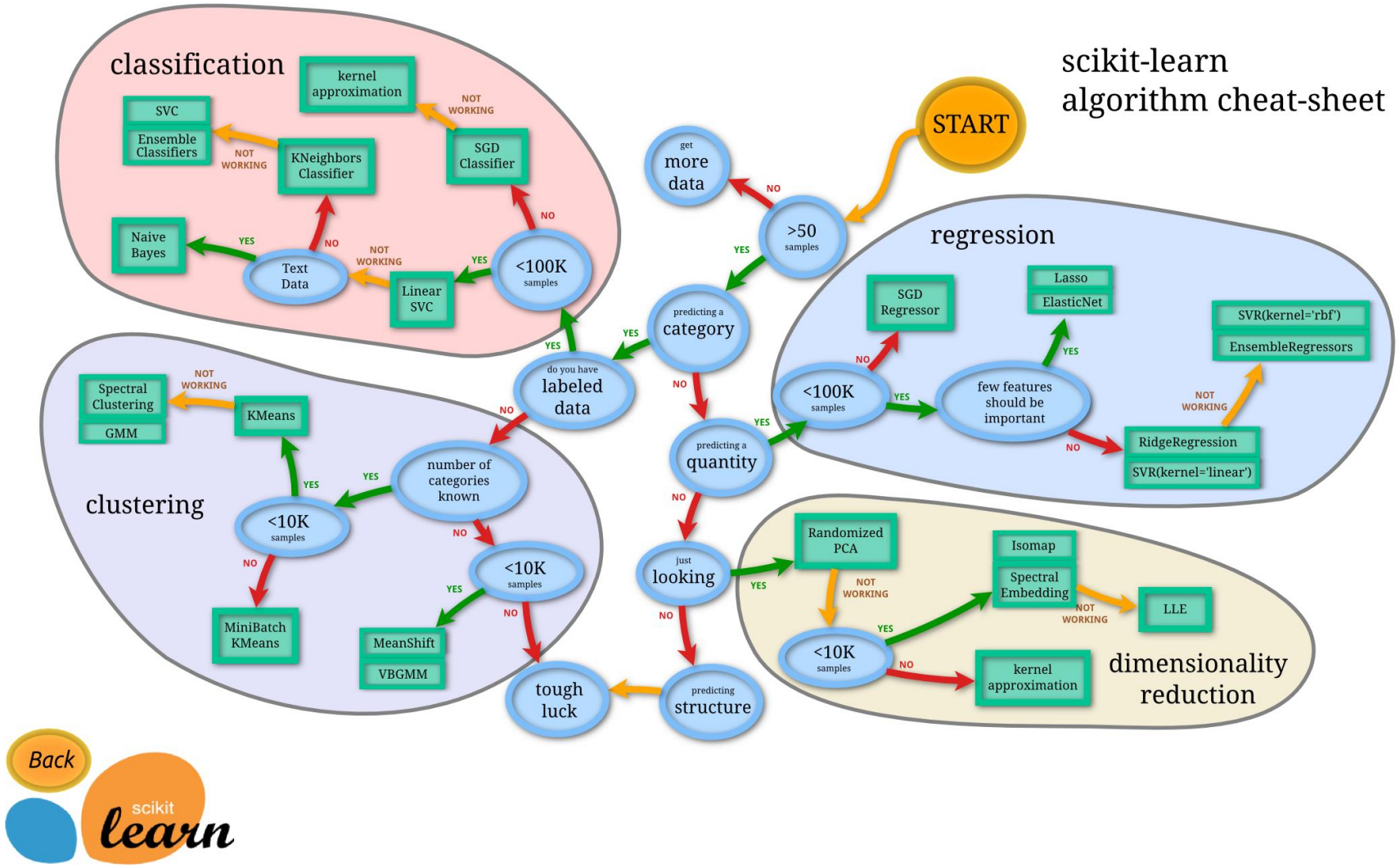
```
# to plot
plt.figure(figsize=(10, 8))
plt.title('Regression', fontsize=15)
plt.xlabel('X', fontsize=15)
plt.ylabel('Y', fontsize=15)
plt.plot(x, y, 'ko', label="data")

# to plot a straight line (fitted line)
xp = np.arange(0, 5, 0.01).reshape(-1, 1)
yp = theta[0,0] + theta[1,0]*xp

plt.plot(xp, yp, 'r', linewidth=2, label="regression")
plt.legend(fontsize=15)
plt.axis('equal')
plt.grid(alpha=0.3)
plt.xlim([0, 5])
plt.show()
```



Scikit-Learn

- Machine Learning in Python
- Simple and efficient tools for data mining and data analysis
- Accessible to everybody, and reusable in various contexts
- Built on NumPy, SciPy, and matplotlib
- Open source, commercially usable - BSD license
- https://scikit-learn.org/stable/index.html#

Scikit-Learn



Scikit-Learn: Regression

```
from sklearn import linear_model
```

```
reg = linear_model.LinearRegression()
reg.fit(x, y)
```

```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,
         normalize=False)
```

```
reg.coef_
```

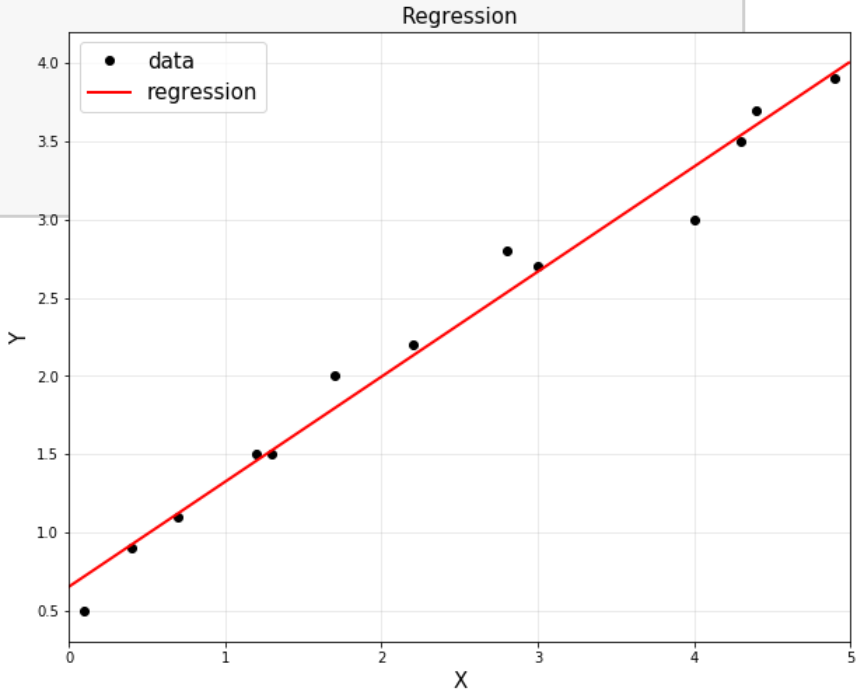
```
array([[0.67129519]])
```

```
reg.intercept_
```

```
array([0.65306531])
```

Scikit-Learn: Regression

```
# to plot
plt.figure(figsize=(10, 8))
plt.title('Regression', fontsize=15)
plt.xlabel('X', fontsize=15)
plt.ylabel('Y', fontsize=15)
plt.plot(x, y, 'ko', label="data")

# to plot a straight line (fitted line)
plt.plot(xp, reg.predict(xp), 'r', linewidth=2, label="regression")
plt.legend(fontsize=15)
plt.axis('equal')
plt.grid(alpha=0.3)
plt.xlim([0, 5])
plt.show()
```



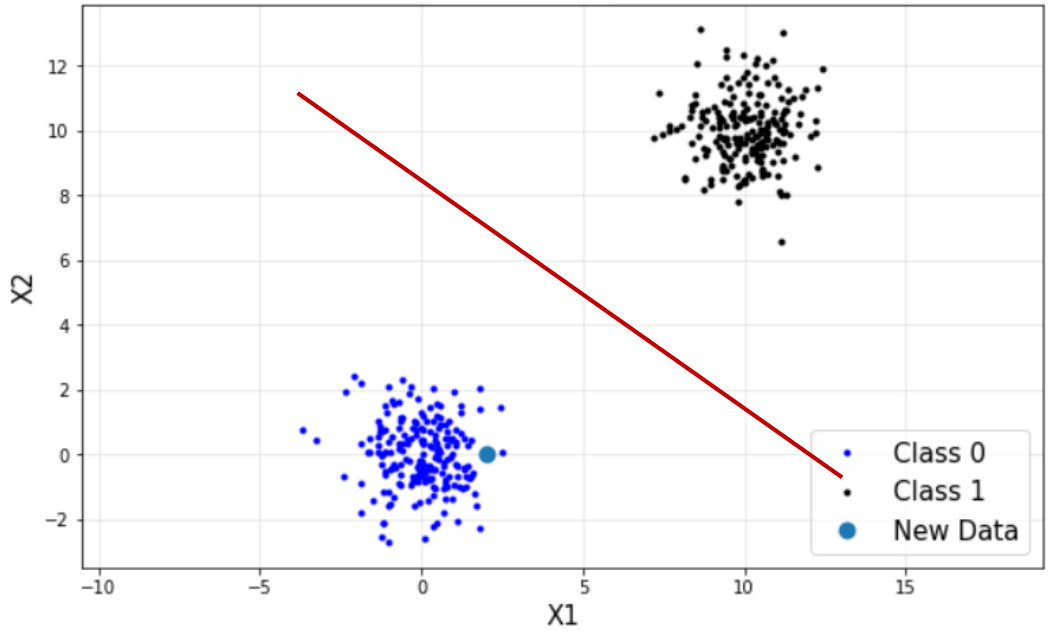
Classification: Perceptron

Classification

- Where y is a discrete value
  - Develop the classification algorithm to determine which class a new input should fall into

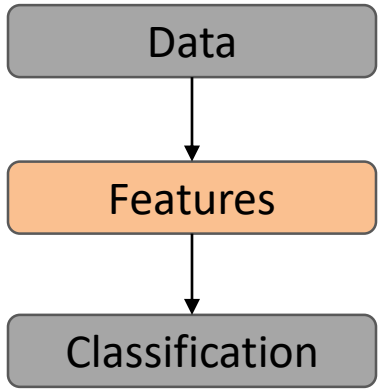
- We will learn
  - Perceptron
  - Logistic regression

- To find a classification boundary



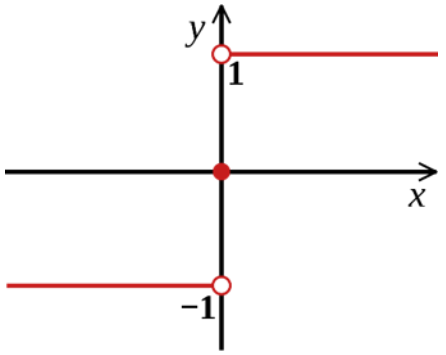
Perceptron

- For input $x = \begin{bmatrix} x_1 \\ \vdots \\ x_d \end{bmatrix}$ 'attributes of a customer'
- Weights $\omega = \begin{bmatrix} \omega_1 \\ \vdots \\ \omega_d \end{bmatrix}$



$$\text{Approve credit if } \sum_{i=1}^{d} \omega_i x_i > \text{threshold},$$

$$\text{Deny credit if } \sum_{i=1}^{d} \omega_i x_i < \text{threshold}.$$



$$h(x) = \text{sign}\left(\left(\sum_{i=1}^{d} \omega_i x_i\right) - \text{threshold}\right) = \text{sign}\left(\left(\sum_{i=1}^{d} \omega_i x_i\right) + \omega_0\right)$$

Perceptron

$$h(x) = \text{sign}\left(\left(\sum_{i=1}^{d} \omega_i x_i\right) - \text{threshold}\right) = \text{sign}\left(\left(\sum_{i=1}^{d} \omega_i x_i\right) + \omega_0\right)$$

- Introduce an artificial coordinate $x_0 = 1$:

$$h(x) = \text{sign}\left(\sum_{i=0}^{d} \omega_i x_i\right)$$

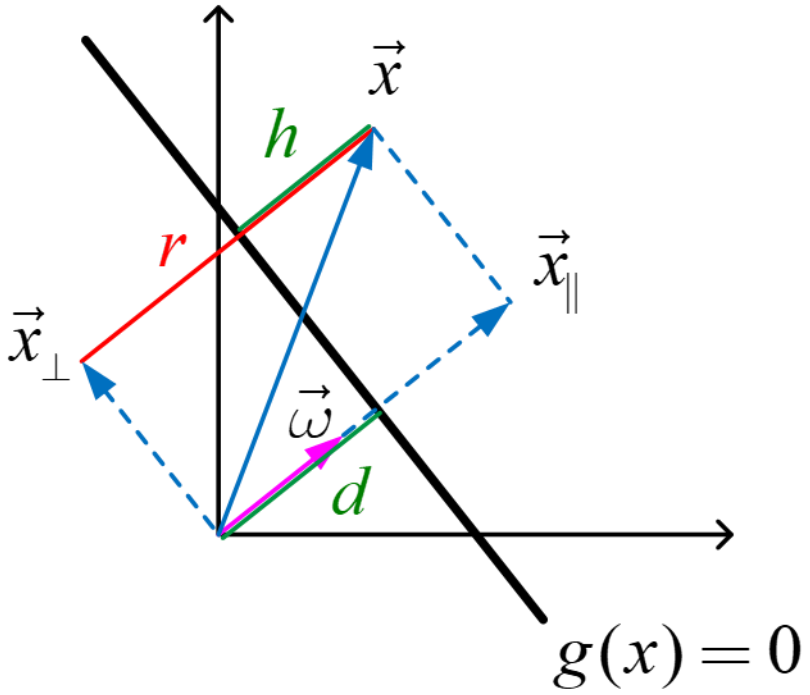
- In a vector form, the perceptron implements

$$h(x) = \text{sign}\left(\omega^T x\right)$$

- Let's see geometrical meaning of perceptron

Distance from a Line

$$\omega = \begin{bmatrix} \omega_1 \\ \omega_2 \end{bmatrix}, \; x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \implies g(x) = \omega^T x + \omega_0 = \omega_1 x_1 + \omega_2 x_2 + \omega_0$$

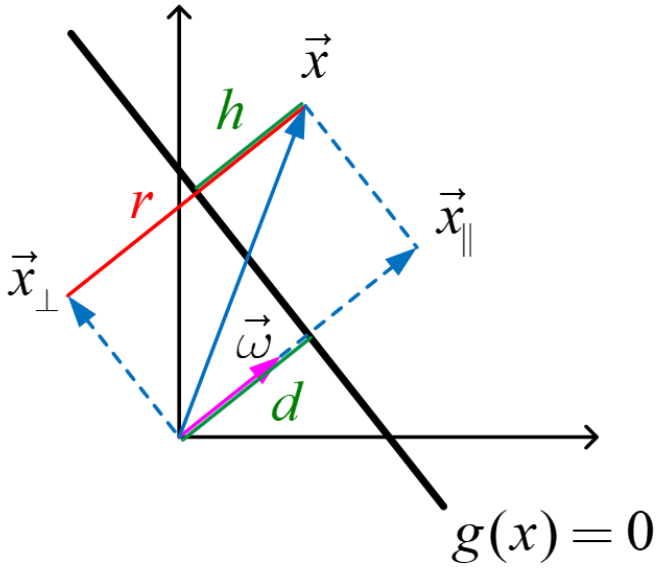


ω

- If $\vec{p}$ and $\vec{q}$ are on the decision line

$$g(\vec{p}) = g(\vec{q}) = 0 \quad \Rightarrow \quad \omega_0 + \omega^T \vec{p} = \omega_0 + \omega^T \vec{q} = 0$$
$$\Rightarrow \quad \omega^T (\vec{p} - \vec{q}) = 0$$

$\therefore$ ω : normal to the line (orthogonal)
$\Longrightarrow$ tells the direction of the line



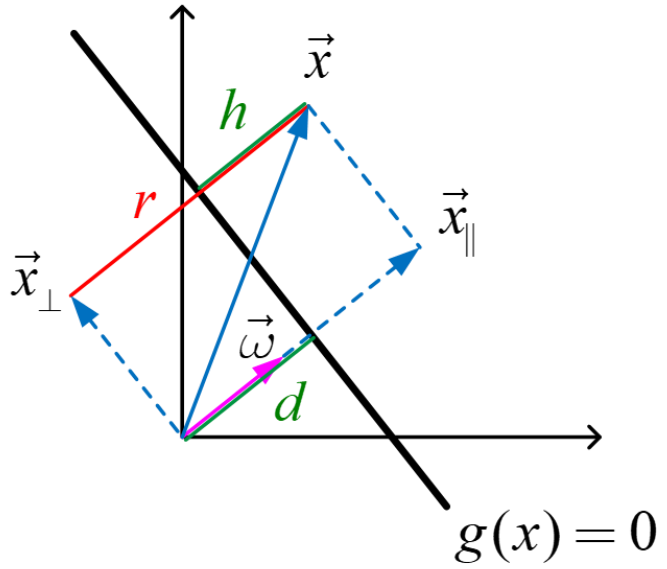
Signed Distance d

- If x is on the line and $x = d\frac{\omega}{\|\omega\|}$ (where d is a normal distance from the origin to the line)

$$g(x) = \omega_0 + \omega^T x = 0$$

$$\Rightarrow \omega_0 + \omega^T d\frac{\omega}{\|\omega\|} = \omega_0 + d\frac{\omega^T\omega}{\|\omega\|} = \omega_0 + d\|\omega\| = 0$$

$$\therefore d = -\frac{\omega_0}{\|\omega\|}$$



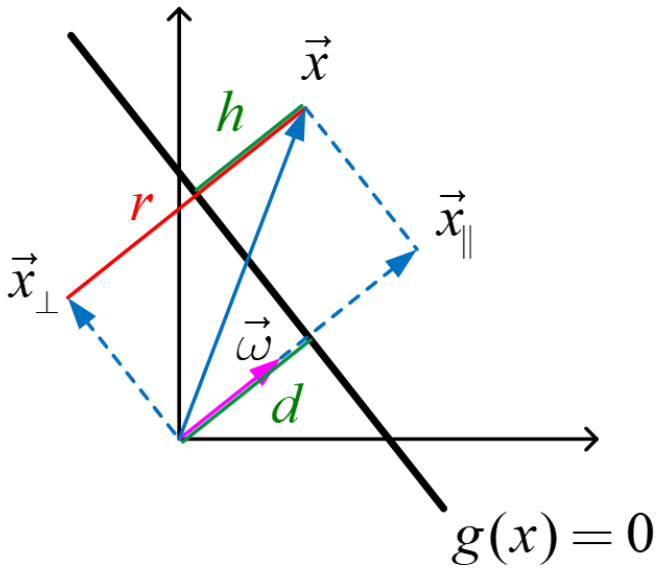
Distance from a Line: h

- for any vector of x

$$x = x_\perp + r\frac{\omega}{\|\omega\|}$$

$$\omega^T x = \omega^T \left(x_\perp + r\frac{\omega}{\|\omega\|} \right) = r\frac{\omega^T\omega}{\|\omega\|} = r\|\omega\|$$

$$\begin{aligned} g(x) &= \omega_0 + \omega^T x \\ &= \omega_0 + r\|\omega\| \qquad (r = d + h) \\ &= \omega_0 + (d+h)\|\omega\| \\ &= \omega_0 + \left(-\frac{\omega_0}{\|\omega\|} + h \right) \|\omega\| \\ &= h\|\omega\| \end{aligned}$$



$$\therefore\ h = \frac{g(x)}{\|\omega\|} \implies \text{orthogonal signed distance from the line}$$

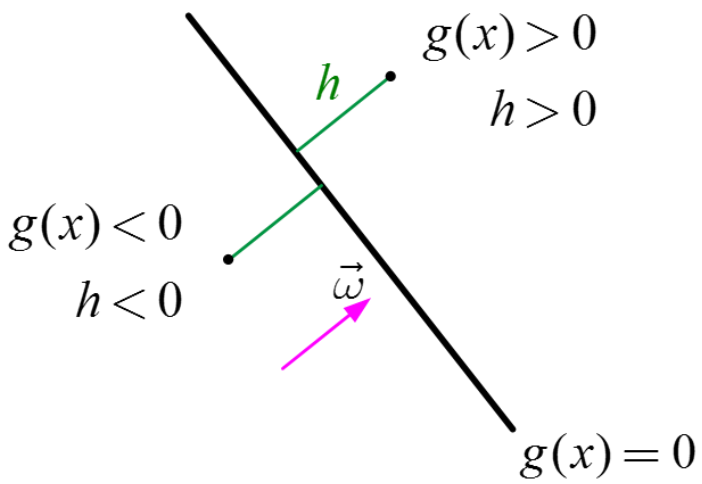
Sign

$$\therefore\ h = \frac{g(x)}{\|\omega\|} \implies \text{orthogonal signed distance from the line}$$

- Sign with respect to a line

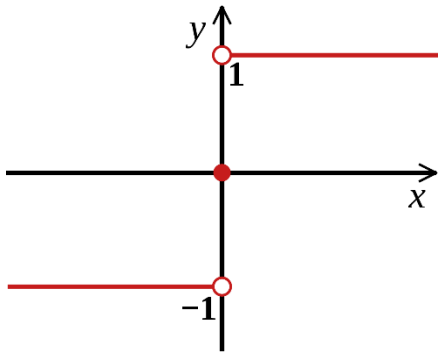
$$\omega = \begin{bmatrix} \omega_1 \\ \omega_2 \end{bmatrix}, \quad x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \implies g(x) = \omega_1 x_1 + \omega_2 x_2 + \omega_0 = \omega^T x + \omega_0$$

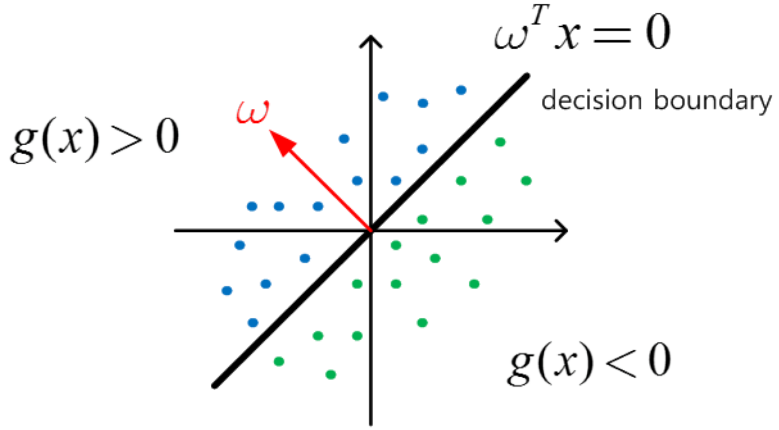
$$\omega = \begin{bmatrix} \omega_0 \\ \omega_1 \\ \omega_2 \end{bmatrix}, \quad x = \begin{bmatrix} 1 \\ x_1 \\ x_2 \end{bmatrix} \implies g(x) = \omega_0 + \omega_1 x_1 + \omega_2 x_2 = \omega^T x$$



How to Find ω

- All data in class 1 ($y = 1$)
  - $g(x) > 0$
- All data in class 0 ($y = -1$)
  - $g(x) < 0$





Perceptron Algorithm

- The perceptron implements

$$h(x) = \text{sign}\left(\omega^T x\right)$$

- Given the training set

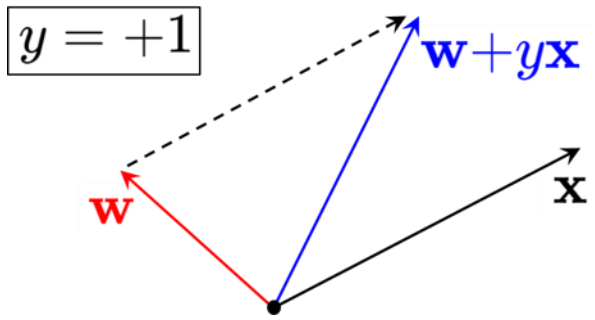
$$(x_1, y_1), (x_2, y_2), \cdots, (x_N, y_N) \quad \text{where } y_i \in \{-1, 1\}$$

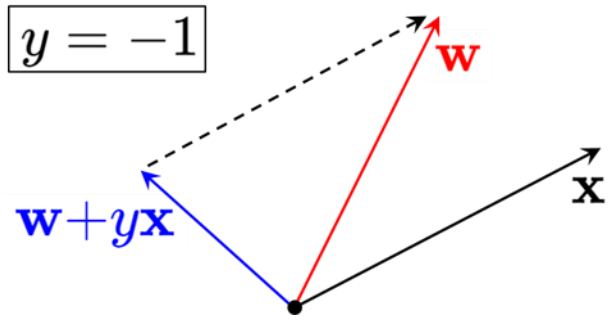
1) pick a misclassified point

$$\text{sign}\left(\omega^T x_n\right) \neq y_n$$

2) and update the weight vector

$$\omega \leftarrow \omega + y_n x_n$$





Iterations of Perceptron

1. Randomly assign ω

2. One iteration of the PLA (perceptron learning algorithm)

$$\omega \leftarrow \omega + yx$$

   where (x, y) is a misclassified training point

3. At iteration $t = 1,2,3, \cdots$, pick a misclassified point from

$$(x_1, y_1), (x_2, y_2), \cdots, (x_N, y_N)$$

4. And run a PLA iteration on it

5. That's it!

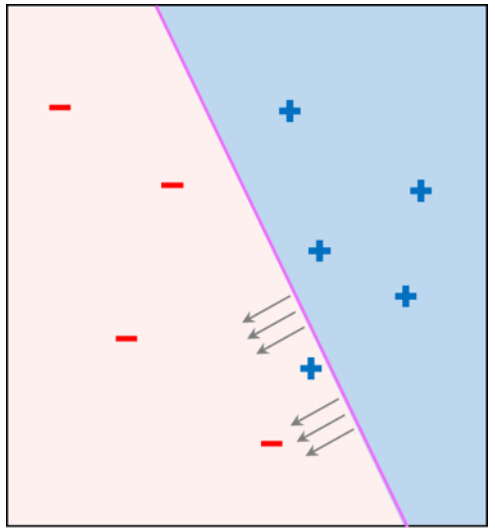
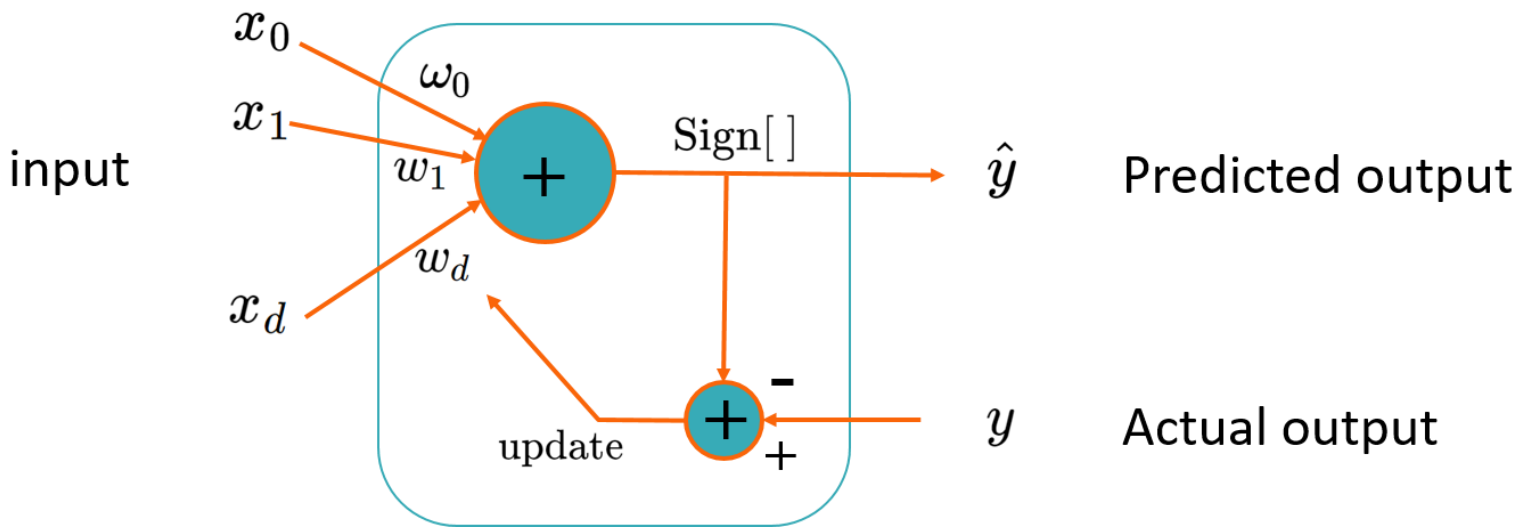


Diagram of Perceptron



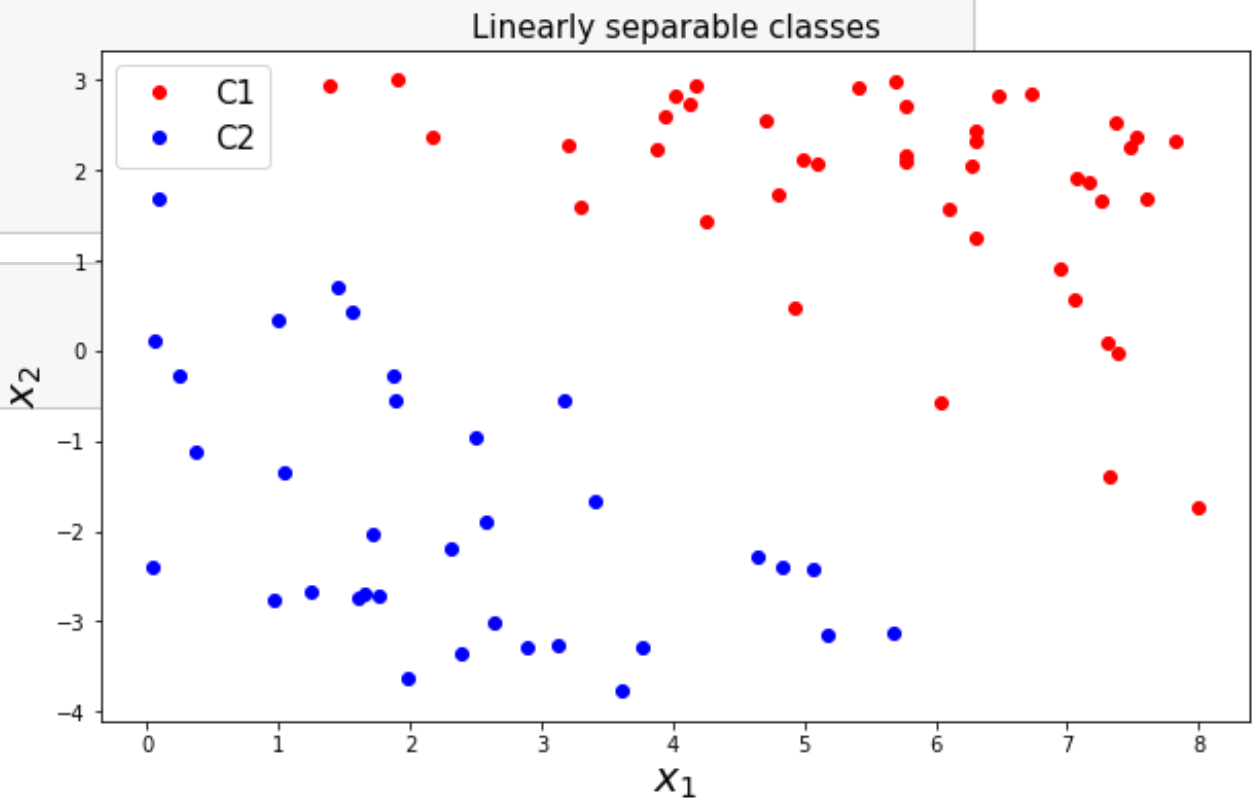
- Perceptron will be shown to be a basic unit for neural networks and deep learning later

Perceptron Algorithm in Python

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

```
#training data gerneration
m = 100
x1 = 8*np.random.rand(m, 1)
x2 = 7*np.random.rand(m, 1) - 4

g = 0.8*x1 + x2 - 3
```

```
C1 = np.where(g >= 1)
C2 = np.where(g < -1)
print(C1)
```



Perceptron Algorithm in Python

- Unknown parameters ω

$$g(x) = \omega_0 + \omega^T x = \omega_0 + \omega_1 x_1 + \omega_2 x_2 = 0$$

$$\omega = \begin{bmatrix} \omega_0 \\ \omega_1 \\ \omega_2 \end{bmatrix} \qquad x = \begin{bmatrix} \left(x^{(1)}\right)^T \\ \left(x^{(2)}\right)^T \\ \left(x^{(3)}\right)^T \\ \vdots \\ \left(x^{(m)}\right)^T \end{bmatrix} = \begin{bmatrix} 1 & x_1^{(1)} & x_2^{(1)} \\ 1 & x_1^{(2)} & x_2^{(2)} \\ 1 & x_1^{(3)} & x_2^{(3)} \\ \vdots & \vdots & \vdots \\ 1 & x_1^{(m)} & x_2^{(m)} \end{bmatrix} \qquad y = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ y^{(3)} \\ \vdots \\ y^{(m)} \end{bmatrix}$$

```
X1 = np.hstack([np.ones([C1.shape[0],1]), x1[C1], x2[C1]])
X2 = np.hstack([np.ones([C2.shape[0],1]), x1[C2], x2[C2]])
X = np.vstack([X1, X2])

y = np.vstack([np.ones([C1.shape[0],1]), -np.ones([C2.shape[0],1])])

X = np.asmatrix(X)
y = np.asmatrix(y)
```

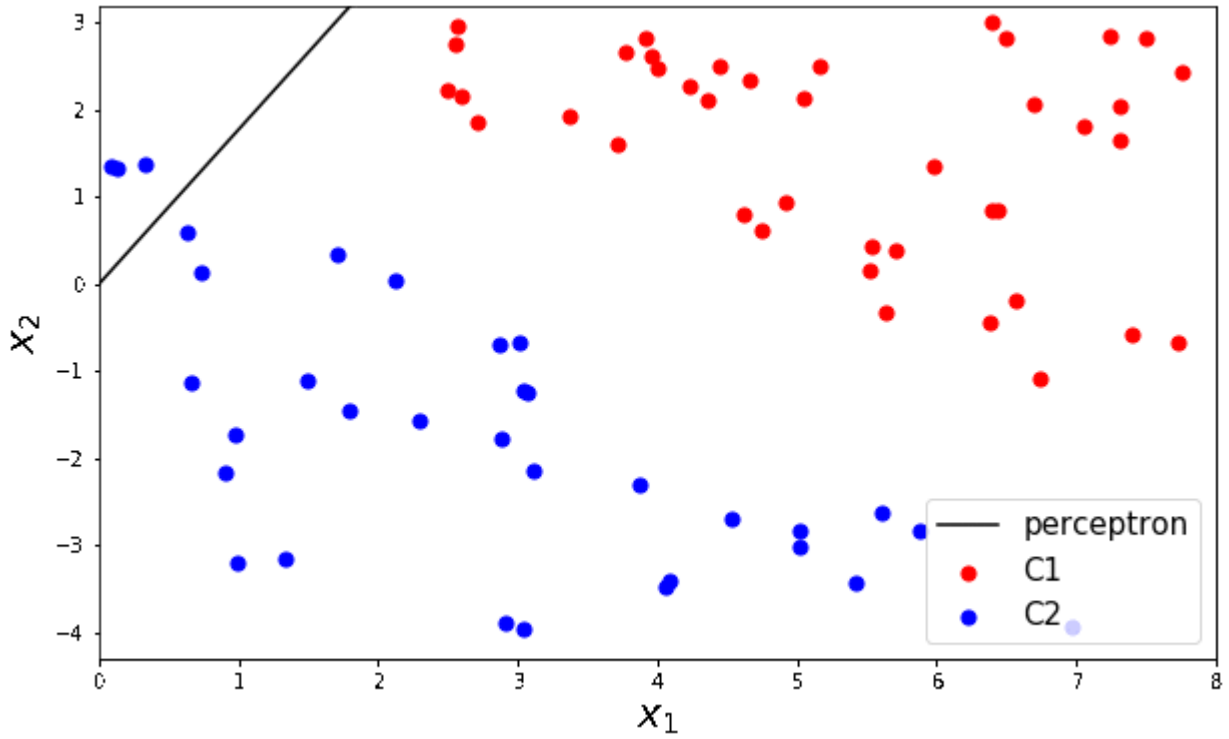
Perceptron Algorithm in Python

$$\omega = \begin{bmatrix} \omega_0 \\ \omega_1 \\ \omega_2 \end{bmatrix}$$

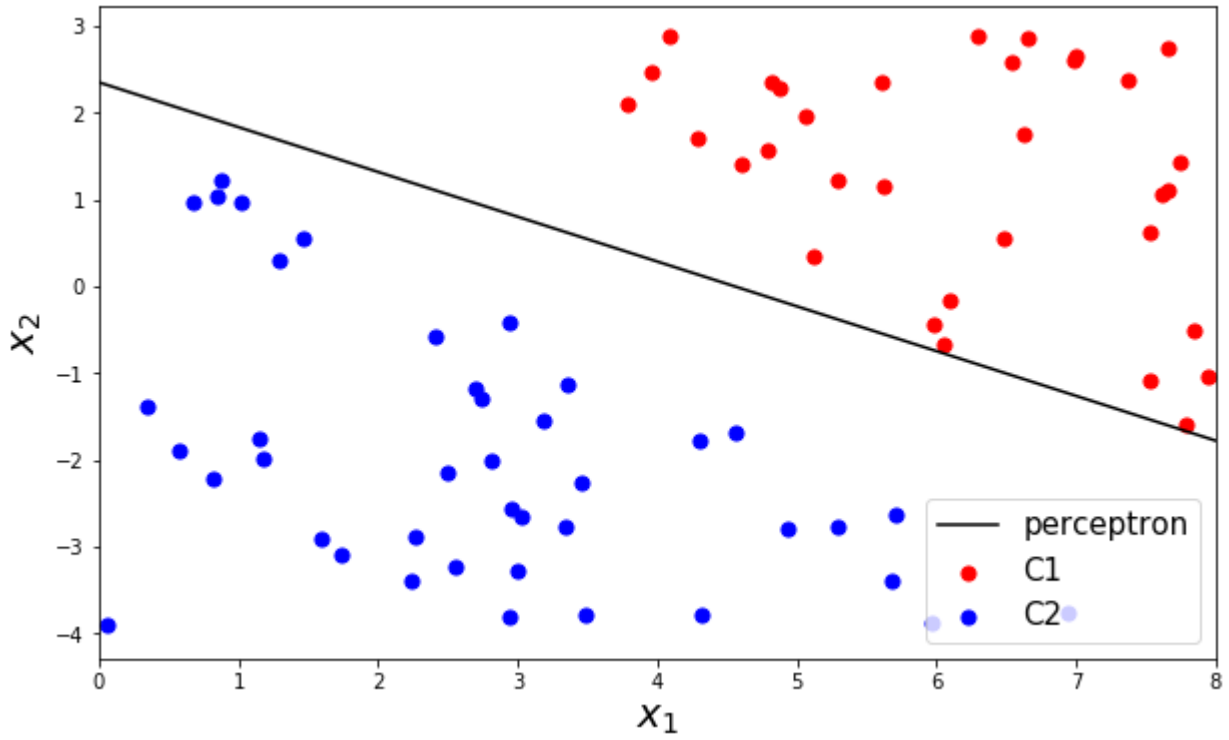
$\omega \leftarrow \omega + yx$ where (x, y) is a misclassified training point

```
w = np.ones([3,1])
w = np.asmatrix(w)

n_iter = y.shape[0]
for k in range(n_iter):
    for i in range(n_iter):
        if y[i,0] != np.sign(X[i,:]*w)[0,0]:
            w += y[i,0]*X[i,:].T

print(w)
```

Perceptron Algorithm in Python



Perceptron Algorithm in Python



Scikit-Learn for Perceptron

```
X1 = np.hstack([x1[C1], x2[C1]])
X2 = np.hstack([x1[C2], x2[C2]])
X = np.vstack([X1, X2])

y = np.vstack([np.ones([C1.shape[0],1]), -np.ones([C2.shape[0],1])])
```

```
from sklearn import linear_model

clf = linear_model.Perceptron(tol=1e-3)
clf.fit(X, np.ravel(y))
```

```
clf.predict([[3, -2]])
```

```
array([-1.])
```

$$x = \begin{bmatrix} \left(x^{(1)}\right)^T \\ \left(x^{(2)}\right)^T \\ \left(x^{(3)}\right)^T \\ \vdots \\ \left(x^{(m)}\right)^T \end{bmatrix} = \begin{bmatrix} x_1^{(1)} & x_2^{(1)} \\ x_1^{(2)} & x_2^{(2)} \\ x_1^{(3)} & x_2^{(3)} \\ \vdots & \vdots \\ x_1^{(m)} & x_2^{(m)} \end{bmatrix}$$

$$y = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ y^{(3)} \\ \vdots \\ y^{(m)} \end{bmatrix}$$

The Best Hyperplane Separator?

- Perceptron finds one of the many possible hyperplanes separating the data if one exists
- Of the many possible choices, which one is the best?

- Utilize distance information from all data samples
  - We will see this formally when we discuss the logistic regression

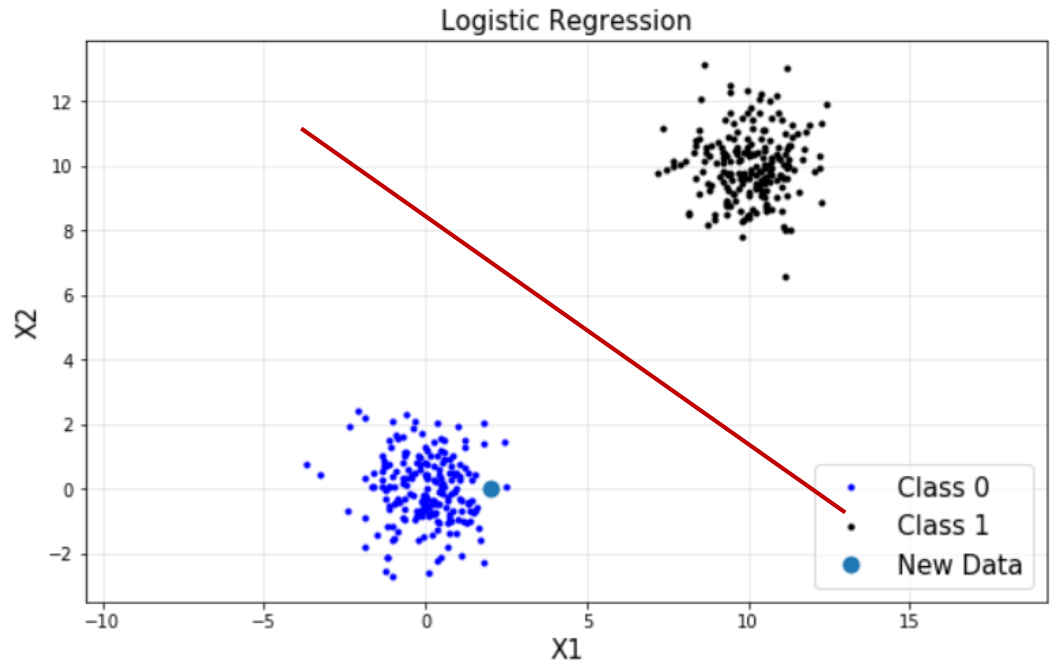
Classification: Logistic Regression

Classification: Logistic Regression

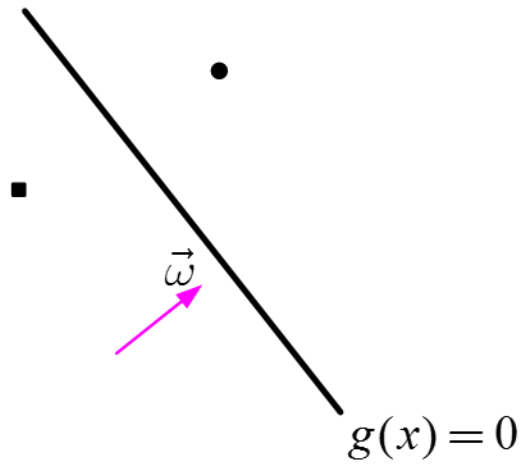
- Perceptron: make use of sign of data
- Logistic regression: make use of distance of data

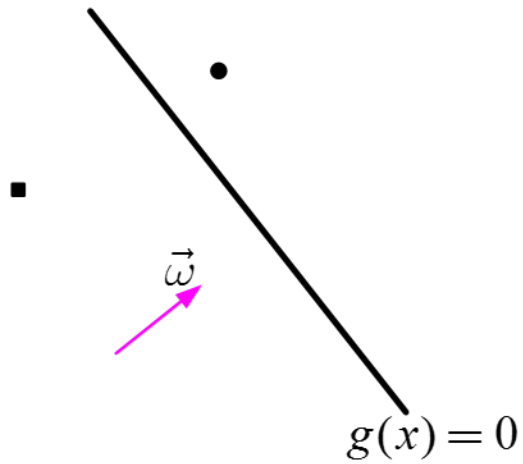
- Logistic regression is a classification algorithm
  - don't be confused from its name

- To find a classification boundary

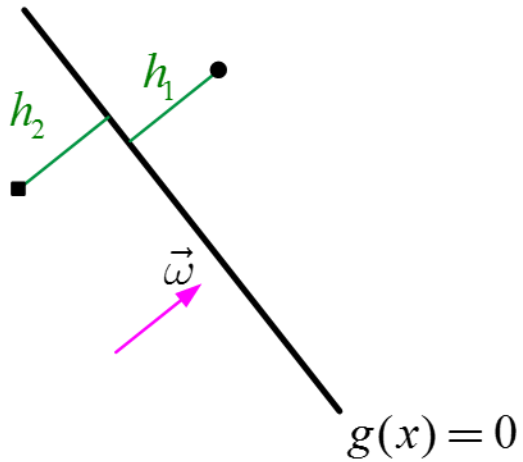


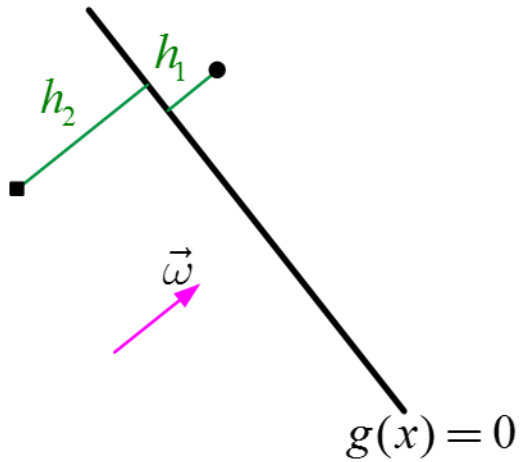
Using Distances





Using Distances





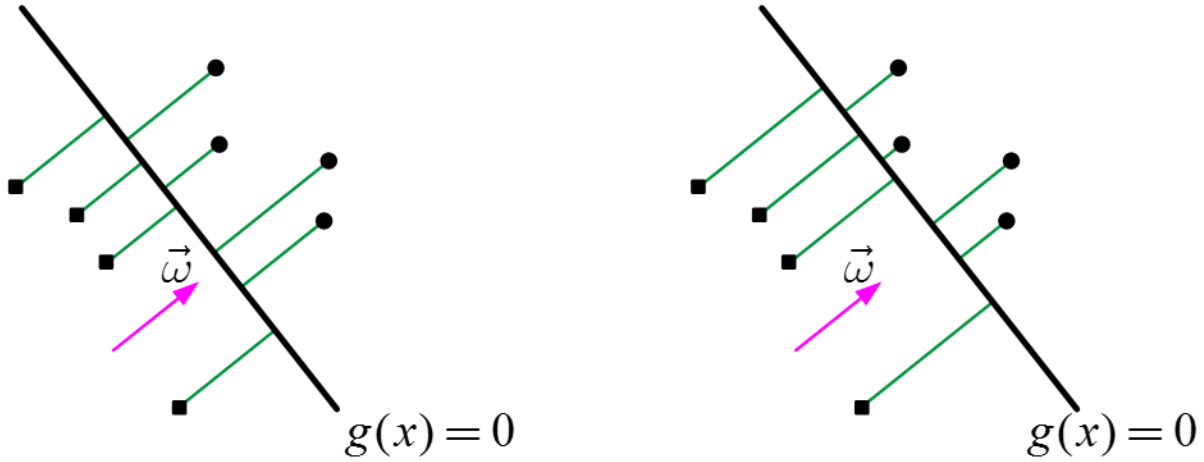
$$|h_1| + |h_2| \qquad |h_1| + |h_2|$$

$$|h_1| \cdot |h_2| \qquad |h_1| \cdot |h_2|$$

$$\frac{|h_1| + |h_2|}{2} \geq \sqrt{|h_1| \cdot |h_2|} \qquad \text{equal iff} \quad |h_1| = |h_2|$$

Using all Distances

- basic idea: to find the decision boundary (hyperplane) of $g(x) = \omega^T x = 0$ such that maximizes $\prod_i |h_i| \rightarrow$ optimization

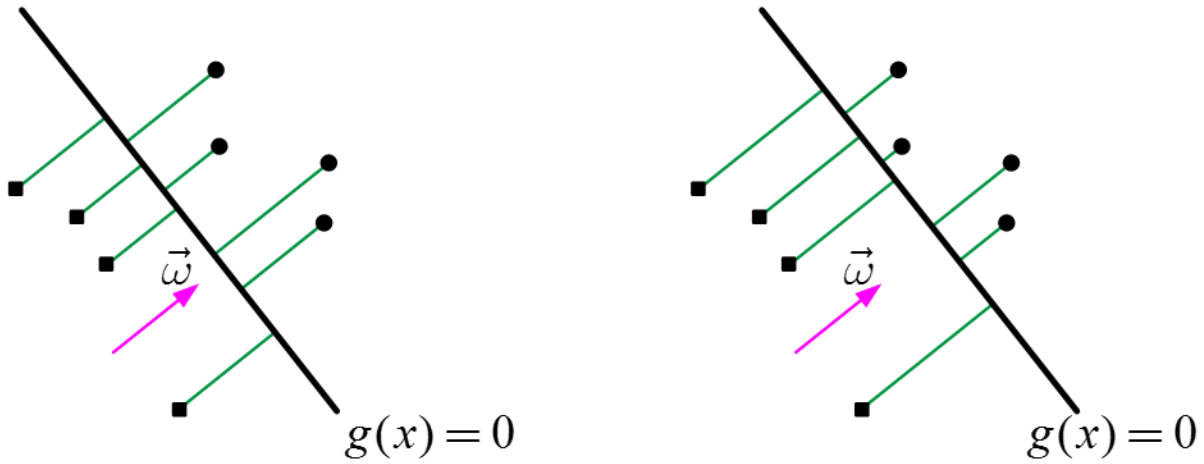


  - Inequality of arithmetic and geometric means

$$\frac{x_1 + x_2 + \cdots + x_m}{m} \geq \sqrt[m]{x_1 \cdot x_2 \ldots x_m}$$

and that equality holds if and only if $x_1 = x_2 = \cdots = x_m$

Using all Distances

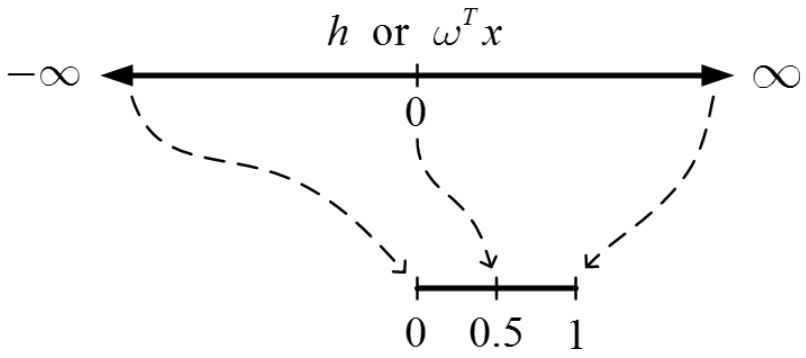


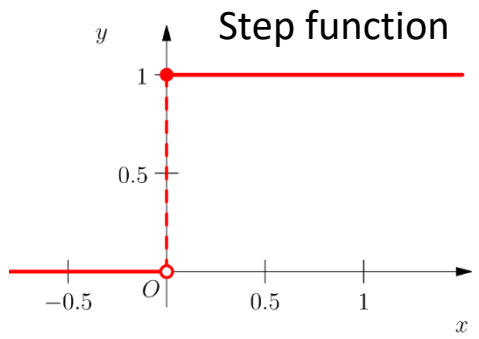
- Roughly speaking, this optimization of $\max \prod_i |h_i|$ tends to position a hyperplane in the middle of two classes

$$h = \frac{g(x)}{\|\omega\|} = \frac{\omega^T x}{\|\omega\|} \sim \omega^T x$$

Sigmoid Function

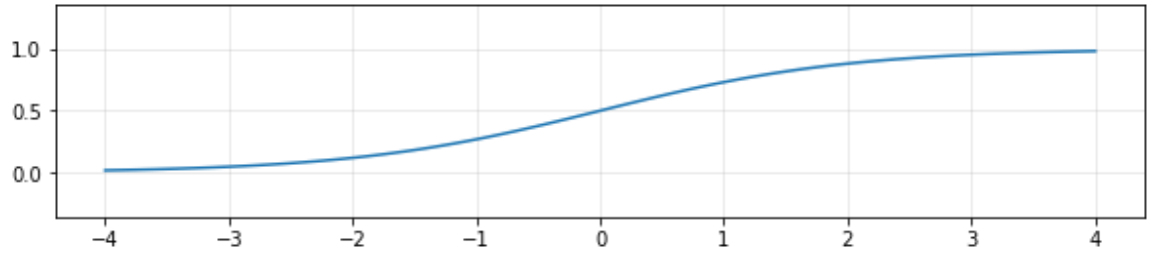
- We link or squeeze $(-\infty, +\infty)$ to $(0, 1)$ for several reasons:





- $\sigma(z)$ is the sigmoid function, or the logistic function
  - Logistic function always generates a value between 0 and 1
  - Crosses 0.5 at the origin, then flattens out

$$\sigma(z) = \frac{1}{1 + e^{-z}} \implies \sigma(\omega^T x) = \frac{1}{1 + e^{-\omega^T x}}$$



Sigmoid Function

- Benefit of mapping via the logistic function
  - Monotonic: same or similar optimization solution
  - Continuous and differentiable: good for gradient descent optimization
  - Probability or confidence: can be considered as probability

$$P\left(y=+1 \mid x, \omega\right)=\frac{1}{1+e^{-\omega^{T} x}} \quad \in \quad [0,1]$$

  - Probability that the label is $+1$

$$P\left(y=+1 \mid x\,; \omega\right)$$

  - Probability that the label is 0

$$P\left(y=0 \mid x\,; \omega\right)=1-P\left(y=+1 \mid x\,; \omega\right)$$

Goal: we need to fit ω to our data

- For a single data point (x, y) with parameters ω

$$P\left(y = +1 \mid x\,; \omega\right) = h_\omega(x) = \sigma\left(\omega^T x\right)$$
$$P\left(y = 0 \mid x\,; \omega\right) = 1 - h_\omega(x) = 1 - \sigma\left(\omega^T x\right)$$

- It can be compactly written as

$$P\left(y \mid x\,; \omega\right) = \left(h_\omega(x)\right)^y \left(1 - h_\omega(x)\right)^{1-y}$$

- For m training data points, the likelihood function of the parameters:

$$\begin{aligned}
\mathscr{L}(\omega) &= P\left(y^{(1)}, \cdots, y^{(m)} \mid x^{(1)}, \cdots, x^{(m)}\,; \omega\right) \\
&= \prod_{i=1}^{m} P\left(y^{(i)} \mid x^{(i)}\,; \omega\right) \\
&= \prod_{i=1}^{m} \left(h_\omega\left(x^{(i)}\right)\right)^{y^{(i)}} \left(1 - h_\omega\left(x^{(i)}\right)\right)^{1-y^{(i)}} \qquad \left(\sim \prod_i |h_i|\right)
\end{aligned}$$

Goal: we need to fit ω to our data

$$\begin{aligned}\mathscr{L}(\omega) &= P\left(y^{(1)}, \cdots, y^{(m)} \mid x^{(1)}, \cdots, x^{(m)}; \omega\right) \\ &= \prod_{i=1}^{m} P\left(y^{(i)} \mid x^{(i)}; \omega\right) \\ &= \prod_{i=1}^{m} \left(h_\omega\left(x^{(i)}\right)\right)^{y^{(i)}} \left(1 - h_\omega\left(x^{(i)}\right)\right)^{1-y^{(i)}} \qquad \left(\sim \prod_i |h_i|\right)\end{aligned}$$

- It would be easier to work on the log likelihood.

$$\ell(\omega) = \log \mathscr{L}(\omega) = \sum_{i=1}^{m} y^{(i)} \log h_\omega\left(x^{(i)}\right) + \left(1 - y^{(i)}\right) \log\left(1 - h_\omega\left(x^{(i)}\right)\right)$$

cross-entropy later

- The logistic regression problem can be solved as a (convex) optimization problem:

$$\hat{\omega} = \arg\max_{\omega} \ell(\omega)$$

- Again, it is an optimization problem

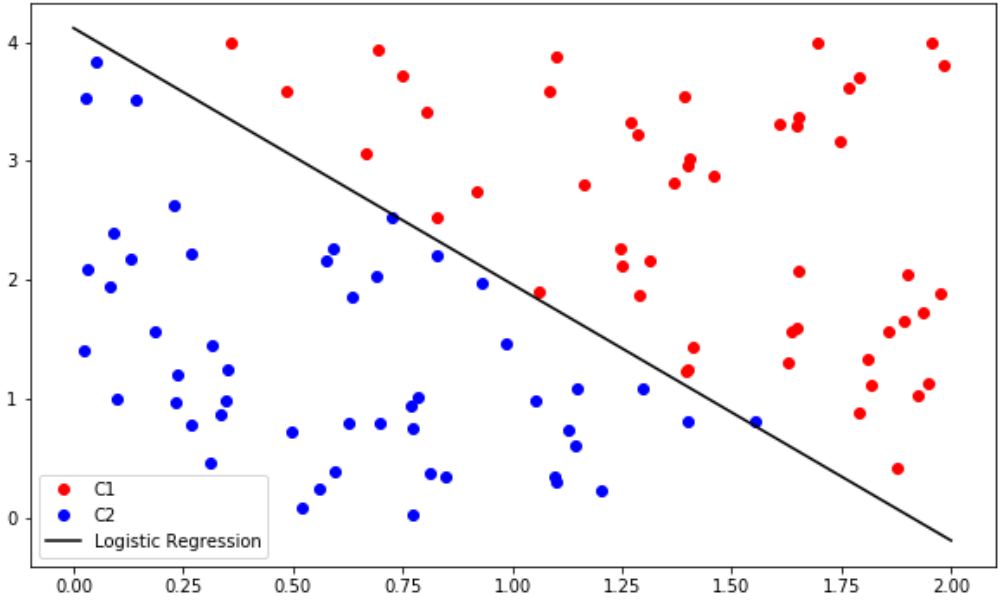
Scikit-Learn for Logistic Regression

```
from sklearn import linear_model

clf = linear_model.LogisticRegression(solver='lbfgs')
clf.fit(X,np.ravel(y))
```

```
w1 = clf.coef_[0,0]
w2 = clf.coef_[0,1]
w0 = clf.intercept_[0]

xp = np.linspace(0,2,100).reshape(-1,1)
yp = - w1/w2*xp - w0/w2

plt.figure(figsize = (10,6))
plt.plot(X[C1,0], X[C1,1], 'ro', label='C1')
plt.plot(X[C2,0], X[C2,1], 'bo', label='C2')
plt.plot(xp, yp, 'k', label='Logistic Regression')
plt.legend()
plt.show()
```

$$\omega = \begin{bmatrix} \omega_1 \\ \omega_2 \end{bmatrix}, \qquad \omega_0, \qquad x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

$$X = \begin{bmatrix} \left(x^{(1)}\right)^T \\ \left(x^{(2)}\right)^T \\ \left(x^{(3)}\right)^T \\ \vdots \end{bmatrix} = \begin{bmatrix} x_1^{(1)} & x_2^{(1)} \\ x_1^{(2)} & x_2^{(2)} \\ x_1^{(3)} & x_2^{(3)} \\ \vdots & \vdots \end{bmatrix} \qquad y = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ y^{(3)} \\ \vdots \end{bmatrix}$$



Multiclass Classification: Softmax

- Generalization to more than 2 classes is straightforward
  - one vs. all (one vs. rest)
  - one vs. one

- Using the softmax function instead of the logistic function
  - See them as probability

$$P\left(y = k \mid x, \omega\right) = \frac{\exp\left(\omega_k^T x\right)}{\sum_k \exp\left(\omega_k^T x\right)} \in [0, 1]$$

  - We maintain a separator weight vector ω_k for each class k

  - Note: sigmoid function

$$P\left(y = +1 \mid x, \omega\right) = \frac{1}{1 + e^{-\omega^T x}} \in [0, 1]$$

Summary

- From parameter estimation of machine learning to optimization problems

Machine learning	Optimization
	Loss (or objective functions)
Regression	$\min_{\theta_1,\theta_2} \sum_{i=1}^{m} (\hat{y}_i - y_i)^2$
Classification	$\ell(\omega) = \log \mathscr{L} = \log P(y \mid x, \omega) = \log \prod_{n=1}^{m} P(y_n \mid x_n, \omega)$ $= \sum_{n=1}^{m} \log P(y_n \mid x_n, \omega)$

Machine Learning with TensorFlow

Prof. Seungchul Lee

Industrial AI Lab.

Training Neural Networks: Deep Learning Libraries

- TensorFlow
  - Platform: Linux, Mac OS, Windows
  - Written in: C++, Python
  - Interface: Python, C/C++, Java, Go, R

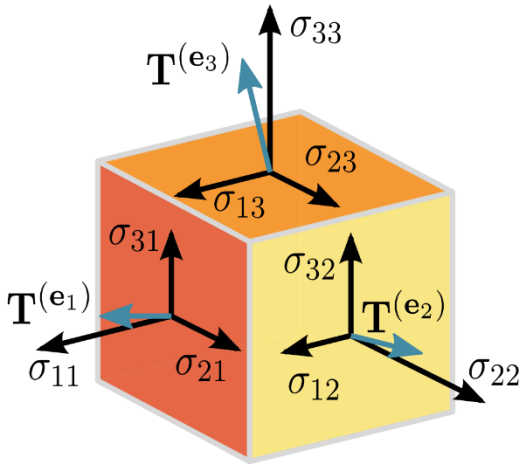
- Keras

- PyTorch

TensorFlow

- Developed by Google and it is one of the most popular Machine Learning libraries on GitHub.
- It is a framework to perform computation very efficiently, and it can tap into the GPU in order to speed it up even further.
- TensorFlow is one of the widely used libraries for implementing machine learning and deep learning involving large number of mathematical operations.

- Tensor and Flow
  - TensorFlow gets its name from tensors, which are arrays of arbitrary dimensionality.
  - The "flow" part of the name refers to computation flowing through a graph.



Computational Graph

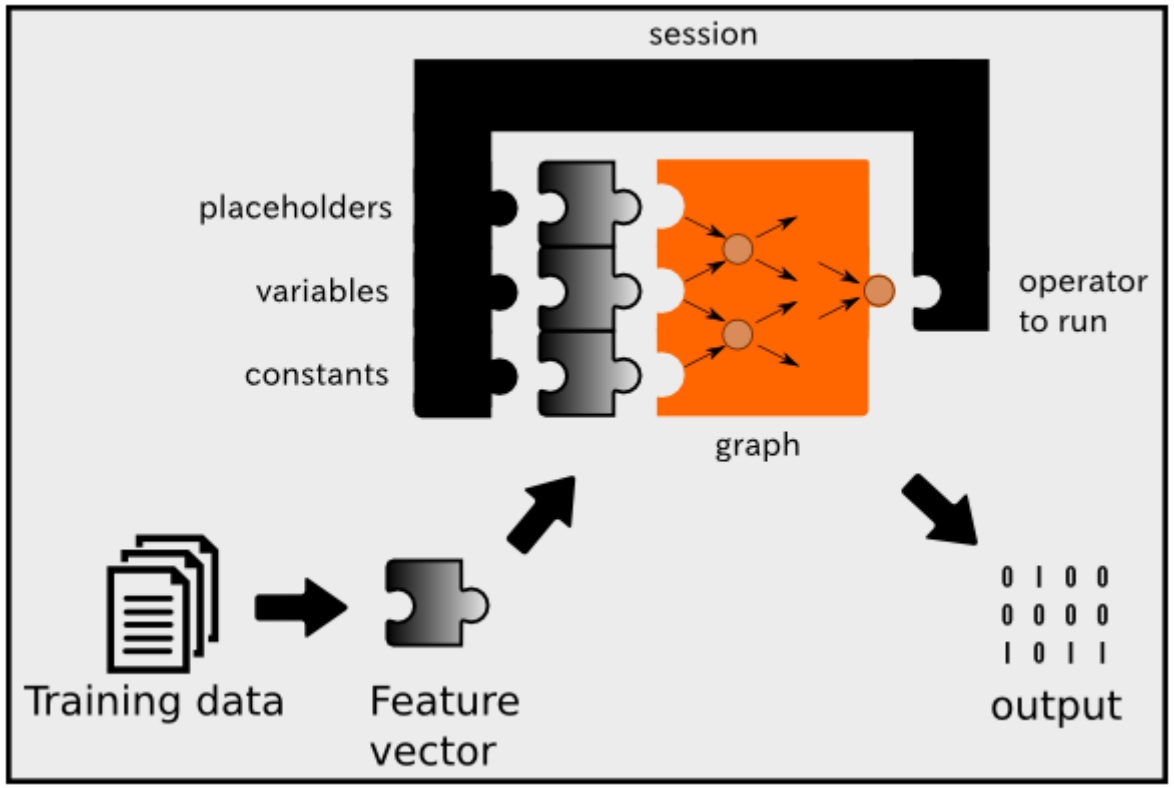
- TensorFlow is an open-source software library for deep learning
  - tf.constant
  - tf.Variable
  - tf.placeholder

```
import tensorflow as tf

a = tf.constant([1,2,3])
b = tf.constant(4, shape=[1,3])

A = a + b
B = a*b

print(A)
```

```
Tensor("add_1:0", shape=(1, 3), dtype=int32)
```

TensorFlow: Session

- To run any of the three defined operations, we need to create a session for that graph. The session will also allocate memory to store the current value of the variable.



TensorFlow

```
import tensorflow as tf

a = tf.constant([1,2,3])
b = tf.constant(4, shape=[1,3])

A = a + b
B = a*b

print(A)
```

```
sess = tf.Session()
sess.run(A)
```

```
array([[5, 6, 7]])
```

```
sess.run(B)
```

```
array([[ 4,  8, 12]])
```

```
a = tf.constant([1,2,3])
b = tf.constant([4,5,6])

result = tf.multiply(a, b)

with tf.Session() as sess:
    output = sess.run(result)
    print(output)
```

← Interactive Session:
run the result and close the Session automatically

TensorFlow: tf.Variable

- tf.Variable is regarded as the decision variable in optimization.
- We should initialize variables.

```
x1 = tf.Variable([1, 1], dtype = tf.float32)
x2 = tf.Variable([2, 2], dtype = tf.float32)
y = x1 + x2

print(y)
```

```
<tf.Tensor 'add_8:0' shape=(2,) dtype=float32>
```

```
sess = tf.Session()

init = tf.global_variables_initializer()
sess.run(init)

sess.run(y)
```

```
array([ 3.,  3.], dtype=float32)
```

TensorFlow: Placeholder

- The value of tf.placeholder must be fed using the feed_dict optional argument to Session.run()

```
sess = tf.Session()
x = tf.placeholder(tf.float32, shape = [2,2])

sess.run(x, feed_dict = {x : [[1,2],[3,4]]})
```

```
array([[ 1.,  2.],
       [ 3.,  4.]], dtype=float32)
```

```
a = tf.placeholder(tf.float32, shape = [2])
b = tf.placeholder(tf.float32, shape = [2])

sum = a + b

sess.run(sum, feed_dict = {a : [1,2], b : [3,4]})
```

```
array([ 4.,  6.], dtype=float32)
```

Tensor Manipulation: Adding

```
x1 = tf.constant(1, shape = [3])
x2 = tf.constant(2, shape = [3])
output = tf.add(x1, x2)

with tf.Session() as sess:
    result = sess.run(output)
    print(result)
```

```
[3 3 3]
```

```
x1 = tf.constant(1, shape = [2, 3])
x2 = tf.constant(2, shape = [2, 3])
output = tf.add(x1, x2)

with tf.Session() as sess:
    result = sess.run(output)
    print(result)
```

```
[[3 3 3]
 [3 3 3]]
```

Tensor Manipulation: Multiplying

```
x1 = tf.constant([[1, 2],
                  [3, 4]])
x2 = tf.constant([[2],[3]])
```

```
output1 = tf.matmul(x1, x2)

with tf.Session() as sess:
    result = sess.run(output1)
    print(result)
```

```
[[ 8]
 [18]]
```

```
output2 = x1*x2

with tf.Session() as sess:
    result = sess.run(output2)
    print(result)
```

```
[[ 2  4]
 [ 9 12]]
```

Tensor Manipulation: Reshape

```
x = [1, 2, 3, 4, 5, 6, 7, 8]
```

```
x_re = tf.reshape(x, [4,2])

sess = tf.Session()
sess.run(x_re)
```

```
array([[1, 2],
       [3, 4],
       [5, 6],
       [7, 8]])
```

```
x_re = tf.reshape(x, [2,-1])

sess = tf.Session()
sess.run(x_re)
```

```
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])
```

TensorFlow as an Optimization Solver

```
w = tf.Variable(0, dtype = tf.float32)
cost =  w*w - 8*w + 16

LR = 0.05
optm = tf.train.GradientDescentOptimizer(LR).minimize(cost)

init = tf.global_variables_initializer()

sess = tf.Session()
sess.run(init)

print(sess.run(w))
```

```
0.0
```

$$\min_{\omega} \ (\omega - 4)^2$$

```
# runs one step of gradient descent
sess.run(optm)
print(sess.run(w))

# runs two step of gradient descent
sess.run(optm)
print(sess.run(w))
```

```
0.4
0.76
```

```
for _ in range(100):
    sess.run(optm)

print(sess.run(w))
sess.close()
```

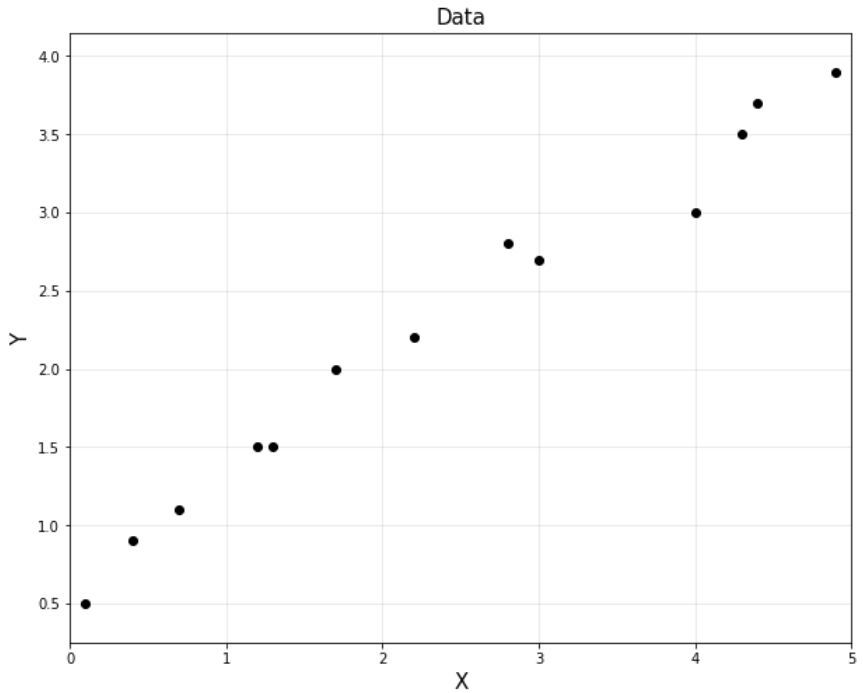
```
3.99991
```

Machine Learning with TensorFlow

Regression

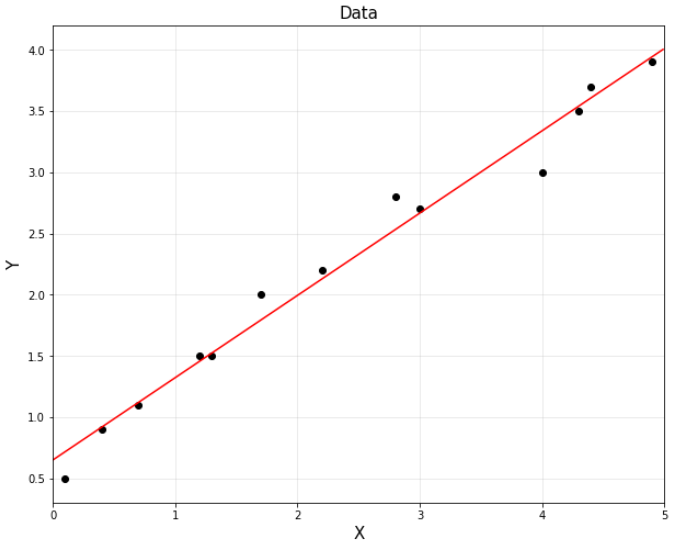
- Given (x_i, y_i) for $i = 1, \cdots, m$,
- Want to estimate

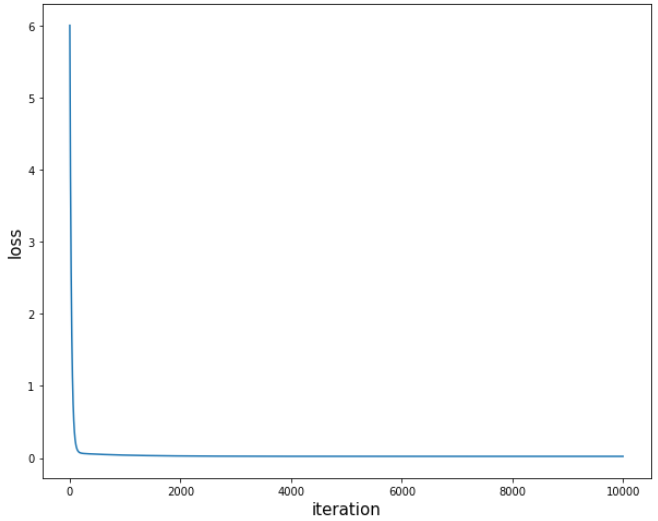
$$\hat{y}_i = \omega x_i + b \quad \text{such that} \quad \min_{\omega,b} \sum_{i=1}^{m} (\hat{y}_i - y_i)^2$$



Regression with TensorFlow

```
LR = 0.001
n_iter = 10000

x = tf.placeholder(tf.float32, [m, 1])
y = tf.placeholder(tf.float32, [m, 1])

w = tf.Variable([[0]], dtype = tf.float32)
b = tf.Variable([[0]], dtype = tf.float32)

#y_pred = tf.matmul(x, w) + b
y_pred = tf.add(tf.matmul(x, w), b)
loss = tf.square(y_pred - y)
loss = tf.reduce_mean(loss)

optm = tf.train.GradientDescentOptimizer(LR).minimize(loss)

sess = tf.Session()
sess.run(tf.global_variables_initializer())

loss_record = []
for epoch in range(n_iter):
    _, c = sess.run([optm, loss], feed_dict = {x: train_x, y: train_y})
    loss_record.append(c)

w_val = sess.run(w)
b_val = sess.run(b)

sess.close()
```

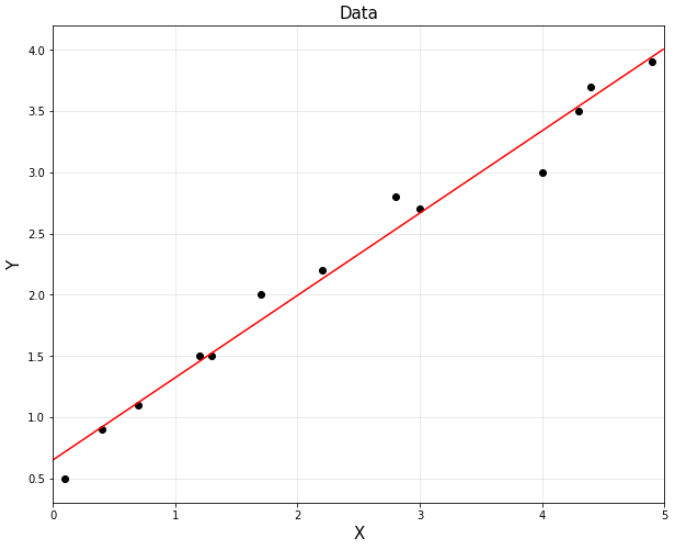


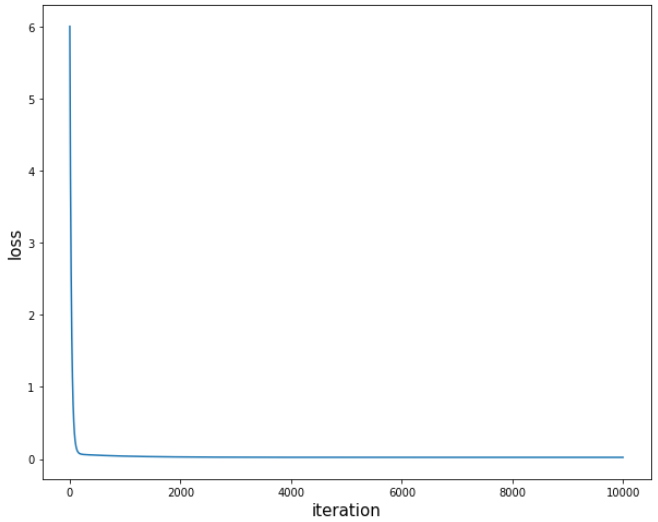


Regression with TensorFlow

- with tf.Session() as sess:

```
LR = 0.001
n_iter = 10000

x = tf.placeholder(tf.float32, [m, 1])
y = tf.placeholder(tf.float32, [m, 1])

w = tf.Variable([[0]], dtype = tf.float32)
b = tf.Variable([[0]], dtype = tf.float32)

#y_pred = tf.matmul(x, w) + b
y_pred = tf.add(tf.matmul(x, w), b)
loss = tf.square(y_pred - y)
loss = tf.reduce_mean(loss)

optm = tf.train.GradientDescentOptimizer(LR).minimize(loss)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    for epoch in range(n_iter):
        sess.run(optm, feed_dict = {x: train_x, y: train_y})
    w_val = sess.run(w)
    b_val = sess.run(b)
```

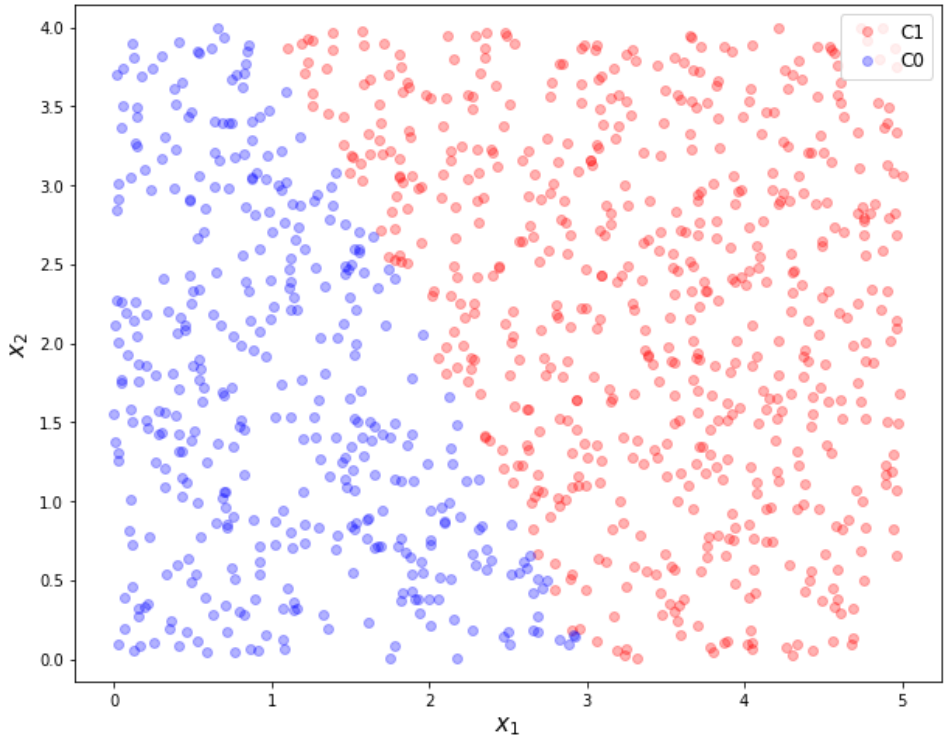




Logistic Regression

$$\omega = \begin{bmatrix} \omega_0 \\ \omega_1 \\ \omega_2 \end{bmatrix}, \qquad x = \begin{bmatrix} 1 \\ x_1 \\ x_2 \end{bmatrix}$$

$$X = \begin{bmatrix} \left(x^{(1)}\right)^T \\ \left(x^{(2)}\right)^T \\ \left(x^{(3)}\right)^T \\ \vdots \end{bmatrix} = \begin{bmatrix} 1 & x_1^{(1)} & x_2^{(1)} \\ 1 & x_1^{(2)} & x_2^{(2)} \\ 1 & x_1^{(3)} & x_2^{(3)} \\ \vdots & \vdots & \vdots \end{bmatrix}, \qquad y = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ y^{(3)} \\ \vdots \end{bmatrix}$$

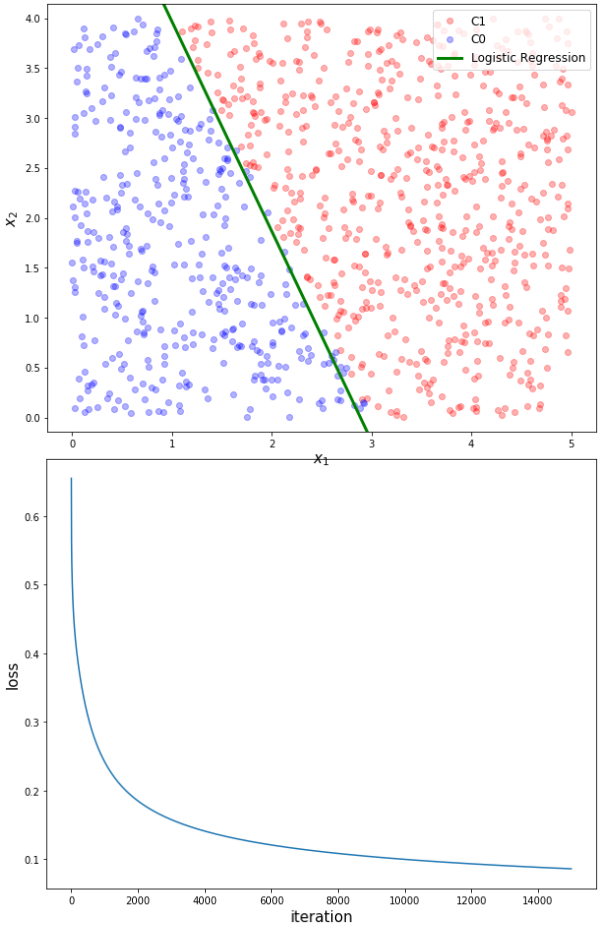


Logistic Regression with TensorFlow

$$\ell(\omega) = \log \mathscr{L}(\omega) = \sum_{i=1}^{m} y^{(i)} \log h_\omega \left(x^{(i)}\right) + \left(1 - y^{(i)}\right) \log\left(1 - h_\omega \left(x^{(i)}\right)\right)$$

$$\Rightarrow \frac{1}{m} \sum_{i=1}^{m} y^{(i)} \log h_\omega \left(x^{(i)}\right) + \left(1 - y^{(i)}\right) \log\left(1 - h_\omega \left(x^{(i)}\right)\right)$$

```
LR = 0.05
n_iter = 15000

X = tf.placeholder(tf.float32, [m, 3])
y = tf.placeholder(tf.float32, [m, 1])

w = tf.Variable([[0],[0],[0]], dtype = tf.float32)

y_pred = tf.sigmoid(tf.matmul(X,w))
loss = - y*tf.log(y_pred) - (1-y)*tf.log(1-y_pred)
loss = tf.reduce_mean(loss)

optm = tf.train.GradientDescentOptimizer(LR).minimize(loss)

loss_record = []
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    for epoch in range(n_iter):
        _, c = sess.run([optm, loss], feed_dict = {X: train_X, y: train_y})
        loss_record.append(c)

    w_hat = sess.run(w)
```



Logistic Regression with TensorFlow

- TensorFlow embedded functions
  - tf.nn.sigmoid_cross_entropy_with_logits for binary classification
  - tf.nn.softmax_cross_entropy_with_logits for multiclass classification

```
LR = 0.05
n_iter = 30000

X = tf.placeholder(tf.float32, [m, 3])
y = tf.placeholder(tf.float32, [m, 1])

w = tf.Variable(tf.random_normal([3,1]), dtype = tf.float32)

y_pred = tf.matmul(X,w)
loss = tf.nn.sigmoid_cross_entropy_with_logits(logits = y_pred, labels = y)
loss = tf.reduce_mean(loss)

optm = tf.train.GradientDescentOptimizer(LR).minimize(loss)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    for epoch in range(n_iter):
        sess.run(optm, feed_dict = {X: train_X, y: train_y})

    w_hat = sess.run(w)
```