



Machine Learning

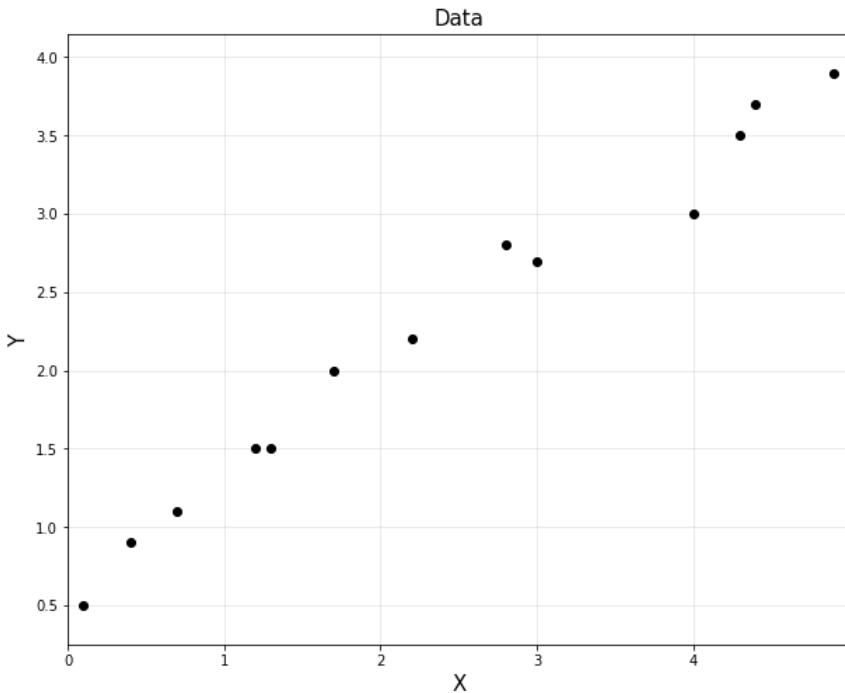
Industrial AI Lab.

Prof. Seungchul Lee

Regression

Assumption: Linear Model

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

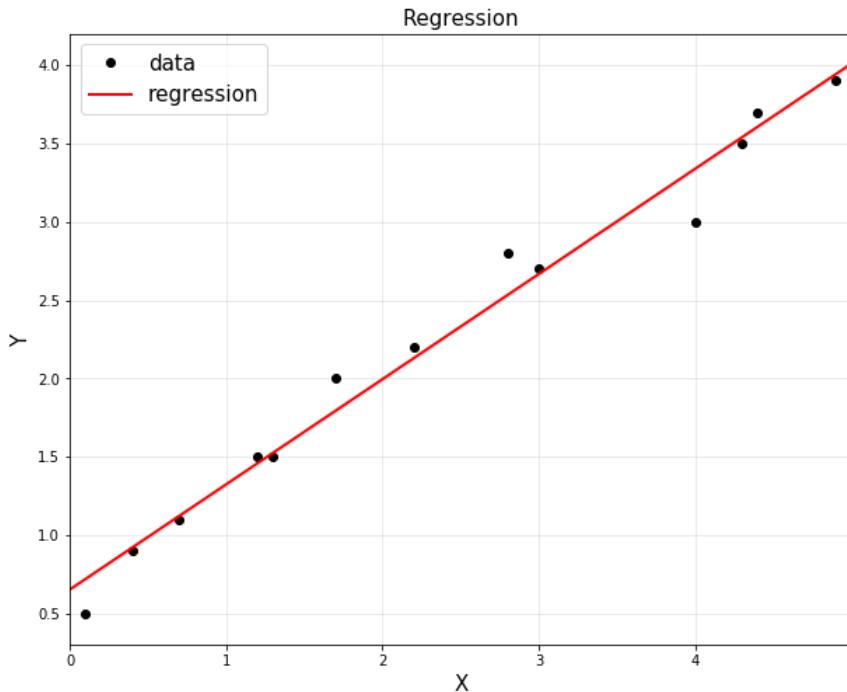


- In many cases, a linear model is used to predict y_i

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

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

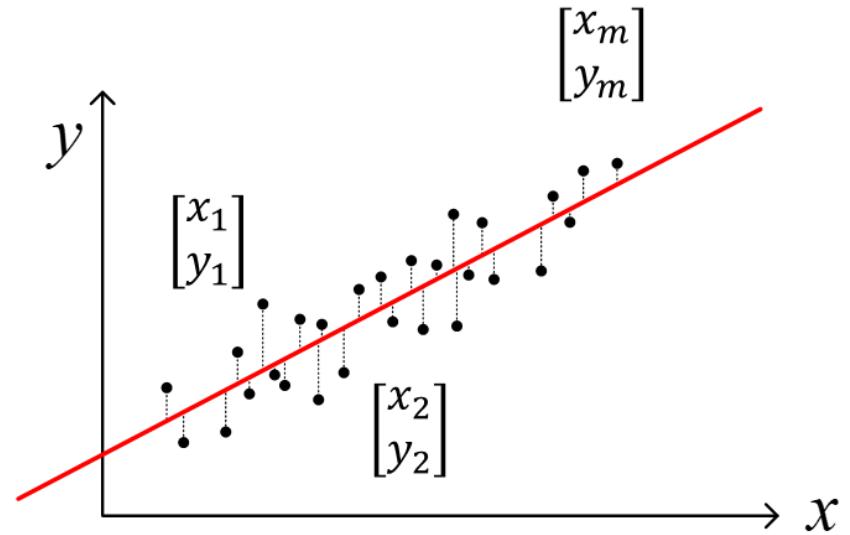
$$\hat{y}_i = \theta_1 x_i + \theta_2$$

Linear Regression

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

Given $\begin{cases} x_i : \text{inputs} \\ y_i : \text{outputs} \end{cases}$, Find θ_0 and θ_1

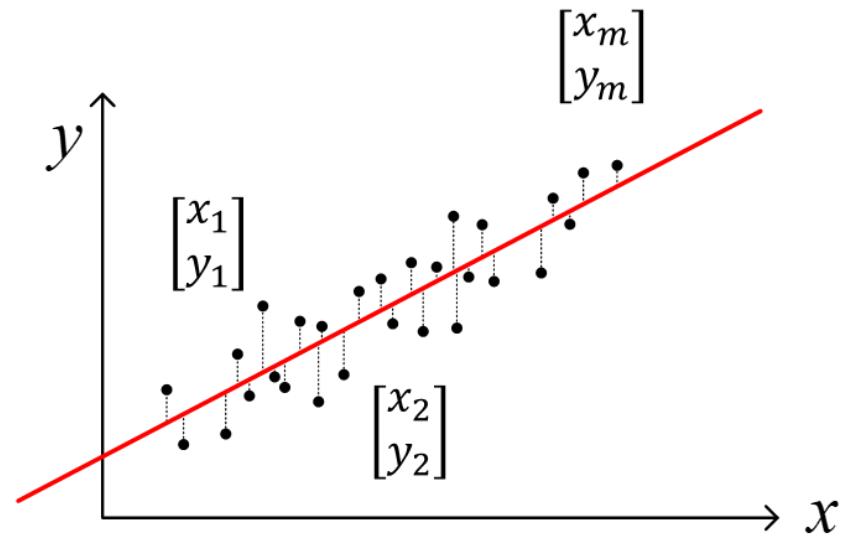
$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix}, \quad y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} \approx \hat{y}_i = \theta_0 + \theta_1 x_i$$



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

Linear Regression as Optimization

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



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

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

Re-cast Problem as Least Squares

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

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

$$= [1 \quad x_i] \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix}$$

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

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

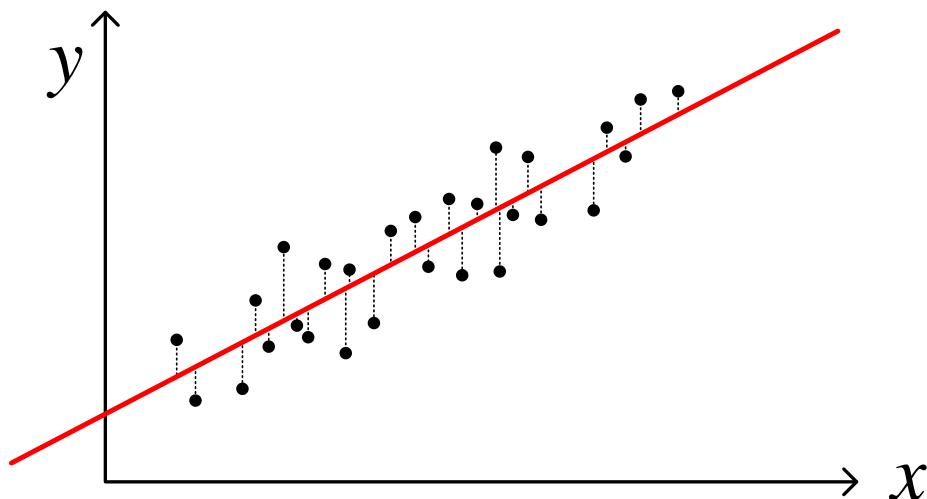
feature vector $\phi(x_i) = \begin{bmatrix} 1 \\ x_i \end{bmatrix}$

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

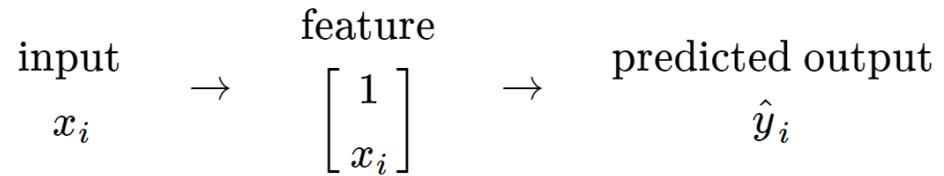
Optimization

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

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



Optimization: Note

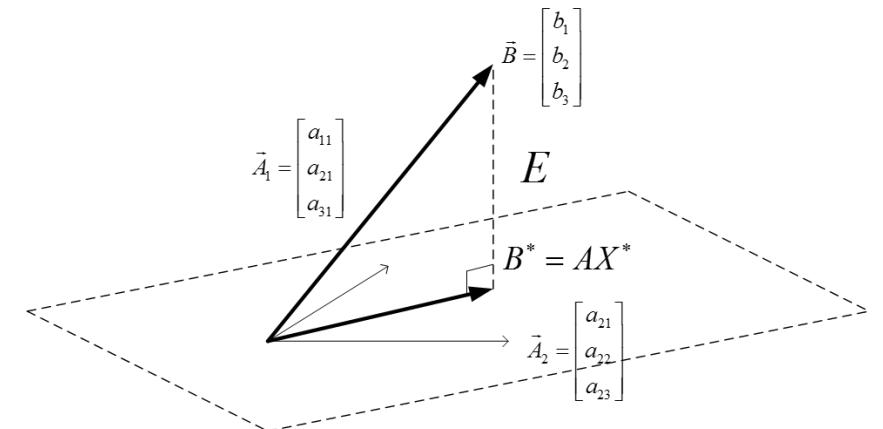


$$\begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_m \end{bmatrix} \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix}$$

over-determined or
projection

$\vec{A}_1 \quad \vec{A}_2 \quad \vec{x} \quad \vec{B}$

$$A(= \Phi) = [\vec{A}_1 \ \vec{A}_2]$$



the same principle in a higher dimension

Solve using Linear Algebra

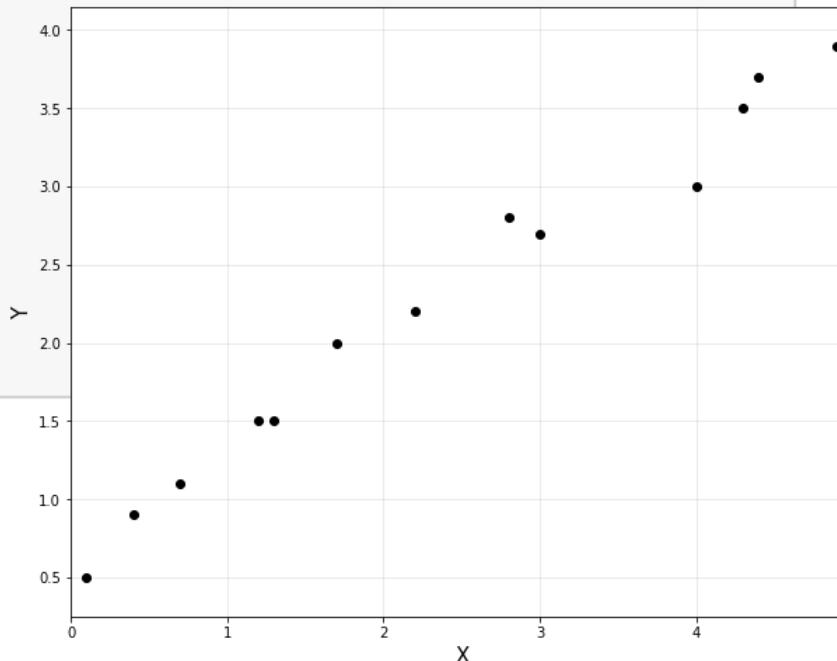
- known as *least square*

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

```
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, 4.9]).reshape(-1, 1)
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, 3.9]).reshape(-1, 1)

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()
```



Solving 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)
```

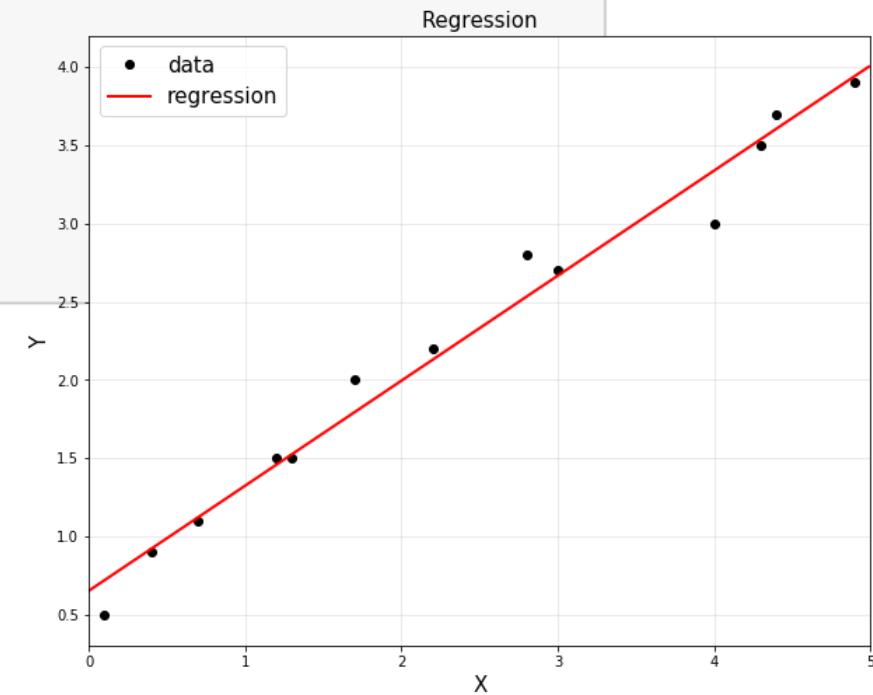
```
theta:
[[0.65306531]
 [0.67129519]]
```

Solving using Linear Algebra

```
# 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()
```



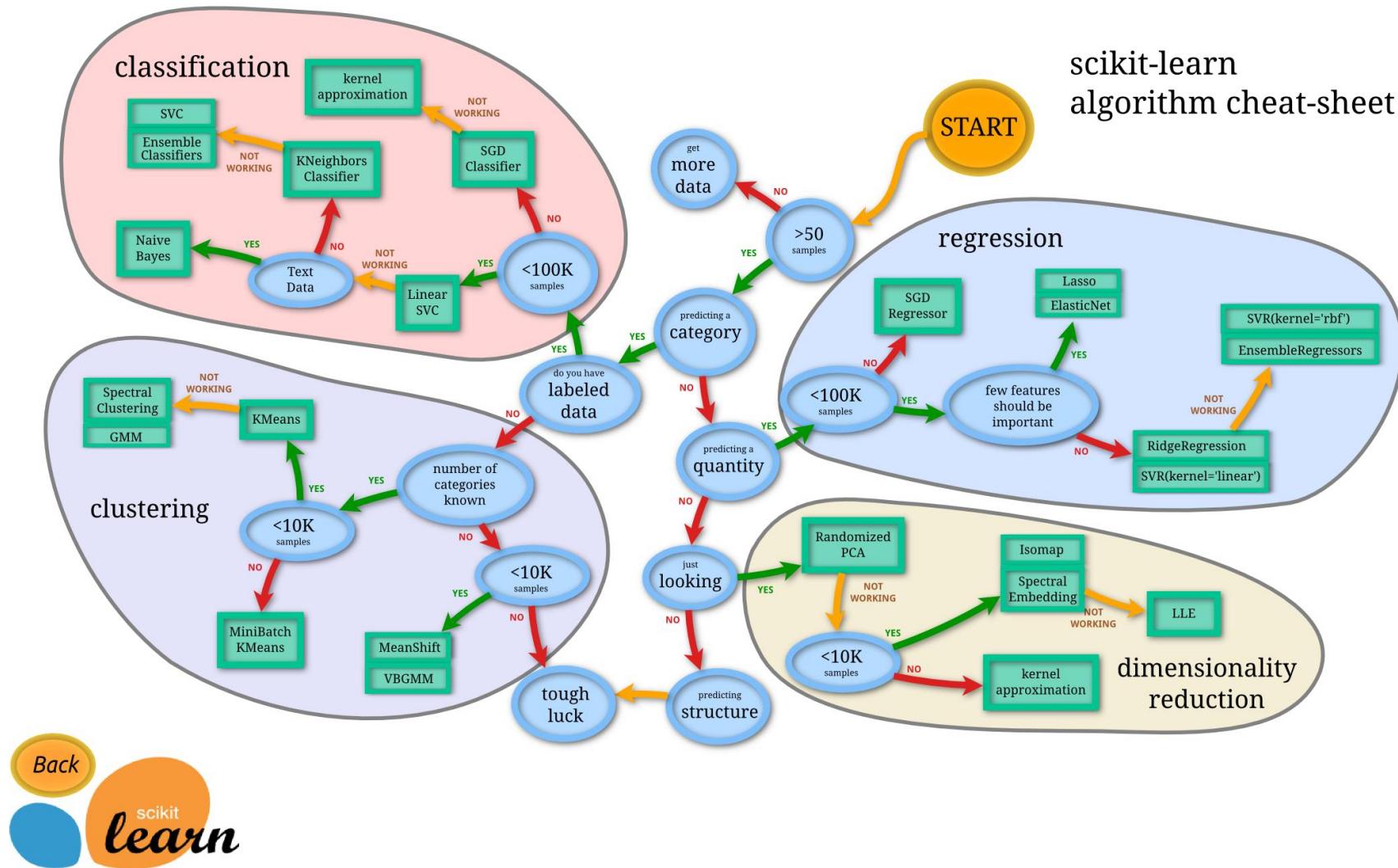
Scikit-Learn

- 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 algorithm cheat-sheet



Back
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]])
```

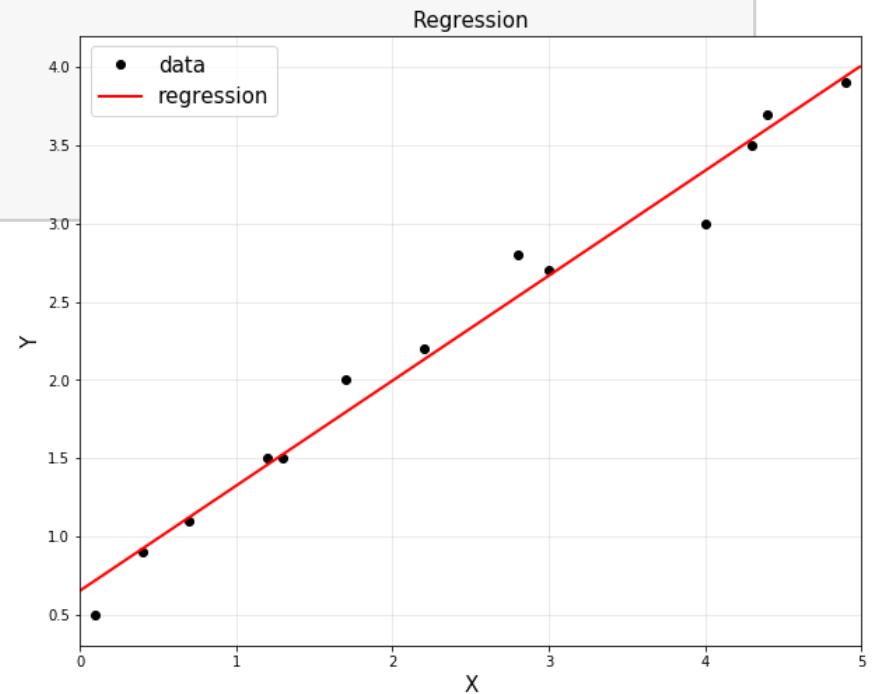
```
reg.intercept_
```

```
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()
```



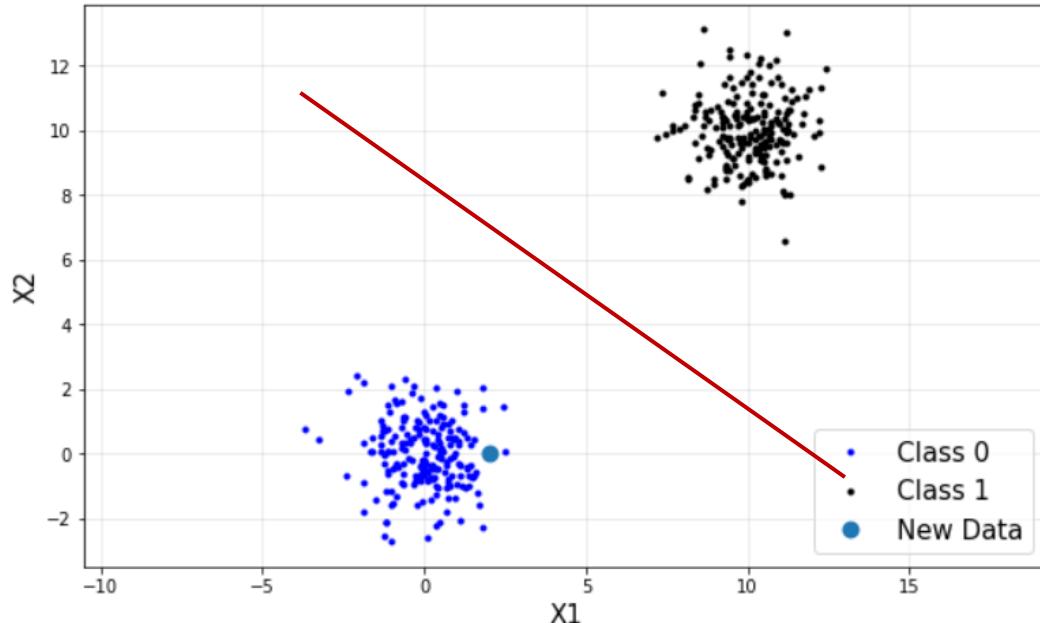
Classification: Perceptron

Classification

- Where y is a discrete value
 - Develop the classification algorithm to determine which class a new input should fall into
- Start with a binary class problem
 - Later look at multiclass classification problem, although this is just an extension of binary classification
- We could use linear regression
 - Then, threshold the classifier output (i.e. anything over some value is yes, else no)
 - linear regression with thresholding seems to work

Classification

- We will learn
 - Perceptron
 - Logistic regression
- To find a classification boundary



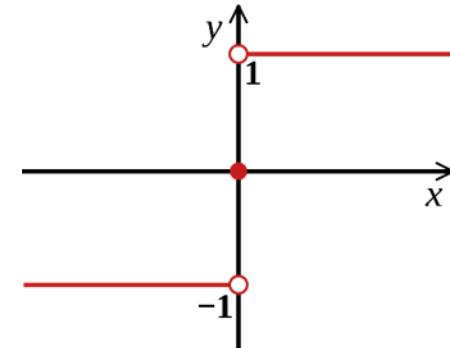
Perceptron

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

- Weights $\omega = \begin{bmatrix} \omega_1 \\ \vdots \\ \omega_d \end{bmatrix}$

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

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



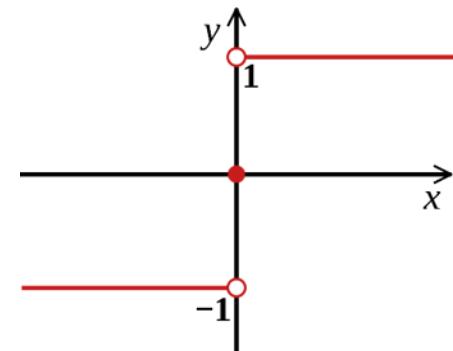
$$h(x) = \text{sign}\left(\left(\sum_{i=1}^d \omega_i x_i\right) - \text{threshold}\right) = \text{sign}\left(\left(\sum_{i=1}^d \omega_i x_i\right) + \omega_0\right)$$

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 $x_0 = 1$:

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

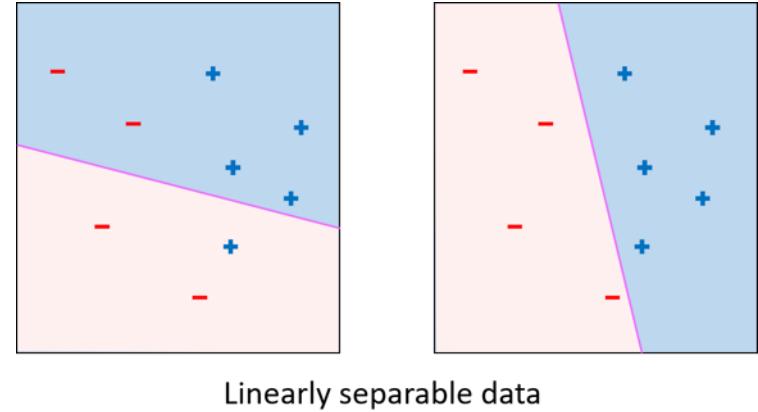


- In a vector form, the perceptron implements

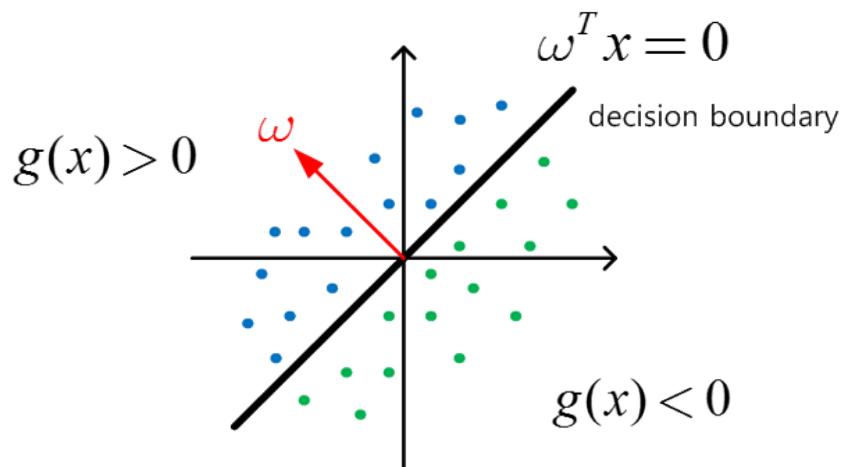
$$h(x) = \text{sign} (\omega^T x)$$

Perceptron

- Works for linearly separable data
- Hyperplane
 - Separates a D-dimensional space into two half-spaces
 - Defined by an outward pointing normal vector
 - ω is orthogonal to any vector lying on the hyperplane
 - Assume the hyperplane passes through origin, $\omega^T x = 0$ with $x_0 = 1$



Linearly separable data

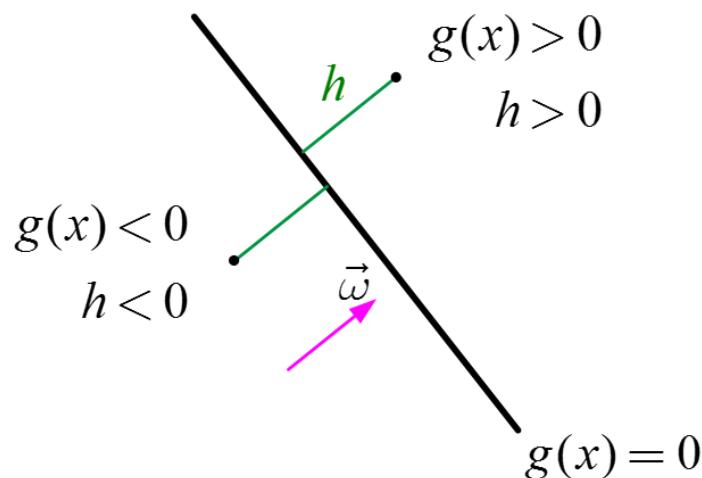


Sign

- Sign with respect to a line

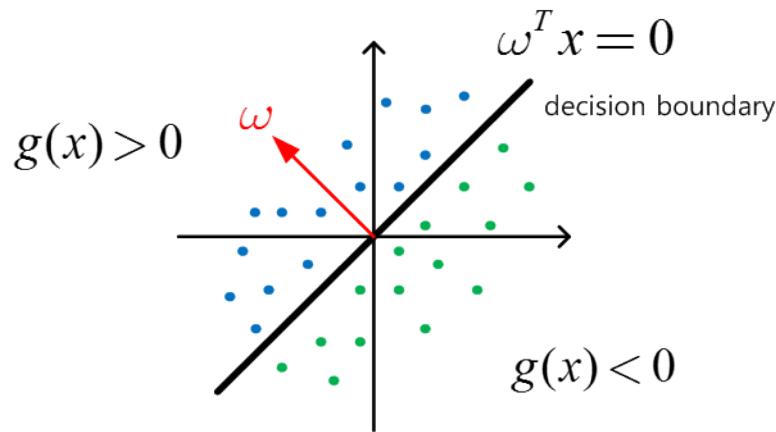
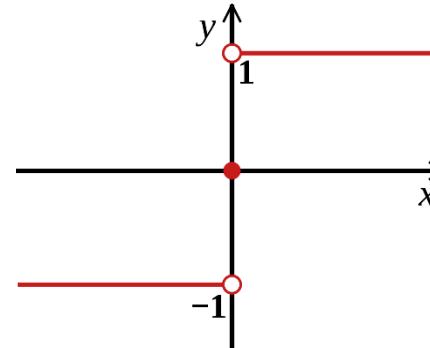
$$\omega = \begin{bmatrix} \omega_1 \\ \omega_2 \end{bmatrix}, \quad x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \implies g(x) = \omega_1 x_1 + \omega_2 x_2 + \omega_0 = \omega^T x + \omega_0$$

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



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

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

- Given the training set

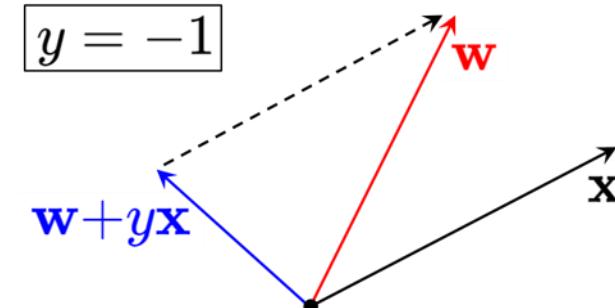
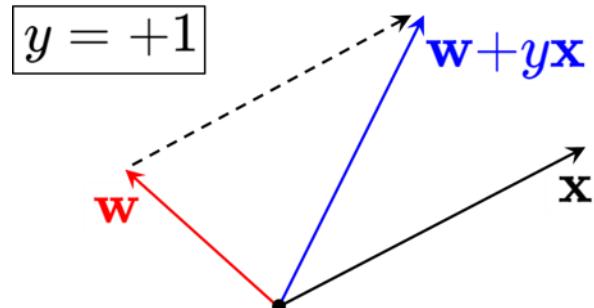
$$(x_1, y_1), (x_2, y_2), \dots, (x_N, y_N) \quad \text{where } y_i \in \{-1, 1\}$$

- pick a misclassified point

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

- and update the weight vector

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



Perceptron Algorithm

- Why perceptron updates work ?
- Let's look at a misclassified positive example ($y_n = +1$)
 - Perceptron (wrongly) thinks $\omega_{old}^T x_n < 0$
 - Updates would be

$$\omega_{new} = \omega_{old} + y_n x_n = \omega_{old} + x_n$$

$$\omega_{new}^T x_n = (\omega_{old} + x_n)^T x_n = \omega_{old}^T x_n + x_n^T x_n$$

- Thus $\omega_{new}^T x_n$ is **less negative** than $\omega_{old}^T x_n$

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 $t = 1, 2, 3, \dots$, pick a misclassified point from

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

4. And run a PLA iteration on it
5. That's it!

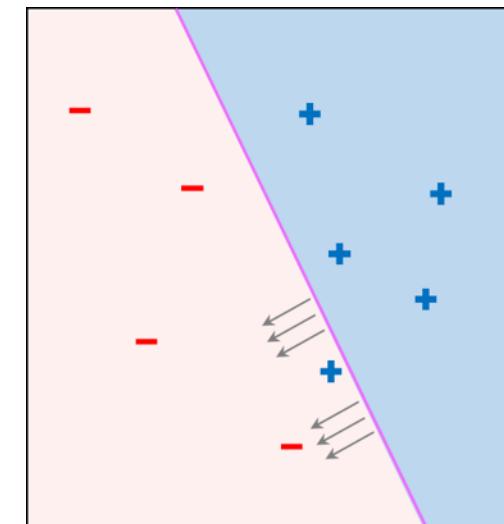
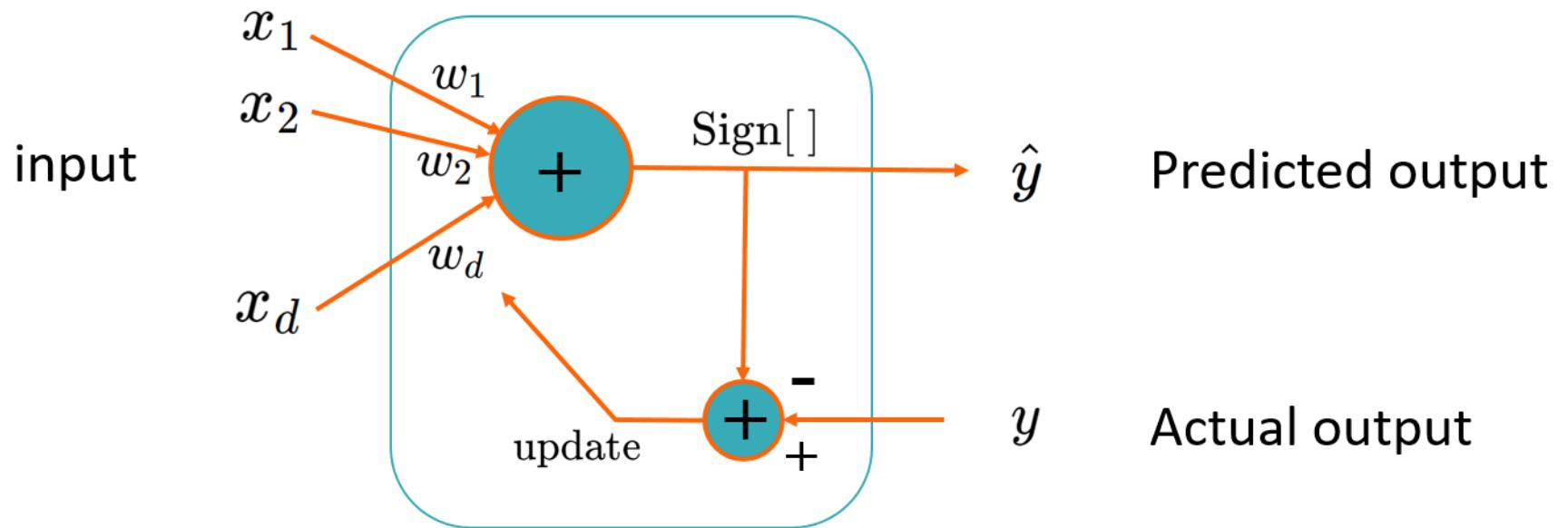


Diagram of Perceptron



Perceptron Algorithm in Python

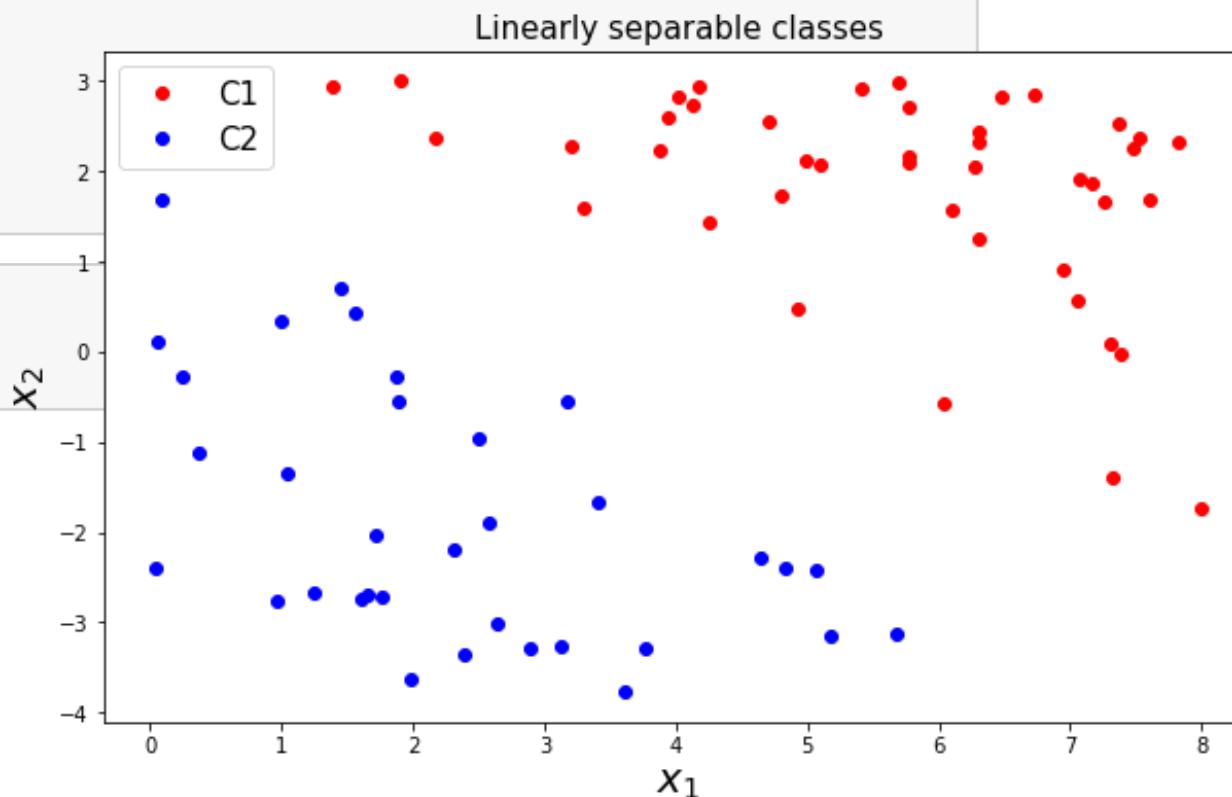
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

```
#training data generation
```

```
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

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

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

```
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

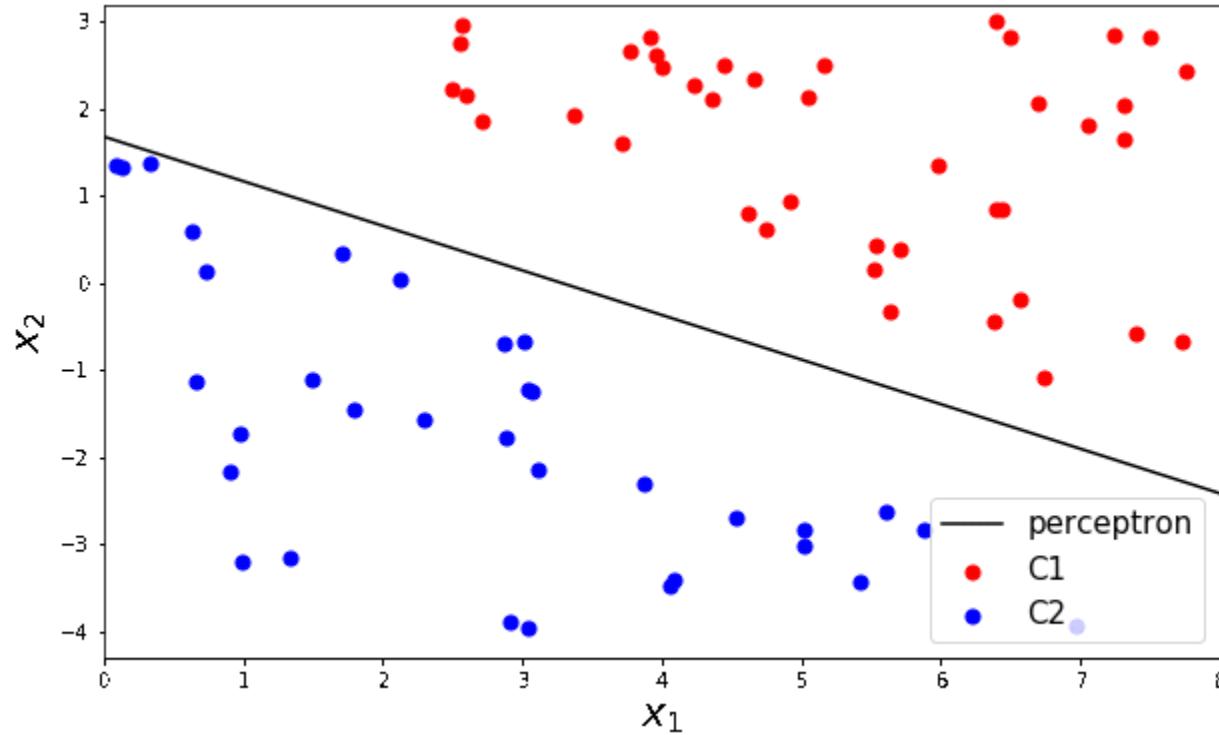
$$g(x) = \omega_0 + \omega^T x = \omega_0 + \omega_1 x_1 + \omega_2 x_2 = 0$$

$$\implies x_2 = -\frac{\omega_1}{\omega_2}x_1 - \frac{\omega_0}{\omega_2}$$

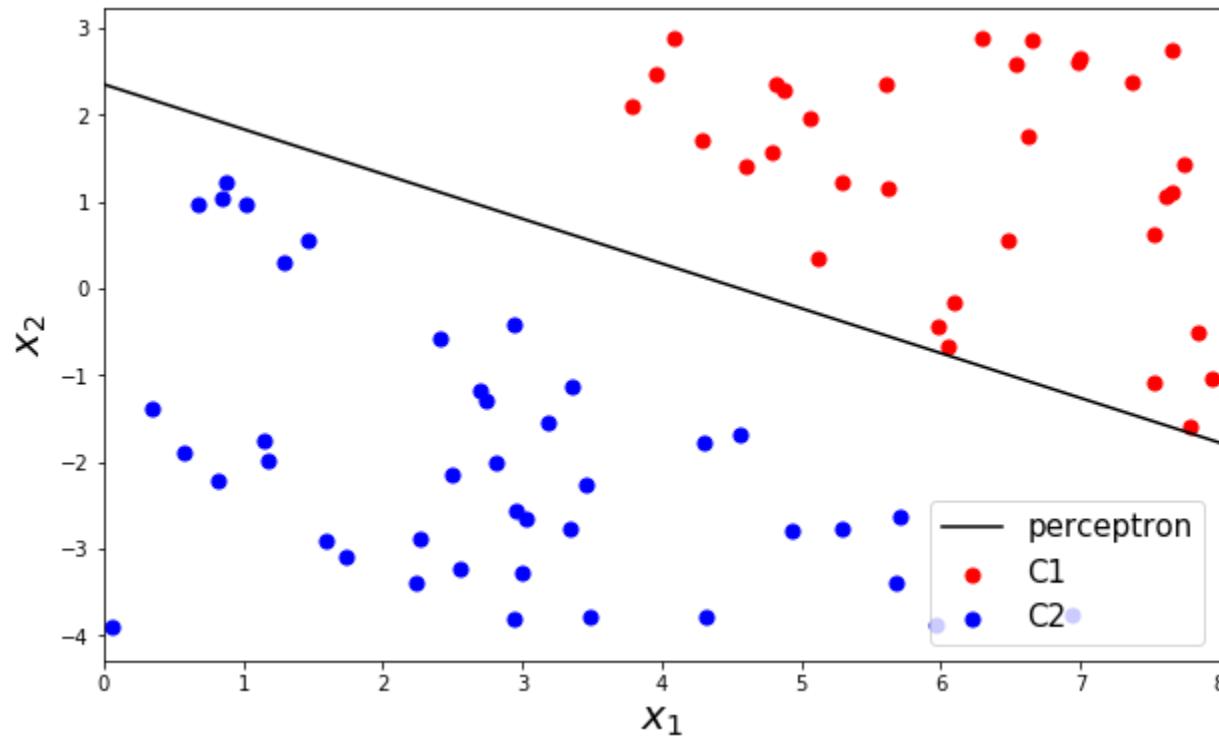
```
x1p = np.linspace(0,8,100).reshape(-1,1)
x2p = - w[1,0]/w[2,0]*x1p - w[0,0]/w[2,0]

plt.figure(figsize=(10, 6))
plt.scatter(x1[C1], x2[C1], c='r', s=50, label='C1')
plt.scatter(x1[C2], x2[C2], c='b', s=50, label='C2')
plt.plot(x1p, x2p, c='k', label='perceptron')
plt.xlim([0,8])
plt.xlabel('$x_1$', fontsize = 20)
plt.ylabel('$x_2$', fontsize = 20)
plt.legend(loc = 1, fontsize = 15)
plt.show()
```

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.])
```

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

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

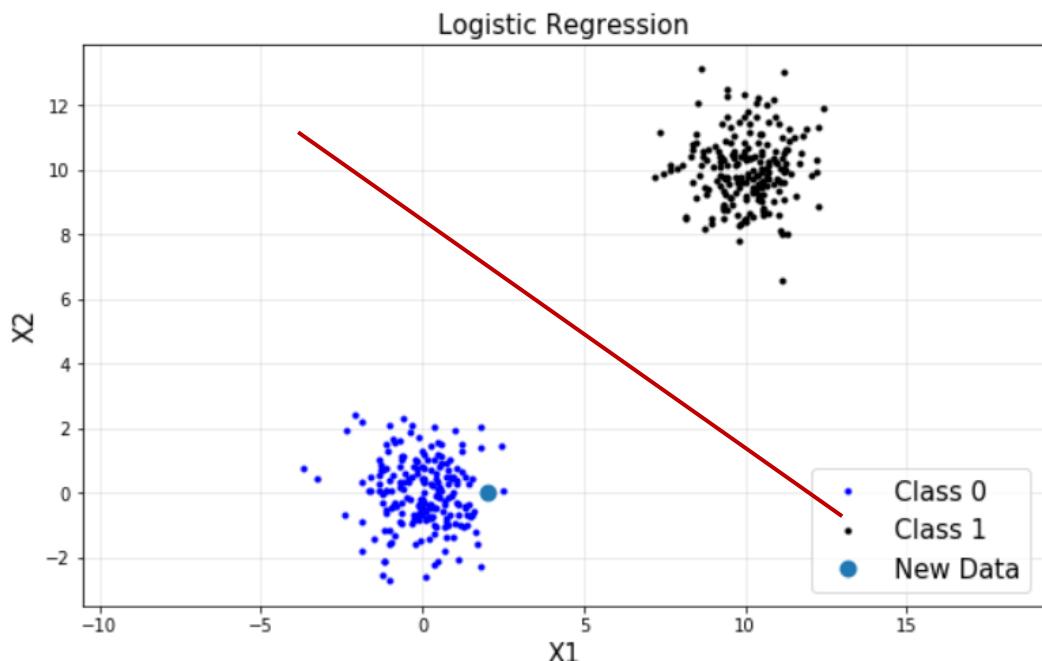
The Best Hyperplane Separator?

- 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
 - Intuitively we want the hyperplane having the maximum margin
 - Large margin leads to good generalization on the test data
 - we will see this formally when we discuss Support Vector Machine (SVM)
- Utilize distance information from all data samples
 - We will see this formally when we discuss the logistic regression
- Perceptron will be shown to be a basic unit for neural networks and deep learning later

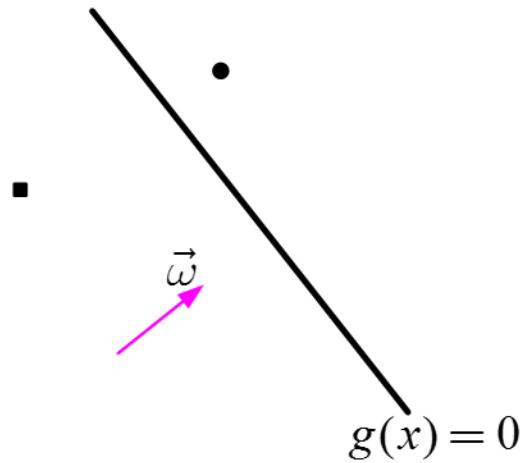
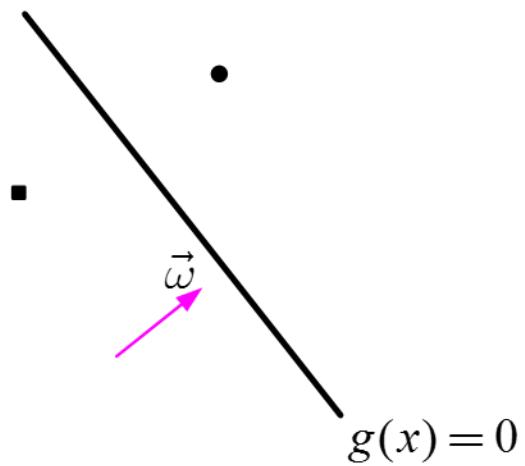
Classification: Logistic Regression

Classification: Logistic Regression

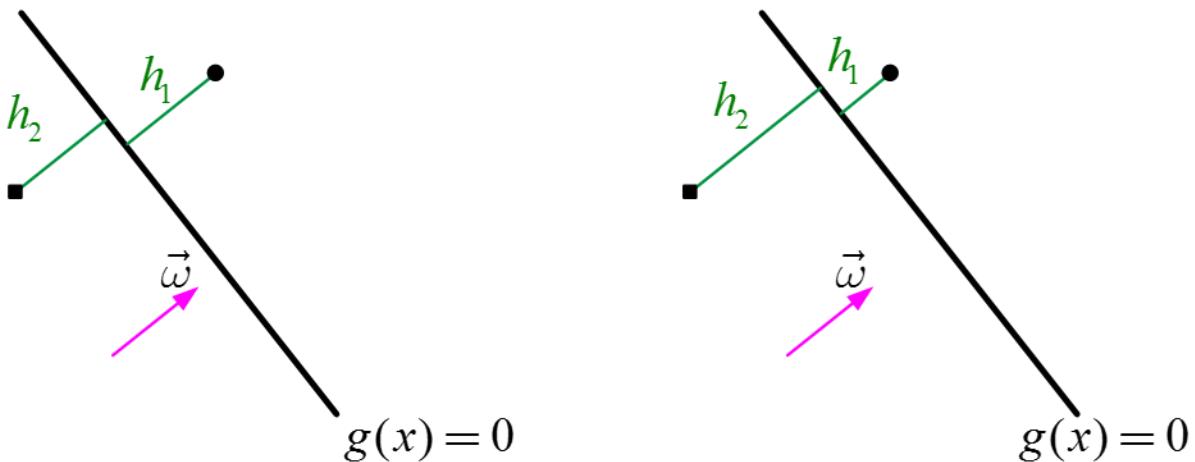
- Perceptron: make use of sign of data
- Logistic regression: make use of distance of data
- Logistic regression is a classification algorithm
 - don't be confused from its name
- To find a classification boundary



Using Distances



Using Distances



$$|h_1| + |h_2|$$

$$|h_1| + |h_2|$$

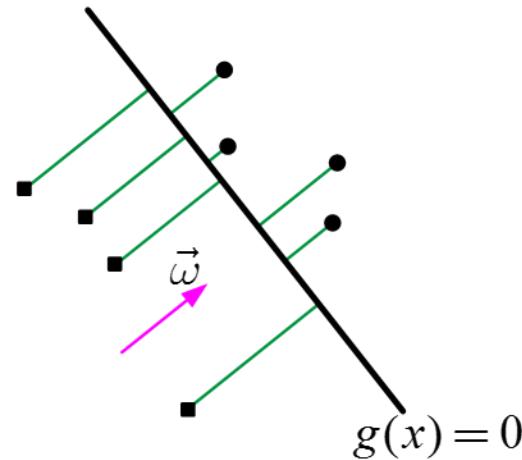
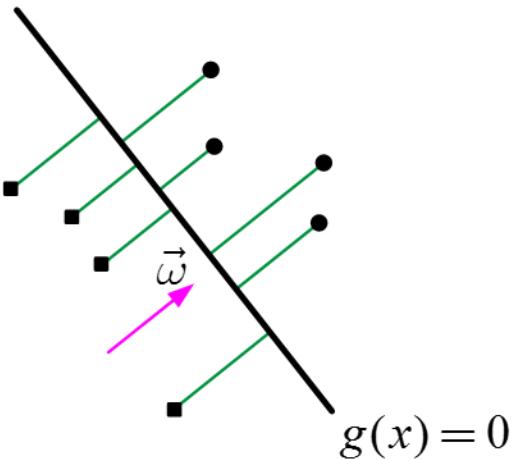
$$|h_1| \cdot |h_2|$$

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

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

Using all Distances

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

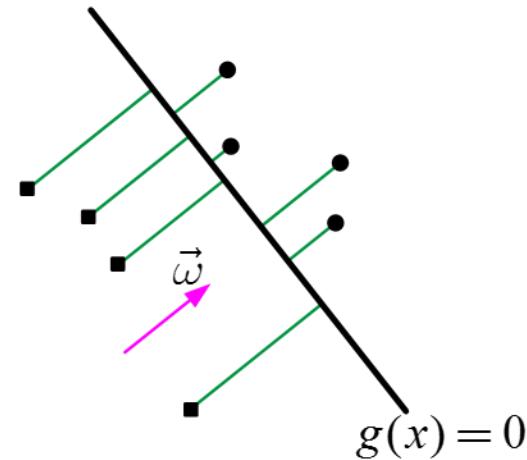
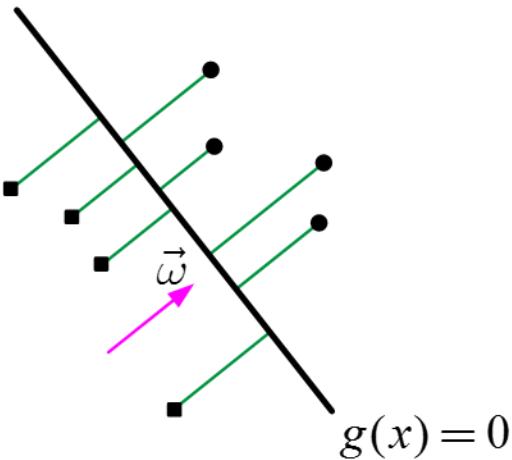


- Inequality of arithmetic and geometric means

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

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

Using all Distances

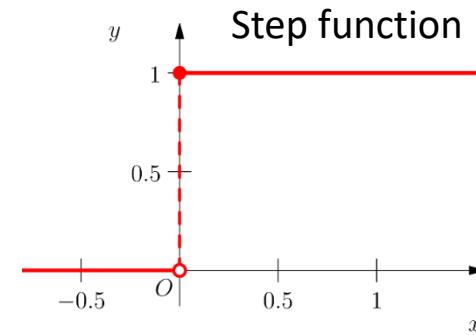
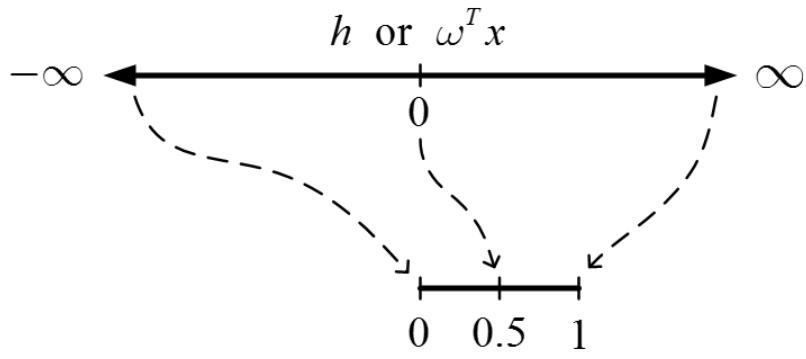


- Roughly speaking, this optimization of $\max \prod_i |h_i|$ tends to position a hyperplane in the middle of two classes

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

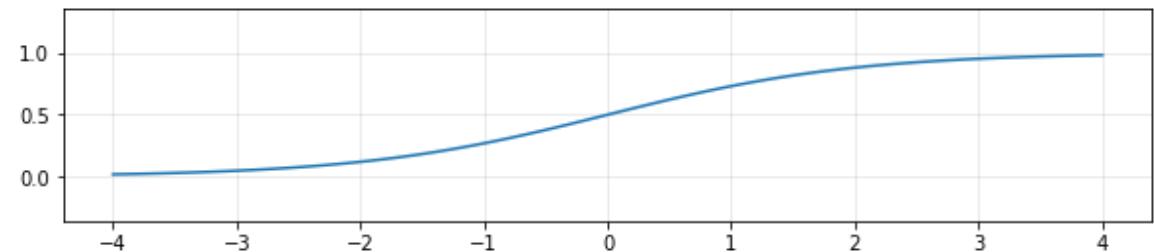
Sigmoid Function

- 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

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

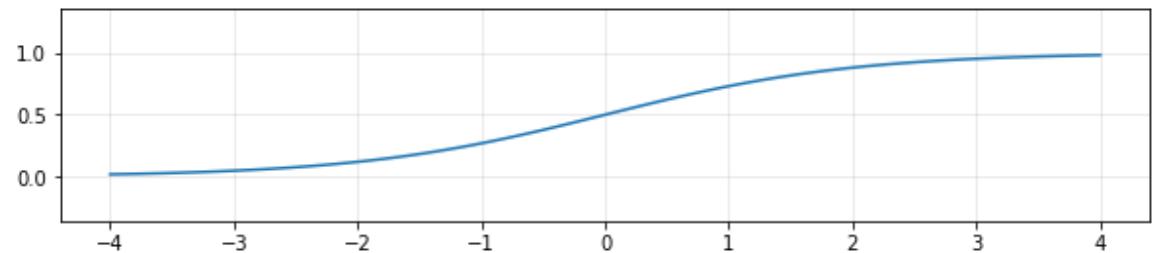


Sigmoid Function

- The derivative of the sigmoid function satisfies

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

$$\sigma'(z) = \sigma(z)(1 - \sigma(z))$$



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

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

- Probability that the label is +1

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

- Probability that the label is 0

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

Goal: we need to fit ω to our data

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

$$P(y = +1 \mid x; \omega) = h_\omega(x) = \sigma(\omega^T x)$$

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

- It can be compactly written as

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

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

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

Goal: we need to fit ω to our data

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

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

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

- 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

Gradient Descent

- To use the gradient descent method, we need to find the derivative of it

$$\nabla \ell(\omega) = \begin{bmatrix} \frac{\partial \ell(\omega)}{\partial \omega_1} \\ \vdots \\ \frac{\partial \ell(\omega)}{\partial \omega_n} \end{bmatrix}$$

- We need to compute $\frac{\partial \ell(\omega)}{\partial \omega_j}$

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

Gradient Descent

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

- Think about a single data point with a single parameter ω for the simplicity.

$$\begin{aligned}\frac{\partial}{\partial \omega} [y \log(\sigma) + (1 - y) \log(1 - \sigma)] \\ &= y \frac{\sigma'}{\sigma} + (1 - y) \frac{-\sigma}{1 - \sigma} \\ &= \left(\frac{y}{\sigma} - \frac{1 - y}{1 - \sigma} \right) \sigma' \\ &= \frac{y - \sigma}{\sigma(1 - \sigma)} \sigma' \\ &= \frac{y - \sigma}{\sigma(1 - \sigma)} \sigma(1 - \sigma)x \\ &= (y - \sigma)x\end{aligned}$$

- For m training data points with parameters ω

$$\frac{\partial \ell(\omega)}{\partial \omega_j} = \sum_{i=1}^m \left(y^{(i)} - h_\omega(x^{(i)}) \right) x_j^{(i)} \stackrel{\text{vectorization}}{=} (y - h_\omega(x))^T x_j = x_j^T (y - h_\omega(x))$$

Gradient Descent for Logistic Regression

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

$$\frac{\partial \ell(\omega)}{\partial \omega_j} = \sum_{i=1}^m \left(y^{(i)} - h_\omega(x^{(i)}) \right) x_j^{(i)}$$

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

$$\stackrel{\text{vectorization}}{=} (y - h_\omega(x))^T x_j = x_j^T (y - h_\omega(x))$$

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

$$\nabla \ell(\omega) = \begin{bmatrix} \frac{\partial \ell(\omega)}{\partial \omega_0} \\ \frac{\partial \ell(\omega)}{\partial \omega_1} \\ \frac{\partial \ell(\omega)}{\partial \omega_2} \end{bmatrix} = X^T (y - h_\omega(x)) = X^T (y - \sigma(X\omega))$$

- Maximization problem
- Be careful on matrix shape

$$\omega \leftarrow \omega - \eta (-\nabla \ell(\omega))$$

Gradient Descent for Logistic Regression

```
# data generation

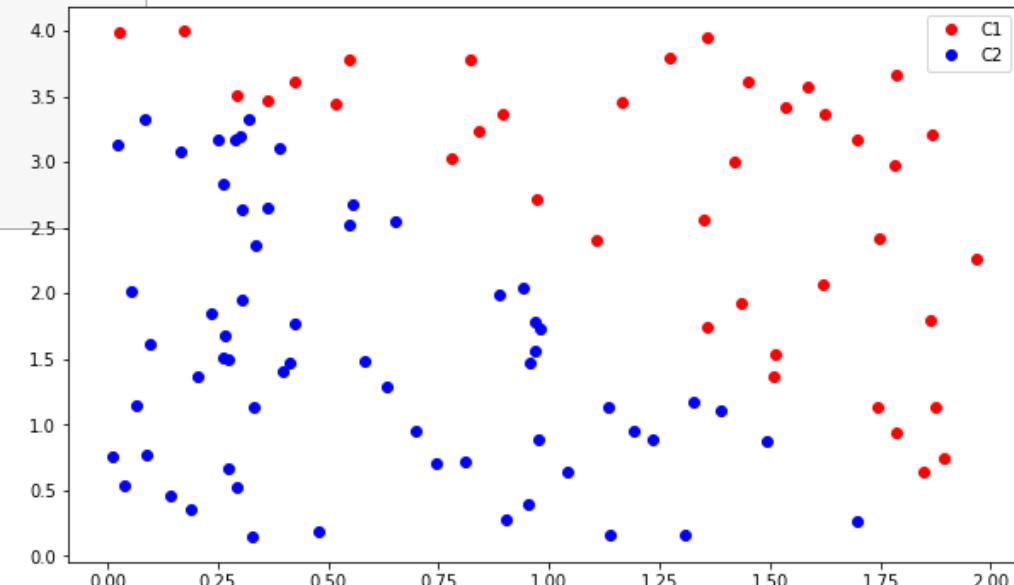
m = 100
w = np.array([[-4], [2], [1]])
X = np.hstack([np.ones([m,1]), 2*np.random.rand(m,1), 4*np.random.rand(m,1)])

w = np.asmatrix(w)
X = np.asmatrix(X)

y = (np.exp(X*w)/(1 + np.exp(X*w))) > 0.5

C1 = np.where(y == True)[0]
C2 = np.where(y == False)[0]

y = np.empty([m,1])
y[C1] = 1
y[C2] = 0
y = np.asmatrix(y)
```



Gradient Descent for Logistic Regression

```
# be careful with matrix shape
def h(x,w):
    return 1/(1 + np.exp(-x*w))
```

```
alpha = 0.0001
w = np.zeros([3,1])

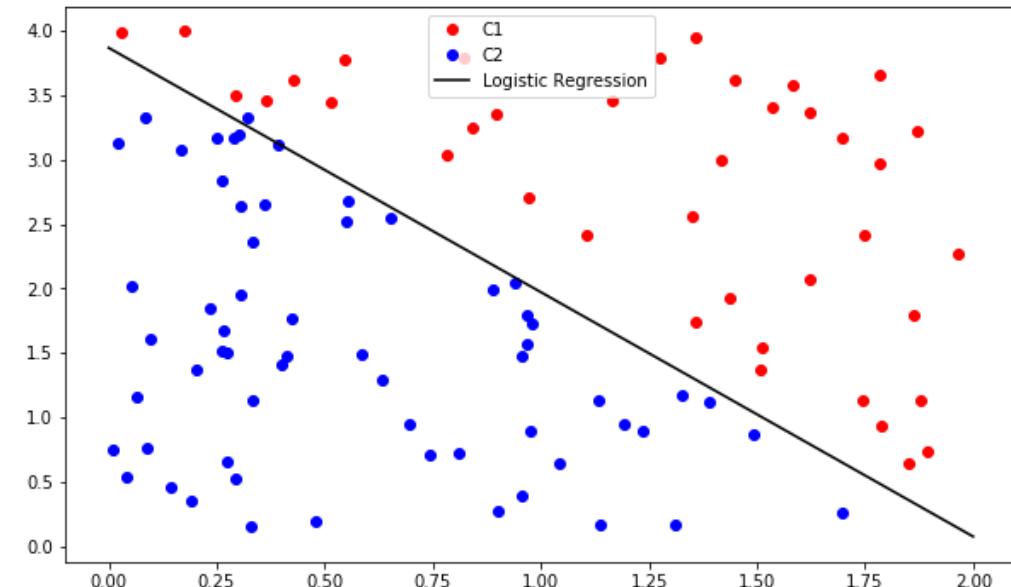
for i in range(1000):
    df = -X.T*(y - h(X,w))
    w = w - alpha*df

print(w)
```

$$h_{\omega}(x) = h(x; \omega) = \sigma(\omega^T x) = \frac{1}{1 + e^{-\omega^T x}}$$

$$\nabla \ell(\omega) = \begin{bmatrix} \frac{\partial \ell(\omega)}{\partial \omega_0} \\ \frac{\partial \ell(\omega)}{\partial \omega_1} \\ \frac{\partial \ell(\omega)}{\partial \omega_2} \end{bmatrix} = X^T (y - h_{\omega}(x)) = X^T (y - \sigma(X\omega))$$

$$\omega \leftarrow \omega - \eta (-\nabla \ell(\omega))$$



Scikit-Learn for Logistic Regression

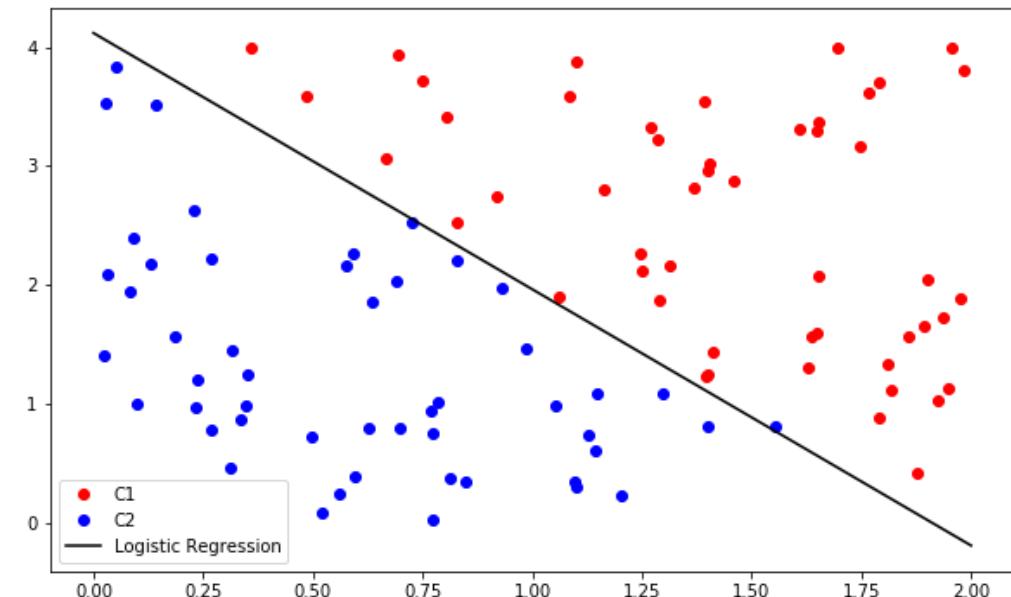
```
X = X[:,1:3]
```

```
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()
```

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

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



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

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

- We maintain a separator weight vector ω_k for each class k
- Note: sigmoid function

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

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	$\begin{aligned}\ell(\omega) = \log \mathcal{L} &= \log P(y \mid x, \omega) = \log \prod_{n=1}^m P(y_n \mid x_n, \omega) \\ &= \sum_{n=1}^m \log P(y_n \mid x_n, \omega)\end{aligned}$