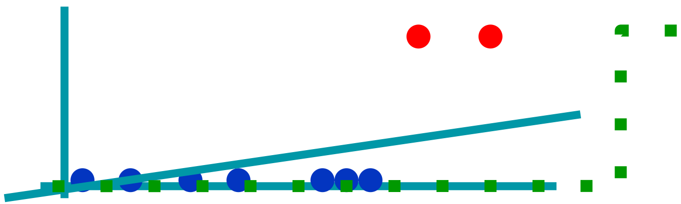
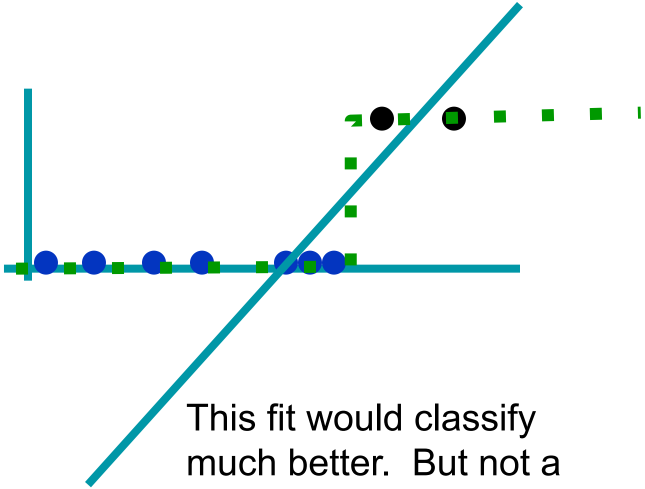
Logistic Regression

Dr. Dong-Chul Kim

Problem of Linear Regression

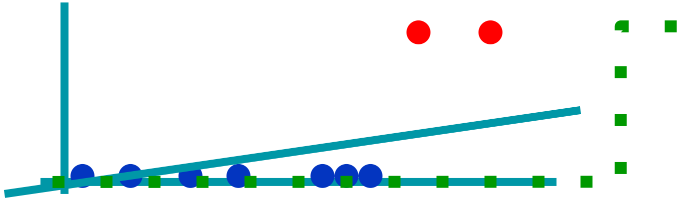


Least squares fit useless

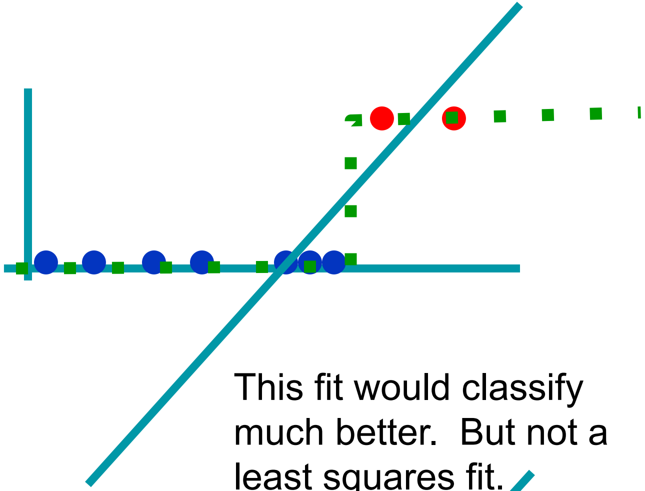


This fit would classify much better. But not a least squares fit.

Problem of Linear Regression



Least squares fit useless



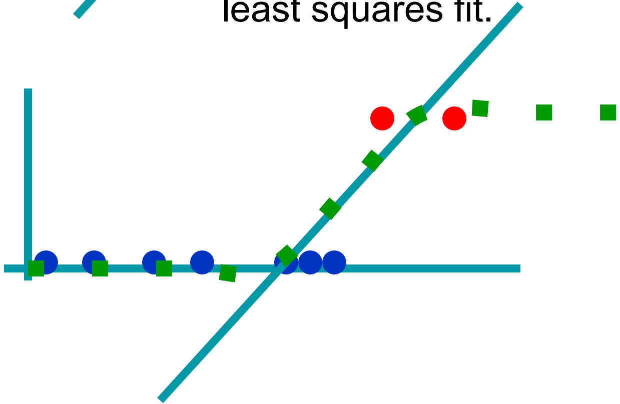
This fit would classify much better. But not a least squares fit.

SOLUTION:

Instead of Out($\boldsymbol{x}$) = $\boldsymbol{w}^T\boldsymbol{x}+\boldsymbol{b}$

We will use Out($\boldsymbol{x}$) = $g(\boldsymbol{w}^T\boldsymbol{x}+\boldsymbol{b})$

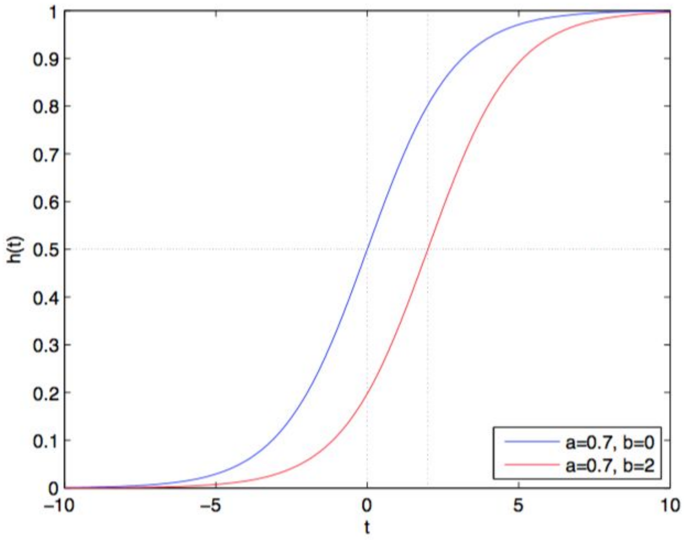
where is a squashing function



The Sigmoid function

$$h(t) = \frac{1}{1 + exp^{-(at+b)}}$$

a is a coefficient to adjust the slope
b is to adjust the position of the center

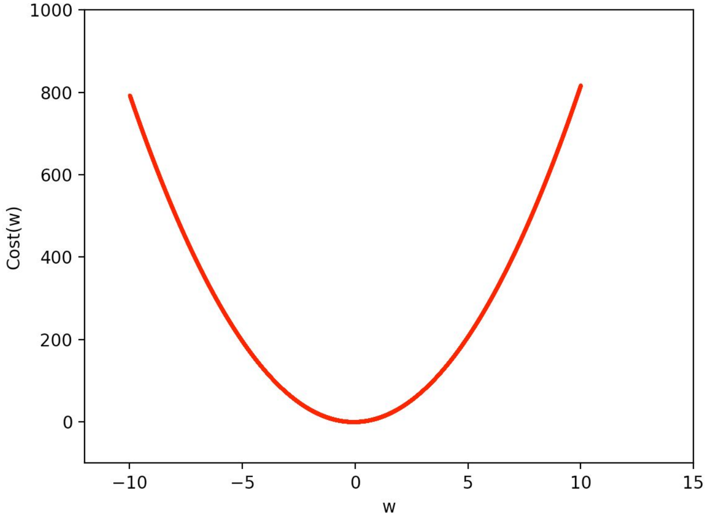


Cost Function

Least Square-based Cost function

$$h(x) = wx + b$$

$$Cost(w) = \sum_{i=1}^{n}(w \cdot x^{(i)} + b - y^{(i)})^2$$



Convex function!

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

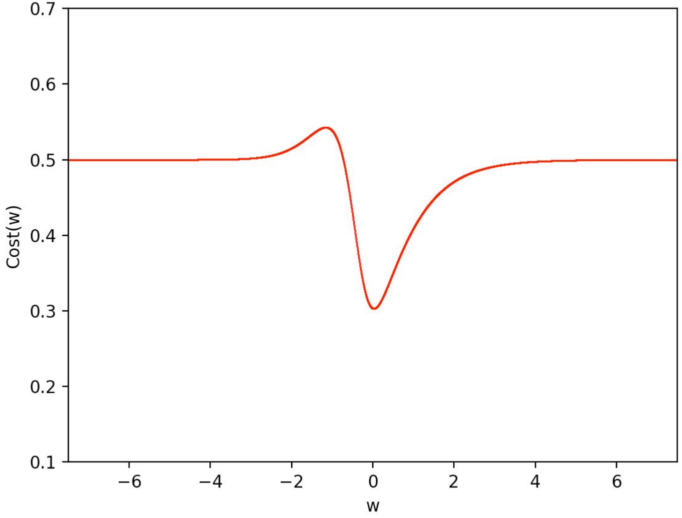
```python
w = np.arange(-10, 10, 0.001)
b = 1
x = np.array([1.3, 1.2, 3.5, 4.1])
y = np.array([0, 0, 1, 1])
cost = []
for i in range(len(w)):
    cost.append(sum((w[i] * x + b - y) ** 2) / len(x))
```

```python
plt.plot(w, cost, 'ro', markersize=0.1)
plt.axis([-12, 15, -100, 1000])
plt.xlabel("w")
plt.ylabel("Cost(w)")
plt.show()
```

Least Square-based Cost function

$$h(x) = \frac{1}{1+e^{-(wx+b)}}$$

$$Cost(w) = \sum_{i=1}^{n}(h(x^{(i)}) - y^{(i)})^2$$



non-Convex function!

```python
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
cost = []
for i in range(len(w)):
    cost.append(sum((sigmoid(w[i] * x + b) - y) ** 2) / len(x))
```

```python
plt.plot(w, cost, 'ro', markersize=0.1)
plt.axis([-7.5, 7.5, 0.1, 0.7])
plt.xlabel("w")
plt.ylabel("Cost(w)")
plt.show()
```

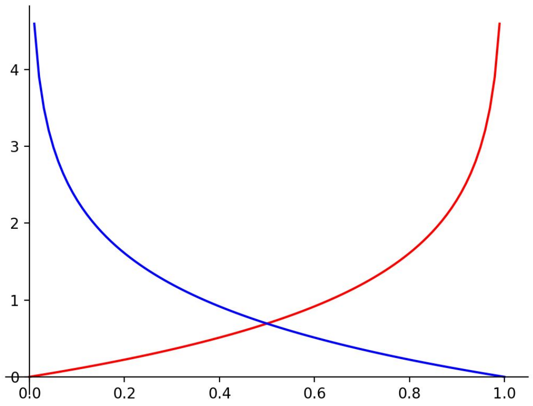
New Cost/Loss Function

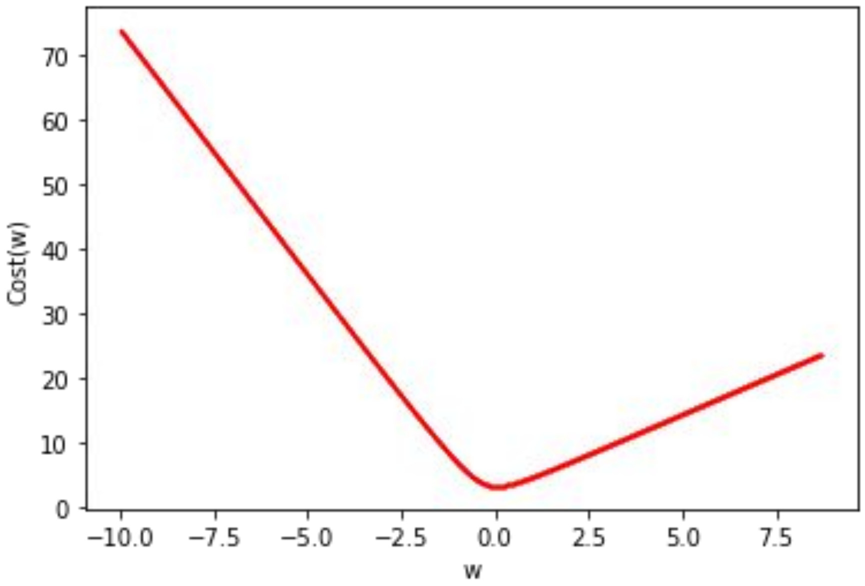
$$Cost(h) = \sum_{i=1}^{n} -y^{(i)} \log h(x^{(i)}) - (1 - y^{(i)}) \log(1 - h(x^{(i)}))$$

New Cost/Loss Function

$$Cost(h) = -y \log h(x) - (1 - y) \log(1 - h(x))$$

When y is 1   When y is 0





Convex function!

```python
cost = []
for i in range(len(w)):
    cost.append(sum(-y*np.log(sigmoid(w[i]*x+b))-(1-y)*np.log(1-sigmoid(w[i]*x+b))))
```

```python
plt.plot(w, cost, 'ro', markersize=0.1)
plt.xlabel("w")
plt.ylabel("Cost(w)")
plt.show()
```

Gradient Descent Algorithm to optimize w and b

1-Dimension

- We can solve it using GD.

Algorithm 1 One dimension logistic regression

Name: LR1
Input: t; y
Output: a; b

1: Initialize a and b
2: **repeat**
3: $\quad a = a + \tau \frac{\partial \ell}{\partial a}$
4: $\quad b = b + \tau \frac{\partial \ell}{\partial b}$
5: **until** convergence of a and b

- The derivative functions in step 3 and 4 are

$$\frac{\partial \ell}{\partial a} = \sum_{i=1}^{n} [y^{(i)} - h(t^{(i)})] t^{(i)}$$

$$\frac{\partial \ell}{\partial b} = \sum_{i=1}^{n} [y^{(i)} - h(t^{(i)})]$$

Multi-Dimension

- When the feature space is high dimension $x \in R^m$, the regression function becomes:

$$h_w(x) = \frac{1}{1 + e^{-(\mathbf{w}^T\mathbf{x})}}$$

$\boldsymbol{x} = [1, x_1, x_2, \cdots, x_m]$, $\boldsymbol{w} = [w_0, w_1, \cdots, w_{m+1}]$ are the coefficients.

- The Log-likelihood function/Loss function is still

$$\ell(h) = \sum_{i=1}^{n} y^{(i)} \ln h(t^{(i)}) + (1 - y^{(i)}) ln(1 - h(t^{(i)}))$$

Multi-Dimension

Algorithm 1 Multi-dimension Logistic Regression

Input: $\mathbf{x}$; y
Output: $\mathbf{w}$

1: Initialize $\mathbf{w}$
2: **repeat**
3:     **for** $j = 1$ to $m + 1$ **do**
4:         $w_j = w_j + \alpha \frac{\partial l}{\partial w_j}$
5:     **end for**
6: **until** convergence of w

- The derivative function in step 4 is $$\frac{\partial \ell}{\partial w_j} = \sum_{i=1}^{n} (y^{(i)} - h_w(\mathbf{x}^{(i)}))x_j^{(i)}$$

Logistic regression 1D

```python

import numpy as np
import matplotlib.pyplot as plt

data = np.array([[2, 0], [4, 0], [6, 0], [8, 1], [10, 1], [12, 1], [14, 1]])
trainx = data[:, 0]
trainy = data[:, 1]
# initialization
a = 0
b = 0
lr = 0.05


def sigmoid(x):
    return 1 / (1 + np.e ** (-x))


# GD
for i in range(2001):
    a_diff = sum((trainy - sigmoid(a*trainx + b))*trainx)
    b_diff = sum(trainy - sigmoid(a*trainx + b))
    a = a + lr * a_diff
    b = b + lr * b_diff

print(a, b)
plt.scatter(trainx, trainy)
plt.xlim(0, 15)
plt.ylim(-.1, 1.1)
x_range = (np.arange(0, 15, 0.1))
plt.plot(np.arange(0, 15, 0.1), np.array([sigmoid(a*x + b) for x in x_range]))
plt.show()
```

Logistic regression MD

```python
import ...

trainX = np.array([[1.5, 2.7, 1.3, 1],
                   [2.4, 1.7, 2.1, 1],
                   [2.5, 1.3, 2.2, 1],
                   [8.5, 5.3, 4.8, 1],
                   [4.9, 6.4, 5.7, 1],
                   [7.2, 7.1, 7.4, 1]])
trainy = np.array([0, 0, 0, 1, 1, 1])
testX = np.array([[2.4, 2.5, 0.7, 1],
                  [5.9, 4.4, 5.2, 1],
                  [0.2, 0.5, 0.6, 1],
                  [4.3, 4.5, 5.5, 1]])
testy = np.array([0, 1, 0, 1])

# initialization
w = np.zeros(4)
lr = 0.05


def sigmoid(x):
    return 1 / (1 + np.e ** (-x))


# GD
for i in range(1000):
    for j in range(4):
        w_diff = np.dot(trainy - sigmoid(np.dot(trainX, w)), trainX[:, j])
        w[j] = w[j] + lr * w_diff

# test
print(sum(np.round(sigmoid(np.dot(testX, w))) == testy)/np.size(testy))
```

Logistic regression MD (matrix)

```
import ...


def sigmoid(x):
    return 1 / (1 + np.e ** (-x))


trainX = np.array([[1.5, 2.7, 1.3, 1],
                   [2.4, 1.7, 2.1, 1],
                   [2.5, 1.3, 2.2, 1],
                   [8.5, 5.3, 4.8, 1],
                   [4.9, 6.4, 5.7, 1],
                   [7.2, 7.1, 7.4, 1]])
trainy = np.array([0, 0, 0, 1, 1, 1])
testX = np.array([[2.4, 2.5, 0.7, 1],
                  [5.9, 4.4, 5.2, 1],
                  [0.2, 0.5, 0.6, 1],
                  [4.3, 4.5, 5.5, 1]])
testy = np.array([0, 1, 0, 1])
# initialization
w = np.zeros(4)
lr = 0.05
# GD
for i in range(1000):
    w_diff = np.dot(np.transpose(trainy - sigmoid(np.dot(trainX, w))), trainX)
    w = w + lr * w_diff
# test
print(sum(np.round(sigmoid(np.dot(testX, w))) == testy)/np.size(testy))
```

Lab

- Implement a logistic regression with Iris data.
- Step 1: Use only first 100 samples (only two classes) to make it binary class data.
- Step 2: Use hold-out method.
  - Shuffle the 100 samples and select the first 10 samples for test.
- Step 3: Repeat step 2 for 100 times, then calculate accuracy on average.