Session 01

From Linear Regression To Neural Network

What is Machine Learning?

A computer program is said to learn from **experience E** with respect to some class of **tasks T** and performance **measure P**, if its performance at tasks in **T**, as measured by **P**, improves with experience **E**.

The Task, **T**

- Classification
- Regression
- Machine translation
- Anomaly detection
- Synthesis
- Denoising

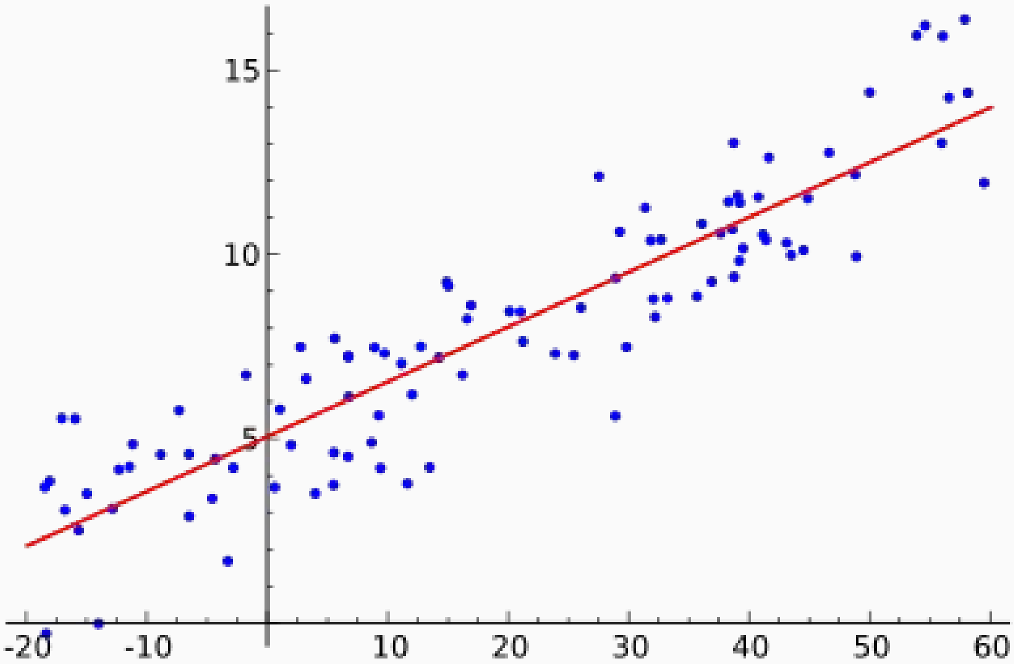
Performance Measure, **P**

- Task specific
- Accuracy
- Win/Loss Ratio
- Revenue

The Experience, **E**

- Supervised
- Unsupervised

Linear Regression



Linear Regression

The task, **T**

Given m features (input) $x_1, x_2, \ldots, x_m$, predict (output) y.

Performance Measure, **P**

Minimize average squared error:

$$\frac{1}{n}\sum_{i=0}^{n}\left(y_{pred}^{(i)} - y^{(i)}\right)^2$$

The Experience, **E**

n sets of data, each with m features $x_1^{(i)}, x_2^{(i)}, \ldots, x_m^{(i)}$ and the output $y^{(i)}$

$$\begin{array}{ccccc}
x_1^{(1)} & x_2^{(1)} & \ldots & x_m^{(1)} \longrightarrow & y^{(1)} \\
x_1^{(2)} & x_2^{(2)} & \ldots & x_m^{(2)} \longrightarrow & y^{(2)} \\
\vdots & \vdots & \ddots & \vdots & \vdots \\
x_1^{(n)} & x_2^{(n)} & \ldots & x_m^{(n)} \longrightarrow & y^{(n)}
\end{array}$$

Linear Model

We assume that y linearly depends on $x_1, \ldots, x_m$.

That is there exists $w_1, w_2, \ldots, w_m$ such that,

$$y_{pred} = w_1 x_1 + \ldots + w_m x_m$$

Then the squared error becomes a function of w,

$$\begin{aligned} F(w) &= \frac{1}{n} \sum_{i=0}^{n} \left(y_{pred}^{(i)} - y^{(i)} \right)^2 \\ &= \frac{1}{n} \sum_{i=0}^{n} \left(\left(\sum_{j=0}^{m} w_j x_j^{(i)} \right) - y^{(i)} \right)^2 \end{aligned}$$

Gradient Descent

Gradient of a function is the vector of all of its partial derivatives.

$$\nabla F(w) = \begin{bmatrix} \frac{\partial F}{\partial w_1} \\ \frac{\partial F}{\partial w_2} \\ \vdots \\ \frac{\partial F}{\partial w_m} \end{bmatrix}$$

To find w that minimizes the function $F(w)$, we start with a random $w^{(0)}$ and then repeatedly subtract its gradient from it,

$$\begin{aligned} w_1^{(t+1)} &= w_j^{(t)} - \lambda \nabla F(w^{(t)})_j \\ w_2^{(t+1)} &= w_j^{(t)} - \lambda \nabla F(w^{(t)})_j \\ &\vdots \\ w_m^{(t+1)} &= w_j^{(t)} - \lambda \nabla F(w^{(t)})_j \end{aligned}$$

Gradient Computation

$$F(w) = \frac{1}{n} \sum_{i=0}^{n} \left(w^T x^{(i)} - y^{(i)} \right)^2$$

$$\frac{\partial F(w)}{\partial w_j} = \frac{\partial \left(\frac{1}{n} \sum_{i=0}^{n} \left(w^T x^{(i)} - y^{(i)} \right)^2 \right)}{\partial w_j}$$

$$= \frac{1}{n} \sum_{i=0}^{n} \frac{\partial \left(w^T x^{(i)} - y^{(i)} \right)^2}{\partial w_j}$$

$$= \frac{1}{n} \sum_{i=0}^{n} \frac{\partial \left(w^T x^{(i)} - y^{(i)} \right)}{\partial w_j} \frac{\partial \left(w^T x^{(i)} - y^{(i)} \right)^2}{\partial \left(w^T x^{(i)} - y^{(i)} \right)}$$

$$= \frac{1}{n} \sum_{i=0}^{n} \frac{\partial \left(w^T x^{(i)} - y^{(i)} \right)}{\partial w_j} \times 2 \left(w^T x^{(i)} - y^{(i)} \right)$$

$$\frac{\partial \left(w^T x - y^{(i)} \right)}{\partial w_j}$$

$$= \frac{\partial \left(w_1 x_1 + \ldots + w_j x_j + \ldots + w_m x_m - y^{(i)} \right)}{\partial w_j}$$

$$= x_j$$

$$\nabla_j = \frac{\partial F}{\partial w_j}$$

$$= \frac{1}{n} \sum_{i=0}^{n} \frac{\partial \left(w^T x^{(i)} - y^{(i)} \right)}{\partial w_j} \times 2 \left(w^T x^{(i)} - y^{(i)} \right)$$

$$= \frac{2}{n} \sum_{i=0}^{n} x_j^{(i)} \left(w^T x^{(i)} - y^{(i)} \right)$$

```python
# Computes gradients from X, Y, w
def get_gradients(X, y, w):
  n, m = len(X), len(X[0])

  wx = [0] * n
  # wx[i] = w[0] * x[i][0] + ... + w[i][m-1]
  for i in range(n):
    for j in range(m):
      wx[i] += w[j] * x[i][j]

  gradient = [0] * m
  # gradient[j]
  # = 2 / n * (x[0] * (wx[0] - y[0]) + ... + 2 * x[n-1] * (wx[n-1] - y[n-1]))
  for j in range(m):
    for i in range(n)
      gradient[j] += (2.0 / n) * x[i][j] * (wx[i] - y[i])

  return gradient

# Given X and y, learns w with learning_rate
def linear_regression(X, y, learning_rate=0.0001):
  n, m = len(X), len(X[0])
  w = [0] * m
  for iteration in range(100000):
    gradient = get_gradients(X, y, w)
    for j in range(m):
      w[j] -= learning_rate * gradient[j]

  return w
```

Vectorization

$$X = \begin{bmatrix} x_1^{(1)} & x_2^{(1)} & \cdots & x_m^{(1)} \\ x_1^{(2)} & x_2^{(2)} & \cdots & x_m^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ x_1^{(n)} & x_2^{(n)} & \cdots & x_m^{(n)} \end{bmatrix} y = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(n)} \end{bmatrix} w = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_m \end{bmatrix}$$

Vectorization

$$x^{(i)} = \begin{bmatrix} x_1^{(i)} \\ x_2^{(i)} \\ \vdots \\ x_m^{(i)} \end{bmatrix} \quad x_j = \begin{bmatrix} x_j^{(1)} \\ x_j^{(2)} \\ \vdots \\ x_j^{(n)} \end{bmatrix}$$

Vectorization

$$\nabla_j = \frac{2}{n}\sum_{i=0}^{n} x_j^{(i)}\left(w^T x^{(i)} - y^{(i)}\right)$$

$$= \frac{2}{n}\begin{bmatrix} x_j^{(1)} & x_j^{(2)} & \dots & x_j^{(n)} \end{bmatrix}\begin{bmatrix} w^T x^{(1)} - y^{(1)} \\ w^T x^{(2)} - y^{(2)} \\ \vdots \\ w^T x^{(3)} - y^{(3)} \end{bmatrix}$$

$$= \frac{2}{n}\begin{bmatrix} x_j^{(1)} & x_j^{(2)} & \dots & x_j^{(n)} \end{bmatrix}\left(\begin{bmatrix} w^T x^{(1)} \\ w^T x^{(2)} \\ \vdots \\ w^T x^{(3)} \end{bmatrix} - \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(3)} \end{bmatrix}\right)$$

$$= \frac{2}{n} x_j^T \left(\begin{bmatrix} w^T x^{(1)} \\ w^T x^{(2)} \\ \vdots \\ w^T x^{(3)} \end{bmatrix} - y\right)$$

$$= \frac{2}{n} x_j^T (Xw - y)$$

```python
from numpy import matrix, zeros

# Computes gradients from X, Y, w
def get_gradients(X, y, w):
  n, m = X.shape

  gradient = zeros((m, 1))
  for j in range(m):
    gradient[j] = (2.0 / n) * X[:,j].T * (X * w - y)

  return gradient

# Given X and y, learns w with learning_rate
def linear_regression(X, y, learning_rate=0.0001):
  n, m = X.shape
  gradient = zeros((m, 1))
  for iteration in range(100000):
    w -= learning_rate * get_gradients(X, y, w)

  return w
```

Vectorization

$$\nabla F(w) = \begin{bmatrix} \frac{\partial F}{\partial w_1} \\ \frac{\partial F}{\partial w_2} \\ \vdots \\ \frac{\partial F}{\partial w_m} \end{bmatrix} = \frac{2}{n} \begin{bmatrix} x_1^T(Xw - y) \\ x_2^T(Xw - y) \\ \vdots \\ x_m^T(Xw - y) \end{bmatrix}$$

$$= \frac{2}{n} \begin{bmatrix} x_1^T & x_2^T & \dots & x_m^T \end{bmatrix} \begin{bmatrix} (Xw - y) \\ (Xw - y) \\ \vdots \\ (Xw - y) \end{bmatrix}$$

$$= \frac{2}{n} X^T (Xw - y)$$

```python
from numpy import matrix, zeros

# Computes gradients from X, Y, w
def get_gradients(X, y, w): return (2.0 / n) * X.T * (X * w - y)

# Given X and y, learns w with learning_rate
def linear_regression(X, y, learning_rate=0.0001):
  n, m = X.shape
  gradient = zeros((m, 1))
  for iteration in range(100000):
    w -= learning_rate * get_gradients(X, y, w)

  return w
```

Normal Equation

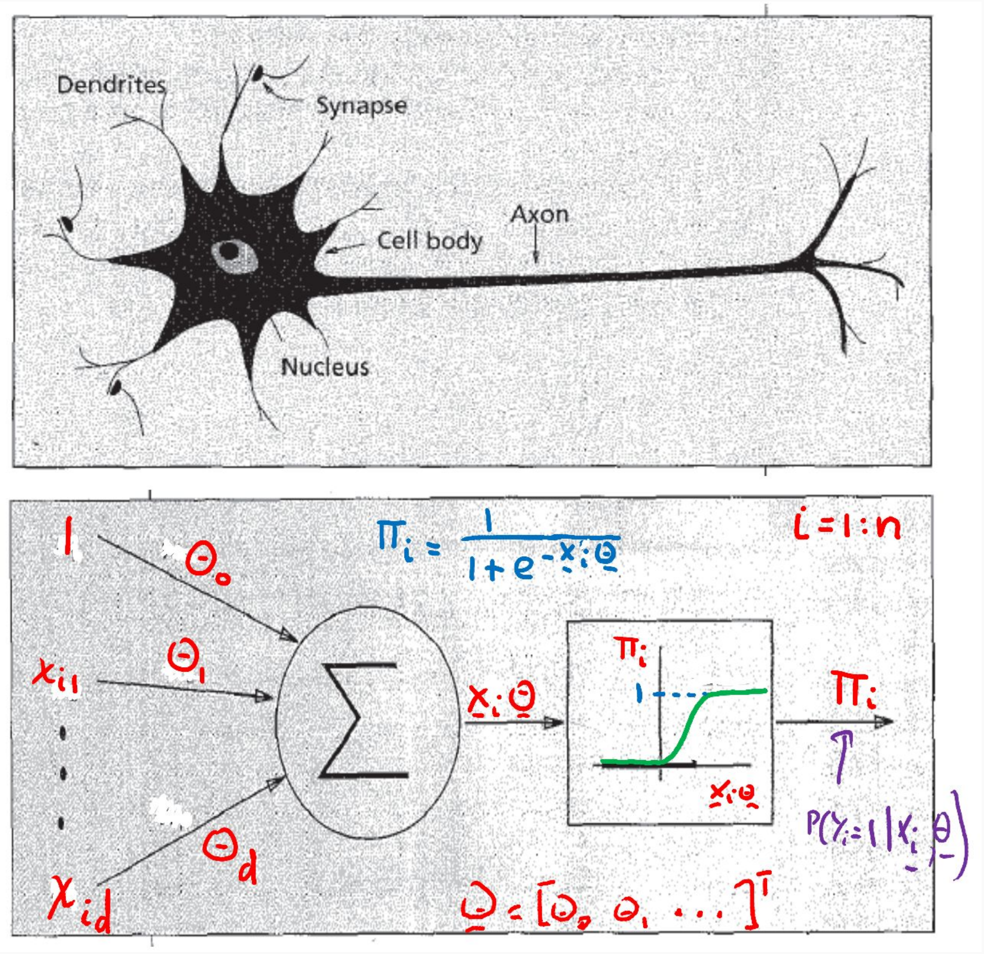
$$
\begin{aligned}
& w^* = w^* - \lambda * \nabla F(w^*) \\
\implies & \nabla F(w^*) = 0 \\
\implies & X^T(Xw^* - y) = 0 \\
\implies & X^T X w^* = X^T y \\
\implies & w^* = (X^T X)^{-1} X^T y
\end{aligned}
$$

```python
from numpy import matrix, zeros
from numpy.linalg import inv # matrix inverse

# Given X and y, learns w
def linear_regression(X, y): return inv(X.T * X) * X.T * y

```

Logistic Regression



Logistic Regression

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

$$F(w) = \frac{1}{n} \sum_{i=0}^{n} cost\left(h_w(x^{(i)}), y^{(i)}\right)$$

Cost Function

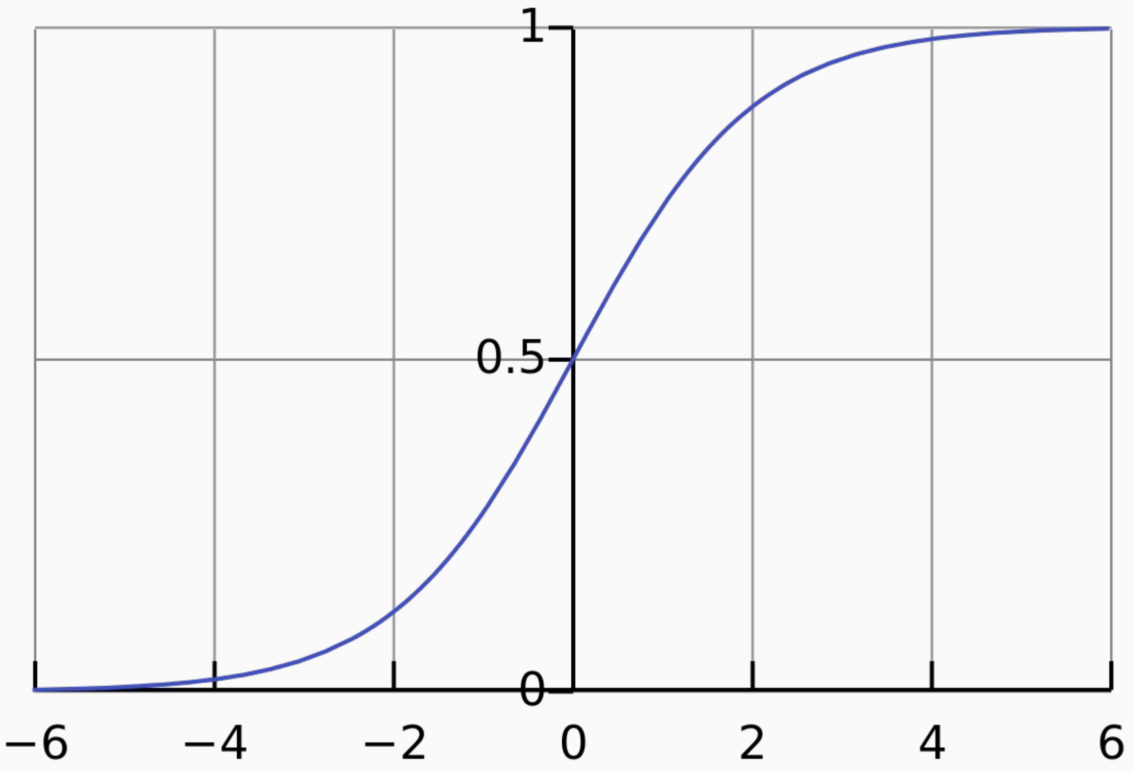
$$cost\left(h_w(x), y\right)$$

$$= \begin{cases} -\log(h_w(x)) & \text{if } y = 1 \\ -\log(1 - h_w(x)) & \text{if } y = 0 \end{cases}$$

$$= -(y\log(h_w(x)) + (1-y)\log(1-h_w(x)))$$

$$F(w) = -\frac{1}{n}\sum_{i=0}^{n}(y\log(h_w(x)) + (1-y)\log(1-h_w(x)))$$

Sigmoid Function



Neural Network