Logistic Regression

Homework #2

32181928 박찬호

Dankook University

Mobile Systems Engineering

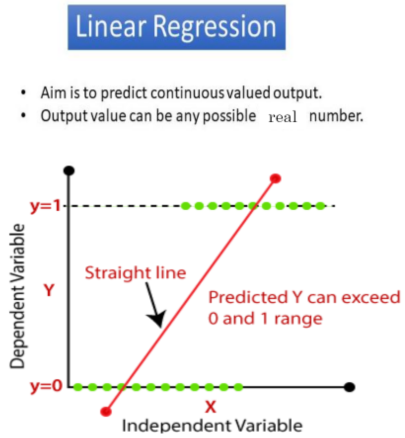
2022 Fall

Contents 목차

서론

Logistic Regression

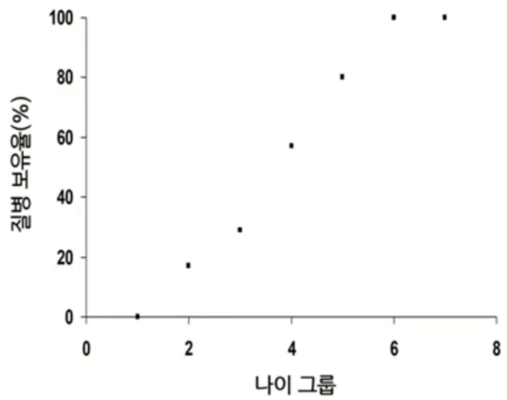
Logistic Regression 이란 무엇일까? 값을 예측하는 regression 문제에서 연속적인 label 이 존재하고 이들이 linear 한 관계를 가질 경우 linear regression 은 훌륭한 예측 모델이자 분석 방법이 되어준다. 그러나 label 이 연속적이지 않고 어떠한 class 에 속한다면 linear regression 은 적용하기 어려워진다. 즉, classification 을 위해서는 다른 방식이 필요하다.



출처 : 강의자료

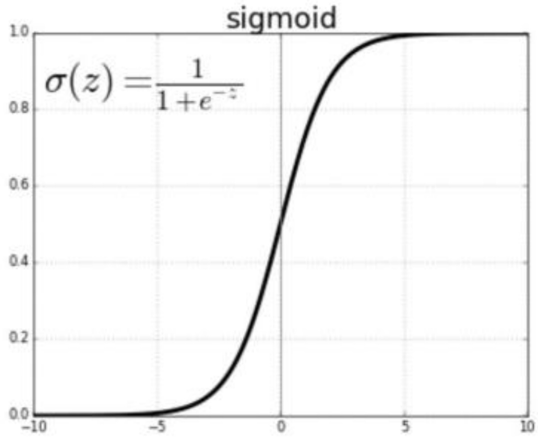
예를 들어 가장 간단한 binary classification 을 생각해보자. Y 가 0 또는 1 이라는 이산적인 값들로 정해져 있기 때문에, 위의 그림처럼 linear regression 의 선형 모델을 직접 적용시킬 수 없다. 이때, 각 데이터의 feature 에 의해 어떤 클래스가 결정되는 확률을 살펴보면 아래와 같다. 나이라는 feature 를 기준으로 Y=1 에 속할 확률을 그려보면 우측과 같은 그래프를 만드는 것을 알 수 있다.

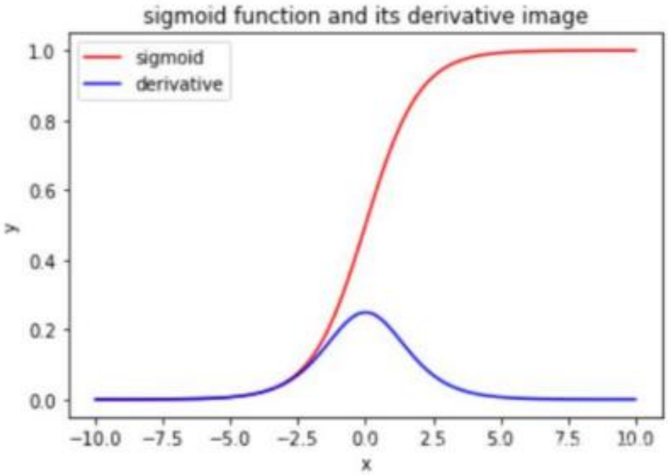
		질병	
나이 그룹	그룹내 수	질병보유자 수	%
20 - 29	5	0	0
30 - 39	6	1	17
40 - 49	7	2	29
50 - 59	7	4	57
60 - 69	5	4	80
70 - 79	2	2	100
80 - 89	1	1	100



출처: 유튜브[핵심 머신러닝] 로지스틱회귀모델 1 (로지스틱함수, 승산) https://youtu.be/l_8XEj2_9rk

이러한 형태를 띠는 함수 중, sigmoid function 은 그 first derivative 가 bell shaped 이고, first derivative 가 sigmoid 자기 자신을 포함하는 식으로 나타나기 때문에 많이 사용된다.





출처: 강의자료

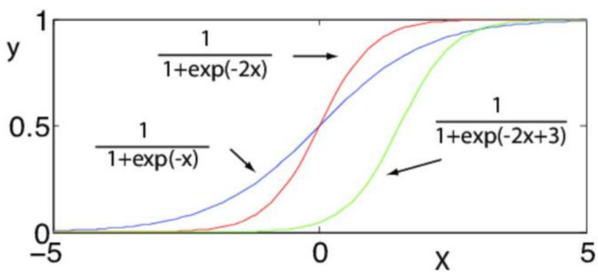
정리하면, sigmoid 함수는 binary classification 에서 feature 에 따라 label 에 속할 확률을 설명할 수 있는 모델이고, 따라서 이 모델을 그 확률에 맞게 조정하면 label 이 구분되는 decision boundary 를 정해 예측할 수 있다. 이때, sigmoid 는

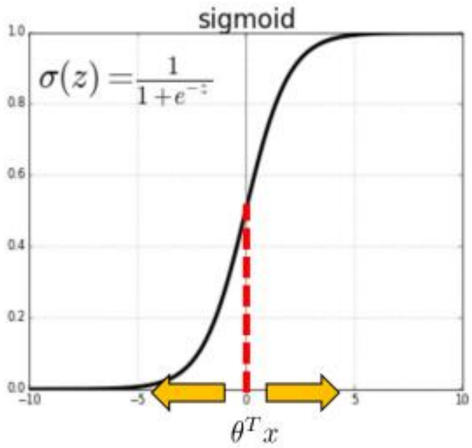
$$f(x) = \frac{L}{1 + e^{-k(x-x_0)}}$$

꼴로 표현되는 Logistic function 의 특이 케이스이기 때문에 우리는 이러한 방식을 Logistic Regression 이라고 부른다. 실제 logistic regression 에서는 sigmoid(x)의 x 에 $\theta^T x$ 를 사용하여

$$h_\theta(x) = \frac{L}{1 + e^{-\theta^T x}}$$

라는 모델을 만들고, 이를 통해 linear decision boundary 를 찾는다.





Cost Function

Linear regression 에서 수행했던 과정과 마찬가지로, 우리는 decision boundary 를 찾기 위해서, cost function ($J(\theta)$에 대해 $\min_{\theta} J(\theta)$를 통해 θ를 학습해야 한다. 이를 위해 linear regression 에서 사용한 MSE 는 logistic regression 에 적합하지 않다. Logistic regression 에서는 Cross Entropy 를 cost function 으로 사용한다.

Cross Entropy 는

$$Cross - Entropy(h_\theta(x), y) = \begin{cases} -\log\big(h_\theta(x)\big) & if\ y = 1 \\ -\log\big(1 - h_\theta(x)\big) & if\ y = 0 \end{cases} = -y\log\big(h_\theta(x)\big) - (1-y)\log\big(1 - h_\theta(x)\big)$$

이다.

따라서,

$$J(\theta) = -\frac{1}{m}[\sum_{i=1}^{m} y^{(i)} \log\left(h_\theta\big(x^{(i)}\big)\right) + (1 - y^{(i)}) \log\left(1 - h_\theta\big(x^{(i)}\big)\right)$$

로 나타낼 수 있다. 이 함수 $J(\theta)$ 에 대해 gradient descent 를 수행하면 $\min_{\theta} J(\theta)$ 를 만족하는 방향으로 θ를 학습할 수 있다.

$$\frac{\partial}{\partial\theta_j} J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\big[h_\theta\big(x^{(i)}\big) - y^{(i)}\big]x_j^{(i)}$$

이므로 최종적으로 우리가 향할 방향은 아래와 같다.

$$repeat\ until\ convergence\ \{$$

$$\theta_j := \theta_j - \alpha\frac{\partial}{\partial\theta_j}J(\theta) = \theta_j - \frac{\alpha}{m}\sum_{i=1}^{m}\big[h_\theta\big(x^{(i)}\big) - y^{(i)}\big]x_j^{(i)}$$

$$\}$$

본론

행렬 곱으로의 전환

Cost function 과 gradient descent 를 여러 행의 데이터에 대해 구하는 과정을 iteration 을 취하거나, 또는 sum 과 multiplication 을 지원하는 함수를 사용할 수도 있다. 그러나, 두 함수에서 모두 multiplication 의 합 형태가 나타나는데, sum of multiplication 은 곧 행렬 곱과의 관계가 있다는 것에서 착안하면 더 수식을 간단하게 만들고, 구현 상의 편의도 챙길 수 있다.

먼저 $J(\theta)$를 살펴보자. $J(\theta)$의 내부를 시그마에 대해 분리하면,

$$J(\theta) = -\frac{1}{m} * [\sum_{i=1}^{m} y^{(i)} \log\left(h_\theta\left(x^{(i)}\right)\right) + \sum_{i=1}^{m}\left(1 - y^{(i)}\right) \log\left(1 - h_\theta\left(x^{(i)}\right)\right)$$

의 꼴로 나타낼 수 있다. 이때 이를 A 와 B 의 행렬 곱이 C 일 때, $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$ 임에서 착안하면, k, j라는 새로운 차원을 도입했을 때,

$$J(\theta) = -\frac{1}{m} * \sum_{i=1}^{m} y_{ki} \left\{\log\left(h_\theta(x)\right)\right\}_{ij} \quad + \sum_{i=1}^{m} (1 - y)_{ki} \left\{\log\left(1 - h_\theta\left(x^{(i)}\right)\right)\right\}_{ij}$$

로 생각할 수 있을 것이다. 입력 데이터의 형태를 고려하여,

$$\boldsymbol{X} = \begin{bmatrix} x_0^{(0)} & \cdots & x_n^{(0)} \\ \vdots & \ddots & \vdots \\ x_0^{(m)} & \cdots & x_n^{(m)} \end{bmatrix} \boldsymbol{Y} = \begin{bmatrix} y^{(1)} \\ \vdots \\ y^{(m)} \end{bmatrix}$$

라면, 결과적으로

$$J(\theta) = -\frac{1}{m}\left\{\boldsymbol{Y}^T \log\left(sigmoid(\boldsymbol{X\theta})\right) + (1 - \boldsymbol{Y})^T \log\left(1 - sigmoid(\boldsymbol{X\theta})\right)\right\}$$

로 나타낼 수 있을 것이다. 물론 이러한 결과는 함수나 scalar 의 broadcasting 을 고려한 rough 한 결과이다.

마찬가지의 방법으로, gradient descent 의 $\min_\theta J(\theta)$ 역시

$$repeat\ until\ convergence\ \{$$

$$\boldsymbol{\theta} := \boldsymbol{\theta} - \frac{\alpha}{m} \boldsymbol{X}^T \{sigmoid(\boldsymbol{X\theta}) - \boldsymbol{Y}\}$$

$$\}$$

로 나타낼 수 있다.

코드 설명

코드 설명은 전체 부분을 다 설명하지 않고, 수식과 연관되는 부분 위주로 설명하도록 하겠다.

먼저 $sigmoid$ 와 $h_\theta(\boldsymbol{X})$ 를 정의하였다.

```
def sigmoid(x):
  return 1 / (1 + np.exp(-x))

def func(theta, X):
  return sigmoid(np.dot(X,theta))
```

이어서 $J(\boldsymbol{\theta})$를 아래와 같이 정의하였다

```
def compute_cost(X, Y, theta):
  m = len(Y)
  h = func(theta, X)
  epsilon = 1e-5
  cost = -(1/m)*(np.dot(Y.T, np.log(h + epsilon)) + np.dot((1-Y).T, np.log(1-h + epsilon)))
  return cost
```

수식과 색깔 박스로 비교하면 아래와 같다.

$$J(\theta) = -\frac{1}{m}\{\boldsymbol{Y}^T \log(sigmoid(\boldsymbol{X\theta})) + (1-\boldsymbol{Y})^T \log(1 - sigmoid(\boldsymbol{X\theta}))\}$$

Gradient descent 의 theta update 부분은 아래와 같이 구현하였다.

```
Y_pred = func(theta, X_batch)

div = 1/m

theta -= alpha * div * np.dot(X_batch.T, Y_pred - Y_batch)
```

이를 수식과 색깔 박스로 비교하면 아래와 같다

$$\boldsymbol{\theta} := \boldsymbol{\theta} - \frac{\alpha}{m}\boldsymbol{X}^T\{sigmoid(\boldsymbol{X\theta}) - \boldsymbol{Y}\}$$

이 외에 추가적으로 논의할 부분은 grid search 이다. $\theta^T x$의 다항식 꼴에서 최대 차수 depth, 그리고 batch size, epoch, alpha 등의 값을 list 로 후보군을 만들고, 그 grid 안에서 최적의 하이퍼 파라미터를 찾도록 반복하여 개별 grid 마다 학습을 진행했다. 이 부분에서 cross validation 을 구현하지 못한 점이 아쉽지만, grid search 의 구현은 성공적이었다. 구현 결과는 아래와 같다

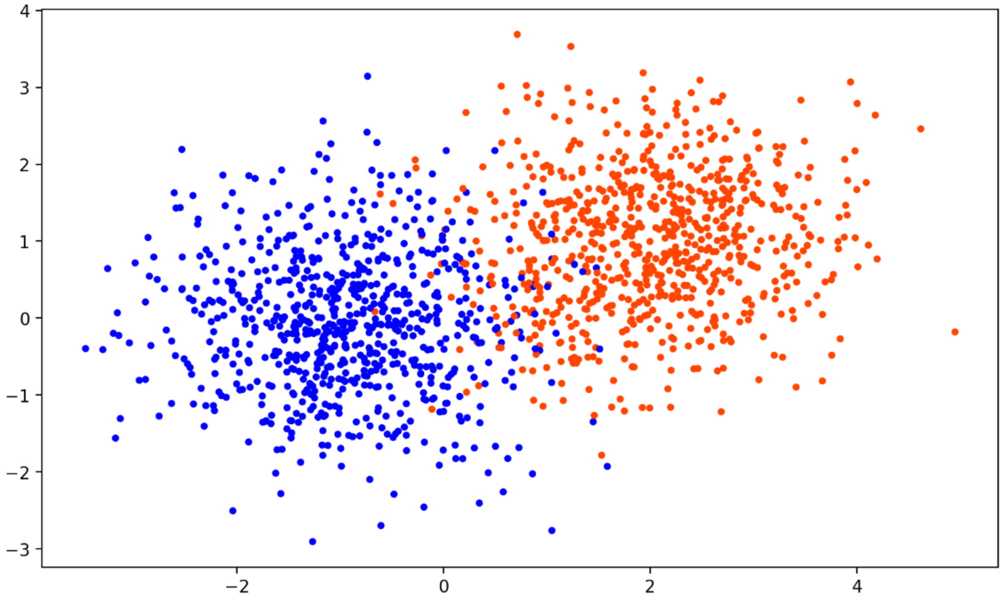
```
for depth in range(1, 6):
  X_0 = np.ones(m)
  X_1 = train_x[:, 0]
  X_2 = train_x[:, 1]
  X_mat = np.c_[X_0, np.c_[X_1, X_2]]

  for i in range(1, depth):
    X_1_exp = X_1**i
    X_2_exp = X_2**i
    X_mat = np.c_[X_mat, np.c_[X_1_exp, X_2_exp]]


  for batch in [0, 1, 50, 200, 500, 1000]:
    for epoch in [1000, 5000, 10000, 15000]:
      for alpha in [0.03, 0.07, 0.1]:
        n = np.size(X_mat,1)
        theta = np.zeros((n,1))
        initial_cost = compute_cost(X_mat, train_y, theta)
        history = np.array((initial_cost))

        (history, theta) = gradient_descent(X_mat, train_y, func, theta, depth, history, batch, alpha, epoch)
        if minimum_cost > history[-1]:
          minimum_cost = history[-1]
          optimal_history = history
          optimal_theta = theta
          optimal_depth = depth
          optimal_batch = batch
          optimal_epoch = epoch
          optimal_alpha = alpha
        cost = history[-1]
        y_pred = predict(theta, X_mat)
        score = float((y_pred == train_y).sum())/ float(len(train_y))
        print(f'depth:{depth}, batch:{batch}, epoch:{epoch}, alpha:{alpha}, cost:{cost}, score:{score}')
```

그러나 grid search 의 구현이 성공적인 것과는 별개로, grid search 를 진행했을 때, depth 나 epoch, batch 등에 따라 minimum cost 와 score 의 차이가 거의 없었다.

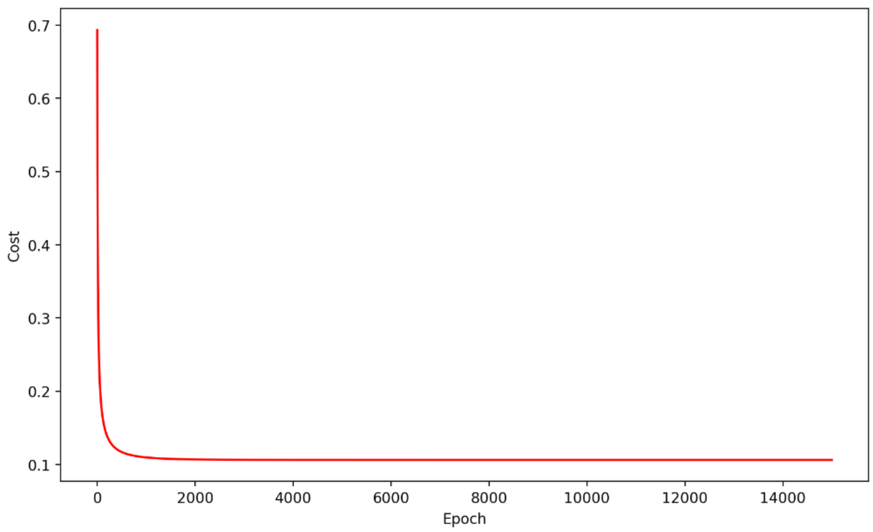
따라서, 최종적으로 depth 1(= $\theta_0 + \theta_1 x_1 + \theta_2 x_2$)에 대해서 반복을 수행했다.

결과 분석

먼저 training dataset 은 아래와 같이 구성되어 있다.



학습 과정에서 loss 는 아래와 같이 감소하였다.

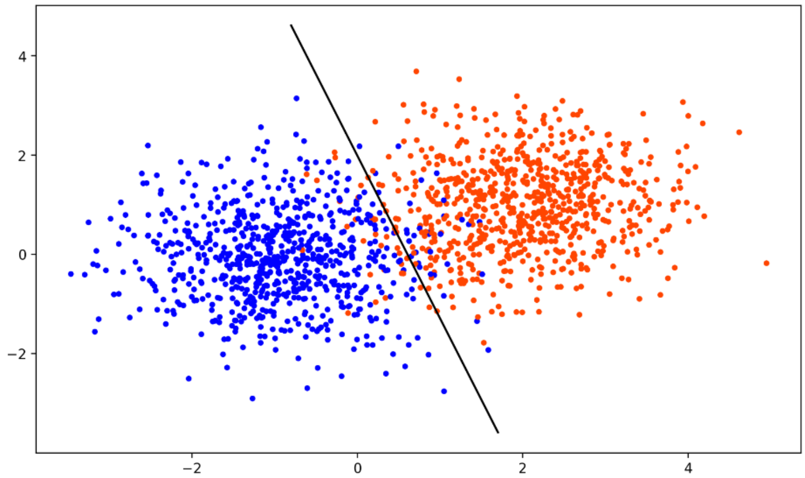


위의 과정을 통해 얻어낸 최종 theta 는 아래와 같다.

optimal depth:1, optimal batch:0, optimal epoch:15000, optimal alpha:0.1

optimal theta:[[2.28917126], [-3.77578216], [-1.15149672]]

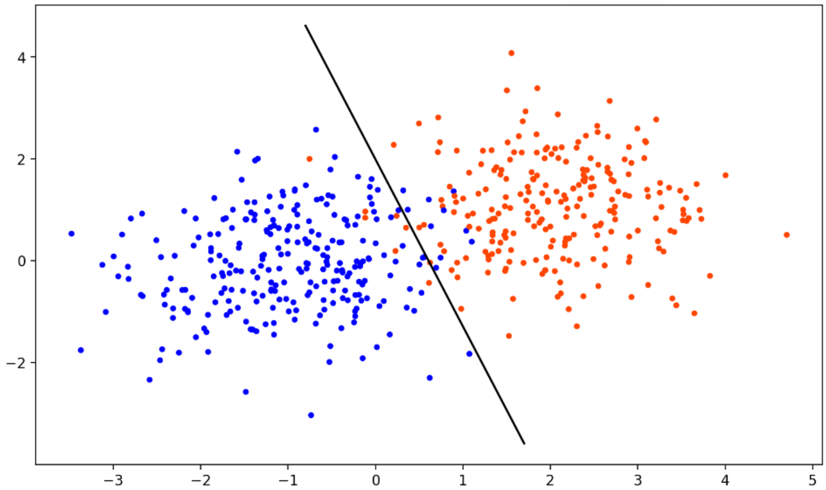
이 theta 에 대해 training data set 의 decision boundary 는 아래와 같이 그려진다



또한 test data 에 대해 prediction 을 수행한 결과는 아래의 캡쳐와 같다. 최종 결과는 마지막에 표로 정리하였다.

```
y_pred = predict(optimal_theta, test_X_mat)
test_score = float((y_pred == test_y).sum())/ float(len(test_y))
test_cost = compute_cost(test_X_mat, test_y, theta)

print(test_score)
print(test_cost)
```

```
0.966
[[0.08810934]]
```



이러한 결과를 통해 알 수 있는 것은, 데이터셋의 label 이 혼재된 영역 때문에 생기는 irreducible error 로 인해 logistic regression 이 충분히 학습 되었음에도 loss 가 irreducible error 에 수렴하여 더 이상 작아지지 않음을 알 수 있다.

결론

최종 결과

A. for $\theta_0 + \theta_1 x_1 + \theta_2 x_2$

	θ_0	θ_1	θ_2
Task1	2.28917126	-3.77578216	-1.15149672

B.

	Cost	Accuracy
Task1 (train)	0.10635371	0.958
Task2 (test)	0.08810934	0.966

실행 방법

HW2.ipynb 를 jupyter notebook 이나 colab 에서 열고 하나씩 셀을 실행하면 됩니다. Numpy, matplotlib 이 설치되어 있어야합니다.