2025-04-03T15-24-31
Tags: DLS

DLS C1 week 2

Binary classification

Logistic regression is an algorithm for binary classification.

Our goal in binary classification is to learn a classifier that can input an image represented by a feature vector x that predicts whether label $y = 1$ (True) or $y = 0$ (False).

Important terms and their corresponding notations in binary classification:

Term	Notation
Single training pair	(x, y) where $x \in \mathbb{R}^{n_x}$, $y \in \{0, 1\}$
Number of examples	m
Number of training examples	m_{train}
Number of test examples	m_{test}
Dimensions	D, n, or n_x
Input feature vector	X
Output vector	Y

The input feature vector X is a matrix stacks training set inputs in columns where there are m columns and n_x rows.

$$X = \begin{bmatrix} \vdots & \vdots & \vdots & \vdots \\ x^{(1)} & x^{(2)} & \dots & x^{(m)} \\ \vdots & \vdots & \vdots & \vdots \end{bmatrix}, \text{ where } X \in \mathbb{R}^{n_x \times m}$$

In NumPy, the shape of X is (n_x, m).

The output labels y is also a matrix that stacks its outputs in columns.

$$Y = [y_1 \quad y_2 \quad \dots \quad y_m], \text{ where } Y \in \mathbb{R}^{1\times m}$$

Stacking the training set inputs and outputs in columns makes implementation easier instead of laying it in rows.

Logistic regression

Given:
$\{(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)})\}, \quad \text{where } x \in \mathbb{R}^{n_x}$

Parameters:
$w \in \mathbb{R}^{n_x}, \quad b \in \mathbb{R}$

Output:
$\hat{y} = \sigma(z) = P(y = 1|x)$

Formula of the sigmoid function:
$$\sigma(z) = \frac{1}{1 + e^{-z}}, \quad \text{where } z = w^\intercal x + b$$

- If $z \to +\infty, \quad \sigma(z) \approx 1$
- If $z \to -\infty, \quad \sigma(z) \approx 0$

Goal:
Learn parameters w and b so that $\hat{y}$ accurately estimates $P(y = 1)$, $\hat{y}^{(i)} \approx y^{(i)}$.

Remarks:

- When programming neural networks, parameters w and b are kept seperate.
- b corresponds to an intercept.

Cost function

Loss function	• Error for a *single* example. • How far $\hat{y}$ is from y.	Cost function	• Average loss over all m training examples. • Dependent on w and b. • Minimized during training.

Normal loss function:
$\mathcal{L}(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$

Logistic regression loss function:
$\mathcal{L}(\hat{y}, y) = -\big(y \log \hat{y} + (1 - y)\log(1 - \hat{y})\big)$

- If $y = 1, \quad \mathcal{L}(\hat{y}, y) = -\log \hat{y}$ where we want $\hat{y} \to 1$
- If $y = 0, \quad \mathcal{L}(\hat{y}, y) = -\log(1 - \hat{y})$ where we want $\hat{y} \to 0$

Logistic cost function:

$$\begin{aligned} J(w,b) &= \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y}^{(i)}, y^{(i)}) \\ &= -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log\hat{y}^{(i)} + (1-y^{(i)})\log(1-\hat{y}^{(i)})\right] \end{aligned}$$

Goal:
Find w and b that make the $J(w,b)$ as small as possible.

$p(y|x) = \hat{y}^y(1-\hat{y})^{(1-y)}$

If $y=1: \quad p(y|x) = \hat{y}$
If $y=0: \quad p(y|x) = 1-\hat{y}$

The $\log$ function is a strictly monotonically increasing function:
$\log p(y|x) = \log \hat{y}^y(1-\hat{y})^{(1-y)}$

Gradient descent

Gradient descent updates parameters in the steepest downhill direction (to minimize cost) until it converges close or to the global optimum.

Gradient descent procedure:

1. Initialize w and b (typically to 0).
2. Iterate by updating parameters to reduce cost:

$$w := w - \alpha \cdot dw, \quad b := b - \alpha \cdot db$$

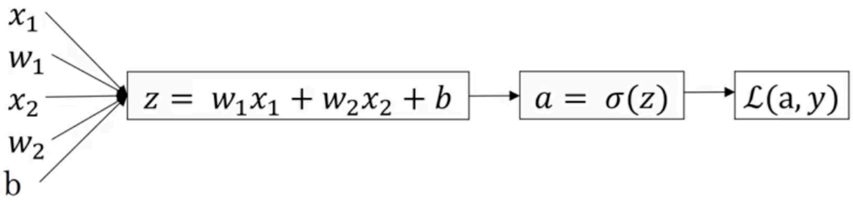
   - α is the learning rate.
   - $dw = \frac{\partial}{\partial w}J(w,b)$ is the slope of a function of w.
   - $db = \frac{\partial}{\partial b}J(w,b)$ is the slope of a function of b.
3. Repeat until it converges close or to the global optima.

Convex function	• Has a single global optimum.	Non-convex function	• May have multiple local and global optima.

Forward and backward pass

Forward pass	• When model takes an input and computes the output. • Input to output. • Computes prediction and loss.	Backward pass	• When model computes gradients. • Output to input using chain rule. • Computes gradients for learning. • Also known as *backpropagation*.

The diagram below displays a forward pass of logistic regression.



Backward pass procedure of the diagram above:

1. $\mathcal{L}(a, y) \to a: \quad da = \dfrac{\partial \mathcal{L}(a, y)}{\partial a} = -\dfrac{y}{a} + \dfrac{1-y}{1-a}$
2. $a \to z: \quad dz = \dfrac{\partial \mathcal{L}(a, y)}{\partial z} = da \cdot \dfrac{\partial a}{\partial z} = a - y$
    - $\dfrac{\partial a}{\partial z} = a(1-a)$
3. $z \to \{w_1, w_2, b\}$
    - $z \to w_1: \quad dw_1 = \dfrac{\partial \mathcal{L}}{\partial w_1} = dz \cdot x_1$
    - $z \to w_2: \quad dw_2 = \dfrac{\partial \mathcal{L}}{\partial w_2} = dz \cdot x_2$
    - $z \to b: \quad db = \dfrac{\partial \mathcal{L}}{\partial b} = dz$

A single step gradient descent with respect to a single example:

1. Compute dz
2. Compute dw_1, dw_2 and db
3. Update w_1, w_2, and b
    - $w_1 := w_1 - \alpha \cdot dw_1$
    - $w_2 := w_2 - \alpha \cdot dw_2$
    - $b := b - \alpha \cdot db$

Gradient descent on many examples

The overall cost function is the average of the individual losses:
$J(w, b) = \dfrac{1}{m} \sum_{i=1}^{m} \mathcal{L}(a^{(i)}, y^{(i)}), \quad \text{where } a^{(i)} = \hat{y}^{(i)} = \sigma(z^{(i)} = \sigma(w^\intercal x^{(i)} + b)$

The derivative of the cost function w.r.t. w_1:

$$\begin{aligned} \frac{\partial}{\partial w_1} J(w, b) &= \frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial w_1} \mathcal{L}(a^{(i)}, y^{(i)}) \\ &= \frac{1}{m} \sum_{i=1}^{m} (a^{(i)} - y^{(i)}) x_1^{(i)} \end{aligned}$$

Gradient descent algorithm on m examples and $n = 2$ features:

1. $J = 0$, $dw_1 = 0$, $dw_2 = 0$, $db = 0$
2. For $i = 1$ to m: (use vectorization instead of for-loop)
    - $z^{(i)} = w^\intercal x^{(i)} + b$

- $a^{(i)} = \sigma(z^{(i)})$
- J += $-\left[y^{(i)} \log a^{(i)} + (1 - y^{(i)}) \log(1 - a^{(i)})\right]$
- $dz^{(i)} = a^{(i)} - y^{(i)}$
- dw_1 += $dz^{(i)} \cdot x_1^{(i)}$
- dw_2 += $dz^{(i)} \cdot x_2^{(i)}$
- db += $dz^{(i)}$

3. J /= m, dw_1 /= m, dw_2 /= m, db /= m

Vectorization

Vectorization is faster than an explicit for-loop.

Imports and initialization:

```
import numpy as np
import time

a = np.array([1, 2, 3, 4])
a = np.random.rand(1000000)
b = np.random.rand(1000000)
```

Speed of vectorization:

```
tic = time.time()
c = np.dot(a, b)
toc = time.time()

print(f'Vectorized version: {1000*(toc-tic)} ms')
```

```
'Vectorized version: 0.5290508270263672 ms'
```

Speed of an explicit for-loop:

```
c = 0
tic = time.time()
for i in range(1000000):
    c += a[i] * b[i]
toc = time.time()

print(f'For loop: {1000*(toc-tic)} ms')
```

```
'For loop: 131.67405128479004 ms'
```

Examples of vectorization:

```
v = np.arange(1, 11)

np.log(v)
np.abs(v)
np.maximum(v, 9)
v**2
1/v
```

Vectorizing logistic regression

$$
\begin{array}{lll}
z^{(1)} = w^\intercal x^{(1)} + b & z^{(2)} = w^\intercal x^{(2)} + b & z^{(3)} = w^\intercal x^{(3)} + b \\
a^{(1)} = \sigma(z^{(1)}) & a^{(2)} = \sigma(z^{(2)}) & a^{(3)} = \sigma(z^{(3)})
\end{array}
$$

Vectorizing $z^{(1)}$, $z^{(2)}$, and $z^{(3)}$ in one step:

$$
\begin{aligned}
Z &= [z^{(1)} \quad z^{(2)} \quad \dots \quad z^{(m)}] \\
&= w^\intercal X + [b \quad b \quad \dots \quad b] \\
&= [w^\intercal x^{(1)} + b \quad w^\intercal x^{(2)} + b \quad \dots \quad w^\intercal x^{(m)} + b]
\end{aligned}
$$

where $X = [x^{(1)} \quad x^{(2)} \quad \dots \quad x^{(m)}]$

Python equivalent of the above:

```
import numpy as np
Z = np.dot(w.T, x) + b
```

Vectorizing $A = \sigma(Z)$:

$$
A = [a^{(1)} \quad a^{(2)} \quad \dots \quad a^{(m)}] = \sigma(Z)
$$

Vectorizing dZ:

$$
Y = [y^{(1)} \quad y^{(2)} \quad \dots \quad y^{(m)}]
$$

$$
\begin{aligned}
dZ &= [dz^{(1)} \quad dz^{(2)} \quad \dots \quad dz^{(m)}] \\
&= [a^{(1)} - y^{(1)} \quad a^{(2)} - y^{(2)} \quad \dots \quad a^{(m)} - y^{(m)}] \\
&= A - Y
\end{aligned}
$$

Vectorizing gradient descent:

```
Z = np.dot(w.T, x) + b
A = σ(Z)
dZ = A - Y
db = (1/m)np.sum(dZ)
dw = (1/m)(X)(dz)^T
w := w - α(dw)
b := b - α(db)
```

Broadcasting

(m, n) with $(1, n) \longrightarrow (m, n)$
(m, n) with $(m, 1) \longrightarrow (m, n)$
$(m, 1)$ with $\mathbb{R} \longrightarrow (m, 1)$
$(1, n)$ with $\mathbb{R} \longrightarrow (1, n)$

Don't use *rank 1* arrays like:

```
a = np.random.randn(5)
a.shape
```

```
(5, )
```

Instead, commit into creating a column or a row vector:

```
a = np.random.randn(5, 1) # column vector
b = np.random.randn(1, 5) # row vector
```

Always call assertions when convenient:

```
assert(a.shape == (5, 1))
```