REPORT

홍익대학교 HONGIK UNIVERSITY

- HW3 Logistic Regression -

iris data & mnist data classification

과목명	:	인공지능
담당교수	:	박 준
학년	:	4학년
학번	:	B411241
이름	:	홍 성 민
제출일	:	20.05.13

1. 과제 개요

과제목표는 **Logistic Regression**을 이용하여 **iris data & mnist data classification**(분류) 문제를 해결해보고 정확도를 측정하여 결과를 내는 것이다.

Input Feature는 iris data는 4개, mnist data는 784개로 지난 과제들과 동일하다.

학습 알고리즘에는 Logistic Regression을 이용하였고 결과 값 상정에 있어 사용한 방법은 **binary classification**의 경우 hypothesis가 0.5가 넘을 경우 1로 상정하였고 **multi classification**의 경우 hypothesis 값이 가장 큰 class로 결과를 상정하였다.

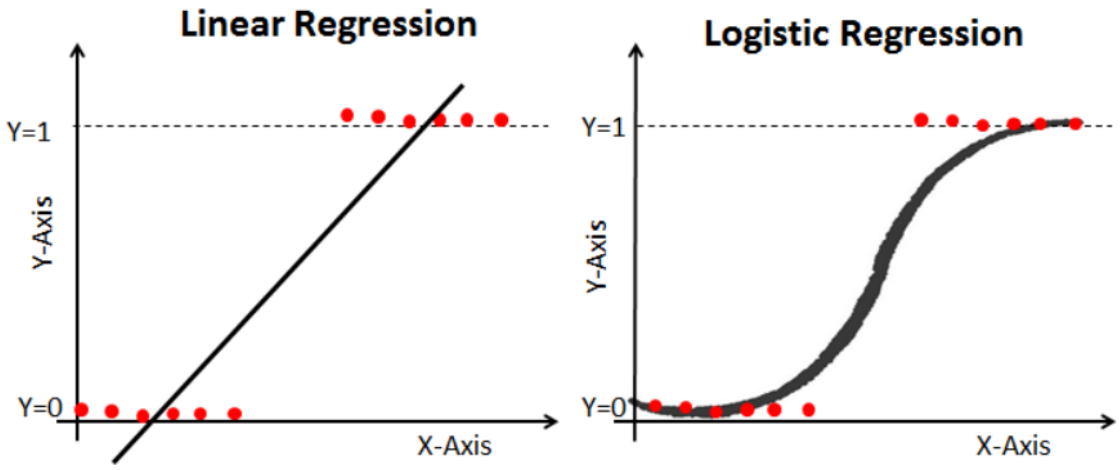
2. 구현 환경

언어 : python3 / tool : pycharm2019.3.4 community / 운영체제 : window10

3. Logistic Regression (Binary & Multi)

3.1 개요

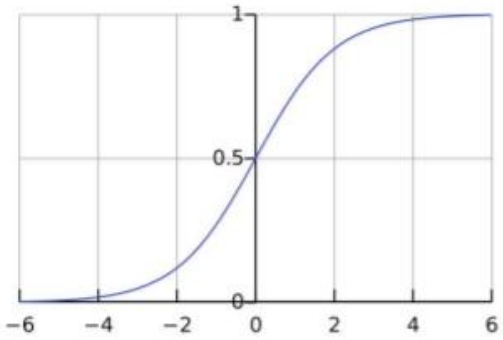
Logistic Regression은 정확도가 높고 Neural Network와 딥 러닝을 이루고 있는 중요한 알고리즘 요소 중 하나이다. Logistic Regression 이전에 다루었던 Linear Regression의 결과는 선형(Linear) 값인데 분류 문제에 이러한 선형회귀를 적용하면 결과 값 자체가 선형 형태를 띄고 있지 않는 이상 오차가 생기게 된다. 특히 binary classification에서는 아래와 같이 문제가 두드러지게 나타난다.



보다시피 결과 값이 class를 의미하는 값일 경우 선형회귀는 이것을 분류하기에 적합하지 않다. 결과 값이 선형이므로 음의 무한에서 양의 무한 값을 가질 수 있으며 분류에 있어 값이 의미하는

바가 명확하지 않게 된다. Logistic Regression은 S자 형태를 띄는 시그모이드(sigmoid) 함수를 사용해 선형에서 벗어났고 0과 1사이의 값을 가지게 되어 분류문제에 있어 보다 정확하고 직관적인 결과를 낼 수 있다. 그렇다면 시그모이드 함수를 어떻게 적용한 것인지 살펴보도록 하자.

3.2 유도 과정

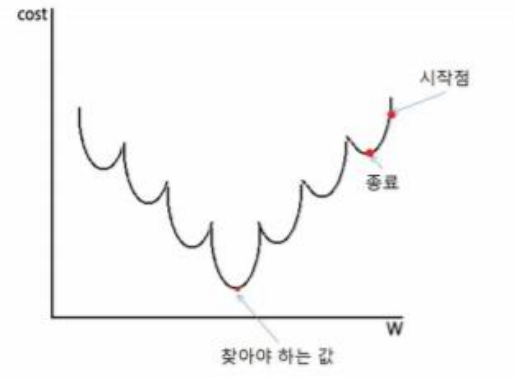


$$f(X) = \frac{1}{1 + e^{-X}}$$

시그모이드 함수의 그래프 모형과 수식은 위와 같으며 함수의 결과값은 0에서 1사이의 값을 가지게 된다. Linear Regression에서 사용한 hypothesis값 즉, H(x)에 sigmoid 함수를 적용한 것을 Logistic Regression에서의 H(x)로 다시 상정한다.

$$cost(W,b) = \frac{1}{m}\sum_{i=1}^{m}(H(x^{(i)}) - y^{(i)})^2 \qquad H(X) = \frac{1}{1 + e^{-W^T X}}$$

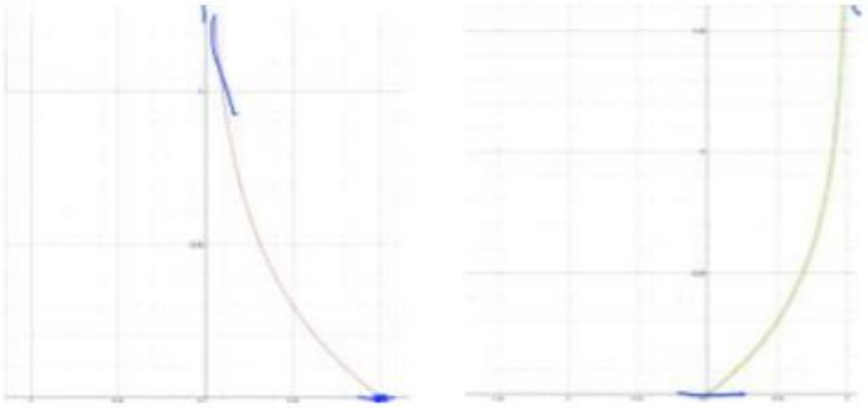
H(x) 값이 바뀜에 따라 Cost 함수도 바뀔 것이다. 바뀐 Cost 함수의 그래프 모형은 아래와 같다.



$$c(H(x),y) = \begin{cases} -log(H(x)) & : y = 1 \\ -log(1 - H(x)) & : y = 0 \end{cases}$$

변화된 Cost 함수를 살펴보면 기울기가 0인 지점이 너무 많이 생기게 되고 우리는 찾아야 하는 Global minimum 값을 찾지 못하고 Local minimum에서 기울기 0을 측정하고 Gradient descent 알고리즘이 결과 값을 내놓게 될 것이다. 그러므로 오른쪽의 식과 같이 Cost 함수의 수정이 필요하다. y에 대한 조건을 따로 나누지 않고 하나의 수식으로 정리하면 다음과 같다.

$$cost(W) = -\frac{1}{m}\sum ylog(H(x)) + (1-y)log(1 - H(x))$$



(좌, y = 1 인 경우 / 우, y = 0 인 경우)

식에 해당하는 Cost 함수로 그래프를 그리게 되면 위와 같은 그림이 나오게 된다. y는 실제 데이터의 정답 즉, 0이냐 1이냐를 말하는 class에 대한 값을 의미한다. 이제 Cost 함수는 Linear Regression에서 사용하던 2차 함수와 비슷한 모양을 띄게 되었고 Cost 함수에 Gradient descent 알고리즘을 적용할 수 있게 된다.

$$W := W - \alpha \frac{\partial}{\partial W} cost(W) \qquad \theta_j := \theta_j - \alpha \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

Gradient descent 알고리즘은 기존과 동일하며 왼쪽 식과 같다. 여기에 새롭게 구한 Cost 함수를 넣게 되면 오른쪽과 같이 hypothesis와 관련된 간단한 식이 나온다. 여기에 있는 Cost 함수 미분 과정에 대해서는 생략하도록 하겠다.

3.3 Binary & Multi Classification

분류(Classification)에는 0또는 1로 분류하는 Binary Classification와 0, 1, 2, 3... 처럼 여러 개의 집단으로 분류하는 Multi Classification이 있다. 각각의 분류 방법에 대해 보도록 하자.

(1) Binary Classification

Binary Classification에서는 단순히 hypothesis 값으로 결과를 판단할 수 있다. 시그모이드 함수가 0~1 사이 값을 가지므로 hypothesis가 0.5 이상인 경우 y=1로 hypothesis가 0.5 미만인 경우 y=0으로 결과를 추정하는 것이 방법이다.

(2) Multi Classification

Multi Classification은 여러 개의 클래스에 대해 결과값을 내야한다. 하지만 class를 의미하는 값

은 단순히 정수로 사용할 수 없다. Iris data를 예를 들면 Iris data에는 총 3개의 class가 존재하고 해당 class의 답을 의미하는 iris.target 은 [0, 1, 2] 과 같은 값으로 구성되어 있다. 이렇게 되면 0과 1의 차이와 0과 2의 차이는 다른 값을 계산하게 되지만 이것은 분류에 있어서 의미가 없는 값이며 오히려 Cost 함수에 적용할 수 없는 값이 되어버린다. 이것을 해결하기 위해서는 class 값에 대해 **one-hot encoding**이 필요하다.

one-hot encoding은 모든 데이터 값을 0 또는 1로 만들어 주는 것을 의미하며 iris.target 과 같이 [0, 1, 2] 를 [[1, 0, 0] , [0 1 0] , [0 0 1]] 처럼 3개의 feature로 늘려주고 정답에 해당하는 class는 1로 나머지 class는 0으로 바꾸어 주는 과정을 의미한다. 결과론적으로 이것은 각각의 class에 대해 binary classification을 적용하고 합친 것과 동일한 배열이 된다. 이렇게 되면 각 class에 대한 hypothesis값이 존재하게 되는데 여기서 가장 큰 hypothesis를 가지고 있는 class로 결과를 상정하는 것이 multi classification에서의 분류 방법이다.

4. 데이터에 대한 설명

데이터는 HW1에서의 Iris data, HW2에서의 mnist data와 동일하다. 하지만 이번 Logistic Regression에서는 knn과 다르게 bias값이 추가되어 각각의 data에 대해 feature dimension이 1씩 증가하였다. Iris data에서 train data set과 test data set의 크기는 각각 120, 30으로 8 : 2 비율로 나누었으며 mnist data는 주어진 train data set 60000개, test data set 10000개를 모두 사용했다.

5. 소스코드 설명

5.1 Logistic class (logclass.py) 소스코드 및 구현설명

<소스코드>

```python
import numpy as np
import matplotlib.pyplot as plt
class LOGISTIC():
    def __init__(self, x_input, y_output, weight, x_test, y_test, labelname, learning_rate = 0.01, cyclelen = 20):
        self.x_input = x_input              # train data set - feature
        self.y_output = y_output            # train data set - answer
        self.weight = weight                # 학습과정에서 계속 수정되는 가중치
        self.x_test = x_test                # test data set - feature
        self.y_test = y_test                # test data set - answer
        self.learning_rate = learning_rate  # learning rate
        self.hx = self.sigmoid(np.dot(self.x_input, self.weight.transpose()))   # hypothesis 값 초기화
        self.costrr = []                    # cost 값을 가지고 있는 리스트
        self.dim = x_input[0].__len__()     # feature 갯수 (dimension)
        self.xsize = len(x_input)           # train data set size
        self.cyclelen = cyclelen            # cost 출력 빈도 (단위)
        self.epoch = 0                      # epoch 값 (현재 학습 횟수)
        self.grap_costrr = []               # 그래프 출력을 위해 cost 값을 저장하는 리스트
        self.grap_epoch = []                # 그래프 출력을 위한 epoch 값을 저장하는 리스트
        self.grap_legend = labelname        # multi classification 에서 각 그래프에 이름을 표시하기 위한 legend 리스트
        if y_test.ndim == 1:                # binary classification 이라면
            self.grap_title = 'binary classification : ' + labelname     # 그래프 제목은 binary ... labelname
        else:                               # multi classification 이라면
            self.grap_title = 'multi classification'                     # 그래프 제목은 multi classification

    def sigmoid(self, x):                   # 시그모이드 함수
        eMin = -np.log(np.finfo(type(0.1)).max) # -> -709.78 ...
        xsafe = np.array(np.maximum(x, eMin))   # exp 값 즉, e^x 가 너무 커지는 것을 방지하기 위해 최대값 상정
        return 1 / (1+np.exp(-xsafe))           # 시그모이드 함수 공식 적용후 return

    def cost(self):                         # cost 함수
        # J(θ) = -1/m[Σy(i)log hθ(x(i)) + (1-y(i))log(1-hθ(x(i))] 에 해당하는 코드
        self.costrr = -1 * sum(self.y_output * np.log(self.hx) + (1-self.y_output) * np.log(1-self.hx)) / self.xsize
        self.grap_costrr.append(self.costrr) # 계산한 cost 값은 그래프 출력을 위해 저장

    def learn(self):                        # 학습 1회에 해당하는 함수
        self.grap_epoch.append(self.epoch)  # 그래프 출력을 위해 epoch 저장
        if self.epoch % self.cyclelen == 0: # 출력 단위가 되었으면
            print("epoch :", self.epoch, "cost :", self.costrr) # epoch 와 cost 를 출력한다.
        self.hx = self.sigmoid(np.dot(self.x_input, self.weight.transpose())) # modify hypothesis
        dif = self.hx - self.y_output                                          # hθ(x(i)) - y(i)
        gradient = np.dot(self.x_input.transpose(), dif) / self.xsize          # Σ(hθ(x(i)) - y(i))xj(i)
        self.weight = self.weight - self.learning_rate * gradient.transpose() # θj = θj - Σ(hθ(x(i)) - y(i))xj(i)
        self.epoch += 1 # 학습 횟수 1 증가

    def predict(self):              # 완성된 학습으로 test data set에 대해 결과를 예측하는 함수
        accuracy = 0.               # 정확도
        if  self.y_test.ndim == 1: # binary classification 경우
            hypo = self.sigmoid(np.dot(self.x_test, self.weight.transpose()))  # test data로 hypothesis 값 계산
            for i in range (0, len(self.x_test)):           # for 문 test data 크기 만큼
                if hypo[i] > 0.5 and self.y_test[i] == 1:   # hypothesis가 0.5 이상이면 결과 1로 상정, 답을 맞췄을 경우
                    accuracy += (1 / len(self.x_test))      # 정확도 += 1 / test data 크기
                if hypo[i] <= 0.5 and self.y_test[i] == 0:  # hypothesis가 0.5 이하이면 결과 0로 상정, 답을 맞췄을 경우
                    accuracy += (1 / len(self.x_test))      # 정확도 += 1 / test data 크기
        else:                       # multi classification 경우
            hypo = self.sigmoid(np.dot(self.x_test, self.weight.transpose()))  # test data로 hypothesis 값 계산
            for i in range (0, len(self.x_test)):           # for 문 test data 크기 만큼
                if np.argmax(hypo[i]) == np.argmax(self.y_test[i]): # class 중 hypothesis 값이 가장큰 것을 결과로 상정
                    accuracy += (1 / len(self.x_test))      # argmax로 값이 가장큰 인덱스 추출, 같으면 답을 맞췄을 경우이다
        print("Accuracy :", round(accuracy, 3)) # 계산한 정확도 소숫점 3자리까지 출력

    def printgrap(self):    # 결과에 대한 그래프를 출력하는 함수
        plt.plot(self.grap_epoch, self.grap_costrr) # x축 epoch, y축 cost값
        plt.ylim(0.005, 2.5)          # 보다 직관적인 결과 관찰을 위해 y축 확대
        plt.xlabel('epoch')           # xlabel : epoch
        plt.ylabel('cost')            # ylabel : cost
        plt.title(self.grap_title)    # 그래프 제목 설정
        plt.legend(self.grap_legend)  # 그래프 각각에 대해 legend 표시
        plt.show()                    # 출력
```

<구현설명>

1. 구현의 방향성

우선 모든 함수와 변수는 binary classification, multi classification과 상관없이 돌아가는 것을 상정하여 구현하였으며 np.array와 np.dot을 사용하여 행렬 곱셈을 계산하였다. 결과값을 상정하거나 그래프 출력 제목에 있어서는 다른 논리가 적용되지만 다른 함수에 대해서는 어떤 식으로 데이터가 주어져도 결과가 나오도록 구현하였다.

2. Class 변수

```
def __init__(self, x_input, y_output, weight, x_test, y_test, labelname, learning_rate = 0.01, cyclelen = 20)
    self.x_input = x_input              # train data set - feature
    self.y_output = y_output            # train data set - answer
    self.weight = weight                # 학습과정에서 계속 수정되는 가중치
    self.x_test = x_test                # test data set - feature
    self.y_test = y_test                # test data set - answer
    self.learning_rate = learning_rate  # learning rate
    self.cyclelen = cyclelen            # cost 출력 빈도 (단위)
```

x_input 은 train data set feature 에 대한 데이터, y_output 은 train data set answer 에 대한 데이터, weight 는 J(θ) 가중치 값, x_test 는 test data set feature 에 대한 데이터, y_test 는 test data set answer 에 대한 데이터, labelname 은 target name 에 대한 정보, learning_rate 는 gradient descent 알고리즘에서의 가중치, cyclelen 은 학습과정을 출력하는 주기에 대한 값이다.

```
self.hx = self.sigmoid(np.dot(self.x_input, self.weight.transpose()))   # hypothesis 값 초기화
self.costrr = []                    # cost 값을 가지고 있는 리스트
self.dim = x_input[0].__len__()     # feature 갯수 (dimension)
self.xsize = len(x_input)           # train data set size
self.epoch = 0                      # epoch 값 (현재 학습 횟수)
self.grap_costrr = []               # 그래프 출력을 위해 cost 값을 저장하는 리스트
self.grap_epoch = []                # 그래프 출력을 위한 epoch 값을 저장하는 리스트
self.grap_legend = labelname        # multi classification 에서 각 그래프에 이름을 표시하기 위한 legend 리스트
if y_test.ndim == 1:                # binary classification 이라면
    self.grap_title = 'binary classification : ' + labelname     # 그래프 제목은 binary ... labelname
else:                               # multi classification 이라면
    self.grap_title = 'multi classification'                     # 그래프 제목은 multi classification
```

hx 는 hypothesis 를 담고 있는 리스트를 의미한다. hypothesis 는 형식을 맞추어 주기 위해 우선 가지고 있는 weight 값으로 sigmoid 계산을 하여 초기화 하였다. costrr 는 cost 값을 가지고 있는 리스트이며 사실 multi classification 에서 기능을 위해 리스트 형식으로 존재하는 것이다. dim 은 data feature 개수(차원), xsize 는 train data set 크기이다. 코드 구현상 일일이 계산하기 번거롭기 때문에 편리함을 위해 존재하는 변수이다. epoch 는 학습횟수를 저장하고 있는 변수, grap_costrr 는 그래프 출력을 위해 cost 값을 계산할 때 마다 저장하는 리스트, grap_epoch 도 grap_costrr 과 마찬가지로 그래프 출력을 위해 epoch 값을 저장하는 리스트이다. grap_legend 는 multi classification 에서 그래프를 출력하게 되면 어떤 그래프가 어떤 class 에 해당하는지 알 수 없기 때문에 해당 그래프에 legend 로 이름을 명시하기 위한 target name 리스트이다. grap_title 의 경우

if 문을 통해 데이터의 차원이 1 차원일 때는 binary classification+ labelname 으로 설정하며 이외의 경우는 multi classification 으로 설정했다.

3. Sigmoid function

```
def sigmoid(self, x):                    # 시그모이드 함수
    eMin = -np.log(np.finfo(type(0.1)).max) # -> -709.78 ...
    xsafe = np.array(np.maximum(x, eMin))   # exp 값 즉, e^x 가 너무 커지는 것을 방지하기 위해 최대값 상정
    return 1 / (1+np.exp(-xsafe))           # 시그모이드 함수 공식 적용후 return
```

시그모이드 함수를 의미하는 sigmoid 는 과제설명에 있는 sigmoid 함수와 동일하게 구현하였으며 exp(x)값이 너무 커져 overflow 가 발생하는 것을 방지하기 위해 xsafe 값을 설정하였다.

4. Cost function

```
def cost(self):                          # cost 함수
    # J(θ) = -1/m[Σy(i)log hθ(x(i)) + (1-y(i))log(1-hθ(x(i))] 에 해당하는 코드
    self.costrr = -1 * sum(self.y_output * np.log(self.hx) + (1-self.y_output) * np.log(1-self.hx)) / self.xsize
    self.grap_costrr.append(self.costrr) # 계산한 cost 값은 그래프 출력을 위해 저장
```

loss 함수를 의미하는 cost 함수식은 J(θ) = -1/m[Σy(i)log hθ(x(i)) + (1-y(i))log(1-hθ(x(i))] 과 같으며 해당 식을 구현하면 위와 같다. cost 값을 계산하면 grap_costrr 에 append 하여 저장했다.

5. Learn function

```
def learn(self):                         # 학습 1회에 해당하는 함수
    self.grap_epoch.append(self.epoch)  # 그래프 출력을 위해 epoch 저장
    if self.epoch % self.cyclelen == 0: # 출력 단위가 되었으면
        print("epoch :", self.epoch, "cost :", self.costrr) # epoch 와 cost 를 출력한다.
    self.hx = self.sigmoid(np.dot(self.x_input, self.weight.transpose())) # modify hypothesis
    dif = self.hx - self.y_output                                          # hθ(x(i)) - y(i)
    gradient = np.dot(self.x_input.transpose(), dif) / self.xsize          # Σ(hθ(x(i)) - y(i))xj(i)
    self.weight = self.weight - self.learning_rate * gradient.transpose() # θj = θj - Σ(hθ(x(i)) - y(i))xj(i)
    self.epoch += 1 # 학습 횟수 1 증가
```

Learn 은 학습 1 회에 해당하는 함수이며 학습과정 출력은 출력이 너무 많게 되는 것을 방지하기 위해 epoch 가 cyclelen 으로 나누어 떨어질 때만 출력되도록 설정하였다. learn 에서는 실질적으로 hypothesis 값을 weight 로 계산하고 계산한 hypothesis 를 이용해 gradient descent 알고리즘을 적용하여 weight 값을 수정하게 된다. hypothesis 를 계산하고 gradient 값을 계산하는 것은 np.array 의 np.dot 을 이용해 행렬 곱셈 계산을 하였으며 np.dot 을 위해서 weight 와 gradient 를 transpose 하여 곱하였다.

6. Predict function

```
def predict(self):              # 완성된 학습으로 test data set에 대해 결과를 예측하는 함수
    accuracy = 0.               # 정확도
    if  self.y_test.ndim == 1: # binary classification 경우
        hypo = self.sigmoid(np.dot(self.x_test, self.weight.transpose()))  # test data로 hypothesis 값 계산
        for i in range (0, len(self.x_test)):           # for 문 test data 크기 만큼
            if hypo[i] > 0.5 and self.y_test[i] == 1:   # hypothesis가 0.5 이상이면 결과 1로 상정, 답을 맞췄을 경우
                accuracy += (1 / len(self.x_test))      # 정확도 += 1 / test data 크기
            if hypo[i] <= 0.5 and self.y_test[i] == 0:  # hypothesis가 0.5 이하이면 결과 0로 상정, 답을 맞췄을 경우
                accuracy += (1 / len(self.x_test))      # 정확도 += 1 / test data 크기
    else:                       # multi classification 경우
        hypo = self.sigmoid(np.dot(self.x_test, self.weight.transpose()))  # test data로 hypothesis 값 계산
        for i in range (0, len(self.x_test)):           # for 문 test data 크기 만큼
            if np.argmax(hypo[i]) == np.argmax(self.y_test[i]): # class 중 hypothesis 값이 가장큰 것을 결과로 상정
                accuracy += (1 / len(self.x_test))      # argmax로 값이 가장큰 인덱스 추출, 같으면 답을 맞췄을 경우이다
    print("Accuracy :", round(accuracy, 3)) # 계산한 정확도 소숫점 3자리까지 출력
```

Predict 함수는 학습이 끝난 weight 값을 가지고 test data 에 대한 hypothesis 를 계산해 결과를 예측하는 과정을 의미한다. predict 는 if 문에 test data set 의 차원으로 binary 와 multi classification 두가지로 나누어 결과를 보여준다.

binary classification 은 hypothesis 값이 0.5 가 넘는 경우에 대해서 결과를 1 로 0.5 가 이하인 경우 결과를 0 으로 책정했고 test data set answer 과 같은 경우에는 정확도를 (1 / test data 크기) 만큼 증가 시켜주었다.

multi classification 은 class 만큼의 hypothesis 값이 나오게 되고 가장 큰 값의 index 를 argmax 로 뽑았고 test data set answer 에 있는 1 값도 0 보다 크므로 argmax 를 통해 index 를 추출하였다. 해당 index 가 같은 경우에 답을 맞춘 것이고 이때 정확도를 (1 / test data 크기) 만큼 증가 시켜주었다.

최종적으로 정확도를 출력하고 Predict 함수는 종료된다.

7. Printgrap function

```
def printgrap(self):     # 결과에 대한 그래프를 출력하는 함수
    plt.plot(self.grap_epoch, self.grap_costrr) # x축 epoch, y축 cost값
    plt.ylim(0.005, 2.5)          # 보다 직관적인 결과 관찰을 위해 y축 확대
    plt.xlabel('epoch')           # xlabel : epoch
    plt.ylabel('cost')            # ylabel : cost
    plt.title(self.grap_title)    # 그래프 제목 설정
    plt.legend(self.grap_legend)  # 그래프 각각에 대해 legend 표시
    plt.show()                    # 출력
```

printgrap 는 학습 횟수에 따른 cost 값의 변화를 보여주는 그래프를 출력하는 함수이다. x 축에는 grap_epcoh 를 넣어 epoch 를 넣었고 y 축에는 grap_costrr 를 넣어 cost 값을 넣어주었다. ylim 은 그래프가 지나치게 L 자 모양으로 표현되는 것을 피하기 위해 특정 y 의 범위를 확대했고 각각의

축에는 label 명을 달아주었다. title 의 경우 init 함수에서 설정한 제목으로 되어있으며 legend 는 multi classification 그래프에서 그래프가 어느 class 에 해당하는 것인지 표시하기위해 사용된다.

5.2 Iris main (iris_main.py) 소스코드 및 구현설명

<소스코드>

```python
import numpy as np
from sklearn.datasets import load_iris
import random
from logclass import LOGISTIC    # LOGISTICclass import

iris = load_iris()
X = iris.data[:, :4]         # feature 4종류.
y = iris.target              # 0, 1, 2 로 구성되어있는 데이터
y_name = iris.target_names   # ['setosa' 'versicolor' 'virginica'] 꽃 이름

bias = []                    # bias를 추가하기 위한 리스트
for i in range(0, len(X)):   # data set 크기 만큼 1을 담고 있는 bias 리스트 생성
    bias.append([1])
bias = np.array(bias)        # np.array bias : shape (150, 1)
X = np.array(X)              # np.array X : shape (150, 4)
X = np.hstack((bias, X))     # column으로 np.array를 합침 X : shape (150, 5)

# one-hot encoding
num = np.unique(y, axis=0)  # [0 1 2]
num = num.shape[0]          # 3
y = np.eye(num)[y]          # y = [[1. 0. 0.] [1. 0. 0.] ... [0. 0. 1.]]

l = 5                       # train data set size : test data set size = 8 : 2
# y.shape[0] = 150 이며 i % l == 5 일때를 test data set 으로 설정
for_test = np.array([(i % l == (l-1)) for i in range(y.shape[0])]) # for_test 는 boolean 값을 담는 리스트가 된다.
for_train = ~for_test       # for_train 은 for_test 의 반대 boolean 값을 담는 리스트가 된다.
X_train = X[for_train]      # iris.data X에서 for_train 의 true 값만을 담는 리스트가 된다. 크기 120 train data set 의 feature
y_train = y[for_train]      # iris.target y에서 for_train 의 true 값만을 담는 리스트가 된다. 크기 120 train data set 의 답
X_test = X[for_test]        # iris.data X에서 for_test 의 true 값만을 담는 리스트가 된다. 크기 30 test data set 의 feature
y_test = y[for_test]        # iris.target y에서 for_test 의 true 값만을 담는 리스트가 된다. 크기 30 test data set 의 답

weight = []    # multi classification weight 를 담고 있는 2차원 배열
for i in range(0, len(y[0])):    # class 종류 수 만큼 for 문
    w_n = []                     # class i번째에 대한 weight 리스트
    for j in range(0, len(X[0])):   # data feature 수 만큼 for 문
        w_n.append(random.random()) # 초기 weight 값은 0~1 사이의 random 값으로 초기화
    weight.append(w_n)           # class i번째에 대한 weight 리스트 append
weight = np.array(weight)        # np.array weight : shape (3, 5)

epoch = 10  # 학습 횟수
# binary classification
for i in range(0, weight.shape[0]): # class 종류 수 만큼 for 문
    single_iris = LOGISTIC(X_train, y_train[:, i], weight[i, :], X_test, y_test[:, i], y_name[i]) # i 번째 single class
    # (train data feature, train data answer[i], weight[i], test data feature, test data answer[i], answer_name[i])
    for j in range(0, epoch):   # 학습 횟수 만큼 for 문
        single_iris.cost()      # cost 함수 계산
        single_iris.learn()     # 학습 (modify weight가 일어난다, gradient descent 과정)
    single_iris.predict()       # 완성된 weight로 test data 결과 예측, 정확도 출력
    single_iris.printgrap()     # 그래프 출력 함수 실행
# multi classification
multi_iris = LOGISTIC(X_train, y_train, weight, X_test, y_test, y_name)  # multi class
for i in range(0, epoch):       # 학습 횟수 만큼 for 문
    multi_iris.cost()           # cost 함수 계산
    multi_iris.learn()          # 학습 (modify weight가 일어난다, gradient descent 과정)
multi_iris.predict()            # 완성된 weight로 test data 결과 예측, 정확도 출력
multi_iris.printgrap()          # 그래프 출력 함수 실행
```

<구현설명>

1. data feature engineering

```
11      bias = []                   # bias를 추가하기 위한 리스트
12      for i in range(0, len(X)):  # data set 크기 만큼 1을 담고 있는 bias 리스트 생성
13          bias.append([1])
14      bias = np.array(bias)       # np.array bias : shape (150, 1)
15      X = np.array(X)             # np.array X : shape (150, 4)
16      X = np.hstack((bias, X))    # column으로 np.array를 합침 X : shape (150, 5)
```

iris data를 load_iris() 를 통해 불러온 후 bias 값을 추가해주기 위해 위와 같이 np.hstack 을 이용해 bias 를 첫번째 열에 추가해주었다. bias는 값이 모두 1인 (150, 1) array 이다.

```
18      # one-hot encoding
19      num = np.unique(y, axis=0)  # [0 1 2]
20      num = num.shape[0]          # 3
21      y = np.eye(num)[y]          # y = [[1. 0. 0.] [1. 0. 0.] ... [0. 0. 1.]]
```

앞에서 언급한 multi classification에서 분류를 위해 결과값에 대해 one-hot encoding을 적용했다. unique를 통해 어떤 결과값 종류를 가지고 있는지 보고, 해당 종류만큼 차원을 증가시켜 다차원 배열로 변경시켰다. 기존 y의 shape은 (150, 1) 이였지만 이제 y는 (150, 3,)으로 변경되었다.

```
32      weight = []    # multi classification weight 를 담고 있는 2차원 배열
33      for i in range(0, len(y[0])):    # class 종류 수 만큼 for 문
34          w_n = []                     # class i번째에 대한 weight 리스트
35          for j in range(0, len(X[0])):   # data feature 수 만큼 for 문
36              w_n.append(random.random()) # 초기 weight 값은 0~1 사이의 random 값으로 초기화
37          weight.append(w_n)           # class i번째에 대한 weight 리스트 append
38      weight = np.array(weight)        # np.array weight : shape (3, 5)
```

weight 설정과정이다. 초기 weight는 random.random을 통해 0~1사이 값을 가지게 했다. weight의 axis=0은 클래스의 수 이다. weight의 axis=1은 데이터 X에 이미 bias를 추가한 상태이므로 len(X[0])와 동일하게 설정하면 된다. 최종적으로 weight를 np.array로 만들었으며 iris data에서 weight의 shape은 (3, 5)가 된다.

2. start learning

```
42      for i in range(0, weight.shape[0]): # class 종류 수 만큼 for 문
43          single_iris = LOGISTIC(X_train, y_train[:, i], weight[i, :], X_test, y_test[:, i], y_name[i]) # i 번째 single class
44          # (train data feature, train data answer[i], weight[i], test data feature, test data answer[i], answer_name[i])
45          for j in range(0, epoch):   # 학습 횟수 만큼 for 문
46              single_iris.cost()      # cost 함수 계산
47              single_iris.learn()     # 학습 (modify weight가 일어난다, gradient descent 과정)
48          single_iris.predict()       # 완성된 weight로 test data 결과 예측, 정확도 출력
49          single_iris.printgrap()     # 그래프 출력 함수 실행
50      # multi classification
51      multi_iris = LOGISTIC(X_train, y_train, weight, X_test, y_test, y_name)  # multi class
52      for i in range(0, epoch):       # 학습 횟수 만큼 for 문
53          multi_iris.cost()           # cost 함수 계산
54          multi_iris.learn()          # 학습 (modify weight가 일어난다, gradient descent 과정)
55      multi_iris.predict()            # 완성된 weight로 test data 결과 예측, 정확도 출력
56      multi_iris.printgrap()          # 그래프 출력 함수 실행
```

마지막으로 실제 학습과정을 for문을 통해 구현하였다. binary classification의 경우 class 수 만큼

for문을 돌게 되며 [:, i]와 같이 인덱싱을 통해 해당 class에 weight 값과 답에 해당하는 y 값을 넣어주는 LOGISTIC 클래스 객체를 생성하였다. 이후 epoch 크기만큼 cost값을 계산하고 learn을 통해 gradient descent 알고리즘을 실행하였다. epoch 수만큼 학습이 끝난 이후에는 predict 함수를 통해 정확도를 출력하였고 printgrap 함수를 통해 epoch 당 cost 값의 변화를 그래프로 출력하였다. multi classification의 경우 LOGISTIC 클래스 객체를 한번 생성하고 모든 data set을 넣어주었고 학습은 동일하게 epoch 수만큼 진행한다.

5.3 mnist main (iris_main.py) 소스코드 및 구현설명

<소스코드>

```python
import sys, os
sys.path.append(os.pardir)
import numpy as np
from dataset.mnist import load_mnist
import random
from PIL import Image
from logclass import LOGISTIC    # LOGISTICclass import
# training data, test data,
(X, y), (x_test, y_test) = load_mnist(flatten=True, normalize=True)
# flattern : 이미지를 1차원 배열로 읽음 , normalize : 0~1 실수로. 그렇지 않으면 0~255
bias = []                   # bias를 추가하기 위한 리스트
for i in range(0, len(X)):  # data set 크기 만큼 1을 담고 있는 bias 리스트 생성
    bias.append([1])
bias = np.array(bias)       # np.array bias : shape (60000, 1)
X = np.array(X)             # np.array X : shape (60000, 784)
X = np.hstack((bias, X))    # column으로 np.array를 합침 X : shape (60000, 785)

num = np.unique(y, axis=0)  # [0 1 2 ... 8 9]
num = num.shape[0]          # 10
y = np.eye(num)[y]          # y = [[0 0 0 ... 0 0 0] [1 ... 0] ... [0 0 0 ... 0 1 0]]

bias = np.array(bias[:len(x_test)]) # np.array bias : shape (10000, 1)
x_test = np.array(x_test)           # np.array x_test : shape (10000, 784)
x_test = np.hstack((bias, x_test))  # column으로 np.array를 합침 x_test : shape (10000, 785)
y_test = np.eye(num)[y_test] # y_test = [[0 0 0 ... 0 0 0] [1 ... 0] ... [0 0 0 ... 0 1 0]]
```

```python
weight = []   # multi classification weight 를 담고 있는 2차원 배열
for i in range(0, len(y[0])): # class 종류 수 만큼 for 문
    w_n = []                  # class i번째에 대한 weight 리스트
    for j in range(0, len(X[0])):         # data feature 수 만큼 for 문
        w_n.append(random.random() / 100) # 초기 weight 값은 0/100 ~ 1/100 사이의 값
    weight.append(w_n)        # class i번째에 대한 weight 리스트 append
weight = np.array(weight)     # np.array weight : shape (10, 785)

epoch = 5000   # 학습 횟수
label_name = ['0','1','2','3','4','5','6','7','8','9'] # 출력을 위한 target name
# single classification
for i in range(0, weight.shape[0]): # class 종류 수 만큼 for 문
    single_mnist = LOGISTIC(X, y[:, i], weight[i, :], x_test, y_test[:, i], label_name[i]) # i 번째 single class
    for j in range(0, epoch):   # 학습 횟수 만큼 for 문
        single_mnist.cost()     # cost 함수 계산
        single_mnist.learn()    # 학습 (modify weight가 일어난다, gradient descent 과정)
    single_mnist.predict()      # 완성된 weight로 test data 결과 예측, 정확도 출력
    single_mnist.printgrap()
# multi classification
multi_mnist = LOGISTIC(X, y, weight, x_test, y_test, label_name) # multi class
for i in range(0, epoch):       # 학습 횟수 만큼 for 문
    multi_mnist.cost()          # cost 함수 계산
    multi_mnist.learn()         # 학습 (modify weight가 일어난다, gradient descent 과정)
multi_mnist.predict()           # 완성된 weight로 test data 결과 예측, 정확도 출력
multi_mnist.printgrap()         # 그래프 출력 함수 실행
```

<구현설명>

1. data feature engineering

```
(X, y), (x_test, y_test) = load_mnist(flatten=True, normalize=True)
# flattern : 이미지를 1차원 배열로 읽음 , normalize : 0~1 실수로. 그렇지 않으면 0~255
bias = []                   # bias를 추가하기 위한 리스트
for i in range(0, len(X)):  # data set 크기 만큼 1을 담고 있는 bias 리스트 생성
    bias.append([1])
bias = np.array(bias)       # np.array bias : shape (60000, 1)
X = np.array(X)             # np.array X : shape (60000, 784)
X = np.hstack((bias, X))    # column으로 np.array를 합침 X : shape (60000, 785)
num = np.unique(y, axis=0)  # [0 1 2 ... 8 9]
num = num.shape[0]          # 10
y = np.eye(num)[y]          # y = [[0 0 0 ... 0 0 0] [1 ... 0] ... [0 0 0 ... 0 1 0]]
bias = np.array(bias[:len(x_test)]) # np.array bias : shape (10000, 1)
x_test = np.array(x_test)           # np.array x_test : shape (10000, 784)
x_test = np.hstack((bias, x_test))  # column으로 np.array를 합침 x_test : shape (10000, 785)
y_test = np.eye(num)[y_test] # y_test = [[0 0 0 ... 0 0 0] [1 ... 0] ... [0 0 0 ... 0 1 0]]
```

iris data와 동일하게 bias값을 데이터에 추가해주고 class 값에 대해 one-hot encoding을 해주는 과정이다. load minist를 통해 얻어온 train data set과 test data set 둘에 대해 똑 같은 과정을 실행해주었다. 최종적으로 데이터 feature dimension은 785가 되었고 y의 shape은 (60000, 10)이 된다.

```
weight = []   # multi classification weight 를 담고 있는 2차원 배열
for i in range(0, len(y[0])): # class 종류 수 만큼 for 문
    w_n = []                  # class i번째에 대한 weight 리스트
    for j in range(0, len(X[0])):         # data feature 수 만큼 for 문
        w_n.append(random.random() / 100) # 초기 weight 값은 0/100 ~ 1/100 사이의 값
    weight.append(w_n)        # class i번째에 대한 weight 리스트 append
weight = np.array(weight)     # np.array weight : shape (10, 785)
```

weight 설정 과정이다. iris와 과정은 동일하나 weight의 초기 값을 0에서 1이 아니라 0/100에서 1/100 값이 나오게 설정했다. 그 이유는 다음과 같다. 초기 weight 값을 0~1로 random하게 설정했을 경우 초기 hypothesis는 785개의 sum (weight(i) * x_input(i)) 값에 시그모이드 함수를 적용한 값이 된다. weigh * x_input의 평균값을 0.1 이라 책정해도 hypothesis는 sigmoid(70) 정도가 되며 이 값은 어떤 데이터 셋에 대해서도 0.99999... 로 모두 1로 나와버리게 된다. 이것을 통해 학습을 진행하면 제대로 된 학습이 진행되지 않는 것을 관찰했으며 초기 hypothesis가 모두 1로 나타나는 것을 방지하기 위해 초기 weight 값을 줄여주게 된 것이다. 일반적인 방법으로는 weight를 feature 수만큼 나누는 것과 동일하다고 볼 수 있겠다.

2. start learning

```
33  epoch = 5000   # 학습 횟수
34  label_name =  ['0','1','2','3','4','5','6','7','8','9'] # 출력을 위한 target name
35  # single classification
36  for i in range(0, weight.shape[0]): # class 종류 수 만큼 for 문
37      single_mnist = LOGISTIC(X, y[:, i], weight[i, :], x_test, y_test[:, i], label_name[i]) # i 번째 single class
38      for j in range(0, epoch):    # 학습 횟수 만큼 for 문
39          single_mnist.cost()      # cost 함수 계산
40          single_mnist.learn()     # 학습 (modify weight가 일어난다, gradient descent 과정)
41      single_mnist.predict()       # 완성된 weight로 test data 결과 예측, 정확도 출력
42      single_mnist.printgrap()
43  # multi classification
44  multi_mnist = LOGISTIC(X, y, weight, x_test, y_test, label_name) # multi class
45  for i in range(0, epoch):        # 학습 횟수 만큼 for 문
46      multi_mnist.cost()           # cost 함수 계산
47      multi_mnist.learn()          # 학습 (modify weight가 일어난다, gradient descent 과정)
48  multi_mnist.predict()            # 완성된 weight로 test data 결과 예측, 정확도 출력
49  multi_mnist.printgrap()          # 그래프 출력 함수 실행
```

학습 과정을 구현한 for문은 iris에서의 구조와 동일하다.

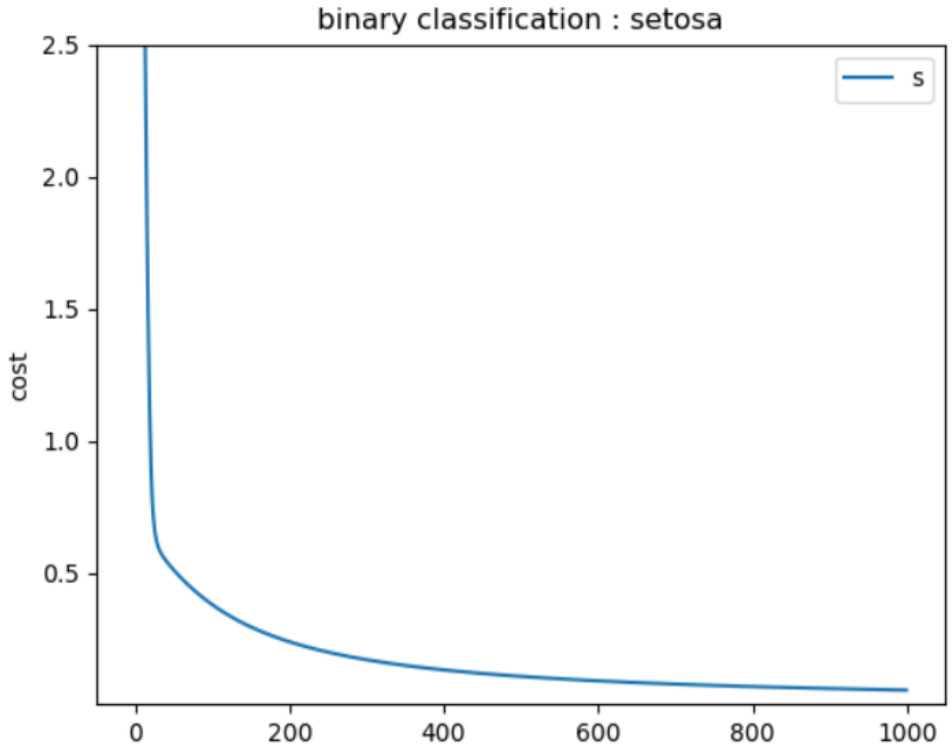
6. (7.) 학습 과정, 학습 결과 및 분석

학습 과정이 제대로 진행되고 있는지 보기 위해 이미 LOGISTIC 클래스에서 cyclelen 주기 마다 cost 값을 출력하고 있기 때문에 해당 cost 값의 추이를 보면서 gradient descent 알고리즘이 적용되어 cost 값이 감소하고 있는지 확인하면 되겠다. cost 값의 추이를 본 뒤 정확도와 그래프를 보고 어떤 상관관계가 있는지, 학습이 잘 진행된 것인지 결과를 보고 분석하도록 하겠다.

6.1 Iris data

- IRIS Single Class / target class : setosa / epoch = 1000, learning rate = 0.01

```
epoch : 0 cost : 6.499227818702109
epoch : 20 cost : 0.9771847817350136
epoch : 40 cost : 0.5482338090459052
epoch : 60 cost : 0.48169126886114405
epoch : 80 cost : 0.42707759540382556
epoch : 100 cost : 0.381538175716668
epoch : 120 cost : 0.34342702846380163
epoch : 140 cost : 0.3113494915473701
epoch : 160 cost : 0.28416261127897174
epoch : 180 cost : 0.26094793089520424
epoch : 200 cost : 0.2409741943128316
epoch : 220 cost : 0.22366079113945833
epoch : 240 cost : 0.20854610654548086
epoch : 260 cost : 0.19526171047653776
epoch : 280 cost : 0.18351195636367928
epoch : 300 cost : 0.17305811313699423
epoch : 320 cost : 0.16370611356330297
epoch : 340 cost : 0.1552971139654946
epoch : 360 cost : 0.1477002109032586
epoch : 640 cost : 0.08778498510896311
epoch : 660 cost : 0.08533643216237875
epoch : 680 cost : 0.08302452182873116
epoch : 700 cost : 0.08083817590025884
epoch : 720 cost : 0.0787674712243959
epoch : 740 cost : 0.07680349452584233
epoch : 760 cost : 0.07493821836133169
epoch : 780 cost : 0.073164394740405
epoch : 800 cost : 0.07147546357382345
epoch : 820 cost : 0.06986547361562787
epoch : 840 cost : 0.0683290139715925
epoch : 860 cost : 0.06686115457625513
epoch : 880 cost : 0.06545739430868086
epoch : 900 cost : 0.06411361563599105
epoch : 920 cost : 0.0628260448531824
epoch : 940 cost : 0.0615912171355182
epoch : 960 cost : 0.06040594574187459
epoch : 980 cost : 0.05926729480867607
Accuracy : 1.0
```

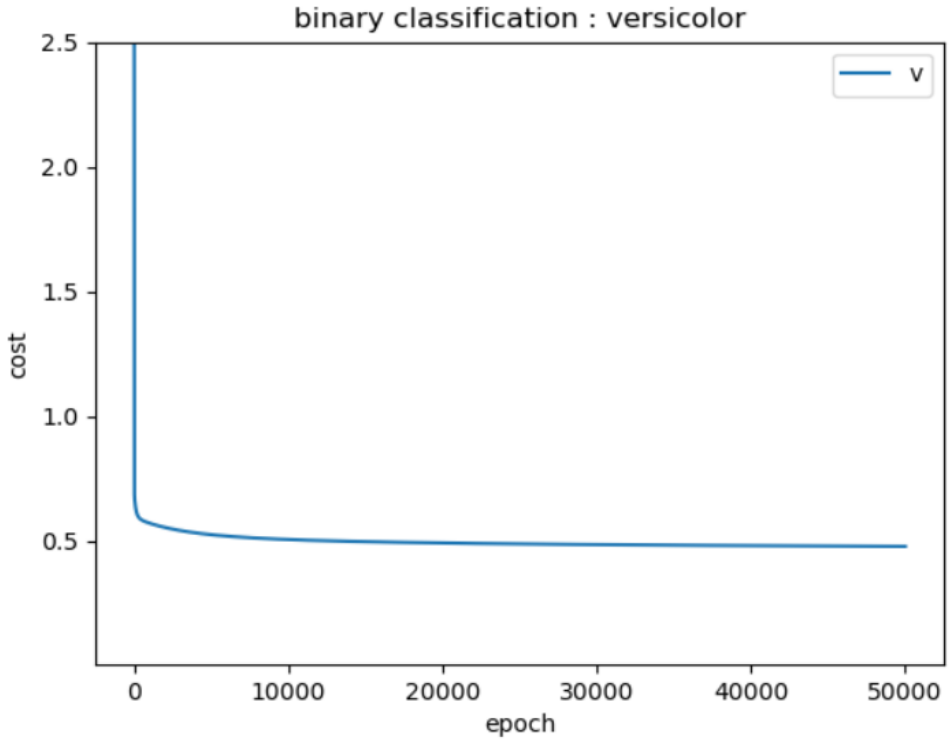


그래프를 보면 cost값이 성공적으로 감소하는 것을 관찰할 수 있다. 또한 epoch를 많이 진행하지 않은 초기 일수록 cost값이 큰 폭으로 감소하였다. epoch를 진행하면 진행할수록 cost 값이 줄어들기는 하지만 연산량에 비해 유의미한 값의 감소를 더 이상 보여주지 않는다. epoch값을 50000 이상으로 했을 때도 유사한 결과가 나왔기 때문이다.

정확도는 1.0으로 매우 높은데 이것은 cost값이 0과 가깝기 때문에 높은 정확도를 보여주는 것이다. 다른 요인으로는 test data set의 크기가 작기 때문인 점을 고려할 수 있다. 적은 테스트 양에 대해 모든 결과를 맞추었기 때문에 정확도가 1.0이 나올 수 있는 것이고 많는 양의 테스트를 진행하게 되면 1.0 보다는 떨어진 정확도가 나올 가능성이 크다. 물론 cost값의 추이를 보면 학습 모델이 성공적으로 작동하므로 1.0에 근사한 정확도가 나올 것이다.

- IRIS Single Class / target class : versicolor / epoch = 50000, learning rate = 0.01

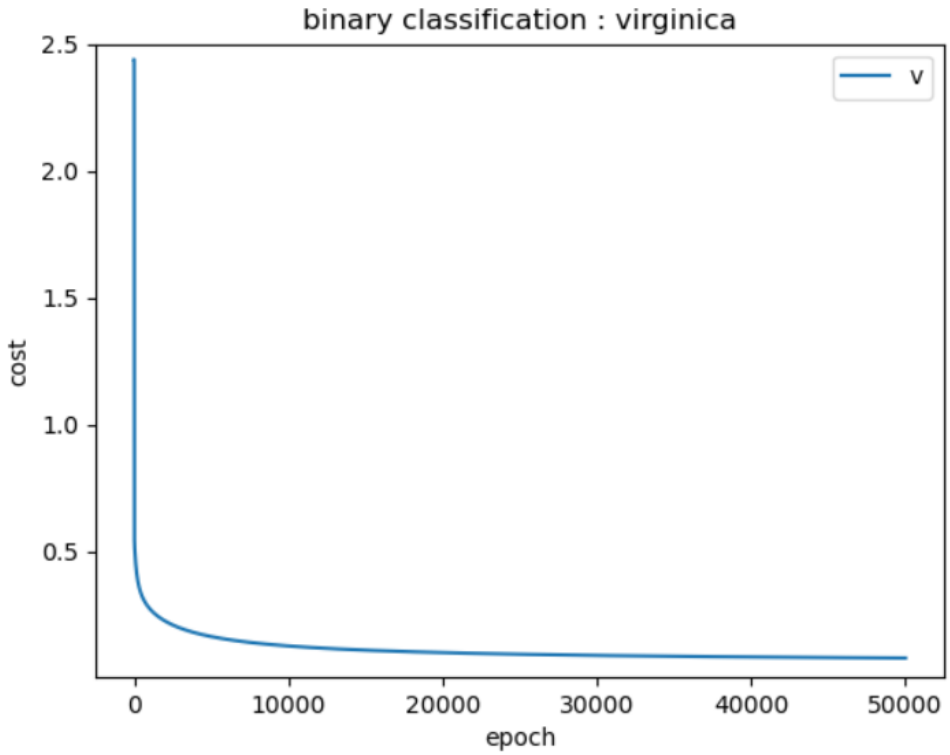
```
epoch : 0 cost : 6.7118358246630505    epoch : 10000 cost : 0.5065790920638549    epoch : 49700 cost : 0.4793092936831943
epoch : 20 cost : 1.8995722524280503   epoch : 10020 cost : 0.5065318496216112    epoch : 49720 cost : 0.479304396287986
epoch : 40 cost : 0.6767287411059958   epoch : 10040 cost : 0.5064847529469565    epoch : 49740 cost : 0.47929950204626537
epoch : 60 cost : 0.6597397601245466   epoch : 10060 cost : 0.506437801313531     epoch : 49760 cost : 0.4792946109554066
epoch : 80 cost : 0.6481583119576435   epoch : 10080 cost : 0.5063909939994317    epoch : 49780 cost : 0.47928972301278533
epoch : 100 cost : 0.6385885793666192  epoch : 10100 cost : 0.5063443302871783    epoch : 49800 cost : 0.47928483821578205
epoch : 120 cost : 0.6306410501757218  epoch : 10120 cost : 0.5062978094636859    epoch : 49820 cost : 0.4792799565617784
epoch : 140 cost : 0.6240068228573622  epoch : 10140 cost : 0.5062514308202302    epoch : 49840 cost : 0.47927507804816066
epoch : 160 cost : 0.6184377852974435  epoch : 10160 cost : 0.5062051936524202    epoch : 49860 cost : 0.479270202672317
epoch : 180 cost : 0.6137349913813125  epoch : 10180 cost : 0.506159097260167     epoch : 49880 cost : 0.4792653304316386
epoch : 200 cost : 0.6097389040542692  epoch : 10200 cost : 0.5061131409476535    epoch : 49900 cost : 0.47926046132352
epoch : 220 cost : 0.6063214117343041  epoch : 10220 cost : 0.5060673240233039    epoch : 49920 cost : 0.47925559534535844
epoch : 240 cost : 0.6033794018422474  epoch : 10240 cost : 0.5060216457997562    epoch : 49940 cost : 0.4792507324945541
epoch : 260 cost : 0.6008296400755241  epoch : 10260 cost : 0.5059761055938323    epoch : 49960 cost : 0.4792458727685101
epoch : 280 cost : 0.5986047143056237  epoch : 10280 cost : 0.5059307027265071    epoch : 49980 cost : 0.4792410161646328
epoch : 300 cost : 0.596649831329634   epoch : 10300 cost : 0.5058854365228815    Accuracy : 0.7
epoch : 320 cost : 0.5949202893733171  epoch : 10320 cost : 0.5058403063121536
epoch : 340 cost : 0.5933794825440016  epoch : 10340 cost : 0.5057953114275896
epoch : 360 cost : 0.5919973225988483  epoch : 10360 cost : 0.5057504512064966
```



두번째 target 인 versicolor는 cost 값이 초기에는 매우 빠르게 감소하였으나 0.5를 기점으로 많은 학습을 진행해도 값이 거의 줄어들지 않는 모습을 보인다. epoch를 늘려가며 관찰하였으나 5만번을 반복한 결과도 0.479...에 그친 것을 볼 수 있다. 이것은 cost 함수자체의 모형이 최저가 0.5에 가까운 값을 가지고 있기 때문이라 예상된다. 정확도는 0.7로 오차가 분명 존재하며 이것은 cost값이 0에 가깝지 않기 때문이다.

- IRIS Single Class / target class : verginica / epoch = 50000, learning rate = 0.01

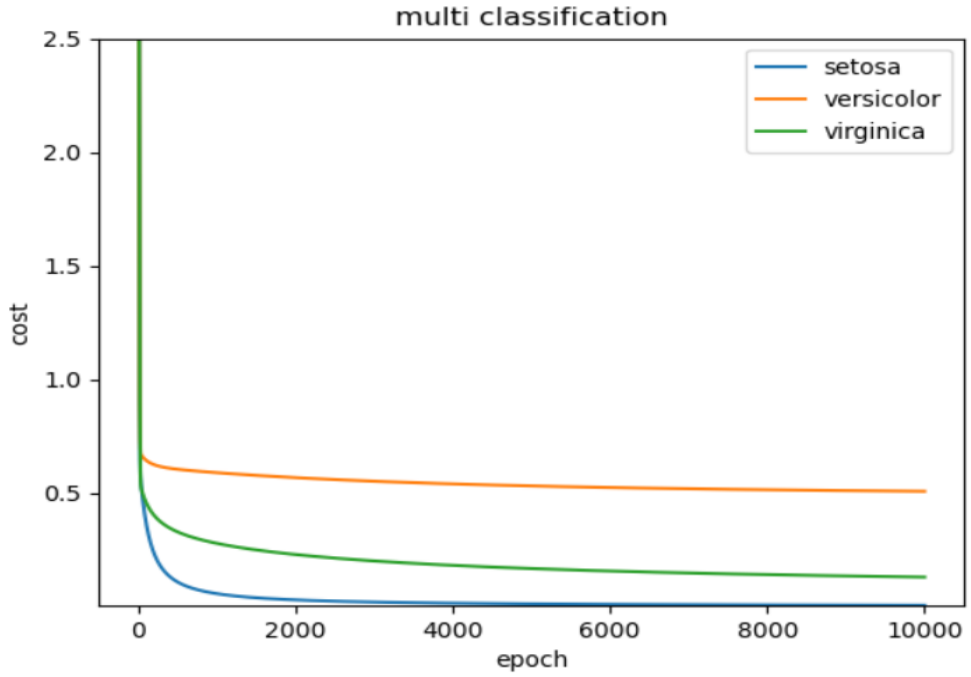
```
epoch : 0 cost : 2.438279330889812
epoch : 20 cost : 0.5562051094761313
epoch : 40 cost : 0.512787196386243
epoch : 60 cost : 0.4907121826050404
epoch : 80 cost : 0.47194618988936754
epoch : 100 cost : 0.45580640654946425
epoch : 120 cost : 0.4417856101073971
epoch : 140 cost : 0.4294906515457319
epoch : 160 cost : 0.4186146042169629
epoch : 180 cost : 0.40891590369957054
epoch : 200 cost : 0.4002027980992182
epoch : 220 cost : 0.392321754223379
epoch : 240 cost : 0.3851487841797643
epoch : 260 cost : 0.37858292245364444
epoch : 280 cost : 0.37254128859994967
epoch : 300 cost : 0.36695532334585035
epoch : 320 cost : 0.3617678977446933
epoch : 340 cost : 0.3569310758946902
epoch : 360 cost : 0.35240437028390714
epoch : 10000 cost : 0.12879355837358913
epoch : 10020 cost : 0.12870283011485242
epoch : 10040 cost : 0.12861237576619833
epoch : 10060 cost : 0.12852219403332882
epoch : 10080 cost : 0.12843228363020048
epoch : 10100 cost : 0.128342643278959
epoch : 10120 cost : 0.12825327170987297
epoch : 10140 cost : 0.12816416766126967
epoch : 10160 cost : 0.1280753298794705
epoch : 10180 cost : 0.12798675711872795
epoch : 10200 cost : 0.12789844814116164
epoch : 10220 cost : 0.1278104017166971
epoch : 10240 cost : 0.1277226166230031
epoch : 10260 cost : 0.127635091645431
epoch : 10280 cost : 0.12754782557695388
epoch : 10300 cost : 0.12746081721810648
epoch : 10320 cost : 0.12737406537692597
epoch : 10340 cost : 0.12728756886889286
epoch : 10360 cost : 0.12720132651687285
epoch : 49640 cost : 0.08138172793666561
epoch : 49660 cost : 0.08137473684023802
epoch : 49680 cost : 0.08136775030518821
epoch : 49700 cost : 0.08136076832664764
epoch : 49720 cost : 0.08135379089975457
epoch : 49740 cost : 0.08134681801965447
epoch : 49760 cost : 0.08133984968150007
epoch : 49780 cost : 0.08133288588045091
epoch : 49800 cost : 0.08132592661167379
epoch : 49820 cost : 0.08131897187034229
epoch : 49840 cost : 0.0813120216516373
epoch : 49860 cost : 0.08130507595074658
epoch : 49880 cost : 0.0812981347628649
epoch : 49900 cost : 0.08129119808319402
epoch : 49920 cost : 0.08128426590694264
epoch : 49940 cost : 0.08127733822932659
epoch : 49960 cost : 0.08127041504556848
epoch : 49980 cost : 0.08126349635089773
Accuracy : 0.967
```



마지막 target verginica에 대한 결과이다. 마찬가지로 cost값이 성공적으로 감소하였다. 앞의 두 target과 동일하게 초기 cost 값이 빠르게 감소하지만 이후의 epoch에서도 유의미한 감소를 보여주는 다른 결과를 띈다. epoch가 10000일 때 cost 값은 0.13에 가까운 값이지만 epoch가 50000일 때는 0.08까지 cost값이 감소하였다. cost값이 0에 매우 가깝기 때문에 정확도는 1.0에 근접한 0.967이 나왔다.

- IRIS Multi Class / epoch = 10000, learning rate = 0.01

```
epoch : 0 cost : [7.37863418 5.96833743 5.81861271]    epoch : 4000 cost : [0.01645106 0.54099056 0.18290713]  epoch : 9540 cost : [0.00751989 0.5098651  0.1333951 ]
epoch : 20 cost : [1.37160011 1.35284355 1.75446136]   epoch : 4020 cost : [0.01637685 0.54078725 0.1825819 ]  epoch : 9560 cost : [0.0075058  0.50980481 0.1332959 ]
epoch : 40 cost : [0.54376041 0.67008412 0.52731333]   epoch : 4040 cost : [0.01630335 0.54058505 0.18225865]  epoch : 9580 cost : [0.00749177 0.50974471 0.13319701]
epoch : 60 cost : [0.47755393 0.65997075 0.49978577]   epoch : 4060 cost : [0.01623055 0.54038397 0.18193735]  epoch : 9600 cost : [0.00747779 0.50968482 0.13309842]
epoch : 80 cost : [0.42394499 0.65253647 0.48006312]   epoch : 4080 cost : [0.01615843 0.54018399 0.181618  ]  epoch : 9620 cost : [0.00746387 0.50962514 0.13300015]
epoch : 100 cost : [0.37919597 0.64629043 0.46316113]  epoch : 4100 cost : [0.01608699 0.5399851  0.18130056]  epoch : 9640 cost : [0.00745001 0.50956565 0.13290217]
epoch : 120 cost : [0.34170257 0.64100799 0.448524  ]  epoch : 4120 cost : [0.01601622 0.53978729 0.18098503]  epoch : 9660 cost : [0.0074362  0.50950636 0.1328045 ]
epoch : 140 cost : [0.31010857 0.63651095 0.43572653]  epoch : 4140 cost : [0.0159461  0.53959057 0.18067138]  epoch : 9680 cost : [0.00742244 0.50944727 0.13270714]
epoch : 160 cost : [0.28330132 0.63265617 0.42443742]  epoch : 4160 cost : [0.01587663 0.53939491 0.18035959]  epoch : 9700 cost : [0.00740874 0.50938837 0.13261007]
epoch : 180 cost : [0.26038572 0.6293285  0.4143965 ]  epoch : 4180 cost : [0.01580781 0.53920032 0.18004965]  epoch : 9720 cost : [0.00739509 0.50932968 0.1325133 ]
epoch : 200 cost : [0.24064844 0.62643507 0.40539773]  epoch : 4200 cost : [0.01573961 0.53900678 0.17974155]  epoch : 9740 cost : [0.0073815  0.50927117 0.13241683]
epoch : 220 cost : [0.22352253 0.62390075 0.39727654]  epoch : 4220 cost : [0.01567203 0.53881429 0.17943526]  epoch : 9760 cost : [0.00736796 0.50921287 0.13232066]
epoch : 240 cost : [0.20855689 0.62166456 0.38990034]  epoch : 4240 cost : [0.01560506 0.53862284 0.17913077]  epoch : 9780 cost : [0.00735447 0.50915475 0.13222478]
epoch : 260 cost : [0.19539115 0.6196768  0.38316145]  epoch : 4260 cost : [0.0155387  0.53843243 0.17882806]  epoch : 9800 cost : [0.00734103 0.50909683 0.13212919]
epoch : 280 cost : [0.18373591 0.61789686 0.37697165]  epoch : 4280 cost : [0.01547293 0.53824304 0.17852711]  epoch : 9820 cost : [0.00732765 0.5090391  0.1320339 ]
epoch : 300 cost : [0.17335731 0.61629142 0.37125813]  epoch : 4300 cost : [0.01540776 0.53805467 0.17822791]  epoch : 9840 cost : [0.00731432 0.50898155 0.1319389 ]
epoch : 320 cost : [0.16406507 0.61483302 0.36596037]  epoch : 4320 cost : [0.01534316 0.53786731 0.17793045]  epoch : 9860 cost : [0.00730104 0.5089242  0.13184418]
epoch : 340 cost : [0.15570336 0.61349898 0.36102768]  epoch : 4340 cost : [0.01527913 0.53768095 0.1776347 ]  epoch : 9880 cost : [0.00728781 0.50886704 0.13174976]
epoch : 360 cost : [0.14814365 0.61227048 0.35641733]  epoch : 4360 cost : [0.01521567 0.5374956  0.17734065]  epoch : 9900 cost : [0.00727463 0.50881006 0.13165562]
epoch : 380 cost : [0.14127922 0.61113186 0.35209308]  epoch : 4380 cost : [0.01515276 0.53731123 0.17704829]  epoch : 9920 cost : [0.00726151 0.50875327 0.13156176]
epoch : 400 cost : [0.1350209  0.61007006 0.34802398]  epoch : 4400 cost : [0.01509041 0.53712785 0.1767576 ]  epoch : 9940 cost : [0.00724843 0.50869667 0.13146819]
epoch : 420 cost : [0.12929373 0.60907412 0.3441835 ]  epoch : 4420 cost : [0.01502859 0.53694545 0.17646857]  epoch : 9960 cost : [0.0072354  0.50864024 0.1313749 ]
epoch : 440 cost : [0.12403427 0.60813487 0.34054873]  epoch : 4440 cost : [0.01496732 0.53676402 0.17618118]  epoch : 9980 cost : [0.00722243 0.50858401 0.13128189]
epoch : 460 cost : [0.11918859 0.60724454 0.33709981]  epoch : 4460 cost : [0.01490656 0.53658355 0.17589541]  Accuracy : 0.933
```

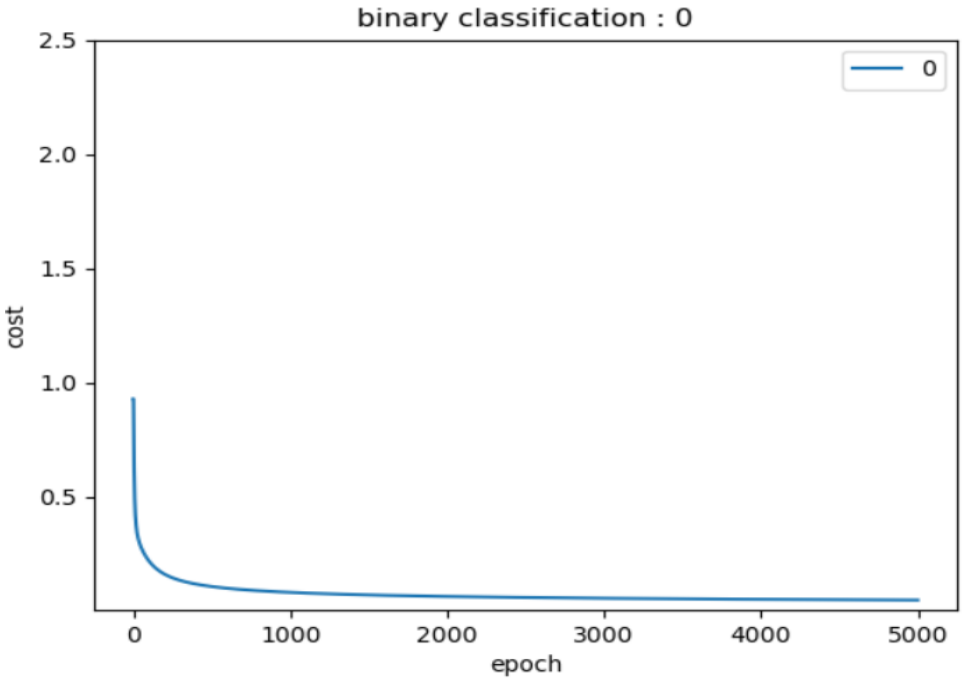


multi classification에서는 동일한 epoch 수를 적용해야 하므로 10000을 적정 값으로 상정하여 진행하였다. versicolor와 setosa는 epoch수가 많이 반복되어도 유의미한 cost값 변화가 없는 반면 virginica의 cost값은 꾸준히 감소함을 관찰할 수 있다. 정확도는 setosa와 virginica에 비해서 떨어지는 0.933이 나왔는데 이것은 versicolor의 cost 값이 크기 때문에 결과를 도출함에 있어 versicolor 클래스 예측을 잘하지 못했을 것이기 때문으로 생각된다. 하지만 binary classification의 정확도 평균은 1.0 + 0.7 + 0.967 / 3 = 0.889로 0.933은 이것보다 높은 확률로 class 예측에 성공한 결과를 보여주어 multi classification도 좋은 결과를 보여줬음을 알 수 있다.

6.2 Mnist data

- Mnist Single Class / target class : 0 / epoch = 5000, learning rate = 0.01

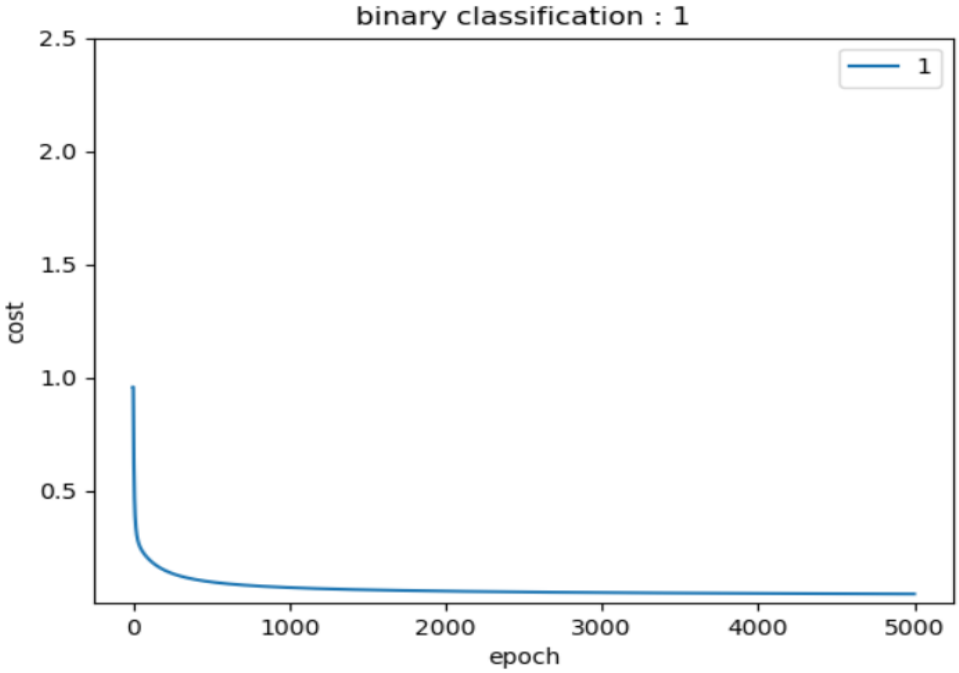
```
epoch : 0 cost : 0.9293491959123508     epoch : 2000 cost : 0.06504529322113538   epoch : 4540 cost : 0.05065319678338919
epoch : 20 cost : 0.36305012141988446   epoch : 2020 cost : 0.0648329345667157    epoch : 4560 cost : 0.05059089009428667
epoch : 40 cost : 0.2978785489622548    epoch : 2040 cost : 0.06462370159414725   epoch : 4580 cost : 0.05052899305204179
epoch : 60 cost : 0.26457212564411065   epoch : 2060 cost : 0.06441751831338693   epoch : 4600 cost : 0.05046750119238597
epoch : 80 cost : 0.23977685003603377   epoch : 2080 cost : 0.06421431131013534   epoch : 4620 cost : 0.05040641011863549
epoch : 100 cost : 0.2199712705959792   epoch : 2100 cost : 0.06401400963390763   epoch : 4640 cost : 0.05034571550038539
epoch : 120 cost : 0.20385072149912684  epoch : 2120 cost : 0.06381654469202182   epoch : 4660 cost : 0.0502854130722299
epoch : 140 cost : 0.1905675572317308   epoch : 2140 cost : 0.06362185014913553   epoch : 4680 cost : 0.05022549863251862
epoch : 160 cost : 0.17948686194395602  epoch : 2160 cost : 0.06342986183199324   epoch : 4700 cost : 0.05016596804213639
epoch : 180 cost : 0.17012823445576766  epoch : 2180 cost : 0.06324051763907969   epoch : 4720 cost : 0.050106817223318285
epoch : 200 cost : 0.1621295603416829   epoch : 2200 cost : 0.06305375745484083   epoch : 4740 cost : 0.050048042158484844
epoch : 220 cost : 0.1552173572182446   epoch : 2220 cost : 0.0628695230682678    epoch : 4760 cost : 0.04998963888910679
epoch : 240 cost : 0.1491835853526257   epoch : 2240 cost : 0.06268775809551139   epoch : 4780 cost : 0.0499316035145974
epoch : 260 cost : 0.14386838818115805  epoch : 2260 cost : 0.06250840790634567   epoch : 4800 cost : 0.04987393219123031
epoch : 280 cost : 0.13914754771087642  epoch : 2280 cost : 0.06233141955423348 5  epoch : 4820 cost : 0.049816621131079084
epoch : 300 cost : 0.1349234294143092   epoch : 2300 cost : 0.06215674170978437   epoch : 4840 cost : 0.049759666600981016
epoch : 320 cost : 0.13111843144377308  epoch : 2320 cost : 0.061984324597417664  epoch : 4860 cost : 0.04970306492152945
epoch : 340 cost : 0.1276702128585024   epoch : 2340 cost : 0.0618141199350543    epoch : 4880 cost : 0.04964681246607906
epoch : 360 cost : 0.12452818579459275  epoch : 2360 cost : 0.061646080876652874  epoch : 4900 cost : 0.04959090565978551
epoch : 380 cost : 0.12165091050789544  epoch : 2380 cost : 0.06148016195744333   epoch : 4920 cost : 0.049535340978655834
epoch : 400 cost : 0.11900414060254284  epoch : 2400 cost : 0.06131631904170806   epoch : 4940 cost : 0.04948011494862635
epoch : 420 cost : 0.11655934088633546  epoch : 2420 cost : 0.061154509272965056  epoch : 4960 cost : 0.04942522414465767
epoch : 440 cost : 0.11429255220136296  epoch : 2440 cost : 0.060994691026426545  epoch : 4980 cost : 0.049370665189853155
epoch : 460 cost : 0.11218351354282173  epoch : 2460 cost : 0.06083682386362078   Accuracy : 0.99
```



성공적으로 cost값이 감소하였으며 epoch 2000까지는 어느정도 유의미한 감소폭을 보여준다. 정확도는 0.99로 10000개의 test data set에 대해 매우 성공적인 classification을 보여주었다. test data set이 iris와는 다르게 매우 많으므로 정확도를 결정짓는 것은 cost값의 크기임을 알 수 있다.

- Mnist Single Class / target class : 1 / epoch = 5000, learning rate = 0.01

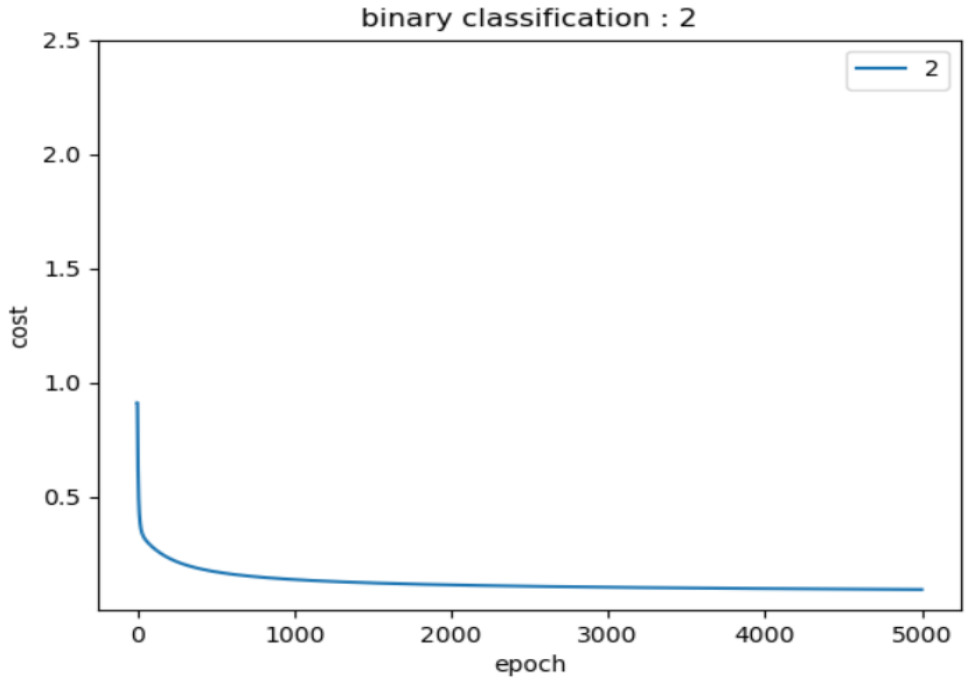
```
epoch : 0 cost : 0.9588990568073193     epoch : 2000 cost : 0.057976734753228364  epoch : 4540 cost : 0.04650731833896489
epoch : 20 cost : 0.32556762636405323   epoch : 2020 cost : 0.05779832006707822   epoch : 4560 cost : 0.046460542298176435
epoch : 40 cost : 0.2608832511833099    epoch : 2040 cost : 0.05762278696030963   epoch : 4580 cost : 0.046414099086287686
epoch : 60 cost : 0.23356049975257925   epoch : 2060 cost : 0.05745006103389891   epoch : 4600 cost : 0.04636798484906571
epoch : 80 cost : 0.21467635553206457   epoch : 2080 cost : 0.057280070518298906  epoch : 4620 cost : 0.0463221957930765
epoch : 100 cost : 0.1995122938037801   epoch : 2100 cost : 0.05711274615590566   epoch : 4640 cost : 0.046276728184464246
epoch : 120 cost : 0.1867354982549591   epoch : 2120 cost : 0.05694802108985943 4  epoch : 4660 cost : 0.046231578347775644
epoch : 140 cost : 0.17577284724876863  epoch : 2140 cost : 0.05678583075875813   epoch : 4680 cost : 0.04618674266479452
epoch : 160 cost : 0.1662784938218998   epoch : 2160 cost : 0.0566261127969368    epoch : 4700 cost : 0.046142217573421795
epoch : 180 cost : 0.15799924092767592  epoch : 2180 cost : 0.0564688069399556    epoch : 4720 cost : 0.046097999566568425
epoch : 200 cost : 0.1507345825747089   epoch : 2200 cost : 0.056313854934994544  epoch : 4740 cost : 0.046054085191081486
epoch : 220 cost : 0.14432166849908906  epoch : 2220 cost : 0.056161200455844495  epoch : 4760 cost : 0.046010471046698885
epoch : 240 cost : 0.13862729515634248  epoch : 2240 cost : 0.05601078902222406   epoch : 4780 cost : 0.04596715378501556
epoch : 260 cost : 0.1335422377347238   epoch : 2260 cost : 0.0558625679231768    epoch : 4800 cost : 0.04592413010849121
epoch : 280 cost : 0.12897672508050753  epoch : 2280 cost : 0.055716486144284136  epoch : 4820 cost : 0.04588139676946647
epoch : 300 cost : 0.12485673471999616  epoch : 2300 cost : 0.055572494298507503  epoch : 4840 cost : 0.045838950569213804
epoch : 320 cost : 0.1211209741124723   epoch : 2320 cost : 0.0554305445640745    epoch : 4860 cost : 0.04579678835699885
epoch : 340 cost : 0.11771844428047162  epoch : 2340 cost : 0.055290590603582344  epoch : 4880 cost : 0.04575490702917526
epoch : 360 cost : 0.11460648738710298  epoch : 2360 cost : 0.0551525875411 2823  epoch : 4900 cost : 0.045713303528288625
epoch : 380 cost : 0.11174922784317384  epoch : 2380 cost : 0.055016491868942634  epoch : 4920 cost : 0.045671974842212214
epoch : 400 cost : 0.10911632853418253  epoch : 2400 cost : 0.05488226141173524   epoch : 4940 cost : 0.045630918003292616
epoch : 420 cost : 0.10668199705870526  epoch : 2420 cost : 0.05474985527156784   epoch : 4960 cost : 0.04559013008752383
epoch : 440 cost : 0.1044241894210973   epoch : 2440 cost : 0.054619233778810894  epoch : 4980 cost : 0.04554960821372998
epoch : 460 cost : 0.10232396947705687  epoch : 2460 cost : 0.05449035844536277   Accuracy : 0.989
```



0과 비교했을 때 cost값이 더욱 빠른 폭으로 감소하였다. cost 값은 0과 비슷하게 나왔지만 정확도는 0.989로 10개의 test data를 더 맞추지 못했다. test data set의 크기가 10000임을 고려했을 때 큰 차이는 아니지만 test data set이 어떤 형태를 띄고 있는지 프로그램이 미리 예측할 수는 없으므로 반드시 실제 정확도가 cost값이 0과 가까운 것만큼 비례하지 않음을 알 수 있다. 간단히 말해 test data set의 형태가 train data set과 다르면 다를수록 학습 모델의 정확도는 떨어질 수 있다는 것이다.

- Mnist Single Class / target class : 2 / epoch = 5000, learning rate = 0.01

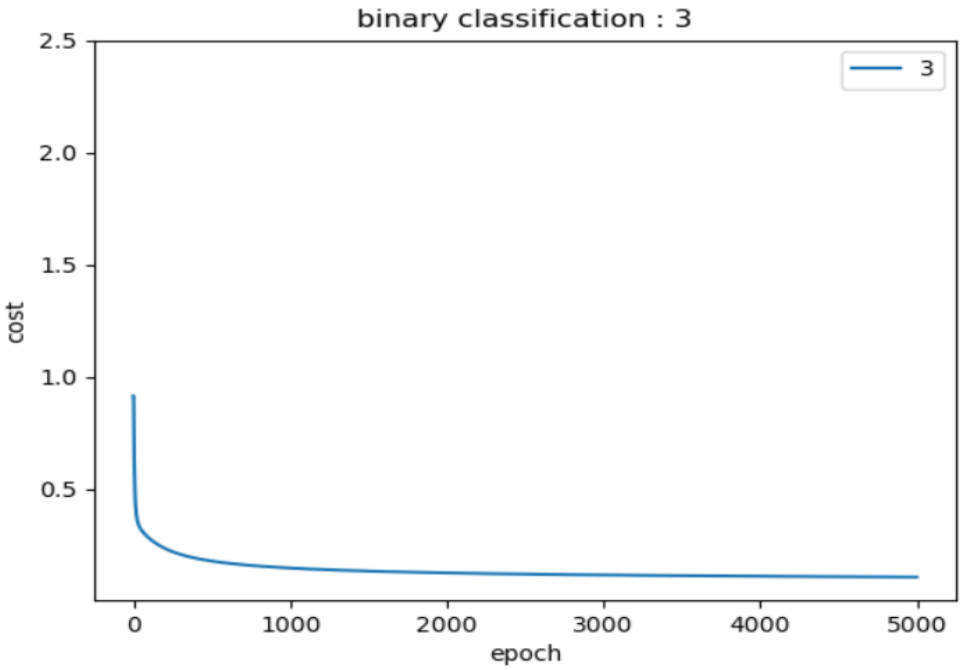
```
epoch : 0 cost : 0.9125330654822881      epoch : 2000 cost : 0.11543201824128745   epoch : 4540 cost : 0.09713644645277998
epoch : 20 cost : 0.3751628551304288     epoch : 2020 cost : 0.11515551216379664   epoch : 4560 cost : 0.09705916055173947
epoch : 40 cost : 0.3265330192058443     epoch : 2040 cost : 0.1148832524971004    epoch : 4580 cost : 0.09698239678698094
epoch : 60 cost : 0.306660695407901      epoch : 2060 cost : 0.11461513392400993   epoch : 4600 cost : 0.09690614928936674
epoch : 80 cost : 0.292022372613139      epoch : 2080 cost : 0.11435105469648225   epoch : 4620 cost : 0.09683041228121045
epoch : 100 cost : 0.2794343506630934    epoch : 2100 cost : 0.11409091648263239   epoch : 4640 cost : 0.09675518007446268
epoch : 120 cost : 0.26820964229867605   epoch : 2120 cost : 0.11383462422163171   epoch : 4660 cost : 0.09668044706896106
epoch : 140 cost : 0.2581146319395105    epoch : 2140 cost : 0.11358208598607637   epoch : 4680 cost : 0.09660620775069138
epoch : 160 cost : 0.24901186135876474   epoch : 2160 cost : 0.11333321285132666   epoch : 4700 cost : 0.09653245669011888
epoch : 180 cost : 0.24078882736101356   epoch : 2180 cost : 0.11308791877143878   epoch : 4720 cost : 0.09645918854054106
epoch : 200 cost : 0.23334463909626826   epoch : 2200 cost : 0.11284612046129432   epoch : 4740 cost : 0.09638639803647844
epoch : 220 cost : 0.22658829669592975   epoch : 2220 cost : 0.11260773728454132   epoch : 4760 cost : 0.0963140799921227
epoch : 240 cost : 0.22043864755970866   epoch : 2240 cost : 0.11237269114704891   epoch : 4780 cost : 0.0962422292997881
epoch : 260 cost : 0.214824119708528     epoch : 2260 cost : 0.11214090639552275   epoch : 4800 cost : 0.09617084092843822
epoch : 280 cost : 0.20968208465647095   epoch : 2280 cost : 0.11191230972101443   epoch : 4820 cost : 0.09609990992221093
epoch : 300 cost : 0.20495800333541478   epoch : 2300 cost : 0.11168683006703971   epoch : 4840 cost : 0.09602943139899557
epoch : 320 cost : 0.20060449953642193   epoch : 2320 cost : 0.11146439854204086   epoch : 4860 cost : 0.09595940054904176
epoch : 340 cost : 0.19658045224446835   epoch : 2340 cost : 0.1112449483359579    epoch : 4880 cost : 0.0958898126336003
epoch : 360 cost : 0.19285015525775906   epoch : 2360 cost : 0.11102841464069857   epoch : 4900 cost : 0.09582066298358648
epoch : 380 cost : 0.18938256490976504   epoch : 2380 cost : 0.11081473457425824   epoch : 4920 cost : 0.09575194699828228
epoch : 400 cost : 0.186150640706808     epoch : 2400 cost : 0.11060384710832939   epoch : 4940 cost : 0.0956836601440622
epoch : 420 cost : 0.18313077511957787   epoch : 2420 cost : 0.11039569299919944   epoch : 4960 cost : 0.09561579795316213
epoch : 440 cost : 0.1803023046631812    epoch : 2440 cost : 0.11019021472174184   epoch : 4980 cost : 0.09554835602244635
epoch : 460 cost : 0.17764709286835692   epoch : 2460 cost : 0.10998735640638317   Accuracy : 0.974
```



0과 1에 비해서는 cost값이 큰 편이고 정확도는 낮은 편이다. cost값에 영향을 받아 정확도가 떨어졌음을 예상할 수 있다.

- Mnist Single Class / target class : 3 / epoch = 5000, learning rate = 0.01

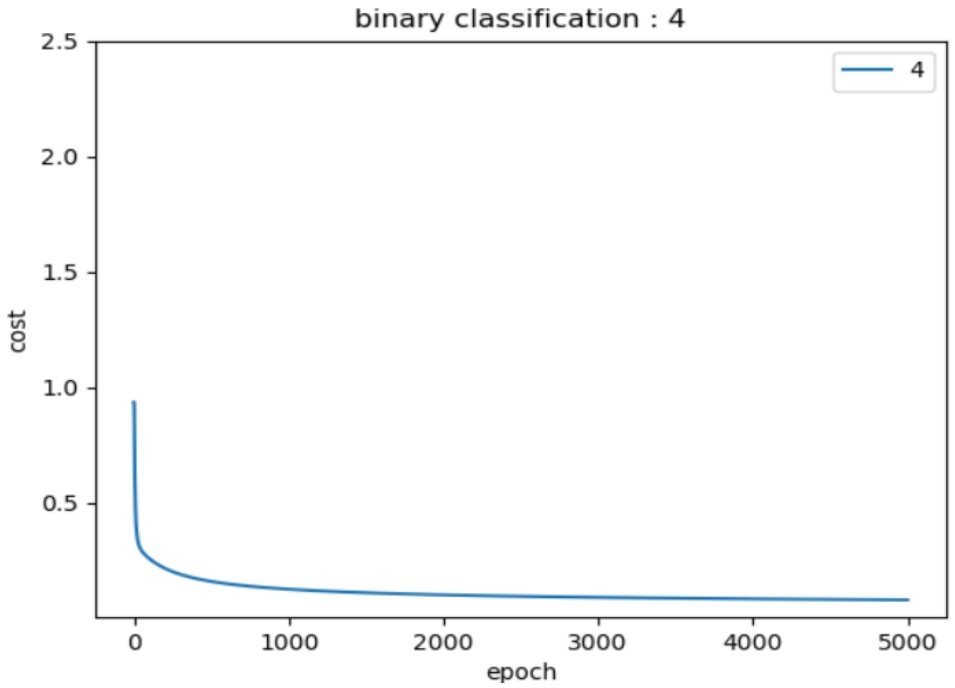
```
epoch : 0 cost : 0.9161111564995837     epoch : 2000 cost : 0.1275213076362076   epoch : 4540 cost : 0.11063960242316212
epoch : 20 cost : 0.37954358674572364   epoch : 2020 cost : 0.12726659440625504  epoch : 4560 cost : 0.1105677360967944
epoch : 40 cost : 0.33053557683792995   epoch : 2040 cost : 0.12701579421905565  epoch : 4580 cost : 0.11049634268413616
epoch : 60 cost : 0.3101299688908572    epoch : 2060 cost : 0.12676881008351845  epoch : 4600 cost : 0.11042541679655606
epoch : 80 cost : 0.29508217356177235   epoch : 2080 cost : 0.12652554827350038  epoch : 4620 cost : 0.11035495313025731
epoch : 100 cost : 0.28226683390932944  epoch : 2100 cost : 0.12628591818886314  epoch : 4640 cost : 0.1102849464645999
epoch : 120 cost : 0.27098230139250207  epoch : 2120 cost : 0.1260498322236751   epoch : 4660 cost : 0.11021539166045315
epoch : 140 cost : 0.26095943337601774  epoch : 2140 cost : 0.1258172056411483   epoch : 4680 cost : 0.11014628365859876
epoch : 160 cost : 0.25202421448531187  epoch : 2160 cost : 0.12558795645489465  epoch : 4700 cost : 0.11007761747815714
epoch : 180 cost : 0.2440333493031474   epoch : 2180 cost : 0.12536200531613645  epoch : 4720 cost : 0.1100093882150626
epoch : 200 cost : 0.23686222663610892  epoch : 2200 cost : 0.12513927540649702  epoch : 4740 cost : 0.10994159104056742
epoch : 220 cost : 0.23040244997736606  epoch : 2220 cost : 0.12491969233609042  epoch : 4760 cost : 0.10987422119978724
epoch : 240 cost : 0.224560503104446    epoch : 2240 cost : 0.12470318404655877  epoch : 4780 cost : 0.10980727401027493
epoch : 260 cost : 0.2192562379648553   epoch : 2260 cost : 0.12448968071880714  epoch : 4800 cost : 0.10974074486062885
epoch : 280 cost : 0.21442121511099188  epoch : 2280 cost : 0.12427911468513751  epoch : 4820 cost : 0.10967462920912796
epoch : 300 cost : 0.2099970670830625   epoch : 2300 cost : 0.12407142034557755  epoch : 4840 cost : 0.10960892258241504
epoch : 320 cost : 0.20593399369329868  epoch : 2320 cost : 0.12386653408813042  epoch : 4860 cost : 0.10954362057419216
epoch : 340 cost : 0.2021894353175746   epoch : 2340 cost : 0.12366439421274997  epoch : 4880 cost : 0.1094787188439555
epoch : 360 cost : 0.19872693209263267  epoch : 2360 cost : 0.12346494085882652  epoch : 4900 cost : 0.10941421311574327
epoch : 380 cost : 0.1955151576557102   epoch : 2380 cost : 0.12326811593601111  epoch : 4920 cost : 0.10935009917695189
epoch : 400 cost : 0.19252710800999479  epoch : 2400 cost : 0.12307386305817984  epoch : 4940 cost : 0.10928637287712388
epoch : 420 cost : 0.1897394240172182   epoch : 2420 cost : 0.12288212748036374  epoch : 4960 cost : 0.10922303012680724
epoch : 440 cost : 0.1871318268636128   epoch : 2440 cost : 0.12269285603852306  epoch : 4980 cost : 0.10916006689641912
epoch : 460 cost : 0.18468664795700573  epoch : 2460 cost : 0.12250599709197599  Accuracy : 0.97
```



0, 1, 2보다 cost값이 크고 이에 따라 정확도는 가장 낮다.

- Mnist Single Class / target class : 4 / epoch = 5000, learning rate = 0.01

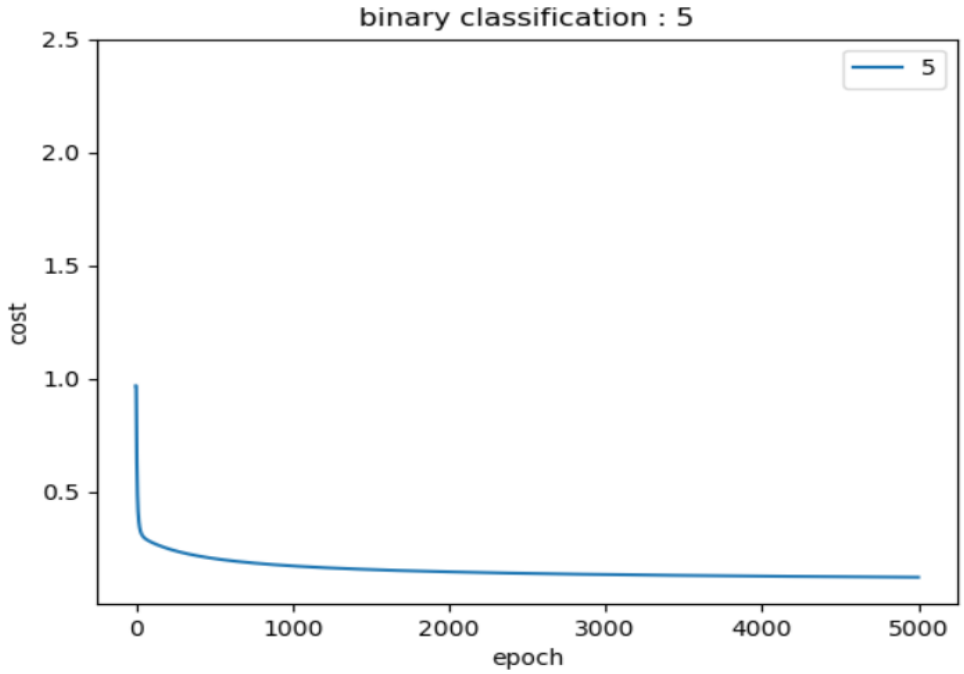
```
epoch : 0 cost : 0.9365151987548623      epoch : 2000 cost : 0.10240544383816061  epoch : 4540 cost : 0.08257404376853077
epoch : 20 cost : 0.35676588307290613    epoch : 2020 cost : 0.10211046737605596  epoch : 4560 cost : 0.08248961676682616
epoch : 40 cost : 0.30429405910536744    epoch : 2040 cost : 0.10181985269679687  epoch : 4580 cost : 0.08240576611436395
epoch : 60 cost : 0.2842649284405897     epoch : 2060 cost : 0.10153349501295757  epoch : 4600 cost : 0.08232248550782478
epoch : 80 cost : 0.2704303265489347     epoch : 2080 cost : 0.10125129302654401  epoch : 4620 cost : 0.08223976873882291
epoch : 100 cost : 0.25889463761482806   epoch : 2100 cost : 0.10097314878107151  epoch : 4640 cost : 0.08215760969207658
epoch : 120 cost : 0.2487303846284215    epoch : 2120 cost : 0.10069896752122569  epoch : 4660 cost : 0.08207600234362726
epoch : 140 cost : 0.23962026421227095   epoch : 2140 cost : 0.1004286575596338   epoch : 4680 cost : 0.08199494075908971
epoch : 160 cost : 0.2314015678853539    epoch : 2160 cost : 0.1001621301503413   epoch : 4700 cost : 0.08191441909196105
epoch : 180 cost : 0.22395912812339155   epoch : 2180 cost : 0.0998992993685895   epoch : 4720 cost : 0.08183443158194999
epoch : 200 cost : 0.21719806962763305   epoch : 2200 cost : 0.09964008199654029  epoch : 4740 cost : 0.08175497255335089
epoch : 220 cost : 0.21103648642943995   epoch : 2220 cost : 0.09938439741459142  epoch : 4760 cost : 0.08167603641347237
epoch : 240 cost : 0.2054030507684146    epoch : 2240 cost : 0.09913216749796568  epoch : 4780 cost : 0.08159761765105879
epoch : 260 cost : 0.20023572674445606   epoch : 2260 cost : 0.09888331651828633  epoch : 4800 cost : 0.08151971083480883
epoch : 280 cost : 0.19548069200219936   epoch : 2280 cost : 0.09863777104984711  epoch : 4820 cost : 0.08144231061186505
epoch : 300 cost : 0.19109131584267763   epoch : 2300 cost : 0.09839545988030585  epoch : 4840 cost : 0.08136541170638445
epoch : 320 cost : 0.1870272025696222    epoch : 2320 cost : 0.09815631392558839  epoch : 4860 cost : 0.08128900891811565
epoch : 340 cost : 0.1832533241686362    epoch : 2340 cost : 0.09792026614874695  epoch : 4880 cost : 0.08121309712102026
epoch : 360 cost : 0.17973925421822684   epoch : 2360 cost : 0.097687251482549    epoch : 4900 cost : 0.08113767126191174
epoch : 380 cost : 0.1764585031799817    epoch : 2380 cost : 0.0974572067556278   epoch : 4920 cost : 0.08106272635914397
epoch : 400 cost : 0.17338794821895645   epoch : 2400 cost : 0.09723007062198576  epoch : 4940 cost : 0.08098825750130924
epoch : 420 cost : 0.17050734752506336   epoch : 2420 cost : 0.09700578349365398  epoch : 4960 cost : 0.08091425984598113
epoch : 440 cost : 0.1677989282543296    epoch : 2440 cost : 0.09678428747638644  epoch : 4980 cost : 0.08084072861847115
epoch : 460 cost : 0.1652470375963097    epoch : 2460 cost : 0.09656552630818205  Accuracy : 0.975
```



cost값이 2와 가장 비슷하며 정확도는 0.975로 2의 정확도인 0.974와 매우 비슷하다. 마찬가지로 여기서도 cost값의 크기와 정확도의 상관관계를 알 수 있다.

- Mnist Single Class / target class : 5 / epoch = 5000, learning rate = 0.01

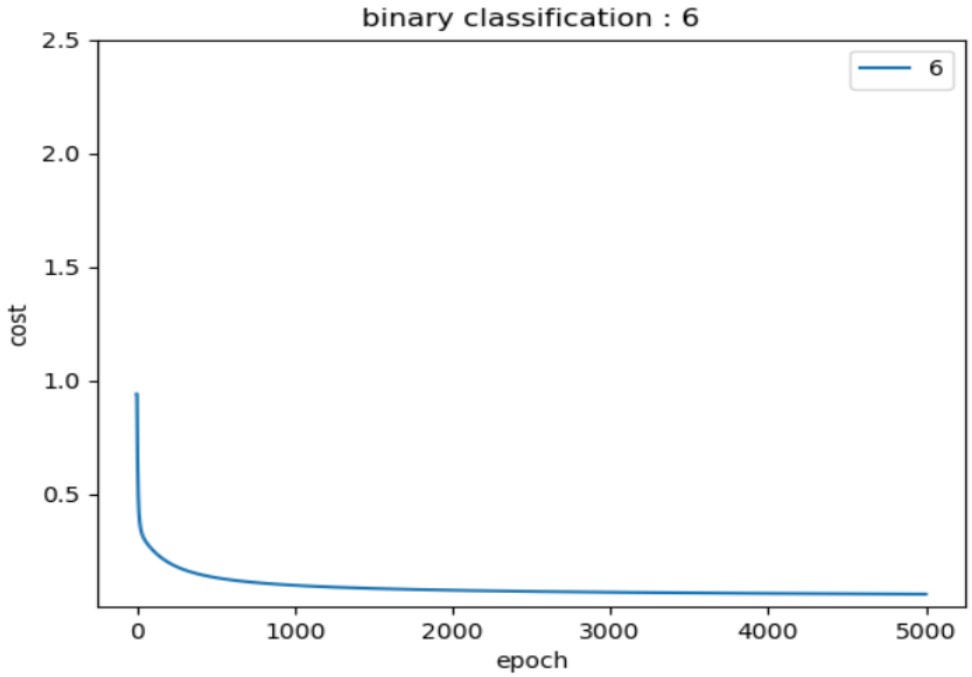
```
epoch : 0 cost : 0.9692188234020183     epoch : 2000 cost : 0.14746550691300117   epoch : 4540 cost : 0.12566872320511202
epoch : 20 cost : 0.35677279506532117   epoch : 2020 cost : 0.14714348905648839   epoch : 4560 cost : 0.12557552248090167
epoch : 40 cost : 0.3083004330626531    epoch : 2040 cost : 0.14682611806561044   epoch : 4580 cost : 0.12548295200419668
epoch : 60 cost : 0.29357117705933383   epoch : 2060 cost : 0.1465132872231414    epoch : 4600 cost : 0.12539100480642287
epoch : 80 cost : 0.28475761032096586   epoch : 2080 cost : 0.14620489316065585   epoch : 4620 cost : 0.12529967402388575
epoch : 100 cost : 0.27770508243774944  epoch : 2100 cost : 0.145900835725205     epoch : 4640 cost : 0.12520895289578898
epoch : 120 cost : 0.2714529386901938   epoch : 2120 cost : 0.14560101785237997   epoch : 4660 cost : 0.12511883476227265
epoch : 140 cost : 0.2657196941093681   epoch : 2140 cost : 0.14530534544547768   epoch : 4680 cost : 0.1250293130625291
epoch : 160 cost : 0.2604009733952645   epoch : 2160 cost : 0.14501372726034428   epoch : 4700 cost : 0.12494038133292944
epoch : 180 cost : 0.2554440463191712   epoch : 2180 cost : 0.14472607479564534   epoch : 4720 cost : 0.1248520332052194
epoch : 200 cost : 0.2508128796190431   epoch : 2200 cost : 0.1444423021882486    epoch : 4740 cost : 0.12476426240473816
epoch : 220 cost : 0.24647795528719071  epoch : 2220 cost : 0.14416232611342808   epoch : 4760 cost : 0.12467706274868333
epoch : 240 cost : 0.24241326464192273  epoch : 2240 cost : 0.14388606568962706   epoch : 4780 cost : 0.12459042814441756
epoch : 260 cost : 0.2385953902332195   epoch : 2260 cost : 0.1436134423875758    epoch : 4800 cost : 0.12450435258780915
epoch : 280 cost : 0.23500316632968704  epoch : 2280 cost : 0.1433443799434821    epoch : 4820 cost : 0.12441883016160842
epoch : 300 cost : 0.23161748053151138  epoch : 2300 cost : 0.1430788042760949    epoch : 4840 cost : 0.12433385503386188
epoch : 320 cost : 0.22842110106943195  epoch : 2320 cost : 0.1428166434074649    epoch : 4860 cost : 0.1242494214563658
epoch : 340 cost : 0.22539850701199451  epoch : 2340 cost : 0.1425578273871763    epoch : 4880 cost : 0.1241655237631484
epoch : 360 cost : 0.2225357220210561   epoch : 2360 cost : 0.1423022882198898    epoch : 4900 cost : 0.12408215636897858
epoch : 380 cost : 0.2198201560086384   epoch : 2380 cost : 0.1420499597960198    epoch : 4920 cost : 0.12399931376792524
epoch : 400 cost : 0.21724045799458294  epoch : 2400 cost : 0.14180077782538758   epoch : 4940 cost : 0.12391699053193134
epoch : 420 cost : 0.21478638182909898  epoch : 2420 cost : 0.14155467977370634   epoch : 4960 cost : 0.12383518130942701
epoch : 440 cost : 0.2124486651915049   epoch : 2440 cost : 0.14131160480175553   epoch : 4980 cost : 0.12375388082396835
epoch : 460 cost : 0.21021892148548119  epoch : 2460 cost : 0.14107149370710015   Accuracy : 0.962
```



cost 값이 0.12가 넘으며 정확도는 이에 반비례해 0.962로 제일 부정확하다.

- Mnist Single Class / target class : 6 / epoch = 5000, learning rate = 0.01

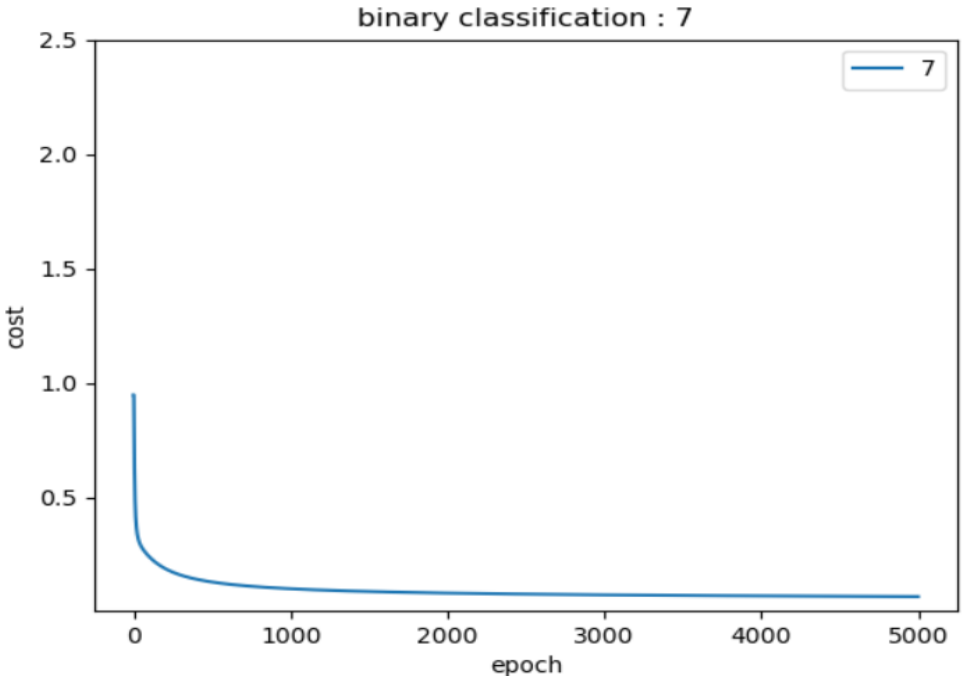
```
epoch : 0 cost : 0.9414862939657264      epoch : 2000 cost : 0.07809681179114213  epoch : 4540 cost : 0.062354149742559956
epoch : 20 cost : 0.3677389142903078     epoch : 2020 cost : 0.07785189483941952  epoch : 4560 cost : 0.06228978823556079
epoch : 40 cost : 0.31286617567685193    epoch : 2040 cost : 0.07761094835197004  epoch : 4580 cost : 0.06222588252239265
epoch : 60 cost : 0.2888575727280783     epoch : 2060 cost : 0.07737386923149754  epoch : 4600 cost : 0.062162427362435185
epoch : 80 cost : 0.27085382104812067    epoch : 2080 cost : 0.07714055802734744  epoch : 4620 cost : 0.0620994175980608
epoch : 100 cost : 0.25542055830983496   epoch : 2100 cost : 0.07691091877333968  epoch : 4640 cost : 0.06203684815296056
epoch : 120 cost : 0.2417788245355594    epoch : 2120 cost : 0.0766848588342533   epoch : 4660 cost : 0.06197471403051009
epoch : 140 cost : 0.22963612612623585   epoch : 2140 cost : 0.07646228876041877  epoch : 4680 cost : 0.061913010312179696
epoch : 160 cost : 0.2188039190348984    epoch : 2160 cost : 0.07624312214991696  epoch : 4700 cost : 0.06185173215598317
epoch : 180 cost : 0.20912265005539185   epoch : 2180 cost : 0.07602727551793546  epoch : 4720 cost : 0.06179087479495871
epoch : 200 cost : 0.20044905082248154   epoch : 2200 cost : 0.07581466817282832  epoch : 4740 cost : 0.061730433535687926
epoch : 220 cost : 0.1926551572525366    epoch : 2220 cost : 0.07560522209851425  epoch : 4760 cost : 0.061670403756857645
epoch : 240 cost : 0.185628365085574     epoch : 2240 cost : 0.07539886184278774  epoch : 4780 cost : 0.06161078090784289
epoch : 260 cost : 0.1792707711979378    epoch : 2260 cost : 0.07519551441125374  epoch : 4800 cost : 0.061551560507334825
epoch : 280 cost : 0.17349790948797886   epoch : 2280 cost : 0.07499510916650547  epoch : 4820 cost : 0.06149273814199621
epoch : 300 cost : 0.16823721364140265   epoch : 2300 cost : 0.07479757773227785  epoch : 4840 cost : 0.06143430946514591
epoch : 320 cost : 0.163426446202859     epoch : 2320 cost : 0.07460285390227811  epoch : 4860 cost : 0.061376270195484105
epoch : 340 cost : 0.15901222511922566   epoch : 2340 cost : 0.07441087355342742  epoch : 4880 cost : 0.06131861611583147
epoch : 360 cost : 0.1549487057955469    epoch : 2360 cost : 0.07422157456326783  epoch : 4900 cost : 0.06126134307191134
epoch : 380 cost : 0.1511964347673084    epoch : 2380 cost : 0.0740348967313103   epoch : 4920 cost : 0.06120444697115695
epoch : 400 cost : 0.14772136940234895   epoch : 2400 cost : 0.07385078170407669  epoch : 4940 cost : 0.06114792378154118
epoch : 420 cost : 0.14449404814890715   epoch : 2420 cost : 0.07366917290367694  epoch : 4960 cost : 0.06109176953043538
epoch : 440 cost : 0.14148889240595985   epoch : 2440 cost : 0.07349001545969003  epoch : 4980 cost : 0.06103598030349875
epoch : 460 cost : 0.1386836209840056    epoch : 2460 cost : 0.07331325614418405  Accuracy : 0.981
```



cost값을 보면 0과 1보다는 크고 2와 4보다는 작다. 정확도는 0.981로 0과 1보다는 낮고 2와 4보다는 높다.

- Mnist Single Class / target class : 7 / epoch = 5000, learning rate = 0.01

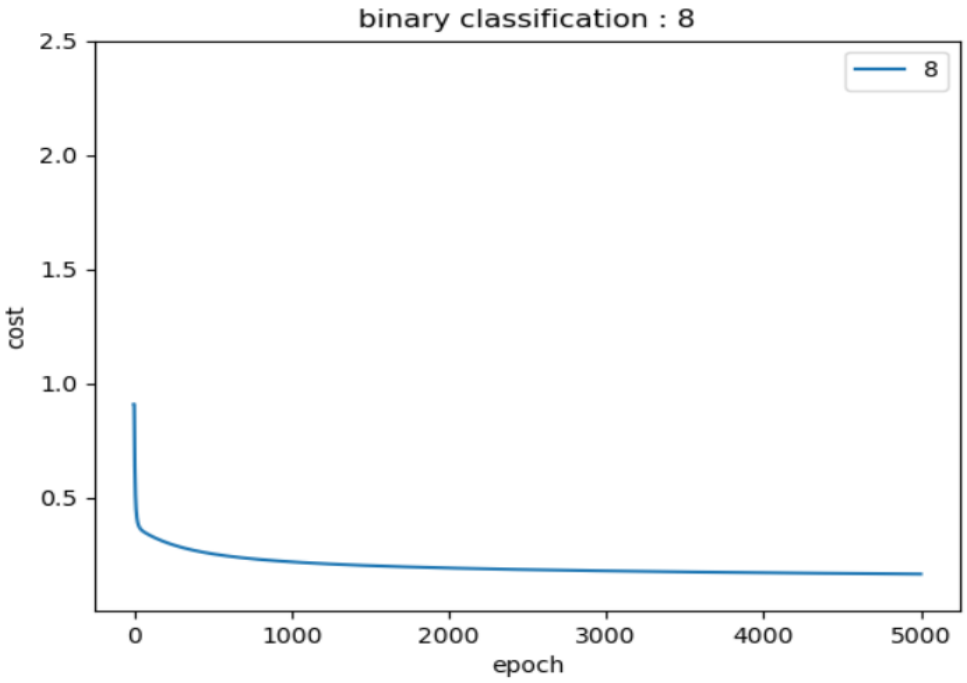
```
epoch : 0 cost : 0.9488334703842606
epoch : 20 cost : 0.3559398475005221
epoch : 40 cost : 0.2985852988951211
epoch : 60 cost : 0.27411334738013077
epoch : 80 cost : 0.25635955880100453
epoch : 100 cost : 0.24149302375009996
epoch : 120 cost : 0.2285718860999389
epoch : 140 cost : 0.21722297941688853
epoch : 160 cost : 0.2072117334772828
epoch : 180 cost : 0.1983500627162746
epoch : 200 cost : 0.1904766274765791
epoch : 220 cost : 0.18345234917946404
epoch : 240 cost : 0.17715831247333905
epoch : 260 cost : 0.17149360422368412
epoch : 280 cost : 0.16637296067653204
epoch : 300 cost : 0.16172444685712792
epoch : 320 cost : 0.15748733426650133
epoch : 340 cost : 0.1536102488966082
epoch : 360 cost : 0.15004959987834474
epoch : 380 cost : 0.1467682679767116
epoch : 400 cost : 0.1437345210024928
epoch : 420 cost : 0.14092112103142787
epoch : 440 cost : 0.13830459074824702
epoch : 460 cost : 0.1358646104010318
epoch : 2000 cost : 0.08322101281251142
epoch : 2020 cost : 0.08300617431908996
epoch : 2040 cost : 0.0827947804712216
epoch : 2060 cost : 0.08258674209082606
epoch : 2080 cost : 0.08238197314193507
epoch : 2100 cost : 0.08218039059146744
epoch : 2120 cost : 0.08198191427737851
epoch : 2140 cost : 0.08178646678378503
epoch : 2160 cost : 0.08159397332260655
epoch : 2180 cost : 0.08140436162133337
epoch : 2200 cost : 0.0812175618165617
epoch : 2220 cost : 0.08103350635292965
epoch : 2240 cost : 0.08085212988715859
epoch : 2260 cost : 0.0806733691968876
epoch : 2280 cost : 0.08049716309401564
epoch : 2300 cost : 0.08032345234230738
epoch : 2320 cost : 0.08015217957900926
epoch : 2340 cost : 0.0799832892402464
epoch : 2360 cost : 0.07981672748999355
epoch : 2380 cost : 0.07965244215242069
epoch : 2400 cost : 0.07949038264740622
epoch : 2420 cost : 0.07933049992909506
epoch : 2440 cost : 0.07917274642726971
epoch : 2460 cost : 0.0790170759914312
epoch : 4560 cost : 0.06922259704300039
epoch : 4580 cost : 0.06916506585535695
epoch : 4600 cost : 0.06910792774185559
epoch : 4620 cost : 0.0690511782376579
epoch : 4640 cost : 0.06899481294857183
epoch : 4660 cost : 0.06893882754961775
epoch : 4680 cost : 0.06888321778365378
epoch : 4700 cost : 0.06882797946001828
epoch : 4720 cost : 0.06877310845320865
epoch : 4740 cost : 0.06871860070159118
epoch : 4760 cost : 0.0686644522061475
epoch : 4780 cost : 0.06861065902923993
epoch : 4800 cost : 0.06855721729341943
epoch : 4820 cost : 0.06850412318024822
epoch : 4840 cost : 0.0684513729291605
epoch : 4860 cost : 0.06839896283634572
epoch : 4880 cost : 0.06834688925365738
epoch : 4900 cost : 0.0682951485875451
epoch : 4920 cost : 0.0682437372980178
epoch : 4940 cost : 0.06819265189762916
epoch : 4960 cost : 0.068141888950476
epoch : 4980 cost : 0.06809144507123734
Accuracy : 0.981
```



cost값이 0.068... 로 6의 cost값과 매우 비슷하고 정확도는 0.981로 6의 정확도 0.981과 일치한다.

- Mnist Single Class / target class : 8 / epoch = 5000, learning rate = 0.01

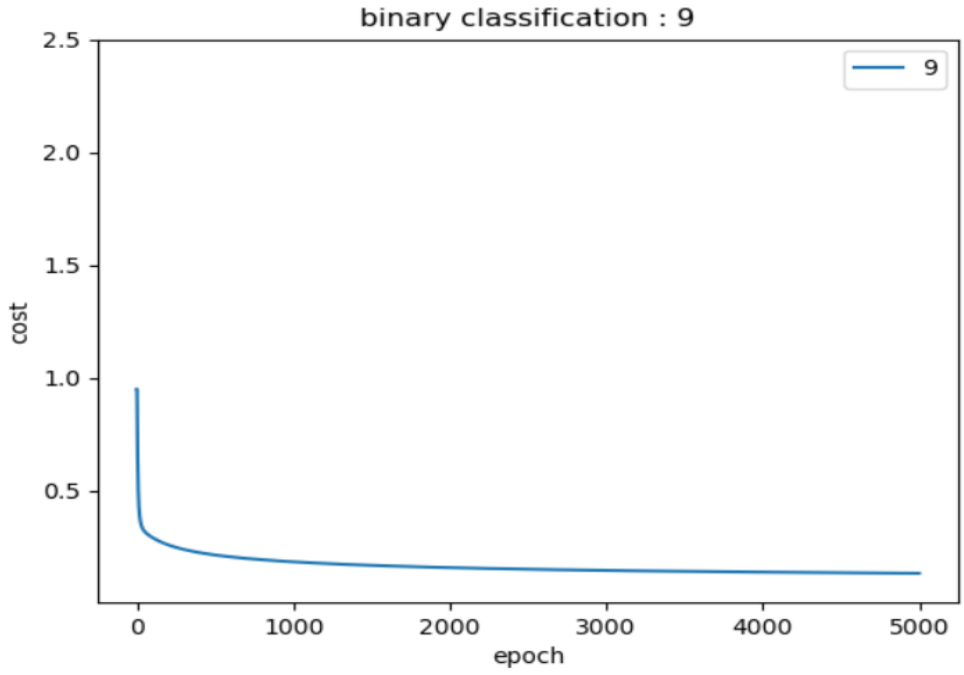
```
epoch : 0 cost : 0.9098609836915442      epoch : 2000 cost : 0.19315400294600857   epoch : 4540 cost : 0.1691278800761375
epoch : 20 cost : 0.40367452970771056    epoch : 2020 cost : 0.1928201018286342    epoch : 4560 cost : 0.1690137200291325
epoch : 40 cost : 0.36510535460659693    epoch : 2040 cost : 0.1924907415996347    epoch : 4580 cost : 0.16890016986692777
epoch : 60 cost : 0.3525126084885111     epoch : 2060 cost : 0.19216581354499543   epoch : 4600 cost : 0.16878722316012196
epoch : 80 cost : 0.3438585181877441     epoch : 2080 cost : 0.19184521254673692   epoch : 4620 cost : 0.16867487357814248
epoch : 100 cost : 0.33633711448306125   epoch : 2100 cost : 0.19152883693173015   epoch : 4640 cost : 0.16856311488731912
epoch : 120 cost : 0.32941563256855433   epoch : 2120 cost : 0.19121658832821187   epoch : 4660 cost : 0.16845194094900184
epoch : 140 cost : 0.3229601281545654    epoch : 2140 cost : 0.19090837152950244   epoch : 4680 cost : 0.1683413457177339
epoch : 160 cost : 0.3169187565328514    epoch : 2160 cost : 0.1906040943645604    epoch : 4700 cost : 0.16823132323943452
epoch : 180 cost : 0.31125835119416545   epoch : 2180 cost : 0.19030366757491357   epoch : 4720 cost : 0.1681218676496514
epoch : 200 cost : 0.3059509334425088    epoch : 2200 cost : 0.19000700469767465   epoch : 4740 cost : 0.16801297317185515
epoch : 220 cost : 0.3009708450848967    epoch : 2220 cost : 0.189714021954211     epoch : 4760 cost : 0.16790463411575887
epoch : 240 cost : 0.29629415865552794   epoch : 2240 cost : 0.189424638144184     epoch : 4780 cost : 0.1677968448756724
epoch : 260 cost : 0.29189854679892724   epoch : 2260 cost : 0.18913877454470374   epoch : 4800 cost : 0.16768959992892699
epoch : 280 cost : 0.2877632252929443    epoch : 2280 cost : 0.18885635481421956   epoch : 4820 cost : 0.16758289383427732
epoch : 300 cost : 0.28386889641579893   epoch : 2300 cost : 0.18857730490093427   epoch : 4840 cost : 0.16747672123041038
epoch : 320 cost : 0.28019768345709734   epoch : 2320 cost : 0.18830155295552076   epoch : 4860 cost : 0.167371076834422
epoch : 340 cost : 0.2767330582648893    epoch : 2340 cost : 0.18802902924784232   epoch : 4880 cost : 0.1672659554403808
epoch : 360 cost : 0.27345976460321014   epoch : 2360 cost : 0.1877596660875012    epoch : 4900 cost : 0.16716135191788617
epoch : 380 cost : 0.27036373952711185   epoch : 2380 cost : 0.18749339774801305   epoch : 4920 cost : 0.16705726121067838
epoch : 400 cost : 0.2674320344547305    epoch : 2400 cost : 0.18723016039440343   epoch : 4940 cost : 0.1669536783352728
epoch : 420 cost : 0.26465273724281635   epoch : 2420 cost : 0.18696989201400352   epoch : 4960 cost : 0.16685059837962654
epoch : 440 cost : 0.26201489628923663   epoch : 2440 cost : 0.18671253235038476   epoch : 4980 cost : 0.1667480165018437
epoch : 460 cost : 0.2595084474493125    epoch : 2460 cost : 0.18645802284014049   Accuracy : 0.945
```



숫자 중 가장 큰 cost값, 가장 낮은 정확도를 가지고 있다. 아마 이것은 feature로 들어온 데이터 중 8의 feature가 가장 특색이 없기 때문으로 예상한다. 8은 모양상 넓은 분포로 그려지다 보니 784개의 feature중 특정 feature에 표시 되지 않고 데이터 마다 다른 양상을 보이기 때문이라고 생각한다.

- Mnist Single Class / target class : 9 / epoch = 5000, learning rate = 0.01

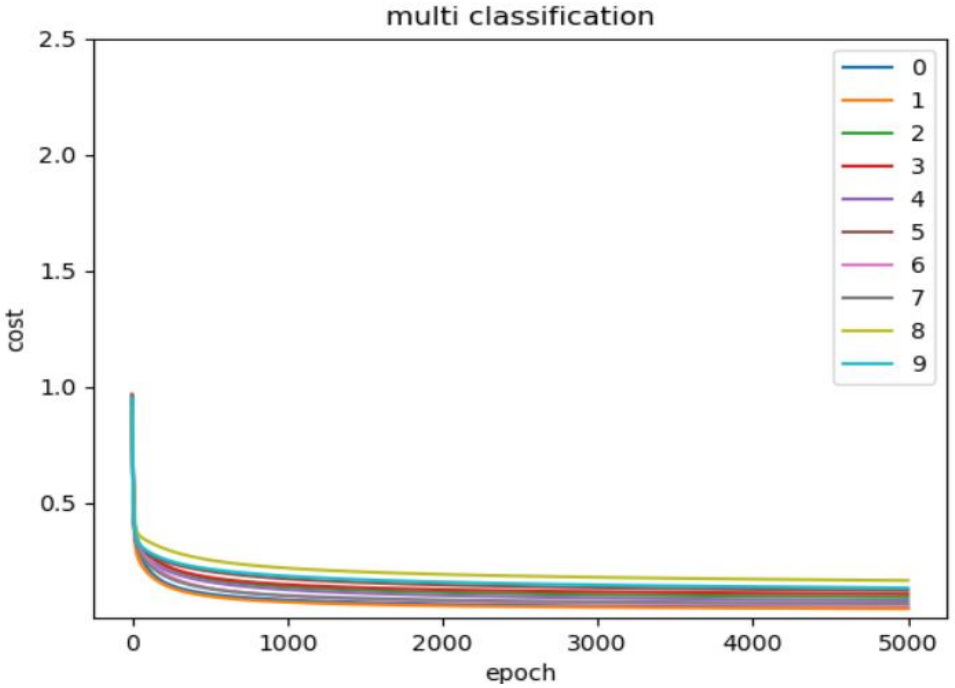
```
epoch : 0 cost : 0.9504026495567801      epoch : 2000 cost : 0.1599399328431941    epoch : 4540 cost : 0.13712843524541227
epoch : 20 cost : 0.3761742965471692     epoch : 2020 cost : 0.15961145703957139   epoch : 4560 cost : 0.13702877276844525
epoch : 40 cost : 0.32903004080428616    epoch : 2040 cost : 0.15928744291196506   epoch : 4580 cost : 0.1369297708259037
epoch : 60 cost : 0.3126877803338256     epoch : 2060 cost : 0.15896779426690924   epoch : 4600 cost : 0.1368314223128123
epoch : 80 cost : 0.3018281648405132     epoch : 2080 cost : 0.15865241776461816   epoch : 4620 cost : 0.13673372022808195
epoch : 100 cost : 0.2928829167009798    epoch : 2100 cost : 0.15834122280968801   epoch : 4640 cost : 0.13663665767259606
epoch : 120 cost : 0.2850420537682284    epoch : 2120 cost : 0.1580341214469891    epoch : 4660 cost : 0.13654022784734676
epoch : 140 cost : 0.2780353651207235    epoch : 2140 cost : 0.15773102826245544   epoch : 4680 cost : 0.13644442405160329
epoch : 160 cost : 0.27172435004656625   epoch : 2160 cost : 0.1574318602884896    epoch : 4700 cost : 0.13634923968112156
epoch : 180 cost : 0.26601125991647506   epoch : 2180 cost : 0.1571365369137393    epoch : 4720 cost : 0.13625466822640395
epoch : 200 cost : 0.2608167569941901    epoch : 2200 cost : 0.15684497979698026   epoch : 4740 cost : 0.1361607032709788
epoch : 220 cost : 0.25607370203561974   epoch : 2220 cost : 0.1565571127848929    epoch : 4760 cost : 0.13606733848973412
epoch : 240 cost : 0.2517247739770949    epoch : 2240 cost : 0.1562728618335459    epoch : 4780 cost : 0.13597456764727792
epoch : 260 cost : 0.24772099958747193   epoch : 2260 cost : 0.1559921549333283    epoch : 4800 cost : 0.13588238459633792
epoch : 280 cost : 0.24402054550091765   epoch : 2280 cost : 0.15571492203720463   epoch : 4820 cost : 0.13579078327619035
epoch : 300 cost : 0.2405876553629499    epoch : 2300 cost : 0.15544109499208336   epoch : 4840 cost : 0.13569975771113202
epoch : 320 cost : 0.23739171926173178   epoch : 2320 cost : 0.15517060747317094   epoch : 4860 cost : 0.13560930200897697
epoch : 340 cost : 0.23440647229474523   epoch : 2340 cost : 0.1549033949210932    epoch : 4880 cost : 0.13551941035958154
epoch : 360 cost : 0.2316093136264789    epoch : 2360 cost : 0.15463939448172165   epoch : 4900 cost : 0.13543007703341448
epoch : 380 cost : 0.2289807330670727    epoch : 2380 cost : 0.15437854494849998   epoch : 4920 cost : 0.13534129638014802
epoch : 400 cost : 0.2265038307210452    epoch : 2400 cost : 0.15412078670720372   epoch : 4940 cost : 0.13525306282728133
epoch : 420 cost : 0.2241639156614962    epoch : 2420 cost : 0.15386606168293174   epoch : 4960 cost : 0.1351653708787782
epoch : 440 cost : 0.22194817091577715   epoch : 2440 cost : 0.15361431328934053   epoch : 4980 cost : 0.1350782151137651
epoch : 460 cost : 0.21984537370329066   epoch : 2460 cost : 0.15336548637986003   Accuracy : 0.957
```

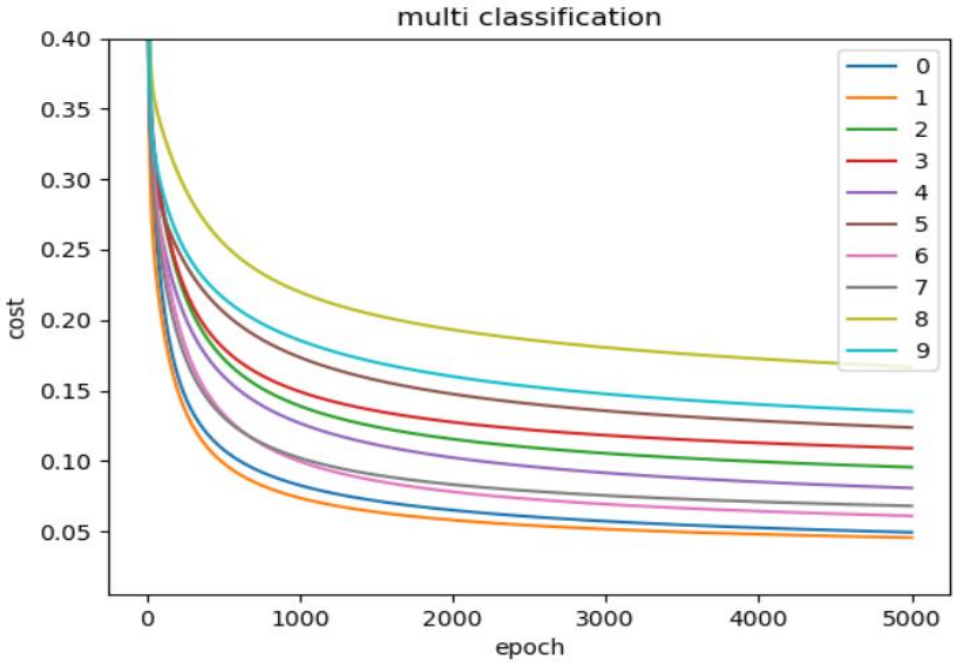


9도 8만큼은 아니지만 8과 모양이 매우 비슷하고 숫자가 넓은 분포로 그려지다보니 특색을 띄기 어렵다. 그렇기 때문에 9도 높은 cost값과 낮은 정확도를 가지고 있는 것으로 생각된다.

- Mnist Multi Class / epoch = 5000, learning rate = 0.01

```
epoch : 0 cost : [0.9169775  0.95048788 0.92615157 0.93320843 0.94640611 0.94561647
 0.93970354 0.93752654 0.93348644 0.94482128]
epoch : 20 cost : [0.36283223 0.32714322 0.37624178 0.38219579 0.35785436 0.35584721
 0.3673834  0.35544587 0.40390392 0.37634552]
epoch : 40 cost : [0.29797807 0.26261061 0.3270889  0.33243753 0.30495485 0.30821232
 0.3126707  0.29854503 0.36461192 0.32934105]
epoch : 60 cost : [0.26463726 0.23510404 0.3071227  0.31173198 0.28478026 0.2935887
 0.2887168  0.2741628  0.3519858  0.31296921]
epoch : 80 cost : [0.23979193 0.21601628 0.2924423  0.29645451 0.27084165 0.28479284
 0.27074223 0.25644733 0.34335984 0.30207014]
epoch : 100 cost : [0.21994741 0.20066677 0.27982332 0.28344478 0.25921865 0.27773963
 0.25532755 0.24160263 0.33587286 0.29308937]
epoch : 120 cost : [0.20379953 0.18772923 0.26857148 0.27199443 0.24897904 0.2714822
 0.24169833 0.22869448 0.3289842  0.28521839]
epoch : 140 cost : [0.19049782 0.17663009 0.25845166 0.26183103 0.23980394 0.26574269
 0.22956413 0.21735256 0.32255875 0.27818659]
epoch : 160 cost : [0.17940477 0.16702089 0.24932607 0.25277744 0.23152959 0.26041785
 0.2187376  0.20734393 0.31654473 0.27185477]
epoch : 180 cost : [0.17003804 0.15864529 0.24108213 0.24468726 0.22403982 0.25545534
 0.20906006 0.19848171 0.31090923 0.26612448]
epoch : 200 cost : [0.1620342  0.15129976 0.23361886 0.23743294 0.21723876 0.25081927
 0.20038883 0.19060549 0.30562454 0.26091575]
epoch : 220 cost : [0.15511883 0.14481875 0.22684515 0.23090341 0.21104353 0.24648014
 0.19259639 0.18357688 0.30066523 0.25616091]
epoch : 240 cost : [0.1490833  0.13906679 0.22067972 0.22500286 0.2053819  0.24241191
 0.18557044 0.1772775  0.2960076  0.25180218]
epoch : 260 cost : [0.14376731 0.13393277 0.21505088 0.21964922 0.20019105 0.23859112
 0.17921329 0.17160684 0.2916295  0.24779021]
epoch : 2160 cost : [0.06338903 0.05663193 0.11337349 0.12558363 0.10010571 0.14501873
 0.0762052  0.08156105 0.19062411 0.15741583]
epoch : 2180 cost : [0.06319997 0.05647451 0.11312794 0.12535727 0.09984339 0.14473122
 0.07598958 0.08137142 0.19032399 0.1571203 ]
epoch : 2200 cost : [0.06301348 0.05631945 0.11288588 0.12513413 0.09958469 0.14444759
 0.07577719 0.0811846  0.19002763 0.15682853]
epoch : 2220 cost : [0.06282951 0.05616669 0.11264724 0.12491416 0.0993295  0.14416776
 0.07556796 0.08100053 0.18973493 0.15654046]
epoch : 2240 cost : [0.06264802 0.05601618 0.11241195 0.12469726 0.09907776 0.14389164
 0.07536182 0.08081915 0.18944583 0.15625601]
epoch : 2260 cost : [0.06246893 0.05586786 0.11217992 0.12448338 0.09882939 0.14361916
 0.07515869 0.08064038 0.18916024 0.15597511]
epoch : 2280 cost : [0.0622922  0.05572168 0.11195107 0.12427245 0.09858432 0.14335023
 0.0749585  0.08046417 0.18887808 0.15569768]
epoch : 2300 cost : [0.06211778 0.0555776  0.11172535 0.12406439 0.09834248 0.1430848
 0.07476119 0.08029046 0.18859929 0.15542366]
epoch : 2320 cost : [0.06194562 0.05543556 0.11150269 0.12385915 0.09810379 0.14282277
 0.07456667 0.0801192  0.18832378 0.15515299]
epoch : 2340 cost : [0.06177567 0.05529551 0.111283   0.12365666 0.09786819 0.14256409
 0.07437491 0.07995031 0.1880515  0.15488559]
epoch : 2360 cost : [0.06160788 0.05515742 0.11106624 0.12345686 0.09763561 0.14230869
 0.07418582 0.07978376 0.18778237 0.15462141]
epoch : 2380 cost : [0.06144221 0.05502124 0.11085233 0.1232597  0.097406   0.1420565
 0.07399935 0.07961949 0.18751633 0.15436038]
epoch : 2400 cost : [0.06127861 0.05488693 0.11064122 0.12306512 0.09717928 0.14180745
 0.07381544 0.07945744 0.18725331 0.15410244]
epoch : 2420 cost : [0.06111704 0.05475444 0.11043284 0.12287306 0.09695541 0.14156149
 0.07363403 0.07929758 0.18699325 0.15384754]
epoch : 4800 cost : [0.04985433 0.04592487 0.09619316 0.1097164  0.08149155 0.12452198
 0.06153287 0.06853153 0.16771872 0.13585355]
epoch : 4820 cost : [0.04979711 0.04588212 0.09612216 0.10965025 0.08141423 0.12443651
 0.06147414 0.0684785  0.16761201 0.13576191]
epoch : 4840 cost : [0.04974025 0.04583966 0.09605161 0.10958452 0.08133741 0.12435159
 0.0614158  0.06842581 0.16750582 0.13567085]
epoch : 4860 cost : [0.04968374 0.04579749 0.09598151 0.10951918 0.08126108 0.1242672
 0.06135785 0.06837346 0.16740017 0.13558036]
epoch : 4880 cost : [0.04962758 0.04575559 0.09591186 0.10945425 0.08118524 0.12418336
 0.06130028 0.06832145 0.16729503 0.13549043]
epoch : 4900 cost : [0.04957177 0.04571397 0.09584264 0.10938971 0.08110989 0.12410004
 0.0612431  0.06826977 0.16719042 0.13540107]
epoch : 4920 cost : [0.0495163  0.04567263 0.09577385 0.10932557 0.08103502 0.12401725
 0.06118629 0.06821842 0.16708631 0.13531225]
epoch : 4940 cost : [0.04946116 0.04563156 0.0957055  0.10926181 0.08096063 0.12393498
 0.06112985 0.0681674  0.16698272 0.13522398]
epoch : 4960 cost : [0.04940636 0.04559076 0.09563757 0.10919844 0.0808867  0.12385322
 0.06107379 0.06811669 0.16687962 0.13513626]
epoch : 4980 cost : [0.04935189 0.04555022 0.09557007 0.10913545 0.08081324 0.12377197
 0.06101808 0.06806631 0.16677703 0.13504907]
Accuracy : 0.892
```



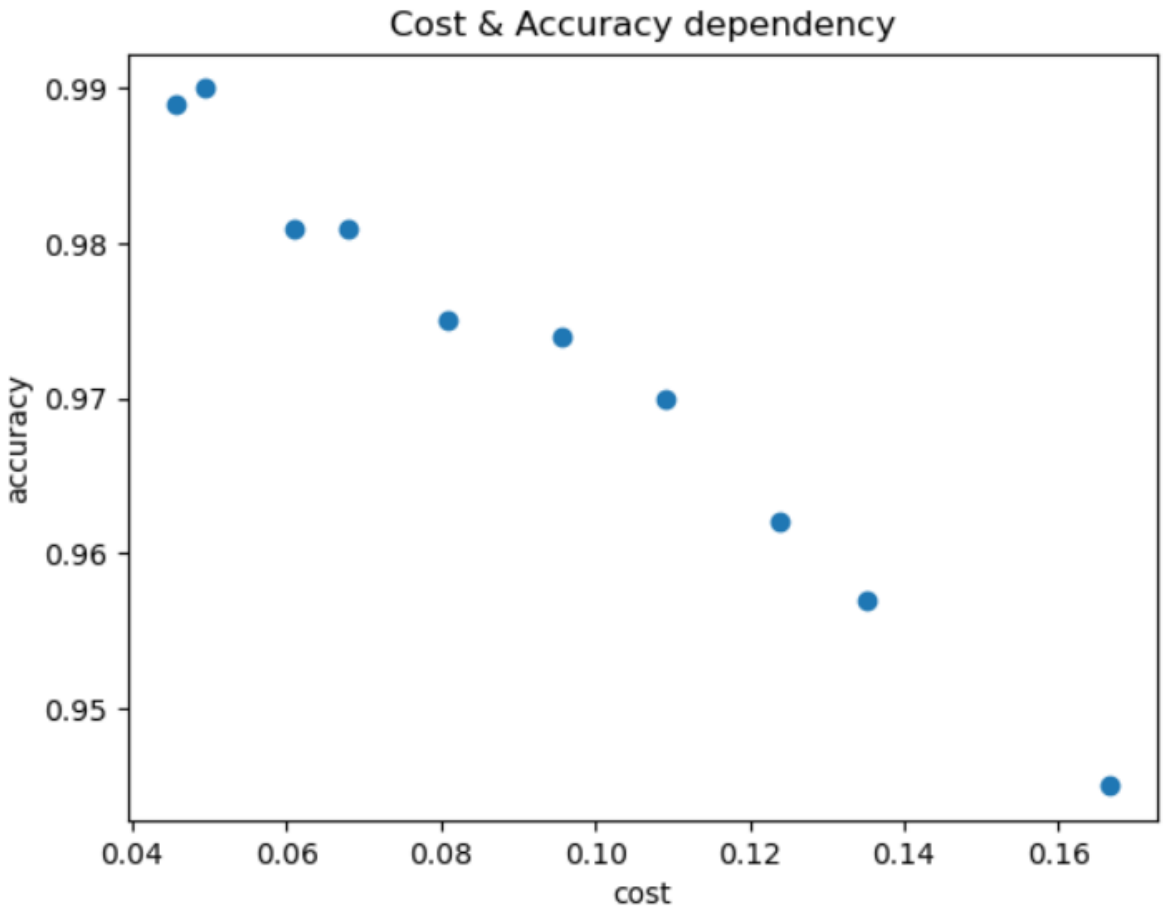


기존 multi classification 그래프를 보게 되면 cost 값에 차이가 잘 보이지 않기 때문에 아래의 그래프와 같이 0 ~ 0.4 부분을 확대해 그래프를 표시했다. 확실히 숫자간 cost 값의 차이가 보인다. 정확도는 0.892로 매우 낮게 나왔는데 이것은 지난번 knn 알고리즘의 정확도에 비해 저조한 성적이다. 각각의 binary classification에 대해서는 성공적인 정확도를 보여주었지만 이것이 multi classification에서의 정확도로 이어지는 것이 아니라는 것을 알 수 있다. iris data의 경우 binary classification의 정확도의 평균보다 multi classification의 정확도가 높았지만 mnist data는 반대이다. 이것은 분류해야하는 집단의 종류가 많아 질수록 정확도가 떨어진 다는 것을 의미하며 당연히 알고리즘 입장에서 상정해야 하는 답의 선택지가 많아 질수록 정확도는 수학적으로도 떨어지

게 될 것이다. 또한 multi classification에서는 가장 높은 hypothesis를 답으로 상정하기 때문에 예기치 못한 test data에 대해서는 각자 저조한 hypothesis를 내더라도 누군가는 답으로 뽑히기 때문에 나는 그 숫자가 아니라고 맞추는 binary classification보다 맞출 확률이 낮다. binary classification에서는 0.5가 되지 않는다면 0이라는 결과만 반환하면 되지만 multi는 누구의 것에 해당하는 것인지 정확히 해야 하기 때문이다.

마지막으로 mnist data 에서 class 별로 cost 값과 정확도는 정확히 반비례 관계인지 두 값을 기준으로 그래프를 출력해보자. 코드는 아래와 같다.

```
plt.scatter(cos, acc)  # x축 epoch, y축 cost값
plt.xlabel('cost')  # xlabel : epoch
plt.ylabel('accuracy')  # ylabel : cost
plt.title('Cost & Accuracy dependency')  # 그래프 제목 설정
plt.show()  # 출력
```



mnist data 의 경우 test data set 이 10000 개이므로 표본이 충분하다고 생각할 수 있고 100%는 아니지만 cost 와 accuracy 간 유의미한 반비례 관계를 관찰할 수 있다. 즉, mnist data 에서 cost 값과 정확도는 반비례한다고 할 수 있다.