KNN and Logistic Regression with MNIST Data

기계학습기초
Homework #2

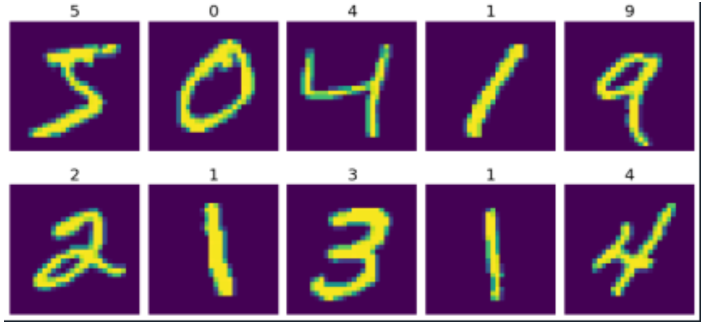
개요

- MNIST data
- KNN
- Logistic Regression

KNN, Logistic Regression에 대해 이해하고 이를 mnist data에 적용하여

multiple classification을 수행하기.

MNIST Data 특성



Input feature data type = image
Input feature dimension = 784(28x28)
Training data size = 60,000
Test data size = 10,000

머신러닝 최고 Guru중 한명인 뉴욕대 교수 Yann Lecun이 제공하는 데이터 셋

숫자 0~9 까지의 손글씨 이미지의 집합
학습데이터와 테스트데이터로 구성

size-normalized & centered
사이즈 : 28x28

패턴인식이나 기계학습 기술을 적용하기 위해 사용할 수 있는 최적의 이미지 셋
preprocessing이나 formatting이 모두 완료 되었기 때문.

Data load

Dataset 폴더를 현재 프로젝트의 부모 폴더에 위치(제공된 폴더 그대로)

```
import sys, os
sys.path.append(os.pardir)
# 부모 디렉토리에서 import할 수 있도록 설정
```

```
import numpy as np
from dataset.mnist import load_mnist
# mnist data load할 수 있는 함수 import
```

```
from PIL import Image
# python image processing library
# python 버전 3.x 에서는 pillow package install해서 사용
```

Data load

```
(x_train, t_train), (x_test, t_test) = \
 load_mnist(flatten=True, normalize=False)
# training data, test data
# flatten: 이미지를 1차원 배열로 읽음
# normalize: 0~1 실수로. 그렇지 않으면 0~255

image = x_train[0]
label = t_train[0]
# 첫번째 데이터

print(label)
print(image.shape)
```

Data Visualization

```
def img_show(img):
 pil_img = Image.fromarray(np.uint8(img))
 pil_img.show()
# image를 unsigned int로

image = image.reshape(28,28)
# 1차원 —> 2차원 (28x28)

print(image.shape)
img_show(image)
```

K Nearest Neighbors(KNN)

Classification using KNN

K-Nearest Neighbor 알고리즘 input

1. 784개 input을 그대로 사용

2. 최적의 k값 찾아보기

Classification using KNN

1. 784개 input을 그대로 사용

결과가 그리 나쁘지는 않음.
결과 계산에 오랜 시간이 소요됨.

- **Majority vote, Weighted majority vote 두가지 방법으로 output 산출**
- **KNN의 특성 : 대부분의 계산이 학습보다는 inference에 소요.**

ex) learn : 0.08 sec
inference : 27.32 sec

Output Example

```
6 6
2 2
7 7
8 8
4 4
7 7
3 3
6 6
1 1
3 3
6 6
9 9
3 3
1 1
4 4
1 1
7 7
6 6
9 9
Fit time :  14.309977293014526
0.9688
```

accuracy = 0.9688
(10000개 test data 중 100개 사용)

Optimal K

2. 최적의 K값 도출하기

- K값을 수정해보며 Accuracy 토대로 최적의 k를 heuristic하게 도출
- 분석 결과를 보고서에 첨가

Logistic Regression

Classification using Logistic Regression

동일한 MNIST Data 사용

-data 수 : m
-feature 수(입력 차수) : n
- 여기에 bias term 추가

Classification using Logistic Regression

- **Fit θ parameters**
- **data**

Training set: $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \cdots, (x^{(m)}, y^{(m)})\}$

m examples $\quad x \in \begin{bmatrix} x_0 \\ x_1 \\ \cdots \\ x_n \end{bmatrix} \quad x_0 = 1, y \in \{0, 1\}$

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

Logistic regression cost function and gradient descent

- **cost function**

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

- **gradient descent**

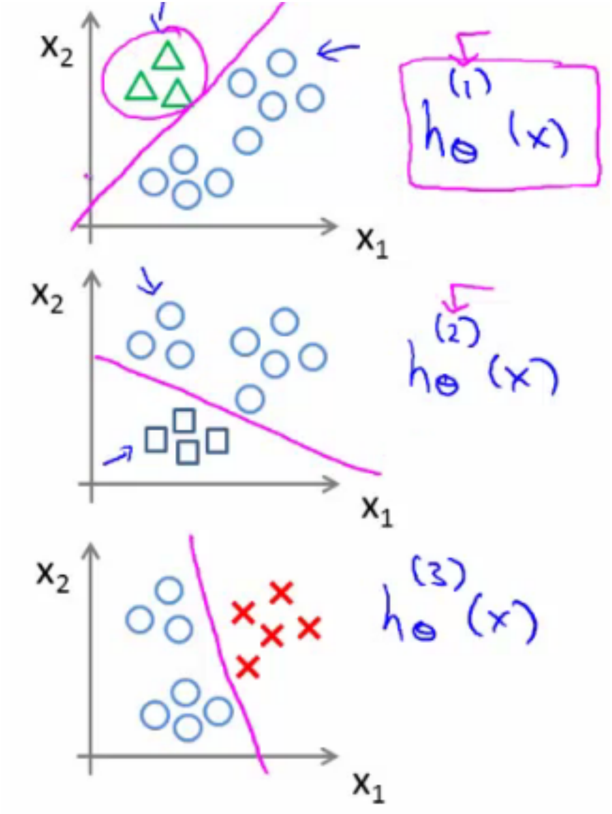
Repeat $\{$

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

(simultaneously update all θ_j)

$\}$

Multiclass classification



- Overall
Train a logistic regression classifier hθ
(i)(x) for each
class i to predict the probability that y = i
- On a new input, x to make a prediction, pick the class i that maximizes the probability that hθ
(i)(x) = 1

Multi-class classification : introduction

$$y = \begin{pmatrix} 1^1 \\ y^2 \\ \vdots \\ y^m \end{pmatrix}$$

y denotes label info
label = [5 8 4 …]

방법
1.logistic regression class instance 를 target class 수 만큼 만들어 따로 학습

class LogisticRegression(...):

target0 = LogisticRegression(...)
target1 = LogisticRegression(...)

Multi-class classification : introduction

방법

1.logistic regression class instance 를 target class 수 만큼 만들어 따로 학습

class LogisticRegression(...):

target0 = LogisticRegression(...)
target1 = LogisticRegression(...)

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

Repeat $\{$

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

(simultaneously update all θ_j)

$\}$

Target class가 달라지면 바뀌는 부분:

y, h()

그러나… Remember that Vector/Matrix Calculation is more efficient

Multi-class classification : input Data, Weight Parameter

$$X = \begin{pmatrix} x_0^1 & x_1^1 & \cdots & x_n^1 \\ x_0^2 & x_1^2 & \cdots & x_n^2 \\ \vdots & \vdots & \ddots & \vdots \\ x_0^m & x_1^m & \cdots & x_n^m \end{pmatrix}$$

$$y = \begin{pmatrix} y^{1,1} & y^{1,2} & \cdots & y^{1,t} \\ y^{2,1} & y^{2,2} & \cdots & y^{2,t} \\ \vdots & \vdots & \ddots & \vdots \\ y^{m,1} & y^{m,2} & \cdots & y^{m,t} \end{pmatrix}$$

y[:,k]: one-hot encoding

$$w = \begin{pmatrix} w_0^1 & w_0^2 & \cdots & w_0^t \\ w_1^1 & w_1^2 & \cdots & w_1^t \\ \vdots & \vdots & \ddots & \vdots \\ w_n^1 & w_n^2 & \cdots & w_n^t \end{pmatrix}$$

np.dot(X, w) —> (m x t) 개의 $\theta^T x$

(m x n) * (n x t) = (m x t)
h(np.dot(X,w))—>(m x t) 개의 logistic regression output:
가장 높은 값의 class 선택

Multi-class classification : Cost function

$$X = \begin{pmatrix} x_0^1 & x_1^1 & \cdots & x_n^1 \\ x_0^2 & x_1^2 & \cdots & x_n^2 \\ \vdots & \vdots & \ddots & \vdots \\ x_0^m & x_1^m & \cdots & x_n^m \end{pmatrix} \quad y = \begin{pmatrix} y^{1,1} & y^{1,2} & \cdots & y^{1,t} \\ y^{2,1} & y^{2,2} & \cdots & y^{2,t} \\ \vdots & \vdots & \ddots & \vdots \\ y^{m,1} & y^{m,2} & \cdots & y^{m,t} \end{pmatrix} \quad w = \begin{pmatrix} w_0^1 & w_0^2 & \cdots & w_0^t \\ w_1^1 & w_1^2 & \cdots & w_1^t \\ \vdots & \vdots & \ddots & \vdots \\ w_n^1 & w_n^2 & \cdots & w_n^t \end{pmatrix}$$

np.dot(X, w) —> m개의 $\theta^T x$

(m x n) * (n x t) = (m x t)

h(np.dot(X,w)) —> (m x t)개의 logistic regression output

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

y : m x t, h(): m x t —> 벡터연산으로 m x t 개 data 에 대한 계산을 동시에

Multi-class classification : Gradient Descent

Repeat {

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

(simultaneously update all θ_j)

}

h(): m x t,   y: m x t, x_j: m —> numpy broadcast does not work
Reshape x_j: $xx_j = x_j$.reshape(m,1) 실행 후에는 가능
—> np.sum(⋯, axis=0)   t: 하나의 θ_j 에 대해서
θ (위에는 W로 표기한 행렬): n x t

t개의 h( )중 가장 큰 값을 가진 class 선택

Logistic Regression Class

Attributes

x : input data
y : target output
w : weights
…

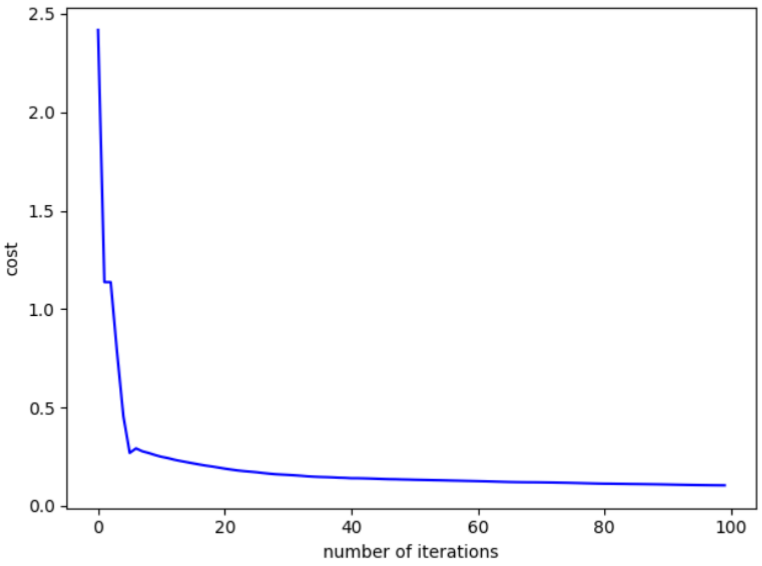
Methods

__init__() # 예시로 생성자가 작성되어있음. 수정 가능
cost()
learn()
predict()
…

MNIST Output Example: Single Class - target class 0

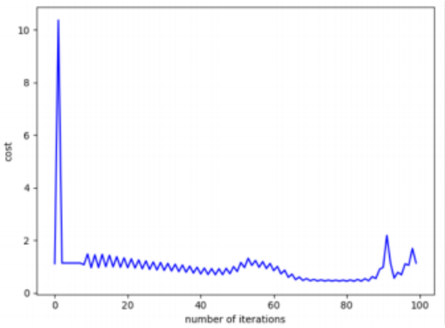
epoch: 0 cost: 2.418264237232712
epoch: 1 cost: 1.136508612695375
epoch: 2 cost: 1.136508612695375
epoch: 3 cost: 0.779711618995576
…
epoch: 94 cost: 0.10530206032889827
epoch: 95 cost: 0.10487256172926512
epoch: 96 cost: 0.1044720380659441
epoch: 97 cost: 0.10404782125472133
epoch: 98 cost: 0.10379578992895847
epoch: 99 cost: 0.10351212094793127

Accuracy = 0.987

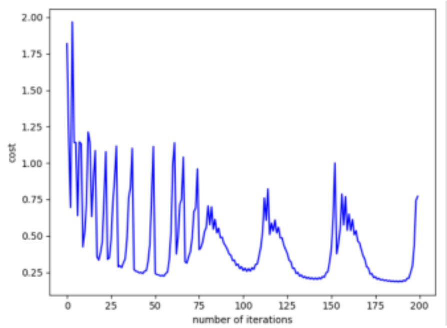


MNIST Output Example: Single Class - target class 9

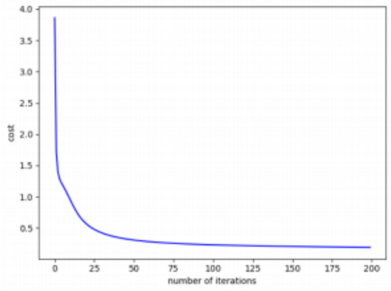
lr=0.005, epoch=100
Accuracy = 0.793

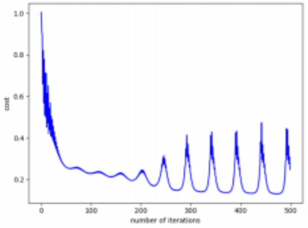


lr = 0.0001, epoch=200
Accuracy = 0.915



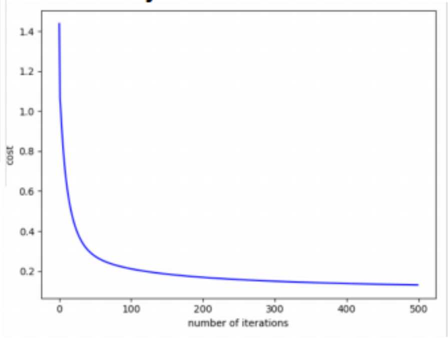
lr=0.00001. epoch=200
Accuracy = 0.813





lr=0.00005

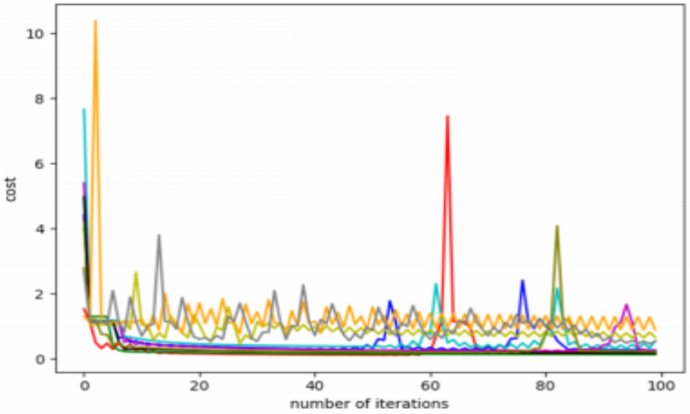
lr=0.00001. epoch=500
Accuracy = 0.823



MNIST Output Example: Multi-class

epoch: 0   cost: [1.52504087 4.0049692 4.40313771
7.66589674 5.40073341 4.17625397
 4.96748467 2.78740626 1.30071932 2.65993052]
epoch: 1   cost: [1.13650861 1.29366018 1.14322449
1.17642012 1.12096615 1.04018372
 1.1355492 1.20213234 1.12270775 1.14149755]
...
epoch: 98   cost: [0.16887424 0.1245994 0.25854562
0.32175146 0.237849   0.8091529
 0.16591243 0.21307945 1.27820775
epoch: 99   cost: [0.16710608 0.124
0.5491724 0.2419721 0.65435579
 0.16565944 0.2110298 0.88324233 (

Accuracy = 0.847

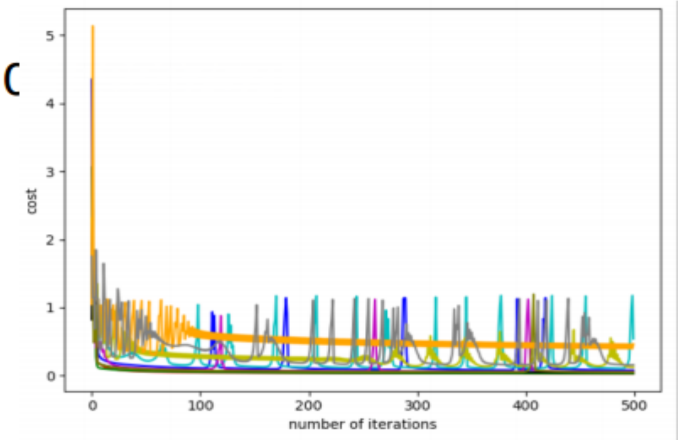


MNIST Output Example: Multiple Class

...

epoch: 497 cost: [0.03082739 0.03206372 0.08053439
0.97133602 0.0603551 0.12609442
0.04805654 0.05405963 0.46484578 0.15769116]
epoch: 498 cost: [0.03081144 0.03204918 0.0804409
1.17628992 0.06030812 0.12226177
0.048031 0.05402784 0.39609652 0.15534708]
epoch: 499 cost: [0.03079554 0.03203469 0.08034915
0.54257252 0.06026169 0.12419998
0.04800554 0.05399661 0.46455226 (

Accuracy = 0.882



주의할 점

- cost function J()값의 추이를 볼 것
- learning rate, epoch를 여러 값으로 바꾸어가며 학습해 볼 것

주의할 점

- for MNIST data: overflow with np.exp( ) - sigmoid function
  - 1) ignore runtime warning “OverflowError”
  - 2) update sigmoid
    - np.log(np.finfo(type(0.1)).max)

you can ignore it. overflow happens when the result of exp(-value) exceeds the maximum number representable by value’s floating point data type format.

so, you can prevent the overflow by checking if value is too small

EX)

```
def sigmoid(value):
    if -value > np.log(np.finfo(type(value)).max):
        return 0.0
    a = np.exp(-value)
    return 1.0/ (1.0 + a)
```

Submission

- Source code (with comments) files
  - KNN class python file
  - LogisticRegression class python file
  - KNN 학습 및 테스트 python file
  - LogisticRegression 학습 및 테스트 python file
- 결과 보고서 pdf file (output 결과 포함)—> 하나의 zip 파일로 압축해서 제출

- Due
  - date: 11월 2일 23시
  - Late penalty: 20% per day