Some answers are written in the notebook submitted separately.

1.

a) Refer to notebook

b) Refer to notebook

import numpy as np

import matplotlib.pyplot as plt

def func(x,y):

    return 100\*(y-x\*\*2)\*\*2 + (1-x)\*\*2

# create two one-dimensional grids using linspace

x = np.linspace(-5, 5, 50)

y = np.linspace(-5, 5, 50)

# combine the two one-dimensional grids into one two-dimensional grid

X, Y = np.meshgrid(x,y)

# evaluate the function at each element of the two-dimensional grid

Z = func(X, Y)

# create plot

fig = plt.figure(figsize=(7,7))

ax = plt.axes(projection='3d')

ax.plot\_surface(X, Y, Z, cmap='viridis')

plt.show()

Chart, surface chart

Description automatically generated

c)

k=0, [-1.2 1. ]

k=1, [-1.1752809 1.38067416]

k=2, [ 0.76311487 -3.17503385]

k=3, [0.76342968 0.58282478]

k=4, [0.99999531 0.94402732]

k=5, [0.9999957 0.99999139]

k=6, [1. 1.]

k=7, [1. 1.]

import numpy as np

start = np.array([-1.2,1])

v = start

def nab(x, y):

    return np.array([-400\*(y-(x\*\*2))\*x - 2\*(1-x), 200\*(y-x\*\*2)])

def hes(x, y):

    return np.array([[1200\*(x\*\*2)-400\*y+2, -400\*x], [-400\*x, 200]])

new\_nab = nab(v[0], v[1])

print(f'k=0, {v}')

iteration = 1

while not np.linalg.norm(new\_nab, ord=2) <= 10\*\*(-6):

    new\_nab = nab(v[0], v[1])

    new\_hes = hes(v[0], v[1])

    v = v - np.linalg.inv(new\_hes) @ new\_nab

    print(f'k={iteration}, {v}')

    iteration += 1

2.a) Refer to notebook

b) Refer to notebook

c) Refer to notebook

d)

import pandas as pd

import numpy as np

from sklearn.preprocessing import StandardScaler

from sklearn.model\_selection import train\_test\_split

df=pd.read\_csv('songs.csv', sep=',', header=0)

# I

df.drop(columns=['Artist Name', 'Track Name', 'key', 'mode', 'time\_signature', 'instrumentalness'], inplace=True)

# II

df.drop(df[(df.Class != 5) & (df.Class != 9)].index, inplace=True)

df['Class'].replace([5, 9], [1, 0], inplace=True)

# III

df.dropna(axis=0, inplace=True)

# IV

X\_train, X\_test, Y\_train, Y\_test = train\_test\_split(df.drop(columns='Class'), df['Class'], test\_size=0.3, random\_state=23)

# V

scaler = StandardScaler()

scaler.fit(X\_train)

X\_train = scaler.transform(X\_train)

X\_test = scaler.transform(X\_test)

# VI

print('first row X\_train:',X\_train[0][0:3])

print('last row X\_train:',X\_train[-1][0:3])

print('first row X\_test:',X\_test[0][0:3])

print('last row X\_test:',X\_test[-1][0:3])

print('first row Y\_train:',Y\_train.iloc[0])

print('last row Y\_train:',Y\_train.iloc[-1])

print('first row Y\_test:',Y\_test.iloc[0])

print('last row Y\_test:',Y\_test.iloc[-1])

first row X\_train: [-0.93555843 0.67519298 1.3849985 ]

last row X\_train: [-1.13301479 -1.09458877 0.96702449]

first row X\_test: [-0.29382524 1.36005105 0.26306826]

last row X\_test: [-0.29382524 -1.05390413 -1.34833155]

first row Y\_train: 0

last row Y\_train: 1

first row Y\_test: 0

last row Y\_test: 1

e)

final train loss: 0.3142969702921616

test loss: 0.31750699444279507

import numpy as np

from sklearn.metrics import log\_loss

import matplotlib.pyplot as plt

data = \_\_import\_\_('2d')

def sigmoid(x):

    # logistic sigmoid

    return np.exp(-np.logaddexp(0, -x))

def loss(gamma, X, y, lam):

    # gamma has first coordinate = beta0 = intercept, and second coordinate = beta

    norm\_beta\_sq = np.linalg.norm(gamma[1:], ord=2)\*\*2

    z = np.dot(X, gamma[1:]) + gamma[0]

    sig\_z = sigmoid(z)

    return lam \* log\_loss(y, sig\_z, normalize=True) + 0.5 \* norm\_beta\_sq

def nab\_loss(gamma, X, y, lam):

    n = X.shape[0]

    summ = np.zeros(p+1)

    for i in range(0,n):

        summ += (y[i] - sigmoid(np.dot(gamma, np.insert(X[i], 0, 1)))) \* np.insert(X[i], 0, 1)

    return np.insert(gamma[1:], 0, 0) - (lam / n) \* summ

p = data.X\_train.shape[1]

gamma = np.zeros(p + 1)

lam = 0.5

alpha = 1

a = 0.5

b = 0.8

epochs\_lim = 60

epochs = np.arange(1, epochs\_lim+1)

step\_sizes = np.full(epochs\_lim, -1.0)

losses = np.full(epochs\_lim, -1.0)

for ep in range(1, epochs\_lim+1):

    cur\_nab\_loss = nab\_loss(gamma, data.X\_train, data.Y\_train, lam)

    cur\_loss = loss(gamma, data.X\_train, data.Y\_train, lam)

    if loss(gamma - alpha \* cur\_nab\_loss, data.X\_train, data.Y\_train, lam) > cur\_loss - a \* alpha \* np.linalg.norm(cur\_nab\_loss, ord=2)\*\*2:

        alpha = alpha \* b

    step\_sizes[ep-1] = alpha

    # Update equation

    gamma = gamma - alpha \* cur\_nab\_loss

    losses[ep-1] = loss(gamma, data.X\_train, data.Y\_train, lam)

fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(10,10))

def plot(x, y, xlabel, ylabel, title, loc):

    axes[loc].scatter(x, y)

    axes[loc].set\_title(title)

    axes[loc].set\_xlabel(xlabel)

    axes[loc].set\_ylabel(ylabel)

plot(epochs, step\_sizes, 'epoch', 'step size', 'Change of step size over each epoch', 0)

plot(epochs, losses, 'epoch', 'losses', 'Change of losses over each epoch', 1)

print('final train loss:', losses[-1])

print('test loss:', loss(gamma, data.X\_test, data.Y\_test, lam))

plt.savefig("2e.png", dpi=300)

plt.show()

Chart

Description automatically generated

f) I’m unable to make this work. The inverse of hessian keeps blowing up to inf in numpy when doing update for newton’s method. Code is given below. If you set epochs\_lim = 3, you will see that log loss of train (and test) data blows very quickly to inf.

import numpy as np

from sklearn.metrics import log\_loss

import matplotlib.pyplot as plt

data = \_\_import\_\_('2d')

def sigmoid(x):

    # logistic sigmoid

    return np.exp(-np.logaddexp(0, -x))

def loss(gamma, X, y, lam):

    # gamma has first coordinate = beta0 = intercept, and second coordinate = beta

    norm\_beta\_sq = np.linalg.norm(gamma[1:], ord=2)\*\*2

    z = np.dot(X, gamma[1:]) + gamma[0]

    sig\_z = sigmoid(z)

    return lam \* log\_loss(y, sig\_z, normalize=True) + 0.5 \* norm\_beta\_sq

def nab\_loss(gamma, X, y, lam):

    n = X.shape[0]

    summ = np.zeros(p+1)

    for i in range(0,n):

        summ += (y[i] - sigmoid(np.dot(gamma, np.insert(X[i], 0, 1)))) \* np.insert(X[i], 0, 1)

    return np.insert(gamma[1:], 0, 0) - (lam / n) \* summ

def hes\_loss(gamma, X, y, lam):

    n = X.shape[0]

    summ = np.zeros((p+1, p+1))

    for i in range(0,n):

        # appending the extra 1 as first element of each x datapoint

        xi = np.insert(X[i], 0, 1)

        summ += sigmoid(np.dot(gamma, xi)) \* (1 - sigmoid(np.dot(gamma, xi))) \* np.outer(xi, np.transpose(xi))

    return (lam / n) \* summ

p = data.X\_train.shape[1]

gd\_gamma = np.zeros(p + 1)

newt\_gamma = np.zeros(p + 1)

lam = 0.5

alpha = 1

a = 0.5

b = 0.8

epochs\_lim = 60

epochs = np.arange(1, epochs\_lim+1)

step\_sizes = np.full(epochs\_lim, -1.0)

gd\_losses = np.full(epochs\_lim, -1.0)

newt\_losses = np.full(epochs\_lim, -1.0)

# Log regression with gradient descent

for ep in range(1, epochs\_lim+1):

    cur\_nab\_loss = nab\_loss(gd\_gamma, data.X\_train, data.Y\_train, lam)

    cur\_loss = loss(gd\_gamma, data.X\_train, data.Y\_train, lam)

    if loss(gd\_gamma - alpha \* cur\_nab\_loss, data.X\_train, data.Y\_train, lam) > cur\_loss - a \* alpha \* np.linalg.norm(cur\_nab\_loss, ord=2)\*\*2:

        alpha = alpha \* b

    step\_sizes[ep-1] = alpha

    # Update equation

    gd\_gamma = gd\_gamma - alpha \* cur\_nab\_loss

    gd\_losses[ep-1] = loss(gd\_gamma, data.X\_train, data.Y\_train, lam)

# Log regression with newton's method

for ep in range(1, epochs\_lim+1):

    cur\_nab\_loss = nab\_loss(newt\_gamma, data.X\_train, data.Y\_train, lam)

    cur\_loss = loss(newt\_gamma, data.X\_train, data.Y\_train, lam)

    if loss(newt\_gamma - alpha \* cur\_nab\_loss, data.X\_train, data.Y\_train, lam) > cur\_loss - a \* alpha \* np.linalg.norm(cur\_nab\_loss, ord=2)\*\*2:

        alpha = alpha \* b

    step\_sizes[ep-1] = alpha

    # Update equation

    newt\_gamma = newt\_gamma - alpha \* (np.linalg.inv(hes\_loss(newt\_gamma, data.X\_train, data.Y\_train, lam)) @ cur\_nab\_loss)

    newt\_losses[ep-1] = loss(newt\_gamma, data.X\_train, data.Y\_train, lam)

fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(10,10))

axes[0].scatter(epochs, step\_sizes)

axes[0].set\_xlabel('epoch')

axes[0].set\_ylabel('step size')

axes[0].set\_title('Change of step size over each epoch')

axes[1].scatter(epochs, gd\_losses, color='red', label='gradient descent')

axes[1].scatter(epochs, newt\_losses, color='blue', label='newton\'s method')

axes[1].set\_xlabel('epoch')

axes[1].set\_ylabel('loss')

axes[1].set\_title('Comparing GD and newtown\'s method on log-loss')

axes[1].legend(loc='upper right')

print('final train loss:', loss(newt\_gamma, data.X\_train, data.Y\_train, lam))

print('test loss:', loss(newt\_gamma, data.X\_test, data.Y\_test, lam))

plt.savefig("2f.png", dpi=300)

plt.show()

g)

Although Newton’s method converges much quicker than GD, it does have some disadvantages that makes it not as popular in machine learning. The main disadvantage is that it requires computing the second derivative of the function we want to minimise. The second derivative may often be intractable, or comes with a very high computation cost. The higher computation cost is because whilst computing the first derivative has p values (p = the number of features), the second order derivative has p^2 values.

3.a) Refer to notebook

b) Refer to notebook

import numpy as np

import matplotlib.pyplot as plt

SD = 1

def MLE\_bias\_estimator(n):

    return (2\*SD\*\*4)\*(n-1)/n\*\*2

def MLE\_var\_estimator(n):

    return SD\*\*2/n

def new\_bias\_estimator(n):

    return 0

def new\_var\_estimator(n):

    return (2\*SD\*\*4)/(n-1)

MLE\_bias\_estimator = np.vectorize(MLE\_bias\_estimator)

MLE\_var\_estimator = np.vectorize(MLE\_var\_estimator)

new\_bias\_estimator = np.vectorize(new\_bias\_estimator)

new\_var\_estimator = np.vectorize(new\_var\_estimator)

fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(10,10))

fig.suptitle('Comparing bias and variance of MLE and new estimator')

def plot(xrange, func1, func2, ylabel, title, x):

    axes[x].plot(xrange, func1(xrange), color='red', label='MLE')

    axes[x].plot(xrange, func2(xrange), color='blue', label='new')

    axes[x].set\_title(title)

    axes[x].set\_xlabel('n')

    axes[x].set\_ylabel(ylabel)

    axes[x].legend(loc='upper right')

xrange = np.arange(2, 201, dtype=np.longdouble)

plot(xrange, MLE\_bias\_estimator, new\_bias\_estimator, 'bias', 'bias of MLE and new estimators', 0)

plot(xrange, MLE\_var\_estimator, new\_var\_estimator, 'variance', 'variance of MLE and new estimators', 1)

plt.savefig("3b.png", dpi=300)

plt.show()

A picture containing histogram

Description automatically generated

c) Refer to notebook

import numpy as np

import matplotlib.pyplot as plt

SD = 1

def MLE\_estimator\_MSE(n):

    return (2\*SD\*\*4/n) - (SD\*\*4/n\*\*2)

def new\_estimator\_MSE(n):

    return (2\*SD\*\*4) / (n-1)

MLE\_estimator\_MSE = np.vectorize(MLE\_estimator\_MSE)

new\_estimator\_MSE = np.vectorize(new\_estimator\_MSE)

xrange = np.arange(2, 101, dtype=np.longdouble)

plt.plot(xrange, MLE\_estimator\_MSE(xrange), color='red', label='MLE estimator')

plt.plot(xrange, new\_estimator\_MSE(xrange), color='blue', label='new estimator')

plt.title('Comparing MSE of MLE and new estimator')

plt.legend(loc='upper right')

plt.xlabel('n')

plt.ylabel('MSE')

plt.ylim(0, 2.2)

plt.savefig("3c.png", dpi=300)

plt.show()

Chart

Description automatically generated