Bryan Kryitz, Michael Shusta, Connor Dick

Applied Modeling and Optimization

Exam 2

1. (a) In order for second order necessary conditions to be satisfied, the gradient at x\* must be equal to zero and the hessian must be either positive semi-definite or positive-definite. In this case, both of these conditions are satisfied.

(b) Shape, square

Description automatically generated

(c) By the 1000th iteration, the function nearly reached the stopping threshold. This can be sped up by increasing the learning rate. Additionally, it only requires a couple hundred iterations before it approaches its minimum value. Gradient descent in this example converges at a rate 1/k.

*Python Script:*

import matplotlib.pyplot as plt

*#initial values*

cur\_a = 1

cur\_b = 1

cur\_c = 1

rate = 0.001 *# Learning rate*

precision = pow(10,-6) *#This tells us when to stop the algorithm*

previous\_step\_size\_a = 1

previous\_step\_size\_b = 1

previous\_step\_size\_c = 1

max\_iters = 10000 *# maximum number of iterations*

iters = 0 *#iteration counter*

da = lambda a,b,c: (2\*a) + 10 + (4\*pow(b,2)\*a) + (8\*pow(c,2)\*a) *#d/da*

db = lambda a,b: (2\*b) + 16 + (4\*pow(a,2)\*b) *#d/db*

dc = lambda a,c: (2\*c) + 14 + (8\*pow(a,2)\*c) *#d/dc*

iterations = []

value\_of\_objctive\_function = []

while previous\_step\_size\_a > precision or previous\_step\_size\_b > precision or previous\_step\_size\_c > precision  and iters < max\_iters:

    prev\_a = cur\_a *#Store current a value in prev\_a*

    prev\_b = cur\_b *#Store current a value in prev\_a*

    prev\_c = cur\_c *#Store current a value in prev\_a*

    cur\_a = cur\_a - rate \* da(prev\_a,prev\_b, prev\_c) *#Grad descent a*

    cur\_b = cur\_b - rate \* db(prev\_a,prev\_b) *#Grad descent b*

    cur\_c = cur\_c - rate \* dc(prev\_a,prev\_c) *#Grad descent c*

    previous\_step\_size\_a = abs(cur\_a - prev\_a) *#Change in a*

    previous\_step\_size\_b = abs(cur\_b - prev\_b) *#Change in b*

    previous\_step\_size\_c = abs(cur\_c - prev\_c) *#Change in c*

    iters = iters+1 *#iteration count*

    iterations.append(iters)

    value\_of\_objctive\_function.append(f(cur\_a,cur\_b))

print("The local minimum occurs at", cur\_a,cur\_b,cur\_c, " After: ",iters, " iterations" )

*#Create the plot*

plt.plot(iterations, value\_of\_objctive\_function)

plt.xlabel('Iterations')

plt.ylabel('Objective Function Value')

import matplotlib.pyplot as plt

*#initial values*

cur\_a = 1

cur\_b = 1

cur\_c = 1

rate = 0.001 *# Learning rate*

precision = pow(10,-6) *#This tells us when to stop the algorithm*

previous\_step\_size\_a = 1

previous\_step\_size\_b = 1

previous\_step\_size\_c = 1

max\_iters = 10000 *# maximum number of iterations*

iters = 0 *#iteration counter*

da = lambda a,b,c: (2\*a) + 10 + (4\*pow(b,2)\*a) + (8\*pow(c,2)\*a) *#d/da*

db = lambda a,b: (2\*b) + 16 + (4\*pow(a,2)\*b) *#d/db*

dc = lambda a,c: (2\*c) + 14 + (8\*pow(a,2)\*c) *#d/dc*

iterations = []

value\_of\_objctive\_function = []

while previous\_step\_size\_a > precision or previous\_step\_size\_b > precision or previous\_step\_size\_c > precision  and iters < max\_iters:

    prev\_a = cur\_a *#Store current a value in prev\_a*

    prev\_b = cur\_b *#Store current a value in prev\_a*

    prev\_c = cur\_c *#Store current a value in prev\_a*

    cur\_a = cur\_a - rate \* da(prev\_a,prev\_b, prev\_c) *#Grad descent a*

    cur\_b = cur\_b - rate \* db(prev\_a,prev\_b) *#Grad descent b*

    cur\_c = cur\_c - rate \* dc(prev\_a,prev\_c) *#Grad descent c*

    previous\_step\_size\_a = abs(cur\_a - prev\_a) *#Change in a*

    previous\_step\_size\_b = abs(cur\_b - prev\_b) *#Change in b*

    previous\_step\_size\_c = abs(cur\_c - prev\_c) *#Change in c*

    iters = iters+1 *#iteration count*

    iterations.append(iters)

    value\_of\_objctive\_function.append(f(cur\_a,cur\_b))

print("The local minimum occurs at", cur\_a,cur\_b,cur\_c, " After: ",iters, " iterations" )

*#Create the plot*

plt.plot(iterations, value\_of\_objctive\_function)

plt.xlabel('Iterations')

plt.ylabel('Objective Function Value')

1. (a) Chart, surface chart

   Description automatically generatedChart, surface chart

   Description automatically generated

(b) We started our barrier technique method at [0.5 0.5]. The purpose of the log barrier technique is to ensure that our optimization technique always stays within our feasible set. Thus, as we approach the edge of our feasible set, the penalty function increases toward infinity. The following calculations were performed by hand:

Here we will let A = x1 and B = x2

We can then find the critical points with the following calculations:

(c) Shape, square

Description automatically generated

*Python Script:*

import matplotlib.pyplot as plt

import numpy as np

import math

f = lambda a,b: pow((a-1),2) + 2\* pow(b-2,2)

h1 = lambda a,b: 1 - pow(a,2) - pow(b,2)

h2 = lambda a,b: a+b

x = np.linspace(0, 20, 30)

y = np.linspace(0, 20, 30)

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

Z = f(X, Y)

H1 = h1(X, Y)

H2 = h2(X, Y)

fig = plt.figure()

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

ax.plot\_surface(X, Y, Z, rstride=1, cstride=1,

                cmap='viridis', edgecolor='none')

ax.contour3D(X, Y, H1, 50, cmap='binary')

ax.contour3D(X, Y, H2, 50, cmap='binary')

ax.set\_xlabel('x')

ax.set\_ylabel('y')

ax.set\_zlabel('z');

ax.view\_init(40,30)

ax.set\_title('f(x)');

*#========================================*

f = lambda a,b: pow((a-1),2) + 2\* pow(b-2,2) - np.log(a+b)

x = np.linspace(1, 20, 30)

y = np.linspace(1, 20, 30)

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

Z\_two = f(X, Y)

fig = plt.figure()

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

ax.plot\_surface(X, Y, Z\_two, rstride=1, cstride=1,

                cmap='viridis', edgecolor='none')

ax.set\_xlabel('x')

ax.set\_ylabel('y')

ax.set\_zlabel('z');

ax.view\_init(40,0)

ax.set\_title('f(x) with barrier functions (feasible set)');

*#========================================*

*#Find a solution to the problem using the natural logarithmic barrier function, i.e., the barrier*

*#function is -log(h1(𝑥)) - log(h2(𝑥)). Use initialization vector [0.5 0.5]T and the initial penalty*

*#parameter equal to 1 and reduce it by ½ in each iteration. Use a stopping threshold of 0.002;*

penalty = 1

rate = 0.001

stopping\_threshold = 0.002

cur\_a = 0.5

cur\_b = 0.5

precision = pow(10,-6) *#This tells us when to stop the algorithm*

max\_iters = 10000 *# maximum number of iterations*

previous\_step\_size\_a = 1

previous\_step\_size\_b = 1

iters = 0 *#iteration counter*

da = lambda a,b: 2\*a-2 *#d/da*

db = lambda a,b: 4\*b-8 *#d/db*

da\_log = lambda a,b: 1/(math.log(10)\*(a\*b)) *#d/da LOG PART*

db\_log = lambda a,b: 1/(math.log(10)\*(a\*b)) *#d/db LOG PART*

iterations = []

value\_of\_objctive\_function = []

while previous\_step\_size\_a > precision or previous\_step\_size\_b > precision and iters < max\_iters:

    prev\_a = cur\_a *#Store current a value in prev\_a*

    prev\_b = cur\_b *#Store current a value in prev\_a*

    cur\_a = cur\_a - rate \* da(prev\_a,prev\_b) - penalty\*da\_log(prev\_a,prev\_b) *#Grad descent a*

    cur\_b = cur\_b - rate \* db(prev\_a,prev\_b) - penalty\*db\_log(prev\_a,prev\_b) *#Grad descent b*

    previous\_step\_size\_a = abs(cur\_a - prev\_a) *#Change in a*

    previous\_step\_size\_b = abs(cur\_b - prev\_b) *#Change in b*

    penalty\*=0.5

    iters = iters+1 *#iteration count*

    iterations.append(iters)

    value\_of\_objctive\_function.append(f(cur\_a,cur\_b))

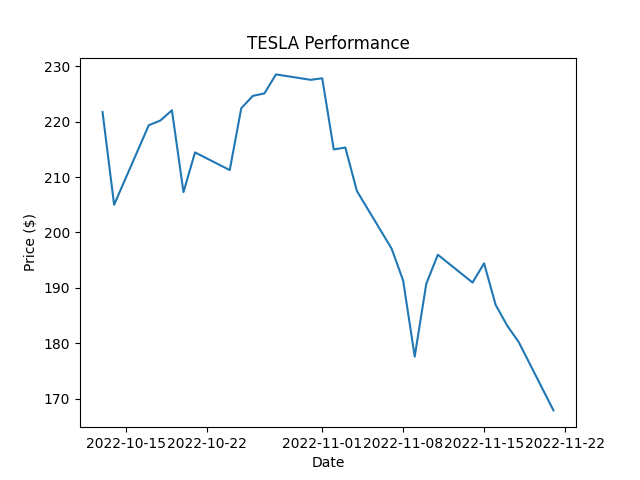
print("The local minimum occurs at", cur\_a,cur\_b, " After: ",iters, " iterations" )

*#========================================*

plt.plot(iterations, value\_of\_objctive\_function)

plt.xlabel('Iterations')

plt.ylabel('Objective Function Value')

1. (a) 

(b)

(c)

(d)

(e) In both gradient descent and stochastic gradient descent, we update parameters on each iteration to minimize our error function. The main difference is that descent uses the entire training set in order to find the gradient and optimal solution. On the other hand, stochastic gradient descent uses only one sample of the training set per iteration. This makes stochastic gradient descent algorithms a bit faster though less accurate and less likely to converge properly. With that said, gradient descent is more likely to converge, but it uses much more data per iteration. This makes standard gradient descent much more computationally difficult.

*Python Script:*

from sklearn.linear\_model import SGDClassifier

import pandas as pd

import yfinance as yf

from datetime import date, timedelta

from matplotlib import pyplot as plt

import numpy as np

*# calculating the start date*

Start = date.today() - timedelta(40)

Start.strftime('%Y-%m-%d')

*# calculating the end date*

End = date.today()

*# print(date.today())*

*# print(End)*

End.strftime('%Y-%m-%d')

*# func accepts ticker, return df of date & closing price*

def closing\_price(ticker):

    testData = []

    asset = pd.DataFrame(yf.download(ticker, start=Start, end=End)['Adj Close'])

    for index in asset.index:

*#print(asset['Adj Close'][index])*

        testData.append(asset['Adj Close'][index])

    return asset, testData

TESLA, testData = closing\_price('TSLA')                  *# CALL THE FUNCTION*

testData = np.array(testData).reshape(-1, 1)

*#testData.reshape(-1, 1)*

targetValues = []

for i in range(28):

    targetValues.append(i)

print(testData)

*# print(TESLA)*

plt.plot(TESLA)

plt.title('TESLA Performance')

plt.ylabel('Price ($)')

plt.xlabel('Date')

plt.show()

clf = SGDClassifier(loss="hinge", penalty="l2", max\_iter=1000)

clf.fit(testData, targetValues)

for i in range(90):

    test = clf.predict([[]])

    print(test)