CS168 Project 3

Henry Lin, Kaylee Bement

1.

A. Objective function value: 220.34264027

Objective function value with a consisting of all 0s: 91827.65959497

Since there is randomness in some of the function values, these numbers fluctuate, but are close to the numbers given.

B.

gd\_a = defaultdict(list)

def gradientDescent(): #does it work or not? hmmmmmm

for lr in learning\_rates:

a = np.zeros(shape=(d, 1))

for i in range(iterations):

totalGradient = 0.0

for point in range(n): #loop through all the datapoints and calculate loss?

curr = X[point].reshape((d, 1))

totalGradient += 2 \* curr \* (a.T.dot(curr) - y[point])

a -= lr \* totalGradient

loss = obj\_fn(a)

gd\_a[lr].append(loss)

def makePlot(objectiveFnValues, lr, numIterations, separate, outputFileName):

with warnings.catch\_warnings():

warnings.simplefilter("ignore")

plt.title("Objective fn Value vs Iteration #")

# plt.axis([0, 1000, 0.5, 0.75])

iterations = [i for i in range(1, numIterations + 1)]

plt.plot(iterations, objectiveFnValues[lr[0]], 'r--', label=lr[0])

plt.plot(iterations, objectiveFnValues[lr[1]], 'bs', label=lr[1])

if not separate:

plt.plot(iterations, objectiveFnValues[lr[2]], 'g^', label=lr[2])

plt.xlabel("Iteration #")

plt.ylabel("Objective fn Value")

plt.legend(shadow=True, fontsize='x-large', title="Step Sizes", loc = 0)

plt.savefig(outputFileName + ".png", format = 'png')

plt.close()

if separate:

plt.title("Objective fn Value vs Iteration #")

plt.plot(iterations, objectiveFnValues[lr[2]], 'g^', label='0.0007')

plt.xlabel("Iteration #")

plt.ylabel("Objective fn Value")

plt.legend(shadow=True, fontsize='x-large', title="Step Sizes", loc = 0)

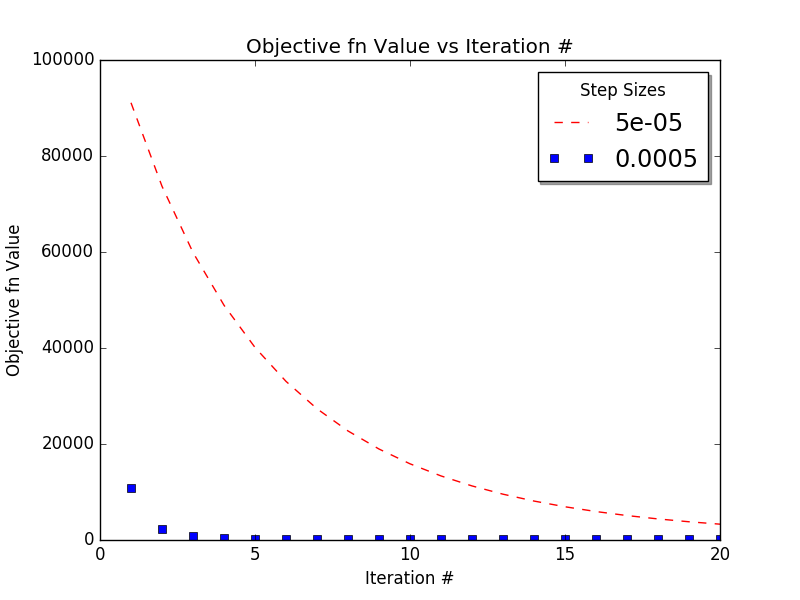
plt.savefig(outputFileName + "\_2.png", format = 'png')

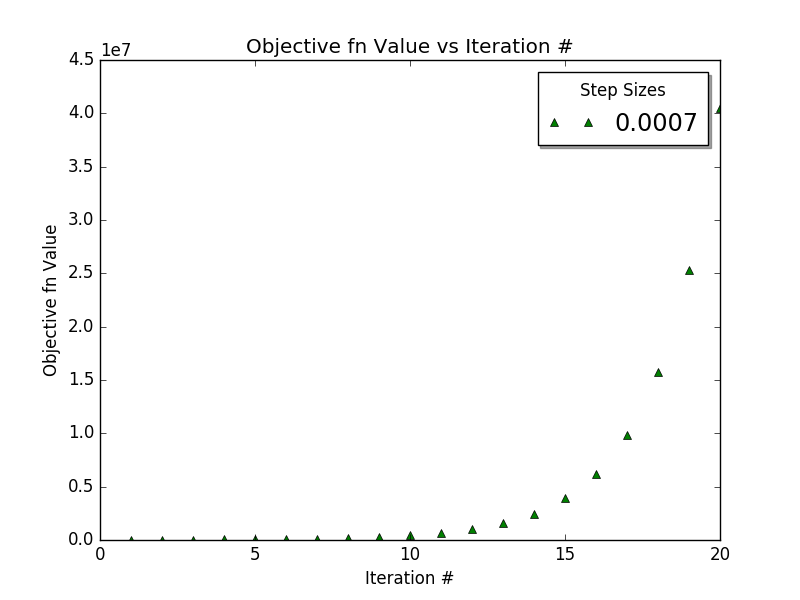
plt.close()

# print("1B")

# gradientDescent()

# makePlot(gd\_a, learning\_rates, iterations, True, "1b")





Gradient of f at at:

Optimal step size: 0.0005 with final objective function value of 233.09586745

The step size has a great impact on the convergence of gradient descent. If the step size is too small, gradient descent might be heading in the right direction, but never hit convergence, such as with step size = 0.00005, or it might find a local minimum and converge at the wrong value. With a small step size, it is also very easy to get stuck in a plateau without a large enough update to escape the plateau If the step size is too large, gradient descent might overstep the minimum and step towards an incorrect local minimum, such as with step size 0.0007. A common problem with large step sizes is oscillating around the minimum due to too large of updates. However, step size 0.0005 seemed to work well, and gave us an objective function value close to the one in part A.

C.

sgd\_iterations = 1000

sgd\_a = defaultdict(list)

sgd\_lr = [0.0005, 0.005, 0.01]

def SGD(): #I think this is working?

for lr in sgd\_lr:

a = np.zeros(shape=(d, 1))

for i in range(sgd\_iterations):

random\_point = random.randint(0, n - 1)

curr = X[random\_point].reshape((d, 1))

gradient = 2 \* curr \* (a.T.dot(curr) - y[random\_point])

a -= lr \* gradient

loss = obj\_fn(a)

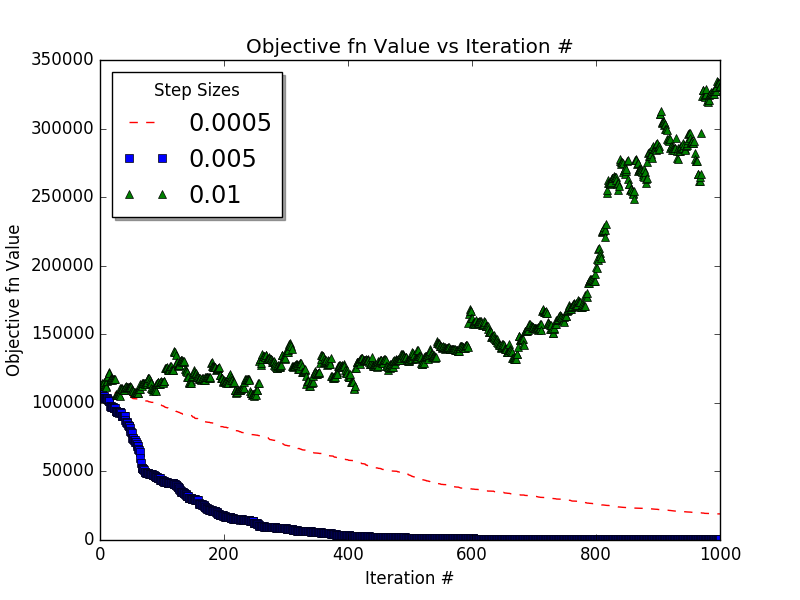
sgd\_a[lr].append(loss)

# print("1C")

# SGD()

# makePlot(sgd\_a, sgd\_lr, sgd\_iterations, False, "1c")

Optimal step size: 0.005 with final objective function value of 499.53798953



Step size influences the convergence of stochastic gradient descent in the same way it affects regular gradient descent, but it takes more iterations for SGD to converge. Regular gradient descent has a better final value than SGD, but in SGD, each data point is only used once on average since there are 1000 iterations to choose a random point and there are 1000 data points, while in regular gradient descent, each data point is used 20 times since every point is used in each iteration.

2.

A.

def normalized\_error(x, a , y):

numerator = np.matmul(x, a) - y

numerator = math.sqrt(np.sum(np.square(numerator)))

denominator = math.sqrt(np.sum(np.square(y)))

return float(numerator) / denominator

def part\_2a():

a = solve\_for\_a\_2()

train\_error = normalized\_error(X\_train, a, y\_train) # idk if this is correct but i switched both to using the normalized error fn

test\_error = normalized\_error(X\_test, a, y\_test)

return train\_error, test\_error

def solve\_for\_a\_2():

inv = np.linalg.inv(X\_train)

a = np.matmul(inv, y\_train)

return a

train\_err = 0

test\_err = 0

for n in range(num\_trials):

curr\_train\_err, curr\_test\_err = part\_2a()

train\_err += curr\_train\_err

test\_err += curr\_test\_err

train\_err /= num\_trials

test\_err /= num\_trials

# print("2A")

# print("Avg train err: ", train\_err)

# print("Avg test err: ", test\_err)

Avg train err: 1.08482770211125681e-14

Avg test err: 0.5823632339833625

B.

def solve\_for\_a\_2b(x, y, lmda):

temp = np.matmul(x.T, x) + lmda \* np.identity(d)

temp = np.linalg.inv(temp)

a = np.matmul(temp, x.T)

a = np.matmul(a, y)

return a

lambdas = [0.0005, 0.005, 0.05, 0.5, 5, 50, 500]

lambda\_train = []

lambda\_test = []

def part\_b():

num\_trials = 10

for l in lambdas:

training\_error = 0.0

test\_error = 0.0

for iteration in range(num\_trials):

a = solve\_for\_a\_2b(X\_train, y\_train, l)

#print("Current Lambda: ", l)

print("Iteration: ", iteration)

train\_error\_curr = normalized\_error(X\_train, a, y\_train)

test\_error\_curr = normalized\_error(X\_test, a, y\_test)

#print("Training Error: ", train\_error)

#print("Testing Error: ", test\_error)

training\_error += train\_error\_curr

test\_error += test\_error\_curr

print("Current Lambda: ", l)

print("Training Error: ", training\_error/num\_trials)

print("Testing Error: ", test\_error/num\_trials)

lambda\_train.append(training\_error/num\_trials)

lambda\_test.append(test\_error/num\_trials)

return

def partb\_plot(lambdas, train, test, outputFileName):

with warnings.catch\_warnings():

warnings.simplefilter("ignore")

plt.title("Training and Test Errors vs. Log of Lambda Values")

lambdas = [math.log(x) for x in lambdas]

plt.plot(lambdas, train, 'r--', label = "Training")

plt.plot(lambdas, test, 'bs', label = "Test")

plt.xlabel("Log Lambda")

plt.ylabel("Error")

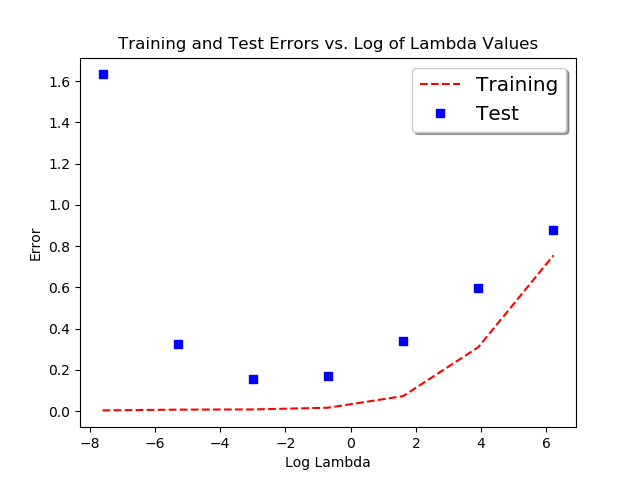
plt.legend(shadow=True, fontsize='x-large', loc = 0)

plt.savefig(outputFileName + ".png", format = 'png')

plt.close()

# part\_b()

# partb\_plot(lambdas, lambda\_train, lambda\_test, "2b")



Based on the graph, we can see that different values of lambda will drastically affect the test error reported. From part A, we can see a drastic decrease in test error once we implement regularization at the cost of slightly higher training error. This tradeoff is necessary if we want the model to be able to generalize to an unknown dataset however. Not all values of lambda work however, so it is clear that lambda is a hyperparameter that needs to be tuned to have the optimal results. There seems to be an ideal around 0.005, 0.05 where the test error is at its lowest.

C.

#CCCCC

num\_iterations = 1000000

step\_sizes = [0.00005, 0.0005, 0.005]

num\_trials = 10

#step\_sizes\_dict = {}

training\_sgd2\_error = {

0.00005: [],

0.0005: [],

0.005: []

}

test\_sgd2\_error = {

0.00005: [],

0.0005: [],

0.005: []

}

def SGD\_2(): #does for train and test simultaneously

for step in step\_sizes:

#print(step)

training\_error = 0.0

test\_error = 0.0

for trial in range(num\_trials):

print("Trial ", trial)

#a = solve\_for\_a\_2()

a = np.zeros(shape=(d, 1))

for i in range(num\_iterations):

random\_point = random.randint(0, train\_n - 1)

curr\_train = X\_train[random\_point].reshape((d, 1))

gradient = 2 \* curr\_train \* (a.T.dot(curr\_train) - y\_train[random\_point])

a -= step \* gradient

#loss = obj\_fn(a)

#sgd\_a[lr].append(loss)

curr\_training\_error = normalized\_error(X\_train, a, y\_train)

curr\_test\_error = normalized\_error(X\_test, a, y\_test)

training\_error += curr\_training\_error

test\_error += curr\_test\_error

training\_sgd2\_error[step].append(curr\_training\_error)

#print(len(training\_sgd2\_error[step]))

test\_sgd2\_error[step].append(curr\_test\_error)

print("Step size: ", step)

print("Average Training Error: ", training\_error/ num\_trials)

print("Average Test Error: ", test\_error / num\_trials)

# print("2c")

# SGD\_2()

Step size: 5e-05

Average Training Error: 0.016675896681903734

Average Test Error: 0.1562577003529706

Step size: 0.0005

Average Training Error: 0.004554721052220292

Average Test Error: 0.2173294324894938

Step size: 0.005

Average Training Error: 0.00036890858831097083

Average Test Error: 0.38507917451441664

For each step size, we can see a different training/test error along with a clear trend for the relationship between the two. As the training error decreases, the test error increases, meaning that the model is overfitting the training set with certain step sizes. This is ultimately undesirable because we want the model to be able to generalize to the unknown, so it is clear that step size is a hyperparameter that needs to be tuned to have optimal performance on a test set. Regardless of the step size chosen here however, in comparison to a, the training error is higher in exchange for a higher capability to generalize as shown by the lower test error.

D.

total\_iterations = 1000000

step\_sizes = [0.00005, 0.005]

error\_training\_each\_iteration = {

0.00005: [],

0.005: []

}

error\_test\_each100\_iteration = {

0.00005: [],

0.005: []

}

l2\_norms = {

0.00005: [],

0.005: []

}

def SGD\_3():

for step in step\_sizes:

print("Current Step: ", step)

a = np.zeros(shape=(d, 1))

for i in range(total\_iterations):

if i % 100000 == 0:

print("Current Iteration: ", i)

random\_point = random.randint(0, train\_n - 1)

curr\_train = X\_train[random\_point].reshape((d, 1))

gradient = 2 \* curr\_train \* (a.T.dot(curr\_train) - y\_train[random\_point])

a -= step \* gradient

curr\_training\_error = normalized\_error(X\_train, a, y\_train)

error\_training\_each\_iteration[step].append(curr\_training\_error)

if i % 100 == 0:

curr\_test\_error = normalized\_error(X\_test, a, y\_test)

error\_test\_each100\_iteration[step].append(curr\_test\_error)

l2\_norms[step].append(np.linalg.norm(a))

def plot\_2d(outputFileName, y\_dict, x\_axis, title, y\_label = "Error", true = False):

with warnings.catch\_warnings():

plt.title(title)

plt.plot(x\_axis, y\_dict[step\_sizes[0]], 'rs', label = "0.00005")

plt.plot(x\_axis, y\_dict[step\_sizes[1]], 'bs', label = "0.005")

temp = normalized\_error(X\_train, a\_true, y\_train)

list\_temp = [temp for x in range(len(x\_axis))]

if true:

plt.plot(x\_axis, list\_temp, 'gs', label = "True Error")

plt.xlabel("Iterations")

plt.ylabel(y\_label)

plt.legend(shadow=True, fontsize='x-large', loc = 0)

plt.savefig(outputFileName + ".png", format = 'png')

plt.close()

# NEEDS TO BE RUN

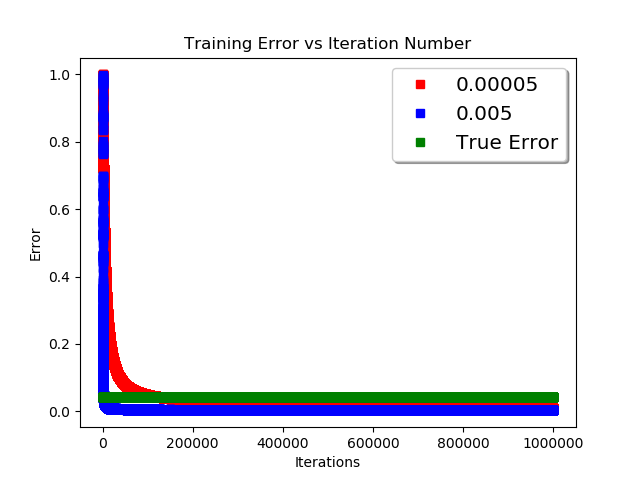
print("2D")

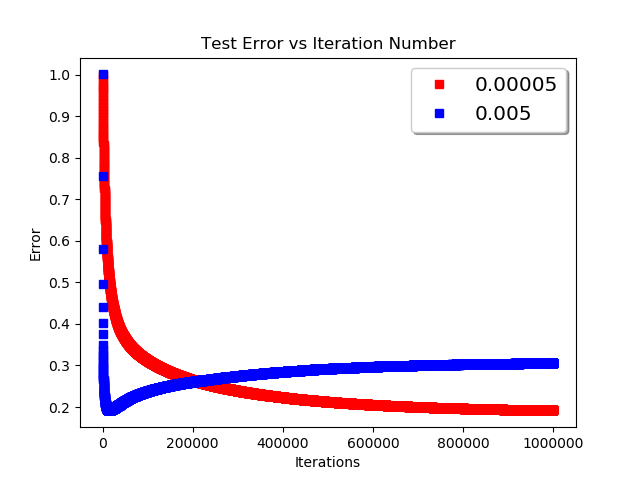
SGD\_3()

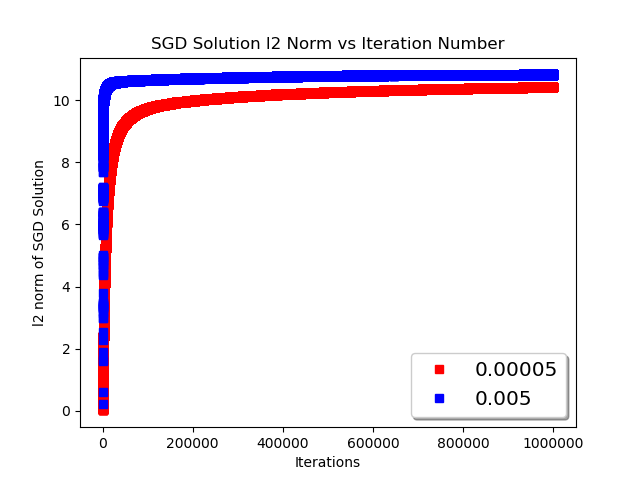
plot\_2d("2d\_1\_1mill\_log", error\_training\_each\_iteration, [x for x in range(1, total\_iterations + 1)], "Training Error vs Iteration Number", true = True)

plot\_2d("2d\_2\_1mill\_log", error\_test\_each100\_iteration, [x \* 100 for x in range(1, 10001)], "Test Error vs Iteration Number")

plot\_2d("2d\_3\_1mill\_log", l2\_norms, [x for x in range(1, total\_iterations + 1)], "SGD Solution l2 Norm vs Iteration Number", "l2 norm of SGD Solution")







From the plots, it seems that the generalization ability of a model decreases as the step size increases. With the step size of 0.005, the model begins to overfit pretty quickly as shown by the increase in test error around the 200,000th iteration. While test error increases, the training error continues to decrease, confirming that when the training error gets too low, it is a sign of the model possibly overfitting. From the third plot of l2 norm, we can see that with the bigger step size, our l2 norm is also bigger. This means that with 0.005, the model is learning larger weights that would be penalized with regularization. This is another sign of how regularization uses l2 norm to stop the model from overfitting.

E.

step\_size = 0.00005

radius\_opts = [0, 0.1, 0.5, 1, 10, 20, 30]

training\_errors = []

test\_errors = []

def SGD\_4():

for radius in radius\_opts:

print("Radius: ", radius)

a = np.random.uniform(size=(d, 1)) \* radius

total\_train\_err = 0.0

total\_test\_err = 0.0

for i in range(total\_iterations):

if i % 100000 == 0:

print("Current Iteration: ", i)

random\_point = random.randint(0, train\_n - 1)

curr\_train = X\_train[random\_point].reshape((d, 1))

gradient = 2 \* curr\_train \* (a.T.dot(curr\_train) - y\_train[random\_point])

a -= step\_size \* gradient

curr\_training\_error = normalized\_error(X\_train, a, y\_train)

total\_train\_err += curr\_training\_error

curr\_test\_error = normalized\_error(X\_test, a, y\_test)

total\_test\_err += curr\_test\_error

print("Training Error: ", total\_train\_err/total\_iterations)

training\_errors.append(total\_train\_err/total\_iterations)

test\_errors.append(total\_test\_err/total\_iterations)

def plot\_2e():

with warnings.catch\_warnings():

plt.title("Average errors vs r")

plt.plot(radius\_opts, training\_errors, 'rs', label = "Training Error")

plt.plot(radius\_opts, test\_errors, 'bs', label = "Test Error")

plt.xlabel("r")

plt.ylabel("Error")

plt.legend(shadow=True, fontsize='x-large', loc = 0)

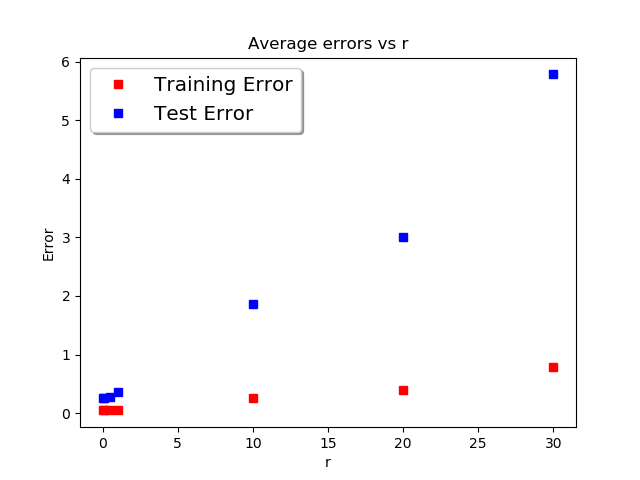
plt.savefig("2e.png", format = 'png')

plt.close()

# print("2E")

# SGD\_4()

# plot\_2e()



From the graph, we can see the drastic effect large weight initializations have on the ability to generalize. There is a small gradual increase in training error with larger weights, and we hypothesize this is because bad, large initializations are harder to correct, but with 1,000,000 iterations, this is generally not a big problem. For testing however, small weight initializations are ideal. Regularization from part b penalizes large weights, so it makes sense that regularization helps generalize since it would work against large weights through training and/or initialization.

3.

l = 0.000005

def custom\_regularization():

training\_error = 0.0

testing\_error = 0.0

for iteration in range(num\_trials):

X\_train = np.random.normal(0,1, size=(train\_n,d))

a\_true = np.random.normal(0,1, size=(d,1))

y\_train = X\_train.dot(a\_true) + np.random.normal(0,0.5,size=(train\_n,1))

X\_test = np.random.normal(0,1, size=(test\_n,d))

y\_test = X\_test.dot(a\_true) + np.random.normal(0,0.5,size=(test\_n,1))

a = solve\_for\_a\_2b(X\_train, y\_train, l)

#print("Current Lambda: ", l)

if iteration % 100 == 0:

print("Iteration: ", iteration)

train\_error\_curr = normalized\_error(X\_train, a, y\_train)

test\_error\_curr = normalized\_error(X\_test, a, y\_test)

#print("Training Error: ", train\_error)

#print("Testing Error: ", test\_error)

training\_error += train\_error\_curr

testing\_error += test\_error\_curr

return training\_error, testing\_error

# print("custom")

# training\_error , testing\_error = custom\_regularization()

# train\_avg = training\_error / custom\_trials

# test\_avg = testing\_error / custom\_trials

# print(train\_avg)

# print(test\_avg)

Average test error: 0.7062

We decided to use l2 regularization because the dimensionality is greater than the size of the training data, and we wanted to avoid overfitting. After trying lambdas ranging from 5e-8 to 5e2, we found that a lambda of 5e-6 worked best. We were disappointed that the test error was so high given the results we found in 2b, thus we believe that a better test error is achievable. However, the test error won’t be as low as the results in 2b given the challenge that our dimensionality is so high relative to the training dataset size.

Here is our pseudocode:

set λ to 0.000005

set training error and testing error to 0

for each trial:

pick a true, X train and test, and y train and test

set a to (X\_trainTX\_train + λI)-1X\_trainTy\_train

find current normalized train and test error for a

increment training and testing error by current train and test error

divide training and testing error by number of trials to find average training and testing error

**CODE**

import random

import numpy as np

from collections import defaultdict

import warnings

import matplotlib.pyplot as plt

import math

#PART 1

d = 100 # dimensions of data

n = 1000 # number of data points

X = np.random.normal(0,1, size=(n,d))

a\_true = np.random.normal(0,1, size=(d,1))

y = X.dot(a\_true) + np.random.normal(0,0.5,size=(n,1)) #(1000, 1)

#print("shape of y: ", y.shape)

learning\_rates = [0.00005, 0.0005, 0.0007]

iterations = 20

#a = np.zeros(shape=(d, 1))

##print("shape of a: ", a.shape)

#print("shape of X: ", X.shape)

#print("shape of y: ", y.shape)

#print(a)

def part\_a(zeros = False):

a = solve\_for\_a(zeros)

sqrd\_error = obj\_fn(a)

return sqrd\_error

def solve\_for\_a(zeros = False):

if zeros:

return np.zeros(shape=(d, 1))

X\_t = X.T

X\_t\_X = np.matmul(X\_t, X) # (XTX)

inv = np.linalg.inv(X\_t\_X) # (XTX)^-1

inv\_X\_t = np.matmul(inv, X\_t) # (XTX)^-1 \* XT

a = np.matmul(inv\_X\_t, y) # (XTX)^-1 \* XTy

return a

def obj\_fn(a):

sqrd\_error = 0

a\_t = a.T

for i in range(n):

curr = X[i].reshape((d, 1))

error = a\_t.dot(curr) - y[i]

sqrd\_error += np.power(error, 2)

return sqrd\_error[0]

# print("1A")

# print("part a value: ", part\_a())

# print("part a w 0s: ", part\_a(True))

#normal a: 220.34264027, a w all 0s: 91827.65959497

gd\_a = defaultdict(list)

def gradientDescent(): #does it work or not? hmmmmmm

for lr in learning\_rates:

a = np.zeros(shape=(d, 1))

for i in range(iterations):

totalGradient = 0.0

for point in range(n): #loop through all the datapoints and calculate loss?

curr = X[point].reshape((d, 1))

totalGradient += 2 \* curr \* (a.T.dot(curr) - y[point])

a -= lr \* totalGradient

loss = obj\_fn(a)

gd\_a[lr].append(loss)

def makePlot(objectiveFnValues, lr, numIterations, separate, outputFileName):

with warnings.catch\_warnings():

warnings.simplefilter("ignore")

plt.title("Objective fn Value vs Iteration #")

# plt.axis([0, 1000, 0.5, 0.75])

iterations = [i for i in range(1, numIterations + 1)]

plt.plot(iterations, objectiveFnValues[lr[0]], 'r--', label=lr[0])

plt.plot(iterations, objectiveFnValues[lr[1]], 'bs', label=lr[1])

if not separate:

plt.plot(iterations, objectiveFnValues[lr[2]], 'g^', label=lr[2])

plt.xlabel("Iteration #")

plt.ylabel("Objective fn Value")

plt.legend(shadow=True, fontsize='x-large', title="Step Sizes", loc = 0)

plt.savefig(outputFileName + ".png", format = 'png')

plt.close()

if separate:

plt.title("Objective fn Value vs Iteration #")

plt.plot(iterations, objectiveFnValues[lr[2]], 'g^', label='0.0007')

plt.xlabel("Iteration #")

plt.ylabel("Objective fn Value")

plt.legend(shadow=True, fontsize='x-large', title="Step Sizes", loc = 0)

plt.savefig(outputFileName + "\_2.png", format = 'png')

plt.close()

# print("1B")

# gradientDescent()

# makePlot(gd\_a, learning\_rates, iterations, True, "1b")

# for lr in learning\_rates:

# print(lr, " final value: ", gd\_a[lr][iterations - 1]) # 2092.7466054, 233.09586745, 6.11484504+08

# #5e-05 final value: [2370.91484997]

# 0.0005 final value: [219.10093293]

# 0.0007 final value: [1.0681867e+09]

sgd\_iterations = 1000

sgd\_a = defaultdict(list)

sgd\_lr = [0.0005, 0.005, 0.01]

def SGD(): #I think this is working?

for lr in sgd\_lr:

a = np.zeros(shape=(d, 1))

for i in range(sgd\_iterations):

random\_point = random.randint(0, n - 1)

curr = X[random\_point].reshape((d, 1))

gradient = 2 \* curr \* (a.T.dot(curr) - y[random\_point])

a -= lr \* gradient

loss = obj\_fn(a)

sgd\_a[lr].append(loss)

# print("1C")

# SGD()

# makePlot(sgd\_a, sgd\_lr, sgd\_iterations, False, "1c")

# for lr in sgd\_lr:

# print(lr, " final value: ", sgd\_a[lr][sgd\_iterations - 1])

print("done with1")

#PART 2

train\_n = 100

test\_n = 1000

d = 100

X\_train = np.random.normal(0,1, size=(train\_n,d))

a\_true = np.random.normal(0,1, size=(d,1))

y\_train = X\_train.dot(a\_true) + np.random.normal(0,0.5,size=(train\_n,1))

X\_test = np.random.normal(0,1, size=(test\_n,d))

y\_test = X\_test.dot(a\_true) + np.random.normal(0,0.5,size=(test\_n,1))

num\_trials = 10

def normalized\_error(x, a , y):

numerator = np.matmul(x, a) - y

numerator = math.sqrt(np.sum(np.square(numerator)))

denominator = math.sqrt(np.sum(np.square(y)))

return float(numerator) / denominator

def part\_2a():

a = solve\_for\_a\_2()

train\_error = normalized\_error(X\_train, a, y\_train) # idk if this is correct but i switched both to using the normalized error fn

test\_error = normalized\_error(X\_test, a, y\_test)

return train\_error, test\_error

def solve\_for\_a\_2():

inv = np.linalg.inv(X\_train)

a = np.matmul(inv, y\_train)

return a

train\_err = 0

test\_err = 0

for n in range(num\_trials):

curr\_train\_err, curr\_test\_err = part\_2a()

train\_err += curr\_train\_err

test\_err += curr\_test\_err

train\_err /= num\_trials

test\_err /= num\_trials

# print("2A")

# print("Avg train err: ", train\_err)

# print("Avg test err: ", test\_err)

# Avg train err: 1.08482770211125681e-14

# Avg test err: 0.5823632339833625

#BBBBBBBBBB

def solve\_for\_a\_2b(x, y, lmda):

temp = np.matmul(x.T, x) + lmda \* np.identity(d)

temp = np.linalg.inv(temp)

a = np.matmul(temp, x.T)

a = np.matmul(a, y)

return a

lambdas = [0.0005, 0.005, 0.05, 0.5, 5, 50, 500]

lambda\_train = []

lambda\_test = []

def part\_b():

num\_trials = 10

for l in lambdas:

training\_error = 0.0

test\_error = 0.0

for iteration in range(num\_trials):

a = solve\_for\_a\_2b(X\_train, y\_train, l)

#print("Current Lambda: ", l)

print("Iteration: ", iteration)

train\_error\_curr = normalized\_error(X\_train, a, y\_train)

test\_error\_curr = normalized\_error(X\_test, a, y\_test)

#print("Training Error: ", train\_error)

#print("Testing Error: ", test\_error)

training\_error += train\_error\_curr

test\_error += test\_error\_curr

print("Current Lambda: ", l)

print("Training Error: ", training\_error/num\_trials)

print("Testing Error: ", test\_error/num\_trials)

lambda\_train.append(training\_error/num\_trials)

lambda\_test.append(test\_error/num\_trials)

return

def partb\_plot(lambdas, train, test, outputFileName):

with warnings.catch\_warnings():

warnings.simplefilter("ignore")

plt.title("Training and Test Errors vs. Log of Lambda Values")

lambdas = [math.log(x) for x in lambdas]

plt.plot(lambdas, train, 'r--', label = "Training")

plt.plot(lambdas, test, 'bs', label = "Test")

plt.xlabel("Log Lambda")

plt.ylabel("Error")

plt.legend(shadow=True, fontsize='x-large', loc = 0)

plt.savefig(outputFileName + ".png", format = 'png')

plt.close()

# part\_b()

# partb\_plot(lambdas, lambda\_train, lambda\_test, "2b")

#CCCCC

num\_iterations = 1000000

step\_sizes = [0.00005, 0.0005, 0.005]

num\_trials = 10

#step\_sizes\_dict = {}

training\_sgd2\_error = {

0.00005: [],

0.0005: [],

0.005: []

}

test\_sgd2\_error = {

0.00005: [],

0.0005: [],

0.005: []

}

def SGD\_2(): #does for train and test simultaneously

for step in step\_sizes:

#print(step)

training\_error = 0.0

test\_error = 0.0

for trial in range(num\_trials):

print("Trial ", trial)

#a = solve\_for\_a\_2()

a = np.zeros(shape=(d, 1))

for i in range(num\_iterations):

random\_point = random.randint(0, train\_n - 1)

curr\_train = X\_train[random\_point].reshape((d, 1))

gradient = 2 \* curr\_train \* (a.T.dot(curr\_train) - y\_train[random\_point])

a -= step \* gradient

#loss = obj\_fn(a)

#sgd\_a[lr].append(loss)

curr\_training\_error = normalized\_error(X\_train, a, y\_train)

curr\_test\_error = normalized\_error(X\_test, a, y\_test)

training\_error += curr\_training\_error

test\_error += curr\_test\_error

training\_sgd2\_error[step].append(curr\_training\_error)

#print(len(training\_sgd2\_error[step]))

test\_sgd2\_error[step].append(curr\_test\_error)

print("Step size: ", step)

print("Average Training Error: ", training\_error/ num\_trials)

print("Average Test Error: ", test\_error / num\_trials)

# print("2c")

# SGD\_2()

# Step size: 5e-05

# Average Training Error: 0.016675896681903734

# Average Test Error: 0.1562577003529706

# Step size: 0.0005

# Average Training Error: 0.004554721052220292

# Average Test Error: 0.2173294324894938

# Step size: 0.005

# Average Training Error: 0.00036890858831097083

# Average Test Error: 0.38507917451441664

#DDDDDDDDDD

total\_iterations = 1000000

step\_sizes = [0.00005, 0.005]

error\_training\_each\_iteration = {

0.00005: [],

0.005: []

}

error\_test\_each100\_iteration = {

0.00005: [],

0.005: []

}

l2\_norms = {

0.00005: [],

0.005: []

}

def SGD\_3():

for step in step\_sizes:

print("Current Step: ", step)

a = np.zeros(shape=(d, 1))

for i in range(total\_iterations):

if i % 100000 == 0:

print("Current Iteration: ", i)

random\_point = random.randint(0, train\_n - 1)

curr\_train = X\_train[random\_point].reshape((d, 1))

gradient = 2 \* curr\_train \* (a.T.dot(curr\_train) - y\_train[random\_point])

a -= step \* gradient

curr\_training\_error = normalized\_error(X\_train, a, y\_train)

error\_training\_each\_iteration[step].append(curr\_training\_error)

if i % 100 == 0:

curr\_test\_error = normalized\_error(X\_test, a, y\_test)

error\_test\_each100\_iteration[step].append(curr\_test\_error)

l2\_norms[step].append(np.linalg.norm(a))

def plot\_2d(outputFileName, y\_dict, x\_axis, title, y\_label = "Error", true = False):

with warnings.catch\_warnings():

plt.title(title)

plt.plot(x\_axis, y\_dict[step\_sizes[0]], 'rs', label = "0.00005")

plt.plot(x\_axis, y\_dict[step\_sizes[1]], 'bs', label = "0.005")

temp = normalized\_error(X\_train, a\_true, y\_train)

list\_temp = [temp for x in range(len(x\_axis))]

if true:

plt.plot(x\_axis, list\_temp, 'gs', label = "True Error")

plt.xlabel("Iterations")

plt.ylabel(y\_label)

plt.legend(shadow=True, fontsize='x-large', loc = 0)

plt.savefig(outputFileName + ".png", format = 'png')

plt.close()

# NEEDS TO BE RUN

print("2D")

SGD\_3()

plot\_2d("2d\_1\_1mill\_log", error\_training\_each\_iteration, [x for x in range(1, total\_iterations + 1)], "Training Error vs Iteration Number", true = True)

plot\_2d("2d\_2\_1mill\_log", error\_test\_each100\_iteration, [x \* 100 for x in range(1, 10001)], "Test Error vs Iteration Number")

plot\_2d("2d\_3\_1mill\_log", l2\_norms, [x for x in range(1, total\_iterations + 1)], "SGD Solution l2 Norm vs Iteration Number", "l2 norm of SGD Solution")

#EEEEEEEEEEEE

step\_size = 0.00005

radius\_opts = [0, 0.1, 0.5, 1, 10, 20, 30]

training\_errors = []

test\_errors = []

def SGD\_4():

for radius in radius\_opts:

print("Radius: ", radius)

a = np.random.uniform(size=(d, 1)) \* radius

total\_train\_err = 0.0

total\_test\_err = 0.0

for i in range(total\_iterations):

if i % 100000 == 0:

print("Current Iteration: ", i)

random\_point = random.randint(0, train\_n - 1)

curr\_train = X\_train[random\_point].reshape((d, 1))

gradient = 2 \* curr\_train \* (a.T.dot(curr\_train) - y\_train[random\_point])

a -= step\_size \* gradient

curr\_training\_error = normalized\_error(X\_train, a, y\_train)

total\_train\_err += curr\_training\_error

curr\_test\_error = normalized\_error(X\_test, a, y\_test)

total\_test\_err += curr\_test\_error

print("Training Error: ", total\_train\_err/total\_iterations)

training\_errors.append(total\_train\_err/total\_iterations)

test\_errors.append(total\_test\_err/total\_iterations)

def plot\_2e():

with warnings.catch\_warnings():

plt.title("Average errors vs r")

plt.plot(radius\_opts, training\_errors, 'rs', label = "Training Error")

plt.plot(radius\_opts, test\_errors, 'bs', label = "Test Error")

plt.xlabel("r")

plt.ylabel("Error")

plt.legend(shadow=True, fontsize='x-large', loc = 0)

plt.savefig("2e.png", format = 'png')

plt.close()

# print("2E")

# SGD\_4()

# plot\_2e()

#PART 3

train\_n = 100

test\_n = 10000

d = 200

custom\_trials = 1000

num\_trials = 1000

custom\_train\_error = []

custom\_test\_error = []

# lambdas = [0.00000005, 0.0000005, 0.000005, 0.00005, 0.0005, 0.005, 0.05, 0.5, 5, 50, 500]

l = 0.000005

def custom\_regularization():

training\_error = 0.0

testing\_error = 0.0

for iteration in range(num\_trials):

X\_train = np.random.normal(0,1, size=(train\_n,d))

a\_true = np.random.normal(0,1, size=(d,1))

y\_train = X\_train.dot(a\_true) + np.random.normal(0,0.5,size=(train\_n,1))

X\_test = np.random.normal(0,1, size=(test\_n,d))

y\_test = X\_test.dot(a\_true) + np.random.normal(0,0.5,size=(test\_n,1))

a = solve\_for\_a\_2b(X\_train, y\_train, l)

#print("Current Lambda: ", l)

if iteration % 100 == 0:

print("Iteration: ", iteration)

train\_error\_curr = normalized\_error(X\_train, a, y\_train)

test\_error\_curr = normalized\_error(X\_test, a, y\_test)

#print("Training Error: ", train\_error)

#print("Testing Error: ", test\_error)

training\_error += train\_error\_curr

testing\_error += test\_error\_curr

return training\_error, testing\_error

# print("custom")

# training\_error , testing\_error = custom\_regularization()

# train\_avg = training\_error / custom\_trials

# test\_avg = testing\_error / custom\_trials

# print(train\_avg)

# print(test\_avg)