CS168 Project 3

Henry Lin, Kaylee Bement

1.

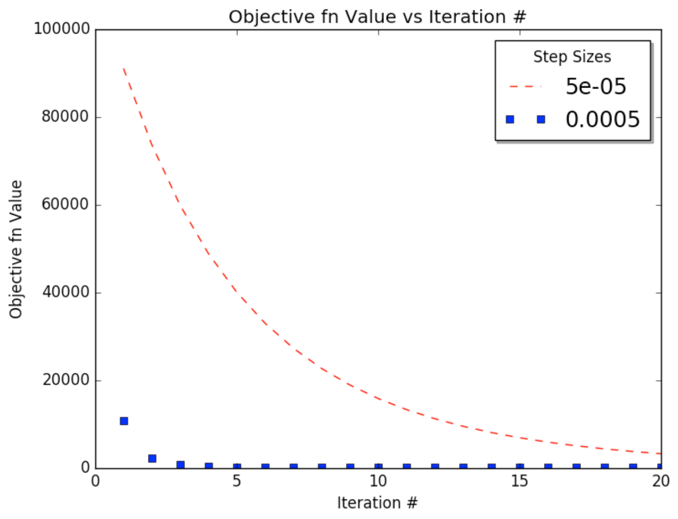
A. Objective function value: 220.34264027

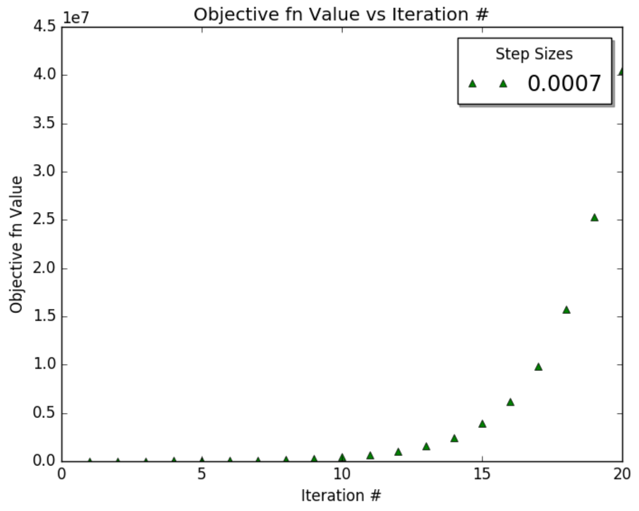
Objective function value with a consisting of all 0s: 91827.65959497

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 a_t: $2\sum_{i=1}^{n} \boldsymbol{x}^{(i)}(\boldsymbol{a}^T\boldsymbol{x}^{(i)} - y^{(i)})$

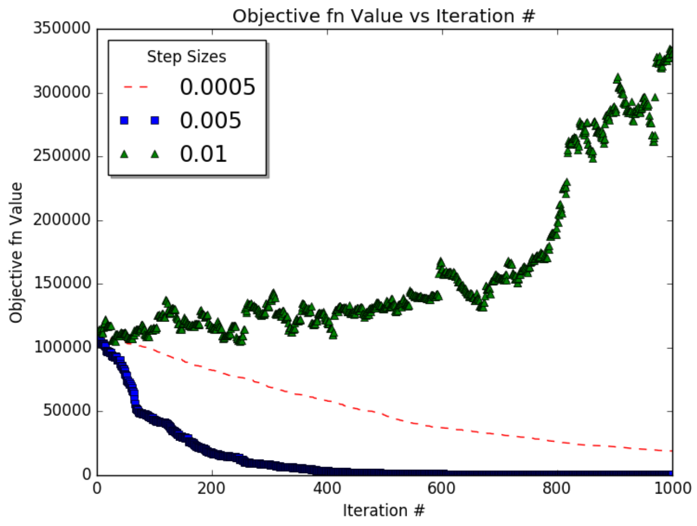
Optimal step size: 0.0005 with final objective function value of 233.09586745

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.

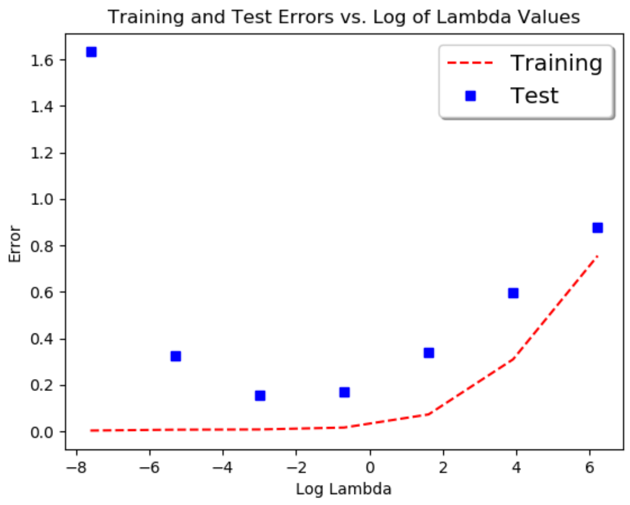
```
def normalized_error(x, a , y):

        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:  1.0848277021112568le-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()
```

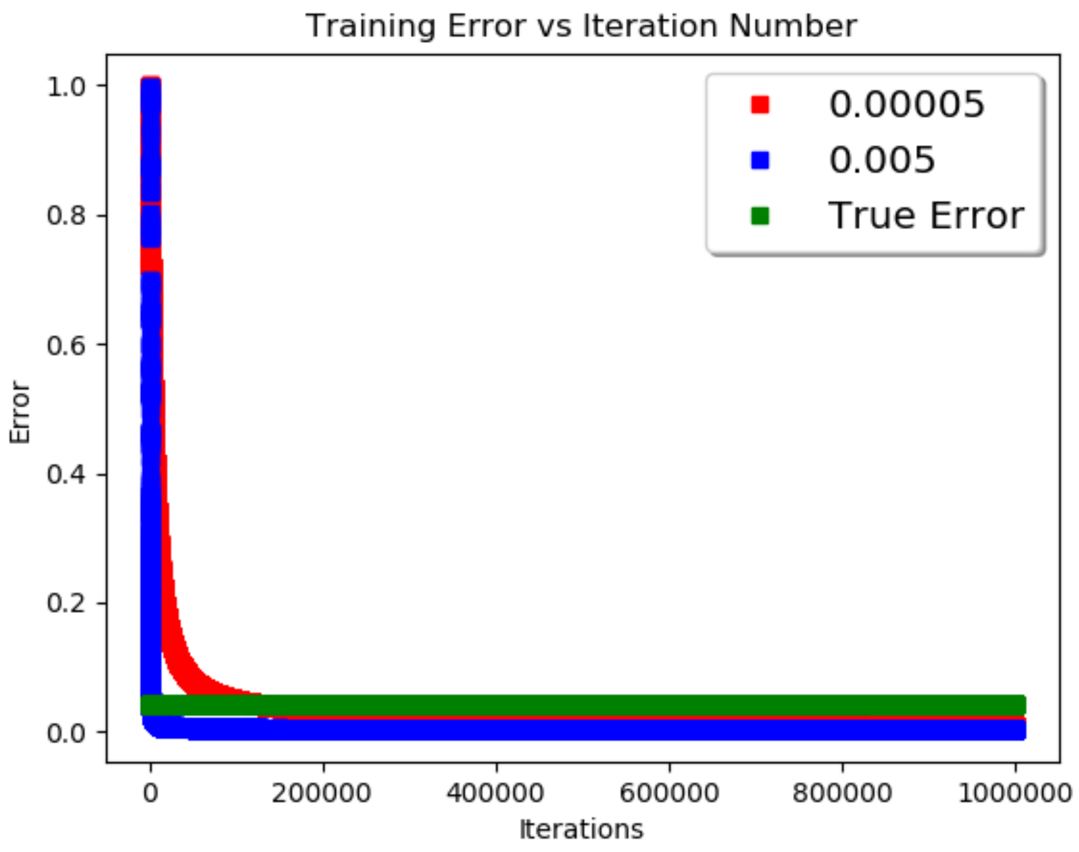
```
Step size:  5e-05

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
```

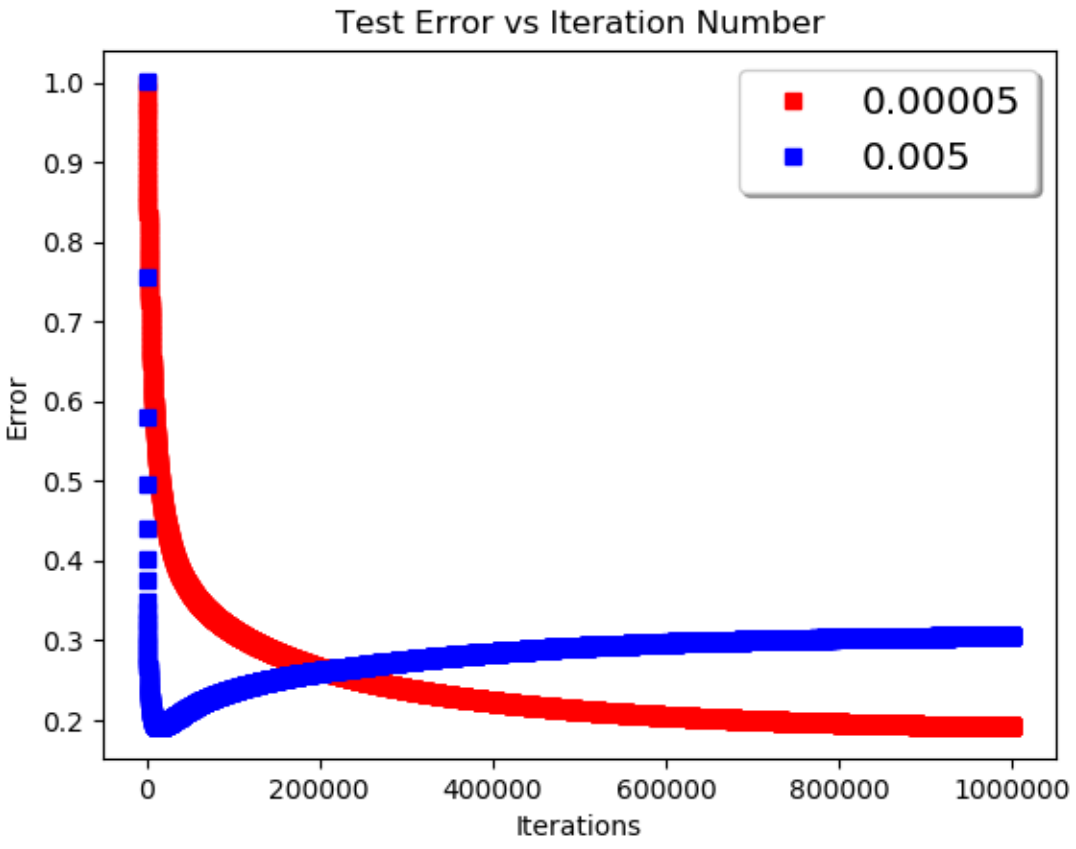
For each step size, we can see a different training/test error along with a clear trend for the relationship between the two. As the training error decreases, the test error increases, meaning that the model is overfitting the training set with certain step sizes. This is ultimately undesirable because 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.

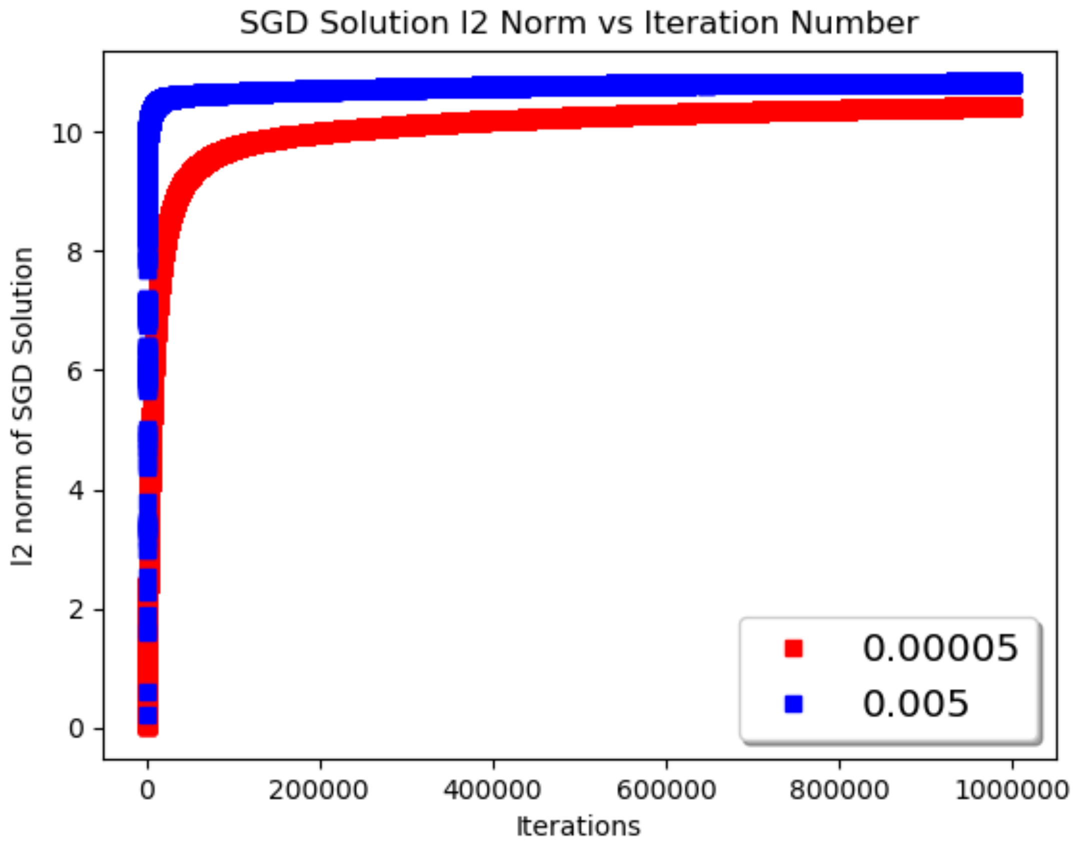
D.

```
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")
```


Training Error vs Iteration Number
Error
Iterations
1.0
0.8
0.6
0.4
0.2
0.0
0
200000
400000
600000
800000
1000000
0.00005
0.005
True Error


Test Error vs Iteration Number
0.00005
0.005
Error
Iterations
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0
200000
400000
600000
800000
1000000

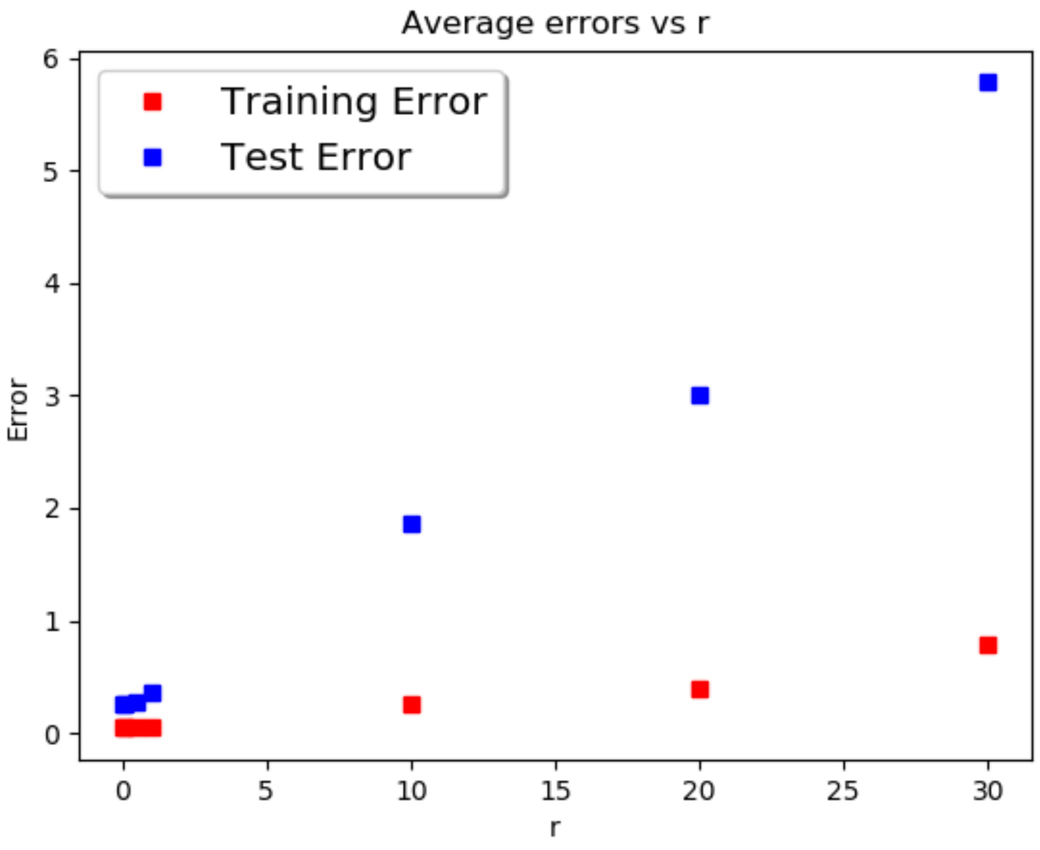


From the plots, it seems that the generalization ability of a model decreases as the step size increases. With the step size of 0.005, the model begins to overfit pretty quickly as shown by the increase in test error around the 200,000th iteration. While test error increases, the training error continues 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 $\lambda$ 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 $(X\_train^T X\_train + \lambda I)^{-1} X\_train^T y\_train$
    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:  1.0848277021112568le-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)
```