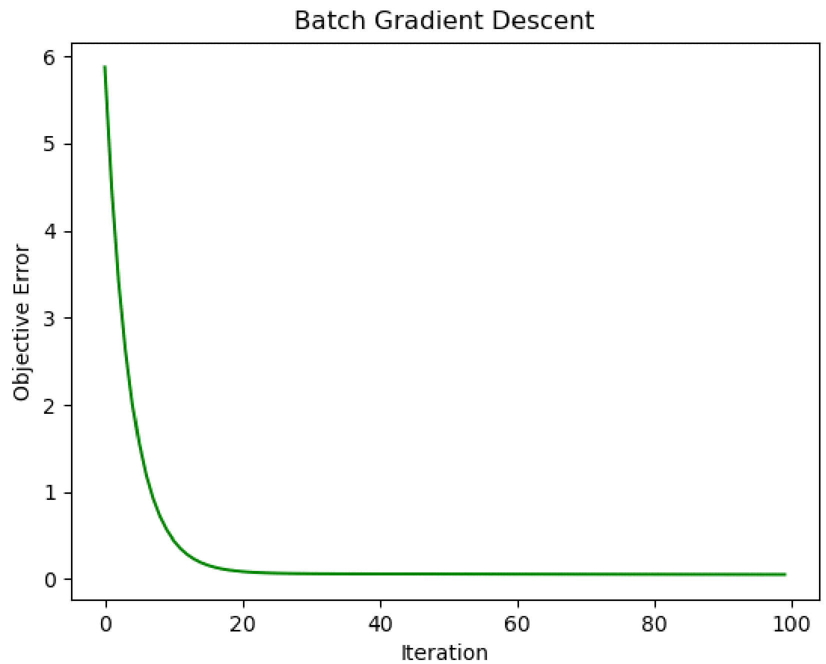
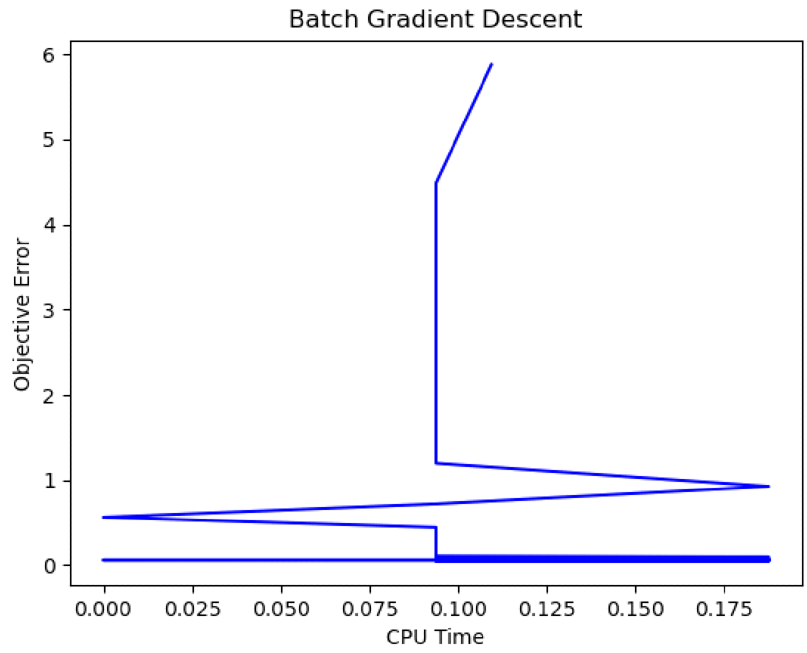
HW 2

Monday, October 11, 2021 5:17 PM

Problem 1:

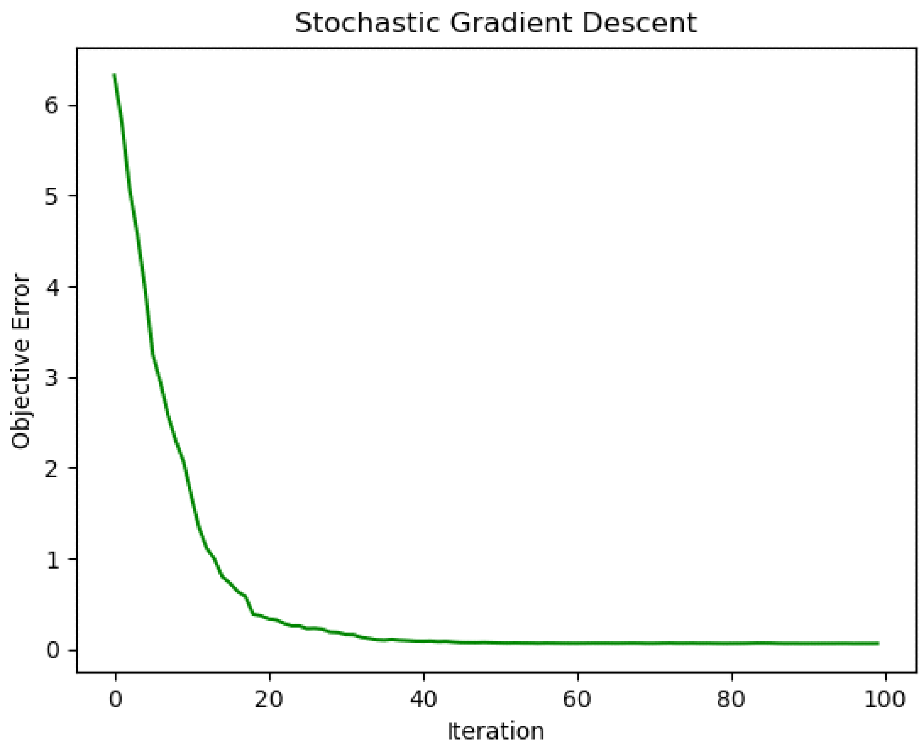


Plot of Batch Gradient Descent for Problem 1 (Iteration vs. Objective Error)

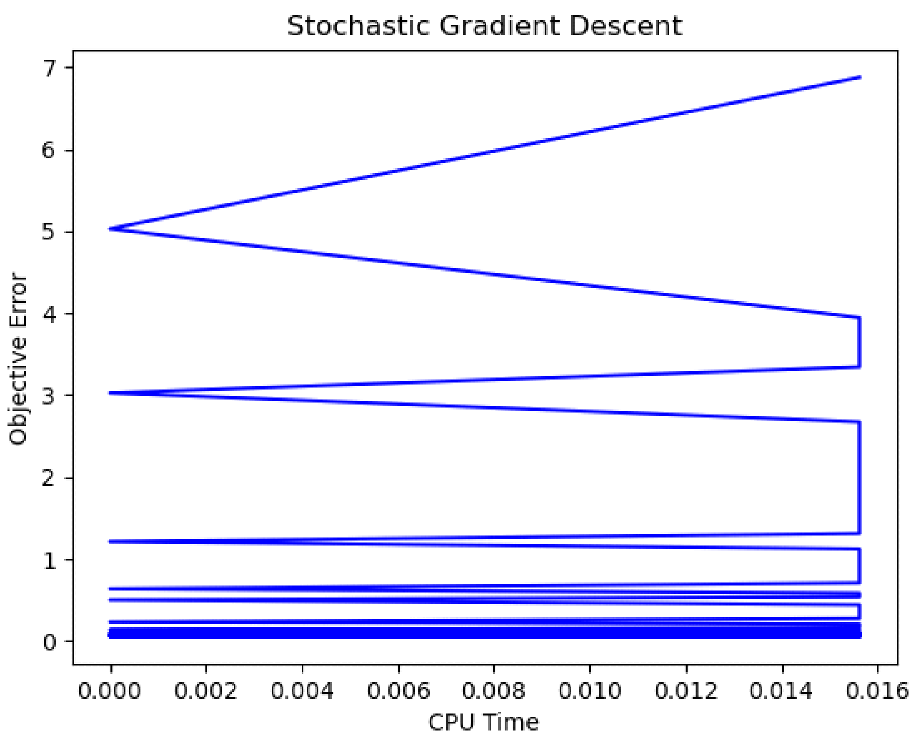


Plot of CPU Time vs. Objective Error for Problem 1 Batch Gradient Descent

Problem 1 Continued:

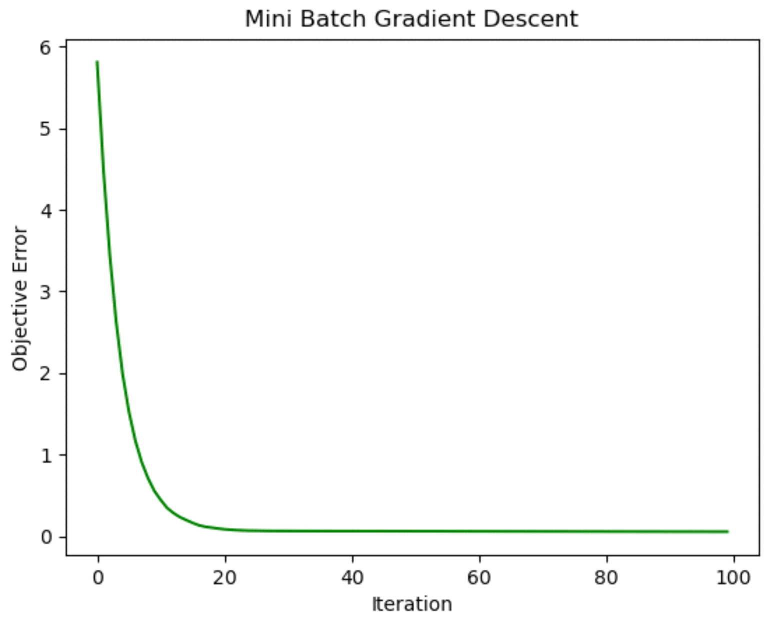


Plot of Stochastic Gradient Descent for Problem 1 (Iteration vs. Objective Error)

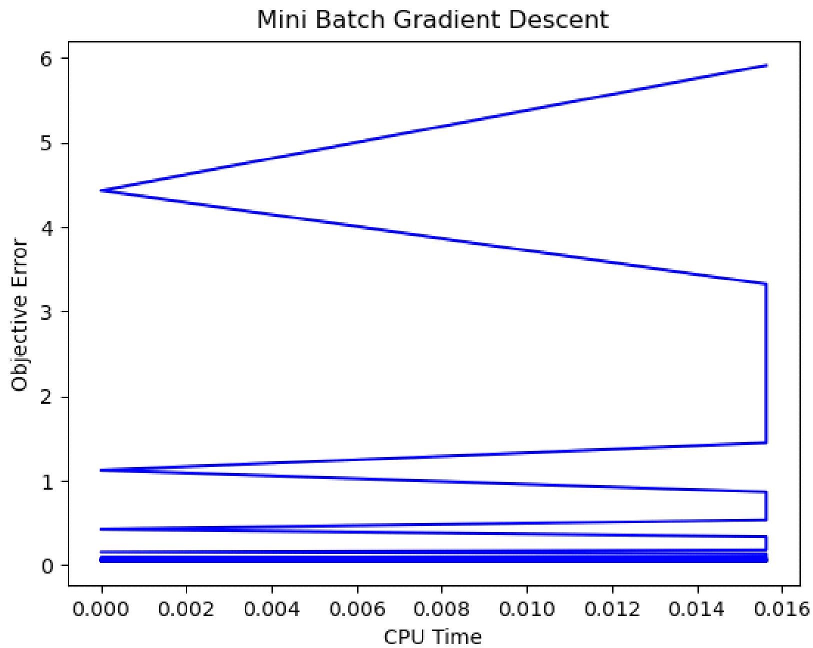


Plot of CPU Time vs. Objective Error for Problem 1 Stochastic Gradient Descent

Problem 1 Even More Continued:



Plot of Mini Batch Gradient Descent for Problem 1 (Iteration vs. Objective Error)



Plot of CPU Time vs. Objective Error for Problem 1 Mini Batch Gradient Descent

Problem 1 Discussion:

We see that in all cases we are generating a Theta vector that is clearly minimizing the objective error. Because this is gradient descent we expect to see the objective error decrease and it indeed does.

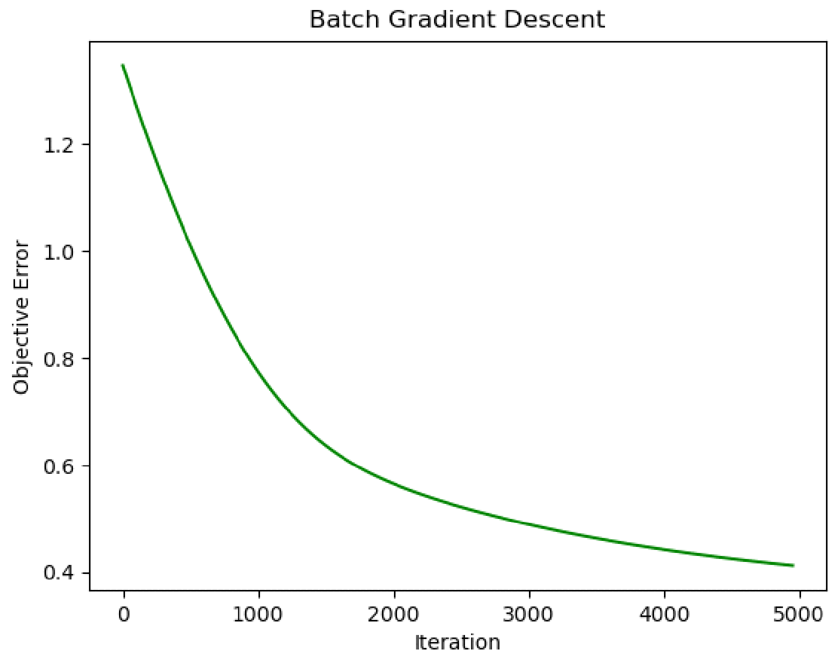
For Batch Gradient Descent we see that it becomes incredibly accurate in a short number of iterations but takes a lot of time between iterations

For Stochastic Gradient Descent we see that it does become quite accurate but not as accurate as Batch Gradient Descent. But the per iteration cost/time is incredibly small

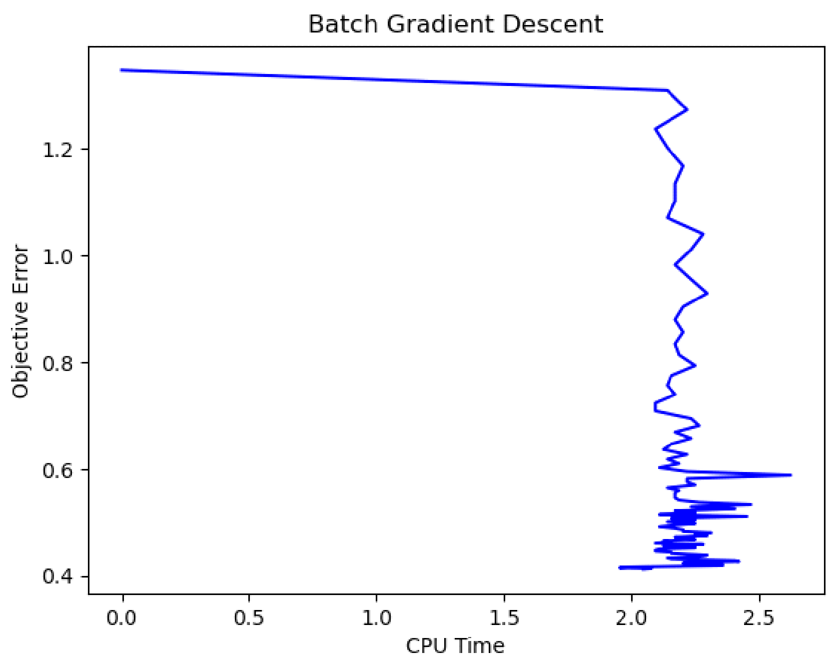
For Mini Batch Gradient Descent we have a happy medium between the 2. It minimizes the objective error quickly without sacrificing too much time for iterations but is still not as fast as Stochastic Gradient Descent or as accurate as Batch Gradient Descent.

The linear regression convergence theory holds

Problem 2:

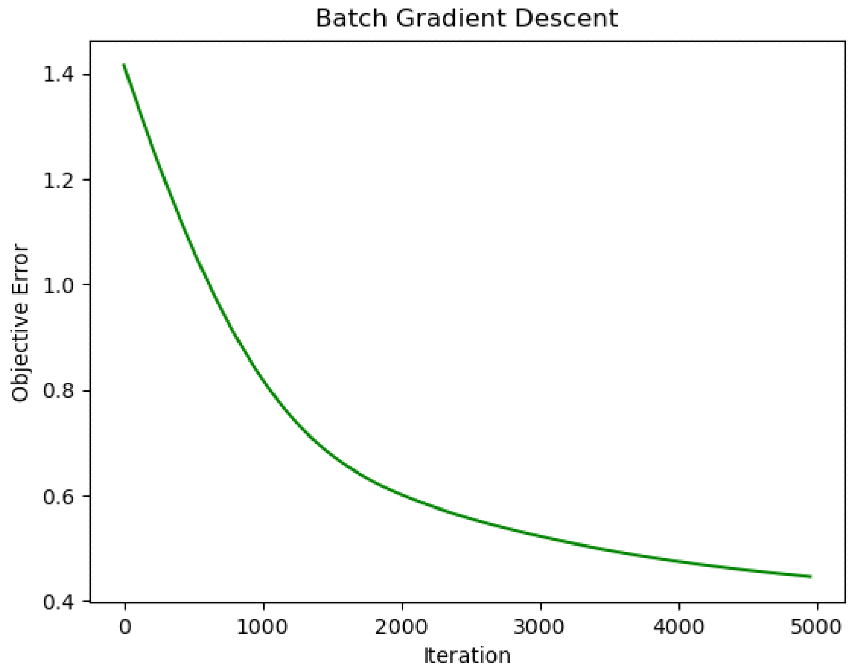


Plot of Batch Gradient Descent for Problem 2 (Iteration vs. Objective Error)
With $\lambda = 0$

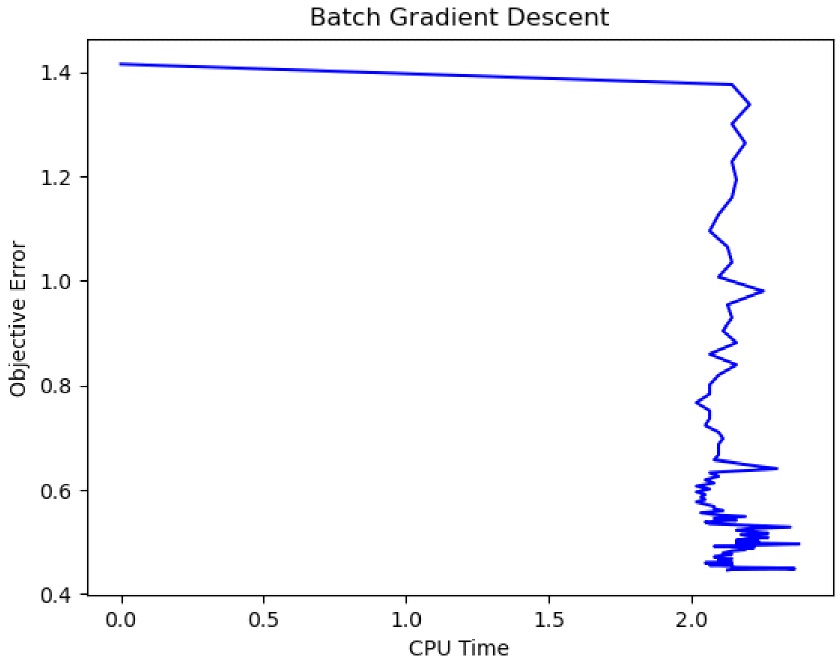


Plot of CPU Time vs. Objective Error for Problem 2 Batch Gradient Descent
With $\lambda = 0$

Problem 2 Continued:

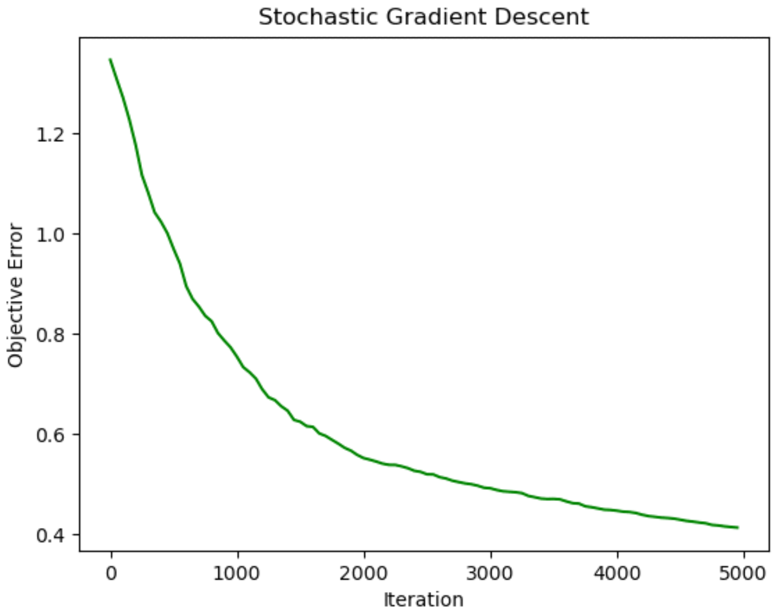


Plot of Batch Gradient Descent for Problem 2 (Iteration vs. Objective Error)
With $\lambda = 0.01$

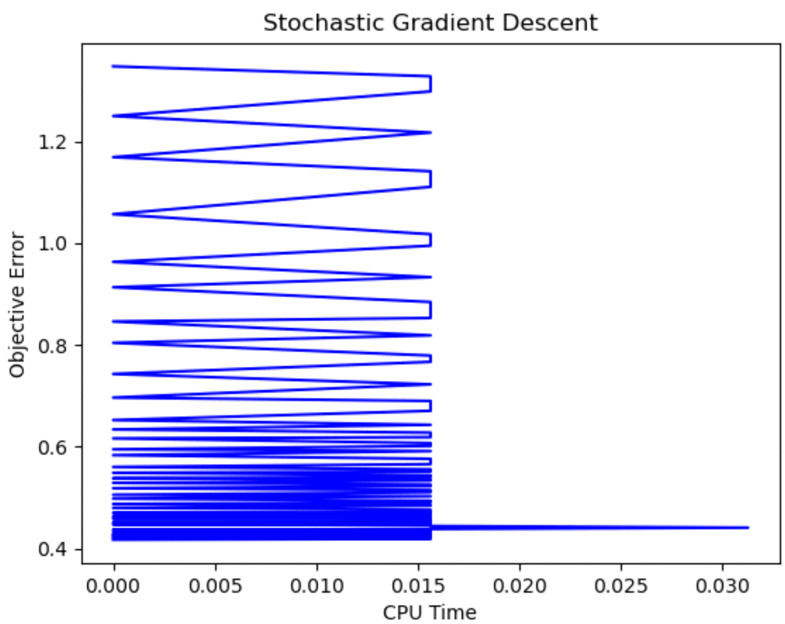


Plot of CPU Time vs. Objective Error for Problem 2 Batch Gradient Descent
With $\lambda = 0.01$

Problem 2 Even More Continued

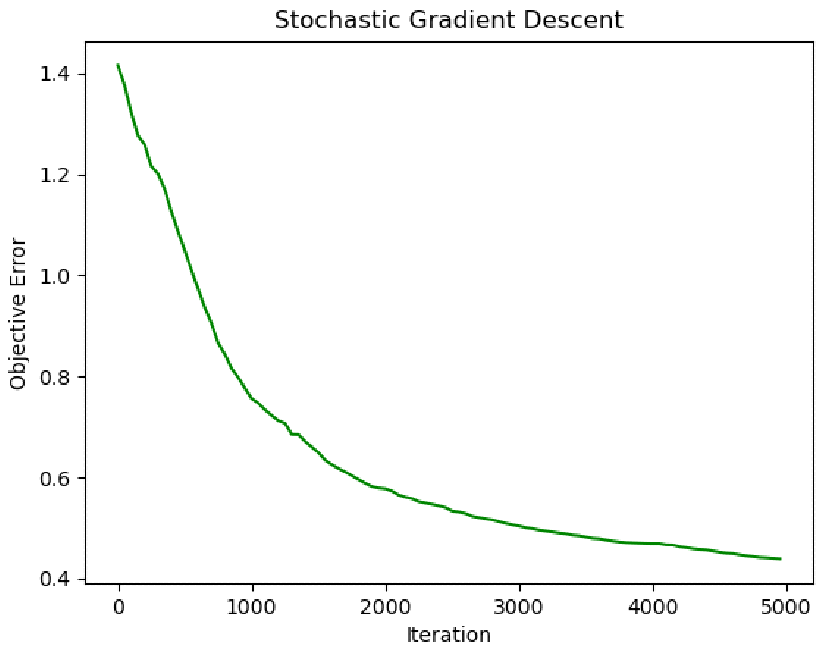


Plot of Stochastic Gradient Descent for Problem 2 (Iteration vs. Objective Error)
With $\lambda = 0$

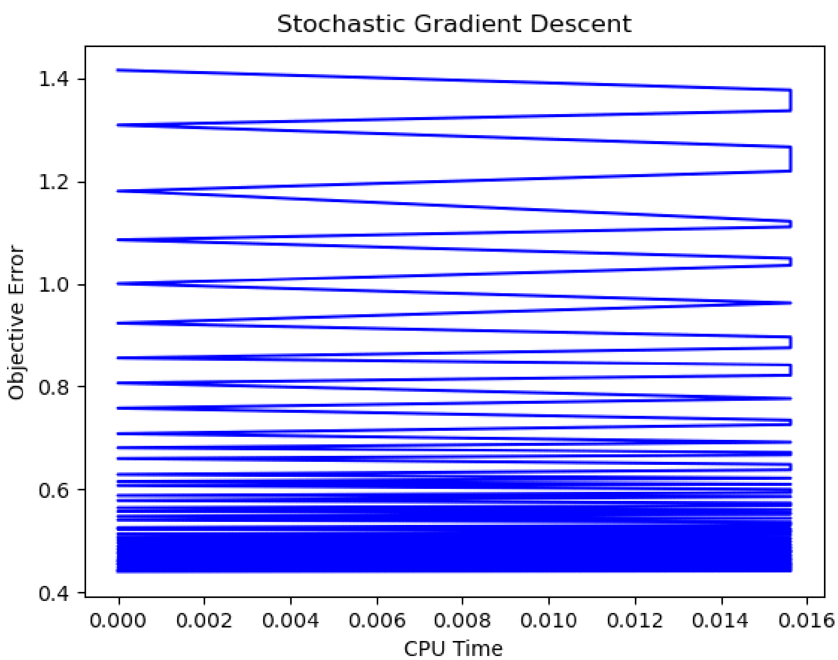


Plot of CPU Time vs. Objective Error for Problem 2 Stochastic Gradient Descent
With $\lambda = 0$

Problem 2 Even Further Continued:

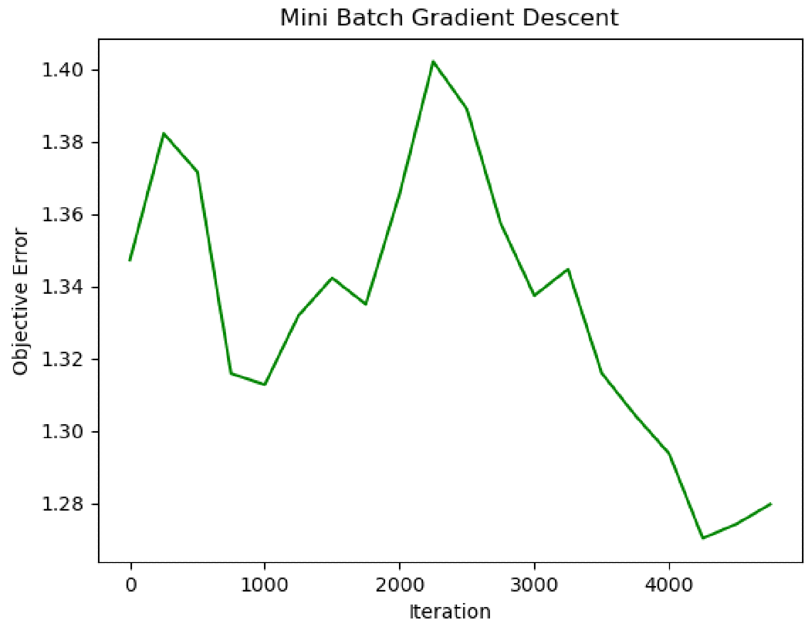


Plot of Stochastic Gradient Descent for Problem 2 (Iteration vs. Objective Error)
With $\lambda = 0.01$

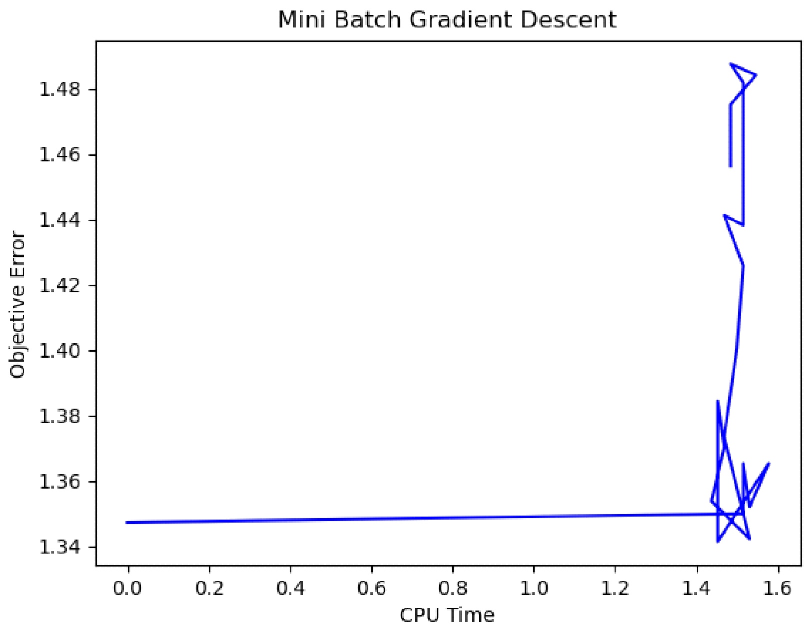


Plot of CPU Time vs. Objective Error for Problem 2 Stochastic Gradient Descent
With $\lambda = 0.01$

Problem 2 Even More Further Continued:

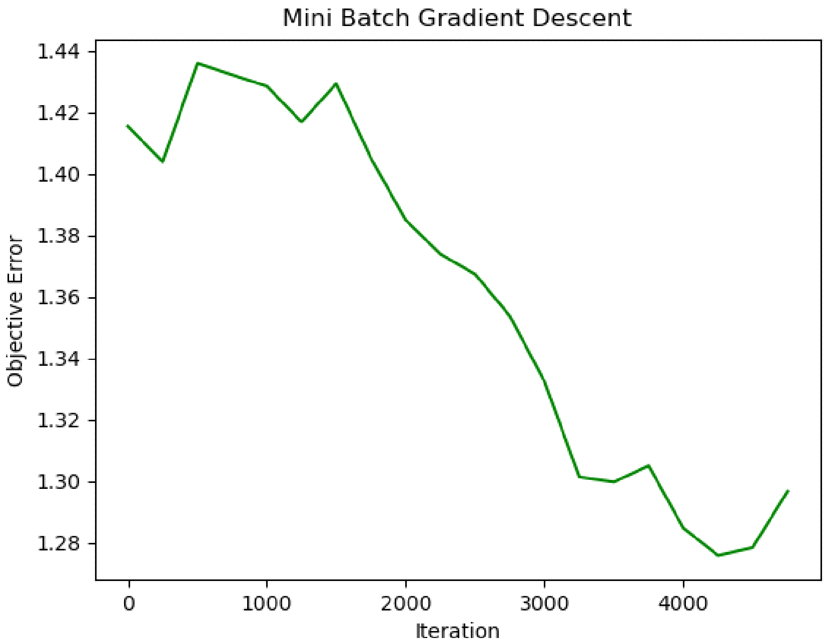


Plot of Mini Batch Gradient Descent for Problem 2 (Iteration vs. Objective Error)
With $\lambda = 0$

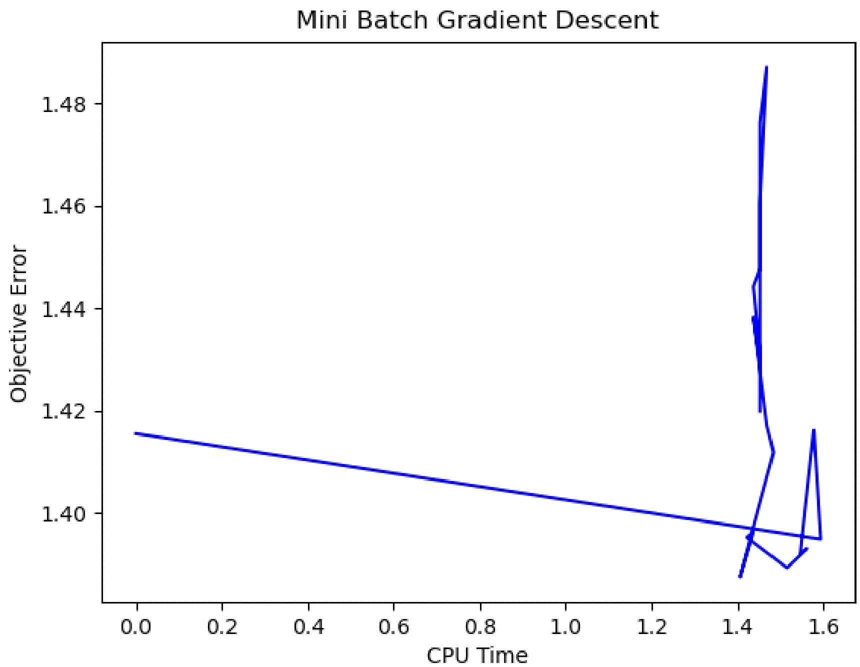


Plot of CPU Time vs. Objective Error for Problem 2 Mini Batch Gradient Descent
With $\lambda = 0$

Problem 2 Final Page



Plot of Mini Batch Gradient Descent for Problem 2 (Iteration vs. Objective Error)
With $\lambda = 0.01$



Plot of CPU Time vs. Objective Error for Problem 2 Mini Batch Gradient Descent
With $\lambda = 0$

Problem 2 Discussion:

We see that in all cases (excluding Mini Batch this will be discussed) we are generating a Theta vector that is minimizing the objective error. Because this is gradient descent we expect to see the objective error decrease and it indeed does.

For Batch Gradient Descent we see that it becomes accurate in over a long time but takes a lot of time between iterations

For Stochastic Gradient Descent we see that it does become quite accurate but not as accurate as Batch Gradient Descent. But the per iteration cost/time is incredibly small.

For Mini Batch Gradient Descent we almost have a happy medium between the 2.

Having the regularization constant in there makes the algorithm a lot more accurate than without

It minimizes the objective error quickly without sacrificing too much time for iterations but is still not as fast as Stochastic Gradient Descent or as accurate as Batch Gradient Descent.

For this problem the logistics regression does hold as $iterations \to \infty$

But for all of these runs we are using many more iterations than Linear Regression. This can be due to a number of reasons. The first one being improper implementation of Mini Batch specifically, but the others as well.

For this problem I did not do any pre-processing to remove outlier data. Because the data set was so small, to my eye I was not able to see any outliers. However this may not be the case and removing those outliers would have made my algorithms more accurate. This was the case for Problem 1.

I also think that due to that inclusion of bad data the Mini Batch algorithm took a significant hit. When running Stochastic Gradient only 1 sample is chosen per iteration. That is much more unlikely to include bad data as compared to the Mini Batch choosing 50/350 samples. Because Mini Batch depends on solely what it randomly chooses the algorithm appears to randomly work and not work in my case.

```
#Problem 1 HW 2 Code
#Saaif Ahmed - 661925946 - ahmeds7
import pandas as pd
import numpy as np
import openpyxl
import matplotlib.pyplot as plt
from time import process_time

#The cost function as defined in the homework
def linreg_cost(output, feature, theta):
    x=0
    for i in range(len(output)):
        x+=(output[i] - np.dot(feature[i],theta))**2
    x=x/(len(output))
    return x

#batch gradient descent update method by using matrices
#butter is just a placeholder term that came to mind
def update_theta_bgd(theta, lr, output,feature):
    butter = np.matmul(feature.transpose(),feature)
    butter = np.matmul(butter,theta.transpose())
    butter = butter - np.matmul(feature.transpose(),output.transpose())
    butter = (lr*2/len(output)) * butter
    butter = theta.transpose() - butter
    return butter.transpose()

#stochastic gradient descent. For 1 iteration
#uniformly randomly chooses an entry in the feature matrix
#to update theta
def update_theta_sgd(theta, lr, output, feature, k):
    i=0
    while i <k:
        index = np.random.randint(0,len(output),1)
        xn = feature[index]
        yn = output[index]
        test = (np.dot(theta, xn[0]) -yn)*xn[0]
        theta = theta - lr * test
        i+=1
    return theta

#Mini batch gradient descent. Uniformly randomly chooses 50
#samples of the whole dataset and performs gradient descent.
#Adapted to use the matrix methodology from batch gradient descent
def update_theta_mbgd(theta, lr, output, feature, k):
    index = []
    index.append(np.random.randint(0,len(output),k))
    features = feature[index,:]
    features = features[0]
    outputs = []
    for i in index:
        outputs.append(output[i])
    outputs = np.array(outputs)[0]
    butter = np.matmul(features.transpose(),features)
    butter = np.matmul(butter,theta.transpose())
    butter = butter - np.matmul(features.transpose(),outputs.transpose())
```

```
    butter = (lr*2/len(outputs)) * butter
    butter = theta.transpose() - butter

    return butter.transpose()


if __name__ == '__main__':
    #control for deciding which operation to run
    operation = float(input("input the number you are doing (1.1, 1.2, 1.3): "))

    #beginning of parsing. Reads in Excel file. Gets rid of Date and Time columns
    df = pd.read_excel('AirQualityUCI.xlsx')
    df = df.drop(['Date', 'Time'], axis=1)

    #inserts x^0 power column to the feature matrix
    df.insert(0,'col1',1)
    feature_matrix = np.array(df)

    #Preprocessing begins
    bad_index = []
    for i in range(len(feature_matrix)):
        count =0
        for j in range(len(feature_matrix[i])):
            if(feature_matrix[i][j] == -200):
                count +=1
        if count >=2:
            bad_index.append(i)
    #If a row has 2 or more -200 (bad sensors) in it remove that row
    feature_matrix = np.delete(feature_matrix,bad_index,0)

    #replacing the remaining -200 with 0
    for i in range(len(feature_matrix)):
        for j in range(len(feature_matrix[i])):
            if(feature_matrix[i][j] == -200):
                feature_matrix[i][j] =0

    #normalize the matrix by scaling by 1/1000
    #grab the benzene column and remove from feature matrix
    #make initial theta guess
    feature_matrix = 1/1000 * feature_matrix
    output_vector = feature_matrix[:,4]
    feature_matrix = np.delete(feature_matrix,4, axis=1)
    theta = np.random.default_rng(42).random((13))

    #Setup for plotting
    i=0
    iteration_index = []
    error =[]
    cpu = []

    #Run a number of iterations of Batch Gradient Descent
    if operation == 1.1:
        print("Running Problem 1 BGD")
        while i<100:
            t_start = process_time()
```

```
        theta = update_theta_bgd(theta,0.01,output_vector,feature_matrix)
        iteration_index.append(i)
        error.append(linreg_cost(output_vector,feature_matrix,theta))
        t_end = process_time()
        cpu.append(t_end-t_start)
        i+=1
    plt.title("Batch Gradient Descent")
    print("Finished P1 BGD")

#Run a number of iterations of Stochastic Gradient Descent
elif operation == 1.2:
    print("Running Problem 1 SGD")
    while i<100:
        t_start = process_time()
        theta = update_theta_sgd(theta,0.01,output_vector,feature_matrix, 1)
        iteration_index.append(i)
        error.append(linreg_cost(output_vector,feature_matrix,theta))
        t_end = process_time()
        cpu.append(t_end-t_start)
        i+=1
    plt.title("Stochastic Gradient Descent")
    print("Finished P1 SGD")

#Run a number of iterations of Mini Batch Gradient Descent
elif operation == 1.3:
    print("Running Problem 1 MBGD")
    while i<100:
        t_start = process_time()
        theta = update_theta_mbgd(theta,0.01,output_vector,feature_matrix,50)
        iteration_index.append(i)
        error.append(linreg_cost(output_vector,feature_matrix,theta))
        t_end = process_time()
        cpu.append(t_end-t_start)
        i+=1
    plt.title("Mini Batch Gradient Descent")
    print("Finished P1 SGD")

#Finish plotting and output a graph
#plt.xlabel("Iteration")
plt.xlabel("CPU Time")
plt.ylabel("Objective Error")
#plt.plot(np.array(iteration_index), np.array(error), color ="green")
plt.plot( np.array(cpu), np.array(error),color ="blue")
plt.show()
```

```
#Problem 2 HW 2 Code
#Saaif Ahmed - 661925946 - ahmeds7
import pandas as pd
import numpy as np
import openpyxl
import matplotlib.pyplot as plt
from time import process_time
import math


#The cost function of log regression as defined in the homework
def logreg_cost(feature, output, theta, lam):
    x = 0
    for i in range(len(output)):
        x+= math.log(1+math.exp(-1*output[i]*np.dot(feature[i],theta)))
    x = x/len(output)
    x+= (lam/2)*(np.linalg.norm(theta)**2)
    return x

#The batch gradient descent using the derived
#gradient of the loss equation in the homework
def logreg_bgd(feature, output, theta, lr ,lam):
    for i in range(len(theta)):
        sum = 0
        for j in range(len(output)):
            a = math.exp(-1*output[j] * np.dot(feature[j],theta))
            c = -1*output[j]*feature[j][i]
            sum+= (a*c)/(a+1)
        sum = sum/len(output)
        sum = sum + (lam)*theta[i] * (1/len(output))
        theta[i] = theta[i] - (lr/len(output))*sum
    return theta

#stochastic gradient descent. Same derivative but
#chooses 1 sample of the data set randomly
def logreg_sgd(feature, output, theta, lr, lam):
    index = np.random.randint(0,len(output))
    for i in range(len(theta)):
        sum = 0
        a = math.exp(-1*output[index] * np.dot(feature[index],theta))
        c = -1*output[index]*feature[index][i]
        sum+= (a*c)/(a+1)
        sum = sum/len(output)
        sum = sum + lam*theta[i] * (1/len(output))
        theta[i] = theta[i] - lr*sum
    return theta

#uniformly chooses 50 random components to make a subset
#of the whole dataset and runs batch gradient descent on that subset
def logreg_mbgd(feature, output, theta, lr, lam, k):
    index = []
    index.append(np.random.randint(0,len(output),k))
    features = feature[index,:]
    features = features[0]
    outputs = []
```

```
    for i in index[0]:
        outputs.append(output[i])
    for i in range(len(theta)):
        sum = 0
        for j in range(len(outputs)):
            a = math.exp(-1*outputs[j] * np.dot(features[j],theta))
            c = -1*output[j]*features[j][i]
            sum+= (a*c)/(a+1)
        sum = sum/50
        sum = sum + (lam)*theta[i] * (1/50)
        theta[i] = theta[i] - (lr/50)*sum
    return theta

if __name__ == '__main__':
    #parsing begins
    #deletes columns that are consistently 0
    #and adds in a colume of 1 to represent x^0
    df = pd.read_csv('ionosphere.csv', header=None)
    df = df.drop(df.columns[[1]], axis=1)
    df.insert(0,'col1',1)

    #does the mapping of b -> -1
    # and g -> 1
    feature_matrix = np.array(df)
    output_vector = feature_matrix[:,34]
    feature_matrix = np.delete(feature_matrix,34, axis=1)
    for i in range(len(output_vector)):
        if output_vector[i] == 'b':
            output_vector[i] = -1
        elif output_vector[i] == 'g':
            output_vector[i] = 1

    #initial theta guess and running of iterations
    theta = np.random.default_rng(42).random((34))
    i = 0
    iteration_index = []
    error =[]
    cpu = []
    t_start = process_time()
    while i < 5000:
        if(i %250 == 0):
            print(i)
            iteration_index.append(i)
            error.append(logreg_cost(feature_matrix,output_vector,theta,0.01))
            t_end = process_time()
            cpu.append(t_end-t_start)
            t_start = process_time()
        #theta = logreg_bgd(feature_matrix,output_vector,theta,1,0.01)
        #theta = logreg_sgd(feature_matrix,output_vector,theta,1,0.01)
        theta = logreg_mbgd(feature_matrix,output_vector,theta,1,0.01,50)

        i+=1

    #plot set up and output
    plt.title("Mini Batch Gradient Descent")
```

```
#plt.xlabel("Iteration")
plt.xlabel("CPU Time")
plt.ylabel("Objective Error")
#plt.plot(np.array(iteration_index), np.array(error), color ="green")
plt.plot( np.array(cpu), np.array(error),color ="blue")
plt.show()
```