

BOĞAZIÇI UNIVERSITY

NONLINEAR MODELS IN OPERATIONS RESEARCH
IE 440

Homework 6

Authors:

M. Akın Elden
Yunus Emre Karataş
Y. Harun Kıvrıl
Sefa Kayraklık

22 December 2019



Department of Industrial Engineering
Boğaziçi University

Introduction

The project is implemented using Python as the programming language. A function is fitted by using the training data in order to estimate the test data via the least square method with the steepest descent algorithm and the nonlinear regression with the neural network.

The source code used to import required dependencies, converting functions to lambda expressions:

```
1 import pandas as pd
2 import numpy as np
3 from sympy import Symbol, lambdify
4 import matplotlib.pyplot as plt
5
6 train_data = pd.read_csv('Input/training.dat', sep=' ', header=None, names=['x', 'y']);
7 test_data = pd.read_csv('Input/test.dat', sep=' ', header=None, names=['x', 'y']);
8
9 x_train = np.array(train_data['x'])
10 y_train = np.array(train_data['y'])
11
12 x_test = np.array(test_data['x'])
13 y_test = np.array(test_data['y'])
14
15 norm_train = np.array(train_data)
16 norm_train = (norm_train - norm_train.mean(0)) / norm_train.std(0)
17
18 # comment out below lines to use normalized data
19
20 #x_train = norm_train[:,0]
21 #y_train = norm_train[:,1]
22
23 w0 = Symbol("w0")
24 w1 = Symbol("w1")
25 w2 = Symbol("w2")
26
27 func_a_coefficients = np.array([np.sum(y_train**2), 1*np.size(y_train), np.sum((x_train
    **2)), np.sum(2*y_train), np.sum(2*y_train*x_train), np.sum(2*x_train)])
28 func_a_variables = np.array([1, w0**2, w1**2, -w0, -w1, w0*w1])
29
30 func_b_coefficient = np.array([np.sum(y_train**2), 1*np.size(y_train), np.sum(x_train
    **2), np.sum(x_train**4), np.sum(2*y_train), np.sum(2*y_train*x_train), np.sum(2*
    y_train*x_train**2), np.sum(2*x_train), np.sum(2*x_train**2), np.sum(2*x_train**3)])
31 func_b_variables = np.array([1, w0**2, w1**2, w2**2, -w0, -w1, -w2, w0*w1, w0*w2, w1*w2
    ])
32
33 func_a = np.sum(func_a_coefficients*func_a_variables)
34 f_a = lambdify([w0, w1], func_a, "numpy")
35 gf_a = lambdify([w0, w1], func_a.diff([w0, w1]), "numpy")
36 grad_fa = lambda x_arr : np.array(gf_a(x_arr), 'float64').reshape(1,len(x_arr))
37
38 func_b = np.sum(func_b_coefficient*func_b_variables)
39 f_b = lambdify([w0, w1, w2], func_b, "numpy")
```

```

40 gf_b = lambdify([[w0, w1, w2]], func_b.diff([[w0, w1, w2]]), "numpy")
41 grad_fb = lambda x_arr : np.array(gf_b(x_arr), 'float64').reshape(1,len(x_arr))
42
43 regA = lambda w_s, x_arr : x_arr * w_s[1,0] + w_s[0,0]
44 regB = lambda w_s, x_arr : x_arr**2 * w_s[2,0] + x_arr * w_s[1,0] + w_s[0,0]

```

Some useful functions for output table and plot construction:

```

1 def plotRegressionGraph(data, regFunc, w_star, title, name="graph"):
2     xmin = data[:,0].min()
3     xmax = data[:,0].max()
4     t1 = np.arange(xmin-1, xmax+1, 0.1)
5     plt.figure()
6     plt.plot(t1, regFunc(w_star, t1), 'b-', label='Regression line')
7     plt.scatter(data[:,0], data[:,1], color="black", label="Data points")
8     plt.title(title)
9     plt.legend()
10    plt.savefig("{0}.png".format(name))
11
12 np_str = lambda x_k : np.array2string(x_k.reshape(len(x_k)), precision=3, separator=',',
13    )
14 f_str = lambda x : "{0:.4f}".format(x)
15
16 class OutputTable:
17     def __init__(self):
18         self.table = pd.DataFrame([], columns=['k', 'x^k', 'f(x^k)', 'd^k', 'a^k', 'x^k+1',
19             ])
20     def add_row(self, k, xk, fxk, dk, ak, xkp):
21         self.table.loc[len(self.table)] = [k, np_str(xk), f_str(fxk.item()), np_str(dk),
22             ak, np_str(xkp)]
23     def print_latex(self):
24         print(self.table.to_latex(index=False))

```

1 Least Square Method with Steepest Descent

In order to fit a function to the training data, the squared error function between the estimated points and the actual data is minimized.

For the minimization problem, the steepest descent with exact line search is implemented. In the exact line search, the following procedure is implemented. First, the bisection method is used to obtain a narrow range for the optimum solution to pass it to the Newton method. Then, the found point with the its range are given to the Newton method with much smaller epsilon value. This method is implemented in order to speed up the line search algorithm by taking the advantage of quadratic convergence of the Newton method.

The source code for the exact line search algorithm is provided below:

```

1 def BisectionMethod(f,epsilon, a=-100,b=100) :
2     iteration=0

```

```

3   while (b - a) >= epsilon:
4       x_1 = (a + b) / 2
5       fx_1 = f(x_1)
6       if f(x_1 + epsilon) <= fx_1:
7           a = x_1
8       else:
9           b = x_1
10      iteration+=1
11  x_star = (a+b)/2
12  return x_star
13
14  def NewtonsMethod(df, ddf, x_0, epsilon, a, b):
15      iteration = 0
16      while True:
17          dfx0 = df(x_0)
18          ddfx0 = ddf(x_0)
19          x_1 = x_0-dfx0/ddfx0
20          iteration +=1
21          if abs(x_0-x_1)<epsilon:
22              break
23          if x_1<a or x_1>b:
24              break
25          x_0 = x_1
26      x_star = x_0
27      return x_star
28
29  def ExactLineSearch(f, x0, d, eps=10**(-10)):
30      alpha = Symbol('alpha')
31      function_alpha = f(np.array(x0)+alpha*np.array(d)).item()
32      f_alp = lambdify(alpha, function_alpha)
33      bisecEps = 10**(-4)
34      alp_star = BisectionMethod(f_alp, epsilon=bisecEps)
35      df_alp = lambdify(alpha, function_alpha.diff(alpha))
36      ddf_alp = lambdify(alpha, function_alpha.diff(alpha).diff(alpha))
37      alp_star = NewtonsMethod(df_alp, ddf_alp, alp_star, eps, alp_star-bisecEps, alp_star
38                          +bisecEps)
39      return alp_star

```

The source code for the steepest descent algorithm which is adopted from the previous homework is provided below:

```

1  def steepestDescentMethod(f, grad_f, x_0, descentEpsilon, exactLineEpsilon=10**(-10)):
2      xk = np.array(x_0).reshape(-1,1)
3      k = 0
4      stop = False
5      output = OutputTable()
6      while(stop == False):
7          d = - np.transpose(grad_f(xk))
8          if(np.linalg.norm(d) < descentEpsilon):
9              stop = True
10         else:
11             a = ExactLineSearch(f,xk,d, exactLineEpsilon)
12             xkp = xk + a*d
13             output.add_row(k, xk, f(xk), d, a, xkp)

```

```

14         k += 1
15         xk = xkp
16         if(k>100):
17             break
18     output.add_row(k,xk,f(xk),d,None,np.array([]))
19     print("Total iteration : {0}".format(k))
20     return xk, f(xk).item(), output

```

1.1 Part A:

For this part, the following linear regression model is used:

$$\mathbf{y} = w_1 \mathbf{x} + w_0$$

, where \mathbf{y} and \mathbf{x} are the given training data. So, the error function which is the objective function of the minimization problem is:

$$\sum_{i=1}^{N_{tra}} (y_i - w_0 - w_1 x_i)^2$$

When the steepest descent algorithm is applied to the given objective function with the initial points, $x_0 = [0, 0]$ and $\varepsilon = 0.005$, it finds w_0, w_1 as 113.35, 0.74, respectively. After obtaining the weights of the linear regression model, the model is used to estimate the test output, \mathbf{y} , given the test input, \mathbf{x} .

$$TrainSSE : 3835642.725$$

$$TestMSE : 50246.685$$

$$Test s^2 : 10689612467.303$$

The plots for the training data with the regression function, and test data with the regression function are shown in the following figure.

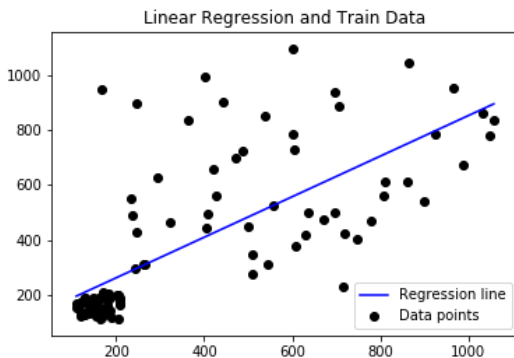


Figure 1: The linear regression and train data

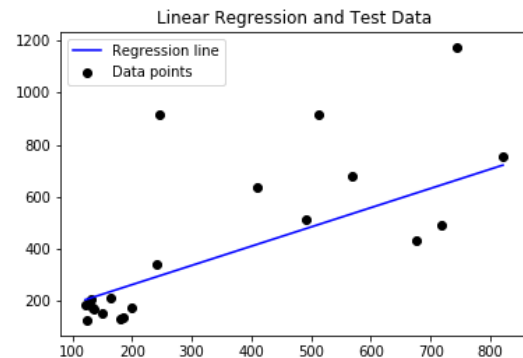


Figure 2: The linear regression and test data

1.2 Part B:

For this part, the following polynomial regression model is used:

$$\mathbf{y} = w_2 \mathbf{x}^2 + w_1 \mathbf{x} + w_0$$

, where \mathbf{y} and \mathbf{x} are the given training data. So, the error function which is the objective function of the minimization problem is:

$$\sum_{i=1}^{N_{tra}} (y_i - w_0 - w_1 x_i - w_2 x_i^2)^2$$

When the steepest descent algorithm is applied to the given objective function with the initial points, $x_0 = [0, 0, 0]$ and $\varepsilon = 0.005$, it finds w_0, w_1, w_2 as 0.0052, 1.495, -0.00076 , respectively. Actually, these values are not the final solution of the algorithm, these are the 100th iteration's solution. This approach is considered since as getting closer to the minimum point, the algorithm starts to take so small steps. This causes lot of iterations to reach the minimum point, and the algorithm spends so much time to obtain the minimum point.

In fact, 100 iterations are enough to determine the weights of the regression model, since if the given function is solved analytically, the resulting objective function is so close to the one that is obtained by the 100 iterations of the steepest descent algorithm. Note that the number 100 is for the given problem, it can be different for another problem.

The source code for the analytic solution of the problem is provided below:

```
1 X = np.array([np.ones(len(x_train)), x_train, x_train**2]).reshape(3, len(x_train)).T
2 Y = y_train.reshape(len(y_train),1)
3 W = np.linalg.inv(X.T @ X) @ X.T @ Y
4 SSE = np.sum((Y - X @ W)**2)
5 x_test = np.array(test_data['x'])
6 y_test = np.array(test_data['y'])
7 X_test = np.array([np.ones(len(x_test)), x_test, x_test**2]).reshape(3, len(x_test)).T
8 Y_test = y_test.reshape(len(y_test),1)
9 SSE_test = np.sum((Y_test - X_test @ W)**2)
```

After obtaining the weights of the linear regression model, the model is used to estimate the test output, \mathbf{y} , given the test input, \mathbf{x} .

The plots for the training data with the regression function, and test data with the regression function are shown in the following figure.

TrainSSE : 3506233.112

TestMSE : 43743.010

Test s² : 8606111510.643

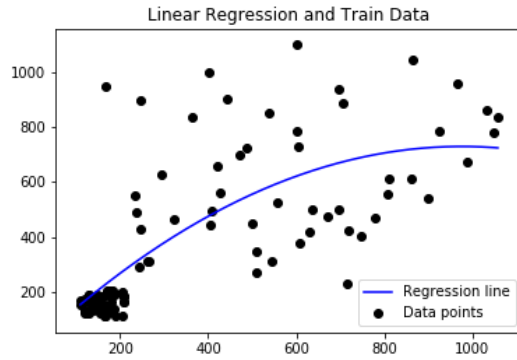


Figure 3: The polynomial regression and train data

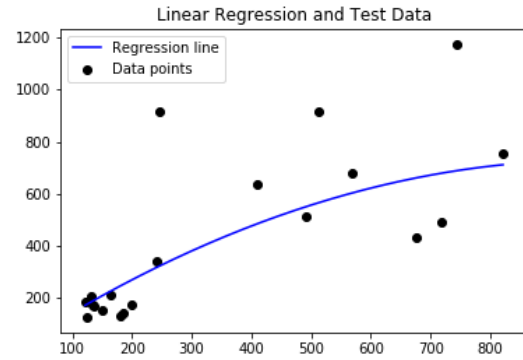


Figure 4: The polynomial regression and test data

2 Nonlinear Regression with the Neural Network

For the non-linear regression; a function for a perceptron with one output unit, one hidden layer with variable hidden unit size and variable input unit size is written. The codes of the function is given below.

```

1 sigmoidalFunc = lambda output_array : 1 / (1 + np.exp(-output_array))
2 sigmoidalDeriv = lambda hiddenlayer : hiddenlayer * (1 - hiddenlayer)
3
4 def backpropagation(trainingData, hiddenLayerSize, alpha = 0.5, momentum = 0.9, epsilon
   = 0.001, seed = 440):
5     np.random.seed(seed)
6     t = 0
7     patterns = np.copy(trainingData)
8     patterns = np.insert(patterns, 0, -1, axis=1) # x0 = -1 unit is added
9     P = np.size(patterns, 0) # pattern size
10    I = 1 # output unit size
11    K = np.size(patterns, 1) - I # input layer size
12    J = hiddenLayerSize + 1 # h0 = -1 is added
13    w_matrix = np.random.rand(J, K) # weights between input and hidden layer (we will
   exclude first row in the result since h0 is excluded)
14    W_matrix = np.random.rand(I, J) # weight between hidden and output layer
15    while(alpha >= epsilon):
16        np.random.shuffle(patterns)
17        x = np.transpose(patterns[:, :-1]).reshape(K, -1)
18        y = patterns[:, -1]
19        H = np.zeros(J)
20        H[0] = -1 # h0 is equal to -1
21        O = np.zeros_like(y)
22        for p in range(P):
23            for j in range(1, J):
24                hj = np.sum(w_matrix[j] * x[:, p])
25                H[j] = sigmoidalFunc(hj)

```

```

26         for i in range(I):
27             o = np.sum(W_matrix[i] * H)
28             O[p] = o # linear function  $g(x) = x$ 
29             S_0 = 0 # since there is only one output unit
30             S_H = np.zeros_like(H)
31             for i in range(I):
32                 S_0 = 1 * (y[p] - O[p])
33             for j in range(1,J):
34                 S_H[j] = sigmoidalDeriv(H[j]) * np.sum(W_matrix[0,j] * S_0)
35             for j in range(J):
36                 dWj = alpha * S_0 * H[j]
37                 W_matrix[0,j] += dWj
38             for k in range(K):
39                 dwk = alpha * S_H * x[k,p]
40                 w_matrix[:,k] += dwk
41         alpha *= momentum
42         t += 1
43         actualHiddens = sigmoidalFunc(w_matrix @ x)
44         actualHiddens[0,:] = -1 #  $h_1, \dots, h_j$ 
45         actualOutputMatrix = W_matrix @ actualHiddens #  $o_1, \dots, o_i$ 
46         error = np.sum(np.square(y - actualOutputMatrix))
47         #print("Iteration {0} : error = {1}".format(t,error))
48     w_matrix = w_matrix[1:] # first row is removed since it corresponds to  $H_0$ 
49     return w_matrix, W_matrix, error
50
51 def averageError(w_matrix, W_matrix, test_data):
52     inputLayers = np.transpose(np.insert(test_data, 0, -1, axis=1)[:,:-1]) #  $h_1, \dots, h_j$ 
53     desiredOutputs = test_data[:,-1].reshape(-1,1)
54     actualHiddens = sigmoidalFunc(w_matrix @ inputLayers)
55     actualOutputMatrix = W_matrix @ np.insert(actualHiddens, 0, -1, axis=0) #  $o_1, \dots, o_i$ 
56     squareResiduals = np.square(desiredOutputs - np.transpose(actualOutputMatrix))
57     sse = np.sum(squareResiduals)
58     mse = sse / np.size(desiredOutputs)
59     variance = np.sum(np.square(mse-squareResiduals)) / (np.size(desiredOutputs) - 1)
60     return mse, variance
61
62 def hiddenUnit(train_data, test_data, Jq = 3, epsilon = 0.001, seed = 440):
63     train = np.array(train_data)
64     test = np.array(test_data)
65     q = 1
66     Et = np.infty
67     while(True):
68         patterns = np.copy(train)
69         w, W, total_error = backpropagation(patterns, Jq, epsilon=epsilon, seed = seed)
70         Etp, var = averageError(w, W, test)
71         print("{0} hidden units : MSE = {1} , variance = {2}".format(Jq,Etp,var))
72         if(Etp >= Et):
73             break
74         Jq += 1
75         q += 1
76         Et = Etp
77     return Jq-1, Et

```


2.1 Part A:

For part A, the training data directly passed to the function by setting initially $hiddenLayerSize = 3$. And the number of hidden units is increased if increasing the number decreases mean square error between the test data, and estimated values.

```
1 | hiddenUnit(train_data, test_data, epsilon=0.001, seed = 440)
```

$TrainSSE : 7953567.819$

$TestMSE : 99539.0395$

$Tests^2 : 20795803720.511$

2.2 Part B:

For part B, the training data manipulated to include x^2 as a column and passed to the function by setting initially $hiddenLayerSize = 3$. And the number of hidden units is increased if increasing the number decreases mean square error between the test data, and estimated values.

```
1 | train_d = np.insert(np.array(train_data), 1, np.square(train_data['x']), axis=1)
2 | test_d = np.insert(np.array(test_data), 1, np.square(test_data['x']), axis=1)
3 | hiddenUnit(train_d, test_d)
```

$TrainSSE : 7954322.338$

$TestMSE : 99377.589$

$Test s^2 : 20473670331.017$

3 Comparison of the Methods

Performances of the methods:

Method	Training SSE $\sum_{i=1}^{N_{tra}} e_i^2$	Test MSE $\frac{1}{N_{test}} \sum_{i=1}^{N_{test}} e_i^2$	s^2 for Test MSE $\frac{1}{N_{test}-1} \sum_{i=1}^{N_{test}} (TestMSE - e_i^2)^2$
1.(a)	$3.835 * 10^6$	$5.025 * 10^4$	$1.069 * 10^{10}$
1.(b)	$3.506 * 10^6$	$4.374 * 10^4$	$0.861 * 10^{10}$
2.(a)	$7.953 * 10^6$	$9.953 * 10^4$	$2.057 * 10^{10}$
2.(b)	$7.954 * 10^6$	$9.937 * 10^4$	$2.141 * 10^{10}$

It can be seen from the table that the training MSE ($SSE/100$) is smaller than the test MSE for all implemented methods since the fitted model is being constructed for the training data not the test data. This doesn't mean that the model can not be used for the test data because the training data and test data are correlated each other, namely their behaviours are similar to each other.

The steepest descent with exactly line search approach which is implemented at the first part gives better results than the neural network approach which is implemented at the second part since the latter uses inaccurate line search to speed up the algorithm, yet this reduces its performance. Fitting the model using not only the x but also x^2 increases the performance of the algorithms.

4 Appendix

The complete source code:

```
1  #!/usr/bin/env python
2  # coding: utf-8
3
4  # # Homework 6
5
6  # In[1]:
7
8
9  import pandas as pd
10 import numpy as np
11 from sympy import Symbol, lambdify
12 import matplotlib.pyplot as plt
13
14
15 # In[2]:
16
17
18 train_data = pd.read_csv('Input/training.dat', sep=' ', header=None, names=['x', 'y']);
19 test_data = pd.read_csv('Input/test.dat', sep=' ', header=None, names=['x', 'y']);
20
21 x_train = np.array(train_data['x'])
22 y_train = np.array(train_data['y'])
23
24 x_test = np.array(test_data['x'])
25 y_test = np.array(test_data['y'])
26
27 norm_train = np.array(train_data)
28 norm_train = (norm_train - norm_train.mean(0)) / norm_train.std(0)
29
30 # comment out below lines to use normalized data
31
32 #x_train = norm_train[:,0]
33 #y_train = norm_train[:,1]
34
35 w0 = Symbol("w0")
36 w1 = Symbol("w1")
37 w2 = Symbol("w2")
38
39 func_a_coefficients = np.array([np.sum(y_train**2), 1*np.size(y_train), np.sum((x_train
    **2)), np.sum(2*y_train), np.sum(2*y_train*x_train), np.sum(2*x_train)])
40 func_a_variables = np.array([1, w0**2, w1**2, -w0, -w1, w0*w1])
41
```

```

42 func_b_coefficient = np.array([np.sum(y_train**2), 1*np.size(y_train), np.sum(x_train
    **2), np.sum(x_train**4), np.sum(2*y_train), np.sum(2*y_train*x_train), np.sum(2*
    y_train*x_train**2), np.sum(2*x_train), np.sum(2*x_train**2), np.sum(2*x_train**3)])
43 func_b_variables = np.array([1, w0**2, w1**2, w2**2, -w0, -w1, -w2, w0*w1, w0*w2, w1*w2
    ])
44
45 func_a = np.sum(func_a_coefficients*func_a_variables)
46 f_a = lambdify([[w0, w1]], func_a, "numpy")
47 gf_a = lambdify([[w0, w1]], func_a.diff([[w0, w1]]), "numpy")
48 grad_fa = lambda x_arr : np.array(gf_a(x_arr), 'float64').reshape(1,len(x_arr))
49
50 func_b = np.sum(func_b_coefficient*func_b_variables)
51 f_b = lambdify([[w0, w1, w2]], func_b, "numpy")
52 gf_b = lambdify([[w0, w1, w2]], func_b.diff([[w0, w1, w2]]), "numpy")
53 grad_fb = lambda x_arr : np.array(gf_b(x_arr), 'float64').reshape(1,len(x_arr))
54
55
56 # ### Useful Functions
57
58 # In[3]:
59
60
61 regA = lambda w_s, x_arr : x_arr * w_s[1,0] + w_s[0,0]
62 regB = lambda w_s, x_arr : x_arr**2 * w_s[2,0] + x_arr * w_s[1,0] + w_s[0,0]
63
64
65 # In[4]:
66
67
68 def plotRegressionGraph(data, regFunc, w_star, title, name="graph"):
69     xmin = data[:,0].min()
70     xmax = data[:,0].max()
71     t1 = np.arange(xmin-1, xmax+1, 0.1)
72     plt.figure()
73     plt.plot(t1, regFunc(w_star, t1), 'b-', label='Regression line')
74     plt.scatter(data[:,0], data[:,1], color="black", label="Data points")
75     plt.title(title)
76     plt.legend()
77     plt.savefig("{0}.png".format(name))
78
79
80 # In[5]:
81
82
83 np_str = lambda x_k : np.array2string(x_k.reshape(len(x_k)), precision=3, separator=',',
    )
84
85 f_str = lambda x : "{0:.4f}".format(x)
86
87
88 # In[6]:
89
90
91 class OutputTable:

```

```

92     def __init__(self):
93         self.table = pd.DataFrame([], columns=['k', 'xk', 'f(xk)', 'dk', 'ak', 'xk+1'])
94     def add_row(self, k, xk, fxk, dk, ak, xkp):
95         self.table.loc[len(self.table)] = [k, np_str(xk), f_str(fxk.item()), np_str(dk),
96         ak, np_str(xkp)]
97     def print_latex(self):
98         print(self.table.to_latex(index=False))
99
100 # ### Part A : Least Square Method with Steepest Descent
101
102 # ### Exact Line Search
103
104 # In[7]:
105
106
107 def BisectionMethod(f, epsilon, a=-100, b=100) :
108     iteration=0
109     while (b - a) >= epsilon:
110         x_1 = (a + b) / 2
111         fx_1 = f(x_1)
112         if f(x_1 + epsilon) <= fx_1:
113             a = x_1
114         else:
115             b = x_1
116         iteration+=1
117     x_star = (a+b)/2
118     return x_star
119
120 def NewtonsMethod(df, ddf, x_0, epsilon, a, b):
121     iteration = 0
122     while True:
123         dfx0 = df(x_0)
124         ddfx0 = ddf(x_0)
125         x_1 = x_0 - dfx0/ddfx0
126         iteration += 1
127         if abs(x_0 - x_1) < epsilon:
128             break
129         if x_1 < a or x_1 > b:
130             break
131         x_0 = x_1
132     x_star = x_0
133     return x_star
134
135 def ExactLineSearch(f, x0, d, eps=10**(-10)):
136     alpha = Symbol('alpha')
137     function_alpha = f(np.array(x0) + alpha*np.array(d)).item()
138     f_alp = lambdify(alpha, function_alpha)
139     bisecEps = 10**(-4)
140     alp_star = BisectionMethod(f_alp, epsilon=bisecEps)
141     df_alp = lambdify(alpha, function_alpha.diff(alpha))
142     ddf_alp = lambdify(alpha, function_alpha.diff(alpha).diff(alpha))

```

```

143     alp_star = NewtonsMethod(df_alp, ddf_alp, alp_star, eps, alp_star-bisecEps, alp_star
144         +bisecEps)
145     return alp_star
146
147 # ### Steepest Descent Method
148
149 # In[8]:
150
151
152 def steepestDescentMethod(f, grad_f, x_0, descentEpsilon, exactLineEpsilon=10**(-10)):
153     xk = np.array(x_0).reshape(-1,1)
154     k = 0
155     stop = False
156     output = OutputTable()
157     while(stop == False):
158         d = - np.transpose(grad_f(xk))
159         if(np.linalg.norm(d) < descentEpsilon):
160             stop = True
161         else:
162             a = ExactLineSearch(f,xk,d, exactLineEpsilon)
163             xkp = xk + a*d
164             output.add_row(k, xk, f(xk), d, a, xkp)
165             k += 1
166             xk = xkp
167             if(k>100):
168                 break
169     output.add_row(k,xk,f(xk),d,None,np.array([]))
170     print("Total iteration : {0}".format(k))
171     return xk, f(xk).item(), output
172
173
174 # In[9]:
175
176
177 ws_a, fs_a, outputs_a = steepestDescentMethod(f_a, grad_fa, [0,0], 0.005)
178
179 SSE_train_a = fs_a
180 MSE_test_a = np.sum((y_test-regA(ws_a,x_test))**2)/np.size(y_test)
181 var_test_a = np.sum((MSE_test_a-(y_test-regA(ws_a,x_test))**2)**2)/(np.size(y_test)-1)
182 ws_a, SSE_train_a, MSE_test_a, var_test_a
183
184
185 # In[10]:
186
187
188 plotRegressionGraph(np.array(train_data), regA, ws_a, "Linear Regression and Train Data
189     ", "part1a_train")
190
191 # In[11]:
192
193

```

```

194 plotRegressionGraph(np.array(test_data), regA, ws_a, "Linear Regression and Test Data",
    "part1a_test")
195
196
197 # In[12]:
198
199
200 ws_b, fs_b, outputs_b = steepestDescentMethod(f_b, grad_fb, [0,0,0], 0.005)
201
202 SSE_train_b = fs_b
203 MSE_test_b = np.sum((y_test-regB(ws_b,x_test))**2)/np.size(y_test)
204 var_test_b = np.sum((MSE_test_b-(y_test-regB(ws_b,x_test))**2)**2)/(np.size(y_test)-1)
205 ws_b, SSE_train_b, MSE_test_b, var_test_b
206
207
208 # In[13]:
209
210
211 plotRegressionGraph(np.array(train_data), regB, ws_b, "Linear Regression and Train Data
    ", "part1b_train")
212
213
214 # In[14]:
215
216
217 plotRegressionGraph(np.array(test_data), regB, ws_b, "Linear Regression and Test Data",
    "part1b_test")
218
219
220 # ## Part B : Neural Network
221
222 # In[15]:
223
224
225 sigmoidalFunc = lambda output_array : 1 / (1 + np.exp(-output_array))
226 sigmoidalDeriv = lambda hiddenlayer : hiddenlayer * (1 - hiddenlayer)
227
228
229 # In[16]:
230
231
232 def backpropagation(trainingData, hiddenLayerSize, alpha = 0.5, momentum = 0.9, epsilon
    = 0.001, seed = 440):
233     np.random.seed(seed)
234     t = 0
235     patterns = np.copy(trainingData)
236     patterns = np.insert(patterns, 0, -1, axis=1) # x0 = -1 unit is added
237     P = np.size(patterns, 0) # pattern size
238     I = 1 # output unit size
239     K = np.size(patterns, 1) - I # input layer size
240     J = hiddenLayerSize + 1 # h0 = -1 is added
241     w_matrix = np.random.rand(J, K) # weights between input and hidden layer (we will
        exclude first row in the result since h0 is excluded)
242     W_matrix = np.random.rand(I, J) # weight between hidden and output layer

```

```

243 while(alpha >= epsilon):
244     np.random.shuffle(patterns)
245     x = np.transpose(patterns[:, :-1]).reshape(K, -1)
246     y = patterns[:, -1]
247     H = np.zeros(J)
248     H[0] = -1 # h0 is equal to -1
249     O = np.zeros_like(y)
250     for p in range(P):
251         for j in range(1, J):
252             hj = np.sum(w_matrix[j] * x[:, p])
253             H[j] = sigmoidalFunc(hj)
254         for i in range(I):
255             o = np.sum(W_matrix[i] * H)
256             O[p] = o # linear function g(x) = x
257         S_0 = 0 # since there is only one output unit
258         S_H = np.zeros_like(H)
259         for i in range(I):
260             S_0 = 1 * (y[p] - O[p])
261         for j in range(1, J):
262             S_H[j] = sigmoidalDeriv(H[j]) * np.sum(W_matrix[0, j] * S_0)
263         for j in range(J):
264             dWj = alpha * S_0 * H[j]
265             W_matrix[0, j] += dWj
266         for k in range(K):
267             dwk = alpha * S_H * x[k, p]
268             w_matrix[:, k] += dwk
269         alpha *= momentum
270         t += 1
271         actualHiddens = sigmoidalFunc(w_matrix @ x)
272         actualHiddens[0, :] = -1 # h1, ..., hj
273         actualOutputMatrix = W_matrix @ actualHiddens # o1, ..., oi
274         error = np.sum(np.square(y - actualOutputMatrix))
275         #print("Iteration {0} : error = {1}".format(t, error))
276     w_matrix = w_matrix[1:] # first row is removed since it corresponds to H0
277     return w_matrix, W_matrix, error
278
279
280 # In[17]:
281
282
283 def backpropagationWithMatrix(patterns, hiddenLayerSize, alpha = 0.5, momentum = 0.9,
284     epsilon = 0.001, seed = 440):
285     np.random.seed(seed)
286     t = 0
287     P = np.size(patterns, 0)
288     w_matrix = np.random.rand(hiddenLayerSize, np.size(patterns, 1))*1 # patterns data
289     # includes y values, its column size is selected since we will add x0 to input
290     # layer
291     W_matrix = np.random.rand(1, hiddenLayerSize+1)*1 # we will add h0 to hidden layer
292     while(alpha > epsilon):
293         np.random.shuffle(patterns)
294         desiredOutputs = patterns[:, -1].reshape(-1, 1)
295         inputLayers = np.transpose(np.insert(patterns, 0, -1, axis=1)[:, :-1]) # x0 is
296         # added to all patterns and its value is -1, output values are excluded

```

```

293     hiddenLayer = np.zeros((hiddenLayerSize+1, 1)) # hiddenlayersize doesn't include
           h0 so it's added
294     hiddenLayer[0,:] = -1 # h0 is equal to -1
295     actualOutput = np.zeros_like(desiredOutputs)
296     for p in range(P):
297         hiddenLayer[1:] = sigmoidalFunc(w_matrix @ inputLayers[:,p].reshape(-1,1))
298         actualOutput[p] = W_matrix @ hiddenLayer
299         # since the function is linear, net output is equal to actual output
300         S_output = (1 * (desiredOutputs[p] - actualOutput[p])).reshape(-1,1)
301         S_hidden = (sigmoidalDeriv(hiddenLayer[1:]) * (np.transpose(W_matrix[:,1:]) @
           S_output)).reshape(-1,1)
302         delta_W = alpha * S_output @ np.transpose(hiddenLayer)
303         W_matrix += delta_W
304         delta_w = alpha * S_hidden @ np.transpose(inputLayers[:,p].reshape(-1,1))
305         w_matrix += delta_w
306     alpha = momentum * alpha
307     t += 1
308     actualHiddens = sigmoidalFunc(w_matrix @ inputLayers) # h1, ..., hj
309     actualOutputMatrix = W_matrix @ np.insert(actualHiddens, 0, -1, axis=0) # o1,
           ..., oi
310     error = np.sum(np.square(desiredOutputs - np.transpose(actualOutputMatrix)))
311     #print("Iteration {0} : error = {1}".format(t,error))
312     return w_matrix, W_matrix, error
313
314
315 # In[18]:
316
317
318 patterns = np.array(train_data)
319 backpropagation(patterns, 3, seed=440)
320
321
322 # In[19]:
323
324
325 patterns = np.array(train_data)
326 backpropagationWithMatrix(patterns, 3, seed=50)
327
328
329 # In[20]:
330
331
332 patterns2 = np.insert(np.array(train_data), 1, np.square(train_data['x']), axis=1)
333 backpropagation(patterns2, 3)
334
335
336 # In[21]:
337
338
339 def averageError(w_matrix, W_matrix, test_data):
340     inputLayers = np.transpose(np.insert(test_data, 0, -1, axis=1)[:,:-1]) # h1, ..., hj
341     desiredOutputs = test_data[:,-1].reshape(-1,1)
342     actualHiddens = sigmoidalFunc(w_matrix @ inputLayers)

```



```

343     actualOutputMatrix = W_matrix @ np.insert(actualHiddens, 0, -1, axis=0) # o1, ...,
        oi
344     squareResiduals = np.square(desiredOutputs - np.transpose(actualOutputMatrix))
345     sse = np.sum(squareResiduals)
346     mse = sse / np.size(desiredOutputs)
347     variance = np.sum(np.square(mse-squareResiduals)) / (np.size(desiredOutputs) - 1)
348     return mse, variance
349
350
351 # In[22]:
352
353
354 def hiddenUnit(train_data, test_data, Jq = 3, epsilon = 0.001, seed = 440):
355     train = np.array(train_data)
356     test = np.array(test_data)
357     q = 1
358     Et = np.infty
359     while(True):
360         patterns = np.copy(train)
361         w, W, total_error = backpropagation(patterns, Jq, epsilon=epsilon, seed = seed)
362         Etp, var = averageError(w, W, test)
363         print("{0} hidden units : MSE = {1} , variance = {2}".format(Jq,Etp,var))
364         if(Etp >= Et):
365             break
366         Jq += 1
367         q += 1
368         Et = Etp
369     return Jq-1, Et
370
371
372 # In[23]:
373
374
375 hiddenUnit(train_data, test_data, epsilon=0.001, seed = 440)
376
377
378 # In[24]:
379
380
381 train_d = np.insert(np.array(train_data), 1, np.square(train_data['x']), axis=1)
382 test_d = np.insert(np.array(test_data), 1, np.square(test_data['x']), axis=1)
383 hiddenUnit(train_d, test_d)
384
385 # In[ ]:

```