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:

```
import pandas as pd
import numpy as np
from sympy import Symbol, lambdify
import matplotlib.pyplot as plt

train_data = pd.read_csv(’Input/training.dat’, sep=’ ’, header=None, names=[’x’, ’y’]);
test_data = pd.read_csv(’Input/test.dat’, sep=’ ’, header=None, names=[’x’, ’y’]);

x_train = np.array(train_data[’x’])
y_train = np.array(train_data[’y’])

x_test = np.array(test_data[’x’])
y_test = np.array(test_data[’y’])

norm_train = np.array(train_data)
norm_train = (norm_train - norm_train.mean(0)) / norm_train.std(0)

# comment out below lines to use normalized data

#x_train = norm_train[:,0]
#y_train = norm_train[:,1]

w0 = Symbol("w0")
w1 = Symbol("w1")
w2 = Symbol("w2")

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)])
func_a_variables = np.array([1, w0**2, w1**2, -w0, -w1, w0*w1])

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)])
func_b_variables = np.array([1, w0**2, w1**2, w2**2, -w0, -w1, -w2, w0*w1, w0*w2, w1*w2])

func_a = np.sum(func_a_coefficients*func_a_variables)
f_a = lambdify([[w0, w1]], func_a, "numpy")
gf_a = lambdify([[w0, w1]], func_a.diff([[w0, w1]]), "numpy")
grad_fa = lambda x_arr : np.array(gf_a(x_arr), ’float64’).reshape(1,len(x_arr))

func_b = np.sum(func_b_coefficient*func_b_variables)
f_b = lambdify([[w0, w1, w2]], func_b, "numpy")
```

```
gf_b = lambdify([[w0, w1, w2]], func_b.diff([[w0, w1, w2]]), "numpy")
grad_fb = lambda x_arr : np.array(gf_b(x_arr), 'float64').reshape(1,len(x_arr))

regA = lambda w_s, x_arr : x_arr * w_s[1,0] + w_s[0,0]
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:

```
def plotRegressionGraph(data, regFunc, w_star, title, name="graph"):
    xmin = data[:,0].min()
    xmax = data[:,0].max()
    t1 = np.arange(xmin-1, xmax+1, 0.1)
    plt.figure()
    plt.plot(t1, regFunc(w_star, t1), 'b-', label='Regression line')
    plt.scatter(data[:,0], data[:,1], color="black", label="Data points")
    plt.title(title)
    plt.legend()
    plt.savefig("{0}.png".format(name))

np_str = lambda x_k : np.array2string(x_k.reshape(len(x_k)), precision=3, separator=',')

f_str = lambda x : "{0:.4f}".format(x)

class OutputTable:
    def __init__(self):
        self.table = pd.DataFrame([],columns=['k', 'x^k', 'f(x^k)', 'd^k', 'a^k', 'x^k+1'])
    def add_row(self, k, xk, fxk, dk, ak, xkp):
        self.table.loc[len(self.table)] = [k, np_str(xk), f_str(fxk.item()), np_str(dk), ak, np_str(xkp)]
    def print_latex(self):
        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:

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

```
    while (b - a) >= epsilon:
        x_1 = (a + b) / 2
        fx_1 = f(x_1)
        if f(x_1 + epsilon) <= fx_1:
            a = x_1
        else:
            b = x_1
        iteration+=1
    x_star = (a+b)/2
    return x_star

def NewtonsMethod(df, ddf, x_0, epsilon, a, b):
    iteration = 0
    while True:
        dfx0 = df(x_0)
        ddfx0 = ddf(x_0)
        x_1 = x_0-dfx0/ddfx0
        iteration +=1
        if abs(x_0-x_1)<epsilon:
            break
        if x_1<a or x_1>b:
            break
        x_0 = x_1
    x_star = x_0
    return x_star

def ExactLineSearch(f, x0, d, eps=10**(-10)):
    alpha = Symbol('alpha')
    function_alpha = f(np.array(x0)+alpha*np.array(d)).item()
    f_alp = lambdify(alpha, function_alpha)
    bisecEps = 10**(-4)
    alp_star = BisectionMethod(f_alp, epsilon=bisecEps)
    df_alp = lambdify(alpha, function_alpha.diff(alpha))
    ddf_alp = lambdify(alpha, function_alpha.diff(alpha).diff(alpha))
    alp_star = NewtonsMethod(df_alp, ddf_alp, alp_star, eps, alp_star-bisecEps, alp_star
        +bisecEps)
    return alp_star
```

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

```
def steepestDescentMethod(f, grad_f, x_0, descentEpsilon, exactLineEpsilon=10**(-10)):
    xk = np.array(x_0).reshape(-1,1)
    k = 0
    stop = False
    output = OutputTable()
    while(stop == False):
        d = - np.transpose(grad_f(xk))
        if(np.linalg.norm(d) < descentEpsilon):
            stop = True
        else:
            a = ExactLineSearch(f,xk,d, exactLineEpsilon)
            xkp = xk + a*d
            output.add_row(k, xk, f(xk), d, a, xkp)
```

```
            k += 1
            xk = xkp
            if(k>100):
                break
    output.add_row(k,xk,f(xk),d,None,np.array([]))
    print("Total iteration : {0}".format(k))
    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.

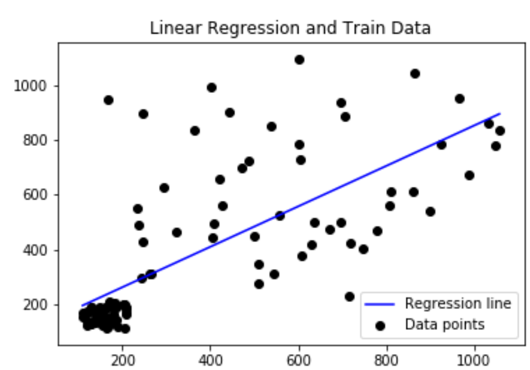

Figure 1: The linear regression and train data

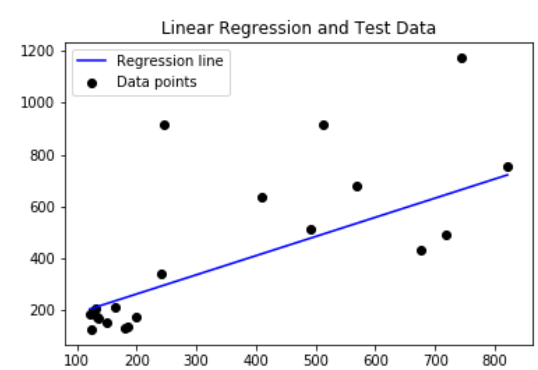

Figure 2: The linear regression and test data

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 **y** and **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 100^{th} 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:

```
X = np.array([np.ones(len(x_train)), x_train, x_train**2]).reshape(3, len(x_train)).T
Y = y_train.reshape(len(y_train),1)
W = np.linalg.inv(X.T @ X) @ X.T @ Y
SSE = np.sum((Y - X @ W)**2)
x_test = np.array(test_data['x'])
y_test = np.array(test_data['y'])
X_test = np.array([np.ones(len(x_test)), x_test, x_test**2]).reshape(3, len(x_test)).T
Y_test = y_test.reshape(len(y_test),1)
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, **y**, given the test input, **x**.

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

$$TrainSSE : 3506233.112$$

$$TestMSE : 43743.010$$

$$Test \quad s^2 : 8606111510.643$$

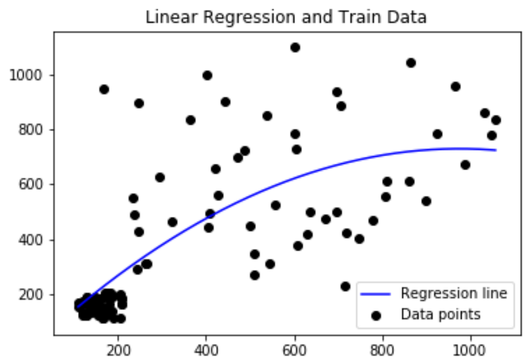

Figure 3: The polynomial regression and train data

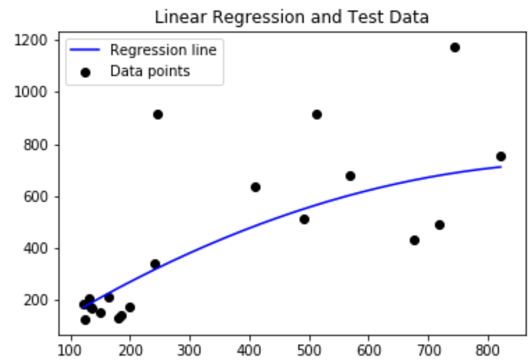

Figure 4: The polynomial regression and test data

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.

```
sigmoidalFunc = lambda output_array : 1 / (1 + np.exp(-output_array))
sigmoidalDeriv = lambda hiddenlayer : hiddenlayer * (1 - hiddenlayer)

def backpropagation(trainingData, hiddenLayerSize, alpha = 0.5, momentum = 0.9, epsilon
     = 0.001, seed = 440):
   np.random.seed(seed)
   t = 0
   patterns = np.copy(trainingData)
   patterns = np.insert(patterns, 0, -1, axis=1) # x0 = -1 unit is added
   P = np.size(patterns, 0) # pattern size
   I = 1 # output unit size
   K = np.size(patterns, 1) - I # input layer size
   J = hiddenLayerSize + 1 # h0 = -1 is added
   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)
   W_matrix = np.random.rand(I, J) # weight between hidden and output layer
   while(alpha >= epsilon):
       np.random.shuffle(patterns)
       x = np.transpose(patterns[:,:-1]).reshape(K, -1)
       y = patterns[:,-1]
       H = np.zeros(J)
       H[0] = -1 # h0 is equal to -1
       O = np.zeros_like(y)
       for p in range(P):
           for j in range(1,J):
               hj = np.sum(w_matrix[j] * x[:,p])
               H[j] = sigmoidalFunc(hj)
```

```python
            for i in range(I):
                o = np.sum(W_matrix[i] * H)
                O[p] = o # linear function g(x) = x
            S_O = 0 # since there is only one output unit
            S_H = np.zeros_like(H)
            for i in range(I):
                S_O = 1 * (y[p] - O[p])
            for j in range(1,J):
                S_H[j] = sigmoidalDeriv(H[j]) * np.sum(W_matrix[0,j] * S_O)
            for j in range(J):
                dWj = alpha * S_O * H[j]
                W_matrix[0,j] += dWj
            for k in range(K):
                dwk = alpha * S_H * x[k,p]
                w_matrix[:,k] += dwk
        alpha *= momentum
        t += 1
        actualHiddens = sigmoidalFunc(w_matrix @ x)
        actualHiddens[0,:] = -1 # h1, ..., hj
        actualOutputMatrix = W_matrix @ actualHiddens # o1, ..., oi
        error = np.sum(np.square(y - actualOutputMatrix))
        #print("Iteration {0} : error = {1}".format(t,error))
    w_matrix = w_matrix[1:] # first row is removed since it corresponds to H0
    return w_matrix, W_matrix, error

def averageError(w_matrix, W_matrix, test_data):
    inputLayers = np.transpose(np.insert(test_data, 0, -1, axis=1)[:,:-1]) # h1, ..., hj
    desiredOutputs = test_data[:,-1].reshape(-1,1)
    actualHiddens = sigmoidalFunc(w_matrix @ inputLayers)
    actualOutputMatrix = W_matrix @ np.insert(actualHiddens, 0, -1, axis=0) # o1, ..., oi
    squareResiduals = np.square(desiredOutputs - np.transpose(actualOutputMatrix))
    sse = np.sum(squareResiduals)
    mse = sse / np.size(desiredOutputs)
    variance = np.sum(np.square(mse-squareResiduals)) / (np.size(desiredOutputs) - 1)
    return mse, variance

def hiddenUnit(train_data, test_data, Jq = 3, epsilon = 0.001, seed = 440):
    train = np.array(train_data)
    test = np.array(test_data)
    q = 1
    Et = np.infty
    while(True):
        patterns = np.copy(train)
        w, W, total_error = backpropagation(patterns, Jq, epsilon=epsilon, seed = seed)
        Etp, var = averageError(w, W, test)
        print("{0} hidden units : MSE = {1} , variance = {2}".format(Jq,Etp,var))
        if(Etp >= Et):
            break
        Jq += 1
        q += 1
        Et = Etp
    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.

```
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.

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

```python
#!/usr/bin/env python
# coding: utf-8

# # Homework 6

# In[1]:



import pandas as pd
import numpy as np
from sympy import Symbol, lambdify
import matplotlib.pyplot as plt


# In[2]:


train_data = pd.read_csv('Input/training.dat', sep=' ', header=None, names=['x', 'y']);
test_data = pd.read_csv('Input/test.dat', sep=' ', header=None, names=['x', 'y']);

x_train = np.array(train_data['x'])
y_train = np.array(train_data['y'])

x_test = np.array(test_data['x'])
y_test = np.array(test_data['y'])

norm_train = np.array(train_data)
norm_train = (norm_train - norm_train.mean(0)) / norm_train.std(0)

# comment out below lines to use normalized data

#x_train = norm_train[:,0]
#y_train = norm_train[:,1]

w0 = Symbol("w0")
w1 = Symbol("w1")
w2 = Symbol("w2")

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)])
func_a_variables = np.array([1, w0**2, w1**2, -w0, -w1, w0*w1])

```

```
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)])
func_b_variables = np.array([1, w0**2, w1**2, w2**2, -w0, -w1, -w2, w0*w1, w0*w2, w1*w2
    ])

func_a = np.sum(func_a_coefficients*func_a_variables)
f_a = lambdify([[w0, w1]], func_a, "numpy")
gf_a = lambdify([[w0, w1]], func_a.diff([[w0, w1]]), "numpy")
grad_fa = lambda x_arr : np.array(gf_a(x_arr), 'float64').reshape(1,len(x_arr))

func_b = np.sum(func_b_coefficient*func_b_variables)
f_b = lambdify([[w0, w1, w2]], func_b, "numpy")
gf_b = lambdify([[w0, w1, w2]], func_b.diff([[w0, w1, w2]]), "numpy")
grad_fb = lambda x_arr : np.array(gf_b(x_arr), 'float64').reshape(1,len(x_arr))


# ### Useful Functions

# In[3]:


regA = lambda w_s, x_arr : x_arr * w_s[1,0] + w_s[0,0]
regB = lambda w_s, x_arr : x_arr**2 * w_s[2,0] + x_arr * w_s[1,0] + w_s[0,0]


# In[4]:


def plotRegressionGraph(data, regFunc, w_star, title, name="graph"):
    xmin = data[:,0].min()
    xmax = data[:,0].max()
    t1 = np.arange(xmin-1, xmax+1, 0.1)
    plt.figure()
    plt.plot(t1, regFunc(w_star, t1), 'b-', label='Regression line')
    plt.scatter(data[:,0], data[:,1], color="black", label="Data points")
    plt.title(title)
    plt.legend()
    plt.savefig("{0}.png".format(name))


# In[5]:


np_str = lambda x_k : np.array2string(x_k.reshape(len(x_k)), precision=3, separator=','
    )

f_str = lambda x : "{0:.4f}".format(x)


# In[6]:


class OutputTable:
```

```python
    def __init__(self):
        self.table = pd.DataFrame([],columns=['k', 'x^k', 'f(x^k)', 'd^k', 'a^k', 'x^k+1
            '])
    def add_row(self, k, xk, fxk, dk, ak, xkp):
        self.table.loc[len(self.table)] = [k, np_str(xk), f_str(fxk.item()), np_str(dk),
             ak, np_str(xkp)]
    def print_latex(self):
        print(self.table.to_latex(index=False))


# ## Part A : Least Square Method with Steepest Descent

# ### Exact Line Search

# In[7]:


def BisectionMethod(f,epsilon, a=-100,b=100) :
    iteration=0
    while (b - a) >= epsilon:
        x_1 = (a + b) / 2
        fx_1 = f(x_1)
        if f(x_1 + epsilon) <= fx_1:
            a = x_1
        else:
            b = x_1
        iteration+=1
    x_star = (a+b)/2
    return x_star

def NewtonsMethod(df, ddf, x_0, epsilon, a, b):
    iteration = 0
    while True:
        dfx0 = df(x_0)
        ddfx0 = ddf(x_0)
        x_1 = x_0-dfx0/ddfx0
        iteration +=1
        if abs(x_0-x_1)<epsilon:
            break
        if x_1<a or x_1>b:
            break
        x_0 = x_1
    x_star = x_0
    return x_star

def ExactLineSearch(f, x0, d, eps=10**(-10)):
    alpha = Symbol('alpha')
    function_alpha = f(np.array(x0)+alpha*np.array(d)).item()
    f_alp = lambdify(alpha, function_alpha)
    bisecEps = 10**(-4)
    alp_star = BisectionMethod(f_alp, epsilon=bisecEps)
    df_alp = lambdify(alpha, function_alpha.diff(alpha))
    ddf_alp = lambdify(alpha, function_alpha.diff(alpha).diff(alpha))
```

```python
    alp_star = NewtonsMethod(df_alp, ddf_alp, alp_star, eps, alp_star-bisecEps, alp_star
        +bisecEps)
    return alp_star


# ### Steepest Descent Method

# In[8]:


def steepestDescentMethod(f, grad_f, x_0, descentEpsilon, exactLineEpsilon=10**(-10)):
    xk = np.array(x_0).reshape(-1,1)
    k = 0
    stop = False
    output = OutputTable()
    while(stop == False):
        d = - np.transpose(grad_f(xk))
        if(np.linalg.norm(d) < descentEpsilon):
            stop = True
        else:
            a = ExactLineSearch(f,xk,d, exactLineEpsilon)
            xkp = xk + a*d
            output.add_row(k, xk, f(xk), d, a, xkp)
            k += 1
            xk = xkp
            if(k>100):
                break
    output.add_row(k,xk,f(xk),d,None,np.array([]))
    print("Total iteration : {0}".format(k))
    return xk, f(xk).item(), output


# In[9]:


ws_a, fs_a, outputs_a = steepestDescentMethod(f_a, grad_fa, [0,0], 0.005)

SSE_train_a = fs_a
MSE_test_a = np.sum((y_test-regA(ws_a,x_test))**2)/np.size(y_test)
var_test_a = np.sum((MSE_test_a-(y_test-regA(ws_a,x_test))**2)**2)/(np.size(y_test)-1)
ws_a, SSE_train_a, MSE_test_a, var_test_a


# In[10]:


plotRegressionGraph(np.array(train_data), regA, ws_a, "Linear Regression and Train Data
    ", "part1a_train")


# In[11]:


```

```
plotRegressionGraph(np.array(test_data), regA, ws_a, "Linear Regression and Test Data", "part1a_test")


# In[12]:


ws_b, fs_b, outputs_b = steepestDescentMethod(f_b, grad_fb, [0,0,0], 0.005)

SSE_train_b = fs_b
MSE_test_b = np.sum((y_test-regB(ws_b,x_test))**2)/np.size(y_test)
var_test_b = np.sum((MSE_test_b-(y_test-regB(ws_b,x_test))**2)**2)/(np.size(y_test)-1)
ws_b, SSE_train_b, MSE_test_b, var_test_b


# In[13]:


plotRegressionGraph(np.array(train_data), regB, ws_b, "Linear Regression and Train Data", "part1b_train")


# In[14]:


plotRegressionGraph(np.array(test_data), regB, ws_b, "Linear Regression and Test Data", "part1b_test")


# ## Part B : Neural Network

# In[15]:


sigmoidalFunc = lambda output_array : 1 / (1 + np.exp(-output_array))
sigmoidalDeriv = lambda hiddenlayer : hiddenlayer * (1 - hiddenlayer)


# In[16]:


def backpropagation(trainingData, hiddenLayerSize, alpha = 0.5, momentum = 0.9, epsilon = 0.001, seed = 440):
    np.random.seed(seed)
    t = 0
    patterns = np.copy(trainingData)
    patterns = np.insert(patterns, 0, -1, axis=1) # x0 = -1 unit is added
    P = np.size(patterns, 0) # pattern size
    I = 1 # output unit size
    K = np.size(patterns, 1) - I # input layer size
    J = hiddenLayerSize + 1 # h0 = -1 is added
    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)
    W_matrix = np.random.rand(I, J) # weight between hidden and output layer
```

```
    while(alpha >= epsilon):
        np.random.shuffle(patterns)
        x = np.transpose(patterns[:,:-1]).reshape(K, -1)
        y = patterns[:,-1]
        H = np.zeros(J)
        H[0] = -1 # h0 is equal to -1
        O = np.zeros_like(y)
        for p in range(P):
            for j in range(1,J):
                hj = np.sum(w_matrix[j] * x[:,p])
                H[j] = sigmoidalFunc(hj)
            for i in range(I):
                o = np.sum(W_matrix[i] * H)
                O[p] = o # linear function g(x) = x
            S_O = 0 # since there is only one output unit
            S_H = np.zeros_like(H)
            for i in range(I):
                S_O = 1 * (y[p] - O[p])
            for j in range(1,J):
                S_H[j] = sigmoidalDeriv(H[j]) * np.sum(W_matrix[0,j] * S_O)
            for j in range(J):
                dWj = alpha * S_O * H[j]
                W_matrix[0,j] += dWj
            for k in range(K):
                dwk = alpha * S_H * x[k,p]
                w_matrix[:,k] += dwk
        alpha *= momentum
        t += 1
        actualHiddens = sigmoidalFunc(w_matrix @ x)
        actualHiddens[0,:] = -1 # h1, ..., hj
        actualOutputMatrix = W_matrix @ actualHiddens # o1, ..., oi
        error = np.sum(np.square(y - actualOutputMatrix))
        #print("Iteration {0} : error = {1}".format(t,error))
    w_matrix = w_matrix[1:] # first row is removed since it corresponds to H0
    return w_matrix, W_matrix, error


# In[17]:


def backpropagationWithMatrix(patterns, hiddenLayerSize, alpha = 0.5, momentum = 0.9, epsilon = 0.001, seed = 440):
    np.random.seed(seed)
    t = 0
    P = np.size(patterns, 0)
    w_matrix = np.random.rand(hiddenLayerSize, np.size(patterns,1))*1 # patterns data includes y values, its column size is selected since we will add x0 to input layer
    W_matrix = np.random.rand(1, hiddenLayerSize+1)*1 # we will add h0 to hidden layer
    while(alpha > epsilon):
        np.random.shuffle(patterns)
        desiredOutputs = patterns[:,-1].reshape(-1,1)
        inputLayers = np.transpose(np.insert(patterns, 0, -1, axis=1)[:,:-1]) # x0 is added to all patterns and its value is -1, output values are excluded
```

```python
        hiddenLayer = np.zeros((hiddenLayerSize+1, 1)) # hiddenlayersize doesn’t include h0 so it’s added
        hiddenLayer[0,:] = -1 # h0 is equal to -1
        actualOutput = np.zeros_like(desiredOutputs)
        for p in range(P):
            hiddenLayer[1:] = sigmoidalFunc(w_matrix @ inputLayers[:,p].reshape(-1,1))
            actualOutput[p] = W_matrix @ hiddenLayer
            # since the function is linear, net output is equal to actual output
            S_output = (1 * (desiredOutputs[p] - actualOutput[p])).reshape(-1,1)
            S_hidden = (sigmoidalDeriv(hiddenLayer[1:]) * (np.transpose(W_matrix[:,1:]) @ S_output)).reshape(-1,1)
            delta_W = alpha * S_output @ np.transpose(hiddenLayer)
            W_matrix += delta_W
            delta_w = alpha * S_hidden @ np.transpose(inputLayers[:,p].reshape(-1,1))
            w_matrix += delta_w
        alpha = momentum * alpha
        t += 1
        actualHiddens = sigmoidalFunc(w_matrix @ inputLayers) # h1, ..., hj
        actualOutputMatrix = W_matrix @ np.insert(actualHiddens, 0, -1, axis=0) # o1, ..., oi
        error = np.sum(np.square(desiredOutputs - np.transpose(actualOutputMatrix)))
        #print("Iteration {0} : error = {1}".format(t,error))
    return w_matrix, W_matrix, error


# In[18]:


patterns = np.array(train_data)
backpropagation(patterns, 3, seed=440)


# In[19]:


patterns = np.array(train_data)
backpropagationWithMatrix(patterns, 3, seed=50)


# In[20]:


patterns2 = np.insert(np.array(train_data), 1, np.square(train_data['x']), axis=1)
backpropagation(patterns2, 3)


# In[21]:


def averageError(w_matrix, W_matrix, test_data):
    inputLayers = np.transpose(np.insert(test_data, 0, -1, axis=1)[:,:-1]) # h1, ..., hj
    desiredOutputs = test_data[:,-1].reshape(-1,1)
    actualHiddens = sigmoidalFunc(w_matrix @ inputLayers)
```

```
    actualOutputMatrix = W_matrix @ np.insert(actualHiddens, 0, -1, axis=0) # o1, ...,
        oi
    squareResiduals = np.square(desiredOutputs - np.transpose(actualOutputMatrix))
    sse = np.sum(squareResiduals)
    mse = sse / np.size(desiredOutputs)
    variance = np.sum(np.square(mse-squareResiduals)) / (np.size(desiredOutputs) - 1)
    return mse, variance


# In[22]:


def hiddenUnit(train_data, test_data, Jq = 3, epsilon = 0.001, seed = 440):
    train = np.array(train_data)
    test = np.array(test_data)
    q = 1
    Et = np.infty
    while(True):
        patterns = np.copy(train)
        w, W, total_error = backpropagation(patterns, Jq, epsilon=epsilon, seed = seed)
        Etp, var = averageError(w, W, test)
        print("{0} hidden units : MSE = {1} , variance = {2}".format(Jq,Etp,var))
        if(Etp >= Et):
            break
        Jq += 1
        q += 1
        Et = Etp
    return Jq-1, Et


# In[23]:


hiddenUnit(train_data, test_data, epsilon=0.001, seed = 440)


# In[24]:


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

# In[ ]:
```