To start with, Logistic Regression is performed using sklearn’s logistic regression package for the cancer data set also from sklearn. This is shown below

**1. Logistic Regression**

import numpy as np

import pandas as pd

import os

import matplotlib.pyplot as plt

from sklearn.model\_selection import train\_test\_split

from sklearn.linear\_model import LogisticRegression

from sklearn.datasets import make\_classification, make\_blobs

from sklearn.metrics import confusion\_matrix

from matplotlib.colors import ListedColormap

from sklearn.datasets import load\_breast\_cancer

# Load the cancer data

(X\_cancer, y\_cancer) = load\_breast\_cancer(return\_X\_y = True)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_cancer, y\_cancer,

random\_state = 0)

# Call the Logisitic Regression function

clf = LogisticRegression().fit(X\_train, y\_train)

print('Accuracy of Logistic regression classifier on training set: {:.2f}'

.format(clf.score(X\_train, y\_train)))

print('Accuracy of Logistic regression classifier on test set: {:.2f}'

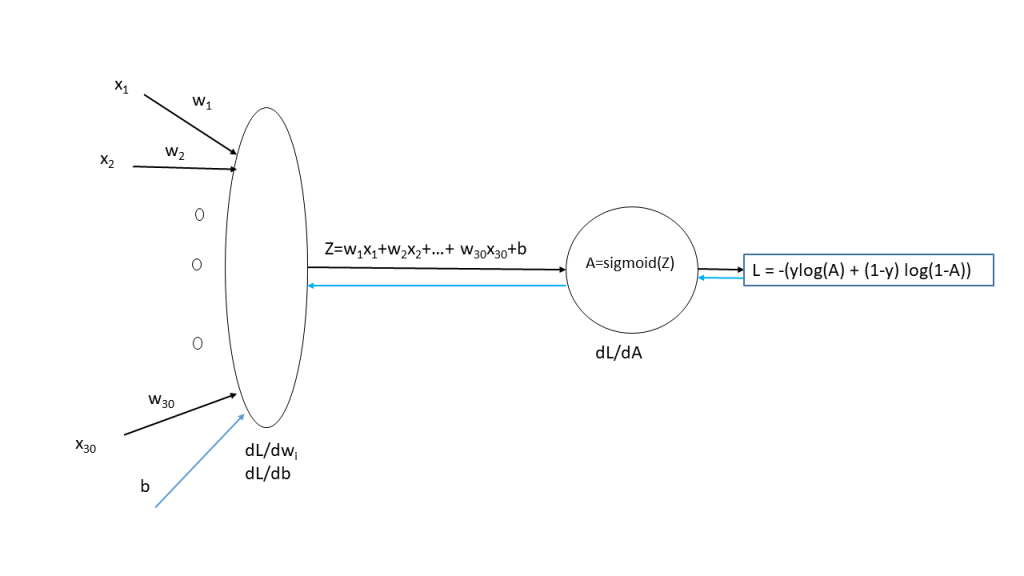
.format(clf.score(X\_test, y\_test)))

## Accuracy of Logistic regression classifier on training set: 0.96

## Accuracy of Logistic regression classifier on test set: 0.96

**2. Logistic Regression as a 2 layer Neural Network**

In the following section Logistic Regression is implemented as a 2 layer Neural Network in Python, R and Octave. The same cancer data set from sklearn will be used to train and test the Neural Network in Python, R and Octave. This can be represented diagrammatically as below



The cancer data set has 30 input features, and the target variable ‘output’ is either 0 or 1. Hence the sigmoid activation function will be used in the output layer for classification.

This simple 2 layer Neural Network is shown below  
At the input layer there are 30 features and the corresponding weights of these inputs which are initialized to small random values.  
Z= w_{1}x_{1} +w_{2}x_{2} +..+ w_{30}x_{30} + b  
where ‘b’ is the bias term

The Activation function is the sigmoid function which is a= 1/(1+e^{-z})  
The Loss, when the sigmoid function is used in the output layer, is given by  
L=-(ylog(a) + (1-y)log(1-a))(1)

**Gradient Descent**

**Forward propagation**

In forward propagation cycle of the Neural Network the output Z and the output of activation function, the sigmoid function, is first computed. Then using the output ‘y’ for the given features, the ‘Loss’ is computed using equation (1) above.

**Backward propagation**

The backward propagation cycle determines how the ‘Loss’ is impacted for small variations from the previous layers upto the input layer. In other words, backward propagation computes the changes in the weights at the input layer, which will minimize the loss. Several cycles of gradient descent are performed in the path of steepest descent to find the local minima. In other words the set of weights and biases, at the input layer, which will result in the lowest loss is computed by gradient descent. The weights at the input layer are decreased by a parameter known as the ‘learning rate’. Too big a ‘learning rate’ can overshoot the local minima, and too small a ‘learning rate’ can take a long time to reach the local minima. This is done for ‘m’ training examples.

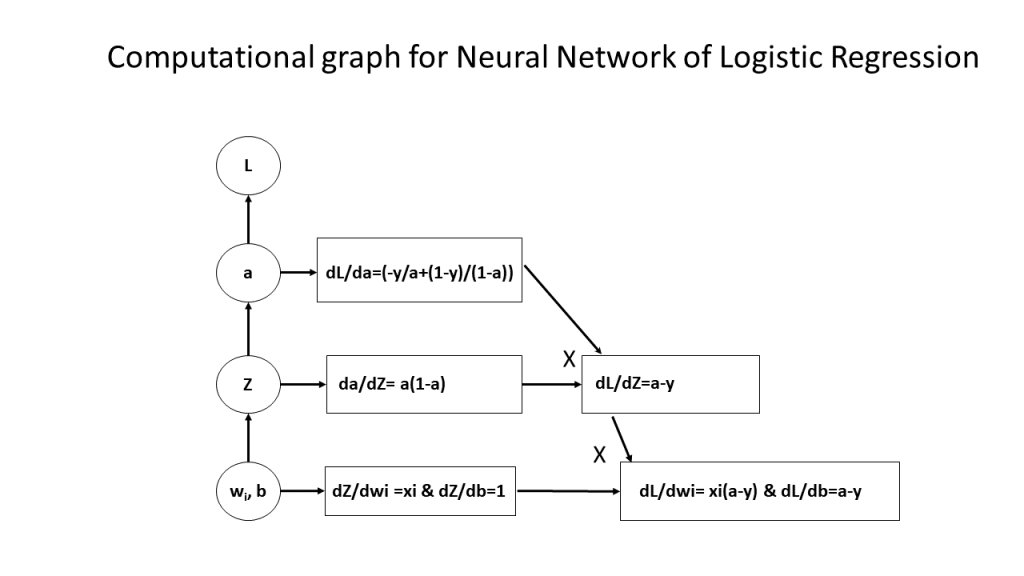
**Chain rule of differentiation**  
Let y=f(u)  
and u=g(x) then  
\partial y/\partial x = \partial y/\partial u * \partial u/\partial x

**Derivative of sigmoid**  
\sigma=1/(1+e^{-z})  
Let x= 1 + e^{-z} then  
\sigma = 1/x  
\partial \sigma/\partial x = -1/x^{2}  
\partial x/\partial z = -e^{-z}  
Using the chain rule of differentiation we get  
\partial \sigma/\partial z = \partial \sigma/\partial x * \partial x/\partial z  
=-1/(1+e^{-z})^{2}* -e^{-z} = e^{-z}/(1+e^{-z})^{2}  
Therefore \partial \sigma/\partial z = \sigma(1-\sigma)        -(2)

The 3 equations for the 2 layer Neural Network representation of Logistic Regression are  
L=-(y*log(a) + (1-y)*log(1-a))      -(a)  
a=1/(1+e^{-Z})      -(b)  
Z= w_{1}x_{1} +w_{2}x_{2} +...+ w_{30}x_{30} +b = Z = \sum_{i} w_{i}*x_{i} + b-(c)

The back propagation step requires the computation of dL/dw_{i}and dL/db_{i}. In the case of regression it would be dE/dw_{i}and dE/db_{i}where dE is the Mean Squared Error function.  
Computing the derivatives for back propagation we have  
dL/da = -(y/a + (1-y)/(1-a))          -(d)  
because d/dx(logx) = 1/x  
Also from equation (2) we get  
da/dZ = a (1-a)                                  – (e)  
By chain rule  
\partial L/\partial Z = \partial L/\partial a * \partial a/\partial Z  
therefore substituting the results of (d) & (e) we get  
\partial L/\partial Z = -(y/a + (1-y)/(1-a)) * a(1-a) = a-y         (f)  
Finally  
\partial L/\partial w_{i}= \partial L/\partial a * \partial a/\partial Z * \partial Z/\partial w_{i}                                                           -(g)  
\partial Z/\partial w_{i} = x_{i}            – (h)  
and from (f) we have  \partial L/\partial Z =a-y  
Therefore  (g) reduces to  
\partial L/\partial w_{i} = x_{i}* (a-y)-(i)  
Also  
\partial L/\partial b = \partial L/\partial a * \partial a/\partial Z * \partial Z/\partial b-(j)  
Since  
\partial Z/\partial b = 1and using (f) in (j)  
\partial L/\partial b = a-y

The gradient computes the weights at the input layer and the corresponding bias by using the values  
of dw_{i}and db  
w_{i} := w_{i} -\alpha * dw_{i}  
b := b -\alpha * db  
I found the computation graph representation in the book Deep Learning: Ian Goodfellow, Yoshua Bengio, Aaron Courville, very useful to visualize and also compute the backward propagation. For the 2 layer Neural Network of Logistic Regression the computation graph is shown below



**3. Neural Network for Logistic Regression -Python code (vectorized)**

import numpy as np

import pandas as pd

import os

import matplotlib.pyplot as plt

from sklearn.model\_selection import train\_test\_split

# Define the sigmoid function

def sigmoid(z):

a=1/(1+np.exp(-z))

return a

# Initialize

def initialize(dim):

w = np.zeros(dim).reshape(dim,1)

b = 0

return w

# Compute the loss

def computeLoss(numTraining,Y,A):

loss=-1/numTraining \*np.sum(Y\*np.log(A) + (1-Y)\*(np.log(1-A)))

return(loss)

# Execute the forward propagation

def forwardPropagation(w,b,X,Y):

# Compute Z

Z=np.dot(w.T,X)+b

# Determine the number of training samples

numTraining=float(len(X))

# Compute the output of the sigmoid activation function

A=sigmoid(Z)

#Compute the loss

loss = computeLoss(numTraining,Y,A)

# Compute the gradients dZ, dw and db

dZ=A-Y

dw=1/numTraining\*np.dot(X,dZ.T)

db=1/numTraining\*np.sum(dZ)

# Return the results as a dictionary

gradients = {"dw": dw,

"db": db}

loss = np.squeeze(loss)

return gradients,loss

# Compute Gradient Descent

def gradientDescent(w, b, X, Y, numIerations, learningRate):

losses=[]

idx =[]

# Iterate

for i in range(numIerations):

gradients,loss=forwardPropagation(w,b,X,Y)

#Get the derivates

dw = gradients["dw"]

db = gradients["db"]

w = w-learningRate\*dw

b = b-learningRate\*db

# Store the loss

if i % 100 == 0:

idx.append(i)

losses.append(loss)

# Set params and grads

params = {"w": w,

"b": b}

grads = {"dw": dw,

"db": db}

return params, grads, losses,idx

# Predict the output for a training set

def predict(w,b,X):

size=X.shape[1]

yPredicted=np.zeros((1,size))

Z=np.dot(w.T,X)

# Compute the sigmoid

A=sigmoid(Z)

for i in range(A.shape[1]):

#If the value is > 0.5 then set as 1

if(A[0][i] > 0.5):

yPredicted[0][i]=1

else:

# Else set as 0

yPredicted[0][i]=0

return yPredicted

#Normalize the data

def normalize(x):

x\_norm = None

x\_norm = np.linalg.norm(x,axis=1,keepdims=True)

x= x/x\_norm

return x

# Run the 2 layer Neural Network on the cancer data set

from sklearn.datasets import load\_breast\_cancer

# Load the cancer data

(X\_cancer, y\_cancer) = load\_breast\_cancer(return\_X\_y = True)

# Create train and test sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_cancer, y\_cancer,

random\_state = 0)

# Normalize the data for better performance

X\_train1=normalize(X\_train)

# Create weight vectors of zeros. The size is the number of features in the data set=30

w=np.zeros((X\_train.shape[1],1))

#w=np.zeros((30,1))

b=0

#Normalize the training data so that gradient descent performs better

X\_train1=normalize(X\_train)

#Transpose X\_train so that we have a matrix as (features, numSamples)

X\_train2=X\_train1.T

# Reshape to remove the rank 1 array and then transpose

y\_train1=y\_train.reshape(len(y\_train),1)

y\_train2=y\_train1.T

# Run gradient descent for 4000 times and compute the weights

parameters, grads, costs,idx = gradientDescent(w, b, X\_train2, y\_train2, numIerations=4000, learningRate=0.75)

w = parameters["w"]

b = parameters["b"]

# Normalize X\_test

X\_test1=normalize(X\_test)

#Transpose X\_train so that we have a matrix as (features, numSamples)

X\_test2=X\_test1.T

#Reshape y\_test

y\_test1=y\_test.reshape(len(y\_test),1)

y\_test2=y\_test1.T

# Predict the values for

yPredictionTest = predict(w, b, X\_test2)

yPredictionTrain = predict(w, b, X\_train2)

# Print the accuracy

print("train accuracy: {} %".format(100 - np.mean(np.abs(yPredictionTrain - y\_train2)) \* 100))

print("test accuracy: {} %".format(100 - np.mean(np.abs(yPredictionTest - y\_test)) \* 100))

# Plot the Costs vs the number of iterations

fig1=plt.plot(idx,costs)

fig1=plt.title("Gradient descent-Cost vs No of iterations")

fig1=plt.xlabel("No of iterations")

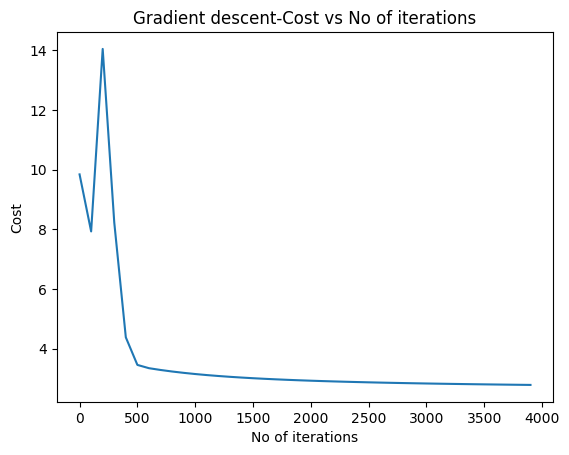
fig1=plt.ylabel("Cost")

fig1.figure.savefig("fig1", bbox\_inches='tight')

## train accuracy: 90.3755868545 %

## test accuracy: 89.5104895105 %

**Note**: It can be seen that the Accuracy on the training and test set is 90.37% and 89.51%. This is comparatively poorer than the 96% which the logistic regression of sklearn achieves! But this is mainly because of the absence of hidden layers which is the real power of neural networks.

****

**4. Neural Network for Logistic Regression -R code (vectorized)**

source("RFunctions-1.R")

# Define the sigmoid function

sigmoid <- function(z){

a <- 1/(1+ exp(-z))

a

}

# Compute the loss

computeLoss <- function(numTraining,Y,A){

loss <- -1/numTraining\* sum(Y\*log(A) + (1-Y)\*log(1-A))

return(loss)

}

# Compute forward propagation

forwardPropagation <- function(w,b,X,Y){

# Compute Z

Z <- t(w) %\*% X +b

#Set the number of samples

numTraining <- ncol(X)

# Compute the activation function

A=sigmoid(Z)

#Compute the loss

loss <- computeLoss(numTraining,Y,A)

# Compute the gradients dZ, dw and db

dZ<-A-Y

dw<-1/numTraining \* X %\*% t(dZ)

db<-1/numTraining\*sum(dZ)

fwdProp <- list("loss" = loss, "dw" = dw, "db" = db)

return(fwdProp)

}

# Perform one cycle of Gradient descent

gradientDescent <- function(w, b, X, Y, numIerations, learningRate){

losses <- NULL

idx <- NULL

# Loop through the number of iterations

for(i in 1:numIerations){

fwdProp <-forwardPropagation(w,b,X,Y)

#Get the derivatives

dw <- fwdProp$dw

db <- fwdProp$db

#Perform gradient descent

w = w-learningRate\*dw

b = b-learningRate\*db

l <- fwdProp$loss

# Stoe the loss

if(i %% 100 == 0){

idx <- c(idx,i)

losses <- c(losses,l)

}

}

# Return the weights and losses

gradDescnt <- list("w"=w,"b"=b,"dw"=dw,"db"=db,"losses"=losses,"idx"=idx)

return(gradDescnt)

}

# Compute the predicted value for input

predict <- function(w,b,X){

m=dim(X)[2]

# Create a ector of 0's

yPredicted=matrix(rep(0,m),nrow=1,ncol=m)

Z <- t(w) %\*% X +b

# Compute sigmoid

A=sigmoid(Z)

for(i in 1:dim(A)[2]){

# If A > 0.5 set value as 1

if(A[1,i] > 0.5)

yPredicted[1,i]=1

else

# Else set as 0

yPredicted[1,i]=0

}

return(yPredicted)

}

# Normalize the matrix

normalize <- function(x){

#Create the norm of the matrix.Perform the Frobenius norm of the matrix

n<-as.matrix(sqrt(rowSums(x^2)))

#Sweep by rows by norm. Note '1' in the function which performing on every row

normalized<-sweep(x, 1, n, FUN="/")

return(normalized)

}

# Run the 2 layer Neural Network on the cancer data set

# Read the data (from sklearn)

cancer <- read.csv("cancer.csv")

# Rename the target variable

names(cancer) <- c(seq(1,30),"output")

# Split as training and test sets

train\_idx <- trainTestSplit(cancer,trainPercent=75,seed=5)

train <- cancer[train\_idx, ]

test <- cancer[-train\_idx, ]

# Set the features

X\_train <-train[,1:30]

y\_train <- train[,31]

X\_test <- test[,1:30]

y\_test <- test[,31]

# Create a matrix of 0's with the number of features

w <-matrix(rep(0,dim(X\_train)[2]))

b <-0

X\_train1 <- normalize(X\_train)

X\_train2=t(X\_train1)

# Reshape then transpose

y\_train1=as.matrix(y\_train)

y\_train2=t(y\_train1)

# Perform gradient descent

gradDescent= gradientDescent(w, b, X\_train2, y\_train2, numIerations=3000, learningRate=0.77)

# Normalize X\_test

X\_test1=normalize(X\_test)

#Transpose X\_train so that we have a matrix as (features, numSamples)

X\_test2=t(X\_test1)

#Reshape y\_test and take transpose

y\_test1=as.matrix(y\_test)

y\_test2=t(y\_test1)

# Use the values of the weights generated from Gradient Descent

yPredictionTest = predict(gradDescent$w, gradDescent$b, X\_test2)

yPredictionTrain = predict(gradDescent$w, gradDescent$b, X\_train2)

sprintf("Train accuracy: %f",(100 - mean(abs(yPredictionTrain - y\_train2)) \* 100))

## [1] "Train accuracy: 90.845070"

sprintf("test accuracy: %f",(100 - mean(abs(yPredictionTest - y\_test)) \* 100))

## [1] "test accuracy: 87.323944"

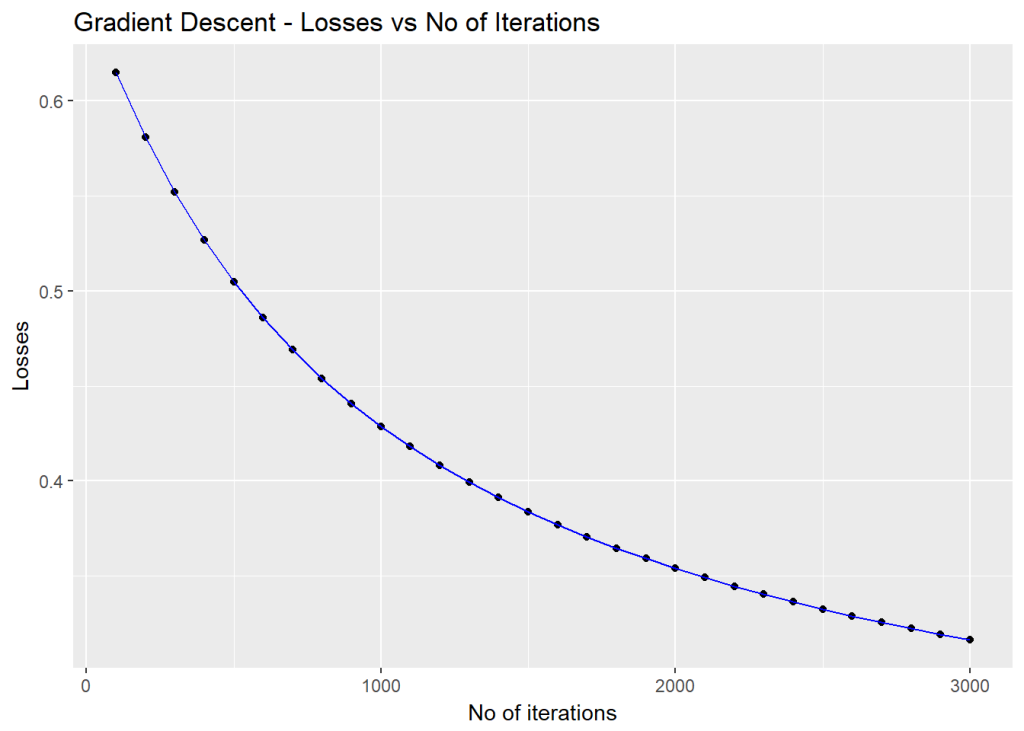
df <-data.frame(gradDescent$idx, gradDescent$losses)

names(df) <- c("iterations","losses")

ggplot(df,aes(x=iterations,y=losses)) + geom\_point() + geom\_line(col="blue") +

ggtitle("Gradient Descent - Losses vs No of Iterations") +

xlab("No of iterations") + ylab("Losses")



**4. Neural Network for Logistic Regression -Octave code (vectorized)**

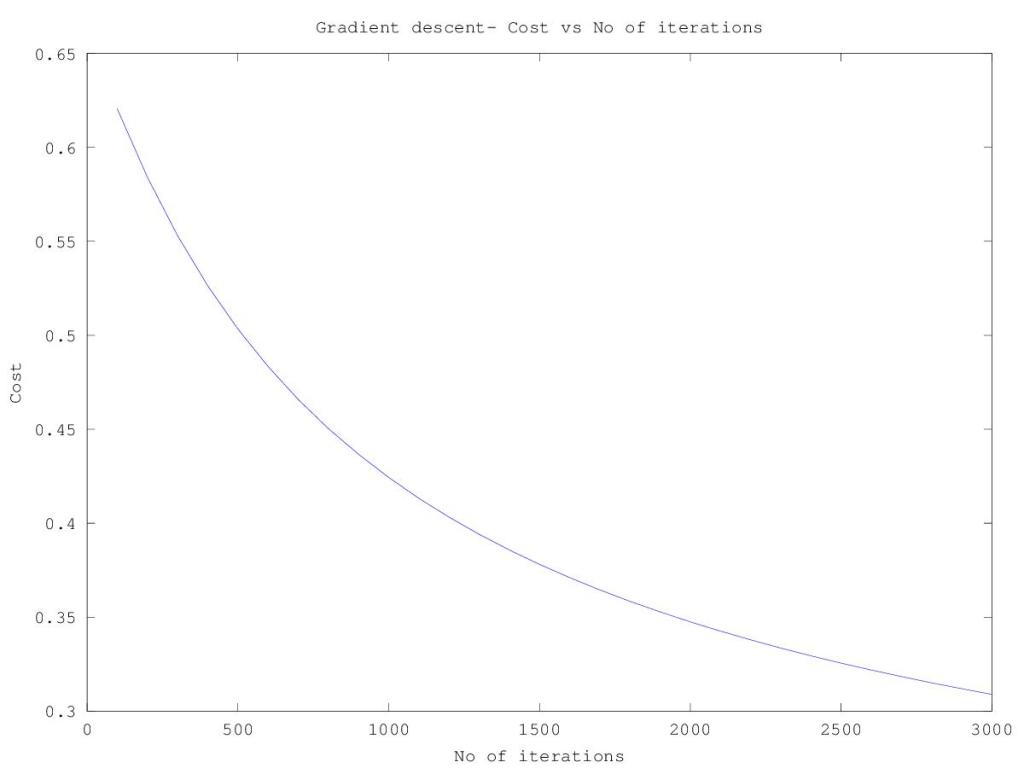
1;  
# Define sigmoid function  
function a = sigmoid(z)  
a = 1 ./ (1+ exp(-z));  
end  
# Compute the loss  
function loss=computeLoss(numtraining,Y,A)  
loss = -1/numtraining \* sum((Y .\* log(A)) + (1-Y) .\* log(1-A));  
end  
  
# Perform forward propagation  
function [loss,dw,db,dZ] = forwardPropagation(w,b,X,Y)  
% Compute Z  
Z = w' \* X + b;  
numtraining = size(X)(1,2);  
# Compute sigmoid  
A = sigmoid(Z);  
  
#Compute loss. Note this is element wise product  
loss =computeLoss(numtraining,Y,A);  
# Compute the gradients dZ, dw and db  
dZ = A-Y;  
dw = 1/numtraining\* X \* dZ';  
db =1/numtraining\*sum(dZ);

end  
  
# Compute Gradient Descent  
function [w,b,dw,db,losses,index]=gradientDescent(w, b, X, Y, numIerations, learningRate)  
#Initialize losses and idx  
losses=[];  
index=[];  
# Loop through the number of iterations  
for i=1:numIerations,  
[loss,dw,db,dZ] = forwardPropagation(w,b,X,Y);  
# Perform Gradient descent  
w = w - learningRate\*dw;  
b = b - learningRate\*db;  
if(mod(i,100) ==0)  
# Append index and loss  
index = [index i];  
losses = [losses loss];  
endif

end  
end  
  
# Determine the predicted value for dataset  
function yPredicted = predict(w,b,X)  
m = size(X)(1,2);  
yPredicted=zeros(1,m);  
# Compute Z  
Z = w' \* X + b;  
# Compute sigmoid  
A = sigmoid(Z);  
for i=1:size(X)(1,2),  
# Set predicted as 1 if A > 0,5  
if(A(1,i) >= 0.5)  
yPredicted(1,i)=1;  
else  
yPredicted(1,i)=0;  
endif  
end  
end  
  
# Normalize by dividing each value by the sum of squares  
function normalized = normalize(x)  
# Compute Frobenius norm. Square the elements, sum rows and then find square root  
a = sqrt(sum(x .^ 2,2));  
# Perform element wise division  
normalized = x ./ a;  
end  
  
# Split into train and test sets  
function [X\_train,y\_train,X\_test,y\_test] = trainTestSplit(dataset,trainPercent)  
# Create a random index  
ix = randperm(length(dataset));  
# Split into training  
trainSize = floor(trainPercent/100 \* length(dataset));  
train=dataset(ix(1:trainSize),:);  
# And test  
test=dataset(ix(trainSize+1:length(dataset)),:);  
X\_train = train(:,1:30);  
y\_train = train(:,31);  
X\_test = test(:,1:30);  
y\_test = test(:,31);  
end

cancer=csvread("cancer.csv");  
[X\_train,y\_train,X\_test,y\_test] = trainTestSplit(cancer,75);  
w=zeros(size(X\_train)(1,2),1);  
b=0;  
X\_train1=normalize(X\_train);  
X\_train2=X\_train1';  
y\_train1=y\_train';  
[w1,b1,dw,db,losses,idx]=gradientDescent(w, b, X\_train2, y\_train1, numIerations=3000, learningRate=0.75);  
# Normalize X\_test  
X\_test1=normalize(X\_test);  
#Transpose X\_train so that we have a matrix as (features, numSamples)  
X\_test2=X\_test1';  
y\_test1=y\_test';  
# Use the values of the weights generated from Gradient Descent  
yPredictionTest = predict(w1, b1, X\_test2);  
yPredictionTrain = predict(w1, b1, X\_train2);

trainAccuracy=100-mean(abs(yPredictionTrain - y\_train1))\*100  
testAccuracy=100- mean(abs(yPredictionTest - y\_test1))\*100  
trainAccuracy = 90.845  
testAccuracy = 89.510  
graphics\_toolkit('gnuplot')  
plot(idx,losses);  
title ('Gradient descent- Cost vs No of iterations');  
xlabel ("No of iterations");  
ylabel ("Cost");



**DLFunctions.m**

|  |
| --- |
| 1; |
|  | # Define sigmoid function |
|  | function a = sigmoid(z) |
|  | a = 1 ./ (1+ exp(-z)); |
|  | end |
|  | # Compute the loss |
|  | function loss=computeLoss(numtraining,Y,A) |
|  | loss = -1/numtraining \* sum((Y .\* log(A)) + (1-Y) .\* log(1-A)); |
|  | end |
|  |  |
|  | # Perform forward propagation |
|  | function [loss,dw,db,dZ] = forwardPropagation(w,b,X,Y) |
|  | % Compute Z |
|  | Z = w' \* X + b; |
|  | numtraining = size(X)(1,2); |
|  | # Compute sigmoid |
|  | A = sigmoid(Z); |
|  |  |
|  | #Compute loss. Note this is element wise product |
|  | loss =computeLoss(numtraining,Y,A); |
|  | # Compute the gradients dZ, dw and db |
|  | dZ = A-Y; |
|  | dw = 1/numtraining\* X \* dZ'; |
|  | db =1/numtraining\*sum(dZ); |
|  |  |
|  | end |
|  |  |
|  | # Compute Gradient Descent |
|  | function [w,b,dw,db,losses,index]=gradientDescent(w, b, X, Y, numIerations, learningRate) |
|  | #Initialize losses and idx |
|  | losses=[]; |
|  | index=[]; |
|  | # Loop through the number of iterations |
|  | for i=1:numIerations, |
|  | [loss,dw,db,dZ] = forwardPropagation(w,b,X,Y); |
|  | # Perform Gradient descent |
|  | w = w - learningRate\*dw; |
|  | b = b - learningRate\*db; |
|  | if(mod(i,100) ==0) |
|  | # Append index and loss |
|  | index = [index i]; |
|  | losses = [losses loss]; |
|  | endif |
|  |  |
|  | end |
|  | end |
|  |  |
|  | # Determine the predicted value for dataset |
|  | function yPredicted = predict(w,b,X) |
|  | m = size(X)(1,2); |
|  | yPredicted=zeros(1,m); |
|  | # Compute Z |
|  | Z = w' \* X + b; |
|  | # Compute sigmoid |
|  | A = sigmoid(Z); |
|  | for i=1:size(X)(1,2), |
|  | # Set predicted as 1 if A > 0,5 |
|  | if(A(1,i) >= 0.5) |
|  | yPredicted(1,i)=1; |
|  | else |
|  | yPredicted(1,i)=0; |
|  | endif |
|  | end |
|  | end |
|  |  |
|  | # Normalize by dividing each value by the sum of squares |
|  | function normalized = normalize(x) |
|  | # Compute Frobenius norm. Square the elements, sum rows and then find square root |
|  | a = sqrt(sum(x .^ 2,2)); |
|  | # Perform element wise division |
|  | normalized = x ./ a; |
|  | end |
|  |  |
|  | # Split into train and test sets |
|  | function [X\_train,y\_train,X\_test,y\_test] = trainTestSplit(dataset,trainPercent) |
|  | # Create a random index |
|  | ix = randperm(length(dataset)); |
|  | # Split into training |
|  | trainSize = floor(trainPercent/100 \* length(dataset)); |
|  | train=dataset(ix(1:trainSize),:); |
|  | # And test |
|  | test=dataset(ix(trainSize+1:length(dataset)),:); |
|  | X\_train = train(:,1:30); |
|  | y\_train = train(:,31); |
|  | X\_test = test(:,1:30); |
|  | y\_test = test(:,31); |
|  | end |

**DLMain.m**

|  |
| --- |
| source("DL-functions.m") |
|  |  |
|  | cancer=csvread("cancer.csv"); |
|  | [X\_train,y\_train,X\_test,y\_test] = trainTestSplit(cancer,75); |
|  |  |
|  | w=zeros(size(X\_train)(1,2),1); |
|  | b=0; |
|  | X\_train1=normalize(X\_train); |
|  | X\_train2=X\_train1'; |
|  | y\_train1=y\_train'; |
|  | [w1,b1,dw,db,losses,idx]=gradientDescent(w, b, X\_train2, y\_train1, numIerations=3000, learningRate=0.75); |
|  |  |
|  | # Normalize X\_test |
|  | X\_test1=normalize(X\_test); |
|  | #Transpose X\_train so that we have a matrix as (features, numSamples) |
|  | X\_test2=X\_test1'; |
|  | y\_test1=y\_test'; |
|  |  |
|  | # Use the values of the weights generated from Gradient Descent |
|  | yPredictionTest = predict(w1, b1, X\_test2); |
|  |  |
|  | yPredictionTrain = predict(w1, b1, X\_train2); |
|  |  |
|  | trainAccuracy=100-mean(abs(yPredictionTrain - y\_train1))\*100 |
|  | testAccuracy=100- mean(abs(yPredictionTest - y\_test1))\*100 |
|  |  |
|  | graphics\_toolkit('gnuplot') |
|  | plot(idx,losses); |
|  | title ('Gradient descent- Cost vs No of iterations'); |
|  | xlabel ("No of iterations"); |
|  | ylabel ("Cost"); |
|  | print ("lr.jpg") |

**DLFunctions1.R**

|  |
| --- |
| source("RFunctions-1.R") |
|  | # Define the sigmoid function |
|  | sigmoid <- function(z){ |
|  | a <- 1/(1+ exp(-z)) |
|  | a |
|  | } |
|  |  |
|  | # Compute the loss |
|  | computeLoss <- function(numTraining,Y,A){ |
|  | loss <- -1/numTraining\* sum(Y\*log(A) + (1-Y)\*log(1-A)) |
|  | return(loss) |
|  | } |
|  |  |
|  | # Compute forward propagation |
|  | forwardPropagation <- function(w,b,X,Y){ |
|  | # Compute Z |
|  | Z <- t(w) %\*% X +b |
|  | #Set the number of samples |
|  | numTraining <- ncol(X) |
|  | # Compute the activation function |
|  | A=sigmoid(Z) |
|  |  |
|  | #Compute the loss |
|  | loss <- computeLoss(numTraining,Y,A) |
|  |  |
|  | # Compute the gradients dZ, dw and db |
|  | dZ<-A-Y |
|  | dw<-1/numTraining \* X %\*% t(dZ) |
|  | db<-1/numTraining\*sum(dZ) |
|  |  |
|  | fwdProp <- list("loss" = loss, "dw" = dw, "db" = db) |
|  | return(fwdProp) |
|  | } |
|  |  |
|  | # Perform one cycle of Gradient descent |
|  | gradientDescent <- function(w, b, X, Y, numIerations, learningRate){ |
|  | losses <- NULL |
|  | idx <- NULL |
|  | # Loop through the number of iterations |
|  | for(i in 1:numIerations){ |
|  | fwdProp <-forwardPropagation(w,b,X,Y) |
|  | #Get the derivatives |
|  | dw <- fwdProp$dw |
|  | db <- fwdProp$db |
|  | #Perform gradient descent |
|  | w = w-learningRate\*dw |
|  | b = b-learningRate\*db |
|  | l <- fwdProp$loss |
|  | # Stoe the loss |
|  | if(i %% 100 == 0){ |
|  | idx <- c(idx,i) |
|  | losses <- c(losses,l) |
|  | } |
|  | } |
|  |  |
|  | # Return the weights and losses |
|  | gradDescnt <- list("w"=w,"b"=b,"dw"=dw,"db"=db,"losses"=losses,"idx"=idx) |
|  |  |
|  |  |
|  | return(gradDescnt) |
|  | } |
|  |  |
|  | # COmpute the predicted value for input |
|  | predict <- function(w,b,X){ |
|  | m=dim(X)[2] |
|  | # Create a ector of 0's |
|  | yPredicted=matrix(rep(0,m),nrow=1,ncol=m) |
|  | Z <- t(w) %\*% X +b |
|  | # Compute sigmoid |
|  | A=sigmoid(Z) |
|  | for(i in 1:dim(A)[2]){ |
|  | # If A > 0.5 set value as 1 |
|  | if(A[1,i] > 0.5) |
|  | yPredicted[1,i]=1 |
|  | else |
|  | # Else set as 0 |
|  | yPredicted[1,i]=0 |
|  | } |
|  |  |
|  | return(yPredicted) |
|  | } |
|  |  |
|  | # Normalize the matrix |
|  | normalize <- function(x){ |
|  | #Create the norm of the matrix.Perform the Frobenius norm of the matrix |
|  | n<-as.matrix(sqrt(rowSums(x^2))) |
|  | #Sweep by rows by norm. Note '1' in the function which performing on every row |
|  | normalized<-sweep(x, 1, n, FUN="/") |
|  | return(normalized) |
|  | } |

**DLFunctions1.py**

|  |
| --- |
|  |
| import numpy as np |
|  | import pandas as pd |
|  | import os |
|  | import matplotlib.pyplot as plt |
|  | from sklearn.model\_selection import train\_test\_split |
|  |  |
|  | # Define the sigmoid function |
|  | def sigmoid(z): |
|  | a=1/(1+np.exp(-z)) |
|  | return a |
|  |  |
|  | # Initialize |
|  | def initialize(dim): |
|  | w = np.zeros(dim).reshape(dim,1) |
|  | b = 0 |
|  | return w |
|  |  |
|  | # Compute the loss |
|  | def computeLoss(numTraining,Y,A): |
|  | loss=-1/numTraining \*np.sum(Y\*np.log(A) + (1-Y)\*(np.log(1-A))) |
|  | return(loss) |
|  |  |
|  | # Execute the forward propagation |
|  | def forwardPropagation(w,b,X,Y): |
|  | # Compute Z |
|  | Z=np.dot(w.T,X)+b |
|  | # Determine the number of training samples |
|  | numTraining=float(len(X)) |
|  | # Compute the output of the sigmoid activation function |
|  | A=sigmoid(Z) |
|  | #Compute the loss |
|  | loss = computeLoss(numTraining,Y,A) |
|  | # Compute the gradients dZ, dw and db |
|  | dZ=A-Y |
|  | dw=1/numTraining\*np.dot(X,dZ.T) |
|  | db=1/numTraining\*np.sum(dZ) |
|  |  |
|  | # Return the results as a dictionary |
|  | gradients = {"dw": dw, |
|  | "db": db} |
|  | loss = np.squeeze(loss) |
|  | return gradients,loss |
|  |  |
|  | # Compute Gradient Descent |
|  | def gradientDescent(w, b, X, Y, numIerations, learningRate): |
|  | losses=[] |
|  | idx =[] |
|  | # Iterate |
|  | for i in range(numIerations): |
|  | gradients,loss=forwardPropagation(w,b,X,Y) |
|  | #Get the derivates |
|  | dw = gradients["dw"] |
|  | db = gradients["db"] |
|  | w = w-learningRate\*dw |
|  | b = b-learningRate\*db |
|  |  |
|  | # Store the loss |
|  | if i % 100 == 0: |
|  | idx.append(i) |
|  | losses.append(loss) |
|  | # Set params and grads |
|  | params = {"w": w, |
|  | "b": b} |
|  | grads = {"dw": dw, |
|  | "db": db} |
|  |  |
|  | return params, grads, losses,idx |
|  |  |
|  | # Predict the output for a training set |
|  | def predict(w,b,X): |
|  | size=X.shape[1] |
|  | yPredicted=np.zeros((1,size)) |
|  | Z=np.dot(w.T,X) |
|  | # Compute the sigmoid |
|  | A=sigmoid(Z) |
|  | for i in range(A.shape[1]): |
|  | #If the value is > 0.5 then set as 1 |
|  | if(A[0][i] > 0.5): |
|  | yPredicted[0][i]=1 |
|  | else: |
|  | # Else set as 0 |
|  | yPredicted[0][i]=0 |
|  |  |
|  | return yPredicted |
|  |  |
|  | #Normalize the data |
|  | def normalize(x): |
|  | x\_norm = None |
|  | x\_norm = np.linalg.norm(x,axis=1,keepdims=True) |
|  | x= x/x\_norm |
|  | return x |
|  |  |

**RFunctions-1.R**

|  |
| --- |
| library(dplyr) |
|  | library(ggplot2) |
|  | library(FNN) |
|  | # Split the data into training and test sets |
|  | trainTestSplit <- function(df,trainPercent,seed1){ |
|  | ## Sample size percent |
|  | samp\_size <- floor(trainPercent/100 \* nrow(df)) |
|  | ## set the seed |
|  | set.seed(seed1) |
|  | idx <- sample(seq\_len(nrow(df)), size = samp\_size) |
|  | idx |
|  |  |
|  | } |
|  |  |
|  | # Compute the R squared rsquared= 1 - RSS/TSS |
|  | Rsquared <- function(lmfit,newdf,y){ |
|  | yhat <- predict(lmfit,newdata=newdf) |
|  | RSS <- sum((y - yhat)^2) |
|  | TSS <- sum((y - mean(y))^2) |
|  | rsquared <-1 - (RSS/TSS) |
|  | rsquared |
|  | } |
|  |  |
|  | # Compute R squared for KNN predictions |
|  | knnRSquared <- function(yhat=knn$pred,y){ |
|  | RSS <- sum((test.Y - yhat)^2) |
|  | TSS <- sum((test.Y - mean(test.Y))^2) |
|  | rsquared <-1 - (RSS/TSS) |
|  | rsquared |
|  | } |
|  |  |
|  | # Create a Min Max scaler |
|  | MinMaxScaler <- function(df){ |
|  | minx=sapply(df,min) |
|  | maxx=sapply(df,max) |
|  | d=(t(df) -minx)/(maxx-minx) |
|  | e=t(d) |
|  | e |
|  | } |
|  |  |
|  |  |
|  | Rsquaredlogit<- function(model,nullmodel){ |
|  | Rsquared <-1-logLik(fit)/logLik(nullmod) |
|  | } |

**Conclusion**  
This post starts with a simple 2 layer Neural Network implementation of Logistic Regression. Clearly the performance of this simple Neural Network is comparatively poor to the highly optimized sklearn’s Logistic Regression. This is because the above neural network did not have any hidden layers. Deep Learning & Neural Networks achieve extraordinary performance because of the presence of deep hidden layers