**DEEP LEARNING**

## Mini project:2

Date:29/06/2020

Presented By:

Vismay Gunda,

Computer science engineering,

KLS Gogte institute of Technology Belagavi

Title:

**“IMPLEMENTING A LINEAR CLASSIFIER AND TRAINING IT USING STOCHASTIC GRADIENT DESCENT.”**

## 1.Packages

Import all the packages that you will need during this mini project.

a) Numpy

b) Matplotlib

c) System-specific parameters and functions.

**Python code:**

In[1]: import numpy as np

%matplotlib inline

import matplotlib.pyplot as plt

In[2]: import sys

sys.path.append("..")

## 2.Two dimensional classification

To make things more intuitive I tried to solve a 2D classification problem with synthetic data.

**Note:** Please load data and .ipynb notebook in the same folder.

**Python Code:**

In[3]: with open('train.npy', 'rb') as fin:

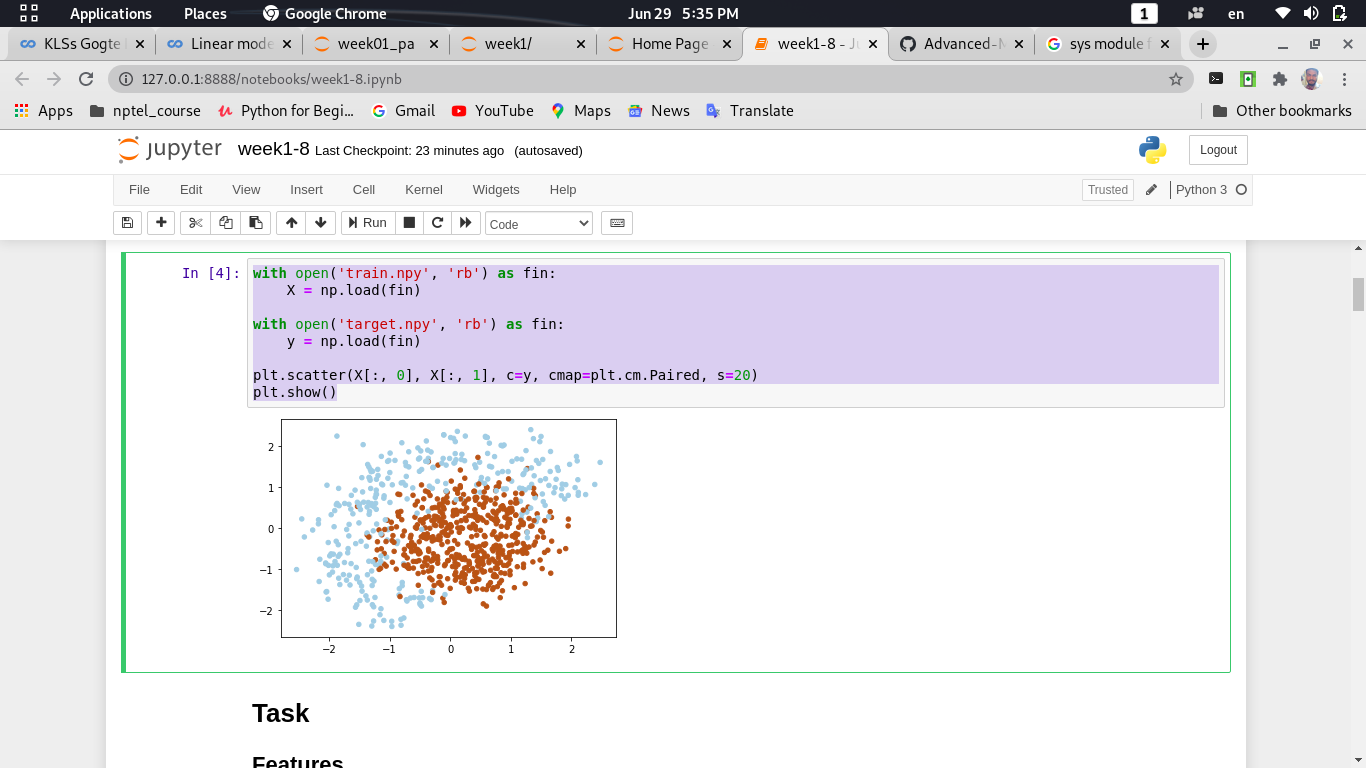
X = np.load(fin)

with open('target.npy', 'rb') as fin:

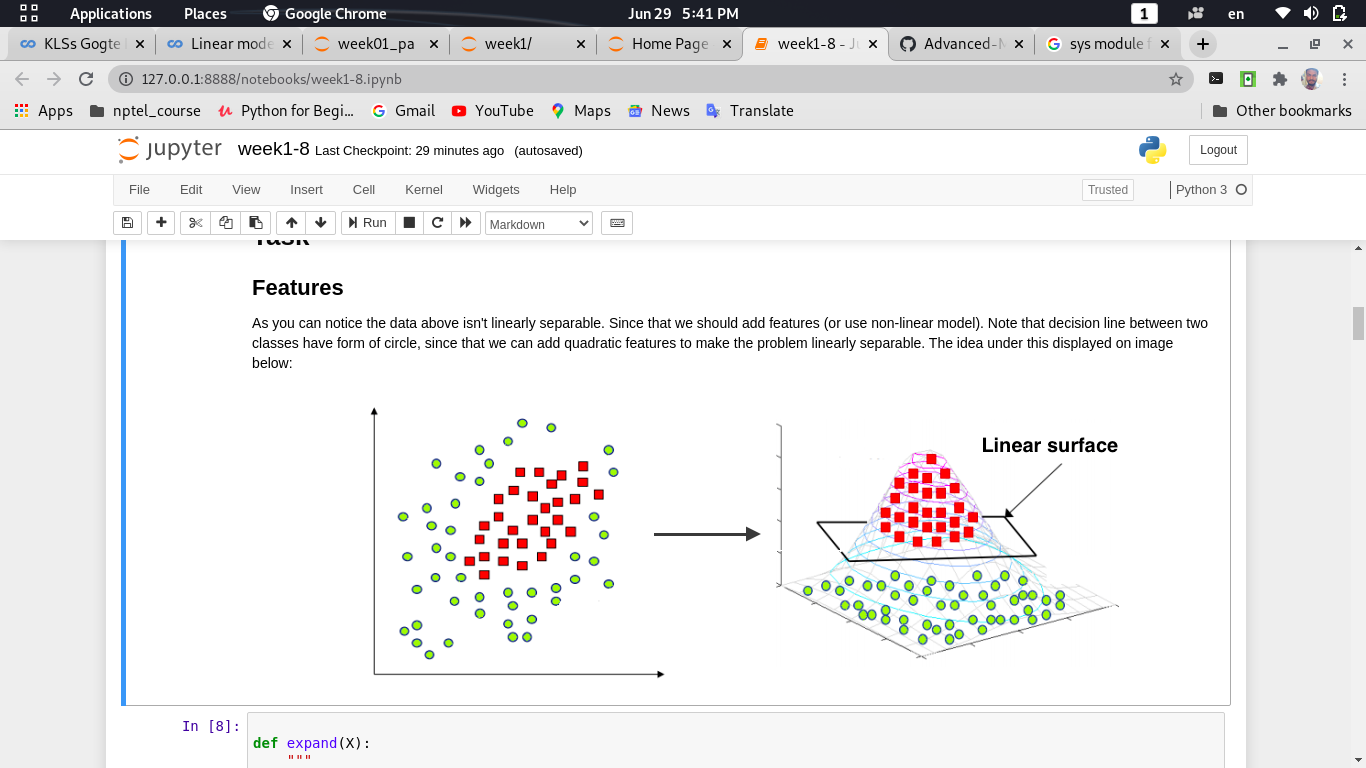
y = np.load(fin)

plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.Paired, s=20)

plt.show()

Out[4]:

## 3.Feautures

 As you can notice the data above isn't linearly separable. Since that we should add features (or use non-linear model). Note that decision line between two classes have form of circle, since that we can add quadratic features to make the problem linearly separable. The idea under this displayed on image below:

**Python Code:**

In[4]:

def expand(X):

"""

Adds quadratic features.

This expansion allows your linear model to make non-linear separation.

For each sample (row in matrix), compute an expanded row:

[feature0, feature1, feature0^2, feature1^2, feature0\*feature1, 1]

:param X: matrix of features, shape [n\_samples,2]

:returns: expanded features of shape [n\_samples,6]

"""

X\_expanded = np.zeros((X.shape[0], 6))

# TODO:<your code here>

X\_expanded[:, 0] = X[:, 0]

X\_expanded[:, 1] = X[:, 1]

X\_expanded[:, 2] = X[:, 0] \*\* 2

X\_expanded[:, 3] = X[:, 1] \*\* 2

X\_expanded[:, 4] = X[:, 0] \* X[:, 1]

X\_expanded[:, 5] = 1

return X\_expanded

In[5]: X\_expanded = expand(X)

In[6]: #Here are some tests for your implementation of expand function.

# simple test on random numbers

dummy\_X = np.array([

[0,0],

[1,0],

[2.61,-1.28],

[-0.59,2.1]

])

# call your expand function

dummy\_expanded = expand(dummy\_X)

# what it should have returned: x0 x1 x0^2 x1^2 x0\*x1 1

dummy\_expanded\_ans = np.array

([[ 0. , 0. , 0. , 0. , 0. , 1. ],

[ 1. , 0. , 1. , 0. , 0. , 1. ],

[ 2.61 , -1.28 , 6.8121, 1.6384, -3.3408,1. ],

[-0.59 , 2.1 , 0.3481, 4.41 , -1.239 , 1. ]])

#tests

assert isinstance(dummy\_expanded,np.ndarray), "please make sure you return numpy array"

assert dummy\_expanded.shape == dummy\_expanded\_ans.shape, "please make sure your shape is correct"

assert np.allclose(dummy\_expanded,dummy\_expanded\_ans,1e-3), "Something's out of order with features"

print("Seems legit!")

Out[6]: Seems legit!.

## 4.Logistic Regression

To classify objects we will obtain probability of object belongs to class '1'. To predict probability we will use output of linear model and logistic function:

𝑎(𝑥;𝑤)=⟨𝑤,𝑥⟩

𝑃(𝑦=1||𝑥,𝑤)=1/(1+exp(−⟨𝑤,𝑥⟩))=𝜎(⟨𝑤,𝑥⟩)

**Python Code:**

In[7]: def probability(X, w):

"""

Given input features and weights

return predicted probabilities of y==1 given x, P(y=1|x), see description above

Don't forget to use expand(X) function (where necessary) in this and subsequent functions.

:param X: feature matrix X of shape [n\_samples,6] (expanded)

:param w: weight vector w of shape [6] for each of the expanded features

:returns: an array of predicted probabilities in [0,1] interval.

"""

# TODO:<your code here>

return 1 / (1 + np.exp(-np.dot(X, w)))

In[8]: dummy\_weights = np.linspace(-1, 1, 6)

ans\_part1 = probability(X\_expanded[:1, :], dummy\_weights)[0]

## 5.Computing Loss

In logistic regression the optimal parameters 𝑤 are found by cross-entropy minimization:

Loss for one sample:

𝑙(𝑥𝑖,𝑦𝑖,𝑤)=−[𝑦𝑖⋅𝑙𝑜𝑔𝑃(𝑦𝑖=1|𝑥𝑖,𝑤)+(1−𝑦𝑖)⋅𝑙𝑜𝑔(1−𝑃(𝑦𝑖=1|𝑥𝑖,𝑤))]

Loss for many samples:

𝐿(𝑋,𝑦⃗ ,𝑤)=1ℓ∑𝑖=1ℓ𝑙(𝑥𝑖,𝑦𝑖,𝑤)

**Python Code:**

In[9]: def compute\_loss(X, y, w):

"""

Given feature matrix X [n\_samples,6], target vector [n\_samples] of 1/0,

and weight vector w [6], compute scalar loss function L using formula above.

Keep in mind that our loss is averaged over all samples (rows) in X.

"""

# TODO:<your code here>

p = probability(X, w)

return -np.mean(y \* np.log(p) + (1 - y) \* np.log(1 – p))

In[10]: #use output of this cell to fill answer field

ans\_part2 = compute\_loss(X\_expanded, y, dummy\_weights)

## 5.Computing Gradient descent

Since we train our model with gradient descent, we should compute gradients.

To be specific, we need a derivative of loss function over each weight [6 of them].

∇𝑤𝐿=1/(ℓ∑𝑖=1ℓ∇𝑤𝑙(𝑥𝑖,𝑦𝑖,𝑤))

**Python Code:**

In[11]: def compute\_grad(X, y, w):

"""

Given feature matrix X [n\_samples,6], target vector [n\_samples] of 1/0,

and weight vector w [6], compute vector [6] of derivatives of L over each weights.

Keep in mind that our loss is averaged over all samples (rows) in X.

"""

# TODO<your code here>

return np.dot(X.T, probability(X, w) - y) / X.shape[0]

In[12]: # use output of this cell to fill answer field

ans\_part3 = np.linalg.norm(compute\_grad(X\_expanded, y, dummy\_weights))

6.Auxiliary function for visualisation

**Python Code:**

In[13]:from IPython import display

h = 0.01

x\_min, x\_max = X[:, 0].min() - 1, X[:, 0].max() + 1

y\_min, y\_max = X[:, 1].min() - 1, X[:, 1].max() + 1

xx, yy = np.meshgrid(np.arange(x\_min, x\_max, h), np.arange(y\_min, y\_max, h))

def visualize(X, y, w, history):

"""draws classifier prediction with matplotlib magic"""

Z = probability(expand(np.c\_[xx.ravel(), yy.ravel()]), w)

Z = Z.reshape(xx.shape)

plt.subplot(1, 2, 1)

plt.contourf(xx, yy, Z, alpha=0.8)

plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.Paired)

plt.xlim(xx.min(), xx.max())

plt.ylim(yy.min(), yy.max())

plt.subplot(1, 2, 2)

plt.plot(history)

plt.grid()

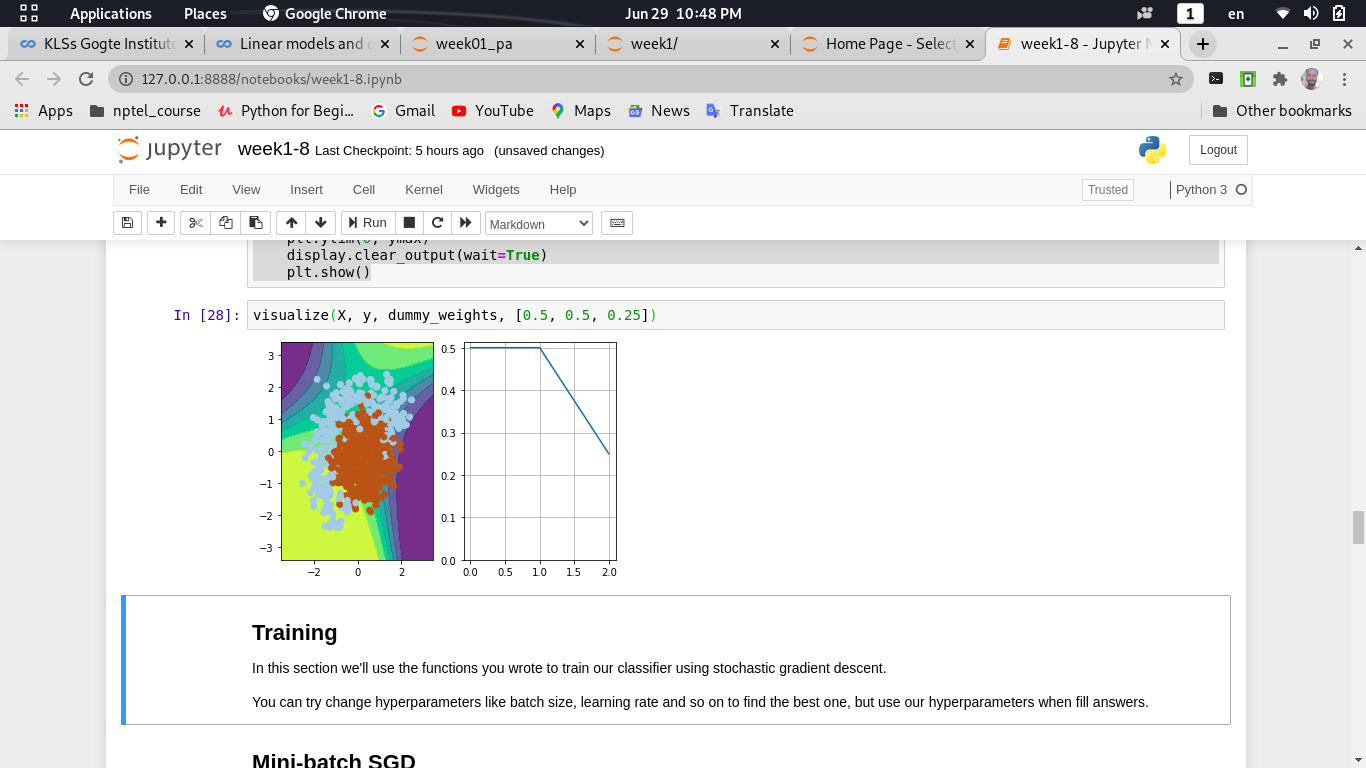
ymin, ymax = plt.ylim()

plt.ylim(0, ymax)

display.clear\_output(wait=True)

plt.show()

In[14]: visualize(X, y, dummy\_weights, [0.5, 0.5, 0.25])

 Out[14]:

7.Training the Model

In this section i will use the functions you wrote to train our classifier using stochastic gradient descent.

You can try change hyperparameters like batch size, learning rate and so on to find the best one, but use our hyperparameters when fill answers.

**Note:**  **Mini-batch SGD:**

Stochastic gradient descent just takes a random batch of 𝑚 samples on each iteration, calculates a gradient of the loss on it and makes a step:

𝑤𝑡=𝑤(𝑡−1)−𝜂(1/𝑚(∑𝑗=1𝑚∇𝑤𝑙(𝑥𝑖𝑗,𝑦𝑖𝑗,𝑤𝑡)))

**Python Code:**

In[15]: # please use np.random.seed(42), eta=0.1, n\_iter=100 and batch\_size=4 for deterministic results

np.random.seed(42)

w = np.array([0, 0, 0, 0, 0, 1])

eta= 0.1 # learning rate

n\_iter = 100

batch\_size = 4

loss = np.zeros(n\_iter)

plt.figure(figsize=(12, 5))

for i in range(n\_iter):

ind = np.random.choice(X\_expanded.shape[0], batch\_size)

loss[i] = compute\_loss(X\_expanded, y, w)

if i % 10 == 0:

visualize(X\_expanded[ind, :], y[ind], w, loss)

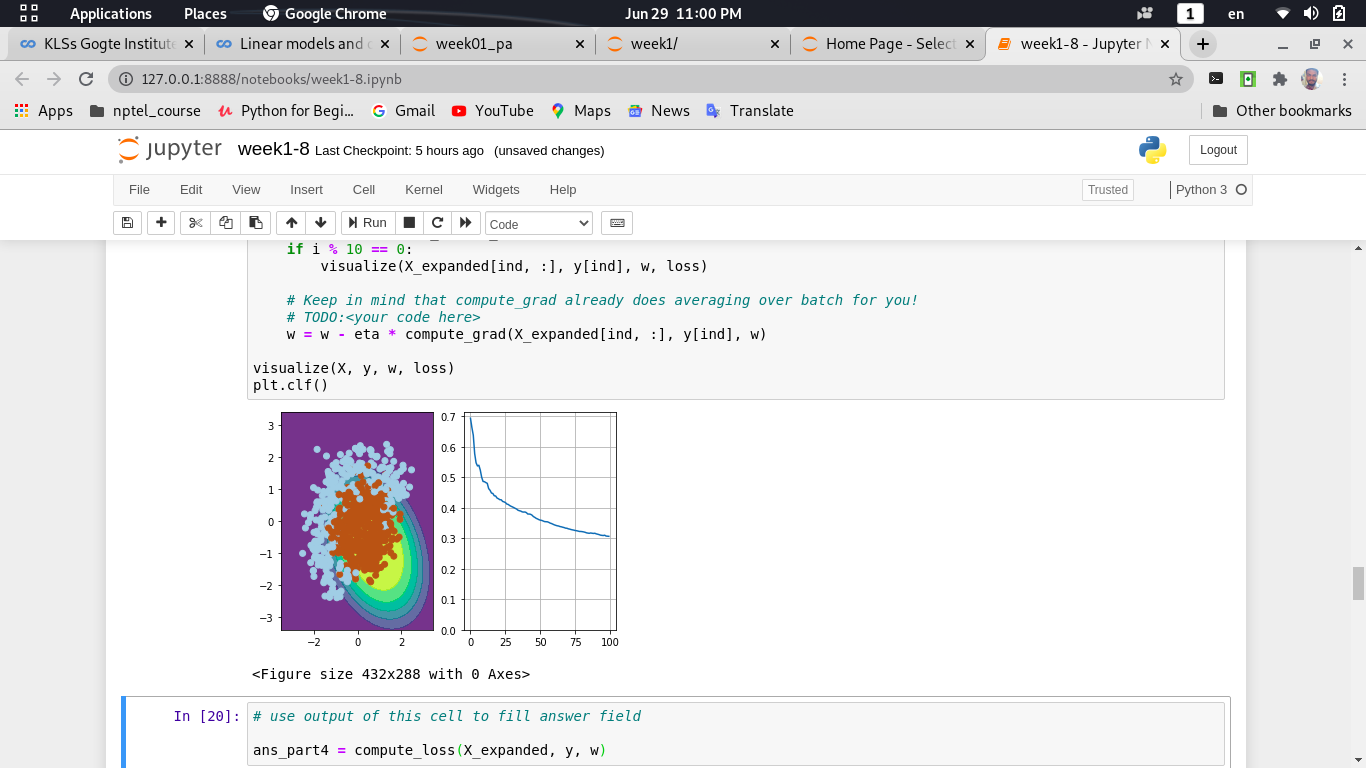
# Keep in mind that compute\_grad already does averaging over batch for you!

# TODO:<your code here>

w = w - eta \* compute\_grad(X\_expanded[ind, :], y[ind], w)

visualize(X, y, w, loss)

plt.clf()

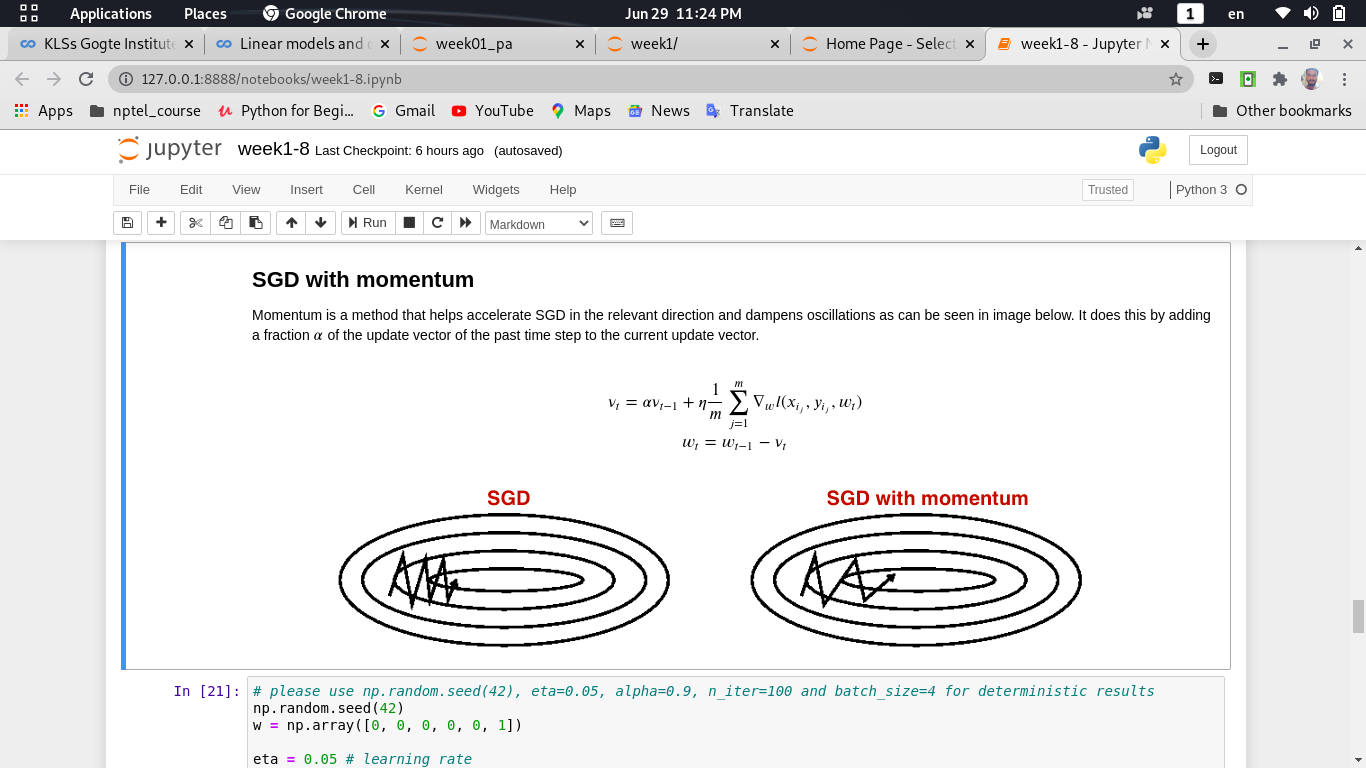
 Out[15]:

In[16]: # use output of this cell to fill answer field

ans\_part4 = compute\_loss(X\_expanded, y, w)

8. SGD With Momentum:

Momentum is a method that helps accelerate SGD in the relevant direction and dampens oscillations as can be seen in image below. It does this by adding a fraction 𝛼 of the update vector of the past time step to the current update vector.



**Python Code:**

In[17]: # please use np.random.seed(42), eta=0.05, alpha=0.9, n\_iter=100 and batch\_size=4 for deterministic results

np.random.seed(42)

w = np.array([0, 0, 0, 0, 0, 1])

eta = 0.05 # learning rate

alpha = 0.9 # momentum

nu = np.zeros\_like(w)

n\_iter = 100

batch\_size = 4

loss = np.zeros(n\_iter)

plt.figure(figsize=(12, 5))

for i in range(n\_iter):

ind = np.random.choice(X\_expanded.shape[0], batch\_size)

loss[i] = compute\_loss(X\_expanded, y, w)

if i % 10 == 0:

visualize(X\_expanded[ind, :], y[ind], w, loss)

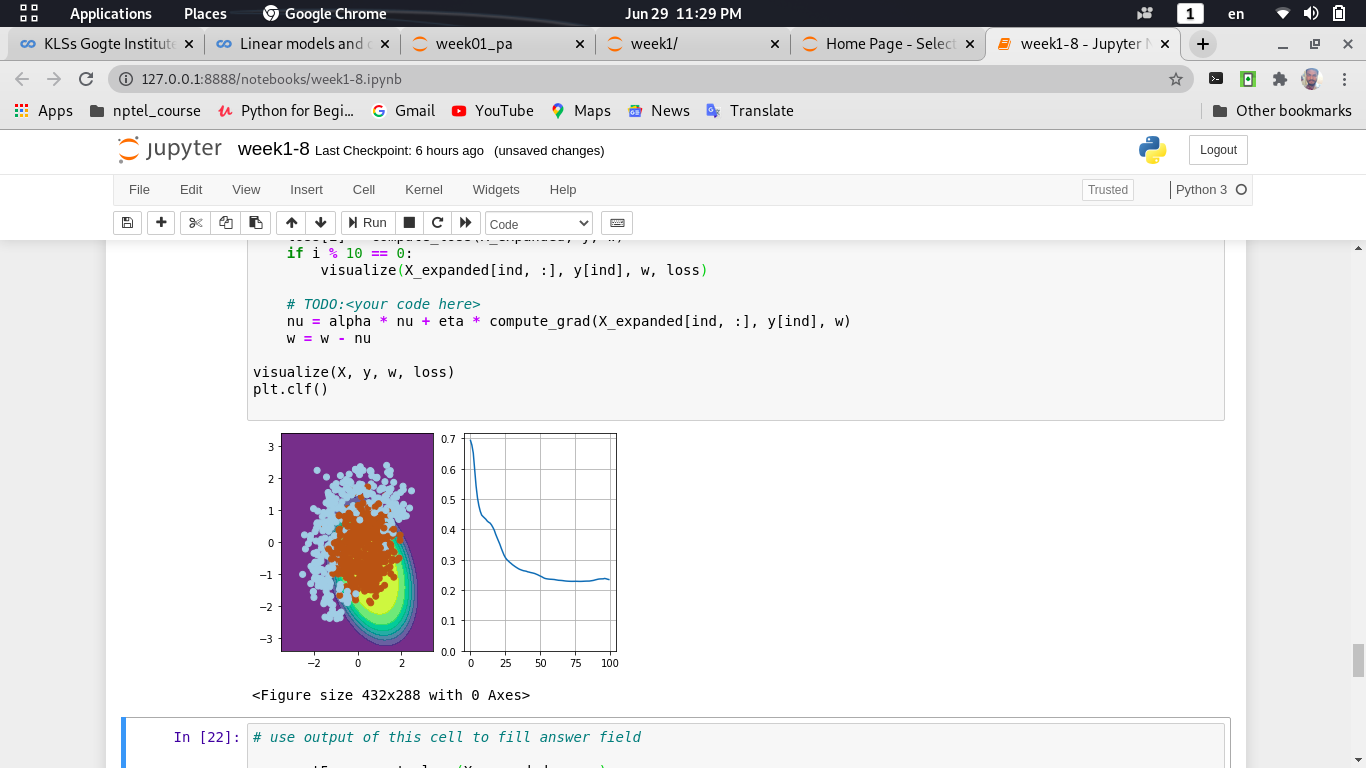
# TODO:

nu = alpha \* nu + eta \* compute\_grad(X\_expanded[ind, :], y[ind], w)

w = w - nu

visualize(X, y, w, loss)

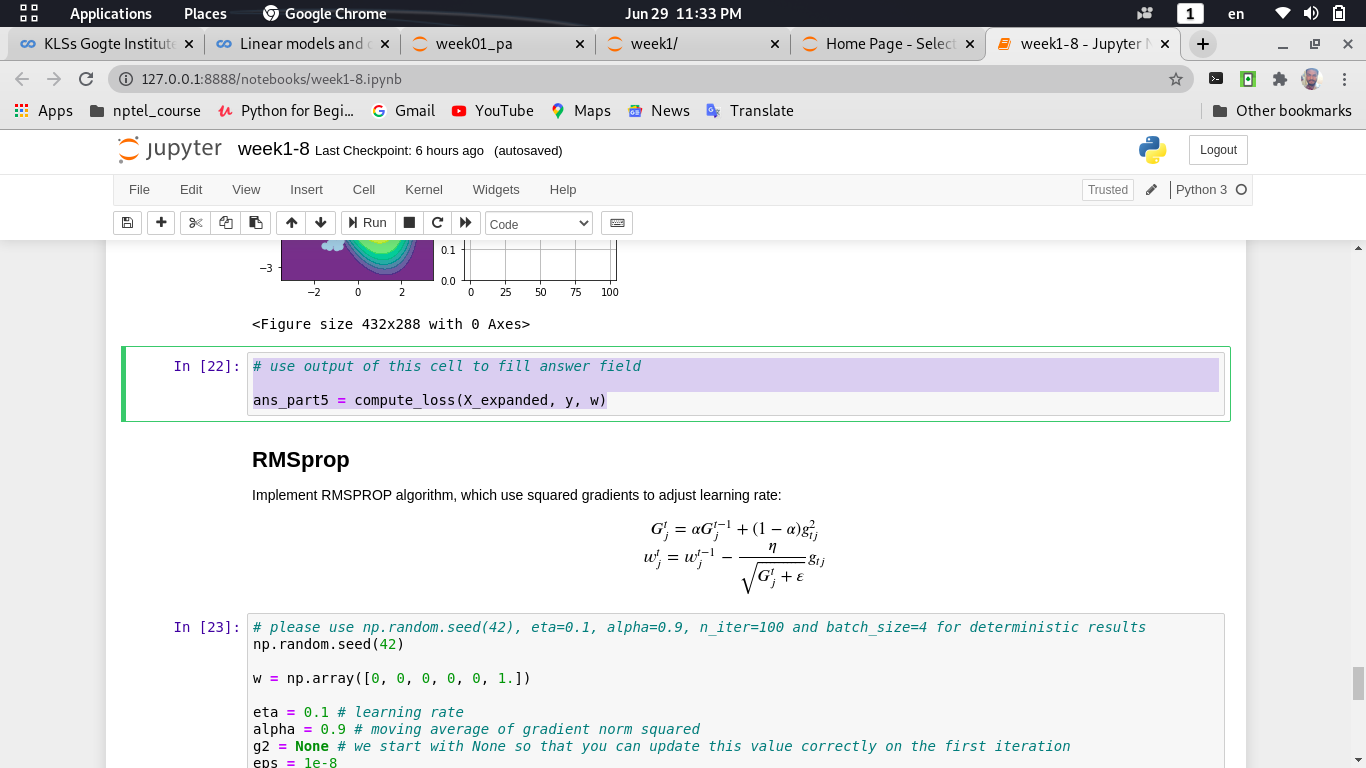
plt.clf()

 Out[17]:

In[18]: # use output of this cell to fill answer field

ans\_part5 = compute\_loss(X\_expanded, y, w)

9.RMSprop

 Implement RMSPROP algorithm,which use squared gradients to adjust learning rate.

**Python Code:**

In[23]: # please use np.random.seed(42), eta=0.1, alpha=0.9, n\_iter=100 and batch\_size=4 for deterministic results

np.random.seed(42)

w = np.array([0, 0, 0, 0, 0, 1.])

eta = 0.1 # learning rate

alpha = 0.9 # moving average of gradient norm squared

g2 = None # we start with None so that you can update this value correctly on the first iteration

eps = 1e-8

G = 0

n\_iter = 100

batch\_size = 4

loss = np.zeros(n\_iter)

plt.figure(figsize=(12,5))

for i in range(n\_iter):

ind = np.random.choice(X\_expanded.shape[0], batch\_size)

loss[i] = compute\_loss(X\_expanded, y, w)

if i % 10 == 0:

visualize(X\_expanded[ind, :], y[ind], w, loss)

g = compute\_grad(X\_expanded[ind, :], y[ind], w)

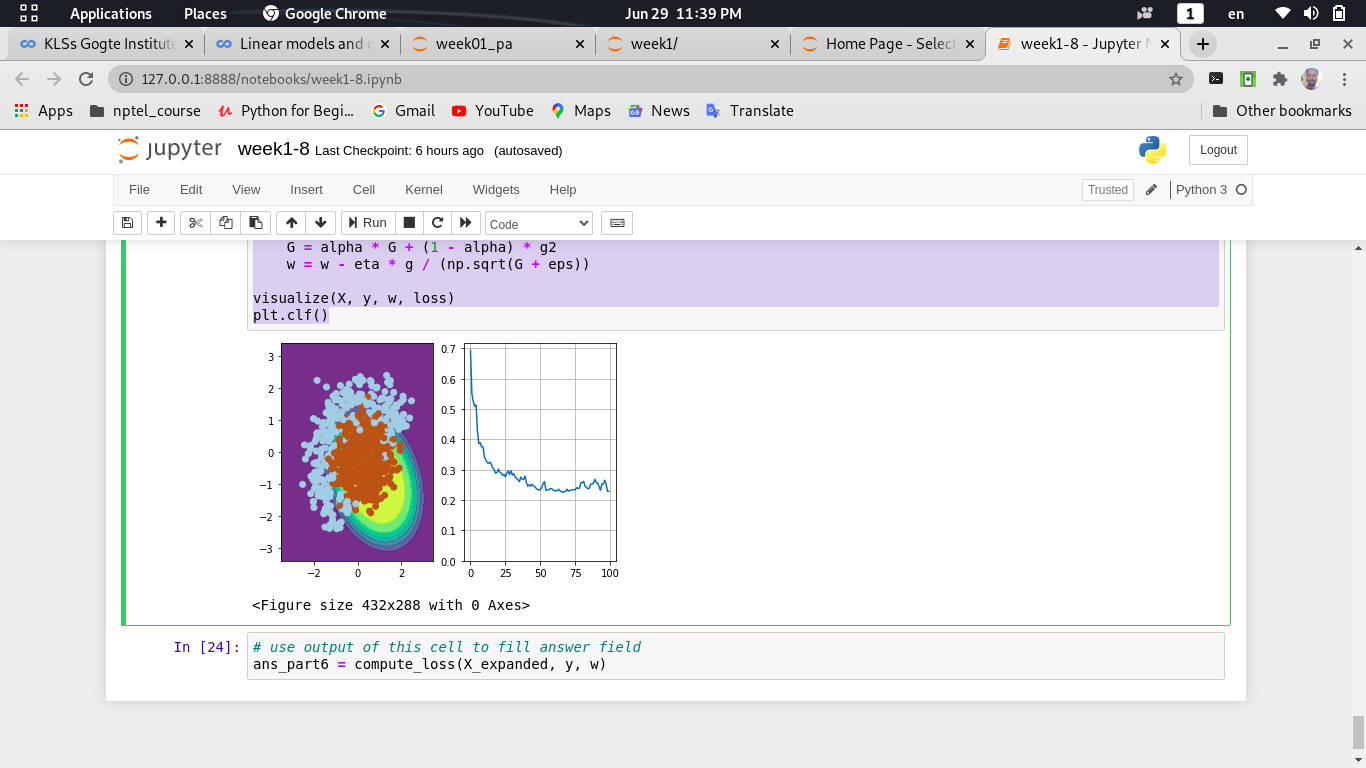
g2 = g \*\* 2

G = alpha \* G + (1 - alpha) \* g2

w = w - eta \* g / (np.sqrt(G + eps))

visualize(X, y, w, loss)

plt.clf()

 Out[23]:

In[24]: # use output of this cell to fill answer field

ans\_part6 = compute\_loss(X\_expanded, y, w)