Assignment 1

September 12, 2021

1 Linear Regression

Instructions for running code

In the folder Q1/, run the following script:

`bash run.sh ../../data/q1/linearX.csv ../../data/q1/linearY.csv`

This will run all the sections of question 1

- Reading data and labels
- Normalizing the data
- Inserting intercept

[3]:
```
# set appropriate data path
data_path = "/Users/suchith720/Desktop/ml_parag/assignment/A1/data/"

# reading data
X = pd.read_csv(f"{data_path}/q1/linearX.csv", header=None).to_numpy()
Y = pd.read_csv(f"{data_path}/q1/linearY.csv", header=None).to_numpy()

# normalizing the data
X = (X - X.mean(axis=0))/X.std(axis=0)

# adding intercept to X_train
X = np.hstack([X, np.ones( (X.shape[0], 1) )])
```

Splitting the data into training and validation set and randomly shuffling it.

[4]:
```
valid_pc = 0.8
n_train = int(X.shape[0]*valid_pc)
```

[5]:
```
rnd_idx = np.random.permutation(X.shape[0])
X_train, Y_train = X[rnd_idx[:n_train]], Y[rnd_idx[:n_train]]
X_valid, Y_valid = X[rnd_idx[n_train:]], Y[rnd_idx[n_train:]]
```

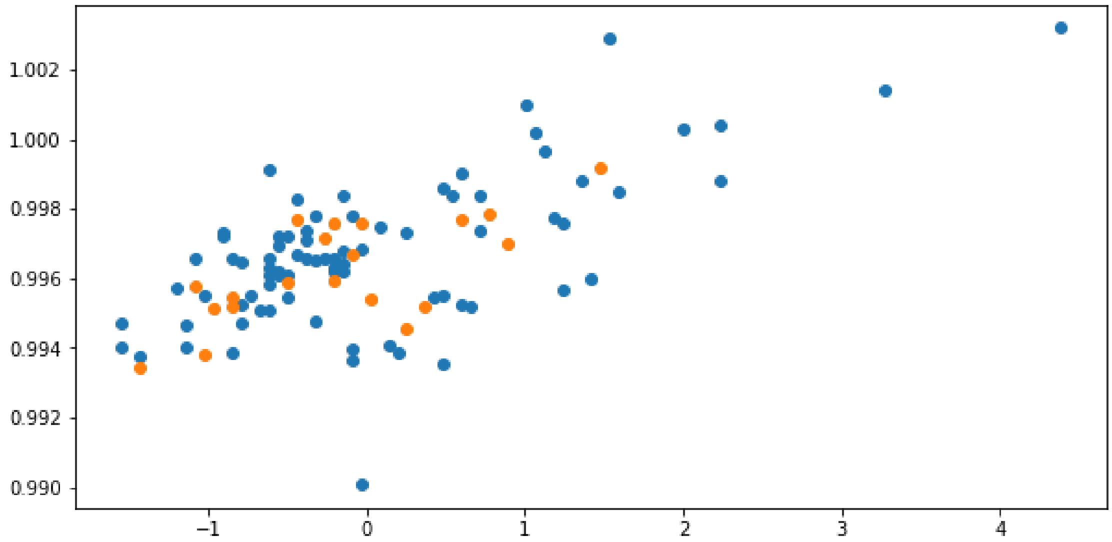
[6]:
```
X_train.shape, Y_train.shape, X_valid.shape, Y_valid.shape
```

[6]:
```
((80, 2), (80, 1), (20, 2), (20, 1))
```

1.1 a. Implementing batch gradient descent

Visualizing the input data

[7]: <matplotlib.collections.PathCollection at 0x7fca62ff6d10>



1.1.1 Important functions

This is the linear function

[15]:
```
def linear(X, theta):
    return X@theta
```

Implementation of square error loss

[16]:
```
def linear_loss(X, theta, Y):
    n = X.shape[0]

    Y_hat = linear(X, theta)
    l = (Y_hat - Y).T @ (Y_hat - Y)
    l /= (2*n)

    return l[0][0]
```

gradient of the loss function

[17]:
```
def linear_grad(X, theta, Y):
    n = X.shape[0]
    Y_hat = linear(X, theta)
    grad_theta = X.T@(Y_hat - Y)
    grad_theta /= n
```

```
    return grad_theta
```

1.1.2 Stopping criteria

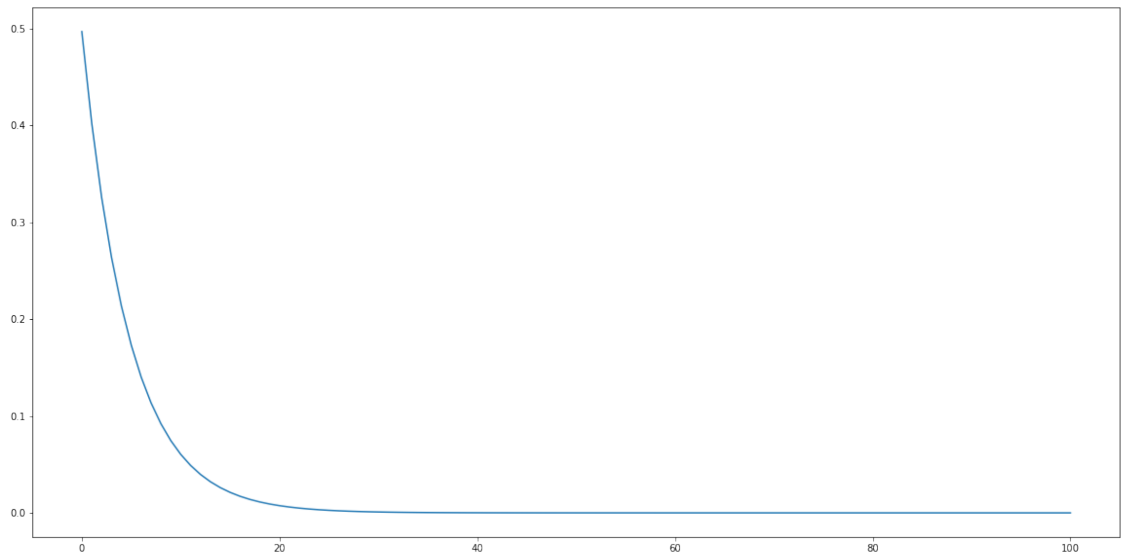
Here I have experimented with three stopping criteria

1. **number of iterations**

function `batch_gradient_descent_1` implements this function present in `1a.py`

[22]:
```
training_losses, theta_values = batch_gradient_descent_1(X_train, Y_train,
                                                         lr=0.1, num_iter=100)
plt.plot(training_losses)
```

[22]: `[<matplotlib.lines.Line2D at 0x7fca63343f10>]`

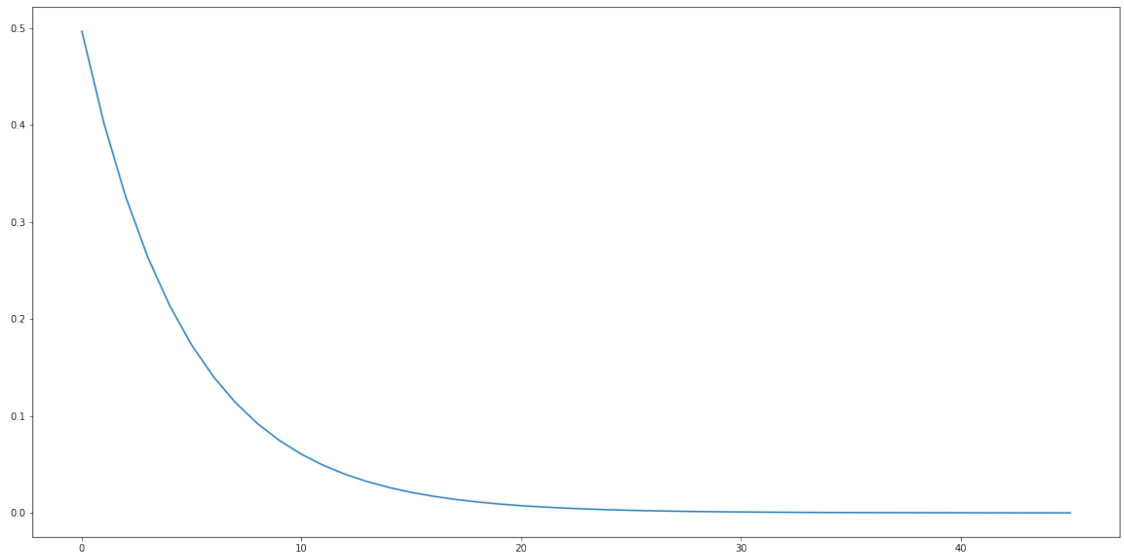


2. **$|\frac{\partial J(\theta)}{\partial x}| < \varepsilon$, where $J(\theta)$ is the cost function.**

function `batch_gradient_descent_2` implements this function present in `1a.py`

[23]:
```
training_losses, theta_values = batch_gradient_descent_2(X_train, Y_train, lr=0.
 ↪1, eps=1e-3)
plt.plot(training_losses)
```

[23]: `[<matplotlib.lines.Line2D at 0x7fca5099c290>]`

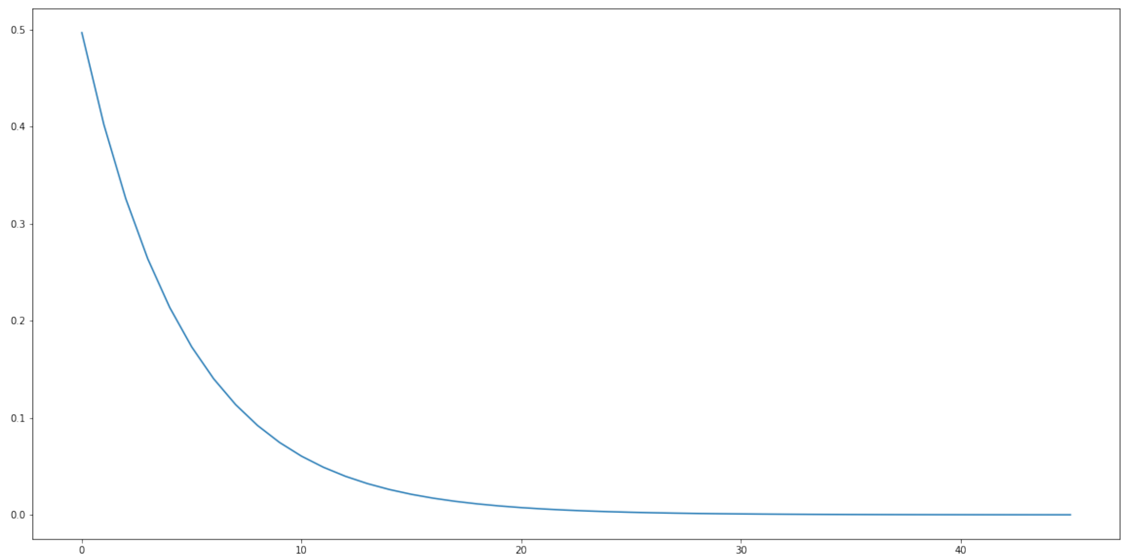


3. $|\theta_{t+1} - \theta_t| < \varepsilon$

function `batch_gradient_descent_3` implements this function present in `1a.py`

[24]:
```
training_losses, theta_values = batch_gradient_descent_3(X_train, Y_train, lr=0.
↪1, eps=1e-2)
plt.plot(training_losses)
```

[24]: `[<matplotlib.lines.Line2D at 0x7fca50a0bcd0>]`



Above it can be seen that the loss graphs for the criteria 2 and 3 are the same confirming

$$|\theta_{t+1} - \theta_t| = |\eta \frac{\partial J(\theta)}{\partial x}|$$

The stopping criteria I have choosen is a combination of number of iterations and the absolute value of the gradient.

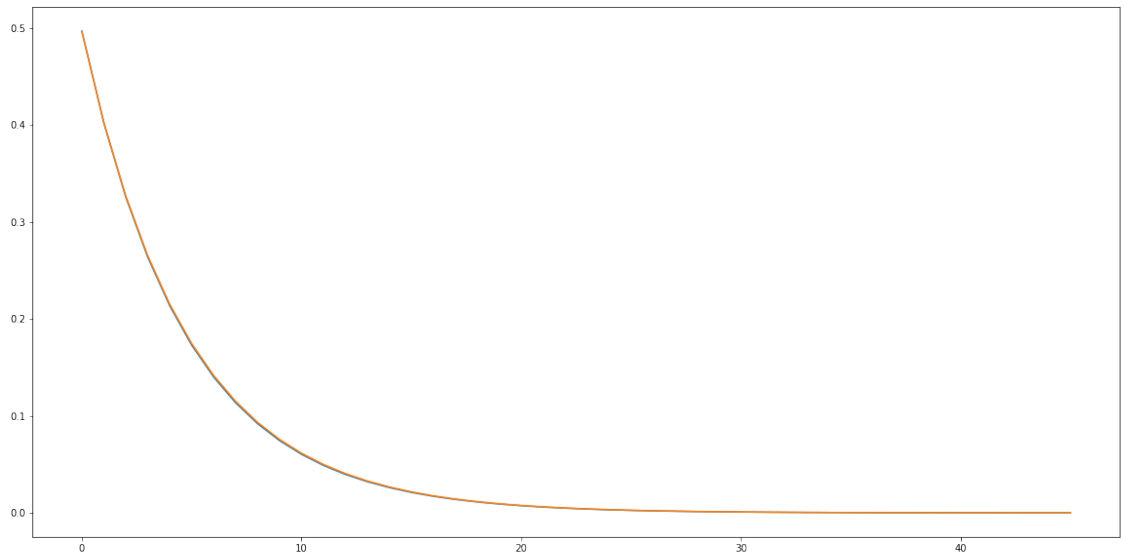
[ ]:
```
if (np.abs(dtheta) < eps).all() or max_iter < num_iter:
    break
```

Implemenation of the batch gradient descent

[25]:
```
def batch_gradient_descent(X_train, Y_train, X_valid, Y_valid, lr=0.1,
                           eps=1e-2, max_iter=100):

    #initializing theta to zero
    theta = np.zeros((2, 1))

    thetas, train_loss, valid_loss = [], [], []
    thetas.append(theta.copy())
    train_loss.append(linear_loss(X_train, theta, Y_train))
    valid_loss.append(linear_loss(X_valid, theta, Y_valid))

    num_iter = 0
    while True:
        #descent step
        dtheta = linear_grad(X_train, theta, Y_train)
        theta -= lr * dtheta

        train_loss.append(linear_loss(X_train, theta, Y_train))
        valid_loss.append(linear_loss(X_valid, theta, Y_valid))
        thetas.append(theta.copy())
        num_iter += 1

        #stopping condition
        if (np.abs(dtheta) < eps).all() or max_iter < num_iter:
            break

    return thetas, train_loss, valid_loss
```

[26]:
```
thetas, train_loss, valid_loss = batch_gradient_descent(X_train, Y_train,␣
↪X_valid, Y_valid,
                                                         lr=0.1, eps=1e-2)
plt.plot(train_loss)
plt.plot(valid_loss)
```

[26]: [<matplotlib.lines.Line2D at 0x7fca400d4050>]

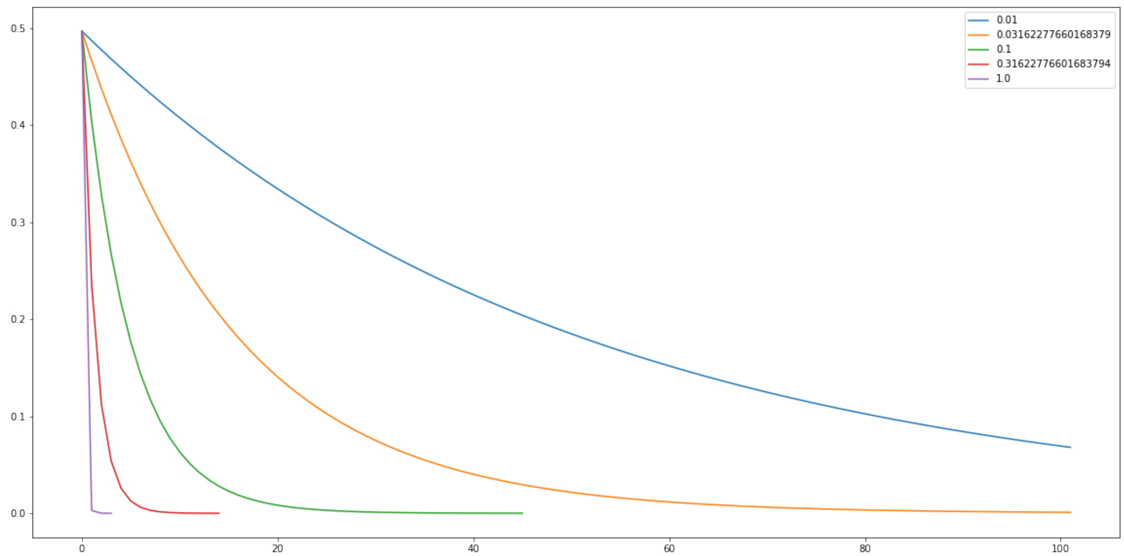


Both training and validation losses are superimposing each other

1.1.3 Selecting the learning rate

[29]:
```python
lrs = 10**np.linspace(-2, 0, 5)
train_losses, valid_losses = [], []
for lr in lrs:
    thetas, train_loss, valid_loss = batch_gradient_descent(X_train, Y_train, X_valid, Y_valid,
                                                             lr=lr, eps=1e-2)
    train_losses.append(train_loss)
    valid_losses.append(valid_loss)
```

[149]:
```
<matplotlib.legend.Legend at 0x7fccc1f222d0>
```



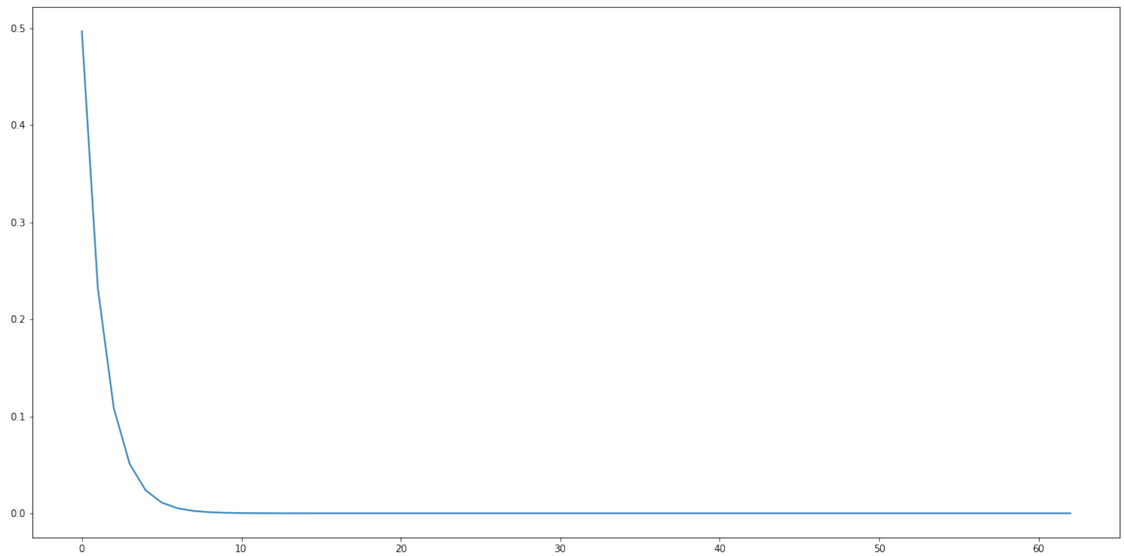
So from the above graph we can see that 0.316 is a good learing rate. The best one is 1, but proper visualization of the training process have choose 0.316

1.1.4 Training entire data

[30]:
```
X_data = np.vstack([X_train, X_valid])
Y_data = np.vstack([Y_train, Y_valid])
```

[31]:
```
thetas, train_loss, valid_loss = batch_gradient_descent(X_data, Y_data,␣
↪X_valid, Y_valid,
                                                        lr=lrs[-2], eps=1e-10)
plt.plot(train_loss)
```

[31]:
```
[<matplotlib.lines.Line2D at 0x7fca50a59ed0>]
```

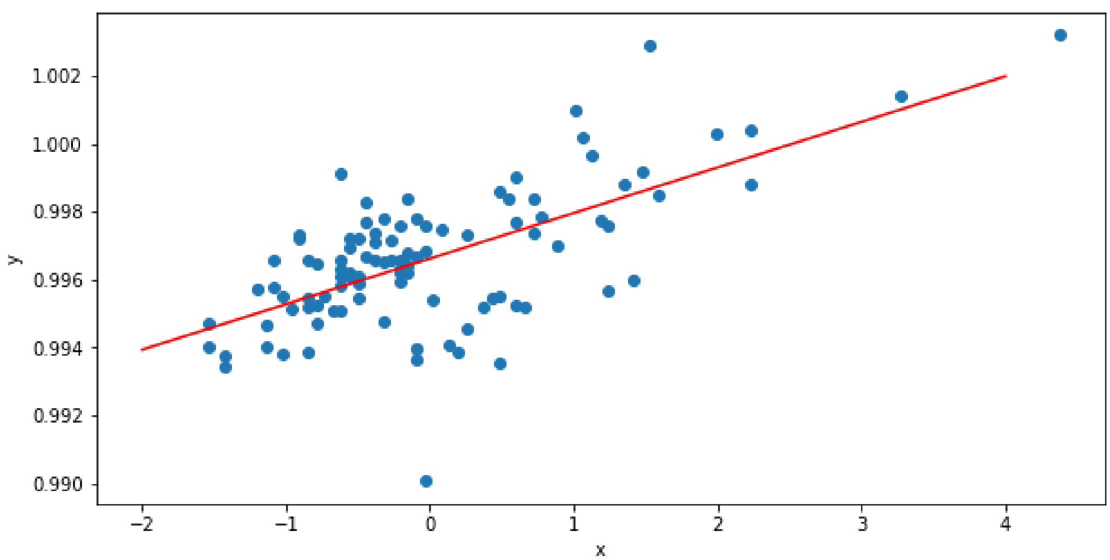


1.1.5 Conclusion

- **Learning rate** : 0.316
- **Final parameters** : $\theta_1 = 0.0013402$, $\theta_0 = 0.9966201$
- **Stopping criteria**: number of iteration and $|\frac{\partial J(\theta)}{\partial x}| < \varepsilon$ where $\varepsilon = 10^{-10}$

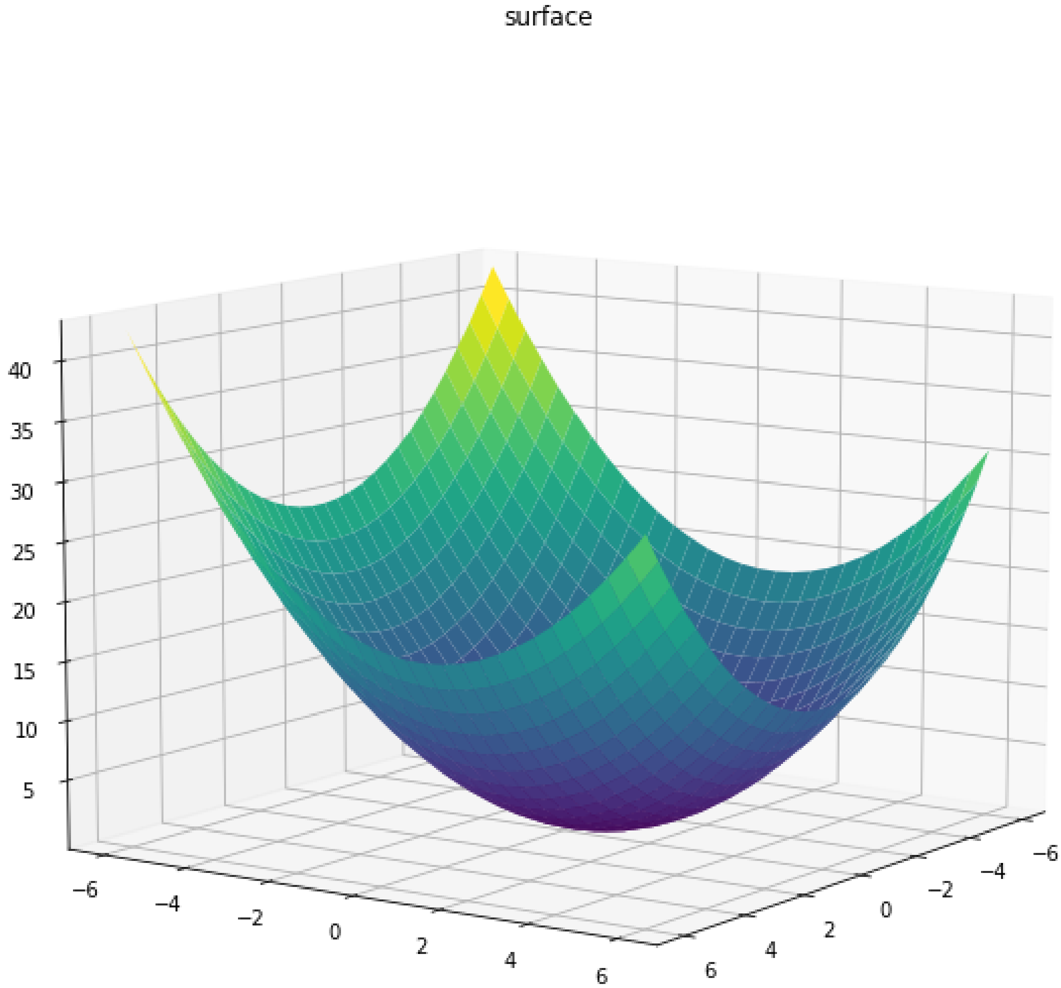
1.2 b. Plotting graphs

```
[33]: [<matplotlib.lines.Line2D at 0x7fca50aab7d0>]
```

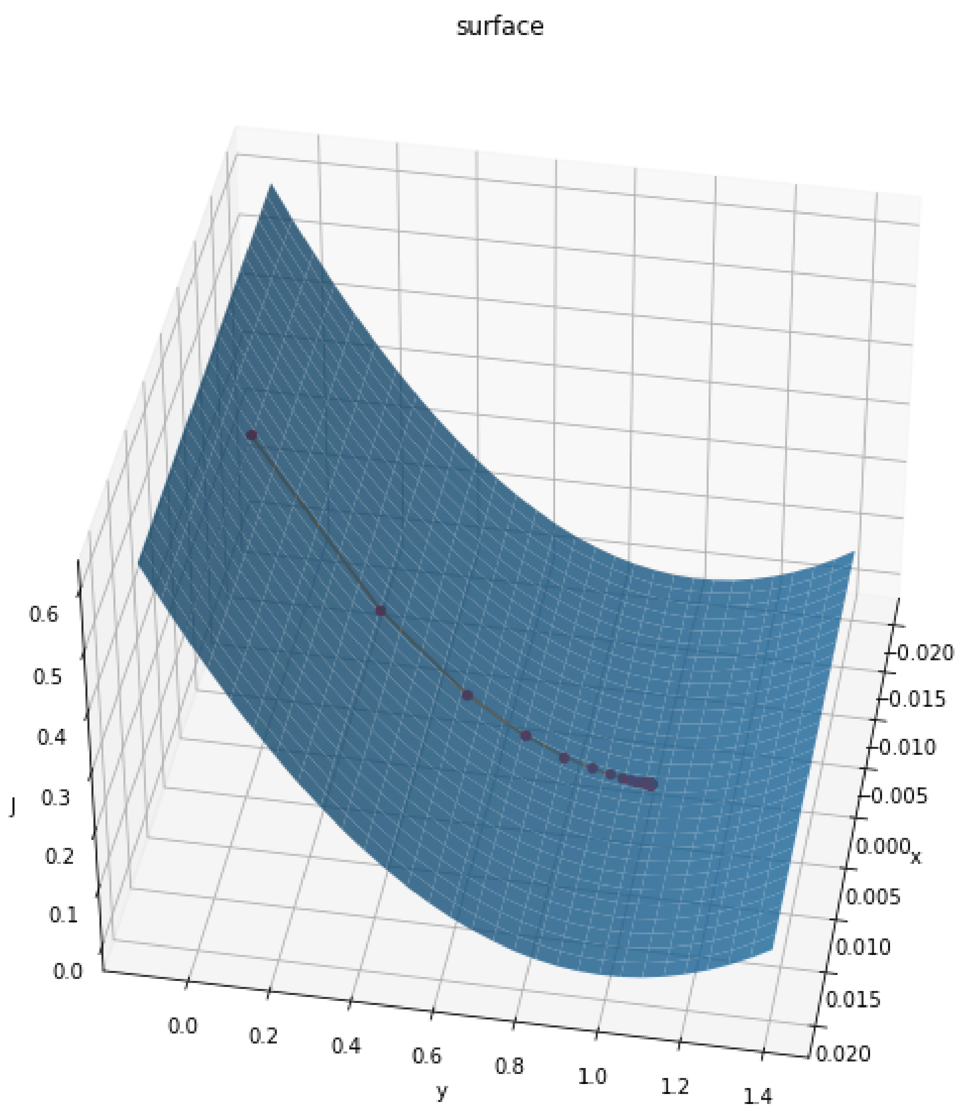


1.3 c. 3D surface

The error function in 3D dimension looks like a bowl



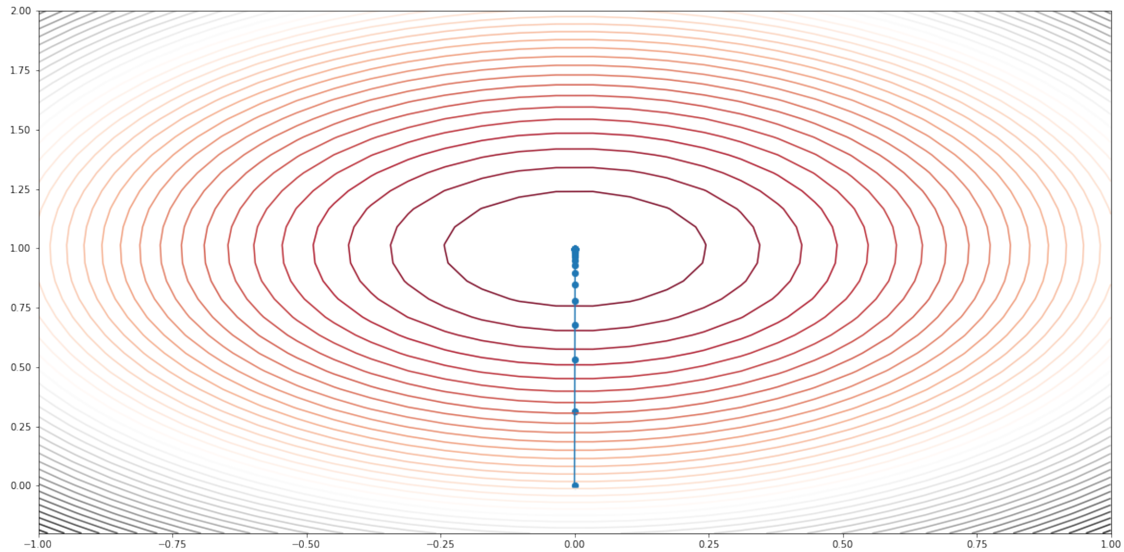
Tracking the change in the error function during training



Please run the code for animation

1.4 d. Contour

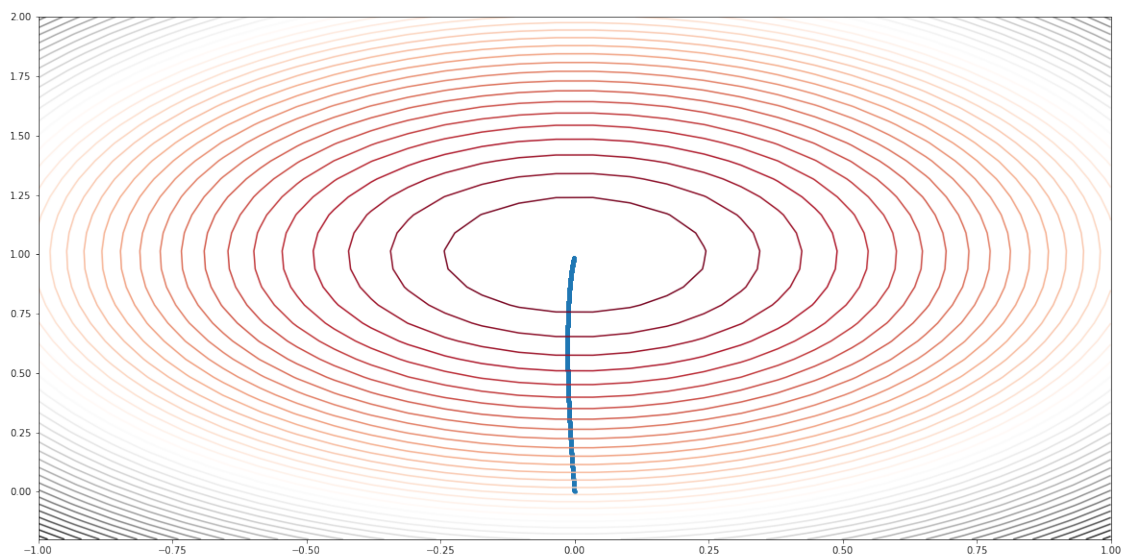
[43]: <matplotlib.contour.QuadContourSet at 0x7fca80a24150>



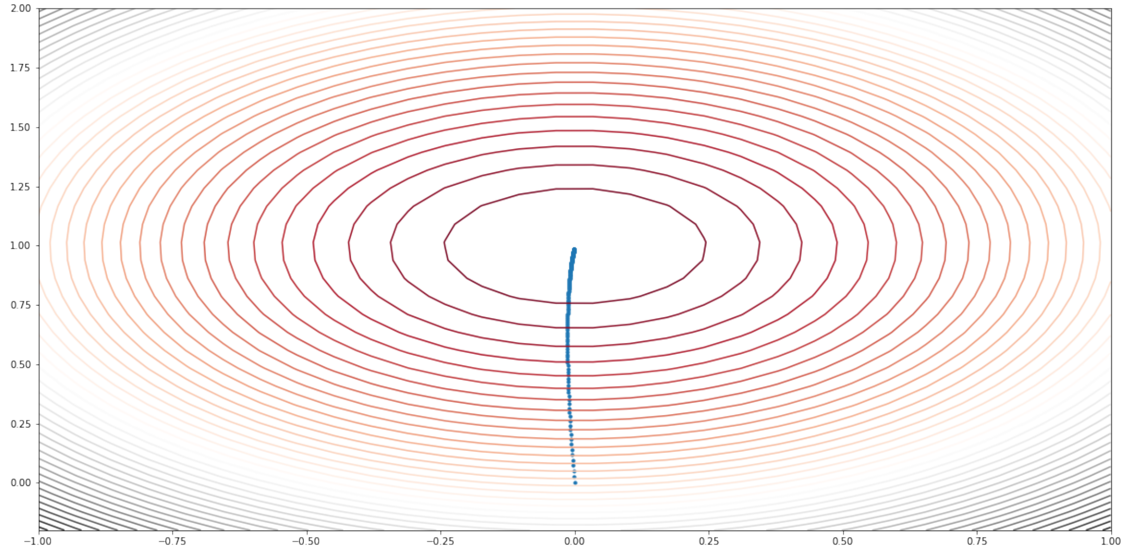
Please run the code for animation

1.5 e. Contours for different learning rate

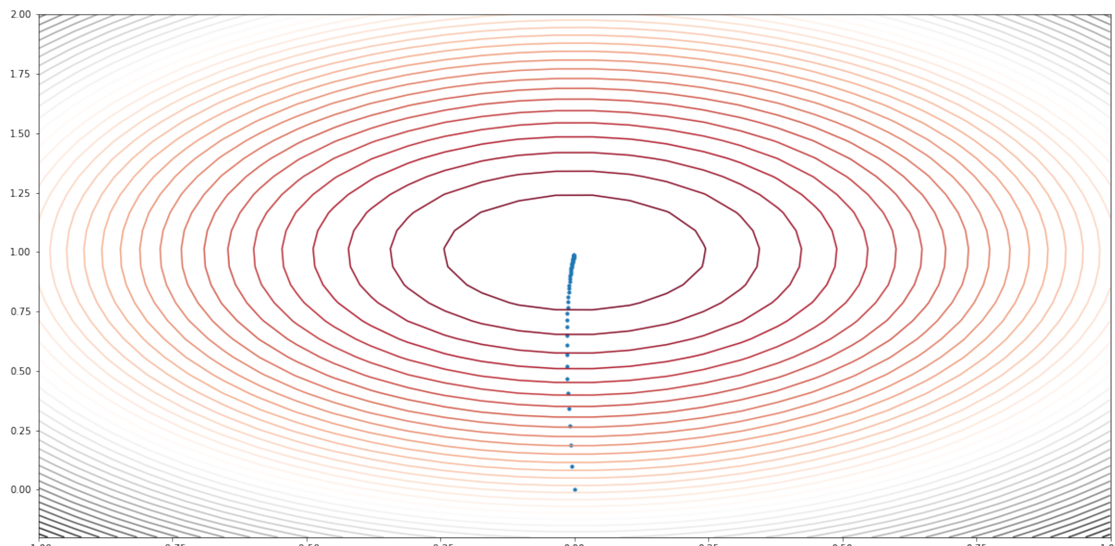
1.5.1 learning rate : 0.001



1.5.2 learning rate : 0.025



1.5.3 learning rate : 0.1



1.5.4 Conclusion

We can see that the density of points drawn is in decreasing order from 0.001 to 0.1. So if the step size very small is takes a large number of iterations to reach the optimal value.

2 Sampling and Stochastic Gradient Descent

Instructions for running code

In the folder Q2/, run the following script:

`bash run.sh ../../data/q2/q2test.csv`

This will run all the sections of question 2

2.1 a. Sampling

Code for sampling the data:

$x_1 \sim \mathcal{N}(3, 4)$

$x_2 \sim \mathcal{N}(-1, 4)$

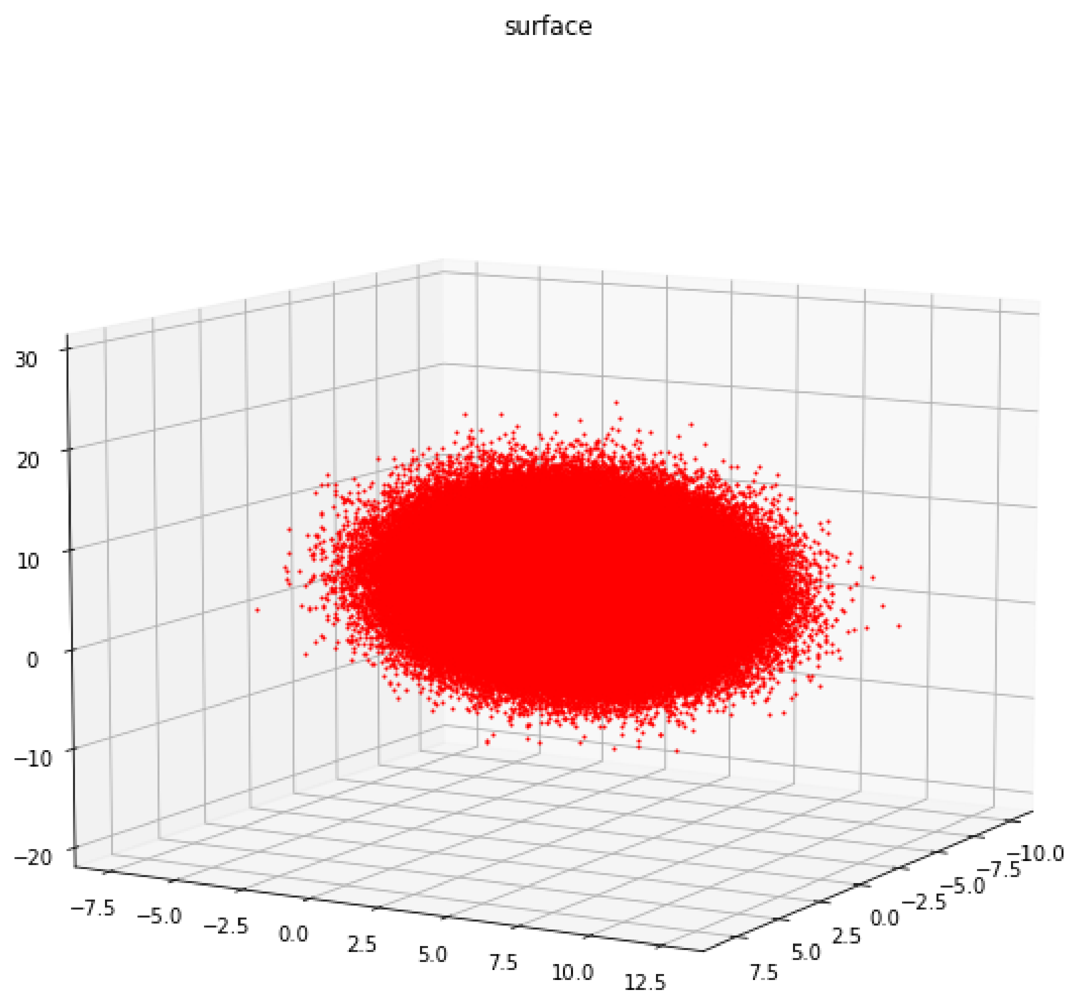
$\epsilon \sim \mathcal{N}(0, 2)$

[301]:
```
np.random.seed(10)
n = 10**6
x1 = np.random.randn(n, 1)
x2 = np.random.randn(n, 1)
e = np.random.randn(n, 1)
```

[302]:
```
x1 = 2*x1 + 3
x2 = 2*x2 - 1
e = np.sqrt(2)*e
```

[303]:
```
X_train = np.hstack([x2, x1, np.ones((n, 1))])

theta = np.array([2, 1, 3]).reshape(-1, 1)

Y_train = linear(X_train, theta) + e
```

Visualize surface



2.2 b. Stochatic gradient descent

Below is the code for SGD:

[645]:
```
def sgd(X_train, Y_train, bs=100, lr=0.1, k=50, eps=1e-6,  max_iter=1000):
    theta = np.zeros((3, 1))

    thetas, train_loss = [], []

    thetas.append(theta.copy())
    train_loss.append(linear_loss(X_train[:bs, :], theta, Y_train[:bs, :]))
```

```
    num_iter = 0
    avg_dtheta = 0

    iter_per_epoch = math.ceil(X_train.shape[0]/bs)

    while True:
        #creating a mini-batch
        bn = num_iter%iter_per_epoch
        b_start = int(bn*bs)
        X_batch, Y_batch = X_train[b_start:b_start+bs, :], Y_train[b_start:
 ↪b_start+bs, :]

        #descent step
        dtheta = linear_grad(X_batch, theta, Y_batch)
        avg_dtheta += dtheta
        theta -= lr * dtheta

        num_iter += 1

        train_loss.append(linear_loss(X_batch, theta, Y_batch))
        thetas.append(theta.copy())

        #stopping condition
        if num_iter%k == 0:
            avg_dtheta /= k
            if (np.abs(avg_dtheta) < eps).all() or num_iter > max_iter:
                break
            avg_dtheta = 0

    return train_loss, thetas
```

Stopping criteria: number of iteration and $|\frac{\partial J(\theta)}{\partial x}| < \varepsilon$

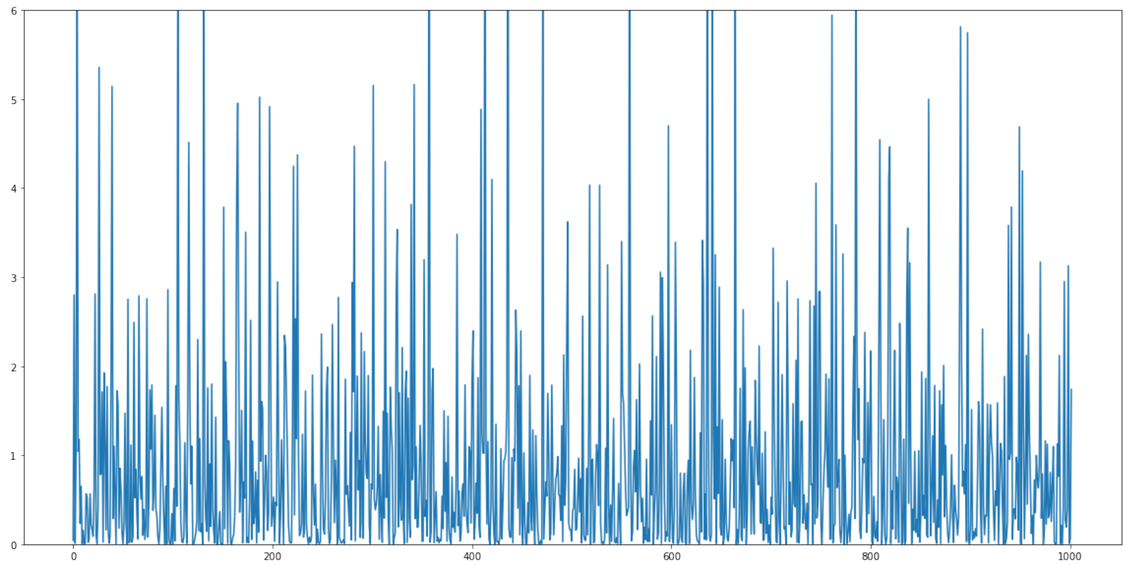
This is computed as an average over k number of batches

[ ]:
```
if num_iter%k == 0:
    avg_dtheta /= k
    if (np.abs(avg_dtheta) < eps).all() or num_iter > max_iter:
        break
    avg_dtheta = 0
```

2.2.1 Loss functions

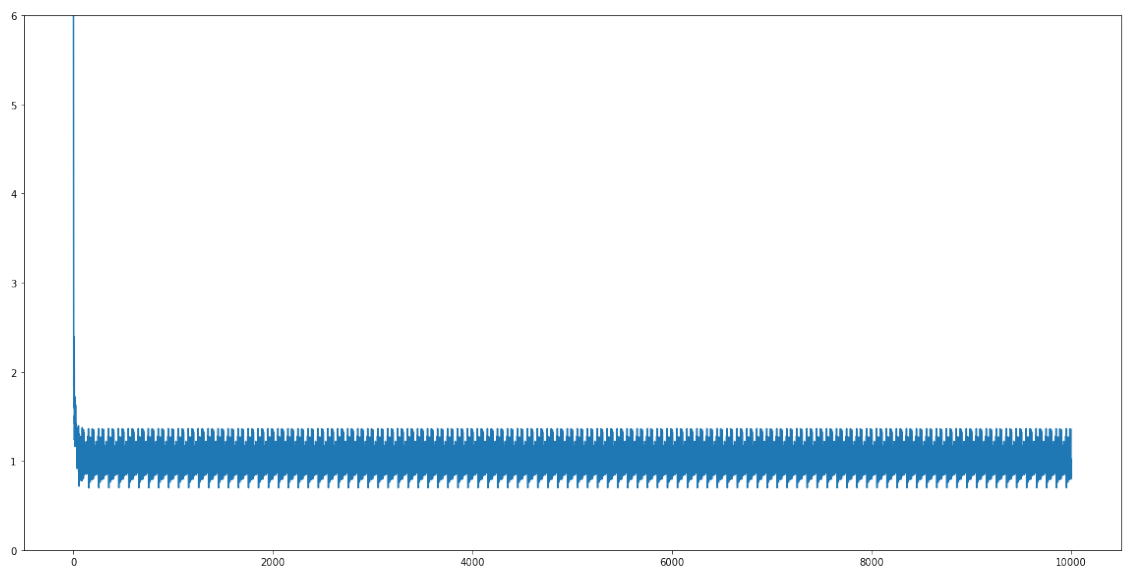
batch size 1

[317]:
```
[<matplotlib.lines.Line2D at 0x7fcb48e0a410>]
```



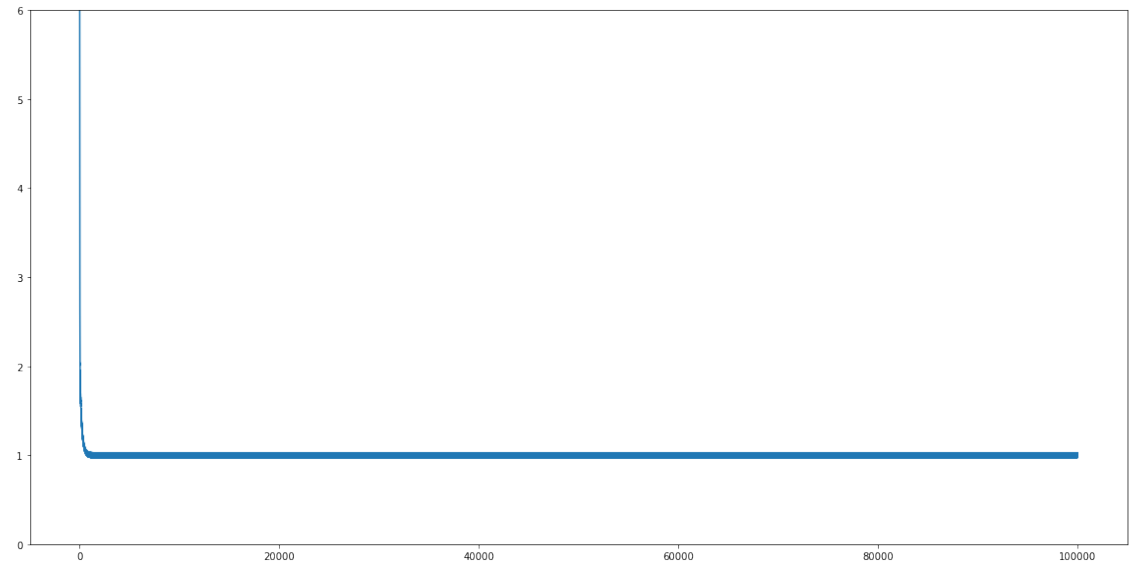
batch size 100

[318]: [<matplotlib.lines.Line2D at 0x7fcbf797e210>]



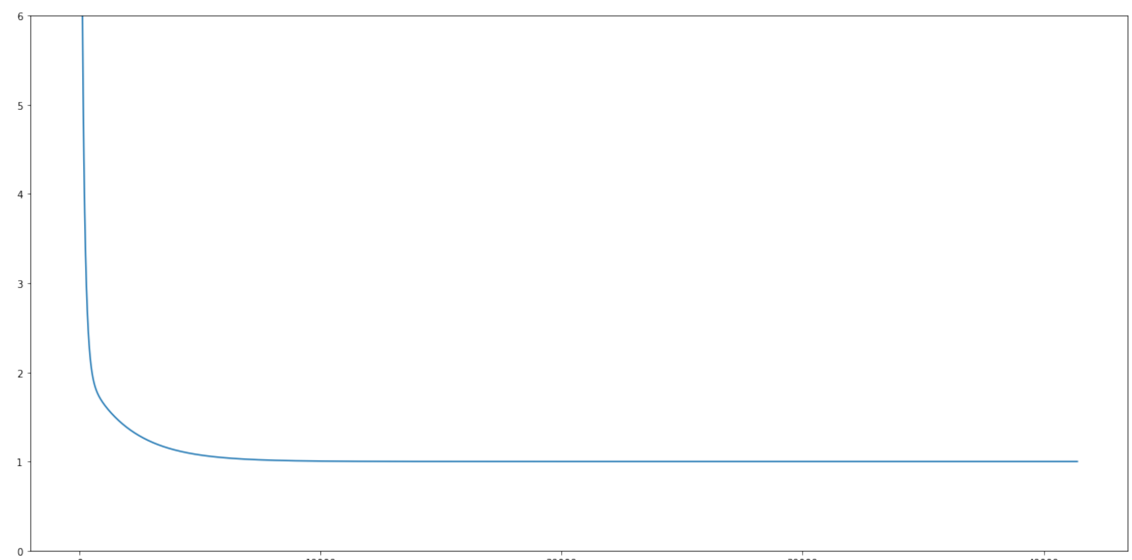
batch size 10000

[319]: [<matplotlib.lines.Line2D at 0x7fcbf79ba050>]



batch size 1000000

[320]: [<matplotlib.lines.Line2D at 0x7fcb97280390>]



Here we can observe that as we increase the batch size the change in the loss function is smooth

2.2.2 θ learned

```
Theta 1 : 2.0268803584334947 0.9603642951124375 2.974903614008454
Theta 2 : 1.9997590154393874 0.9967850847839981 3.0028010634438385
```

```
Theta 3 : 2.0006924504554484 0.9992361704346588 3.003610883799405
Theta 4 : 2.0006612020582857 0.9990746802744249 3.0035452758076504
```

2.3 c. Testing

2.3.1 Observations

Yes, the different algorithms do converge to the same parameters.

[324]:

	theta_2	theta_1	theta_0	l2 error	iterations	timing
0	2.026880	0.960364	2.974904	0.054068	1001001.0	13.360919
1	1.999759	0.996785	3.002801	0.004271	1000101.0	15.486632
2	2.000692	0.999236	3.003611	0.003755	1000011.0	119.535925
3	2.000661	0.999075	3.003545	0.003723	41400.0	207.054515

From the above table it can be seen that the speed for convergence in term of time is faster of smaller batch sizes.

And the number of iterations required is less for bigger batch size but the computation per iteration is large.

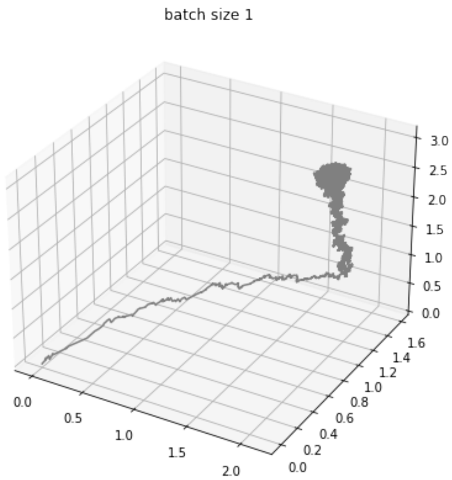
2.3.2 Testing

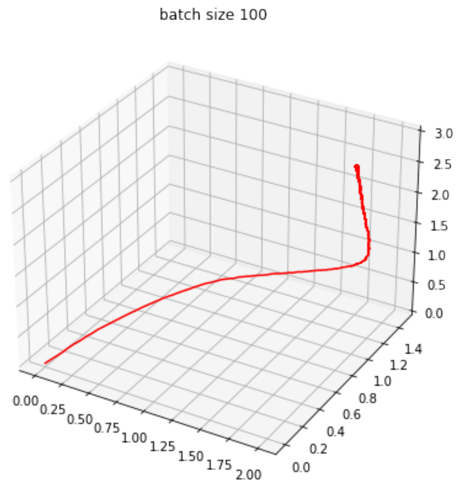
[369]:

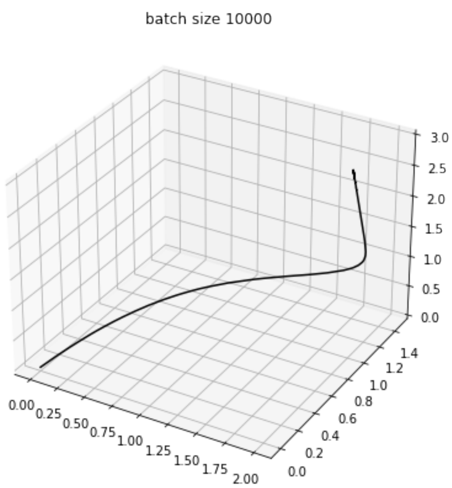
	0	1	2	3	original
l2_error	2.204881	1.967365	1.965865	1.965917	1.965894

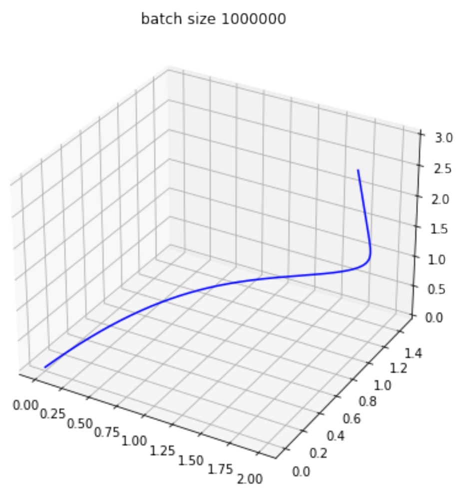
As we can see all the errors are very close to the error of the original hypothesis, the error with batch size 1 is a bit away from the original.

2.4 d. 3d surface









Yes, this makes sense intuitively. As the batch size is smaller the deviation from the original gradient direction should be more. And also when we perform stochastic gradient descent on convex function the parameters are guaranteed to converge within some distance from the minima where this distance is smaller for larger batch size, this can be seen in the above plots.

The batch size 1 converges within a larger region around the minima as compared to others and also each step is more random/ less smooth as compared to others.

3 Logistic regression

Instructions for running code

In the folder Q3/, run the following script:

```
bash run.sh ../../data/q3/logisticX.csv ../../data/q3/logisticY.csv
```

This will run all the sections of question 3

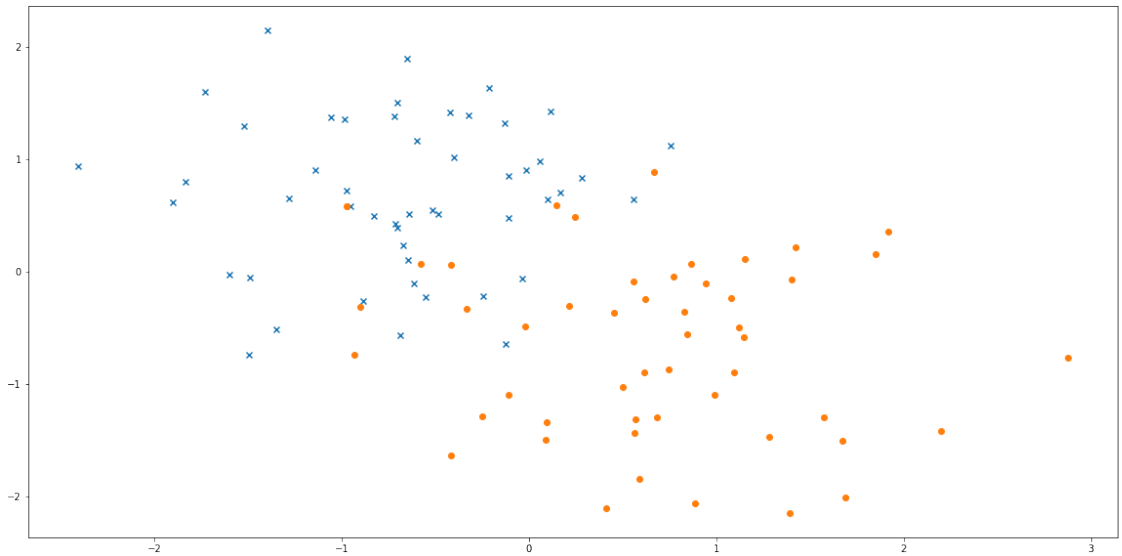
- Reading data and labels
- Normalizing the data
- Inserting intercept

[48]:
```
#reading the input data
X_train = pd.read_csv(f"{data_path}/q3/logisticX.csv", header=None).to_numpy()
Y_train = pd.read_csv(f"{data_path}/q3/logisticY.csv", header=None).to_numpy()

#normalizing the data
X_train = (X_train - X_train.mean(axis=0))/X_train.std(axis=0)
X_train = np.hstack([X_train, np.ones( (X_train.shape[0], 1) )])
```

3.1 a. Implementing Newton's method for descent

Visualizing the data

[49]: <matplotlib.collections.PathCollection at 0x7fca80a36d10>



3.1.1 Important functions

[50]:
```
def sigmoid(z):
    y = np.exp(-z) + 1
    return 1/y
```

[51]:
```
def logistic(X, theta):
    z = linear(X, theta)
    return sigmoid(z)
```

[70]:
```python
def logistic_grad(X, theta, Y):
    Y_hat = logistic(X, theta)
    grad_theta = X.T@(Y - Y_hat)
    return grad_theta
```

[71]:
```python
def logistic_loglikelihood(X, theta, Y):
    Y_hat = logistic(X, theta)
    return np.sum(np.log(np.where(Y, Y_hat, 1 - Y_hat)) )
```

3.1.2 Hessian computation

[91]:
```python
def logistic_hessian(X, theta, Y):
    Y_hat = logistic(X, theta)
    c = -1 * Y_hat * (1 - Y_hat)
    X_tfm = Y_hat * X
    return -X.T @ X_tfm
```

3.1.3 Newton descent

[93]:
```python
def newton_descent(X_train, Y_train, eps=1e-6, max_iter=100):
    theta = np.zeros((3, 1))

    theta_values, loglikelihood = [], []
    theta_values.append(theta.copy())
    loglikelihood.append(logistic_loglikelihood(X_train, theta, Y_train))

    num_iter = 0
    while True:
        #update step
        grad = logistic_grad(X_train, theta, Y_train)
        H = logistic_hessian(X_train, theta, Y_train)
        H_inv = np.linalg.inv(H)
        theta -= H_inv@grad

        theta_values.append(theta.copy())
        loglikelihood.append(logistic_loglikelihood(X_train, theta, Y_train))

        num_iter += 1
        if (np.abs(theta_values[-1] - theta_values[-2]) < eps).all() or␣
↪num_iter > max_iter:
            return loglikelihood, theta_values
```

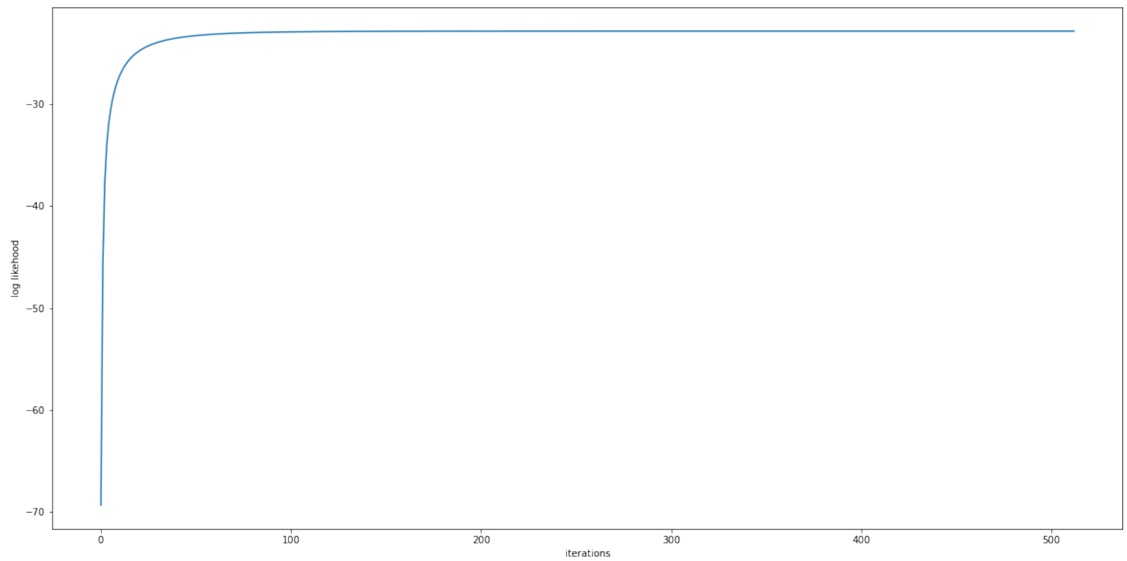
stopping criteria : number of iteration and $|\theta_{t+1} - \theta_t| < \varepsilon$

[ ]:
```python
if (np.abs(theta_values[-1] - theta_values[-2]) < eps).all() or num_iter >␣
↪max_iter:
    return loglikelihood, theta_values
```

3.1.4 Training

[94]:
```
loglikehood, theta_values = newton_descent(X_train, Y_train, eps=1e-5,␣
↪max_iter=1000)
plt.xlabel("iterations")
plt.ylabel("log likehood")
plt.plot(loglikehood)
```

[94]: [<matplotlib.lines.Line2D at 0x7fca50b589d0>]

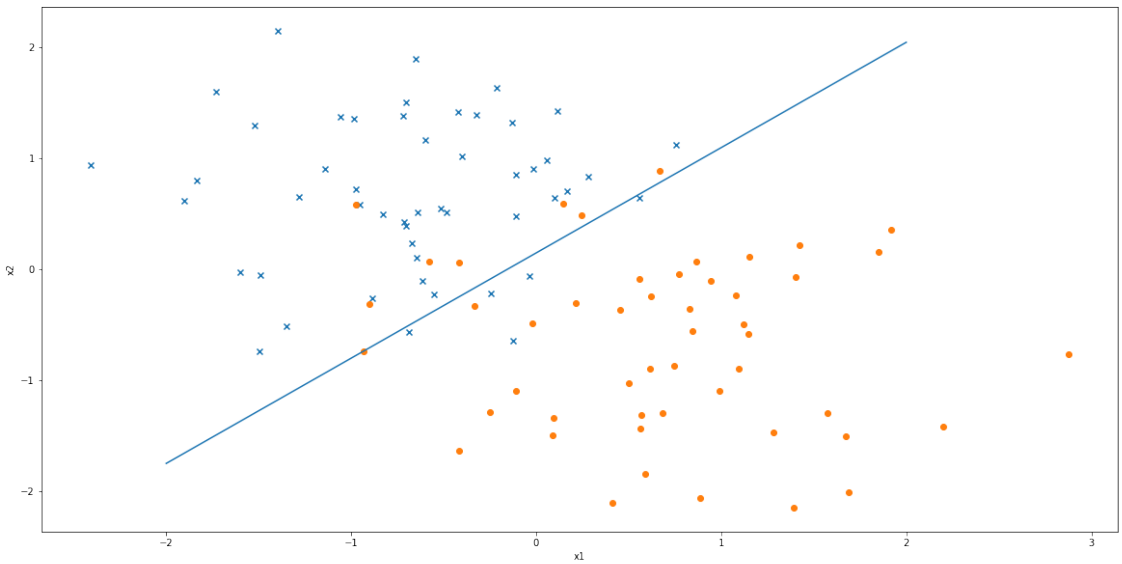


3.1.5 Conclusion

Final parameters : $\theta_2 = 2.58785$, $\theta_1 = -2.7248$, $\theta_0 = 0.40091$

3.2 b. Decision boundary

[96]: <matplotlib.collections.PathCollection at 0x7fca51811fd0>



4 Gaussian Discrmimant Analysis

In the folder Q4/, run the following script:

`bash run.sh ../../data/q4/q4x.dat ../../data/q4/q4y.dat`

This will run all the sections of question 4

- Reading data and labels
- Normalizing the data

[98]:
```
#reading data
X_train = pd.read_csv(f"{data_path}/q4/q4x.dat", header=None, sep="  ",␣
↪engine="python").to_numpy()
Y_train = pd.read_csv(f"{data_path}/q4/q4y.dat", header=None, sep="  ",␣
↪engine="python").to_numpy()
classes = np.unique(Y_train)

#normalizing data
X_train = (X_train - X_train.mean(axis=0))/X_train.std(axis=0)
```

4.1 a. GDA with same co-variance matrix

[99]:
```
c1_pos = np.where(Y_train == classes[0])[0]
c2_pos = np.where(Y_train == classes[1])[0]

X_c1 = X_train[c1_pos]
X_c2 = X_train[c2_pos]
```

```
n = X_train.shape[0]
```

[100]:
```
phi = c1_pos.shape[0]/n

mu_1 = np.mean(X_c1, axis=0).reshape(-1, 1)
mu_2 = np.mean(X_c2, axis=0).reshape(-1, 1)

X_c1_centered = X_c1 - mu_1.T
X_c2_centered = X_c2 - mu_2.T
S = (X_c1_centered.T@X_c1_centered + X_c2_centered.T@X_c2_centered)/n
```

4.1.1 Conclusion

$\phi = 0.5$

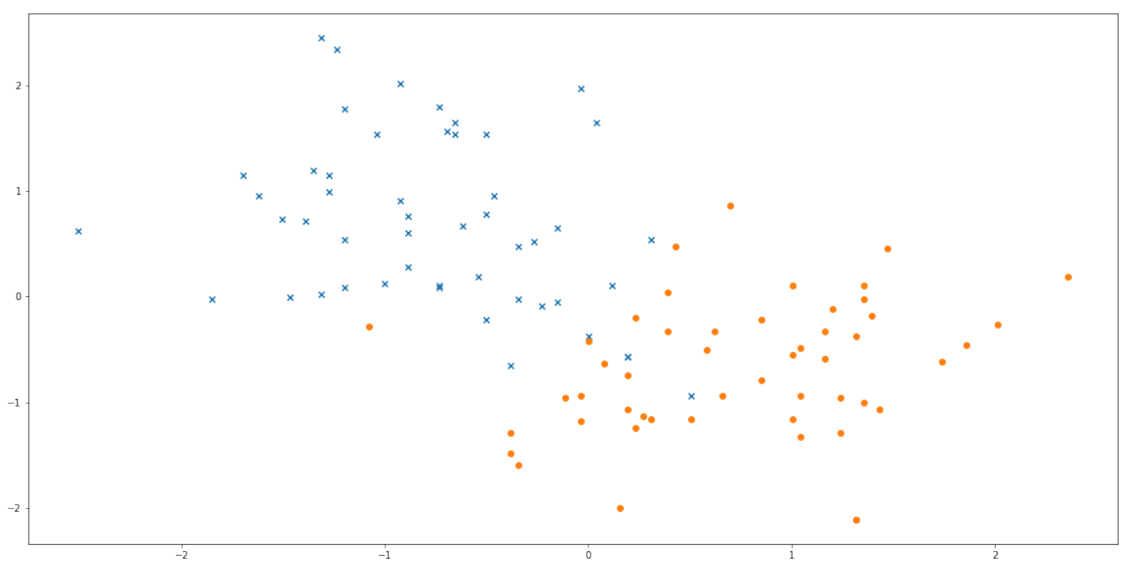
$\mu_0 = (-0.75529433, 0.68509431)$

$\mu_1 = (0.75529433, -0.68509431)$

$$\Sigma = \begin{bmatrix} 0.42953048 & -0.02247228 \\ -0.02247228 & 0.53064579 \end{bmatrix}$$

4.2 b. Visualizing the data

[103]: <matplotlib.collections.PathCollection at 0x7fca51885d10>

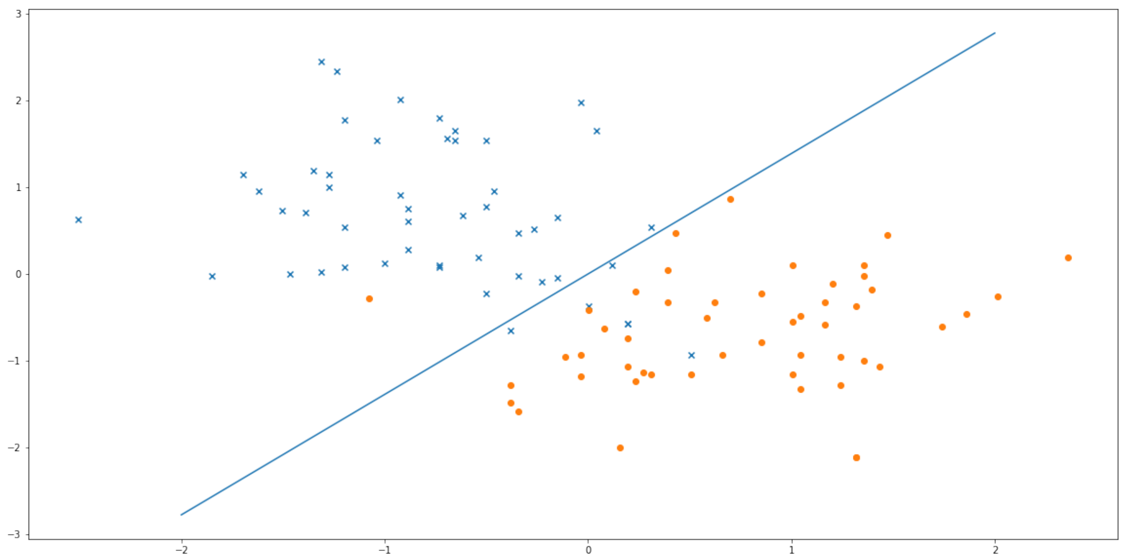


4.3 c. Decision boundary

The equation of the decision boundary:

$$2x^T\Sigma^{-1}(\mu_0 - \mu_1) + \mu_1^T\Sigma^{-1}\mu_1 - \mu_0^T\Sigma^{-1}\mu_0 + 2\log(\frac{1-\phi}{\phi}) = 0$$

[106]: <matplotlib.collections.PathCollection at 0x7fca80963ed0>



4.4 d. GDA with different co-variance matrix

[107]:
```
S_1 = (X_c1_centered.T@X_c1_centered)/X_c1_centered.shape[0]
S_2 = (X_c2_centered.T@X_c2_centered)/X_c2_centered.shape[0]
```

$\phi = 0.5$

$\mu_0 = (-0.75529433, 0.68509431)$

$\mu_1 = (0.75529433, -0.68509431)$

$\Sigma_0 = \begin{bmatrix} 0.38158978 & -0.15486516 \\ -0.15486516 & 0.64773717 \end{bmatrix}$

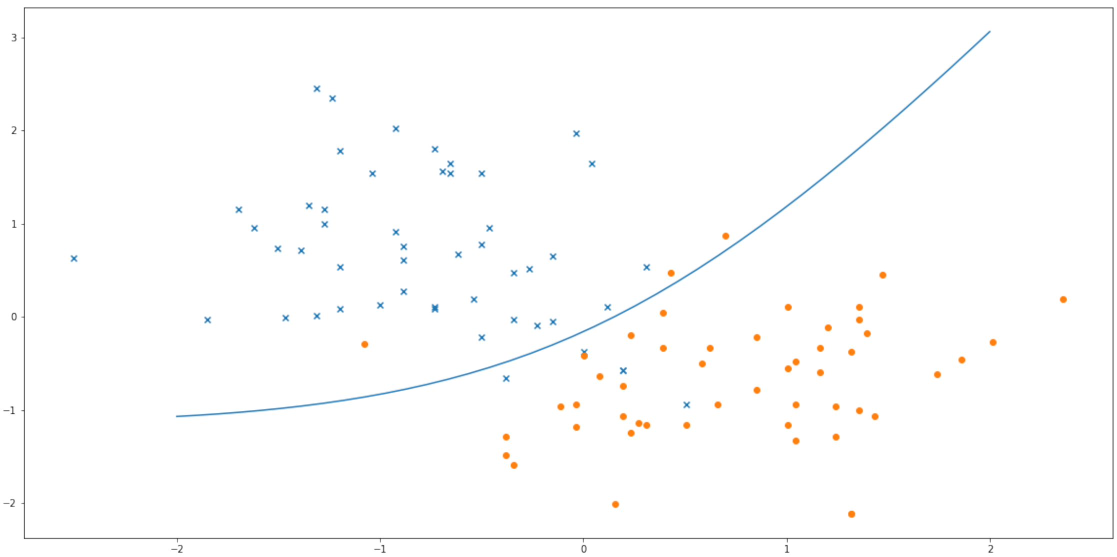
$\Sigma_1 = \begin{bmatrix} 0.47747117 & 0.1099206 \\ 0.1099206 & 0.41355441 \end{bmatrix}$

4.5 e. Decision boundary

The equation of the decision boundary:

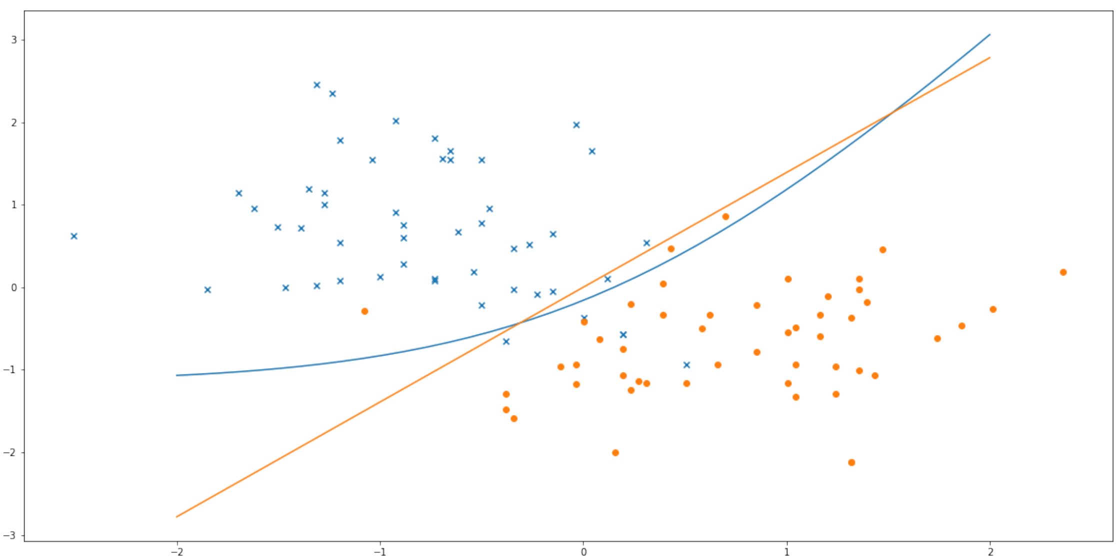
$$x^T(\Sigma_1^{-1} - \Sigma_0^{-1})x - 2x^T(\Sigma_1^{-1}\mu_1 - \Sigma_0^{-1}\mu_0) + \mu_1^T\Sigma^{-1}\mu_1 - \mu_0^T\Sigma^{-1}\mu_0 + 2\log(\frac{1-\phi}{\phi}) + \log(\frac{|\Sigma_1|}{|\Sigma_0|}) = 0 \quad (1)$$

[121]: <matplotlib.collections.PathCollection at 0x7fca518a5fd0>



4.6 f. Analyzing the boundary

[123]: <matplotlib.collections.PathCollection at 0x7fca529b6dd0>



It can be observed that the quadratic boundary is curving towards the `Alaska` class.

It can be seen in the image that the quadratic seperator is overfitting the data. The two classes seem to be linearly seperable and linear separator is doing a good job whereas the quadratic seperator is curving a lot towards class `Alaska`.