INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

MNIST Problem and ML Models

Author : Deeksha Rawat
Project Instructor : Dr. Thirumalanathan D.

July 7, 2023

1 WHAT IS MNIST PROBLEM?

MNIST database stands for **Modified National Institute of Standards and Technology** database. It is a large collection of handwritten digits. The MNIST dataset consists of 60,000 training images and 10,000 testing images, all of which are grayscale images of size 28x28 pixels. Each pixel has an intensity ranging from 0 (white) to 255 (black). Each image corresponds to a handwritten digit from 0 to 9. The goal of the MNIST problem is to develop machine learning models that can accurately classify these images into their corresponding digit classes.

```
[5] from keras.datasets import mnist
    import matplotlib.pyplot as plt
    (X_train,Y_train),(X_test,Y_test) = mnist.load_data()
    row = 2
    column = 2
    fig = plt.figure(figsize=(8, 8))
    for i in range(1,row*column+1,1):
      fig.add_subplot(row,column,i)
      plt.imshow(X_train[i], cmap=plt.get_cmap('gray'))
    plt.show()
```

Figure 1.1: Code Snippet to print the training example

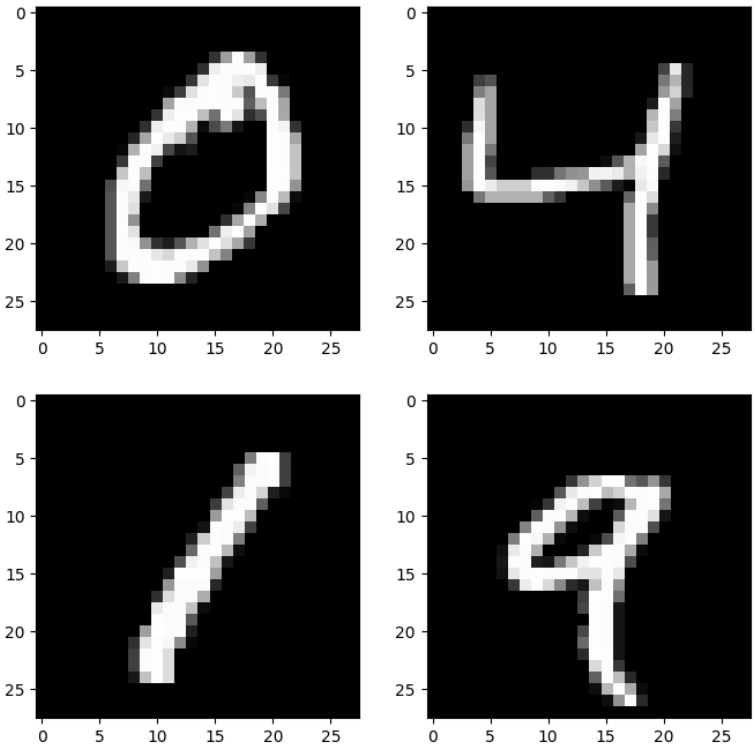


Figure 1.2: Handwritten Digits

1.1 How to Approach the MNIST Problem ?

To approach the MNIST problem, we can follow the below mentioned general steps:

Dataset Preparation : Obtain the MNIST dataset, which is easily available and widely used. It consists of a set of labeled images for training and testing.

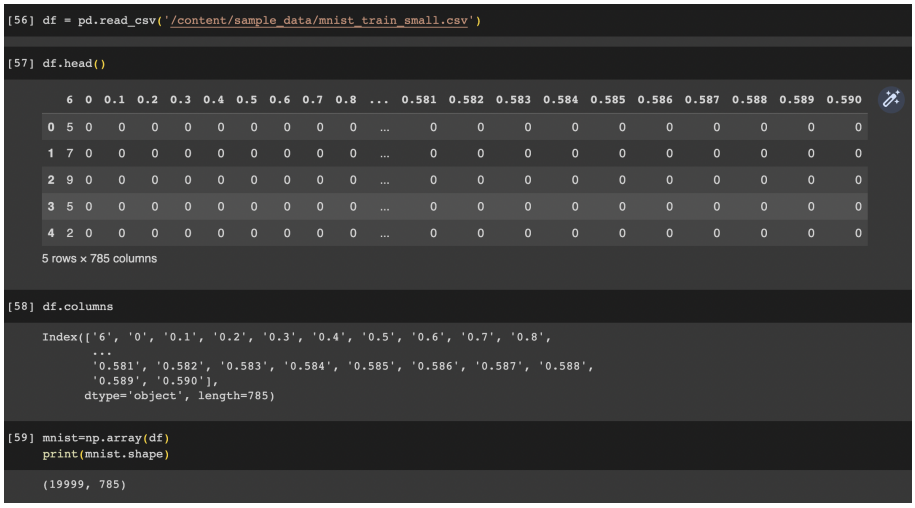


Figure 1.3: Data Preparation

Data Preprocessing : Before training a model, it's important to preprocess the data. Common preprocessing steps include normalizing pixel values, reshaping the images, and potentially applying other techniques like data augmentation to increase the size and diversity of the training set.

Model Selection : Choose an appropriate model architecture for the MNIST problem. Convolutional Neural Networks (CNNs) are commonly used for image classification tasks and have shown excellent performance on MNIST. We can also consider other models such as deep neural networks or traditional machine learning algorithms.

Figure 1.4: Model Selection and Training

Model Training : Split the labeled training data into training and validation sets. Feed the training data into your chosen model and train it using optimization techniques like stochastic gradient descent (SGD) or Adam. Monitor the performance on the validation set and tune hyperparameters such as learning rate, batch size, or network architecture accordingly.

Model Evaluation : Once the model is trained, evaluate its performance on the labeled testing data. Calculate metrics such as accuracy, precision, recall, and F1 score to assess how well the model performs on unseen data. Avoid overfitting by not tuning the model based on the testing data.

```
from sklearn.metrics import accuracy_score
ac_score = accuracy_score(y_test,y_pred)
ac_score
```

```
0.9491949194919492
```

```
print(sum(y_test!=y_pred))
```

```
508
```

Figure 1.5: Model Evaluation

Model Fine-Tuning : If the model's performance is not satisfactory, we can iterate and make improvements. This may involve adjusting hyperparameters, changing the model architecture, or exploring more advanced techniques like regularization.

Prediction : After the model's performance is satisfactory, we can use it to predict the classes of new, unseen images. Providing the test images to the trained model, and it will output the predicted digit labels.

The training example with its pixel intensity is visualised as :

```
[[  0   0   0   0   0  28  59  50   0  23   0   0  32 134 180 254 206   8
    0   0   0   0]
 [  0   0   4  96 216 233 254 248 215 231 215 215 236 254 250 181  27   0
    0   0   0   0]
 [  0   0 108 254 254 247 175 175 175 176 175 175 205 175  60   0   0   0
    0   0   0   0]
 [  0   0  47 254 245  85   0   0   0   0   0   0   8   0   0   0   0   0
    0   0   0   0]
 [  0   0 152 254 158   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0]
 [  0  19 240 255  38   0  41  50   0   0   0   0   0   0   0   0   0   0
    0   0   0   0]
 [  0  87 254 254 178 215 242 248 215  96  19   0   0   0   0   0   0   0
    0   0   0   0]
 [  0 176 254 254 254 217 175 187 254 254 248  85   0   0   0   0   0   0
    0   0   0   0]
 [  0 161 247 214  57  11   0   3  19 177 248 248 129   9   0   0   0   0
    0   0   0   0]
 [  0  18  49   0   0   0   0   0   0   0  57 224 254 171   0   0   0   0
    0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0 213 255 122   0   0   0
    0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0  92 254 196   0   0   0
    0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0  40 254 196   0   0   0
    0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0 145 254 196   0   0   0
    0   0   0   0]
 [  0  31 188  45   0   0   0   0   0   0   0  99 249 254 121   0   0   0
    0   0   0   0]
 [  0 139 245  45   0   0   0   0   0   0 140 254 254 133   0   0   0   0
    0   0   0   0]
 [ 17 242 169   0   0   0   0   4  58 216 248 254 167   9   0   0   0   0
    0   0   0   0]
 [ 14 230 196  79  49  79  79 181 254 254 247 108   6   0   0   0   0   0
    0   0   0   0]
 [  0  44 213 254 247 254 254 254 254 192  32   0   0   0   0   0   0   0
    0   0   0   0]
 [  0   0   9 133 156 193 155 140  58   0   0   0   0   0   0   0   0   0
    0   0   0   0]]
```

Figure 1.6: Digit 5 shown with the pixel intensity

2 Convolutional Neural Network

A Convolutional Neural Network (CNN) is a deep learning model that is primarily used for analyzing visual data, such as images or videos. CNNs are designed to automatically and adaptively learn hierarchical representations from input data, making them highly effective for tasks like **image classification, object detection, and image segmentation.**
It consists of 3 layers -

- Convolutional Layer

- Pooling Layer
- Fully Connected Layer

Before going into this, let's understand a single layer neural network.
Single Layer Neural Network

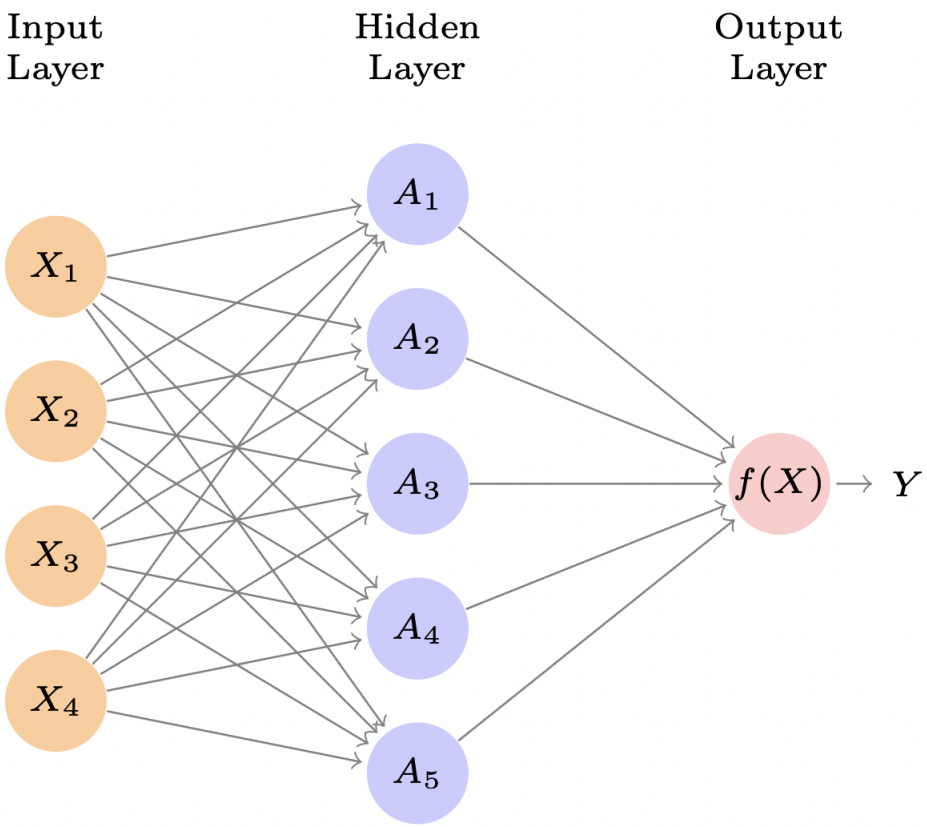


Figure 2.1: Single Layer Neural Network(Source:https://www.statlearning.com/resources-second-edition)

$$f(X) = \beta_0 + \Sigma_{k=1}^{K} \beta_k h_k(X)$$
$$f(X) = \beta_0 + \Sigma_{k=1}^{K} \beta_k g(w_{k_0} + \Sigma_{j=1}^{p} w_{k_j} X_j)$$

$A_k = h_k(X) = g(w_{k_0} + \Sigma_{j=1}^{p} w_{k_j} X_j)$ are called **Activations** in the hidden layer.

What are Activation Functions?
These are the functions used to get the output of node. These are also called **Transfer Function.**These are used to determine the output of neural network like - YES or NO by mapping

the values in between 0 to 1 or -1 to 1, depending upon the function.
Different Types of Activation Functions

- ReLU(Rectified Linear Activation)- The main advantage of using the ReLU function over other activation functions is that it does not activate all the neurons at the same time.This means that the neurons will only be deactivated if the output of the linear transformation is less than 0.

$$f(x) = \mathbf{max}(0, x), x \in \mathbb{R}$$

- Sigmoid function - It is one of the most widely used non-linear activation function. Sigmoid transforms the values between the range 0 and 1.

$$f(x) = \frac{1}{1 + e^{-x}}$$

- Hyperbolic Tanget Activation function(tanh) - It is symmetric around the origin. The range of values is from -1 to 1. Thus the inputs to the next layers will not always be of the same sign.

$$f(x) = \frac{2}{1 + e^{-2x}} - 1$$

- Softmax function - This is the normalised exponential function.

$$\sigma : \mathbb{R} \xrightarrow{(0,1)^K}$$

$$\sigma\left(\mathbb{Z} = \frac{e^{\mathbb{Z}_i}}{\Sigma_{j=1}^{K} e^{\mathbb{Z}}}\right) \text{for i = 1,..,K}$$

$$\mathbb{Z} = (\mathbb{Z}_1, \mathbb{Z}_2, ..., \mathbb{Z}_K) \in \mathbb{R}^K$$

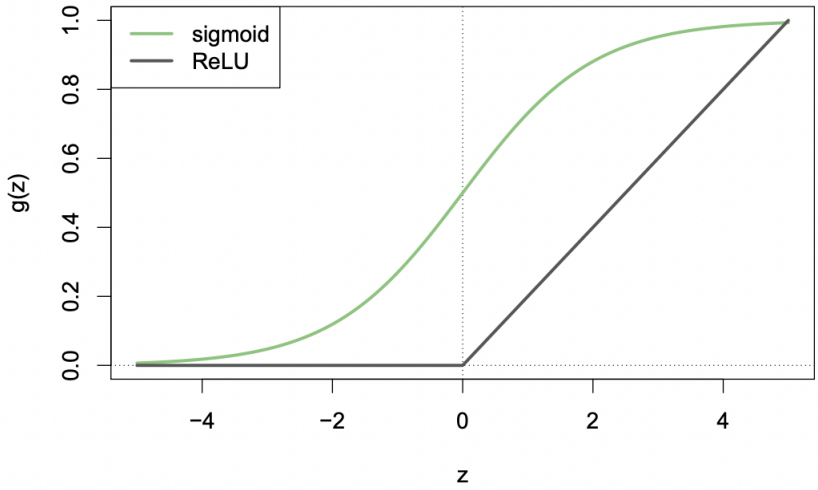


Figure 2.2: ReLU and Sigmoid function (Source:https://www.statlearning.com/resources-second-edition)

For the MNIST problem, we build a 2 layer network : - First Layer has 256 units
- Second Layer has 128 units
- Output Layer has 10 units
The model is fit by minimising the $\Sigma_{i=1}^{n}(y_i - f(x_i))^2$.

Details of the Output Layer

$Z_m = \beta_{m_0} + \Sigma_{l=1}^{K_2}\beta_{m_l}A_l^{(2)}, m = 0,1,2,...,9$ be 10 linear combinations of the activations at the second layer.

Output activation function encodes the **Softmax** function.

$$f_m(X) = Pr(Y = m|X) = \frac{e^{Z_m}}{\Sigma_{l=0}^{9}e^{Z_l}}$$

The model is fit by minimising the negative multinomial log-likelihood (or cross-entropy), i.e.,

$$-\Sigma i = 1n\Sigma m = 0^9 y_{im} log(f_m(x_i))$$

y_{im} is 1 if true class for the observation i is m, else 0.

CNN build up an image in a hierarchical fashion. This hierarchical construction is achieved using **convolution** and **pooling** layers.

Convolution Filter

Input Image = $\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \\ j & k & l \end{bmatrix}$ Convolution Filter = $\begin{bmatrix} \alpha & \beta \\ \gamma & \delta \end{bmatrix}$

Convolved Image = $\begin{bmatrix} \alpha a + \beta b + \gamma d + \delta e & \alpha b + \beta c + \gamma e + \delta f \\ \alpha d + \beta e + \gamma g + \delta h & \alpha e + \beta f + \gamma h + \delta i \\ \alpha g + \beta h + \gamma j + \delta k & \alpha h + \beta i + \gamma k + \delta l \end{bmatrix}$

This is slid across the image, scoring for matches. This scoring is done through dot product. If the subimage of the input image is similar to the filter, the score is high, otherwise low. The filters are learned during training.

Pooling

It sharpens the feature identification. The size of the pooling operation/filter is almost always 2x2 pixels.

Common fuctions used in the pooling operation are:

- Average Pooling - Calculate the average value for each patch on the feature map.
- Maximum Pooling - Calculate the maximum value for each patch of the feature map.
  Example of Max Pooling -
  max pool $\begin{bmatrix} 1 & 2 & 5 & 3 \\ 3 & 0 & 1 & 2 \\ 2 & 1 & 3 & 4 \\ 1 & 1 & 2 & 0 \end{bmatrix}$ gives the result as $\begin{bmatrix} 3 & 5 \\ 2 & 4 \end{bmatrix}$

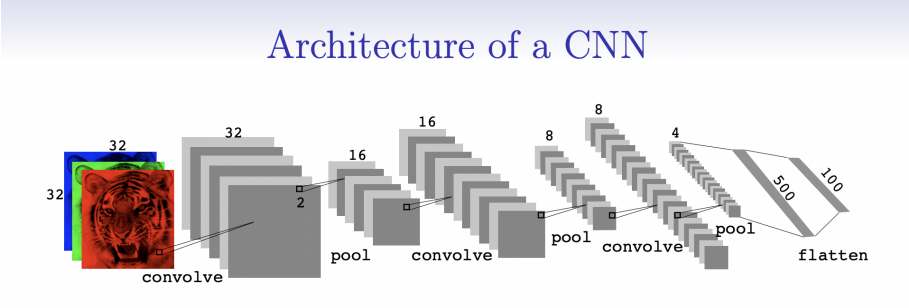


Figure 2.3: Source:https://www.statlearning.com/resources-second-edition

3 Understanding the Optimisation Algorithms

3.1 Gradient Descent Algorithm

Gradient Descent is an iterative optimization algorithm commonly used in machine learning and deep learning for finding the minimum of a function, typically the loss function in the context of training a model.
What is the need of Gradient Descent? Although we can directly use calculus to differentiate the Function and find the optimal solution. However, not all the equations have closed form solution. Therefore, gradient descent algorithm is used to find the minima of the function.
For example :

$$xe^x - 1 = 0$$

The general idea behind Gradient Descent is to iteratively update the parameters of a model in the direction of steepest descent of the loss function. The algorithm calculates the gradient of the loss function with respect to the parameters and takes steps proportional to the negative of the gradient to update the parameter values. The goal is to find the set of parameter values that minimize the loss function.
Let **F(x)** be a continuous and differentiable function. To find the minima of F(x),follow the following steps -

- Start with $x^{(0)}$.
- Update x as shown below

$$x^{(k+1)} = x^{(k)} - \alpha_k \nabla F(x^k)$$

In Machine Learning, F is the Cost Function. The task is to minimise the cost and gain the parameters at that minimum value of F. α is the *Learning Rate*.

3.1.1 What is Learning Rate?

In machine learning, we deal with two types of parameters :
1) machine learnable parameters

2) hyper-parameters.
The Machine learnable parameters are the one which the algorithms learn/estimate on their own during the training for a given dataset.
The Hyper-parameters are the one which the machine learning engineers or data scientists will assign specific values to, to control the way the algorithms learn and also to tune the performance of the model. Learning rate is a hyper-parameter used to control the rate at which an algorithm updates the parameter estimates or learns the values of the parameters.
If the learning rate is too large, then the steps taken are too big and it may not reach the local minimum because it bounces back and forth between the convex function of gradient descent (**Overshooting**).
If the learning rate is too small, then it might take more time to attain the minima.
Lipschitz Continuity : f is Lipschitz continuous if there exists an $L \geq 0$ such that

$$|f(x) - f(y)| \leq L\|x - y\|$$

If f is convex and Δf is Lipschitz continuous, then the gradient descent algorithm converges to x^k with $\alpha_k = \frac{1}{L}$.

3.1.2 Implementation of Gradient Descent Algorithm

```
def grad_des(xo , l_r, tol):
  global x
  global itr
  global i
  x_upd0 = 100000
  x_upd1 = xo
  while abs(x_upd1 - x_upd0)>=tol:
    x.append(x_upd1)
    x_upd0 = x_upd1
    x_upd1 = x_upd0 - (l_r)*(exp.subs(X,x_upd0))
    i += 1
    itr.append(i)
  return x
```

Figure 3.1: Gradient Descent Algorithm

The minima of x^2 is found using gradient descent method with the initial value $x_o = 5$ and the learning rate, $\alpha = 0.1$ and a tolerance of 0.00001.

```
import matplotlib.pyplot as plt
# Plot till the convergence point
plt.plot(x,y,'black',marker='o',mfc='yellow')
plt.xlabel('X Values')
plt.ylabel('Y Values')
```

```
Text(0, 0.5, 'Y Values')
```

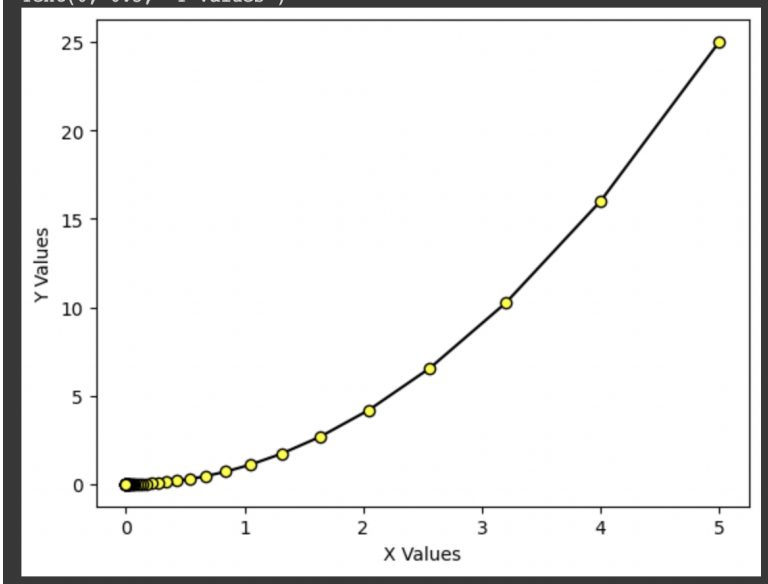


Figure 3.2: Convergence towards Minima

3.2 Newton-Raphson Method

Newton-Raphson method is a powerful numerical technique for finding the roots of non-linear equations. It is comparatively faster than Gradient Descent. However, the convergence is highly determined by the initial value. The update equation is given as

$$x^{k+1} = x^k - \frac{f^{'}(x^{(k)})}{f^{''}(x^{(k)})}$$

For multidimensional data, the update equation is

$$x^{(k+1)} = x^{(k)} - [\nabla^2 f(x^{(k)})]^{-1}(\nabla f(x^{(k)}))$$

3.3 Stochastic Gradient Descent

SGD is a widely used optimization algorithm in machine learning, particularly in scenarios with large datasets. It enables efficient parameter updates while still aiming to minimize the loss function and improve the model's performance.
For gradient descent, we have to compute $\nabla f(x^{(k)})$ for each iteration which can be time-

consuming.
If $f(x) = \Sigma_{i=1}^{n} f_i(x)$, we approximate $\nabla f(x) \approx \Sigma_{i \in S_n} f_i(x)$, S_n is the subset chosen at random.
Let $g(x) = \Delta f(x) + \epsilon$, with $E(\epsilon) = 0$.
The SGD algorithm works as :

- Choose initial point $x^{(0)}$.
- While $|\Delta f(x^{(k)})| \leq tolerance$, randomly pick $g(x^{(k)})$.
- The update equation is : $x^{(k+1)} = x^{(k)} - \gamma_k g(x^{(k)})$.

$$g(x) = \Sigma_{i \in S} \Delta f_i(x)$$

The set S is randomly picked from 1,2,3,....,n. |S| = n is the batch-size.

3.3.1 Implementation of Stochastic Gradient Descent Algorithm

Consider minimising the following objective function with respect to β_1, β_2 :

$$\Sigma_{i=1}^{200}(1 - Y_i(\beta_0 + \beta_1 X_i^{(1)} + \beta_2 X_i^{(2)}))_+ + \frac{1}{2C}(\beta_1^2 + \beta_2^2)$$

where (X_i, Y_i) are generated as follows:
200 data points $X_1, X_2, ..., X_{200} \in \mathbb{R}^2$ i.i.d. $\sim \mathcal{N}(0,1)$. Decide Y_i as follows :

$$\begin{cases} +1 & X_i^{(1)} + X_i^{(2)} > 0 \\ -1 & otherwise \end{cases}$$

Choose C = 10. Let $\beta^{(0)} = [\beta_0, 0, 0]$, and $g(\beta) = \frac{|S|}{200} \cdot \frac{1}{C}[\beta_1, \beta_2] + \Sigma_{i \in S} \Delta f_i(x)$, where β_0 is the value of intercept of the SVM classifier with linear kernel for the features $(X_i^{(1)}, X_i^{(2)})$. And

$$\nabla f_i(\beta) = \begin{cases} -Y_i[X_i^{(1)}, X_i^{(2)}] & \text{if}(Y_i(\beta_0 + \beta_1 X_i^{(1)} + \beta_2 X_i^{(1)})) \leq 1 \\ 0 & \text{otherwise} \end{cases} \tag{3.1}$$

Fix $|S| = 5$. Apply the stochastic gradient descent algorithm with constant learning rate. Choose an appropriate value of the learning rate γ_0. Print the number of iterations needed to get to $||\beta^{(k)} - \beta^*|| \leq 10^{-4}$. Plot the trajectory of $\beta^{(k)}$.

- Generating the required data points.

```
[ ]  np.random.seed(1)
     X = np.random.multivariate_normal([0,0],[[1,0],[0,1]], 200)
     Y = np.zeros(200)

     for i in range(200):
       if X[i][0]+X[i][1] > 0:
         Y[i] = 1
       else :
         Y[i] = -1
```

Figure 3.3: Code to generate Data Points

- Visualising the data points.

```
plt.figure()
plt.title("Examples")
plt.plot(X[Y == 1,0],X[Y == 1,1],'go')
plt.plot(X[Y == -1,0],X[Y == -1,1],'ro')
plt.xlabel('X1')
plt.ylabel('X2')
plt.show()
```

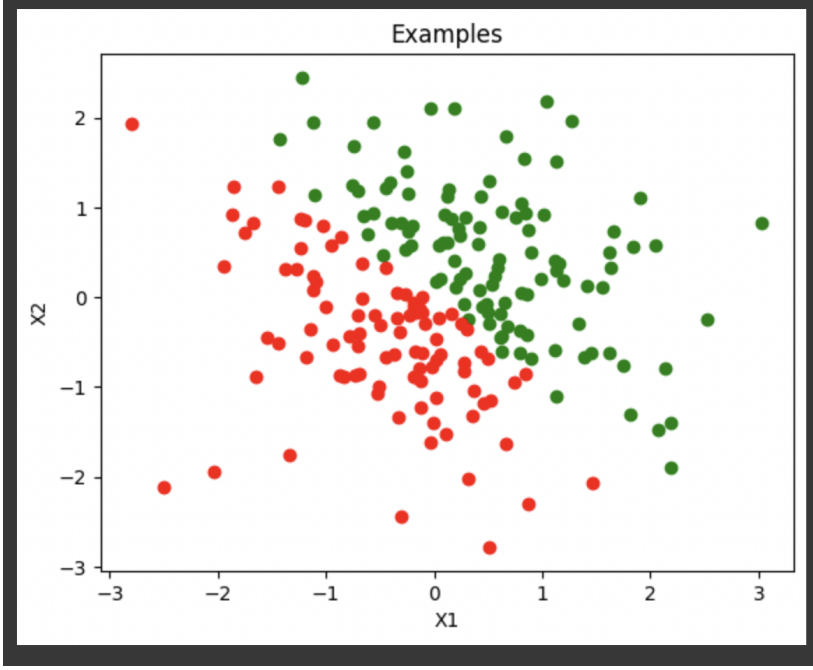


Figure 3.4: Visualisation of the Generated Data Points

- Building a SVM linear classifier on the data points.

```
from sklearn.svm import SVC
clf = SVC(C = 10, kernel = 'linear')
clf.fit(X,Y)
```

```
SVC(C=10, kernel='linear')
```

```
w = clf.coef_[0]
w
```

```
array([5.64930167, 5.69023307])
```

```
bias = clf.intercept_
bias[0]
```

```
0.027149252631840 51
```

Figure 3.5: Code to Build SVM Linear Classifier

Therefore, the value of $\beta_0 \approx 0.0271$. So, $\beta^{(0)} = [\beta_0, 0, 0] = [0.0271, 0, 0]$.

- Visualising the Classifier Line

```
plt.figure(figsize=(8,8))
plt.title("Classifier Line using SVM")
plt.scatter(X[Y == 1,0],X[Y == 1,1],color='green',label='Y = 1',s=20)
plt.scatter(X[Y == -1,0],X[Y == -1,1],color='red',label='Y = -1',s=20)
plt.xlabel('X1')
plt.ylabel('X2')

x1 = np.linspace(-3,3) #we get the equation of hyperplane w[0]x1 + w[1]x2 + c = 0
x2 = -(w[0]/w[1])*x1 - bias[0]/w[1]
plt.plot(x1, x2, 'k-', label = 'Decision Boundary')
plt.legend()
plt.show()
```

Figure 3.6: The Linear Classifier

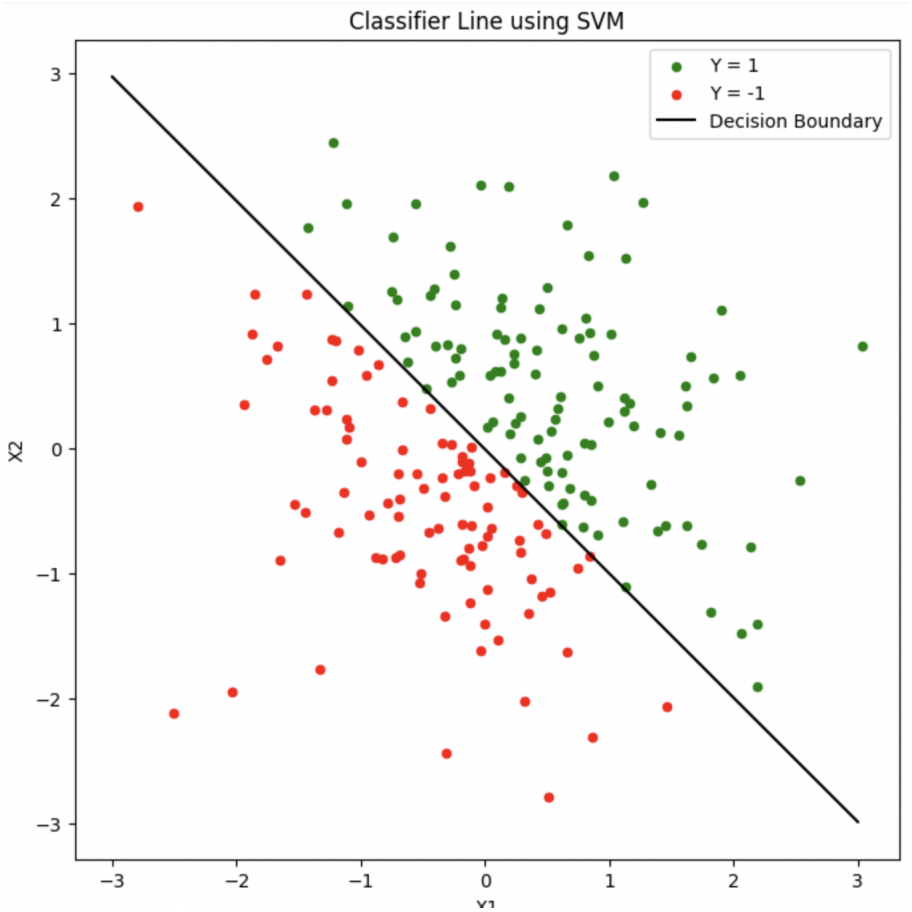


Figure 3.7: SVM Linear Classifier

- Finding the optimum solution of β

```
beta_star = np.array((bias[0],w[0],w[1]))
beta = np.array((bias[0],0,0))
print("beta* : ",beta_star)
print("beta : ",beta)
```

```
beta* :  [0.02714925 5.64930167 5.69023307]
beta :  [0.02714925 0.         0.        ]
```

```
def grad_fi(Xi,Yi,beta):
  if (Yi*(Xi.dot(beta)))<=1:
    return -Yi*(Xi[1:])
  else :
    return 0*(Xi[1:])
```

```
] def SGD(X,Y,beta_, beta_star,C,tol,Sn,gamma):#,max_iter=10000):
    X_ = np.concatenate((np.ones((X.shape[0], 1)), X), axis=1)
    beta = [beta_]
    error = [0]
    error[0] = np.linalg.norm(beta[0]-beta_star)
    k = 0
    g = 0
    while (error[k] > tol):# and k<max_iter):
      g = (Sn/(200*C))*beta[k][1:]
      S = np.random.choice(X.shape[0], Sn)
      for i in S:
        f = grad_fi(X_[i],Y[i],beta[k])
        g+=f
      beta_new = np.concatenate((beta[k][0:1], beta[k][1:] - gamma * g))
      beta.append(beta_new)
      k+=1
      error.append(np.linalg.norm(beta_new - beta_star))
    return beta,error
```

Figure 3.8: Code to find the minima

```
beta_sgd_5, err_sgd_5 = SGD(X, Y, beta, beta_star, C = 10, tol=0.01, Sn = 5, gamma=0.01)
print(beta_sgd_5[-1])
print('Number of Iterations = ',len(beta_sgd_5))
```

```
[0.02714925 5.64495371 5.68136196]
Number of Iterations =  37432
```

Figure 3.9: Value of β from SGD

We get the value $\beta = [0.02714925, 5.64495371, 5.68136196]$ in 37432 iterations.

- Trajectory of β_1 and β_2 as a 2D-plot.

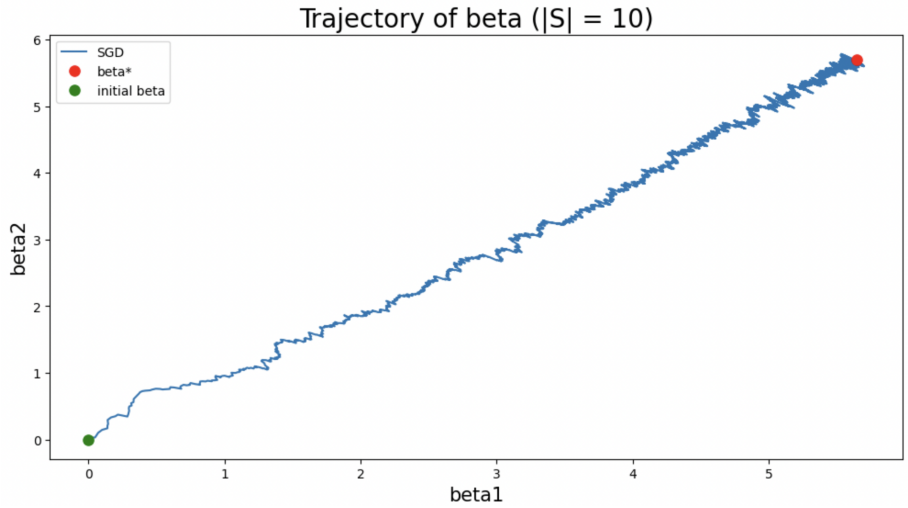


Figure 3.10: 2D plot showing the variation of β

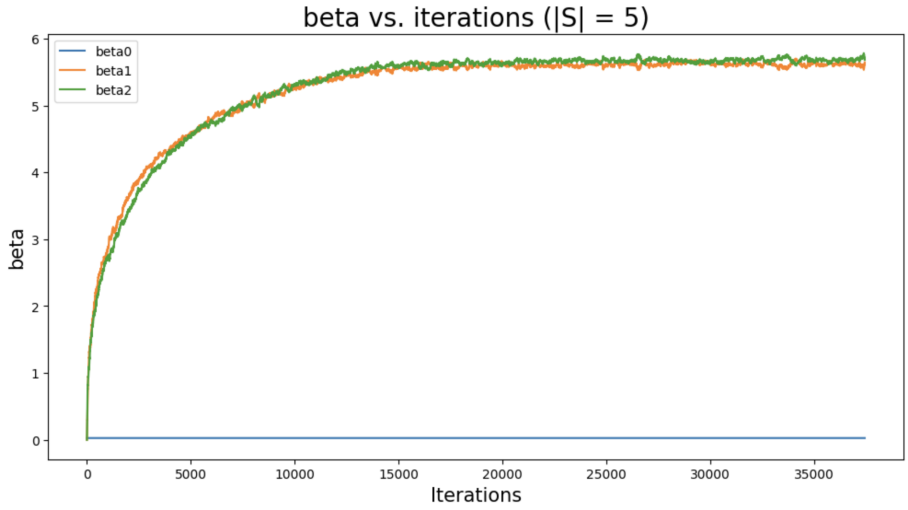


Figure 3.11: β vs No. of Iterations

Choosing the Learning Parameter wisely can reduce the number of iterations and smoothen out the trajectory of β.

- For $\gamma = 0.001$, the no. of iterations = 244980. However, we obtain a less noisy trajectory of β

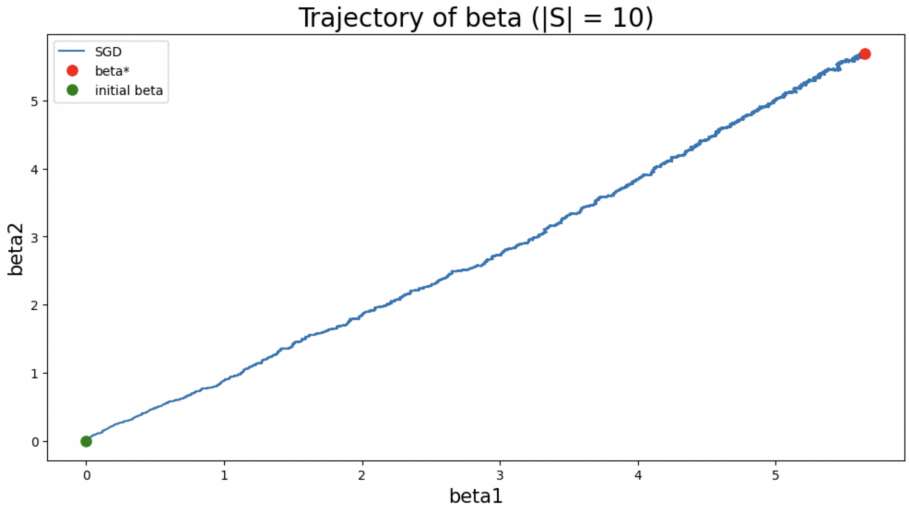


Figure 3.12: For $\gamma = 0.001$

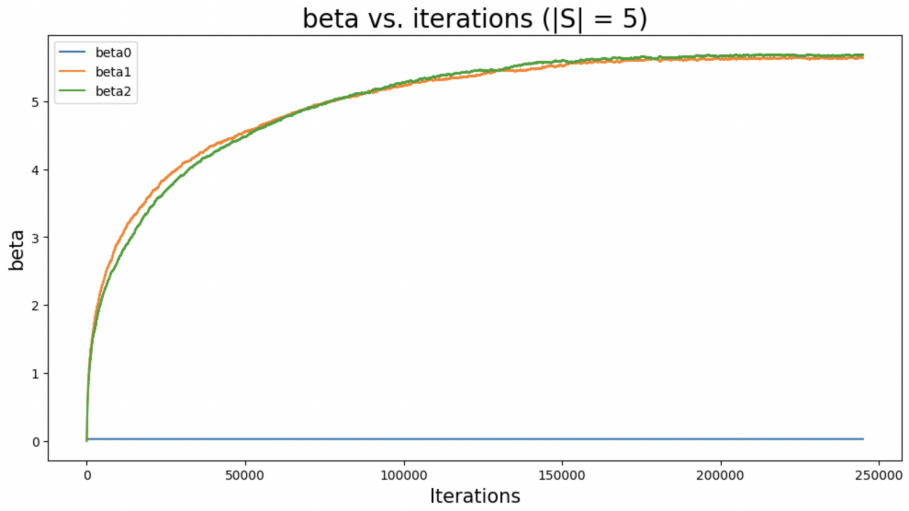


Figure 3.13: For $\gamma = 0.001$

- For $\gamma = 0.1$, the no. of iterations = 1350 which is quite less compared to the previous ones but the trajectory obtained is very noisy.

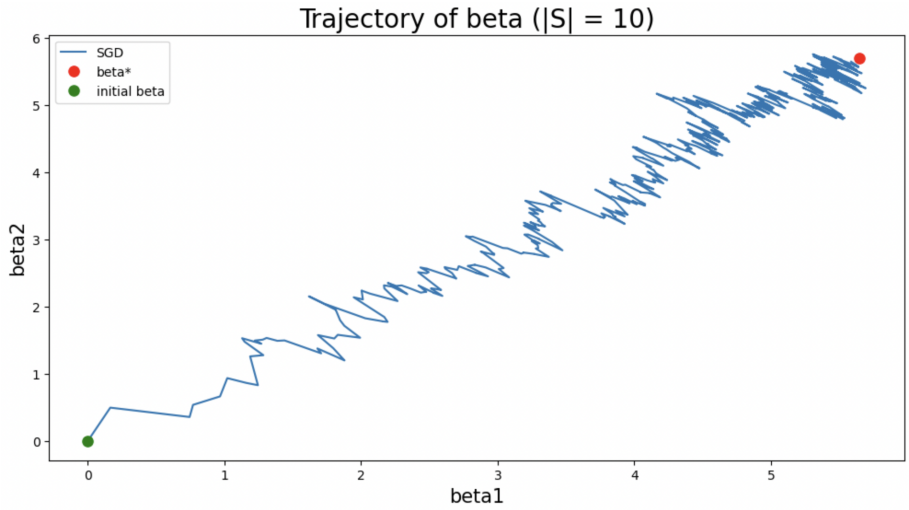


Figure 3.14: For $\gamma = 0.1$

Figure 3.15: For $\gamma = 0.1$

4 Machine Learning Models

4.1 Linear Regression

Linear regression is a widely used statistical modeling technique for predicting a continuous target variable based on one or more input features. It assumes a linear relationship between the input variables and the target variable. The goal of linear regression is to find the best-fitting line that minimizes the difference between the predicted values and the actual values of the target variable.

Consider the hypothesis function for vector of input variables, X as

$$h(x_1, x_2) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + ... + \theta_n x_n$$

$$\theta = \begin{pmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \vdots \\ \theta_n \end{pmatrix}, x = \begin{pmatrix} 1 \\ x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \tag{4.1}$$

θ = Parameters
X = inputs/features
Y = output/target variable
(X, Y) = training example
$(x^{(i)}, y^{(i)}) = i^{th}$ training example
m = No. of training examples
n = No. of features
In general, $h(X) = \Sigma_{j=0}^{n} \theta_j x_j$, where $x_0 = 1$
In the linear regression algorithm, to find the best fit line we need to minimise the **Mean Squared Error**

$$J(\theta) = \frac{1}{2} \Sigma_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$$

This is minimised using gradient descent algorithm.

- θ is updated as

$$\theta_j := \theta_j - \alpha \frac{\partial J(\theta)}{\partial \theta_j}$$

(j = 0,1,2,...,n)

$$\theta_j := \theta_j - \alpha \Sigma_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

- This is repeated until convergence.

4.2 Logistic Regression

Linear Regression is not a good idea for classification problems. Therefore, there is a need for Logistic Regression.
In logistic regression , we want the hypothesis $h_\theta(x) \in [0, 1]$. Sigmoid or Logistic function is used to do this.

$$h_\theta(x) = g(\theta^T X) = \frac{1}{1 + e^{-\theta^T X}}$$

This function forces the output values to be between 0 and 1. Here, θ is the vector of weights and X is the feature vector.
For Binary Classification, if $g(\theta^T X) \geq 0.5$, then $x \in$ Class1, else $x \in$ Class0.

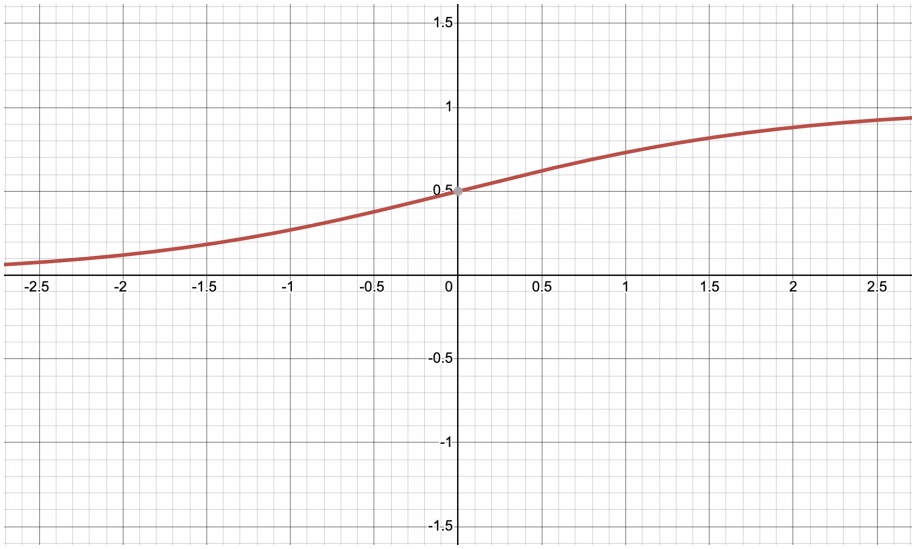


Figure 4.1: Sigmoid Function

4.2.1 Where do we get the Sigmoid Function From?

Let Y $\in 1,2,3,...,K$. The **Likelihood Ratio** is defined as

$$\frac{Pr(Y=k_1|X=x)}{Pr(Y=K|X=x)} \geq 0$$

$\forall k_1 = 1,2,...,K-1$
Assume :

$$log\Big(\frac{Pr(Y=k_1|X=x)}{Pr(Y=K|X=x)}\Big) = \theta_{k_1}^T x$$

$$Pr(Y=k_1|X=x) = \frac{e^{\theta_{k_1} x}}{1+\Sigma_{k=1}^{K-1} e^{\theta_k x}}$$

$$Pr(Y=K|X=x) = \frac{1}{1+\Sigma_{k=1}^{K-1} e^{\theta_k x}}$$

The optimisation problem is :

$$\underset{\theta}{max}\Big(\Sigma_{k_1=1}^{K-1}\Sigma_{i=1}^{n}\Big(log\Big(\frac{Pr(Y=k_1|X=x_i)}{Pr(Y=K|X=x_i)}\Big)\Big)\Big)$$

The problem is solved using **batch gradient ascent**.
The Multiclass Problem is broken down into multiple binary problems and then the classification is done.
Logistic Regression on MNIST dataset

```
from sklearn import linear_model, metrics
reg = linear_model.LogisticRegression()
reg.fit(x, y)
y_pred = reg.predict(x_test)
```

Figure 4.2: Code for Logistic Regression

```
ac_score = accuracy_score(y_test,y_pred)
ac_score
```

```
0.9109910991099109
```

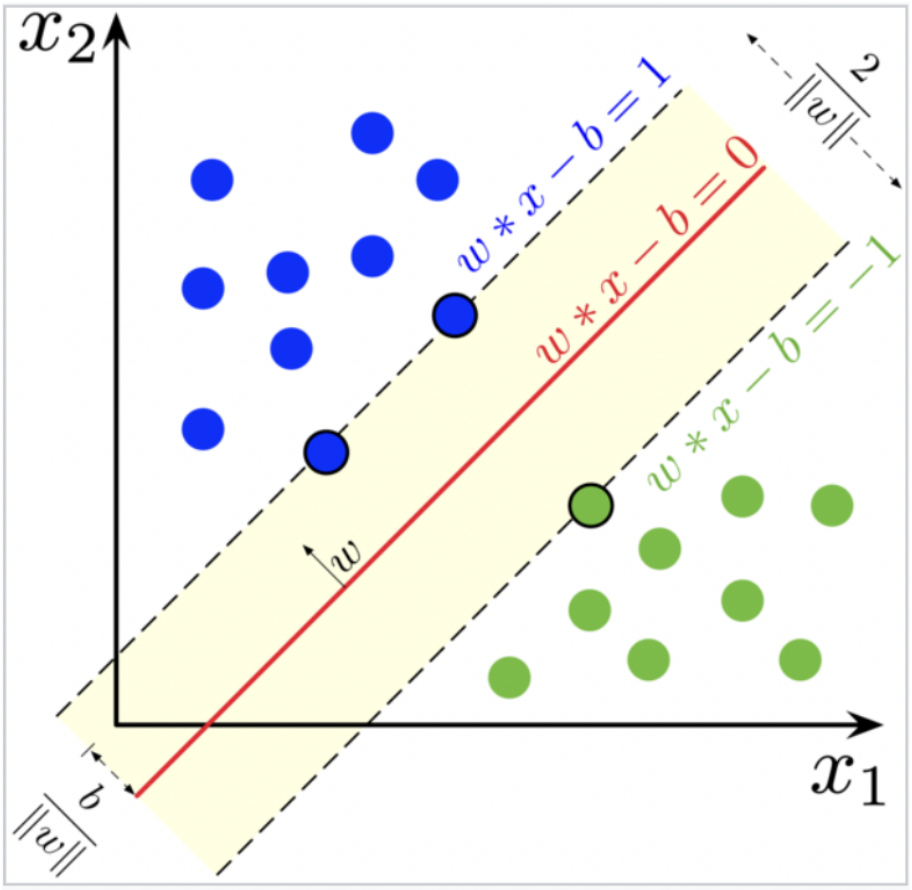
Figure 4.3: Accuracy Score for Logistic Regression

4.3 Support Vector Machines (SVM)

Support Vector Machines (SVMs) are powerful supervised machine learning models used for classification and regression tasks. SVMs are particularly effective in handling high-dimensional feature spaces and cases where the data is not linearly separable.
The basic idea behind SVMs is to find an optimal hyperplane that separates different classes in the feature space while maximizing the margin, which is the distance between the hyperplane and the nearest data points of each class. These data points, called support vectors, are the critical elements for defining the decision boundary.

4.3.1 Linear SVM

Consider we have n examples in the training dataset, represented as $(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)$ where $y_i \in -1, 1$ indicating the class of x_i. The maximum margin hyperplane divides the data-points such that the distance between the hyperplane and the nearest point is maximized.



Maximum-margin hyperplane and margins for an SVM trained with samples from two classes. Samples on the margin are called the support vectors.

Figure 4.4: Hyperplane:Picture taken from WikiPedia

Any Hyperplane can be written as $w^T x - b = 0$.
Hard-margin If the training data is linearly separable, we can select two parallel hyperplanes that separate the two classes of data, so that the distance between them is as large as possible. The region bounded by these two hyperplanes is called the "margin", and the maximum-margin hyperplane is the hyperplane that lies halfway between them.

$w^T x - b = 1$: anything on or above this boundary is of one class, with label 1

$w^T x - b = -1$: anything on or below this boundary is of the other class, with label -1

or $w^T x - b \geq 1$ if $y_i = 1$
and $w^T x - b \leq 1$ if $y_i = -1$

This can be written as :

$$y_i(w^T x_i - b) \geq 1 \forall i \in [1, n]$$

We need to maximise the distance between the two hyperplanes i.e., $\frac{2}{||w||}$ or in other words we need to minimise $||w||$.
The optimisation problem becomes :

$$\underset{w,b}{min} \frac{||w||^2}{2}$$

$$s.t. y_i(w^T x_i - b) \geq 1 \forall i \in 1, 2, 3, ..., n$$

Soft-margin
If the data is not linearly-separable, the hinge-loss function is defined as

$$\xi_i = (1 - y_i(w^T x_i + b))_+$$

The overall optimisation problem is

$$\underset{w,b,\xi_i}{min} \frac{||w||^2}{2} + C\Sigma_{i=1}^{n} \xi_i$$

$$s.t. y_i(w^T x_i + b) \geq (1 - \xi_i) \forall i$$

$$\xi_i \geq 0 \forall i$$

$$\xi_i = 0 : \text{Equivalent to separable setting}$$

$$\xi_i \in (0, 1] : \text{Outlier}$$

$$\xi_i > 1 : \text{Misclassified}$$

Nonlinear Kernels
To solve non-linearly separable data points, we map the data into high- dimensional and then the hyperplane which separates the points in higher dimension is remapped into the original space.
Some of the common kernels are :

- Polynomial(homogeneous):$k(x_i, x_j) = (x_i.x_j)^d$
- Polynomial(inhomogeneous):$k(x_i, x_j) = (x_i.x_j + r)^d$
- Gaussian radial basis function:$k(x_i, x_j) = e^{-\gamma||x_i - x_j||^2}$ for$\gamma > 0$.

4.3.2 SVM Implementation

Generate points satisfying the equation $x^2 + y^2 - 1 = 0$ and assign the labels as

$$f(x) = \begin{cases} -1 & \text{if } x^2 + y^2 < 0 \\ +1 & \text{if } x^2 + y^2 \geq 0 \end{cases} \tag{4.2}$$

Fit SVM to define the decision boundary.

- Generating the data points.

Plotting for the radial function

```
[ ]  X = np.random.randn(400,2)
     sc = StandardScaler()
     X = sc.fit_transform(X)
     X1 = X[:,0]
     X2 = X[:,1]
```

```
     Y = np.zeros(400)
     for i in range(400):
       if (X[i][1]**2 + X[i][0]**2 - 1 <0):
         Y[i] = 1
       else :
         Y[i] = -1
```

```
[ ]  df1 = pd.DataFrame(X, columns=['X1','X2'])
     df2 = pd.DataFrame(Y, columns=['Y'])
     frames = [df1,df2]
     df = pd.concat(frames, axis=1)
     df.head()
```

	X1	X2	Y
0	0.839352	-0.165348	1.0
1	0.843829	1.342426	-1.0
2	0.059637	-0.393144	1.0
3	-0.625098	-2.292090	-1.0
4	1.165525	0.244021	-1.0

Figure 4.5: Generation of points

- Visualising the Data Points generated.

```
sns.scatterplot(x='X1',y='X2',hue='Y',data=df)
```

```
<Axes: xlabel='X1', ylabel='X2'>
```

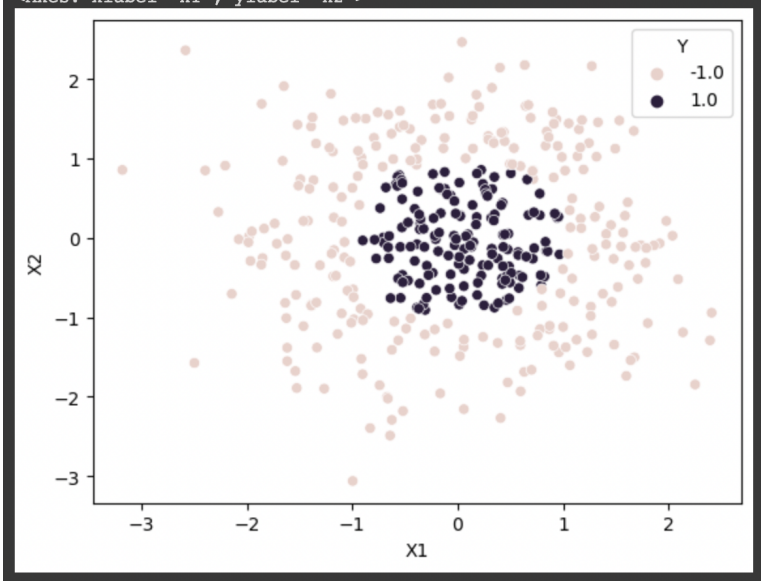


Figure 4.6: Visualising the points

- Fitting and finding the accuracy.

```
svm2 = SVC(kernel='rbf',gamma='auto')
```

```
svm2.fit(X,Y)
```

```
SVC
SVC(gamma='auto')
```

```
yhat = svm2.predict(X)
acc = accuracy_score(Y,yhat)
acc
```

```
0.9825
```

Figure 4.7: Model to fit the Data Points

- Code to visualise the decision boundary.

```
def plot_decision_boundary(pred_func):
    # Set min and max values and give it some padding
    x_min, x_max = X[:, 0].min() - .5, X[:, 0].max() + .5
    y_min, y_max = X[:, 1].min() - .5, X[:, 1].max() + .5
    h = 0.01
    # Generate a grid of points with distance h between them
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
    # Predict the function value for the whole gid
    Z = pred_func(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    # Plot the contour and training examples
    plt.contourf(xx, yy, Z, cmap=plt.cm.Spectral)
    plt.scatter(X[:, 0], X[:, 1], c=Y, cmap=plt.cm.Spectral)
```

Figure 4.8: Code to Plot Decision Boundary

- Visualisation of Decision Boundary.

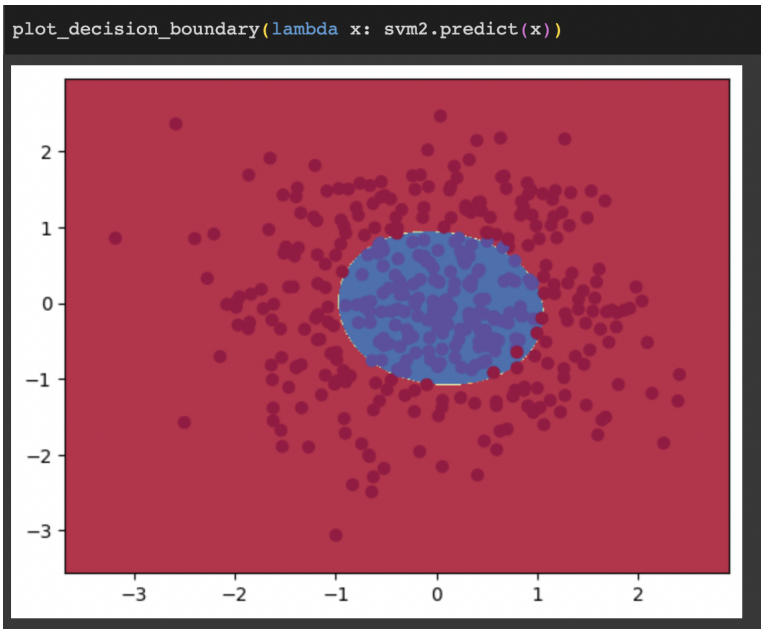


Figure 4.9: Decision Boundary

5 Accuracy Score using Different Learning Models

Model	Accuracy
Linear SVM (C=1)	0.917391
Linear SVM (C=10)	0.917391
Linear SVM (C=100)	0.917391
Polynomial(degree=2) SVM (C=1)	0.9668967
Polynomial(degree=2) SVM (C=10)	0.9716972
Polynomial(degree=2) SVM (C=100)	0.9718971
Polynomial(degree=3) SVM (C=1)	0.9635964
Polynomial(degree=3) SVM (C=10)	0.9716972
Polynomial(degree=3) SVM (C=100)	0.96709671
MLPClassifier(solver=adam, activation=logistic)	0.95669567