CS 6375 – Assignment 1

Lakshmipriya Narayanan

1. **Loss Functions for Linear Regression (6 points):** Assume that the hypothesis function is $f(w, b, x) = w^T x + b$. In the standard linear regression case, given an instance x_i, y_i on the training set, the loss function is defined as $L_i(w, b) = [f(w, b, x_i) - y_i]^2$. Imagine that we are still interested in posing the optimization problem (over the dataset) as:

$$\min_{w,b} \sum_{i=1}^{n} L_i(w, b) \tag{1}$$

What if we were to use some slightly different loss functions? Please look at the following four loss functions and answer the questions below:

(a) $L_i(w, b) = |f(w, b, x_i) - y_i|^3$
(b) $L_i(w, b) = [f(w, b, x_i) - y_i]^3$
(c) $L_i(w, b) = \exp[f(w, b, x_i) - y_i]$
(d) $L_i(w, b) = \max(0, 1 - y_i f(w, b, x_i))$

Part 1: Please answer exactly why these loss functions may or may not be a good choice for regression.
Part 2: Also, compute the gradients of the loss function in each of the cases above.

a) $L_i(w,b) = |f(w,b,x_i) - y_i|^3 = |w^T x_i + b - y_i|^3$

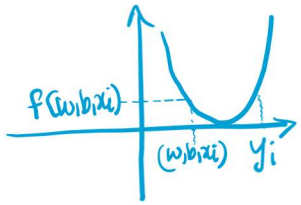
gradients of $L_i(w,b)$: $\nabla_w L_i(w,b) = \frac{(w^T x_i + b - y_i)^3}{|w^T x_i + b - y_i|^3} \cdot 3(w^T x_i + b - y_i)^2 \cdot x_i$

$= 3(w^T x_i + b - y_i)^2 \cdot \text{sign}(w^T x_i b - y_i) \cdot x_i$

$\nabla_b L_i(w,b) = \frac{(w^T x_i + b - y_i)^3}{|w^T x_i + b - y_i|^3} \cdot 3(w^T x_i + b - y_i)^2 (1) = 3(w^T x_i + b - y_i)^2 \cdot \text{Sign}(w^T x_i + b - y_i)$

Convexity: If a function is convex then it's considered a good loss function. For convex functions, their graph must be U-shaped

For $L_i(w,b)$ above,



is the graph

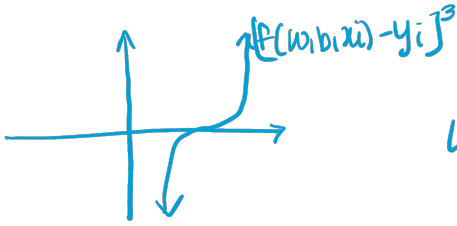
∴ Hence, it's a good loss function

b) $L_i(w,b) = [f(w,b,x_i) - y_i]^3 = (w^T x_i + b - y_i)^3$

Gradients of $L_i(w,b)$: $\nabla_w L_i(w,b) = 3(w^T x_i + b - y_i)^2 \cdot (x_i)$

$\nabla_b L_i(w,b) = 3(w^T x_i + b - y_i)^2 (1)$

Convexity: The graph of $(f(w,b,x_i) - y_i)^3$ is similar to the graph of $y = x^3$ but shifted right by y_i units.

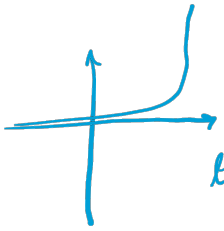


Since it's not U-shaped, this loss function is <u>not</u> a good one.

c) $L_i(w,b) = e^{[f(w,b,x_i) - y_i]} = e^{[w^T x_i + b - y_i]}$

Gradients: $\nabla_w L_i(w,b) = e^{[w^T x_i + b - y_i]} \cdot x_i$

$\nabla_b L_i(w,b) = e^{[w^T x_i + b - y_i]} \cdot (1)$

Convexity: Shape of $L_i(w,b)$'s graph is  (similar to that of $y = e^x$)

Since the graph is not U-shaped,
$L_i(w,b)$ is <u>not a good loss function</u>.

d) $L_i(w,b) = \max[0, 1 - y_i f(w,b,x_i))$

Gradients: $\nabla_w L_i(w,b) \Rightarrow$ <u>Case 1</u>: If $(1 - y_i(w^T x_i + b)) > 0$ then

$L_i(w,b) = 1 - y_i(w^T x_i + b)$

$\therefore \nabla_w L_i = 0 - y_i(x_i) = \underline{-x_i y_i}$

<u>Case 2</u>: If $(1 - y_i(w^T x_i + b)) \leq 0$ then

$L_i(w,b) = 0 \Rightarrow \underline{\nabla_w L_i = 0}$ ——①

$$\therefore \nabla_w L_i(w,b) = \begin{cases} -x_i y_i, & 1 - y_i f(w,b,x_i) > 0 \\ 0 & \text{otherwise} \end{cases}$$

$\nabla_b L_i(w,b)$: <u>Case 1</u>: If $(1 - y_i(w^T x_i + b)) > 0$ then

$L_i(w,b) = 1 - y_i(w^T x_i + b)$

$\therefore \nabla_b L_i = 0 - y_i(1) = -y_i$

<u>Case 2</u>: Same as ① above

$$\therefore \nabla_b L_i(w,b) = \begin{cases} -y_i, & (1 - y_i f(w,b,x_i)) > 0 \\ 0 & \text{otherwise} \end{cases}$$

This function is not a good loss function because when we computed the gradients above, we can notice that this function is not differentiable when the maximum changes & it is also not convex b/c it has more than one local maxima

2. **Loss Functions for Classification (6 Points):** Again, assume that the function is $f(w,b,x) = w^T x + b$. In the case of SVM and Perceptrons, we saw the following two loss functions: $L_i(w,b) = \max(0, -y_i f(w,b,x_i))$ for Perceptron and $L_i(w,b) = \max(0, 1 - y_i f(w,b,x_i))$ for Hinge Loss (SVM). Similar to question 1, let us see if the following loss functions are good choices.

   (a) $L_i(w,b) = \max(0, 1 - y_i f(w,b,x_i))^2$
   (b) $L_i(w,b) = [y_i - f(w,b,x_i]^4$
   (c) $L_i(w,b) = \exp[f(w,b,x_i) - y_i]$
   (d) $L_i(w,b) = \exp[-y_i f(w,b,x_i)]$

   Part 1: Please answer exactly why these loss functions may or may not be a good choice for classification.
   Part 2: Also, compute the gradients of the final loss function in each of the cases above.
   Part 1 is for four points and Part 2 is for two points.

a) $L_i(w,b) = \max(0, 1 - y_i(w^T x_i + b))^2$

Gradients: $\nabla_w L_i(w,b) \Rightarrow$ Case 1 : If $(1 - y_i(w^T x_i + b)) > 0$ then,

$$L_i(w,b) = [1 - y_i(w^T x_i + b)]^2$$

$$\therefore \nabla_w L_i(w,b) = 2[1 - y_i(w^T x_i + b)] \cdot (0 - y_i(x_i))$$

$$= \underline{-2 y_i x_i [1 - y_i(w^T x_i + b)]}$$

Case 2: If $(1 - y_i(w^T x_i + b)) \leq 0$ then

$$L_i(w,b) = 0 \Rightarrow \nabla_w L_i(w,b) = \underline{0} \quad \text{—} \circledast$$

$$\therefore \nabla_w L_i(w,b) = \begin{cases} -2 x_i y_i [1 - y_i f(w, x_i, b)], & [1 - y_i(w^T x_i + b)] > 0 \\ 0 & \text{otherwise} \end{cases}$$

$\nabla_b L_i(w,b) \Rightarrow$ Case 1 : If $(1 - y_i(w^T x_i + b)) > 0$ then

$$\nabla_b L_i(w,b) = 2[1 - y_i(w^T x_i + b)] \cdot (0 - y_i(1))$$

$$= \underline{-2 y_i [1 - y_i(w^T x_i + b)]}$$

Case 2: If $(1 - y_i(w^T x_i + b)) \leq 0$ then

same as (*) in case 1 of $\nabla_w L_i(w,b)$

$$\therefore \nabla_b L_i(w,b) = \begin{cases} -2y_i\,[1 - y_i f(w, x_i, b)] & \text{if } (1 - y_i(w^T x_i + b)) > 0 \\ 0 & \text{otherwise} \end{cases}$$

Here, $L_i(w,b)$ is the square of the hinge loss function. Since, its a squared function it penalizes the misclassifications of data more heavily than the normal hinge loss function.
Therefore, $L_i(w,b)$ is a good loss function.

b) $L_i(w,b) = [y_i - (w^T x_i + b)]^4$

Gradients: $\nabla_w L_i(w,b) = 4[y_i - (w^T x_i + b)]^3 \cdot [0 - (x_i)] = -4x_i[y_i - f(w,b,x_i)]^3$

$$\nabla_b L_i(w,b) = 4[y_i - (w^T x_i + b)]^3 \cdot [0 - (0+1)] = 4[y_i - f(w,b,x_i)]^3$$

$L_i(w,b)$ is a fourth degree function $\Rightarrow$ errors are penalized to the quadric power $\Rightarrow$ sensitivity in misclassifications.

$\therefore$ it is not a good loss function.

c) $L_i(w,b) = e^{[w^T x_i + b - y_i]}$

Gradients: $\nabla_w L_i(w,b) = e^{[w^T x_i + b - y_i]} \cdot [x_i] = x_i\, e^{[f(w,b,x_i) - y_i]}$

$$\nabla_b L_i(w,b) = e^{[w^T x_i + b - y_i]} \cdot [1] = e^{[f(w,b,x_i) - y_i]}$$

from the gradients computed above the curves are not smooth $\Rightarrow$ large values when $w^T x_i + b - y_i$ b/c we've taking a difference of $f(w, x_i, b)$ & y_i which is the label.

$\therefore$ This is not a good loss function.

d) $L_i(w,b) = e^{[-y_i(w^T x_i + b)]}$

Gradients: $\nabla_w L_i(w,b) = e^{[-y_i(w^T x_i + b)]} \cdot (-x_i y_i) = \underline{-x_i y_i e^{-y_i f(w,b,x_i)}}$

$\nabla_b L_i(w,b) = e^{[-y_i(w^T x_i + b)]} \cdot [0 - y_i] = \underline{-y_i e^{-y_i f(w,b,x_i)}}$

This function is a good loss function b/c we can see from the gradient computed above that the graph/curve will be smooth & leading to less sensitivity to misclassifiers.

3. **Polynomial and Higher order Features (3 Points):** Let us use polynomial features with an SVM. Consider the dataset shown below. [Hint: The dataset is not linearly separable]. Note that

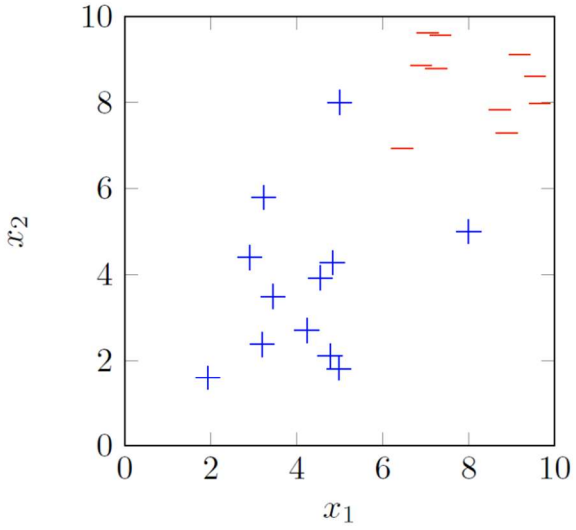


Figure 1: Two Dimensional Data

this dataset consists of 2-dimensional points $x = [x_1, x_2]$.

Part 1 (2 points): Write down the SVM loss function (using Hinge Loss) with quadratic features. First write down what will be the features, the dimensionality of the expanded (quadratic) feature set and the loss function. Is the dataset linearly separable with quadratic features?

Part 2 (1 point): Draw approximately the output of the SVM algorithm on this dataset.

Part 1: Features of this data set:

Since the data is not linearly separable, our features will be

x_1, x_2 are the original features

x_1^2, x_2^2 are the quadratic features

x_1x_2 is the interaction term &

1 is the intercept

∴ Feature vector = $[x_1, x_2, x_1x_2, x_1^2, x_2^2, 1]$

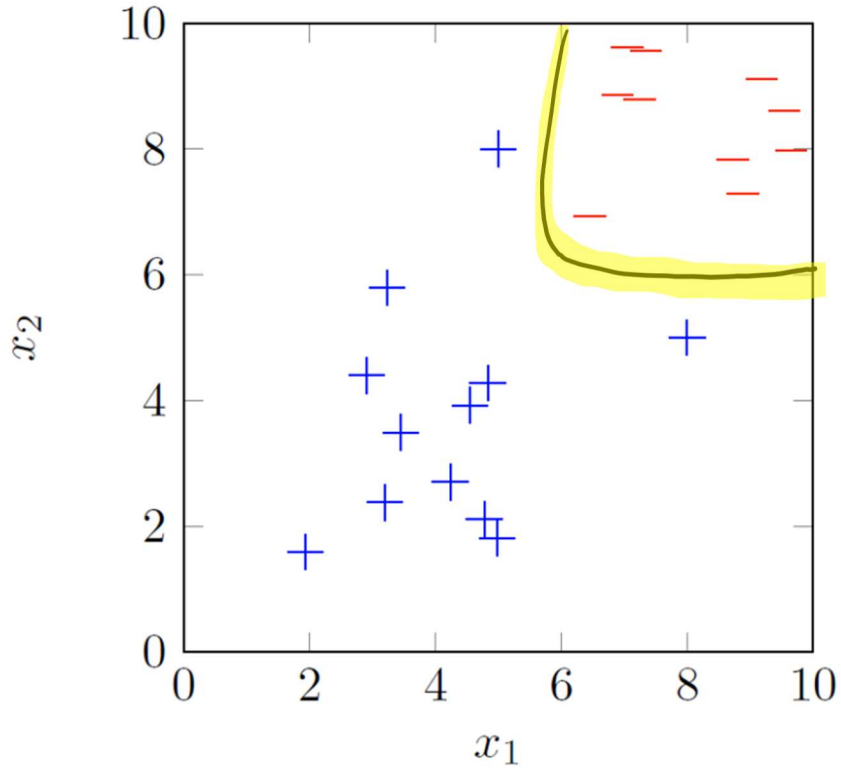
Dimension = # of features in feature vector

= 6

∴ Hinge Loss function = $\max(0, 1 - y_i f(x_i))$ where

$f(x_i) = w^T [x_1, x_2, x_1x_2, x_1^2, x_2^2, 1] + b$.

Part 2: Output of SVM algorithm



QUESTION 4

```
# Import required libraries
import numpy as np
import sklearn
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import pandas as pd
import matplotlib.pyplot as plt
```

Double-click (or enter) to edit

```
# Taken from DEMO 1 provided in class github to read and load the Boston Housing dataset
class BostonHousingDataset:
    def __init__(self):
        self.url = "http://lib.stat.cmu.edu/datasets/boston"
        self.feature_names = ["CRIM", "ZN", "INDUS", "CHAS", "NOX", "RM", "AGE", "DIS", "RAD", "TAX", "PTRATIO", "B", "LSTAT"]

    def load_dataset(self):
        # Fetch data from URL
        raw_df = pd.read_csv(self.url, sep="\s+", skiprows=22, header=None)
        data = np.hstack([raw_df.values[::2, :], raw_df.values[1::2, :2]])
        target = raw_df.values[1::2, 2]

        # Create the dictionary in sklearn format
        dataset = {
            'data': [],
            'target': [],
            'feature_names': self.feature_names,
            'DESCR': 'Boston House Prices dataset'
        }

        dataset['data'] = data
        dataset['target'] = target

        return dataset

# Load the Boston Housing Dataset from sklearn
boston_housing = BostonHousingDataset()
boston_dataset = boston_housing.load_dataset()
boston_dataset.keys(), boston_dataset['DESCR']

# Create the dataset
boston = pd.DataFrame(boston_dataset['data'], columns=boston_dataset['feature_names'])
boston['MEDV'] = boston_dataset['target']
boston.head()

X = boston.to_numpy()
X = np.delete(X, 13, 1)
y = boston['MEDV'].to_numpy()
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state=5)


# We Standardise our features X from the training and test data set to avoid
# 'NaN' (Not a Number) values while calculating predicted value (y_hat) using
# the sklearn library

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

Simple linear regression from scratch

```
class LinReg:
    # Initialization function to initialise weight(w), bias(b), learning
    # rate(gamma) and number of iterations(n) of training data
    def __init__(self):
        # We set weights and bias to 0 during training for linear regression to
        # help during Gradient Descent because Gradient descent updates the
        # weights and bias'.
        self.w = None
```

```
        self.b = None
        self.gamma = 0.01 # A higher learning rate increases the MSE drastically. With 0.01, MSE and MAE match the computaion from sklearn
        self.n = 1000

    # Function to predict linear regression of y_hat = w transpose * X + b
    def predict(self, X):
        y_predicted = np.dot(X, self.w) + self.b

        # Check for NaN values in y_predicted and handle them (e.g., replace with 0). If data is not standradised, y_hat had Nan values.
        # y_predicted[np.isnan(y_predicted)] = 0
        return y_predicted

    # Function to fit our model with features X and labels y
    def fit_trainData(self, X, y):
        # Get the number of samples and number of features from the data and set
        # it to the size of matrix X
        num_sample, num_feat = X.shape

        # Initialize weights and bias to zeros. Here, weight,w = Array of zeroes of size same as
        # X(number of features) because each feature has a weight and there are
        # X such features and we do this for n such samples
        self.w = np.zeros(num_feat)
        self.b = 0

        # Perform Gradient descent for a specified number of n (use of for loop to do it n times)
        for i in range(self.n):
            # Predict the result by using y_hat = w transpose X + b
            y_prediction = np.dot(X, self.w) + self.b

            # Calculate the gradients in order to update weights and bias for each iteration. Here we use the jacobian
            # dw = 2/total number of data points * sum of (2*X_i*(y_hat - y_i)) over i from (1 to n)
            dw = (2 / num_sample) * np.dot(X.T, (y_prediction - y))
            # db = 1/total number of data points * sum of (2*(y_hat - y_i)) over i from (1 to n)
            db = (1 / num_sample) * np.sum(y_prediction - y)

            # Update the weights and bias with respect to learning rate gamma and gradients of w and b respectively
            self.w = self.w - self.gamma * dw
            self.b = self.b - self.gamma * db

            # Debugging process to catch where nan values are being generated -- issue was fixed by standradising the data
            # if i%100 == 0:
              # loss = mse_loss(y, y_prediction)
              # print(f"Iteration {i}: MSE  = {loss}")
              # maeloss = mae_loss(y, y_prediction)
              # print(f"Iteration {i}: MAE  = {maeloss}")

def mse_loss(y_test, y_predicted):
    # Calculate the mean squared error loss function, MSE = 1/N sum of (y_i - y_hat)^2 over i = 1 to N
    return np.mean((y_test - y_predicted) ** 2)

def mae_loss(y_test, y_predicted):
    # Calculate the mean absolute error loss function, MAE = 1/N sum of absolute value of (y_i - y_hat) over samples i = 1 to N
    return np.mean(np.abs(y_test - y_predicted))

# Now, we fit our regression model with our feature from the test data
reg_model = LinReg() # Call class where linear regression is defined
reg_model.fit_trainData(X_train_scaled,y_train) # Function call
predictions = reg_model.predict(X_test_scaled) # Predict values

# Check if there are any NaN values in the predictions array. When data is not standardised I had to deal with nan values
# has_nan = np.isnan(predictions).any()
# print(f"Contains NaN values: {has_nan}")

# Output MSE and MAE results
mse = mse_loss(y_test, predictions)
print(f"Mean squared error, MSE (from scratch): {mse}")
mae = mae_loss(y_test, predictions)
print(f"Mean absolute error, MAE (from scratch): {mae}")
```

```
Mean squared error, MSE (from scratch): 20.66211560646739
Mean absolute error, MAE (from scratch): 3.2114751468373264
```

```
# Compare above obtained results using sklearn library
from sklearn.linear_model import LinearRegression

# Fit linear regression model using sklearn built in functions just like we did above
lin_model_sklearn = LinearRegression()
```

```
lin_model_sklearn.fit(X_train, y_train)
prediction_sklearn = lin_model_sklearn.predict(X_test)

from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error

# Calculate MSE using sklearn
mse_sklearn = mean_squared_error(y_test, prediction_sklearn)
print(f"MSE from sklearn: {mse_sklearn}")

# Calculate MAE using sklearn library
mae_sklearn = mean_absolute_error(y_test, prediction_sklearn)
print(f"MAE from sklearn: {mae_sklearn}")
```

```
MSE from sklearn: 20.86929218377085
MAE from sklearn: 3.2132704958423806
```

We observe that with learning rate, alpha = 0.01 and a 1000 iterartions, our MSE and MAE are very close to he ones obtained by the library. MAE is almost the same if we round to the hundredths ie, MAE = 3.21 for both methods. Whereas, MSE from scratch = 20.66 and MSE form sklearn = 20.86 which are not drastically apart from each other. Therefore, if learning rate changes then loss function values will also vary when done from scratch.

QUESTION 5

```
# Taken from SVM Demo taken from class github
from sklearn.datasets import make_blobs
X, y = make_blobs(n_samples=50, centers=2, random_state=0, cluster_std=0.60)

# Use an indicator function to convert y labels from 0 to -1 for a class and 0 to 1
# for another class. This is a requirement for SVM because the hinge loss function
# is defined as max(0, 1 - y * f(w,xi,b)) and this fits labels wity values -1 and 1
# because it penalizes the misclassified points and points within the margin 1 and -1 precisely.

y = np.where(y == 0, -1, 1)  # Convert labels to -1 and 1


class Lin_Classify:
    # Initialization function to initialise weight(w), bias(b), learning
    # rate(alpha) and number of iterations(n) of training data just as we did
    # for (4) above
    def __init__(self):
        self.gamma = 0.01
        self.w = None
        self.b = 0
        self.n = 1000

    # Define Perceptron loss function = max(0, -y * (w Transpose X + b))
    def perc_loss(self, X, y):
        return np.maximum(0, -y * (X.dot(self.w) + self.b))

    # Define Hinge loss function = max(0, 1 - y * (w Transpose X + b))
    def hinge_loss(self, X, y):
        return np.maximum(0, 1 - y * (X.dot(self.w) + self.b))

    # We use a subgradient descent to minimize the perceptron loss over the sum of
    # all data points
    def subgradient_descent(self, X, y, loss_func):
        # Initialize weight vector to an array of zeroes. We have already
        # initusalized bias, b = 0 in our initialization function above
        self.w = np.zeros(X.shape[1])

        # We now iterate over a 1000 samples and we also iterate over each data
        # point. So we use two for loops in order to do so.
        for num_iters in range(self.n):
            # Iterate for each (X, y)
            for i in range(len(y)):
                # If the sum of perceptron loss and hinge loss function is
                # positive then update weights and bias' accordingly
                if loss_func(X[i], y[i]) > 0:
                    # (see question 1(d) to look at the gradients of w and b
                    # used here to update )
                    # w = w + learning rate * y * X
                    self.w = self.w + self.gamma * y[i] * X[i]
                    # b = b + learning rate * y
```

```
                    self.b = self.b + self.gamma * y[i]
        return self.w, self.b

    # Calculate total loss over each data point (X, y). We do this because we
    # check for model convergence. If total loss is decreasing then the model is
    # good and converges and vice-versa.
    def total_loss(self, X, y, loss_func):
        tot_losses = loss_func(X, y)
        return np.sum(tot_losses)


# Now we fit our model for classification by calling our linear classification
# methods function we defined above
Perc_SVM_classifier = Lin_Classify()

# Optimization Perceptron Loss using subgradient descent
w_perc, b_perc = Perc_SVM_classifier.subgradient_descent(X, y, Perc_SVM_classifier.perc_loss)
print("w_opt (Perceptron Loss):", w_perc)
print("b_opt (Perceptron Loss):", b_perc)

# Optimizing Hinge Loss function using subgradient descent defined above
w_hinge, b_hinge = Perc_SVM_classifier.subgradient_descent(X, y, Perc_SVM_classifier.hinge_loss)
print("w_opt (Hinge Loss):", w_hinge)
print("b_opt (Hinge Loss):", b_hinge)

# Calculate Total perceptron loss and Total hinge loss for model converegnce
Perc_SVM_classifier.w, Perc_SVM_classifier.b = w_perc, b_perc
total_perc_loss = Perc_SVM_classifier.total_loss(X, y, Perc_SVM_classifier.perc_loss)
print("Total Perceptron Loss:", total_perc_loss)

# Calculate Total Hinge Loss
Perc_SVM_classifier.w, Perc_SVM_classifier.b = w_hinge, b_hinge
total_hinge_loss = Perc_SVM_classifier.total_loss(X, y, Perc_SVM_classifier.hinge_loss)
print("Total Hinge Loss:", total_hinge_loss)
```

```
w_opt (Perceptron Loss): [0. 0.]
b_opt (Perceptron Loss): 0
w_opt (Hinge Loss): [ 1.32966611 -1.75550467]
b_opt (Hinge Loss): 1.8800000000000014
Total Perceptron Loss: 0.0
Total Hinge Loss: 0.0
```

For perceptron loss, we see that optimized weight(w) = 0 and the optimized bias(b) is also 0 which indicates the the perceptron loss function and the gradient descent for it have found a linear separation for the data. Therefore, the data is linearly separable and hence, Total perceptron loss = 0.

For the hinge loss function, optmimal weight, w = [1.33, -1.75] and optimal bias, b = 1.88 and this indicates that the SVM has identified a hyperplane that is maximizing the margins between the two classes. But, we notice that the toatl hinge loss = 0 => data is linearly separable.

Comparing the two loss functions, the total loss is 0 for both which implies that our data is linearly separable. But the wights and bias are = 0 for perceptron because the perceptron loss function found a simple classification boundary. Whereas, the weights and bias for the hinge loss function are non-zero even though the total loss = 0. This is because SVM by definition maximizes the margins and for our data it is linearly separable but with a more flexible boundary.