# SVM with linear Kernel

Name: Debatreya Das Roll No. 12212070 CS-A4 ML Lab 11

## Importing Libraries

import numpy as np  
import matplotlib.pyplot as plt  
# Use sklearn for only the train\_test\_split function and make\_blobs for generating data  
from sklearn.datasets import make\_blobs  
from sklearn.model\_selection import train\_test\_split

## Define the SVM class

* y\_: Converted labels to -1 and 1 for ease of computation.
* For each iteration, we check if the data point meets the condition.
* If correctly classified, only the regularization term updates w.
* If incorrectly classified, both the regularization term and the loss term affect w and b.

class SVM:  
 def \_\_init\_\_(self, learning\_rate=0.001, lambda\_param=0.01, n\_iters=1000):  
 self.learning\_rate = learning\_rate  
 self.lambda\_param = lambda\_param  
 self.n\_iters = n\_iters  
 self.w = None  
 self.b = 0  
  
 def fit(self, X, y):  
 n\_samples, n\_features = X.shape  
 # Initialize weights to zero  
 self.w = np.zeros(n\_features)  
 self.b = 0  
  
 # Convert labels to -1 and 1 if not already  
 y\_ = np.where(y <= 0, -1, 1)  
  
 # Gradient Descent for SVM  
 for \_ in range(self.n\_iters):  
 for idx, x\_i in enumerate(X):  
 condition = y\_[idx] \* (np.dot(x\_i, self.w) - self.b) >= 1  
 if condition:  
 # If the data point is correctly classified  
 self.w -= self.learning\_rate \* (2 \* self.lambda\_param \* self.w)  
 else:  
 # If the data point is incorrectly classified  
 self.w -= self.learning\_rate \* (2 \* self.lambda\_param \* self.w - np.dot(x\_i, y\_[idx]))  
 self.b -= self.learning\_rate \* y\_[idx]  
  
 # Predict the class of the data points  
 def predict(self, X):  
 linear\_output = np.dot(X, self.w) - self.b  
 return np.sign(linear\_output)

## Dataset

# make\_blobs is used to generate data  
X, y = make\_blobs(n\_samples=50, centers=2, random\_state=6)  
y = np.where(y == 0, -1, 1) # Convert labels to -1 and 1

## Train the Model

svm = SVM(learning\_rate=0.001, lambda\_param=0.01, n\_iters=1000)  
svm.fit(X, y)

## Visualize Dataset

* We plot the decision boundary by solving for the line equation based on the SVM's learned weights and bias.
* The decision boundary is the line for which w . x + b = 0.
* We add margin lines at ±1 from the decision boundary to visualize the support vectors.

# Step 6: Visualize the decision boundary  
def plot\_svm\_decision\_boundary(X, y, model):  
 def get\_hyperplane\_value(x, w, b, offset):  
 return (-w[0] \* x + b + offset) / w[1]  
  
 fig, ax = plt.subplots()  
 plt.scatter(X[:, 0], X[:, 1], c=y, cmap='winter')  
  
 # Decision boundary  
 x0\_1 = np.amin(X[:, 0])  
 x0\_2 = np.amax(X[:, 0])  
  
 # Calculate decision boundary points  
 x1\_1 = get\_hyperplane\_value(x0\_1, model.w, model.b, 0)  
 x1\_2 = get\_hyperplane\_value(x0\_2, model.w, model.b, 0)  
  
 # Margin lines  
 x1\_1\_m = get\_hyperplane\_value(x0\_1, model.w, model.b, -1)  
 x1\_2\_m = get\_hyperplane\_value(x0\_2, model.w, model.b, -1)  
 x1\_1\_p = get\_hyperplane\_value(x0\_1, model.w, model.b, 1)  
 x1\_2\_p = get\_hyperplane\_value(x0\_2, model.w, model.b, 1)  
  
 ax.plot([x0\_1, x0\_2], [x1\_1, x1\_2], "k")  
 ax.plot([x0\_1, x0\_2], [x1\_1\_m, x1\_2\_m], "k--")  
 ax.plot([x0\_1, x0\_2], [x1\_1\_p, x1\_2\_p], "k--")  
  
 plt.show()  
  
# Plot the decision boundary  
plot\_svm\_decision\_boundary(X, y, svm)

