Building
Support Vector Machine Algorithm
from Scratch in Python

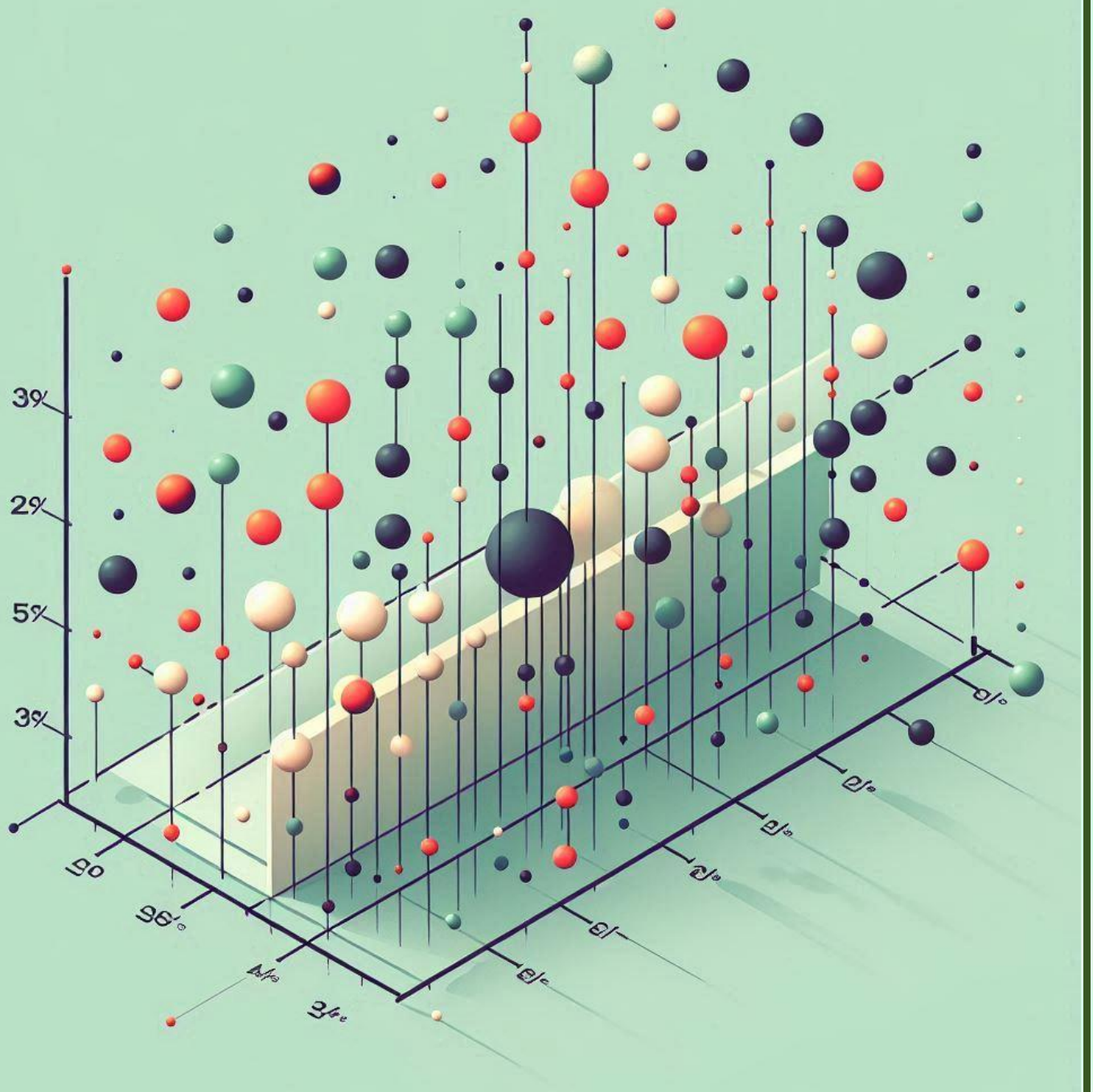


Table of Contents

1. Introduction
2. Fundamental Concepts
3. The Structure of a Support Vector Machine Algorithm
4. Implementation in Python
    a. Import Libraries
    b. Define the SVM Class
    c. Generating Sample Data
    d. Training the SVM Model
    e. Plotting the Decision Boundary
5. Conclusion

1. Introduction to Support Vector Machines Algorithm

Support Vector Machines (SVMs) are powerful and versatile machine learning algorithms used for classification, regression, and outlier detection. The goal of an SVM is to find the optimal hyperplane that best separates the data into different classes. This post will guide you through the process of building an SVM classifier from scratch in Python, **without relying on high-level libraries such as scikit-learn**.

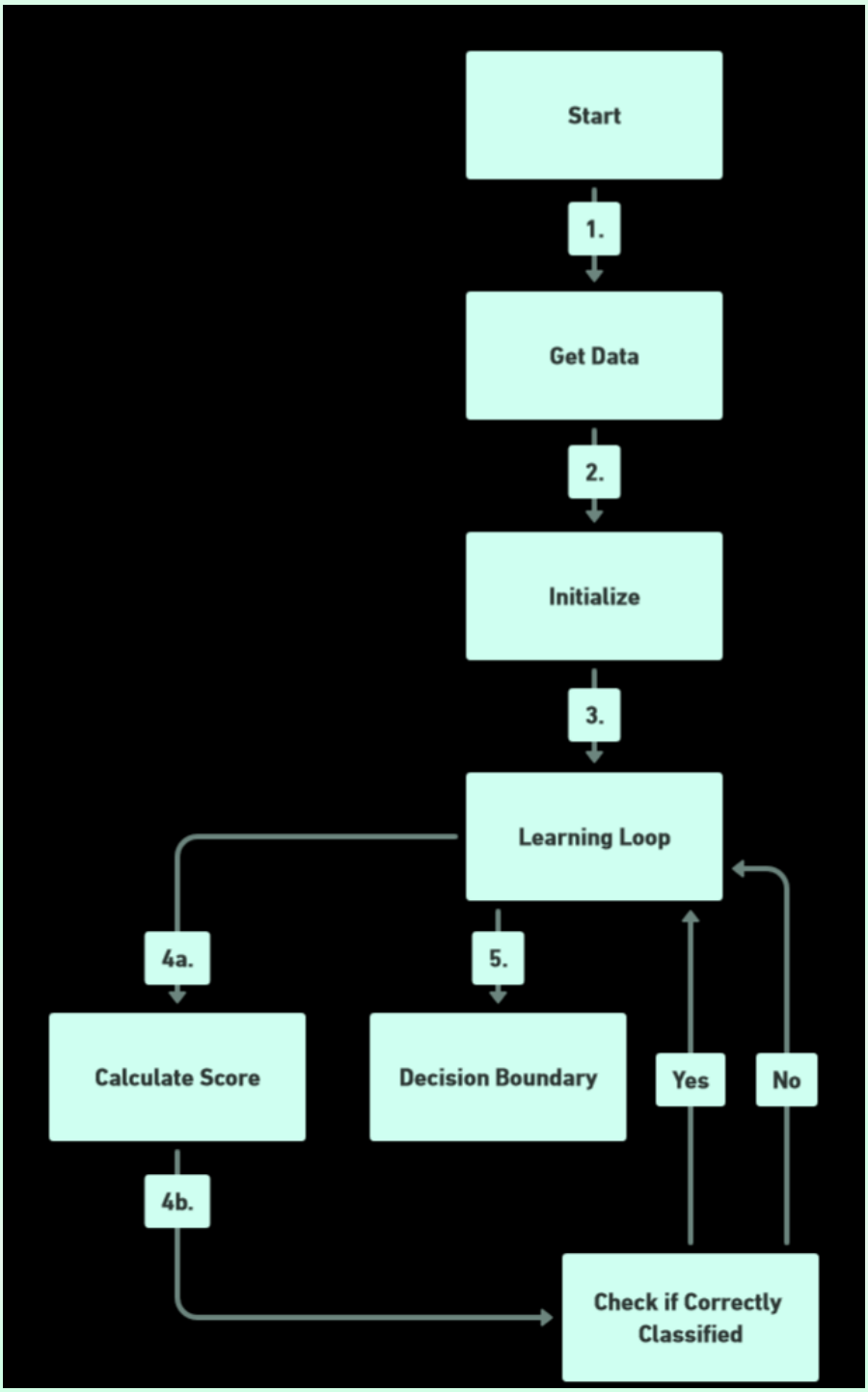
2. Fundamental Concepts

1. **Hyperplane**: A hyperplane in an N-dimensional space is a flat affine subspace of dimension N-1. In 2D, it's a line; in 3D, it's a plane.
2. **Margin**: The distance between the hyperplane and the closest data points from either class. SVMs aim to maximize this margin.
3. **Support Vectors**: Data points that lie closest to the hyperplane. These points are critical in defining the hyperplane.
4. **Kernel Trick**: A method to transform the original non-linear problem into a linear problem by mapping data to a higher-dimensional space.

3. The Structure of a Vector Machine Algorithm

This Structure includes the steps and sub-steps with appropriate labels and connections. Each step corresponds to a function or a key part of the process described in the provided implementation.

- Initialize Parameters: Set initial values for weights and bias.
- Calculate the Hinge Loss: Compute the loss for the current parameters.
- Compute Gradients: Calculate the gradients of the loss function with respect to weights and bias.
- Update Parameters: Update weights and bias using gradient descent.
- Predict: Use the learned weights and bias to make predictions on new data.



5. Implementation in Python

Let's implement a simple Support Vector Machine Algorithm in Python to classify data.

Step 1: Import Libraries

First, we need to import the necessary libraries. We will use NumPy for numerical operations and Matplotlib for plotting

```
import numpy as np
import matplotlib.pyplot as plt
```

Step 2: Define the SVM Class

We define the SVM class, which will contain methods for training the model and making predictions.

Explanation of `fit` Method:

- We start by initializing the weights and bias to zeros.
- Labels are converted to -1 and 1 because SVMs work with these labels.
- The algorithm iterates over the dataset for a specified number of iterations (`n_iters`). For each sample, it checks whether the condition $y_i(w \cdot x_i - b) \geq 1$ is met:
  - If the condition is met, we only apply the regularization term to the weights.
  - If the condition is not met, we update the weights and bias using gradient descent.

Explanation of `predict` Method:

- The predict method computes the dot product of the input features and the learned weights, subtracts the bias, and then applies the sign function to determine the class labels.

```
class SVM:
    def __init__(self, learning_rate=0.001, lambda_param=0.01, n_iters=1000):
        self.lr = learning_rate          # Learning rate for gradient descent
        self.lambda_param = lambda_param # Regularization parameter
        self.n_iters = n_iters           # Number of iterations for training
        self.w = None                    # Weights
        self.b = None                    # Bias

    def fit(self, X, y):
        n_samples, n_features = X.shape

        # Initialize weights and bias to zeros
        self.w = np.zeros(n_features)
        self.b = 0

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

        # Gradient descent for n_iters iterations
        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 condition is met, only apply regularization
                    self.w -= self.lr * (2 * self.lambda_param * self.w)
                else:
                    # Otherwise, apply the full update rule
                    self.w -= self.lr * (2 * self.lambda_param * self.w - np.dot(x_i, y_[idx]))
                    self.b -= self.lr * y_[idx]

    def predict(self, X):
        # Apply the sign function to determine class labels
        approx = np.dot(X, self.w) - self.b
        return np.sign(approx)
```

Building Support Vector Machine Algorithm from Scratch in Python

Step 3: Generating Sample Data

We will generate some sample data using make_blobs from scikit-learn to visualize our SVM classifier.

```
from sklearn.datasets import make_blobs

# Create a simple dataset with 2 classes
X, y = make_blobs(n_samples=50, centers=2, random_state=42)

# Convert labels from {0, 1} to {-1, 1}
y = np.where(y == 0, -1, 1)
```

Step 4: Training the SVM Model

- We initialize the SVM classifier and train it on the sample data.

```
svm = SVM()
svm.fit(X, y)
```

Step 5: Plotting the Decision Boundary

To visualize the results, we need to plot the decision boundary, the margins, and the data points.

```
def plot_hyperplane(X, y, model):
    def get_hyperplane_value(x, w, b, offset):
        return (-w[0] * x + b + offset) / w[1]

    plt.scatter(X[:, 0], X[:, 1], marker='o', c=y)

    x0_1 = np.amin(X[:, 0])
    x0_2 = np.amax(X[:, 0])

     # Plotting the decision boundary
    x1_1 = get_hyperplane_value(x0_1, model.w, model.b, 0)
    x1_2 = get_hyperplane_value(x0_2, model.w, model.b, 0)
    plt.plot([x0_1, x0_2], [x1_1, x1_2], 'k')

    # Plotting the margin
    x1_1 = get_hyperplane_value(x0_1, model.w, model.b, 1)
    x1_2 = get_hyperplane_value(x0_2, model.w, model.b, 1)
    plt.plot([x0_1, x0_2], [x1_1, x1_2], 'k--')

    x1_1 = get_hyperplane_value(x0_1, model.w, model.b, -1)
    x1_2 = get_hyperplane_value(x0_2, model.w, model.b, -1)
    plt.plot([x0_1, x0_2], [x1_1, x1_2], 'k--')

    plt.show()

plot_hyperplane(X, y, svm)
```

5. Conclusion

By breaking down the K-Means clustering algorithm into distinct steps and functions, we gain a better understanding of each component's role and functionality. The initialization of centroids, computation of distances, assignment of clusters, updating of centroids, and checking for convergence are all crucial steps in the algorithm. Implementing these steps in Python provides valuable insights into the mechanics of clustering algorithms.

Feel free to experiment with different datasets and parameters to see how the K-Means algorithm performs and how different initialization methods can impact the clustering results.

Constructive comments and feedback are welcomed