Kernel Trick

Support Vector Machine (SVM) with Kernel Trick: Algorithm Overview

Key Concepts

1. Support Vector Machine (SVM)

SVM is a supervised learning algorithm used for binary classification. It aims to find the optimal hyperplane that maximizes the margin between two classes in the feature space. For non-linearly separable data, SVM uses the **kernel trick** to project the data into a higher-dimensional space where it becomes linearly separable.

2. Kernel Trick

The kernel trick allows us to compute the inner product in the high-dimensional feature space without explicitly transforming the data. Instead, a **kernel function** is used to compute the similarity between data points.

Gaussian RBF Kernel

The Gaussian Radial Basis Function (RBF) kernel is defined as: $K(x_1, x_2) = \exp\left(-\frac{\|x_1 - x_2\|^2}{2\sigma^2}\right)$

This kernel measures the similarity between two data points, with σ controlling the width of the Gaussian.

3. Dual Formulation

In the dual form of SVM, the optimization problem involves Lagrange multipliers (α_i) and the kernel function. The decision function is expressed as: $f(x) = \sum_{i=1}^{n} \alpha_i y_i K(x_i, x) + b$

where:

- x_i: Training samples
- y_i: Corresponding labels (-1 or 1)
- $K(x_i, x)$: Kernel function
- b: Bias term

Algorithm Steps

1. Define the Kernel Function

The RBF kernel computes the similarity between two vectors x_1 and x_2 using the Gaussian formula.

2. Initialize Parameters

- α: Array of Lagrange multipliers, initialized to zeros.
- b: Bias term, initialized to zero.
- C: Regularization parameter to control the trade-off between maximizing the margin and minimizing the hinge loss.

3. Compute the Kernel Matrix

The kernel matrix K is computed for all pairs of training samples: $K[i, j] = K(x_i, x_j)$ This precomputes the similarity between each pair of points in the training set.

4. Train the Model using Gradient Descent

For each training sample:

1. Compute the decision function: $f(x_i) = \sum_{j=1}^{n} \alpha_j y_j K(x_j, x_i) + b$
2. Check the condition for the hinge loss: $y_i f(x_i) < 1$
    - If the condition is satisfied, update α_i to minimize the hinge loss and regularization term.
    - Otherwise, update α_i to minimize the regularization term.
3. Update the bias term b to minimize the overall loss.

5. Make Predictions

For a test sample x, predict the class using the decision function:
$\hat{y} = \text{sign}\left(\sum_{i=1}^{n} \alpha_i y_i K(x_i, x) + b\right)$

Example Workflow

1. Define the kernel function (e.g., RBF kernel).
2. Initialize the SVM with kernel, regularization parameter C, learning rate, and number of iterations.
3. Train the SVM on the training dataset to optimize α and b.
4. Use the trained SVM to predict labels for new test data.

Limitations

- The algorithm uses gradient descent for optimizing α, which may not be as efficient as

Quadratic Programming (QP) methods used in traditional SVM implementations.
- Computational complexity increases with the size of the training set due to the kernel matrix computation.

Why Do We Use a Kernel in SVM?

Key Motivation

In many real-world datasets, the classes are not linearly separable in the original feature space. For instance, imagine a dataset where the positive and negative classes form concentric circles. In such cases, no straight line or hyperplane can separate the two classes in their original 2D space.

What Does a Kernel Do?

1. Maps Data to a Higher-Dimensional Space

The kernel function allows us to transform the data into a higher-dimensional space where the classes become linearly separable. This transformation is defined by a **feature mapping function $\phi(x)$**:

$$x \mapsto \phi(x)$$

2. Avoids Explicit Computation in High Dimensions

Instead of explicitly transforming the data using $\phi(x)$ (which could be computationally expensive or infeasible for infinite-dimensional spaces), the kernel computes the dot product in the transformed space directly:

$$K(x_i, x_j) = \phi(x_i) \cdot \phi(x_j)$$

This means we don't need to know the mapping $\phi(x)$ explicitly. The kernel function computes the similarity between points as if they were in the high-dimensional space.

3. Measures Similarity in High-Dimensional Space

- In the original space, two points x_1 and x_2 might seem far apart.
- In the transformed space, their similarity (dot product) might indicate that they are close to each other in the new feature space.

4. Transforms the Decision Boundary

- In the original space, the decision boundary is a linear hyperplane.

- In the transformed space, the decision boundary becomes non-linear when projected back into the original space.

For example, using an RBF kernel, the SVM can create circular, elliptical, or more complex decision boundaries.

Why Not Transform the Data Explicitly?

Explicitly transforming the data into a high-dimensional space can be:

- **Computationally expensive**, especially for large datasets.
- **Memory-intensive**, as higher dimensions require storing large matrices.
- **Unnecessary**, since the kernel computes the inner product directly, bypassing the need for explicit transformations.

Common Kernel Functions

1. Linear Kernel

$$K(x_1, x_2) = x_1 \cdot x_2$$

Used for linearly separable problems; no transformation is applied.

2. Polynomial Kernel

$$K(x_1, x_2) = (\gamma x_1 \cdot x_2 + r)^d$$

Introduces polynomial relationships between features.

3. Gaussian RBF Kernel

$$K(x_1, x_2) = \exp\left(-\frac{\|x_1 - x_2\|^2}{2\sigma^2}\right)$$

Maps data into an infinite-dimensional space, making it possible to separate complex datasets.

4. Sigmoid Kernel

$$K(x_1, x_2) = \tanh(\gamma x_1 \cdot x_2 + r)$$

Mimics the behavior of a neural network activation function.

Summary

The kernel function is a powerful tool that allows SVMs to handle non-linear problems by:

1. Implicitly transforming data into a higher-dimensional space.
2. Computing similarities in the transformed space efficiently.
3. Enabling the creation of non-linear decision boundaries in the original space.

By using kernels, SVMs become a versatile tool for both linear and non-linear classification problems.

Understanding and Implementing the Gaussian RBF Kernel

Mathematical Definition

The RBF kernel is defined as:

$$K(x_1, x_2) = \exp\left(-\frac{\|x_1 - x_2\|^2}{2\sigma^2}\right)$$

Breakdown of the Formula

1. **Euclidean Distance**:

   $$\|x_1 - x_2\|^2$$

   This term measures how far apart the two vectors x_1 and x_2 are in the input space.

2. **Gaussian Scaling**:

   $$\frac{\|x_1 - x_2\|^2}{2\sigma^2}$$

   - σ is the standard deviation of the Gaussian function.
   - It controls how quickly the similarity decreases as the distance between the points increases.
   - Smaller σ results in a sharper drop-off in similarity, while larger σ makes the kernel smoother.

3. **Exponentiation**:

   $$\exp\left(-\frac{\|x_1 - x_2\|^2}{2\sigma^2}\right)$$

   The exponential function ensures the kernel outputs a similarity score in the range $(0, 1]$, where:

- $K(x_1, x_2) = 1$ when $x_1 = x_2$.
- $K(x_1, x_2) \to 0$ as $\|x_1 - x_2\| \to \infty$.

Implementation Plan

Steps to Implement

1. **Compute the Distance**:
   - Use the Euclidean norm $\|x_1 - x_2\|$.
   - Square the distance to align with the formula.
2. **Scale by σ**:
   - Divide the squared distance by $2\sigma^2$.
   - σ acts as a hyperparameter; it controls how sensitive the kernel is to differences in data points.
3. **Apply the Exponential Function**:
   - Use the exponential function to compute the similarity score.

Debugging and Validation

Input Validation

1. Ensure that x_1 and x_2 have the same dimensions.
2. Check that $\sigma > 0$ to avoid division by zero.

Test Cases

1. **Identical Inputs**:
   - For $x_1 = x_2$, the output should be 1, as $\|x_1 - x_2\| = 0$.
2. **Distant Points**:
   - For large distances between x_1 and x_2, the output should approach 0.
3. **Effect of σ**:
   - For small σ, non-identical points should yield values close to 0.
   - For large σ, the kernel should produce values close to 1 for most points.

Understanding the Initialization of the Kernel SVM

Purpose of `__init__`

The `__init__` method initializes the key parameters and attributes of the Kernel SVM model. These parameters define the behavior of the model during training and prediction.

Key Parameters and Their Roles

1. `kernel`

- **Description**: The kernel function (e.g., RBF kernel) computes the similarity between data points.
- **Mathematical Role**: Defines the transformation to a higher-dimensional space: $K(x_i, x_j) = \phi(x_i) \cdot \phi(x_j)$

2. `C` (Regularization Parameter)

- **Description**: Balances the trade-off between margin size and misclassification.
- **Effect**:
  - Large C: Penalizes misclassifications heavily, resulting in a smaller margin.
  - Small C: Allows more misclassifications but increases the margin size, improving generalization.
- **Mathematical Constraints**: $0 \leq \alpha_i \leq C$

3. `lr` (Learning Rate)

- **Description**: Determines the step size during gradient descent updates.
- **Mathematical Role**: Gradient descent updates the model's parameters iteratively: $\theta^{(t+1)} = \theta^{(t)} - \eta \cdot \nabla J(\theta^{(t)})$ Where:
  - η is the learning rate (`lr`).
  - $J(\theta)$ is the objective function (e.g., hinge loss).

4. `num_iters` (Number of Iterations)

- **Description**: The number of steps in gradient descent.
- **Effect**:
  - More iterations refine the parameters but increase computational cost.
  - Too few iterations may result in suboptimal training.

Tasks for Implementation

1. Store the Parameters

- Save `kernel` , `C` , `lr` , and `num_iters` as attributes of the class for later use.

2. Validate Inputs

- Ensure $C > 0$, $lr > 0$, and `num_iters` is a positive integer.
- Check that `kernel` is a callable function (e.g., RBF kernel).

3. Extendability

- Prepare for additional attributes, such as:
  - Support vectors
  - Lagrange multipliers (α)
  - Bias term (b)

Mathematical Context: Regularization and Gradient Descent

Regularization with C

The optimization problem in the dual form of SVM:
$\min_\alpha \frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n \alpha_i \alpha_j y_i y_j K(x_i, x_j) - \sum_{i=1}^n \alpha_i$ Subject to:
$0 \leq \alpha_i \leq C, \quad \sum_{i=1}^n \alpha_i y_i = 0$

- C controls the penalty for misclassifications, limiting the size of α_i.

Gradient Descent

The iterative update for gradient descent: $\theta^{(t+1)} = \theta^{(t)} - \eta \cdot \nabla J(\theta^{(t)})$ Where:

- η is the learning rate.
- $\nabla J(\theta)$ is the gradient of the loss function.

Implementing the `fit` Method for Kernel SVM

Mathematical Foundations

1. **Dual Form of SVM**

The optimization problem for the dual form of SVM is:
$\min_\alpha \frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n \alpha_i \alpha_j y_i y_j K(x_i, x_j) - \sum_{i=1}^n \alpha_i$ Subject to:
$0 \leq \alpha_i \leq C \quad \text{and} \quad \sum_{i=1}^n \alpha_i y_i = 0$

- α_i: Lagrange multipliers associated with the constraints.
- $K(x_i, x_j)$: Kernel function that computes the similarity between points.
- C: Regularization parameter that controls the margin.

The dual formulation uses the kernel trick, which allows computations to occur in the transformed feature space implicitly.

---

2. **Gradient Descent**

To solve the dual problem, we iteratively update the Lagrange multipliers α and the bias term b using gradient descent.

Decision Function:

The decision function for SVM is: $f(x) = \sum_{i=1}^n \alpha_i y_i K(x_i, x) + b$

- **Condition**: The update for α_i depends on whether the sample satisfies: $y_i f(x_i) \geq 1$
  - If $y_i f(x_i) < 1$: The sample is misclassified or within the margin, so it contributes to the loss.
  - If $y_i f(x_i) \geq 1$: The sample satisfies the margin, so α_i only minimizes regularization.

Gradient Updates:

- If $y_i f(x_i) < 1$, update α_i to minimize the hinge loss and enforce the margin:
  $\alpha_i \leftarrow \alpha_i + \eta \left(C(1 - y_i f(x_i)) - \alpha_i \right)$
- If $y_i f(x_i) \geq 1$, update α_i to minimize regularization: $\alpha_i \leftarrow \alpha_i + \eta(-\alpha_i)$

Bias Term Update:

The bias term b ensures the decision boundary remains optimal:
$b \leftarrow b + \eta \sum_{i=1}^n (y_i - \sum_{j=1}^n \alpha_j y_j K(x_j, x_i))$

Steps for Implementation

1. **Initialize Parameters**:

- Set α (Lagrange multipliers) to zero for all samples.
- Initialize b (bias) to zero.

2. **Compute the Kernel Matrix**:

- Construct the kernel matrix K, where $K[i,j] = K(x_i, x_j)$.
- This precomputes pairwise similarities for efficiency.

3. **Perform Gradient Descent**:

- For each iteration and each sample:
  - Compute the decision function: $f(x_i) = \sum_{j=1}^{n} \alpha_j y_j K(x_j, x_i) + b$
  - Check the margin condition $y_i f(x_i) < 1$ and update α_i accordingly.
- After updating all α_i, adjust the bias term b.

4. **Stop Condition**:

- Iterate for a fixed number of steps (`num_iters`), or until convergence.

Computing the Kernel Matrix in Kernel SVM

What is the Kernel Matrix?

The kernel matrix K is an $n \times n$ matrix where each entry $K[i,j]$ represents the similarity between sample i and sample j in the input space. For a kernel function $K(x_i, x_j)$ (e.g., RBF kernel):

$$K[i,j] = K(x_i, x_j)$$

Why Do We Need the Kernel Matrix?

1. The kernel matrix allows us to compute similarities between all training samples in one step.
2. It encapsulates the "geometry" of the data in the transformed (implicit) feature space.

Mathematics Behind the Kernel Matrix

Entry-Wise Calculation

For every pair of samples x_i and x_j:

1. **Kernel Function**:
   - Compute $K(x_i, x_j)$ using the provided kernel function (e.g., RBF kernel):

$$K(x_i, x_j) = \exp\left(-\frac{\|x_i - x_j\|^2}{2\sigma^2}\right)$$

   - This measures the similarity between x_i and x_j.
2. **Matrix Construction**:
   - Build a symmetric matrix K where:
     - $K[i, j] = K(x_i, x_j)$
     - $K[j, i] = K(x_j, x_i)$ (symmetry holds for kernels like RBF).

Computational Complexity

- Computing K involves n^2 evaluations of the kernel function, where n is the number of samples.

Steps for Implementation

1. **Initialize the Kernel Matrix**:
   - Create an $n \times n$ matrix initialized to zeros.
2. **Compute Pairwise Kernel Values**:
   - Iterate over all pairs of samples i and j.
   - Compute the kernel function $K(x_i, x_j)$ and store the result in $K[i, j]$.
3. **Symmetry Optimization** (Optional):
   - Since $K[i, j] = K[j, i]$, you can reduce computation by calculating only the upper triangle of the matrix and mirroring it.

What to Do in Code

1. Create an empty matrix `K` of shape `(n_samples, n_samples)` initialized to zeros.
2. Loop over each pair of samples `i` and `j`:
   - Compute the kernel function for `X[i]` and `X[j]`.
   - Store the result in `K[i, j]`.
3. Optionally optimize using the symmetry of the matrix.

Gradient Descent Updates in Kernel SVM

Key Steps

1. Compute the Decision Function

The decision function determines how well the current model classifies a sample:

$$f(x_i) = \sum_{j=1}^{n} \alpha_j y_j K(x_j, x_i) + b$$

2. Check the Margin Condition

For each sample, check:

$$y_i f(x_i) < 1$$

- If `True` : The sample contributes to the hinge loss, and we update $\alpha[i]$.
- If `False` : The sample satisfies the margin, and we only adjust $\alpha[i]$ for regularization.

3. Update Rules

- **For** $\alpha[i]$:
  - If $y_i f(x_i) < 1$:

    $$\alpha_i \leftarrow \alpha_i + \eta\left(C \cdot (1 - y_i f(x_i)) - \alpha_i\right)$$

  - Otherwise:

    $$\alpha_i \leftarrow \alpha_i + \eta \cdot (-\alpha_i)$$

- **For Bias Term** b: Update the bias to minimize the overall loss:

$$b \leftarrow b + \eta \sum_{i=1}^{n} \left(y_i - \sum_{j=1}^{n} \alpha_j y_j K(x_j, x_i) \right)$$

4. Repeat for Multiple Iterations

Perform the updates over `num_iters` iterations to refine the parameters.

Steps for Implementation

1. **Outer Loop (Iterations)**:

   - Iterate `num_iters` times to refine the model.
2. **Inner Loop (Samples)**:
   - For each sample, compute the decision function $f(x_i)$.
   - Check the margin condition and update $\alpha[i]$.
3. **Bias Update**:
   - After updating all α, adjust the bias term b.

In [36]:
```
import numpy as np

# Define the Gaussian RBF Kernel
def rbf_kernel(x1, x2, sigma=1.0):
    """
    Compute the Gaussian RBF Kernel between two vectors.
    K(x1, x2) = exp(-||x1 - x2||^2 / (2 * sigma^2))
    """
    return np.exp(-(np.linalg.norm(x1-x2))**2/((2*sigma**2))) # Calculating the Gau

# SVM with Kernel Trick
class KernelSVM:
    def __init__(self, kernel, C=1.0, lr=0.001, num_iters=1000):
        self.kernel = kernel # saving the kernel
        self.C = C #Saving the regularization parameter
        self.lr=lr #Saving the learning rate
        self.num_iters=num_iters #Saving the maximum number of iterations

    def fit(self, X, y):
        """
        Train the SVM model using gradient descent.
        X: Input features (n_samples, n_features)
        y: Labels (-1 or 1)
        """
        n_samples = X.shape[0] #Number of samples is the rows of X
        self.alpha = np.zeros(n_samples) # Array of zeroes for Lagrange Multipliers
        self.bias= 0 #bias term set to 0

        assert len(X.shape) == 2, "X must be a 2D array"
        assert X.shape[0] == len(y), "Number of samples in X and y must match"

        matrix = np.zeros((n_samples,n_samples)) # The matrix of zeroes

        for i in range(n_samples):
            for j in range(i, n_samples):  # Start at `i` to compute only the upper
                matrix[i, j] = self.kernel(X[i], X[j]) # Calculating the kernel for
                matrix[j, i] = matrix[i, j]  # Exploit symmetry: The other half the

        # Gradient Descent
        for _ in range(self.num_iters):
            for i in range(n_samples):
                decision_value = np.sum(self.alpha*y*matrix[:,i])+self.bias
                final = y[i]*decision_value
                if final<1:
                    self.alpha[i]+=self.lr*(self.C*(1-final))-self.alpha[i] #Update
                else:
                    self.alpha[i]+=self.lr*(-self.alpha[i]) # Update of the alpha t

            self.bias += self.lr * np.sum(y - np.sum(self.alpha * y[:, None] * matr



    def predict(self, X_train, X_test):
        """
        Predict the class of each test sample.
        X_train: Training data (for computing kernel with test points)
        X_test: Test data
        """
        y_pred = []
        for x in X_test:
            prediction = 0
            for i in range(len(X_train)):
               prediction += self.alpha[i] * y_train[i] * self.kernel(X_train[i], x
            y_pred.append(np.sign(prediction + self.bias))
        return np.array(y_pred)

# Example usage
if __name__ == "__main__":
    # Example data
    X_train = np.array([[2, 3], [1, 1], [2, 2], [4, 5], [5, 6], [1, 0]])
    y_train = np.array([1, -1, -1, 1, 1, -1])

    # Define the kernel SVM with RBF kernel
    svm = KernelSVM(kernel=rbf_kernel, C=1.0, lr=0.001, num_iters=1000)

    # Train the model
    svm.fit(X_train, y_train)

    # Test the model on new data
    X_test = np.array([[2, 2], [3, 3], [5, 5], [0, 0]])
    y_pred = svm.predict(X_train, X_test)

    # Output the predictions
    print("Predictions:", y_pred)
```

```
Predictions: [1. 1. 1. 1.]
```

Changes SVM for Proper Predictions

In [43]:

```
import numpy as np

# Define the Gaussian RBF Kernel
def rbf_kernel(x1, x2, sigma=1.0):
    """
    Compute the Gaussian RBF Kernel between two vectors.
    K(x1, x2) = exp(-||x1 - x2||^2 / (2 * sigma^2))
    """
    return np.exp(-np.linalg.norm(x1 - x2)**2 / (2 * sigma**2))

# SVM with Kernel Trick
class KernelSVM:
    def __init__(self, kernel, C=1.0, lr=0.001, num_iters=1000):
        self.kernel = kernel  # Kernel function
        self.C = C  # Regularization parameter
        self.lr = lr  # Learning rate
        self.num_iters = num_iters  # Number of iterations

    def fit(self, X, y):
        """
        Train the SVM model using gradient descent.
        X: Input features (n_samples, n_features)
        y: Labels (-1 or 1)
        """
        n_samples = X.shape[0]
        self.alpha = np.zeros(n_samples)  # Lagrange multipliers
        self.bias = 0  # Bias term

        # Compute the kernel matrix
        matrix = np.zeros((n_samples, n_samples))
        for i in range(n_samples):
            for j in range(i, n_samples):
                matrix[i, j] = self.kernel(X[i], X[j])
                matrix[j, i] = matrix[i, j]  # Symmetry of the kernel

        # Gradient descent for alpha and bias
        for _ in range(self.num_iters):
            for i in range(n_samples):
                decision_value = np.sum(self.alpha * y * matrix[:, i]) + self.bias
                margin = y[i] * decision_value

                # Subgradient update
                if margin < 1:  # Misclassified or on the margin
                    self.alpha[i] += self.lr * (self.C - margin)
                else:  # Correct classification
                    self.alpha[i] += self.lr * (-self.alpha[i])

                # Enforce box constraints: 0 <= alpha_i <= C
                self.alpha[i] = max(0, min(self.alpha[i], self.C))

            # Update bias term (simple averaging over support vectors)
            support_vector_indices = np.where((self.alpha > 1e-5) & (self.alpha < s
            if len(support_vector_indices) > 0:
                self.bias = np.mean(
                    y[support_vector_indices]
                    - np.sum(
                        self.alpha * y[:, None] * matrix[:, support_vector_indices]
                        axis=0,
                    )
                )

    def predict(self, X_train, X_test):
        """
        Predict the class of each test sample.
```

```python
        X_train: Training data (for computing kernel with test points)
        X_test: Test data
        """
        y_pred = []
        for x in X_test:
            prediction = 0
            for i in range(len(X_train)):
                prediction += self.alpha[i] * y_train[i] * self.kernel(X_train[i],
            y_pred.append(np.sign(prediction + self.bias))
        return np.array(y_pred)

# Example usage
if __name__ == "__main__":
    # Example data
    X_train = np.array([[2, 3], [1, 1], [2, 2], [4, 5], [5, 6], [1, 0]])
    y_train = np.array([1, -1, -1, 1, 1, -1])

    # Define the kernel SVM with RBF kernel
    svm = KernelSVM(kernel=rbf_kernel, C=1.0, lr=0.001, num_iters=1000)

    # Train the model
    svm.fit(X_train, y_train)

    # Test the model on new data
    X_test = np.array([[2, 2], [3, 3], [5, 5], [0, 0]])
    y_pred = svm.predict(X_train, X_test)

    # Output the predictions
    print("Predictions:", y_pred)
```

```
Predictions: [-1.  1.  1. -1.]
```

Understanding the Predictions

Predictions Output

The model output: Predictions: [-1. 1. 1. -1.]

This means that for the given test dataset, the model predicts **class 1** for some test points and **class -1** for others.

What Does It Mean?

1. **Class Labels**:
   - Your model is a binary classifier, trained with labels `-1` and `1`.
   - The predictions `[-1. 1. 1. -1.]` indicate that the model predicts **some to class -1 and some to class 1**.
2. **Decision Boundary**:

- The SVM decision boundary is designed to separate the two classes based on the training data.
- A prediction of `1` for a test sample means that the sample lies **on or within the region classified as class 1**.

```
In [ ]:
```