Support Vector Machines (SVM) - Extended Breakdown with Kernels and Weighted SVM

---

1. In-Depth and Specific Intuitive Understanding

What is a Support Vector Machine (SVM)?

Support Vector Machines (SVMs) are **supervised learning models** used for classification and regression. They aim to **find the optimal decision boundary** (hyperplane) that **maximizes the margin** between different classes.

Key Idea

- SVM finds a **hyperplane** that best separates the data into classes.
- The **margin** is the distance between the hyperplane and the closest data points (support vectors).
- The goal is to **maximize this margin** for better generalization.
- When data is **not linearly separable**, SVMs use the **kernel trick** to transform it into a higher-dimensional space.

For **linearly separable data**, SVM optimizes:

$$\text{Maximize } \frac{1}{||w||} \quad \text{subject to } y_i(w \cdot x_i + b) \geq 1$$

where:

- w is the normal vector of the hyperplane.
- b is the bias term.
- x_i are the training examples.
- y_i are the class labels ($+1$ or -1).

For **non-linearly separable data**, SVM uses the **Kernel Trick** to map data to a higher-dimensional space where it becomes linearly separable.

2. When SVM is Used and When It Should Be Avoided

✅ When to Use SVM

- When the dataset has **clear margins of separation**.
- When **the number of features is large** (SVM performs well in high-dimensional spaces).
- When **the dataset is small to medium-sized** (SVM scales poorly for large datasets).
- When **classification requires robustness to outliers**.

❌ When to Avoid SVM

- If **the dataset is very large**, training can be slow.
- If **the number of features is much greater than the number of samples**, SVM may overfit.
- If **data is highly imbalanced**, SVM may struggle (alternative: use **Weighted SVM**).
- If **choosing the correct kernel is difficult**, results can be poor.

3. When It Fails to Converge and How to Avoid That

When SVM Fails

- **Poor choice of kernel function** (data remains non-separable).
- **High dimensionality with few samples** (overfitting).
- **Bad regularization parameter** C (too small → underfitting, too large → overfitting).
- **Linearly inseparable data without kernels**.

How to Ensure Convergence

✅ **Normalize or standardize features** (SVM is sensitive to scaling).
✅ **Use an appropriate kernel (Linear, RBF, Polynomial)** for non-linear data.
✅ **Tune regularization parameter** C using cross-validation.
✅ **Use Kernel Approximation (e.g., Nyström method)** for large datasets.

When SVM Always Converges

- When the **data is linearly separable**.
- When **the correct kernel is chosen**.
- When **a proper stopping criterion** is used in optimization.

4. Advantages and Disadvantages

Advantages

✅ Effective in high-dimensional spaces.

✅ Works well with a clear margin of separation.

✅ Can handle non-linear decision boundaries using kernels.

✅ Robust to outliers when using the soft-margin approach.

Disadvantages

❌ Computationally expensive for large datasets.

❌ Choosing the correct kernel function is non-trivial.

❌ Does not provide probability estimates by default.

❌ Requires feature scaling for optimal performance.

5. Intuitive Algorithm / Pseudo Code

1. **Transform input data** (if using kernels).
2. **Optimize the SVM objective function:**

$$\min_{w,b} \frac{1}{2}||w||^2 + C \sum \xi_i$$

   subject to:

$$y_i(w \cdot x_i + b) \geq 1 - \xi_i, \quad \xi_i \geq 0$$

3. **Solve using Quadratic Programming (QP) or Gradient Descent.**
4. **Classify new points based on sign of** $w \cdot x + b$.

plaintext Copy Edit

```plaintext
1. Select kernel function (linear, polynomial, RBF)
2. Compute optimal hyperplane by solving optimization problem
3. Use support vectors to define decision boundary
4. Classify new points based on margin
```

6. Mathematical and Logical Breakdown

Step 1: Define the Optimization Problem

SVM minimizes:

$$\frac{1}{2}||w||^2 + C\sum\xi_i$$

where:

- $||w||^2$ ensures a **maximum margin.**
- C controls the **trade-off between maximizing margin and minimizing classification error**.
- ξ_i are **slack variables** for misclassified points (soft margin SVM).

Step 2: Compute Decision Boundary

For a new point x, classify using:

$$f(x) = w \cdot x + b$$

If $f(x) \geq 0$, classify as **positive.**
If $f(x) < 0$, classify as **negative.**

Step 3: Use the Kernel Trick (for Non-Linear Data)

Instead of computing $w \cdot x$ directly, use a kernel function $K(x_i, x_j)$ to map data to a higher-dimensional space:

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

where $\phi(x)$ is the feature transformation.

Common kernels:

- **Linear Kernel**: $K(x, y) = x \cdot y$
- **Polynomial Kernel**: $K(x, y) = (x \cdot y + c)^d$
- **RBF (Gaussian) Kernel**: $K(x, y) = \exp\left(-\frac{||x-y||^2}{2\sigma^2}\right)$

Fully Commented Manual Implementation of SVM with Kernels

```python
import numpy as np

class SVM:
    def __init__(self, learning_rate=0.001, lambda_param=0.01, epochs=1000, kernel="linear"):
        """
        Initialize SVM with hyperparameters.
        :param learning_rate: Step size for gradient descent updates.
        :param lambda_param: Regularization parameter (controls margin size and error tolerance).
        :param epochs: Number of training iterations.
        :param kernel: The kernel type (linear, polynomial, or rbf).
        """
        self.learning_rate = learning_rate
        self.lambda_param = lambda_param
        self.epochs = epochs
        self.kernel = kernel
        self.weights = None
        self.bias = None
        self.support_vectors = None  # Stores support vectors

    def _kernel(self, x, y):
        """
        Compute the kernel function between two feature vectors.
        :param x: First feature vector.
        :param y: Second feature vector.
        :return: Kernel function result.
        """
        if self.kernel == "linear":
            return np.dot(x, y)  # Standard dot product for linear kernel
        elif self.kernel == "polynomial":
            return (np.dot(x, y) + 1) ** 2  # Polynomial kernel with degree=2
        elif self.kernel == "rbf":
            gamma = 0.1  # Gamma hyperparameter for RBF kernel
            return np.exp(-gamma * np.linalg.norm(x - y) ** 2)  # RBF kernel (Gaussian)
        else:
            raise ValueError("Unsupported kernel type. Choose 'linear', 'polynomial', or 'rbf'.")

```

```python
    def fit(self, X, y):
        """
        Train the SVM using gradient descent.
        :param X: Training feature matrix (shape: m x n).
        :param y: Target labels (shape: m).
        """
        m, n = X.shape  # Number of samples (m) and features (n)
        self.weights = np.zeros(n)  # Initialize weight vector to zeros
        self.bias = 0  # Initialize bias to zero

        y = np.where(y == 0, -1, 1)  # Convert labels from (0,1) to (-1,+1) for SVM

        for _ in range(self.epochs):  # Loop over epochs
            for i in range(m):  # Loop over each training sample
                kernel_output = self._kernel(X[i], X[i])  # Compute kernel transformation

                # Check if the sample satisfies the margin condition
                condition = y[i] * (np.dot(self.weights, kernel_output) + self.bias) >= 1
                if condition:
                    # If correct classification: Apply only L2 regularization (reduce weights)
                    self.weights -= self.learning_rate * (2 * self.lambda_param * self.weights)
                else:
                    # If misclassified: Update weights and bias
                    self.weights -= self.learning_rate * (2 * self.lambda_param * self.weights - y[i] * kernel_output)
                    self.bias -= self.learning_rate * y[i]

    def predict(self, X):
        """
        Predict class labels based on the sign of the decision function.
        :param X: Feature matrix for prediction.
        :return: Predicted class labels (-1 or 1).
        """
        predictions = []  # Store predictions
        for x in X:
            kernel_output = np.array([self._kernel(x, xi) for xi in X])  # Apply kernel function
            prediction = np.sign(np.dot(kernel_output, self.weights) + self.bias)  # Decision function
            predictions.append(prediction)

        return np.array(predictions)  # Return predictions as a NumPy array
```

Memory updated

Fully Commented Manual Implementation of Weighted SVM

What is Weighted SVM?

Weighted SVM modifies the standard SVM by assigning **different penalties (weights) to different classes**. This is useful for **imbalanced datasets**, where one class has significantly more samples than another.

In **Weighted SVM**, we modify the objective function:

$$\min_{w,b} \frac{1}{2}||w||^2 + C \sum w_i \xi_i$$

where:

- w_i is the **weight assigned to each class** (higher for minority class).
- C controls the trade-off between maximizing margin and minimizing classification errors.
- ξ_i are slack variables for misclassified points.

Fully Commented Manual Implementation of Weighted SVM

```python
import numpy as np

class WeightedSVM:
    def __init__(self, learning_rate=0.001, lambda_param=0.01, epochs=1000, kernel="linear", class_weights=None):
        """
        Initialize Weighted SVM with hyperparameters.
        :param learning_rate: Step size for gradient descent updates.
        :param lambda_param: Regularization parameter (controls margin size and error tolerance).
        :param epochs: Number of training iterations.
        :param kernel: The kernel type (linear, polynomial, or rbf).
        :param class_weights: Dictionary {class_label: weight} for handling class imbalance.
        """
        self.learning_rate = learning_rate
        self.lambda_param = lambda_param
        self.epochs = epochs
        self.kernel = kernel
        self.weights = None
        self.bias = None
        self.class_weights = class_weights if class_weights else {1: 1, -1: 1}  # Default weight is 1 for both classes

    def _kernel(self, x, y):
        """
        Compute the kernel function between two feature vectors.
        :param x: First feature vector.
        :param y: Second feature vector.
        :return: Kernel function result.
        """
        if self.kernel == "linear":
            return np.dot(x, y)  # Standard dot product for linear kernel
        elif self.kernel == "polynomial":
            return (np.dot(x, y) + 1) ** 2  # Polynomial kernel with degree=2
        elif self.kernel == "rbf":
            gamma = 0.1  # Gamma hyperparameter for RBF kernel
            return np.exp(-gamma * np.linalg.norm(x - y) ** 2)  # RBF kernel (Gaussian)
        else:
            raise ValueError("Unsupported kernel type. Choose 'linear', 'polynomial', or 'rbf'.")

```

```python
    def fit(self, X, y):
        """
        Train the Weighted SVM using gradient descent.
        :param X: Training feature matrix (shape: m x n).
        :param y: Target labels (shape: m).
        """
        m, n = X.shape  # Number of samples (m) and features (n)
        self.weights = np.zeros(n)  # Initialize weight vector to zeros
        self.bias = 0  # Initialize bias to zero
        # Convert labels from (0,1) to (-1,+1) for SVM
        y = np.where(y == 0, -1, 1)
        for _ in range(self.epochs):  # Loop over epochs
            for i in range(m):  # Loop over each training sample
                kernel_output = self._kernel(X[i], X[i])  # Compute kernel transformation

                # Get the class weight for the current sample
                class_weight = self.class_weights[y[i]]

                # Check if the sample satisfies the margin condition
                condition = y[i] * (np.dot(self.weights, kernel_output) + self.bias) >= 1
                if condition:
                    # If correctly classified: Apply only L2 regularization (reduce weights)
                    self.weights -= self.learning_rate * (2 * self.lambda_param * self.weights)
                else:
                    # If misclassified: Apply class-weighted updates to weights and bias
                    self.weights -= self.learning_rate * (2 * self.lambda_param * self.weights - class_weight * y[i] * kernel_output)
                    self.bias -= self.learning_rate * class_weight * y[i]

    def predict(self, X):
        """
        Predict class labels based on the sign of the decision function.
        :param X: Feature matrix for prediction.
        :return: Predicted class labels (-1 or 1).
        """
        predictions = []  # Store predictions
        for x in X:
            kernel_output = np.array([self._kernel(x, xi) for xi in X])  # Apply kernel function to data
            prediction = np.sign(np.dot(kernel_output, self.weights) + self.bias)  # Decision function
            predictions.append(prediction)

        return np.array(predictions)  # Return predictions as a NumPy array
```

Fully Commented Code for Unweighted SVM in Scikit-Learn

python Copy Edit

```python
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_classification

# Step 1: Generate a synthetic dataset
# - n_samples=200: Generate 200 data points.
# - n_features=2: Use 2 features (for visualization).
# - random_state=42: Ensure reproducibility.
X, y = make_classification(n_samples=200, n_features=2, random_state=42)

# Step 2: Split the dataset into training and testing sets
# - test_size=0.2: 20% of the data is used for testing.
# - random_state=42: Keep results consistent.
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_sta

# Step 3: Initialize an unweighted SVM model with a linear kernel
# - kernel='linear': Use a linear decision boundary.
# - C=1.0: Regularization parameter (higher C = less margin, more focus on misclass
model = SVC(kernel="linear", C=1.0)

# Step 4: Train the SVM model on the training data
model.fit(X_train, y_train)

# Step 5: Predict class labels for the test set
y_pred = model.predict(X_test)

# Step 6: Compute and print accuracy
accuracy = model.score(X_test, y_test)  # Accuracy = (correct predictions / total p
print("Unweighted SVM Accuracy:", accuracy)
```

Example Usage: Training Weighted SVM

python Copy Edit

```python
# Generate imbalanced dataset
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split

X, y = make_classification(n_samples=200, n_features=2, weights=[0.9, 0.1], random_

# Split into training (80%) and testing (20%)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_sta

# Define class weights (assign higher weight to minority class)
class_weights = {1: 5, -1: 1}

# Train the Weighted SVM
weighted_svm = WeightedSVM(kernel="linear", class_weights=class_weights)
weighted_svm.fit(X_train, y_train)

# Predict on test data
y_pred = weighted_svm.predict(X_test)

# Print accuracy
accuracy = np.mean(y_pred == y_test)
print("Weighted SVM Accuracy:", accuracy)
```

Final Summary

- SVM maximizes the margin for classification.
- Kernels allow non-linear classification.
- Weighted SVM balances class imbalance.
- SVM is best for small to medium datasets.