Mathematics Behind Support Vector Machine

Kato Paul	2023/U/MMU/BCS/00104
Karamagi Henry Atuhaire	2023/U/MMU/BCS/00523
Beinomugisha Wyclif	2023/U/MMU/BCS/01681
Namony Kenneth	2023/U/MMU/BCS/01668
Mpagi Derrick Brair	2023/U/MMU/BCS/00101

February 2025

1 S.V.M

upport Vector Machine (SVM) Mathematical Formulas

1. **Equation of the Hyperplane**

$$w \cdot x + b = 0$$

This equation represents the decision boundary that separates two classes.

2. **Classifier Function**

$$h(x) = \{ +1, if w \cdot x + b \geq 0 - 1, if w \cdot x + b < 0$$

This function determines the class of a given data point.

3. **Margin Optimization (Hard Margin SVM)**

$$\min_{w,b} \frac{1}{2}||w||^2$$

Subject to:

$$y_i(w \cdot x_i + b) \geq 1, \quad \forall i$$

This ensures that all points are correctly classified and lie outside the margin.

4. **Soft Margin SVM (Handling Overlapping Classes)**

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

Subject to:

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

The slack variable ξ_i allows some misclassification, controlled by the regularization parameter C.

5. **Lagrangian Dual Formulation**

$$\mathcal{L}(w, b, \alpha) = \sum_i \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j (x_i \cdot x_j)$$

Subject to:

$$\sum_i \alpha_i y_i = 0, \quad 0 \leq \alpha_i \leq C$$

This formulation allows solving SVM efficiently using quadratic programming.

6. **Kernel Trick (Handling Non-Linearity)**

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

This transformation maps data to a higher-dimensional space for better separation.

7. **Common Kernel Functions** - Linear Kernel:

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

- Polynomial Kernel:

$$K(x_i, x_j) = (x_i \cdot x_j + c)^d$$

- **Radial Basis Function (RBF) Kernel**:

$$K(x_i, x_j) = \exp(-\gamma ||x_i - x_j||^2)$$

These formulas define how SVMs work mathematically, including margin maximization, dual optimization, and kernel transformations.

svmf

February 27, 2025

[2]:
```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, confusion_matrix

# Load the Iris dataset
def load_iris_data(filepath):
    """Load and prepare the Iris dataset."""
    # Read the data
    data = pd.read_csv("C:/Users/PRO. BUNOOTI/Desktop/OUR WORK/Iris.csv")

    # Extract features (all columns except Id and Species)
    X = data.iloc[:, 1:5].values

    # Extract target and convert to numerical values
    # For SVM implementation, we'll convert to binary labels (-1, 1)
    # We'll first handle a binary classification problem: setosa vs non-setosa
    y = data.iloc[:, 5].values

    y_binary = np.where(y == 'Iris-setosa', 1, -1)

    return X, y_binary, y

# Kernel functions as described in the SVM document
def linear_kernel(x1, x2):
    """Compute linear kernel: K(x_i, x_j) = x_i · x_j"""
    return np.dot(x1, x2)

def polynomial_kernel(x1, x2, degree=3, c=1.0):
    """Compute polynomial kernel: K(x_i, x_j) = (x_i · x_j + c)^d"""
    return (np.dot(x1, x2) + c) ** degree

def rbf_kernel(x1, x2, gamma=0.1):
    """Compute RBF kernel: K(x_i, x_j) = exp(- ||x_i - x_j||²)"""
    return np.exp(-gamma * np.sum((x1 - x2) ** 2))
```

```
class SVM:
    def __init__(self, kernel=linear_kernel, C=1.0, tol=1e-3, max_iter=100):
        """
        Implementation of Support Vector Machine using SMO algorithm

        Parameters:
        kernel: the kernel function to use
        C: regularization parameter (for soft margin)
        tol: tolerance for stopping criterion
        max_iter: maximum number of iterations
        """
        self.kernel = kernel
        self.C = C
        self.tol = tol
        self.max_iter = max_iter
        self.alpha = None
        self.b = 0
        self.support_vectors = None
        self.support_vector_labels = None
        self.support_vector_indices = None

    def fit(self, X, y):
        """
        Fit the SVM model according to the training data using SMO algorithm

        Parameters:
        X: training features
        y: training labels (-1, 1)
        """
        n_samples, n_features = X.shape

        # Initialize alphas (Lagrange multipliers)
        self.alpha = np.zeros(n_samples)

        # Precompute kernel matrix for efficiency
        self.kernel_matrix = np.zeros((n_samples, n_samples))
        for i in range(n_samples):
            for j in range(n_samples):
                self.kernel_matrix[i, j] = self.kernel(X[i], X[j])

        # Simplified SMO algorithm
        iter_count = 0
        while iter_count < self.max_iter:
            alpha_changed = 0

            for i in range(n_samples):
```

```
                # Calculate error for sample i
                E_i = self._decision_function_internal(i, X, y) - y[i]

                # Check if example violates KKT conditions
                if ((y[i] * E_i < -self.tol and self.alpha[i] < self.C) or
                    (y[i] * E_i > self.tol and self.alpha[i] > 0)):

                    # Randomly select second sample j  i
                    j = i
                    while j == i:
                        j = np.random.randint(0, n_samples)

                    # Calculate error for sample j
                    E_j = self._decision_function_internal(j, X, y) - y[j]

                    # Save old alphas
                    alpha_i_old = self.alpha[i]
                    alpha_j_old = self.alpha[j]

                    # Compute bounds L and H
                    if y[i] != y[j]:
                        L = max(0, self.alpha[j] - self.alpha[i])
                        H = min(self.C, self.C + self.alpha[j] - self.alpha[i])
                    else:
                        L = max(0, self.alpha[i] + self.alpha[j] - self.C)
                        H = min(self.C, self.alpha[i] + self.alpha[j])

                    if L == H:
                        continue

                    # Compute eta
                    eta = 2.0 * self.kernel_matrix[i, j] - self.
↪kernel_matrix[i, i] - self.kernel_matrix[j, j]
                    if eta >= 0:
                        continue

                    # Update alpha_j
                    self.alpha[j] = alpha_j_old - (y[j] * (E_i - E_j)) / eta

                    # Clip alpha_j
                    self.alpha[j] = min(H, self.alpha[j])
                    self.alpha[j] = max(L, self.alpha[j])

                    if abs(self.alpha[j] - alpha_j_old) < 1e-5:
                        continue

                    # Update alpha_i
```

```
                    self.alpha[i] = alpha_i_old + y[i] * y[j] * (alpha_j_old -␣
↪self.alpha[j])

                    # Compute b
                    b1 = self.b - E_i - y[i] * (self.alpha[i] - alpha_i_old) *␣
↪self.kernel_matrix[i, i] - \
                         y[j] * (self.alpha[j] - alpha_j_old) * self.
↪kernel_matrix[i, j]

                    b2 = self.b - E_j - y[i] * (self.alpha[i] - alpha_i_old) *␣
↪self.kernel_matrix[i, j] - \
                         y[j] * (self.alpha[j] - alpha_j_old) * self.
↪kernel_matrix[j, j]

                    if 0 < self.alpha[i] < self.C:
                        self.b = b1
                    elif 0 < self.alpha[j] < self.C:
                        self.b = b2
                    else:
                        self.b = (b1 + b2) / 2

                    alpha_changed += 1

            if alpha_changed == 0:
                iter_count += 1
            else:
                iter_count = 0

        # Extract support vectors
        sv_indices = np.where(self.alpha > 1e-5)[0]
        self.support_vectors = X[sv_indices]
        self.support_vector_labels = y[sv_indices]
        self.support_vector_indices = sv_indices
        self.alpha = self.alpha[sv_indices]

        print(f"Number of support vectors: {len(self.support_vectors)}")

        return self

    def _decision_function_internal(self, i, X, y):
        """Internal decision function used during training"""
        return np.sum(self.alpha * y * self.kernel_matrix[i]) + self.b

    def decision_function(self, X):
        """
        Compute the decision function for samples in X
```

```
        Parameters:
        X: test samples

        Returns:
        Array of decisions (scores)
        """
        n_samples = X.shape[0]
        decision = np.zeros(n_samples)

        for i in range(n_samples):
            decision[i] = self.b
            for alpha, sv, sv_y in zip(self.alpha, self.support_vectors, self.
 ↪support_vector_labels):
                decision[i] += alpha * sv_y * self.kernel(X[i], sv)

        return decision

    def predict(self, X):
        """
        Predict class labels for samples in X

        Parameters:
        X: test samples

        Returns:
        Array of predicted class labels (-1 or 1)
        """
        return np.sign(self.decision_function(X))

# One-vs-Rest SVM for multiclass classification
class OneVsRestSVM:
    def __init__(self, kernel=linear_kernel, C=1.0):
        """
        One-vs-Rest SVM for multiclass classification

        Parameters:
        kernel: the kernel function to use
        C: regularization parameter
        """
        self.kernel = kernel
        self.C = C
        self.models = {}
        self.classes = None

    def fit(self, X, y):
        """
        Fit One-vs-Rest SVM model
```

```
        Parameters:
        X: training features
        y: training labels (can be multiclass)
        """
        self.classes = np.unique(y)

        # Train one SVM for each class
        for cls in self.classes:
            print(f"Training SVM for class: {cls}")
            # Create binary labels (1 for current class, -1 for rest)
            y_binary = np.where(y == cls, 1, -1)

            # Create and train SVM model
            svm = SVM(kernel=self.kernel, C=self.C)
            svm.fit(X, y_binary)

            # Store the model
            self.models[cls] = svm

        return self

    def predict(self, X):
        """
        Predict class labels for samples in X

        Parameters:
        X: test samples

        Returns:
        Array of predicted class labels
        """
        n_samples = X.shape[0]
        # Decision matrix: rows = samples, columns = classes
        decision = np.zeros((n_samples, len(self.classes)))

        # Calculate decision scores for each model
        for i, cls in enumerate(self.classes):
            decision[:, i] = self.models[cls].decision_function(X)

        # Return class with highest decision score
        return self.classes[np.argmax(decision, axis=1)]

# Function to evaluate model performance
def evaluate_model(model, X_train, X_test, y_train, y_test, model_name="Model"):
    """
    Evaluate model performance on both training and test sets
```

```
    Parameters:
    model: trained SVM model
    X_train, X_test: training and test features
    y_train, y_test: training and test labels
    model_name: name of the model for printing
    """
    # Evaluate on training data
    y_train_pred = model.predict(X_train)
    train_accuracy = accuracy_score(y_train, y_train_pred)

    # Evaluate on test data
    y_test_pred = model.predict(X_test)
    test_accuracy = accuracy_score(y_test, y_test_pred)

    print(f"{model_name} - Training Accuracy: {train_accuracy:.4f}")
    print(f"{model_name} - Test Accuracy: {test_accuracy:.4f}")

    return y_test_pred

# Function to visualize decision boundaries for 2D data
def plot_decision_boundary(model, X_train, X_test, y_train, y_test,␣
 ↪title="Decision Boundary"):
    """
    Plot decision boundary for a 2D SVM model and show train/test data

    Parameters:
    model: trained SVM model
    X_train, X_test: training and test features
    y_train, y_test: training and test labels
    title: plot title
    """
    # Combine data for mesh grid boundaries
    X_combined = np.vstack((X_train, X_test))

    # Create a mesh grid
    x_min, x_max = X_combined[:, 0].min() - 1, X_combined[:, 0].max() + 1
    y_min, y_max = X_combined[:, 1].min() - 1, X_combined[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.02),
                         np.arange(y_min, y_max, 0.02))

    # Get predictions for all mesh grid points
    Z = model.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    # Plot decision boundary and points
    plt.figure(figsize=(12, 10))
```

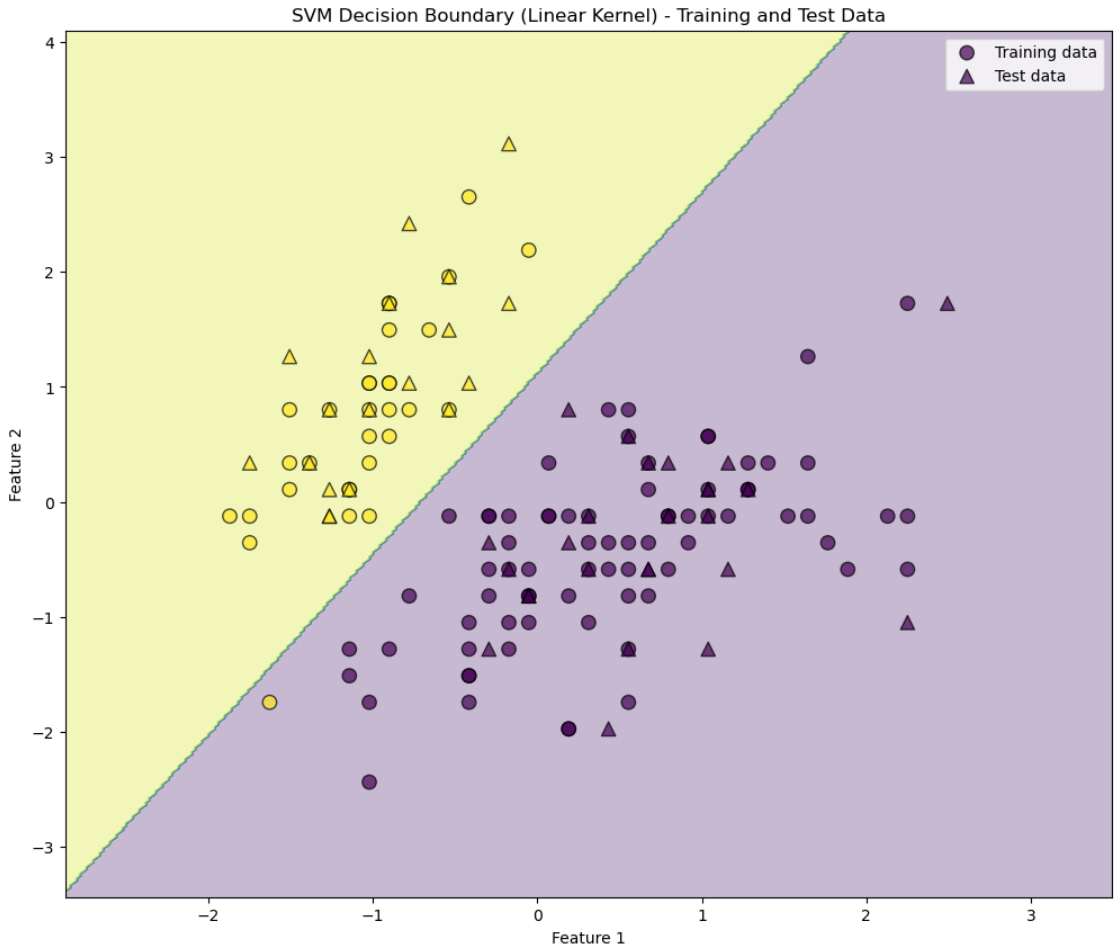
```
    plt.contourf(xx, yy, Z, alpha=0.3)

    # Plot training points
    plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train, edgecolors='k',
                marker='o', s=80, alpha=0.7, label='Training data')

    # Plot test points with different marker
    plt.scatter(X_test[:, 0], X_test[:, 1], c=y_test, edgecolors='k',
                marker='^', s=80, alpha=0.7, label='Test data')

    plt.title(title)
    plt.xlabel("Feature 1")
    plt.ylabel("Feature 2")
    plt.legend()
    plt.show()

# Main function
def main():
    # Load the Iris dataset
    X, y_binary, y_original = load_iris_data("Iris.csv")

    # Scale features
    scaler = StandardScaler()
    X_scaled = scaler.fit_transform(X)

    # Split data into training and testing sets
    X_train, X_test, y_train_binary, y_test_binary = train_test_split(
        X_scaled, y_binary, test_size=0.3, random_state=42)

    # For multiclass classification
    _, _, y_train_multi, y_test_multi = train_test_split(
        X_scaled, y_original, test_size=0.3, random_state=42)

    print("==== Binary Classification: Setosa vs Non-Setosa ====")

    # Train SVM model with linear kernel
    print("Training SVM with Linear Kernel...")
    svm_linear = SVM(kernel=linear_kernel, C=1.0)
    svm_linear.fit(X_train, y_train_binary)

    # Evaluate the model on both training and test data
    print("\nEvaluating Linear Kernel SVM:")
    y_pred_linear = evaluate_model(svm_linear, X_train, X_test,
                                   y_train_binary, y_test_binary,
                                   "Linear Kernel SVM")

    # Train SVM model with RBF kernel
```

```
    print("\nTraining SVM with RBF Kernel...")
    svm_rbf = SVM(kernel=rbf_kernel, C=1.0)
    svm_rbf.fit(X_train, y_train_binary)

    # Evaluate the model on both training and test data
    print("\nEvaluating RBF Kernel SVM:")
    y_pred_rbf = evaluate_model(svm_rbf, X_train, X_test,
                                y_train_binary, y_test_binary,
                                "RBF Kernel SVM")

    # For visualization, let's use only the first two features
    X_2d = X_scaled[:, :2]
    X_train_2d, X_test_2d, y_train_binary_2d, y_test_binary_2d =␣
↪train_test_split(
        X_2d, y_binary, test_size=0.3, random_state=42)

    # Train 2D model for visualization
    print("\nTraining 2D SVM for Visualization...")
    svm_2d = SVM(kernel=linear_kernel, C=1.0)
    svm_2d.fit(X_train_2d, y_train_binary_2d)

    # Evaluate 2D model
    print("\nEvaluating 2D SVM:")
    y_pred_2d = evaluate_model(svm_2d, X_train_2d, X_test_2d,
                               y_train_binary_2d, y_test_binary_2d,
                               "2D Linear Kernel SVM")

    # Plot decision boundary showing both training and test data
    plot_decision_boundary(svm_2d, X_train_2d, X_test_2d,
                           y_train_binary_2d, y_test_binary_2d,
                           "SVM Decision Boundary (Linear Kernel) - Training and␣
↪Test Data")

    print("\n==== Multiclass Classification: All Iris Species ====")

    # Train One-vs-Rest SVM for multiclass classification
    print("Training One-vs-Rest SVM with RBF Kernel...")
    ovr_svm = OneVsRestSVM(kernel=rbf_kernel, C=1.0)
    ovr_svm.fit(X_train, y_train_multi)

    # Evaluate the multiclass model on both training and test data
    print("\nEvaluating One-vs-Rest SVM:")
    # Training accuracy
    y_train_pred_multi = ovr_svm.predict(X_train)
    train_accuracy_multi = accuracy_score(y_train_multi, y_train_pred_multi)
    print(f"One-vs-Rest SVM - Training Accuracy: {train_accuracy_multi:.4f}")
```

```
    # Test accuracy
    y_test_pred_multi = ovr_svm.predict(X_test)
    test_accuracy_multi = accuracy_score(y_test_multi, y_test_pred_multi)
    print(f"One-vs-Rest SVM - Test Accuracy: {test_accuracy_multi:.4f}")

    # Confusion matrix for test data
    cm = confusion_matrix(y_test_multi, y_test_pred_multi)
    print("\nConfusion Matrix (Test Data):")
    print(cm)

    # Plot confusion matrix
    plt.figure(figsize=(8, 6))
    plt.imshow(cm, interpolation='nearest', cmap=plt.cm.Blues)
    plt.title('Confusion Matrix')
    plt.colorbar()
    classes = np.unique(y_original)
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    # Add text annotations in the confusion matrix
    for i in range(len(classes)):
        for j in range(len(classes)):
            plt.text(j, i, str(cm[i, j]), horizontalalignment="center",␣
 ↪color="white" if cm[i, j] > cm.max() / 2 else "black")

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
    plt.show()

    print("\nModel Evaluation Complete!")

if __name__ == "__main__":
    main()
```

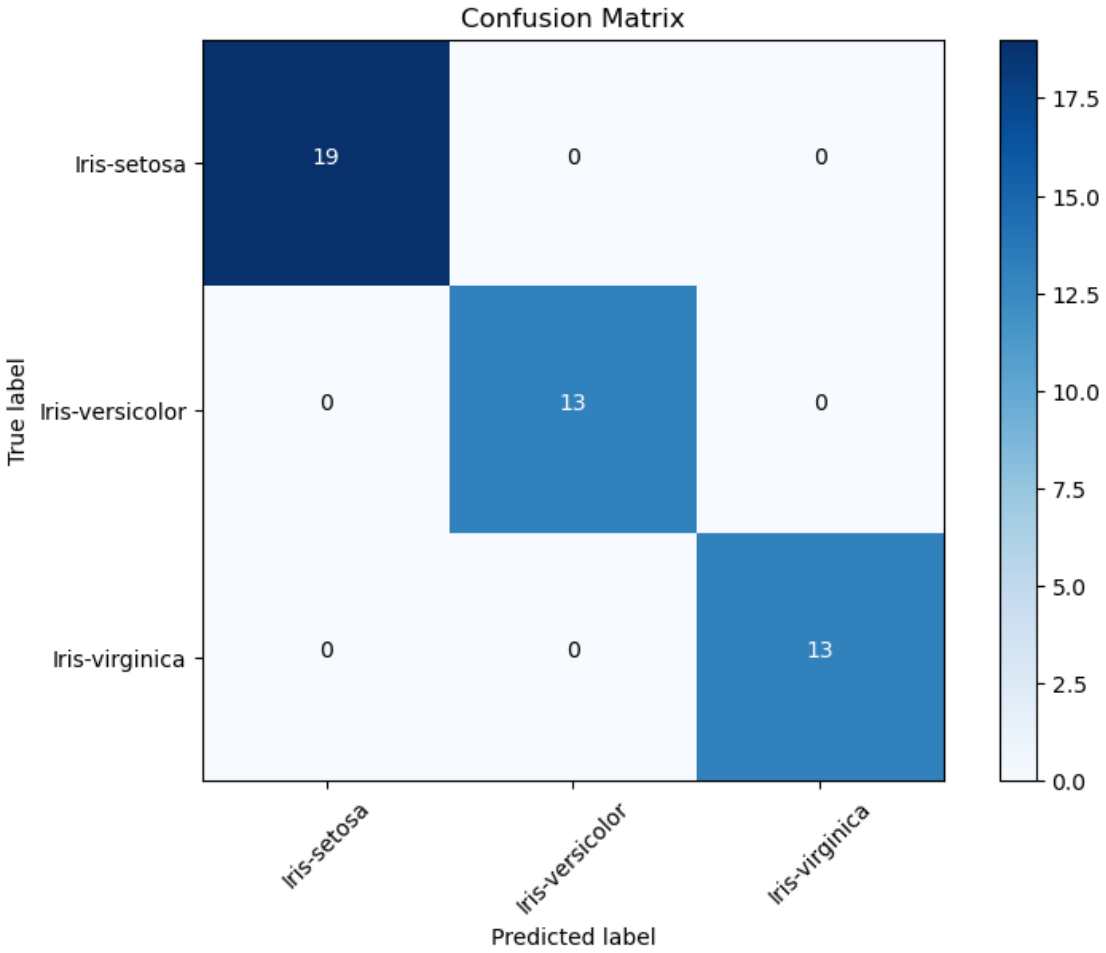
```
==== Binary Classification: Setosa vs Non-Setosa ====
Training SVM with Linear Kernel…
Number of support vectors: 4

Evaluating Linear Kernel SVM:
Linear Kernel SVM - Training Accuracy: 1.0000
Linear Kernel SVM - Test Accuracy: 1.0000

Training SVM with RBF Kernel…
Number of support vectors: 10

Evaluating RBF Kernel SVM:
```

```
RBF Kernel SVM - Training Accuracy: 1.0000
RBF Kernel SVM - Test Accuracy: 1.0000

Training 2D SVM for Visualization…
Number of support vectors: 11

Evaluating 2D SVM:
2D Linear Kernel SVM - Training Accuracy: 0.9905
2D Linear Kernel SVM - Test Accuracy: 1.0000
```




```
==== Multiclass Classification: All Iris Species ====
Training One-vs-Rest SVM with RBF Kernel…
Training SVM for class: Iris-setosa
Number of support vectors: 10
Training SVM for class: Iris-versicolor
Number of support vectors: 49
Training SVM for class: Iris-virginica
```

```
Number of support vectors: 41

Evaluating One-vs-Rest SVM:
One-vs-Rest SVM - Training Accuracy: 0.9524
One-vs-Rest SVM - Test Accuracy: 1.0000

Confusion Matrix (Test Data):
[[19  0  0]
 [ 0 13  0]
 [ 0  0 13]]
```




```
Model Evaluation Complete!
```

[ ]:

Mathematical Implementation of Logistic Rgression

Namonye kenneth	2023/U/MMU/BCS/01668
Karamagi Henry Atuhaire	2023/U/MMU/BCS/00523
Mpagi Derrick Brair	2023/U/MMU/BCS/00101
Beinomugisha wyclif	2023/U/MMU/BCS/01681
Kato Paul	2023/U/MMU/BCS/00104

February 2025

1 Introduction

Logistic Regression: is a statistical and machine learning technique used for binary classification problems, where the goal is to predict one of two possible outcomes (e.g., yes/no, true/false, 0/1). Despite its name, logistic regression is a classification algorithm, not a regression algorithm. It models the probability of a binary outcome using a logistic function.

Here's a step-by-step explanation of the mathematical implementation of logistic regression:

1. **Problem Setup**

- Let X be the input features (independent variables), and y be the binary output (dependent variable) such that $y \in \{0, 1\}$. - The goal is to model the probability $P(y = 1|X)$, i.e., the probability that $y = 1$ given the input features X.

2. **Logistic Function (Sigmoid Function)**

The logistic regression model uses the sigmoid function to map predicted values to probabilities. The sigmoid function is defined as:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Where: - z is a linear combination of the input features X and model parameters (weights) θ. - $\sigma(z)$ outputs a value between 0 and 1, which can be interpreted as a probability.

3. **Linear Combination of Inputs**

The linear combination z is computed as:

$$z = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots + \theta_n x_n$$

Where: - θ_0 is the **bias term** (intercept). - $\theta_1, \theta_2, \ldots, \theta_n$ are the **weights** corresponding to the input features $x_1, x_2, \ldots, x_n$. - n is the number of features.

This can be written in vector form as:

$$z = \theta^T X$$

Where: - $\theta = [\theta_0, \theta_1, \ldots, \theta_n]$ is the weight vector. - $X = [1, x_1, x_2, \ldots, x_n]$ is the feature vector (with 1 added for the bias term).

4. **Probability Prediction**

The probability $P(y = 1|X)$ is modeled using the sigmoid function:

$$P(y = 1|X) = \sigma(z) = \frac{1}{1 + e^{-\theta^T X}}$$

Similarly, the probability $P(y = 0|X)$ is:

$$P(y = 0|X) = 1 - P(y = 1|X) = 1 - \sigma(z)$$

5. **Decision Boundary**

To make predictions, a threshold (usually 0.5) is applied to the predicted probability: - If $P(y = 1|X) \geq 0.5$, predict $y = 1$. - If $P(y = 1|X) < 0.5$, predict $y = 0$.

The decision boundary is the set of points where $P(y = 1|X) = 0.5$, which corresponds to $z = 0$ (i.e., $\theta^T X = 0$)

.

6. **Cost Function (Log Loss)**

To train the logistic regression model, we need a cost function to measure how well the model's predictions match the actual labels. The cost function used in logistic regression is the **log loss** (or binary cross-entropy loss):

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^{m} \left[y^{(i)} \log(h_\theta(X^{(i)})) + (1 - y^{(i)}) \log(1 - h_\theta(X^{(i)})) \right]$$

Where: - m is the number of training examples. - $y^{(i)}$ is the actual label for the i-th example. - $h_\theta(X^{(i)}) = \sigma(\theta^T X^{(i)})$ is the predicted probability for the i-th example.

This cost function penalizes incorrect predictions more heavily, especially when the model is confident but wrong.

7. **Optimization (Gradient Descent)**

The goal is to minimize the cost function $J(\theta)$ by finding the optimal values of θ. This is typically done using **gradient descent**:

1. Initialize the weights θ randomly or to zero. 2. Compute the gradient of the cost function with respect to θ:

$$\frac{\partial J(\theta)}{\partial \theta_j} = \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(X^{(i)}) - y^{(i)} \right) X_j^{(i)}$$

3. Update the weights iteratively:

$$\theta_j := \theta_j - \alpha \frac{\partial J(\theta)}{\partial \theta_j}$$

Where: - α is the learning rate (a hyperparameter controlling the step size). - The process is repeated until convergence.

Untitled

February 27, 2025

[2]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, precision_score, recall_score,␣
 ↪f1_score, confusion_matrix, roc_curve, auc

# Set basic style for visualizations
plt.style.use('ggplot')

# Sigmoid function
def sigmoid(z):
    z = np.clip(z, -500, 500)  # Prevent overflow
    return 1 / (1 + np.exp(-z))

# Logistic Regression Model
class SimpleLogisticRegression:
    def __init__(self, learning_rate=0.01, num_iterations=1000, lambda_=0.0):
        self.learning_rate = learning_rate
        self.num_iterations = num_iterations
        self.lambda_ = lambda_
        self.theta = None
        self.scaler = StandardScaler()

    def fit(self, X, y):
        # Scale the features
        X_scaled = self.scaler.fit_transform(X)

        # Add intercept term
        X_b = np.c_[np.ones((X_scaled.shape[0], 1)), X_scaled]

        # Initialize parameters
        self.theta = np.zeros(X_b.shape[1])

        # Gradient descent
```

```
        for i in range(self.num_iterations):
            y_pred = sigmoid(X_b @ self.theta)
            error = y_pred - y
            gradient = (1/len(y)) * (X_b.T @ error)
            gradient[1:] += (self.lambda_ / len(y)) * self.theta[1:]
            self.theta -= self.learning_rate * gradient

            if i % 500 == 0:
                print(f"Iteration {i}: Training in progress...")

        return self

    def predict_proba(self, X):
        X_scaled = self.scaler.transform(X)
        X_b = np.c_[np.ones((X_scaled.shape[0], 1)), X_scaled]
        return sigmoid(X_b @ self.theta)

    def predict(self, X, threshold=0.5):
        probabilities = self.predict_proba(X)
        return (probabilities >= threshold).astype(int)

# Data loading and preprocessing function
def prepare_data(file_path):
    # Load data
    df = pd.read_csv("C:/Users/PRO. BUNOOTI/Desktop/OUR WORK/heart_attack.csv")
    print(f"Loaded data with shape: {df.shape}")

    # Handle missing values
    for col in df.columns:
        if df[col].isnull().sum() > 0:
            if df[col].dtype.kind in 'ifc':  # numeric
                df[col] = df[col].fillna(df[col].median())
            else:  # categorical
                df[col] = df[col].fillna(df[col].mode()[0])

    # Check for target column
    target_column = 'Heart Attack Risk'
    if target_column not in df.columns:
        raise ValueError(f"Target column '{target_column}' not found in␣
 ↪dataset")

    # One-hot encode categorical features
    categorical_cols = df.select_dtypes(include=['object']).columns
    if len(categorical_cols) > 0:
        df = pd.get_dummies(df, columns=categorical_cols, drop_first=True)

    # Extract features and target
```

```
    y = df[target_column].values
    X = df.drop(target_column, axis=1)
    feature_names = X.columns

    # Split data
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)
    print(f"Training set size: {X_train.shape[0]}, Test set size: {X_test.
 ↪shape[0]}")

    return X_train, X_test, y_train, y_test, feature_names

# Function to evaluate model and detect overfitting/underfitting
def evaluate_model(model, X_train, y_train, X_test, y_test, feature_names):
    # Get predictions
    y_train_prob = model.predict_proba(X_train)
    y_test_prob = model.predict_proba(X_test)

    y_train_pred = model.predict(X_train)
    y_test_pred = model.predict(X_test)

    # Print some test predictions to examine
    print("\n=== Test Data Predictions (First 10 samples) ===")
    print("True Values | Predicted | Probability")
    print("-" * 40)
    for i in range(min(10, len(y_test))):
        print(f"{y_test[i]:^10} | {y_test_pred[i]:^9} | {y_test_prob[i]:.4f}")

    # Calculate metrics
    metrics = ['accuracy', 'precision', 'recall', 'f1_score']
    train_values = [
        accuracy_score(y_train, y_train_pred),
        precision_score(y_train, y_train_pred),
        recall_score(y_train, y_train_pred),
        f1_score(y_train, y_train_pred)
    ]

    test_values = [
        accuracy_score(y_test, y_test_pred),
        precision_score(y_test, y_test_pred),
        recall_score(y_test, y_test_pred),
        f1_score(y_test, y_test_pred)
    ]

    # Print metrics
    print("\n=== Performance Comparison ===")
    print(f"{'Metric':<10} {'Training':<10} {'Testing':<10} {'Gap':<10}")
```

```
    print("-" * 40)
    for i, metric in enumerate(metrics):
        gap = train_values[i] - test_values[i]
        print(f"{metric:<10} {train_values[i]:.4f}     {test_values[i]:.4f}
↪{gap:.4f}")

    # Calculate average metrics for detecting overfitting/underfitting
    avg_train = sum(train_values) / len(train_values)
    avg_test = sum(test_values) / len(test_values)
    avg_gap = avg_train - avg_test

    # Detect fitting status
    if avg_gap > 0.1:
        fitting_status = "Overfitting"
    elif avg_test < 0.6:  # Poor performance on both training and testing
        fitting_status = "Underfitting"
    elif avg_gap > 0.05:
        fitting_status = "Slight Overfitting"
    elif avg_test < 0.7:
        fitting_status = "Slight Underfitting"
    else:
        fitting_status = "Good Fit"

    print(f"\n=== Model Fitting Status: {fitting_status} ===")
    print(f"Average train score: {avg_train:.4f}")
    print(f"Average test score: {avg_test:.4f}")
    print(f"Performance gap: {avg_gap:.4f}")

    # Create visualizations
    # 1. Metrics comparison bar chart
    fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(18, 6))

    # Bar chart for metrics
    x = np.arange(len(metrics))
    width = 0.35

    ax1.bar(x - width/2, train_values, width, label='Training')
    ax1.bar(x + width/2, test_values, width, label='Testing')

    ax1.set_ylabel('Score')
    ax1.set_title('Training vs Testing Performance')
    ax1.set_xticks(x)
    ax1.set_xticklabels(metrics)
    ax1.legend()
    ax1.set_ylim(0, 1.1)
    ax1.grid(axis='y', linestyle='--', alpha=0.7)
```

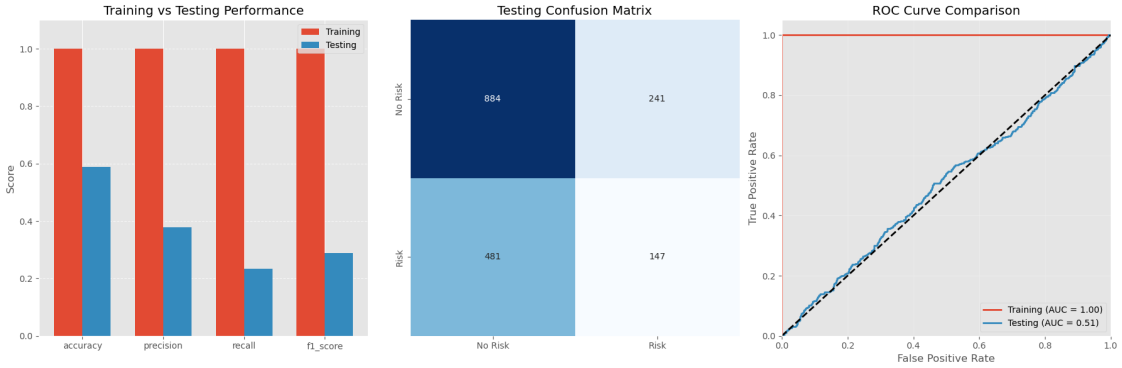
```
    # 2. Confusion Matrix for test data
    test_cm = confusion_matrix(y_test, y_test_pred)

    sns.heatmap(test_cm, annot=True, fmt='d', cmap='Blues', cbar=False,
                xticklabels=['No Risk', 'Risk'], yticklabels=['No Risk',␣
↪'Risk'], ax=ax2)
    ax2.set_title('Testing Confusion Matrix')

    # 3. ROC Curve
    fpr_train, tpr_train, _ = roc_curve(y_train, y_train_prob)
    roc_auc_train = auc(fpr_train, tpr_train)

    fpr_test, tpr_test, _ = roc_curve(y_test, y_test_prob)
    roc_auc_test = auc(fpr_test, tpr_test)

    ax3.plot(fpr_train, tpr_train, lw=2, label=f'Training (AUC = {roc_auc_train:
↪.2f})')
    ax3.plot(fpr_test, tpr_test, lw=2, label=f'Testing (AUC = {roc_auc_test:.
↪2f})')
    ax3.plot([0, 1], [0, 1], 'k--', lw=2)
    ax3.set_xlim([0.0, 1.0])
    ax3.set_ylim([0.0, 1.05])
    ax3.set_xlabel('False Positive Rate')
    ax3.set_ylabel('True Positive Rate')
    ax3.set_title('ROC Curve Comparison')
    ax3.legend(loc="lower right")
    ax3.grid(True, alpha=0.3)

    plt.tight_layout()
    plt.show()

    # 4. Feature importance
    plt.figure(figsize=(10, 6))

    # Skip the intercept term
    coeffs = model.theta[1:]

    # Create DataFrame for plotting
    importance_df = pd.DataFrame({
        'Feature': feature_names,
        'Coefficient': coeffs,
        'Absolute Value': np.abs(coeffs)
    })

    # Sort by absolute value
    importance_df = importance_df.sort_values('Absolute Value',␣
↪ascending=False).head(10)
```

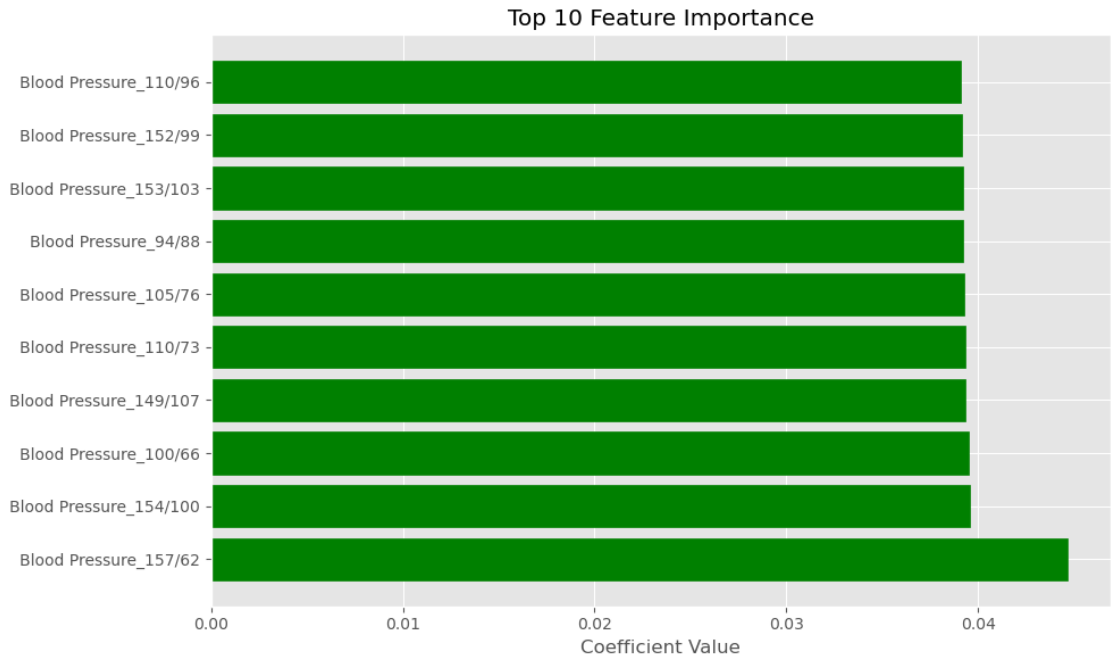
```
    # Plot
    bars = plt.barh(importance_df['Feature'], importance_df['Coefficient'])

    # Color bars based on coefficient sign
    for i, bar in enumerate(bars):
        if importance_df['Coefficient'].iloc[i] < 0:
            bar.set_color('red')
        else:
            bar.set_color('green')

    plt.title('Top 10 Feature Importance')
    plt.xlabel('Coefficient Value')
    plt.tight_layout()
    plt.show()

    return fitting_status, avg_train, avg_test, avg_gap

# Function to train and auto-tune model
def train_and_tune(X_train, X_test, y_train, y_test, feature_names,␣
 ↪max_iterations=3):
    # Initial parameters
    learning_rate = 0.01
    num_iterations = 1000
    lambda_ = 0.1

    for iteration in range(max_iterations):
        print(f"\n=== Training Model (Iteration {iteration+1}) ===")
        print(f"Parameters: learning_rate={learning_rate},␣
 ↪iterations={num_iterations}, lambda={lambda_}")

        # Train model
        model = SimpleLogisticRegression(learning_rate=learning_rate,␣
 ↪num_iterations=num_iterations, lambda_=lambda_)
        model.fit(X_train, y_train)

        # Evaluate model
        fitting_status, avg_train, avg_test, gap = evaluate_model(model,␣
 ↪X_train, y_train, X_test, y_test, feature_names)

        # Check if model is already good
        if fitting_status == "Good Fit":
            print("\n=== Model is well tuned! ===")
            return model, fitting_status

        # Auto-tune based on fitting status
        if "Overfitting" in fitting_status:
```

```
            # Increase regularization and reduce complexity
            lambda_ *= 2
            print(f"\n=== Detected {fitting_status} ===")
            print(f"Tuning strategy: Increasing regularization to␣
 ↪lambda={lambda_}")

        elif "Underfitting" in fitting_status:
            # Reduce regularization and increase complexity
            lambda_ = max(lambda_ / 2, 0.01)
            num_iterations = min(num_iterations * 2, 5000)
            print(f"\n=== Detected {fitting_status} ===")
            print(f"Tuning strategy: Decreasing regularization to␣
 ↪lambda={lambda_} and increasing iterations to {num_iterations}")

        # If this is the last iteration and we haven't found a good fit
        if iteration == max_iterations - 1:
            print("\n=== Maximum tuning iterations reached ===")
            return model, fitting_status

    return model, fitting_status

# Main function
def main(file_path):
    # Load and prepare data
    X_train, X_test, y_train, y_test, feature_names = prepare_data(file_path)

    # Train and auto-tune model
    final_model, fitting_status = train_and_tune(X_train, X_test, y_train,␣
 ↪y_test, feature_names)

    print("\n=== Final Model Summary ===")
    print(f"Fitting Status: {fitting_status}")
    print(f"Regularization (lambda): {final_model.lambda_}")
    print(f"Number of features: {len(feature_names)}")

    return final_model

# Run the program
if __name__ == "__main__":
    file_path = "heart_attack.csv"  # Replace with your file path
    model = main(file_path)
```

```
Loaded data with shape: (8763, 26)
Training set size: 7010, Test set size: 1753

=== Training Model (Iteration 1) ===
Parameters: learning_rate=0.01, iterations=1000, lambda=0.1
```

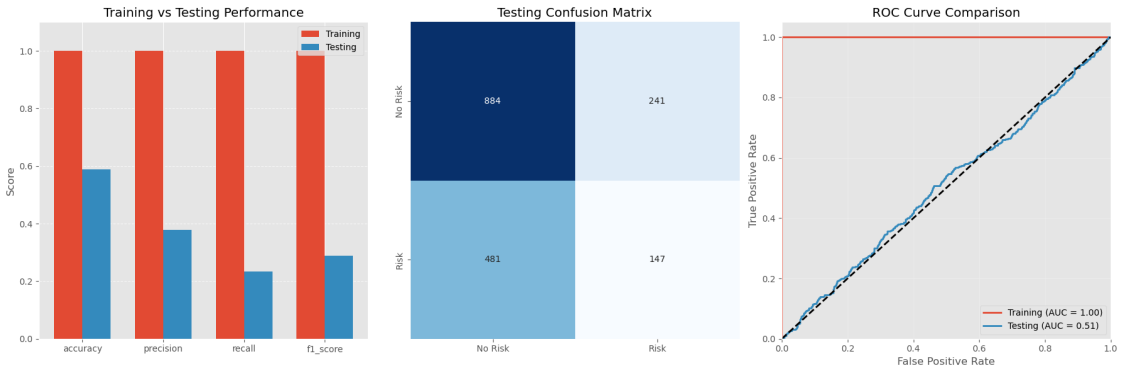
```
Iteration 0: Training in progress…
Iteration 500: Training in progress…

=== Test Data Predictions (First 10 samples) ===
True Values | Predicted | Probability
----------------------------------------
    0       |     0     | 0.3490
    1       |     1     | 0.8042
    1       |     0     | 0.3229
    1       |     0     | 0.1485
    0       |     1     | 0.7652
    1       |     0     | 0.3557
    1       |     0     | 0.4785
    0       |     0     | 0.1387
    0       |     1     | 0.7952
    1       |     0     | 0.1472

=== Performance Comparison ===
Metric     Training   Testing    Gap
----------------------------------------
accuracy   1.0000     0.5881     0.4119
precision  1.0000     0.3789     0.6211
recall     1.0000     0.2341     0.7659
f1_score   1.0000     0.2894     0.7106

=== Model Fitting Status: Overfitting ===
Average train score: 1.0000
Average test score: 0.3726
Performance gap: 0.6274
```

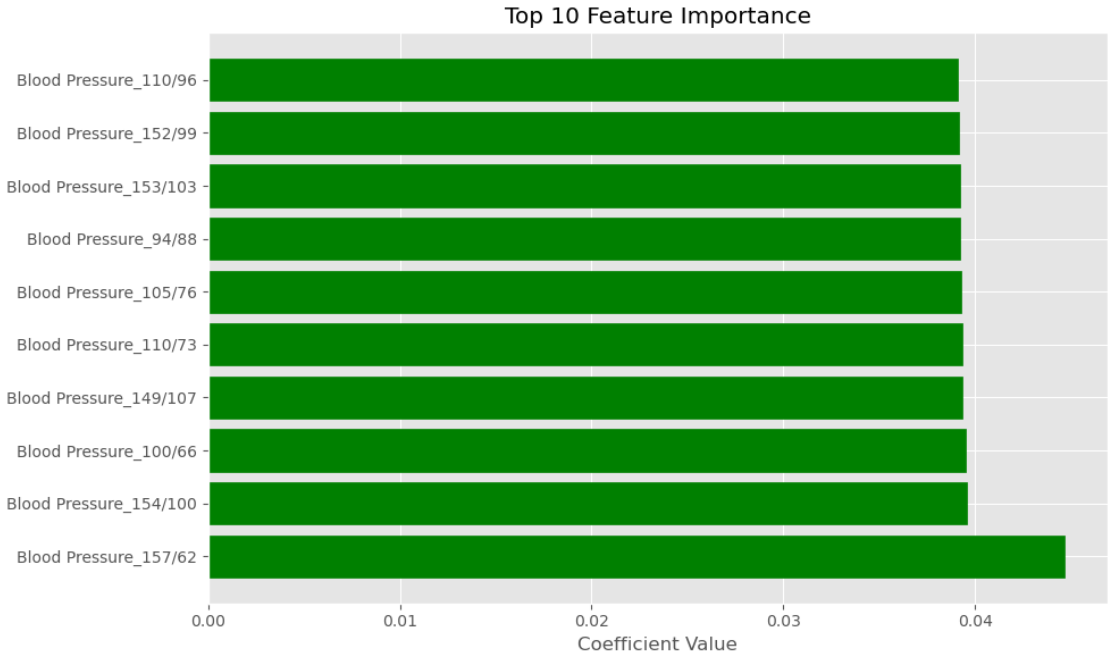





```
=== Detected Overfitting ===
Tuning strategy: Increasing regularization to lambda=0.2

=== Training Model (Iteration 2) ===
Parameters: learning_rate=0.01, iterations=1000, lambda=0.2
Iteration 0: Training in progress…
Iteration 500: Training in progress…

=== Test Data Predictions (First 10 samples) ===
True Values | Predicted | Probability
----------------------------------------
    0       |     0     | 0.3490
    1       |     1     | 0.8041
    1       |     0     | 0.3229
    1       |     0     | 0.1485
    0       |     1     | 0.7652
    1       |     0     | 0.3557
    1       |     0     | 0.4785
    0       |     0     | 0.1387
    0       |     1     | 0.7952
    1       |     0     | 0.1472

=== Performance Comparison ===
Metric     Training   Testing    Gap
----------------------------------------
```

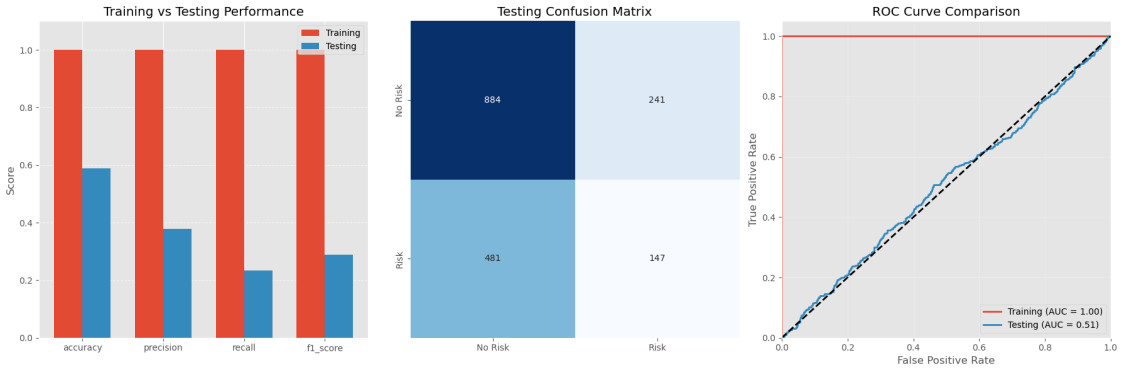
```
accuracy    1.0000      0.5881      0.4119
precision   1.0000      0.3789      0.6211
recall      1.0000      0.2341      0.7659
f1_score    1.0000      0.2894      0.7106

=== Model Fitting Status: Overfitting ===
Average train score: 1.0000
Average test score: 0.3726
Performance gap: 0.6274
```

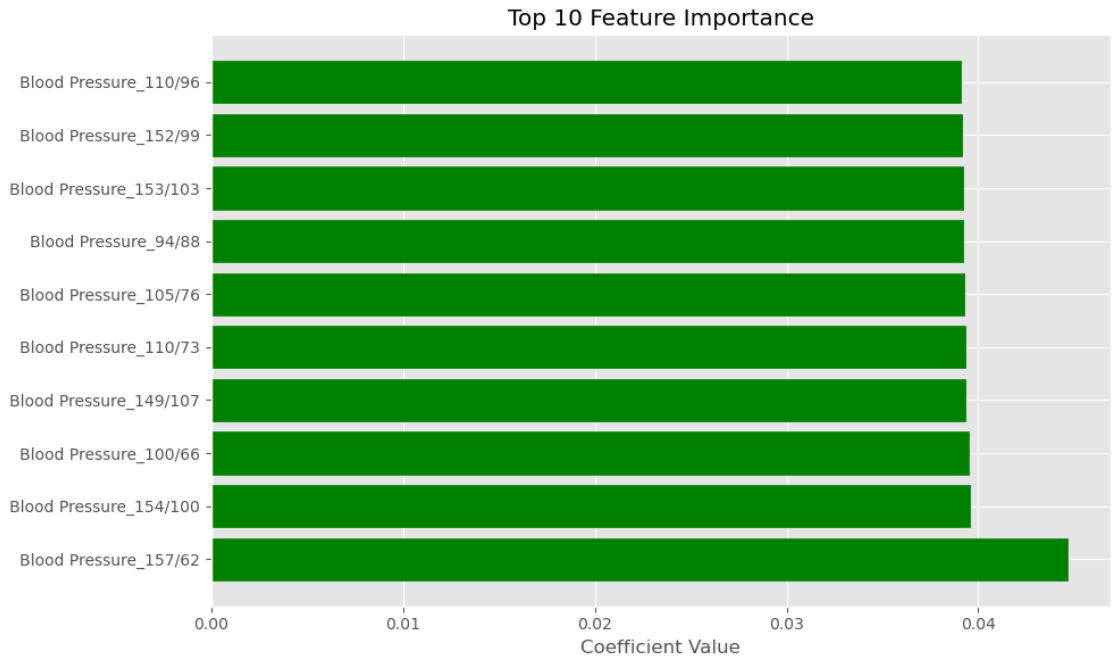





```
=== Detected Overfitting ===
Tuning strategy: Increasing regularization to lambda=0.4
```

```
=== Training Model (Iteration 3) ===
Parameters: learning_rate=0.01, iterations=1000, lambda=0.4
Iteration 0: Training in progress…
Iteration 500: Training in progress…

=== Test Data Predictions (First 10 samples) ===
True Values | Predicted | Probability
----------------------------------------
    0       |     0     | 0.3490
    1       |     1     | 0.8041
    1       |     0     | 0.3229
    1       |     0     | 0.1485
    0       |     1     | 0.7652
    1       |     0     | 0.3557
    1       |     0     | 0.4785
    0       |     0     | 0.1387
    0       |     1     | 0.7952
    1       |     0     | 0.1472

=== Performance Comparison ===
Metric     Training   Testing    Gap
----------------------------------------
accuracy   1.0000     0.5881     0.4119
precision  1.0000     0.3789     0.6211
recall     1.0000     0.2341     0.7659
f1_score   1.0000     0.2894     0.7106

=== Model Fitting Status: Overfitting ===
Average train score: 1.0000
Average test score: 0.3726
Performance gap: 0.6274
```






```
=== Detected Overfitting ===
Tuning strategy: Increasing regularization to lambda=0.8

=== Maximum tuning iterations reached ===

=== Final Model Summary ===
Fitting Status: Overfitting
Regularization (lambda): 0.4
Number of features: 12722
```

[ ]: