Index

1. Question 1

In a Support Vector Machine (SVM), we aim to separate two classes using a decision boundary (hyperplane). This boundary is defined as: $w^T x_i + b = 0$, where w represents the weight vector, b is the bias that adjusts the position of the hyperplane.

Next, define two supporting hyperplane ensuring that data points fall on the correct side of the margin.

- For positive class ($y_i = +1$): $w^T x_i + b \geq +1$
- For negative class ($y_i = -1$): $w^T x_i + b \leq -1$

To simplify and unify these conditions, we multiply y_i which is +1 for positive class and -1 for negative class, resulting in: $y_i(w^T x_i + b) \geq 1$. This constraint guarantees that all points are at least 1 unit away from the decision boundary.

- For positive points ($y_i = +1$), it remains $w^T x_i + b \geq 1$
- For negative points ($y_i = -1$), it transforms into $w^T x_i + b \leq -1$

Finally, to maximise the margin, the margin width (distance between the two hyperplanes) is given by $\frac{2}{||w||}$. Since the larger margin leads to better generalization, we minimize $||w||$ and make it mathematically easier, thus, the final SVM optimization problem becomes: $\min_{w,b}(1/2)\,||w||^2$ subject to: $y_i(w^T x_i + b) \geq 1,\ for\ all\ i$

2. Question 2

2.1. Choose initial feasible solution

Given that initial feasible solution $z_0 = (w_0, b_0)$ to satisfy the constraints:

$$y_i(w_0^T x_i + b_0) \geq 1$$

$$-M \leq w_j \leq M, \quad j = 1, \dots, d$$

$$-M \leq b \leq M$$

If the dataset is linearly divisible, initialise the $w_0 = 0, b_0 = 0$, and then adjust it to satisfy the constraints.

2.2. Calculate the gradient

- Objective function: $f(z) = \frac{1}{2}\left||w|\right|^2 = \frac{1}{2}\sum_{j=1}^{d} w_j^2$
- Find the gradient for w: ∇f$(w, b) = (w, 0)$

2.3. Calculate the feasible direction $d_k = v_k - z_k$

v_k is obtained by solving the optimisation problem: $v_k = \arg m\, in \nabla f(z_k)^T v,\ v \in Z$, and is chosen as the feasible point in the direction that decreases the objective function the most.

Plus, objective function in SVM is: $f(w) = \frac{1}{2}\left||w|\right|^2$, its gradient is: $\nabla\, f(w)\ =\ w$

Thus, we need to find the closest feasible point for minimising the gradient. The projection operator $\Pr o\, j_z w_k$ finds the closest feasible point to w_k in the feasible set $v_k = -\Pr o\, j_z(w_k)$. Projection naturally results from gradient minimisation, which results in: $d_k = -\Pr o\, j_z(w_k) - w_k$

2.4. Calculate the step τ_k

τ_k needs to be solved by the following optimisation problem: $\tau_k = \arg \min_{\tau \in [0,1]} f(zk + \tau dk)$

Then: $\tau k = \arg \min_{\tau \in [0,1]} \frac{1}{2} |w_k + \tau d_k|^2$

Calculate the derivative of τ_k to find the optimal solution: $\frac{1}{2} \cdot 2(w_k + \tau d_k)^T d_k = (w_k + \tau d_k)^T d_k$

Let the derivative be equal to zero: $(w_k + \tau d_k)^T d_k = 0$

Result: $\tau_k = -\frac{w_k^T d_k}{|d_k|^2}$ Ranging from [0,1]. Since τ is constrained to [0,1], project τ* into this range:

$\tau * \ =\ \max\left(0, \min\left(1,\ -\frac{w_k^T d_k}{|d_k|^2}\right)\right)$

- If $\tau *$< 0: This would imply moving in the opposite direction, which is not allowed in this optimisation context. So, we set the step size to 0 (no movement).
- If $\tau *$> 0: This would imply taking a step larger than the optimal direction suggested by the gradient and feasible direction, which might cause us to exceed the boundaries. So, we set the step size to 1 (maximum allowed movement).
- If 0 ≤ $\tau *$ ≤ 1: If the computed $\tau *$ already lies in the feasible range, then we can directly use it without modification.

2.5. Updating solutions $z_{(k+1)} = z_{(k)} + \tau^* d_{(k)}$

2.6. Checking for convergence

If $|z_{k+1} - z_k| \leq \epsilon$ then stop, otherwise return to Step 2.2 and continue iteration.

3. Question 3

The algorithm halts when the change in the solution z_k, (including weight w and bias b) between iterations falls below a predefined tolerance ϵ, calculated with the Euclidean norm $\|z_k - z_{k-1}\|$. For classifier, the model perfectly distinguishes the Iris-setosa class (label = +1) and other class (label = -1), achieving 100% accuracy with the classifying weight of sepal length, sepal width, petal length, petal width shows −0.04604523, 0.5216668, −1.0032087, and −0.46445877, respectively. Plus, the bias is 1.45118752 for adjusting the position of the decision boundary. Additionally, tolerance parameter ϵ is set as 1e-5, a smaller ϵ means the algorithm will continue iterating until the change in zk is very small, resulting in a more accurate solution. Finally, M is set as 10, this means that each component of w and b is constrained to lie within the interval $[-M,M]$. Through cross-validation, the small M value 10 restricts the solution space, gives the best performance on a validation set is selected.

4. Question 4

Knowing that Setosa and Versicolor/Virginica are linearly distinguishable, and Versicolor and Virginica are non-linearly distinguishable, therefore, the data's noise and class overlap necessitate a soft-margin SVM with a regularisation parameter C. To balance misclassification and generalisation, optimising C through cross-validation is crucial, shifting from a rigid to a more adaptable hyperplane.

4.1. Model Interpretation and Variable Assumption Method

4.1.1. Additional Decision Variables

We introduce slack variables u_i for each training sample. These variables allow the SVM to handle cases where the data is not perfectly linearly separable, where u_i measures the degree to which the training sample violates the margin constraints.

4.1.2. New Objective Function $\min_{w,b,u} \frac{1}{2}\left||w|\right|^2 + C\sum_{i=1}^{N} u_i$

$\frac{1}{2}\left||w|\right|^2$ represents the original term that maximises the margin by minimising the norm of w.

$\sum_{i=1}^{N} u_i$ represents a penalty term for violating the margin constraint, where C is a hyperparameter controlling the trade-off between maximising the margin and minimising classification errors.

4.1.3. Modified Constraints

Margin Constraint: $(w^T x_i + b) \geq 1 - u_i$, $\forall i = 1,2,\ldots,N$.This allows for a margin violation when $u_i > 0$, enabling SVM to handle non-linearly separable data.

Non-negativity Constraint: $u_i \geq 0$, $\forall i = 1,2,\ldots,N$.This ensures that the slack variables are non-negative.

4.1.4. Set the Value for Parameter $\boldsymbol{C}$

A larger C means the SVM will try to avoid misclassification more strictly, resulting in a smaller margin (risk of overfitting). A smaller C allows for a larger margin, tolerating more classification errors (better generalization).

4.2. Algorithm to Classify a New Iris Record

Apply Hard-Margin SVM to distinguish Setosa first, then perform Soft-Margin SVM to distinguish Setosa.

4.2.1. Train a Hard-Margin SVM to Separate Setosa (0) and Non-Setosa (1)

Train a Hard-Margin SVM: $\min_{w,b} \frac{1}{2}\left||w|\right|^2$ subject to:$y_i(w^T x_i + b) \geq 1, \quad \forall i = 1,2,\ldots,N$

If the new sample is classified as Setosa, the classification is complete. Otherwise, proceed to Step 4.2.2.

4.2.2. Train a Soft-Margin SVM to Classify Versicolor (1) vs. Virginica (2)

Train a Soft-Margin SVM: $\min_{w,b} \frac{1}{2}\left||w|\right|^2 + C\Sigma u_i$ subject to: $y_i(w^T x_i + b) \geq 1 - u_i, \quad u_i \geq 0, \quad \forall i = 1,2,\ldots,N$

If the sample is classified as Versicolor, assign it to that class. Otherwise, classify it as Virginica.

Appendices

APPENDIX A: PYTHON CODE FOR QUESTION 3

```
from pyomo.environ import *

from pyomo.opt import SolverStatus, TerminationCondition

from sklearn.linear_model import LinearRegression

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt


# Load the data

data = pd.read_csv("Data.csv", header=None)

X = data.iloc[:, :-1].values # Features

y = data.iloc[:, -1].values   # Labels


# Convert labels to binary: Iris-setosa = 1, others = -1

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


# Parameters

d = X.shape[1]  # Number of features

M = 10  # Bound for w and b

epsilon = 1e-5  # Convergence threshold

max_iter  = 1000  # Maximum number of iterations
```

```
# Initialize w and b randomly
weights = np.random.uniform(-M, M, d)
bias = np.random.uniform(-M, M)


# Objective function: 1/2 * ||w||^2
def objective(z):
    w=z[:-1]
    return 0.5 * np.dot(w, w)


# Gradient of the objective function
def gradient(z):
    w=z[:-1]
    return np.append(w, 0)  # Gradient w.r.t. w is w, w.r.t. b is 0


# Solve linear programming problem to find v_k
def find_vk(z):
    model = ConcreteModel()

    # Decision variables: v = [w_v; b_v]
    model.weights_v = Var(range(d), bounds=(-M, M))
    model.bias_v = Var(bounds=(-M, M))

    # Objective: minimize gradient(z)^T v
    def objective_function(m):
```

```
        return sum(gradient(z)[j] * m.weights_v[j] for j in range(d))
+ gradient(z)[-1] * m.bias_v

    model.objective = Objective(rule=objective_function,
sense=minimize)

    # Constraints: y_i (w_v^T x_i + b_v) >= 1 for all i
    def constraint_function(m, i):
        return y[i] * (sum(m.weights_v[j] * X[i, j] for j in range(d))
+ m.bias_v) >= 1

    model.constraints = Constraint(range(len(y)),
rule=constraint_function)

    # Solve the problem
    solver = SolverFactory('glpk')
    results = solver.solve(model)

    # Extract the solution
    weights_v = np.array([model.weights_v[j]() for j in range(d)])
    bias_v = model.bias_v()

    return np.append(weights_v, bias_v)

# compute_tau_k using IPOPT
def compute_tau(z, d):
```

```
    model = ConcreteModel()

    # Decision variable: tau_k
    model.tau = Var(bounds=(0, 1))

    # Objective: minimize the objective function at z + tau * d
    def objective_function(m):
        return objective(z + m.tau * d)

    model.objective = Objective(rule=objective_function,
sense=minimize)

    # Solve the problem
    solver = SolverFactory('ipopt')  # Use IPOPT solver
    results = solver.solve(model)

    # Extract the solution
    return model.tau()

# Optimization loop
z = np.append(weights, bias)
z_history = pd.DataFrame(columns=[f'weight_{j+1}' for j in range(d)] +
['bias'])
z_history.loc[0] = z  # Add initial solution to the DataFrame
```

```
for iteration in range(max_iter):
    # Find v_k
    v_k = find_vk(z)

    # Find direction d_k = v_k - z
    d_k = v_k - z

    # Compute_tau_k
    tau_k = compute_tau(z, d_k)

    # Update z
    z_new = z + tau_k * d_k
    z_history.loc[iteration + 1] = z_new

    # Check for convergence
    if np.linalg.norm(z_new - z) < epsilon:
        break

    z = z_new

# Extract optimal weights and bias
optimal_weights = z[:-1]
optimal_bias = z[-1]
```

```
# Predict function
def predict(X):
    return np.sign(np.dot(X, optimal_weights) + optimal_bias)


# Evaluate accuracy
y_pred = predict(X)
accuracy = np.mean(y_pred == y)
print(f"Final Accuracy: {accuracy:.4f}")
print(f"Final SVM Weights: {z}")
print(f"Final Bias: {optimal_bias}")



df_columns = ['sepal_length', 'sepal_width', 'petal_length',
'petal_width', 'class']
feature_importance = np.abs(optimal_weights)
top_2_indices = np.argsort(feature_importance)[-2:]  # Get two largest
weights
print(f"Selected Features:  {df_columns[top_2_indices[0]]} ,
{df_columns[top_2_indices[1]]}")



def plot_decision_boundary(X, y, w, b, feature_indices):
    plt.figure(figsize=(8, 6))


    # Use only the two selected features
```

```
    X_vis = X[:, feature_indices]
    w_vis = w[feature_indices]  # Use the weights corresponding to
selected features

    for label, color in zip([1, -1], ['#3d5a80', '#ee6c4d']):
        subset = X_vis[y == label]
        plt.scatter(subset[:, 0], subset[:, 1], label="Iris-setosa" if
label == 1 else "Others",
                    color=color, alpha=0.7, edgecolors='k')

    # Compute the decision boundary
    x_min, x_max = X_vis[:, 0].min() - 1, X_vis[:, 0].max() + 1
    x_vals = np.linspace(x_min, x_max, 100)
    y_vals = -(w_vis[0] * x_vals + b) / w_vis[1]

    plt.plot(x_vals, y_vals, 'k-', linewidth=2, label='Decision
Boundary')
    plt.xlabel(f'Feature {df_columns[feature_indices[0]] }')
    plt.ylabel(f'Feature {df_columns[feature_indices[1]]}')
    plt.title("Optimized SVM Decision Boundary")
    plt.legend()
    plt.show()

# Select the best two features and plot
plot_decision_boundary(X, y, optimal_weights, optimal_bias,
top_2_indices)
```

APPENDIX B: PLOT FOR QUESTION 3

Bound (M=10): Prevents weights and bias from diverging while allowing sufficient flexibility.

Tolerance ϵ=1e-5: Ensures convergence without unnecessary computations.

Max Iterations max_iter=1000: Provides enough updates for stability without excessive computation.

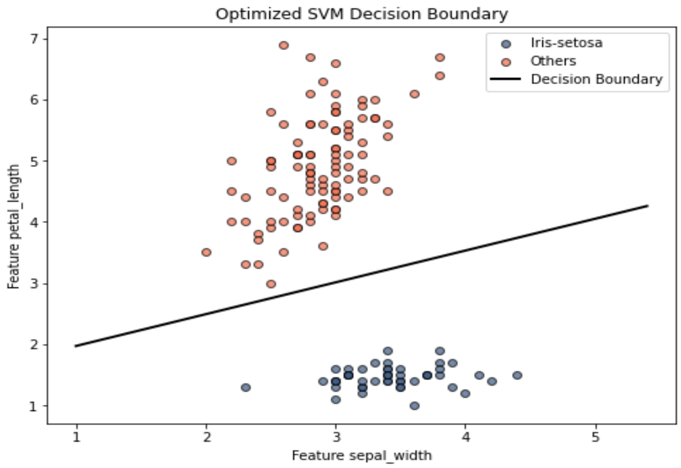

Figure 1: Sepal Width vs Petal Length Features SVM.

APPENDIX C: DATA DISTRIBUTION

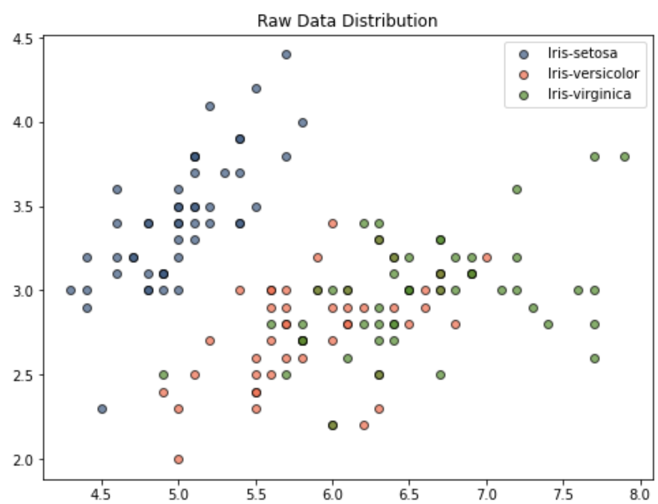

Figure 2: Data Distribution