Department of Electrical Engineering

Indian Institute of Technology, Kharagpur

Algorithms, AI and ML Laboratory (EE22202)

Spring, 2025-26

Report 3: Support Vector Machines

Name: Arijit Dey

Roll No: 24IE10001

Support Vector Machines

Three labeled datasets are given. The input variables lie in R^2. Matrix A contains a collection of vectors in R^2 with label 1 and matrix B contains a collection of vectors in R^2 with label -1.

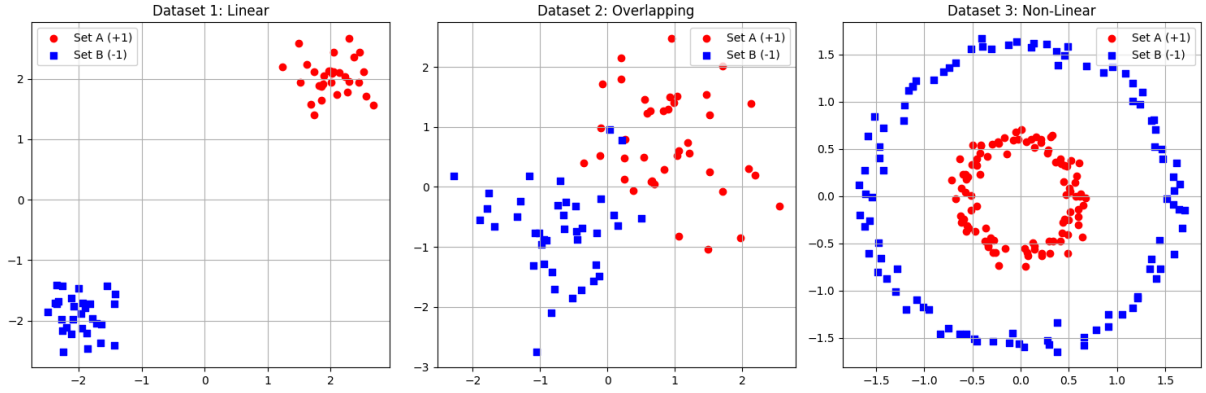
1 Visualizing the Datasets

We plot all the three datasets on the R^2 plane. Each point in set A is denoted as a red circle and set B as blue square.

Visualize datasets using `matplotlib`

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

def generate_svm_datasets():
    np.random.seed(0)
    A1 = np.random.randn(30, 2) * 0.3 + np.array([2, 2])
    B1 = np.random.randn(30, 2) * 0.3 + np.array([-2, -2])
    # Labels: A1 -> +1, B1 -> -1

    np.random.seed(1)
    A2 = np.random.randn(40, 2) * 0.8 + np.array([0.8, 0.8])
    B2 = np.random.randn(40, 2) * 0.8 + np.array([-0.8, -0.8])

    np.random.seed(2)
    n_points = 100
    theta = np.linspace(0, 2*np.pi, n_points)

    r_inner = 0.6
    A3 = np.c_[r_inner * np.cos(theta), r_inner * np.sin(theta)]
    A3 += np.random.randn(n_points, 2) * 0.07

    r_outer = 1.6
    B3 = np.c_[r_outer * np.cos(theta), r_outer * np.sin(theta)]
    B3 += np.random.randn(n_points, 2) * 0.07

    return (A1, B1), (A2, B2), (A3, B3)

(setA1, setB1), (setA2, setB2), (setA3, setB3) = generate_svm_datasets()

plt.figure(figsize=(15, 5))
datasets = [(setA1, setB1, "Dataset 1: Linear"),
            (setA2, setB2, "Dataset 2: Overlapping"),
            (setA3, setB3, "Dataset 3: Non-Linear")]

for i, (A, B, title) in enumerate(datasets):
    plt.subplot(1, 3, i+1)
    plt.scatter(A[:, 0], A[:, 1], c='red', marker='o', label='Set A (+1)')
    plt.scatter(B[:, 0], B[:, 1], c='blue', marker='s', label='Set B (-1)')
    plt.title(title)
    plt.legend()
```

```
    plt.grid(True)

plt.tight_layout()
plt.show()
```



2 Checking the Existence of a Linear Classifier

For a dataset to be separable by linear classifier, there must exist a weight vector w and a bias b such that for every point x_i with label $y_i \in \{1, -1\}$:

$$y_i(w^T x_i + b) \geq 1 \tag{1}$$

To find the separating hyperplane, we treat this as a Hard Margin SVM optimization problem. The best hyperplane (with the maximum margin) associated with the linear classifier can be obtained by solving the following optimization problem:

$$\min_{w \in R^2} \quad f(w) = \frac{1}{2} \|w\|_2^2$$

constrained to the above Equation 1 condition.

Linear Classifier using `cvxpy`

```
import cvxpy as cp

def solve_linear_svm(setA, setB):
    # Prepare data: A is +1, B is -1
    X = np.vstack([setA, setB])
    y = np.hstack([np.ones(len(setA)), -np.ones(len(setB))])

    N, D = X.shape

    w = cp.Variable(D)
    b = cp.Variable()

    # Constraints: y_i * (w^T * x_i + b) >= 1
    constraints = [cp.multiply(y, X @ w + b) >= 1]

    # Objective: Minimize 0.5 * ||w||^2
    prob = cp.Problem(cp.Minimize(0.5 * cp.sum_squares(w)), constraints)

    try:
```

```
        prob.solve()
        if prob.status == cp.OPTIMAL:
            return w.value, b.value
        else:
            return None, None
    except:
        return None, None

# Test the datasets
datasets = [ (setA1, setB1, "Dataset 1"), (setA2, setB2, "Dataset 2"), (setA3, setB3,
"Dataset 3") ]

for A, B, name in datasets:
    w, b = solve_linear_svm(A, B)
    if w is not None:
        print(f"{name}: Linearly Separable!")
        print(f"   Hyperplane: {w[0]:.2f}x + {w[1]:.2f}y + {b:.2f} = 0")
    else:
        print(f"{name}: Not Linearly Separable.")
```

```
Result
Dataset 1: Linearly Separable!
   Hyperplane: 0.34x + 0.32y + -0.03 = 0
Dataset 2: Not Linearly Separable.
Dataset 3: Not Linearly Separable.
```

3 Finding Solution for the Primal and Dual problem and Verification of KKT conditions

3.1 The Primal and Dual Problem

The Primal problem defined as

$$\min_{w \in R^2} \quad f(w) = \frac{1}{2} \|w\|_2^2$$
$$\text{Subject to} \quad y_i(w^T x_i + b) \geq 1$$

is the direct formulation of the SVM goal i.e to find a hyperplane that maximizes the margin while correctly classifying data. Here we directly adjust the orientation and position of the hyperplane in the input feature space.

The Dual problem defined as

$$\max_{\lambda \geq 0} \quad \left(\sum_{i=1}^{N} \lambda_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \lambda_i \lambda_j \hat{y}_i \hat{y}_j (\hat{x}_i)^T \hat{x}_j \right)$$
$$\text{Subject to} \quad \sum_{i=1}^{N} \lambda_i \hat{y}_i = 0 \quad 0 \leq \lambda_i \quad \forall i \in \{N\}$$

is useful for determining how much importance or weight each individual data point has in defining the final boundary.

The most important property of the dual problem is that the kernel trick can be applied on it. This is used for classifying non-linear datasets.

3.2 KKT Conditions

Let (w^*, b^*) be the primal optimal solution and λ^* be the dual optimal solution. Then the KKT conditions are defined as:

1. $\lambda_i^* \geq 0, \quad \sum_{i=1}^{N} \lambda_i^* \hat{y}_i = 0$
2. $1 - \hat{y}_i\left((w^*)^T \hat{x}_i + b^*\right) \leq 0$
3. $\lambda_i^* \left[1 - \hat{y}_i\left((w^*)^T \hat{x}_i + b^*\right)\right] = 0$
4. $w^* = \sum_{i=1}^{N} \lambda_i^* \hat{x}_i \hat{y}_i$

Solution for Primal and Dual problem

```python
X = np.vstack([setA1, setB1])
y = np.hstack([np.ones(len(setA1)), -np.ones(len(setB1))])
N, d = X.shape

# --- PRIMAL PROBLEM ---
w_p = cp.Variable(d)
b_p = cp.Variable()
primal_obj = cp.Minimize(0.5 * cp.sum_squares(w_p))
primal_con = [cp.multiply(y, X @ w_p + b_p) >= 1]
prob_p = cp.Problem(primal_obj, primal_con)
prob_p.solve()

# --- DUAL PROBLEM ---
alpha = cp.Variable(N)
P = np.outer(y, y) * (X @ X.T)
dual_obj = cp.Maximize(cp.sum(alpha) - 0.5 * cp.quad_form(alpha, cp.psd_wrap(P)))
dual_con = [alpha >= 0, cp.sum(cp.multiply(alpha, y)) == 0]
prob_d = cp.Problem(dual_obj, dual_con)
prob_d.solve()

# --- PARAMETER RECOVERY ---
w_from_dual = np.sum((alpha.value[:, None] * y[:, None] * X), axis=0)
sv_indices = np.where(alpha.value > 1e-5)[0]
b_from_dual = y[sv_indices[0]] - w_from_dual @ X[sv_indices[0]]

print("-" * 30)
print("SVM SOLVER RESULTS (Dataset 1)")
print("-" * 30)
print(f"Primal weights (w): {w_p.value}")
print(f"Dual weights (w):   {w_from_dual}")
print(f"Primal bias (b):    {b_p.value:.4f}")
print(f"Dual bias (b):      {b_from_dual:.4f}")
print("\n" + "-" * 30)

# --- VERIFICATION OF KKT CONDITION ---
print("KKT CONDITION VERIFICATION")
print("-" * 30)

# 1. Dual Feasibility
```

```python
min_alpha = np.min(alpha.value)
print(f"3. Dual Feasibility (Min alpha >= 0): {min_alpha:.2e}")

# 2. Primal Feasibility
min_margin = np.min(y * (X @ w_p.value + b_p.value))
print(f"2. Primal Feasibility (Min Margin >= 1): {min_margin:.4f}")

# 3. Complementary Slackness
# alpha_i * (y_i(w.T x_i + b) - 1) should be 0
margin_gap = y * (X @ w_p.value + b_p.value) - 1
slackness = np.abs(alpha.value * margin_gap)
print(f"4. Max Complementary Slackness: {np.max(slackness):.2e}")

# 4. Stationarity
stationarity_err = np.linalg.norm(w_p.value - w_from_dual)
print(f"1. Stationarity (||w_p - w_d||): {stationarity_err:.2e}")

print(f"   Number of Support Vectors: {len(sv_indices)}")
print("-" * 30)
```

Result

```
------------------------------
SVM SOLVER RESULTS (Dataset 1)
------------------------------
Primal weights (w): [0.33696807 0.31613427]
Dual weights (w):   [0.33696807 0.31613427]
Primal bias (b):    -0.0286
Dual bias (b):      -0.0286

------------------------------
KKT CONDITION VERIFICATION
------------------------------
1. Dual Feasibility (Min alpha >= 0): -9.29e-23
2. Primal Feasibility (Min Margin >= 1): 1.0000
3. Max Complementary Slackness: 2.37e-17
4. Stationarity (||w_p - w_d||): 7.85e-17
   Number of Support Vectors: 2
------------------------------
```

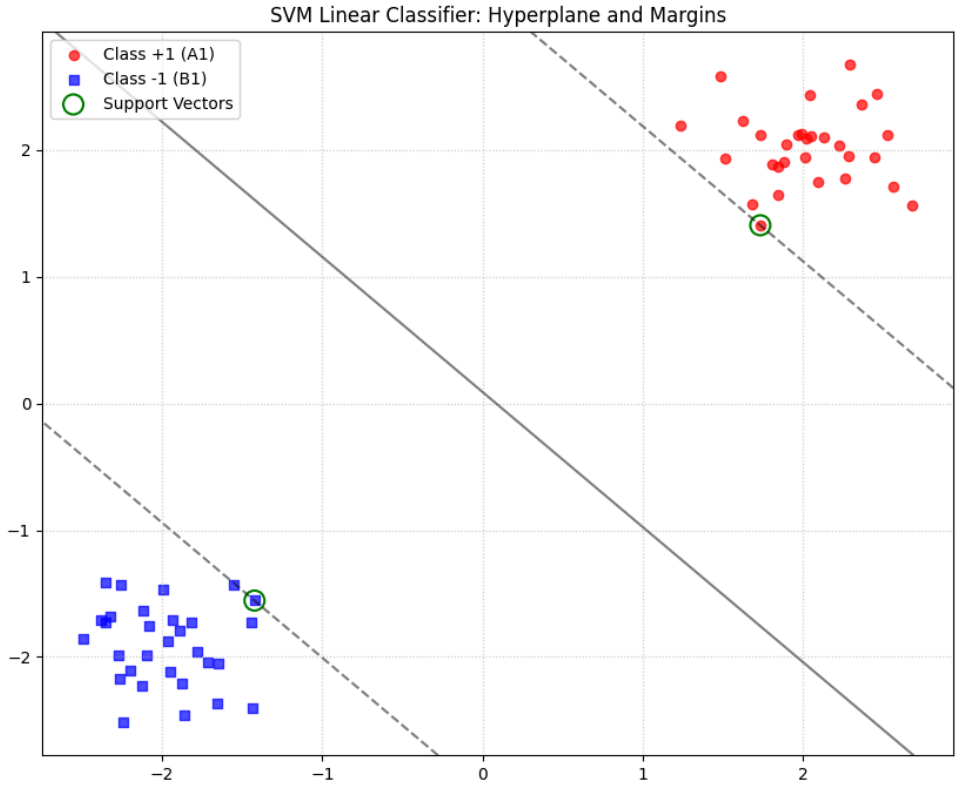
3.3 Observations

1. The primal and dual problems with a hard margin can only be solved for Dataset 1. The other two datasets lack a linear classifier; therefore, they cannot be solved using this method.
2. Both the primal and dual problems converge to the same solution, proving that they are two different ways to view and solve the same underlying problem.
3. **It turns out that the P matrix used while solving the dual problem has some negative eigenvalues that are extremely close to zero but do not converge to it. This prevents the matrix from being positive semi-definite and makes the problem non-convex. To resolve this, we use the `cvxpy.psd_wrap()` function on the P matrix to treat it as a positive semi-definite matrix.**

4. Each of the KKT conditions mentioned earlier is perfectly satisfied by the obtained solution.

4 Plotting the Hyperplane of Dataset 1

```
def plot_svm_results(setA, setB, w, b, alpha):
    plt.figure(figsize=(10, 8))

    # Plot the data points
    plt.scatter(setA[:, 0], setA[:, 1], c='blue', label='Class +1 (A1)', alpha=0.7)
    plt.scatter(setB[:, 0], setB[:, 1], c='red', label='Class -1 (B1)', alpha=0.7)

    # Define plot limits
    ax = plt.gca()
    xlim = ax.get_xlim()
    ylim = ax.get_ylim()

    # Create grid to evaluate model
    xx = np.linspace(xlim[0], xlim[1], 30)
    yy = np.linspace(ylim[0], ylim[1], 30)
    YY, XX = np.meshgrid(yy, xx)
    xy = np.vstack([XX.ravel(), YY.ravel()]).T

    # Calculate decision boundary and margins: Z = w[0]*x + w[1]*y + b
    Z = (xy @ w + b).reshape(XX.shape)

    # Plot decision boundary and margins
    # level 0 is the hyperplane, levels -1 and 1 are the margins
    ax.contour(XX, YY, Z, colors='k', levels=[-1, 0, 1], alpha=0.5,
               linestyles=['--', '-', '--'])

    # Highlight Support Vectors (where alpha > 1e-5)
    X_all = np.vstack([setA, setB])
    sv_idx = np.where(alpha > 1e-5)[0]
    plt.scatter(X_all[sv_idx, 0], X_all[sv_idx, 1], s=150,
                linewidth=1.5, facecolors='none', edgecolors='green',
                label='Support Vectors')

    plt.title("SVM Linear Classifier: Hyperplane and Margins")
    plt.legend()
    plt.grid(True, linestyle=':', alpha=0.6)
    plt.show()

# Run the plotting function using values from the previous code block
plot_svm_results(setA1, setB1, w_p.value, b_p.value, alpha.value)
```



5 The Relaxed SVM Solver

To classify Dataset 2, we fall back to a relaxed SVM solver. We introduce slack variables $\xi_i \geq 0$, which allow some points to be inside the margin or even on the wrong side of the boundary. With this, the new optimization problem becomes

$$\min_{w \in R^2, b \in R, \xi \in R^N} \quad f(w) = \frac{1}{2} \|w\|_2^2 + C \sum_{i=1}^{n} \xi_i$$

$$\text{Subject to} \quad 1 - \hat{y}_i(w^T \phi(\hat{x}_i) + b) \leq \xi_i, \quad \xi_i \geq 0, \quad \forall i \in \{1, 2, ..., N\}$$

where $C > 0$ is a large positive constant that penalizes nonzero values of the slack variables (xi_i).

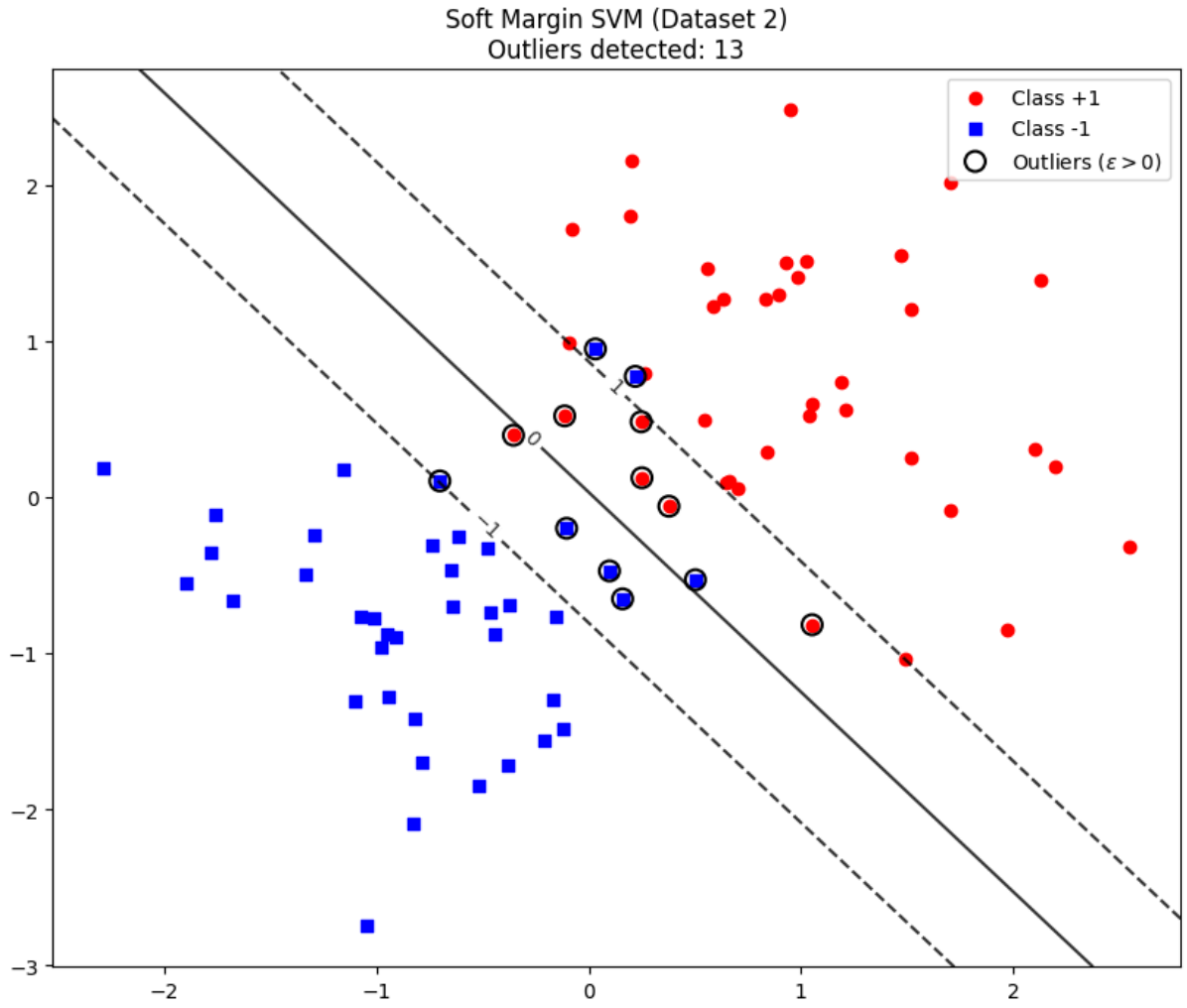
Relaxed SVM solver

```
def solve_soft_margin_svm(setA, setB, C=1.0):
    X = np.vstack([setA, setB])
    y = np.hstack([np.ones(len(setA)), -np.ones(len(setB))])
    N, d = X.shape

    w = cp.Variable(d)
    b = cp.Variable()
    xi = cp.Variable(N)  # Slack variables

    # Objective: 0.5*||w||^2 + C * sum(xi)
    obj = cp.Minimize(0.5 * cp.sum_squares(w) + C * cp.sum(xi))

    # Constraints
    constraints = [
        cp.multiply(y, X @ w + b) >= 1 - xi,
```

```
        xi >= 0
    ]

    prob = cp.Problem(obj, constraints)
    prob.solve()

    # Identify outliers: points where xi > 0 (using a small threshold for noise)
    outliers_idx = np.where(xi.value > 1e-5)[0]

    return w.value, b.value, xi.value, outliers_idx

def plot_soft_margin_results(setA, setB, w, b, xi):
    plt.figure(figsize=(10, 8))

    # Plot the data points
    plt.scatter(setA[:, 0], setA[:, 1], c='red', label='Class +1')
    plt.scatter(setB[:, 0], setB[:, 1], c='blue', label='Class -1')

    # Define plot limits
    ax = plt.gca()
    xlim = ax.get_xlim()
    ylim = ax.get_ylim()

    # Create grid to evaluate model
    xx = np.linspace(xlim[0], xlim[1], 100)
    yy = np.linspace(ylim[0], ylim[1], 100)
    YY, XX = np.meshgrid(yy, xx)
    xy = np.vstack([XX.ravel(), YY.ravel()]).T

    # Calculate decision boundary Z = w.T @ x + b
    Z = (xy @ w + b).reshape(XX.shape)

    # Plot decision boundary (0) and margins (-1, 1)
    cont = ax.contour(XX, YY, Z, colors='k', levels=[-1, 0, 1], alpha=0.8,
                      linestyles=['--', '-', '--'])
    plt.clabel(cont, inline=True, fontsize=10)

    # Highlight Outliers (xi > 1e-5)
    X_all = np.vstack([setA, setB])
    outlier_idx = np.where(xi > 1e-5)[0]
    plt.scatter(X_all[outlier_idx, 0], X_all[outlier_idx, 1], s=100,
                linewidth=1.5, facecolors='none', edgecolors='black',
                label=r'Outliers ($\epsilon > 0$)')

    plt.title(f"Soft Margin SVM (Dataset 2)\nOutliers detected: {len(outlier_idx)}")
    plt.legend()
    plt.show()

# Solve for Dataset 2
w2, b2, xi2, outliers2 = solve_soft_margin_svm(setA2, setB2, C=1.0)
```

```
66
67  print(f"Hyperplane: {w2[0]:.4f}x + {w2[1]:.4f}y + {b2:.4f} = 0")
68  print(f"Number of outliers (points violating the margin): {len(outliers2)}")
69
70  plot_soft_margin_results(setA2, setB2, w2, b2, xi2)
```

```
Result
Hyperplane: 1.5280x + 1.1947y + -0.0364 = 0
Number of outliers (points violating the margin): 13
```



5.1 Observations

1. The classifier was able to find a suitable hyperplane that is able to accurately classify the dataset points with only a handful of outliers.
2. In total, there are 13 outliers, with 6 points from the $+1$ set and 7 from the -1 set for $C = 1$.
3. The number of outliers decreases as we increase C. For example, at $C = 10$, the number of outliers is 9, and at $C = 100$ there are 8 outliers, whereas for smaller C values there are up to 28 outliers.
4. This is because the gap between the support vectors depends on C; for smaller values of C, the gap is large, whereas it is slim for high values of C.
5. We conclude that if C is very high, the classifier is prone to overfitting because of the extremely thin support vector gap, whereas it is prone to underfitting for low values of C as the margins are quite wide.

6 The Non-Linear Gaussian Classifier

The third dataset can be classified by solving the dual SVM problem with the kernel trick.

$$\max_{\lambda \geq 0} \quad \left(\sum_{i=1}^{N} \lambda_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \lambda_i \lambda_j \hat{y}_i \hat{y}_j K(x, y) \right)$$

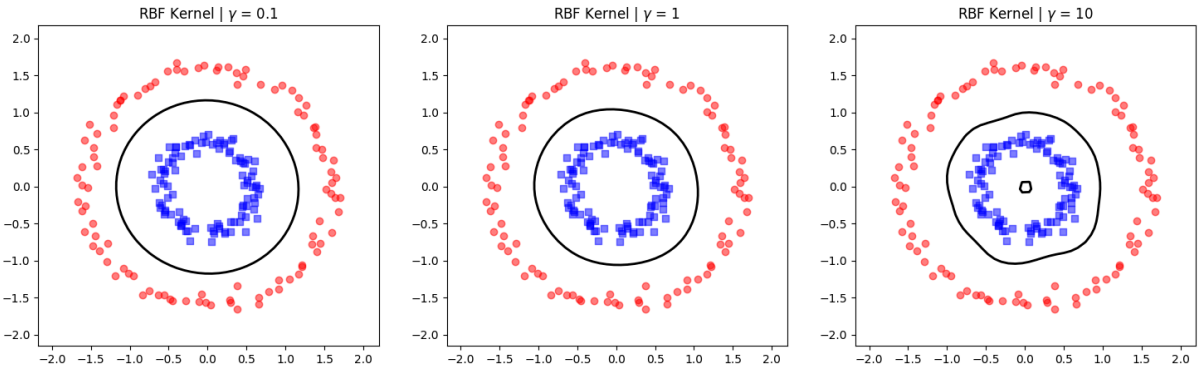
$$\text{Subject to} \quad \sum_{i=1}^{N} \lambda_i \hat{y}_i = 0, \quad 0 \leq \lambda_i \leq C, \quad \forall i \in \{N\}$$

We use the Gaussian/RBF kernel defined as

$$K(x, y) = e^{-\gamma \|x-y\|^2}$$

```
def rbf_kernel(X, Y, gamma):
    # Efficiently compute ||x-y||^2 using expansion: x^2 - 2xy + y^2
    X_sq = np.sum(X**2, axis=1).reshape(-1, 1)
    Y_sq = np.sum(Y**2, axis=1).reshape(1, -1)
    sq_dists = X_sq + Y_sq - 2 * (X @ Y.T)
    return np.exp(-gamma * sq_dists)

def solve_kernel_svm(X, y, gamma, C=1.0):
    N = X.shape[0]
    K = rbf_kernel(X, X, gamma)
    # P matrix for CVXPY: P_ij = y_i * y_j * K(x_i, x_j)
    P = np.outer(y, y) * K

    alpha = cp.Variable(N)
    obj = cp.Maximize(cp.sum(alpha) - 0.5 * cp.quad_form(alpha, cp.psd_wrap(P)))
    constraints = [alpha >= 0, alpha <= C, cp.sum(cp.multiply(alpha, y)) == 0]

    prob = cp.Problem(obj, constraints)
    prob.solve()

    # Calculate bias b using support vectors (where 0 < alpha < C)
    # Average over support vectors for stability
    sv_idx = np.where((alpha.value > 1e-4) & (alpha.value < C - 1e-4))[0]
    if len(sv_idx) == 0: # Fallback to all support vectors if none are strictly inside
        sv_idx = np.where(alpha.value > 1e-4)[0]

    # Decision function: f(x) = sum(alpha_i * y_i * K(x_i, x)) + b
    # We find b such that f(x_sv) = y_sv
    k_sv = K[:, sv_idx]
    b = np.mean(y[sv_idx] - np.sum((alpha.value * y)[:, None] * k_sv, axis=0))

    return alpha.value, b

# Dataset 3 (Circular) is the best candidate for this
X3 = np.vstack([setA3, setB3])
y3 = np.hstack([np.ones(len(setA3)), -np.ones(len(setB3))])

gammas = [0.1, 1, 10]
```

```python
fig, axes = plt.subplots(1, 3, figsize=(18, 5))

for i, g in enumerate(gammas):
    alphas, b = solve_kernel_svm(X3, y3, gamma=g, C=1.0)

    # Plotting logic
    ax = axes[i]
    ax.scatter(setA3[:, 0], setA3[:, 1], c='blue', alpha=0.5)
    ax.scatter(setB3[:, 0], setB3[:, 1], c='red', alpha=0.5)

    # Create grid for decision boundary
    x_min, x_max = X3[:, 0].min() - 0.5, X3[:, 0].max() + 0.5
    y_min, y_max = X3[:, 1].min() - 0.5, X3[:, 1].max() + 0.5
    xx, yy = np.meshgrid(np.linspace(x_min, x_max, 50), np.linspace(y_min, y_max, 50))
    grid_points = np.c_[xx.ravel(), yy.ravel()]

    # Evaluate decision function on grid
    K_grid = rbf_kernel(X3, grid_points, gamma=g)
    Z = (np.sum((alphas * y3)[:, None] * K_grid, axis=0) + b).reshape(xx.shape)

    ax.contour(xx, yy, Z, levels=[0], colors='black', linewidths=2)
    ax.set_title(fr"RBF Kernel | $\gamma$ = {g}")
plt.show()
```



6.1 Observations

1. The RBF kernel was able to find a circular hyperplane perfectly separating class $+1$ and -1 with no outliers.
2. For different values of γ, the shape of the boundary varies significantly:
   - For very small values of γ, the boundary curve is exactly circular with almost equal radial separation between the inner points and outer points.
   - For moderate values of γ, the curve becomes a little distorted with a lesser radial separation between the inner points and more between the outer points.
   - For higher values of γ, the curve becomes most distorted and is almost tight with the inner points and far from the outer points. The boundary distorts in a way that it is able to separate the two classes even with a tighter margin for outliers.
3. This concludes that for very high values of γ, the classifier is prone to overfitting, whereas for very small values, the produced model will be underfitted.

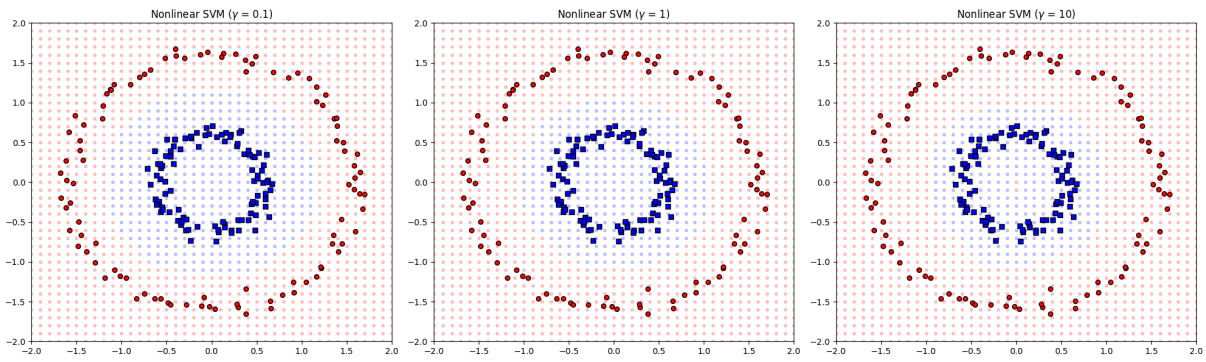
7 Classification of Points in R^2 Space

We take the R^2 vector space and classify each point in it to determine if it belongs to class $+1$ or -1. This visualization will show how the decision boundary changes as the kernel width parameter γ varies.

To determine if a point belongs to class $+1$ or -1, we use the following equation

$$\text{sgn}\left[\sum_{i=1}^{N} \lambda_i^* \hat{y}_i K(x, y) + b^*\right]$$

```
resolution = 0.1
grid_range = np.arange(-2, 2 + resolution, resolution)
xx, yy = np.meshgrid(grid_range, grid_range)
grid_points = np.c_[xx.ravel(), yy.ravel()]

fig, axes = plt.subplots(1, 3, figsize=(20, 6))

for i, gamma in enumerate(gammas):
    alpha_vals, b_val = solve_kernel_svm(X3, y3, gamma)

    # Predict on grid: f(x) = sign(sum(alpha_i * y_i * K(x_i, x_grid)) + b)
    K_grid = rbf_kernel(X3, grid_points, gamma)
    decision_values = np.sum((alpha_vals * y3)[:, None] * K_grid, axis=0) + b_val
    grid_labels = np.sign(decision_values)

    axes[i].scatter(grid_points[grid_labels == 1, 0], grid_points[grid_labels == 1, 1],
                    color='blue', s=10, alpha=0.2, label='Grid +1')
    axes[i].scatter(grid_points[grid_labels == -1, 0], grid_points[grid_labels == -1, 1],
                    color='red', s=10, alpha=0.2, label='Grid -1')

    axes[i].scatter(setA3[:, 0], setA3[:, 1], c='blue', edgecolors='k', s=30, label='Data +1')
    axes[i].scatter(setB3[:, 0], setB3[:, 1], c='red', edgecolors='k', s=30, label='Data -1')

    axes[i].set_title(rf"Nonlinear SVM ($\gamma$ = {gamma})")
    axes[i].set_xlim([-2, 2])
    axes[i].set_ylim([-2, 2])

plt.tight_layout()
plt.show()
```



8 Conclusion

1. The given datasets contain three differently distributed sets of data points, each one requiring a different type of classifier.
   - Dataset 1 has clear linearly separable points that can be classified easily with a linear SVM classifier with hard margins.
   - Dataset 2 has outlier points that cannot be separated by a linear hard margin classifier, so we relax the margin criteria to allow for some tolerance of errors.
   - Dataset 3 cannot be separated by a linear classifier at all, so we use the kernel trick with the RBF kernel to classify the points.
2. For each classification model, we were able to successfully classify the points into their respective classes.
   - The solution produced by the hard margin linear classifier used for Dataset 1 satisfies the KKT conditions.
   - The soft-margin linear classifier used to classify Dataset 2 allowed some outliers, the number of which depended on the C hyperparameter.
   - The non-linear classifier used on Dataset 3 perfectly classified the points into either of the classes with no outliers.
3. The shape of the boundary curve generated by the RBF kernel heavily depended on the γ parameter, with higher values of γ causing more distortion to the circle.