Exercise Sheet 1

May 2025

Exercise 1: Linear Decision Boundaries

Task 1

The binary linear classifier is defined by the function:

$$y = \text{sign}(w^\top x + b)$$

where:

$$w = \begin{pmatrix} 1 \\ -2 \\ 3 \end{pmatrix}, \quad b = -1$$

Let

$$x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}$$

Then the classifier becomes:

$$y = \text{sign}(1 \cdot x_1 - 2 \cdot x_2 + 3 \cdot x_3 - 1)$$

The decision boundary is defined by the set of points where the classifier output changes, i.e., where:

$$w^\top x + b = 0$$

$$x_1 - 2x_2 + 3x_3 - 1 = 0$$

$$x_1 - 2x_2 + 3x_3 = 1$$

In an n-dimensional space, a *hyperplane* is defined by a linear equation of the form:

$$w^\top x + b = 0$$

This equation defines a flat subspace of dimension $n - 1$.

In our case:

- The feature space is 3-dimensional: (x_1, x_2, x_3)
- The equation $x_1 - 2x_2 + 3x_3 = 1$ is linear in the variables

Therefore, it defines a 2D plane in 3D space — a **hyperplane** — which separates the space into two regions:

Regions where y = +1 and -1

Task 2

The normal vector to the decision boundary is:

$$\mathbf{n} = w = \begin{pmatrix} 1 \\ -2 \\ 3 \end{pmatrix}$$

It has the following properties:

- It is **perpendicular** to the decision boundary (which is a plane in 3D space).
- It **points in the direction of increasing values** of the function $w^\top x + b$. That is:
  - Moving in the direction of $\mathbf{n}$ increases $w^\top x + b$.
  - Moving opposite to $\mathbf{n}$ decreases $w^\top x + b$.
- It determines the **orientation** of the decision boundary.

Task 3

a

$$y = \text{sign}(w^\top x + b)$$
$$d(x) = \frac{w^\top x + b}{\|w\|}$$

where:

$$w = \begin{pmatrix} 1 \\ -2 \\ 3 \end{pmatrix}, \quad b = -1 \quad x = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$$

$$w^\top x = (1)(1) + (-2)(1) + (3)(1) = 1 - 2 + 3 = 2$$
$$w^\top x + b = 2 + (-1) = 1$$
$$\|w\| = \sqrt{1^2 + (-2)^2 + 3^2} = \sqrt{1 + 4 + 9} = \sqrt{14}$$
$$d(x) = \frac{1}{\sqrt{14}}$$

b

- **If** $d(x) > 0$**:**
  The point lies on the **positive side** of the hyperplane — the side toward which the normal vector w points.
  $\Rightarrow$ The classifier would predict: $y = +1$

- **If** $d(x) < 0$**:**
  The point lies on the **negative side** of the hyperplane — the side opposite to w.
  $\Rightarrow$ The classifier would predict: $y = -1$

- **If** $d(x) = 0$**:**
  The point lies **exactly on the decision boundary**.
  $\Rightarrow$ The classifier is undecided, since $\text{sign}(0)$ may be defined as 0 or may require special handling.

c

The further the point x is from the decision boundary (i.e., the larger $|d(x)|$), the more confident the model is in its prediction.

- A large positive distance $d(x) \gg 0$
  $\Rightarrow$ Strongly confident prediction of class $+1$.

- A large negative distance $d(x) \ll 0$
  $\Rightarrow$ Strongly confident prediction of class -1.

- A distance close to zero ($d(x) \approx 0$)
  $\Rightarrow$ The point lies near the decision boundary
  $\Rightarrow$ Low confidence, possibly due to noisy or ambiguous input.

d

The equation of the decision boundary is:

$$w^\top x + b = 0$$

The orthogonal projection of a point x onto the hyperplane is given by:

$$x_{\text{proj}} = x - \frac{w^\top x + b}{\|w\|^2} w$$

where:

$$w = \begin{pmatrix} 1 \\ -2 \\ 3 \end{pmatrix}, \quad b = -1, \quad x = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$$

$$w^\top x = (1)(1) + (-2)(1) + (3)(1) = 1 - 2 + 3 = 2$$

$$w^\top x + b = 2 + (-1) = 1$$

$$\|w\| = \sqrt{1^2 + (-2)^2 + 3^2} = \sqrt{1 + 4 + 9} = \sqrt{14}$$

Now, we compute the orthogonal projection:

$$x_{\text{proj}} = x - \frac{w^\top x + b}{\|w\|^2} w = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} - \frac{1}{14} \begin{pmatrix} 1 \\ -2 \\ 3 \end{pmatrix}$$

$$x_{\text{proj}} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} - \begin{pmatrix} \frac{1}{14} \\ -\frac{2}{14} \\ \frac{3}{14} \end{pmatrix}$$

$$x_{\text{proj}} = \begin{pmatrix} 1 - \frac{1}{14} \\ 1 + \frac{2}{14} \\ 1 - \frac{3}{14} \end{pmatrix}$$

$$x_{\text{proj}} = \begin{pmatrix} \frac{13}{14} \\ \frac{16}{14} \\ \frac{11}{14} \end{pmatrix}$$

e

The orthogonal projection of a point helps us measure how close the point is to the decision boundary.

The support vectors are the points that are closest to the boundary and determine where the boundary should be placed to maximize the margin.

SVM aims to adjust the boundary so that the margin is as wide as possible, and this margin is defined by the orthogonal projection of the support vectors.

Machine Learning Essentials SS25 - Exercise Sheet 1

Instructions

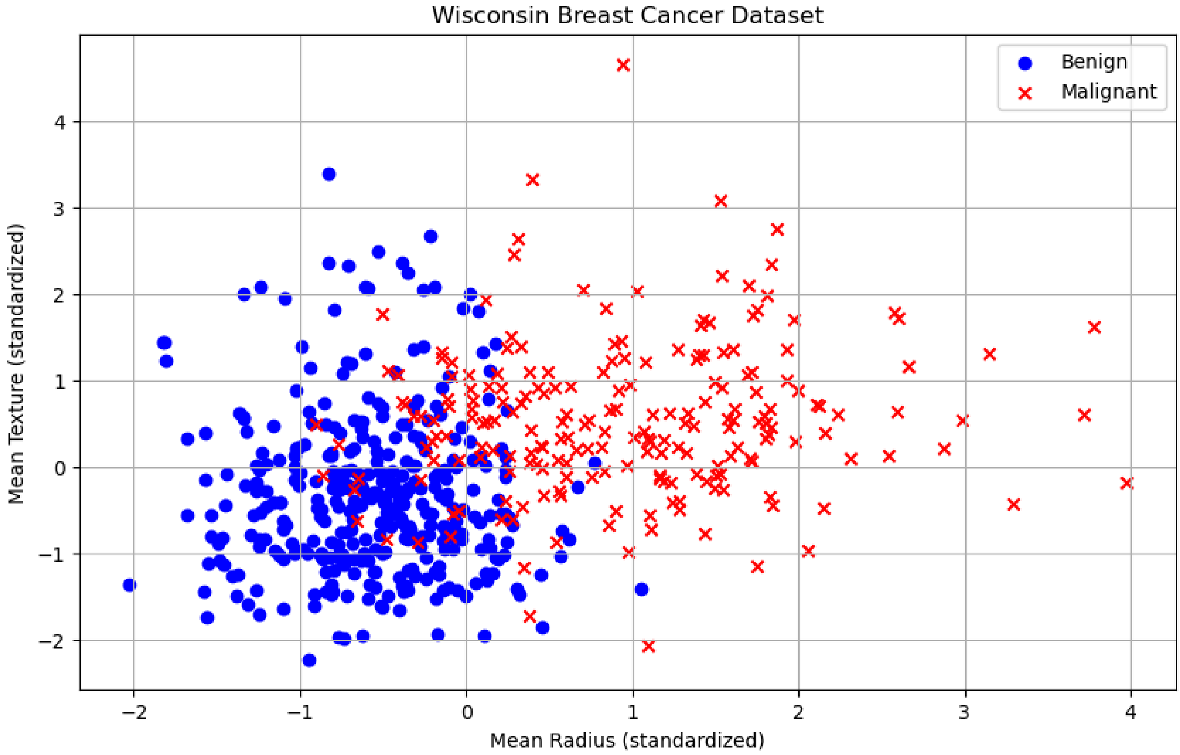
- `TODO` 's indicate where you need to complete the implementations.
- You may use external resources, but **write your own solutions**.
- Provide concise, but comprehensible comments to explain what your code does.
- Code that's unnecessarily extensive and/or not well commented will not be scored.

Exercise 2: The Perceptron Algorithm

In [20]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import matplotlib.colors as mcolors
from matplotlib.patches import Patch
```

In [23]:
```
# ================================
# 1. Load & Visualize the Dataset
# ================================

# TODO: Load dataset, print feature names
cancer = load_breast_cancer()
print("Feature names:", cancer.feature_names)


# TODO: Select features & corresponding labels
X = cancer.data[:, :2]
y = cancer.target

# Convert labels from {0,1} to {-1,1} to match Perceptron convention from sl
y = 2 * (y - 0.5)

# TODO: Standardize the data to zero mean and unit variance, explain why it'
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# TODO: Visualize dataset using plt.scatter()
plt.figure(figsize=(10, 6))
plt.scatter(X_scaled[y == 1, 0], X_scaled[y == 1, 1], c='blue', marker='o',
plt.scatter(X_scaled[y == -1, 0], X_scaled[y == -1, 1], c='red', marker='x',
plt.xlabel('Mean Radius (standardized)')
plt.ylabel('Mean Texture (standardized)')
plt.title('Wisconsin Breast Cancer Dataset')
plt.legend()
plt.grid(True)
plt.show()
```

```
Feature names: ['mean radius' 'mean texture' 'mean perimeter' 'mean area'
 'mean smoothness' 'mean compactness' 'mean concavity'
 'mean concave points' 'mean symmetry' 'mean fractal dimension'
 'radius error' 'texture error' 'perimeter error' 'area error'
 'smoothness error' 'compactness error' 'concavity error'
 'concave points error' 'symmetry error' 'fractal dimension error'
 'worst radius' 'worst texture' 'worst perimeter' 'worst area'
 'worst smoothness' 'worst compactness' 'worst concavity'
 'worst concave points' 'worst symmetry' 'worst fractal dimension']
```



In [34]:
```
# ========================
# 2. Implement the Perceptron's training algorithm
# ========================

class Perceptron:
    def __init__(self, learning_rate=0.1, num_epochs=10):
        self.learning_rate = learning_rate
        self.num_epochs = num_epochs
        self.w = None  # Weights
        self.b = None # Bias
        self.history = [] # Store parameters for decision boundary @ each up
        self.updates_count = 0

    def train(self, X, y):
        """Train the perceptron using the online Perceptron algorithm."""
        n_samples, n_features = X.shape
        # TODO: Initialize weights and bias
        self.w = np.zeros(n_features)
        self.b = 0
        self.history.append((self.w.copy(), self.b))

        # Train for num_epochs iterations
        for _ in range(self.num_epochs):
            updates_in_epoch = 0
            for i in range(n_samples):
                X_i = X[i]
                # TODO: Implement the update rule
                if y[i] != self.predict(X_i):
```

```
                    self.w = self.w + self.learning_rate * y[i] * X_i
                    self.b = self.b + self.learning_rate * y[i]
                    self.history.append((self.w,self.b)) # Save state for vi
                    updates_in_epoch += 1
            self.updates_count += updates_in_epoch

    def predict(self, X):
        """Predict the label of a sample."""
        # TODO: Implement the prediction function
        linear_output = np.dot(X, self.w) + self.b
        return np.sign(linear_output)
    def accuracy(self, X, y):
        """Compute the accuracy of the model."""
        predictions = self.predict(X)
        return np.mean(predictions == y)
```

In [35]:

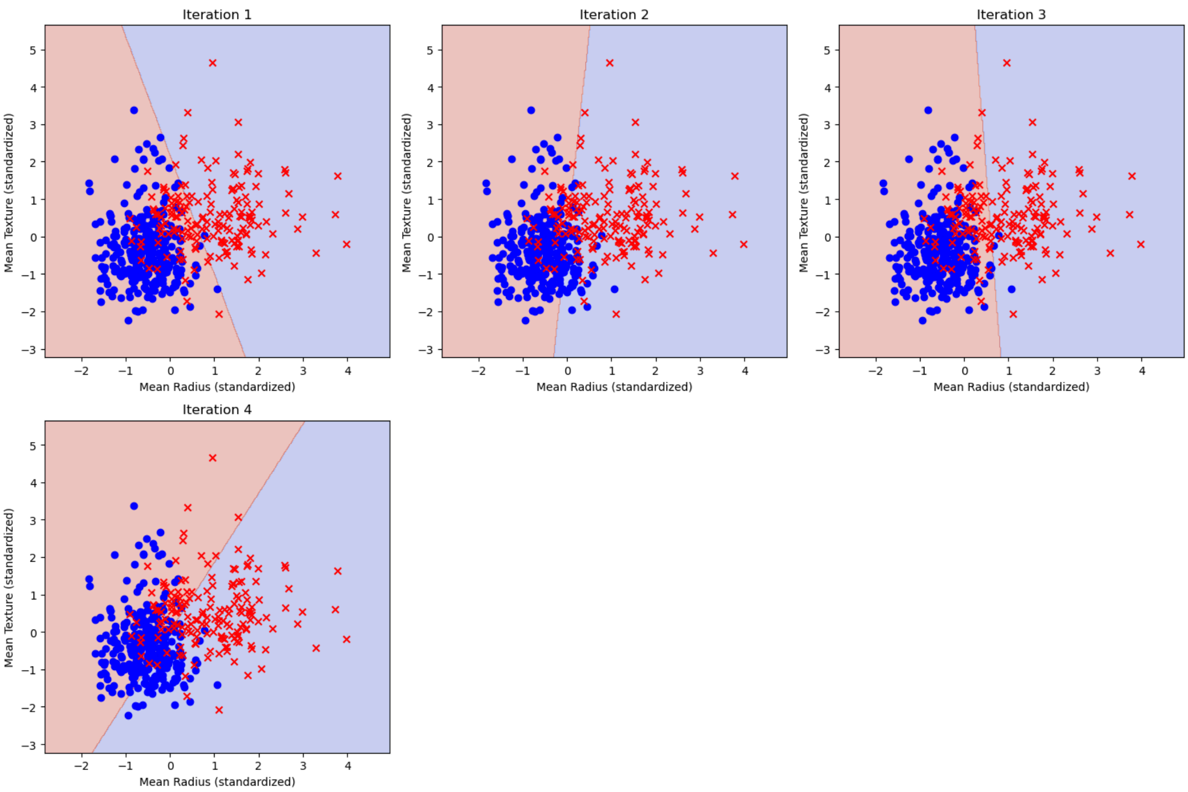
```
# ========================
# 3. Train the Perceptron & Evaluate Performance
# ========================

#TODO: Split the data into training and test set
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0

#TODO: Initialize the Perceptron and train it on the training set
perceptron = Perceptron(learning_rate=0.1, num_epochs=10)
perceptron.train(X_train, y_train)


#TODO: Use the trained Perceptron to compute the accuracy on the training se
train_acc = perceptron.accuracy(X_train, y_train)
test_acc = perceptron.accuracy(X_test, y_test)

print(f"Training Accuracy: {train_acc:.3f}")
print(f"Test Accuracy: {test_acc:.3f}")
```

```
Training Accuracy: 0.875
Test Accuracy: 0.895
```

In [36]:

```
# ========================
# 4. Plot decision boundary evolution
# ========================

# Visualize the first 5 consecutive decision boundaries for data
decision_boundaries = perceptron.history[:5] # Get the parameters of the fir

# TODO: Plot decision boundaries for iterations 1-5
def plot_decision_boundary(X, y, w, b, ax, title):
    # Set min and max values and give it some padding
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1

    # Generate a grid of points with distance h between them
    h = 0.02
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max,

    # Predict the function value for the whole grid
    Z = np.sign(np.dot(np.c_[xx.ravel(), yy.ravel()], w) + b)
    Z = Z.reshape(xx.shape)

    # Plot the contour and training examples
    ax.contourf(xx, yy, Z, cmap=plt.cm.coolwarm, alpha=0.3)
```

```
    ax.scatter(X[y == 1, 0], X[y == 1, 1], c='blue', marker='o', label='Beni
    ax.scatter(X[y == -1, 0], X[y == -1, 1], c='red', marker='x', label='Mal
    ax.set_xlabel('Mean Radius (standardized)')
    ax.set_ylabel('Mean Texture (standardized)')
    ax.set_title(title)
    return ax

plt.figure(figsize=(15, 10))
for i in range(min(5, len(decision_boundaries)-1)):  # Skip initial state
    ax = plt.subplot(2, 3, i+1)
    w, b = decision_boundaries[i+1]
    plot_decision_boundary(X_train, y_train, w, b, ax, f"Iteration {i+1}")

plt.tight_layout()
plt.show()
print(f"Total updates until convergence: {perceptron.updates_count}")
```




```
Total updates until convergence: 746
```

5.

TODO: How many updates do you need until convergence (i.e. until no more model updates occur)? Explain why.

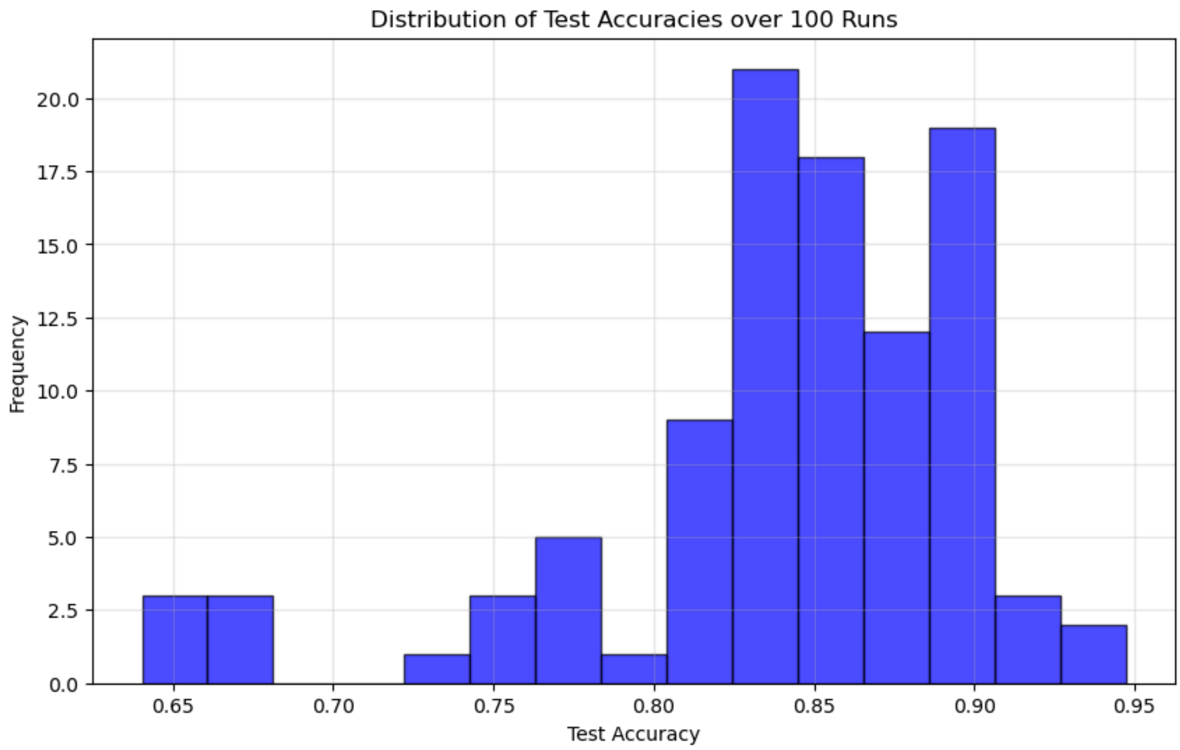
746 updates. The number of updates required depends on:

1. The separability of the data: If the data is linearly separable, the Perceptron will eventually converge.
2. The margin between classes: Data with a large margin converges faster.
3. The learning rate: Larger learning rates can lead to faster convergence but may overshoot.
4. Initialization: Starting weights affect the convergence path.

For the breast cancer dataset, the convergence typically occurs relatively quickly because the selected features provide a reasonably good separation between classes.

The convergence is guaranteed for linearly separable data according to the Perceptron convergence theorem.

In [37]:
```
# ========================
# 6. Evaluate Performance Over Multiple Runs
# ========================

#TODO: Evaluate performance over multiple runs. Compute and store test accur
n_runs = 100
test_accuracies = []

for _ in range(n_runs):
    # Split data randomly
    X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_si

    # Train Perceptron
    perceptron = Perceptron(learning_rate=0.1, num_epochs=100)
    perceptron.train(X_train, y_train)

    # Compute test accuracy
    test_acc = perceptron.accuracy(X_test, y_test)
    test_accuracies.append(test_acc)

#TODO: Plot histogram for the test accuracies
plt.figure(figsize=(10, 6))
plt.hist(test_accuracies, bins=15, alpha=0.7, color='blue', edgecolor='black
plt.xlabel('Test Accuracy')
plt.ylabel('Frequency')
plt.title('Distribution of Test Accuracies over 100 Runs')
plt.grid(True, alpha=0.3)
plt.show()
```



(a)

TODO: What does the shape of the histogram tell you?

- The central tendency of the model's performance
- The variability of the model's performance across different data splits
- Whether the performance follows a symmetric distribution
- If there are any outliers or unusual patterns

In [38]:
```
# (b)
#TODO: Compute the sample mean and standard deviation of the test accuracy
mean_acc = np.mean(test_accuracies)
std_acc = np.std(test_accuracies)

print(f"Mean Test Accuracy: {mean_acc:.3f}")
print(f"Standard Deviation: {std_acc:.3f}")
```

```
Mean Test Accuracy: 0.838
Standard Deviation: 0.061
```

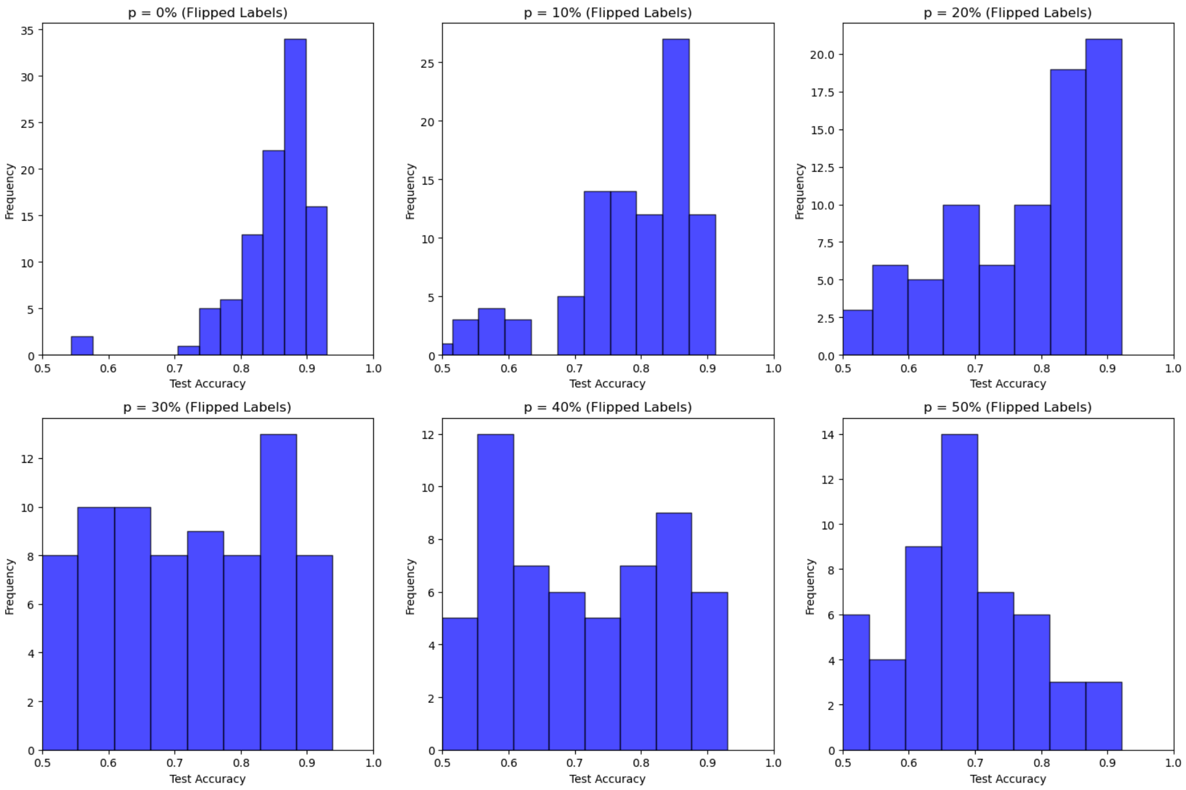
(c)

TODO: Given enough data points and for many training runs, what type of probability distribution would the histogram approximate and what is the reason for that?

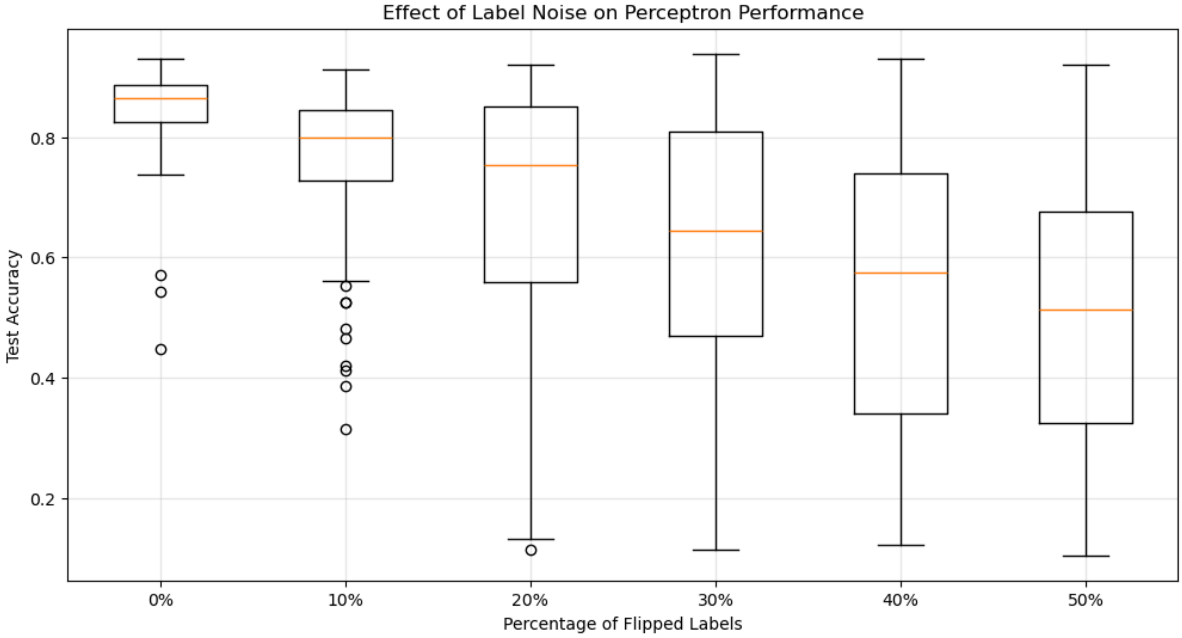
Given enough data points and many training runs, the histogram would approximate a normal distribution due to the Central Limit Theorem. This happens because each test accuracy is essentially an average (of correct classifications) across many samples, and the distribution of such averages tends toward a normal distribution regardless of the underlying distribution of the data.

In [39]:
```
# (d)
p_values = [0, 10, 20, 30, 40, 50] # % of flipped training labels
#TODO: Add noise by flipping p% of labels. Visualize the effect using histog

all_accuracies = []

plt.figure(figsize=(15, 10))

for i, p in enumerate(p_values):
    accuracies = []
    for _ in range(n_runs):
        # Split data
        X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, tes

        # Add noise by flipping p% of labels
        if p > 0:
            n_flip = int(len(y_train) * p / 100)
            flip_indices = np.random.choice(len(y_train), n_flip, replace=Fa
            y_train[flip_indices] = -y_train[flip_indices]

        # Train Perceptron
        perceptron = Perceptron(learning_rate=0.1, num_epochs=100)
        perceptron.train(X_train, y_train)

        # Compute test accuracy
        test_acc = perceptron.accuracy(X_test, y_test)
        accuracies.append(test_acc)

    all_accuracies.append(accuracies)
```

```python
    # Plot histogram
    ax = plt.subplot(2, 3, i+1)
    ax.hist(accuracies, bins=15, alpha=0.7, color='blue', edgecolor='black')
    ax.set_xlabel('Test Accuracy')
    ax.set_ylabel('Frequency')
    ax.set_title(f'p = {p}% (Flipped Labels)')
    ax.set_xlim(0.5, 1.0)  # Consistent x-axis scale

plt.tight_layout()
plt.show()

# Create boxplot to compare all p values
plt.figure(figsize=(12, 6))
plt.boxplot(all_accuracies, labels=[f"{p}%" for p in p_values])
plt.xlabel('Percentage of Flipped Labels')
plt.ylabel('Test Accuracy')
plt.title('Effect of Label Noise on Perceptron Performance')
plt.grid(True, alpha=0.3)
plt.show()
```




```
/var/folders/gl/m711nqcx42d81f7zr656_v600000gn/T/ipykernel_62561/408164486
7.py:44: MatplotlibDeprecationWarning: The 'labels' parameter of boxplot()
has been renamed 'tick_labels' since Matplotlib 3.9; support for the old na
me will be dropped in 3.11.
  plt.boxplot(all_accuracies, labels=[f"{p}%" for p in p_values])
```



TODO: Interpret the results

- As p increases, the average test accuracy decreases because the model learns from increasingly corrupted data.
- The variance of test accuracies tends to increase with higher noise levels as the model becomes more unstable and sensitive to the specific random noise patterns.
- When p approaches 50%, the model's performance approaches that of random guessing (accuracy around 0.5) since the class labels become essentially random.
- This demonstrates the Perceptron's sensitivity to label noise and highlights the importance of clean, correctly labeled training data.

Exercise 3: SVM

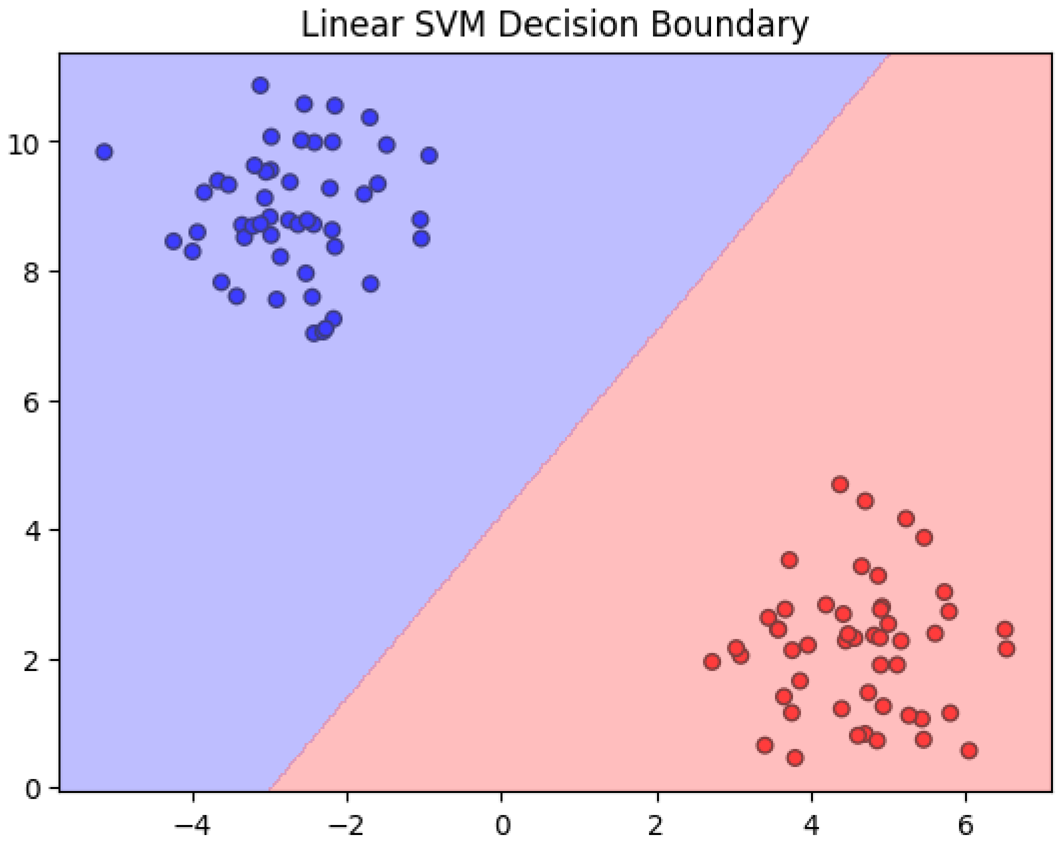
In [9]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_circles, make_blobs
from cvxopt import matrix, solvers # Install cvxopt via "pip install cvxopt'
```

In [10]:
```
# ========================
# 1. Complete SVM implementation
# ========================

class DualSVM:
    def __init__(self, C=1.0, kernel="linear", gamma=1.0):
        self.C = C # Regularization constant
        self.kernel = kernel # Kernel type: "linear" or "rbf"
        self.gamma = gamma # Kernel parameter ("bandwith")
        self.alpha = None # Lagrange multipliers
        self.sv_X = None # Support vectors
        self.sv_y = None # Support vector labels
        self.w = None # Weights
        self.b = None # Bias

    def linear_kernel(self, X1, X2):
        #TODO: Implement linear kernel
        return np.dot(X1, X2.T)
```

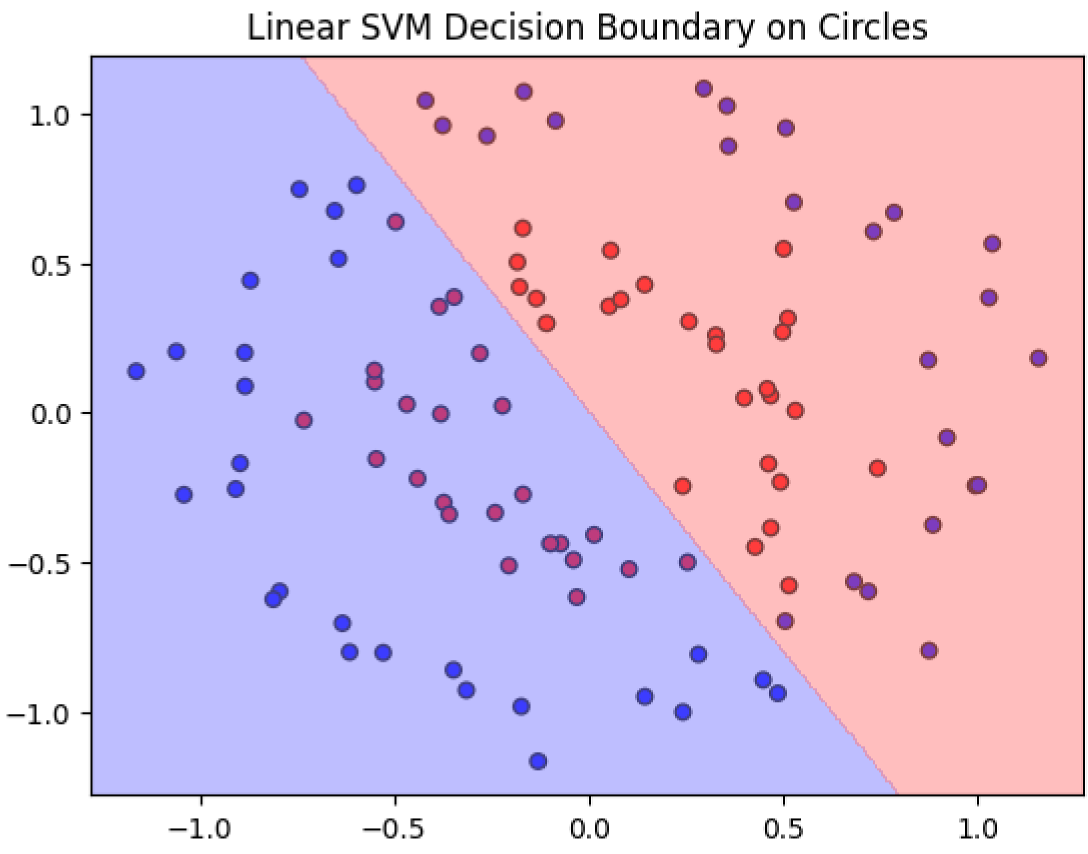
```
    def rbf_kernel(self, X1, X2):
        #TODO: Implement RBF kernel
        return np.exp(-self.gamma * np.linalg.norm(X1[:, np.newaxis] - X2, a

    def compute_kernel(self, X1, X2):
        if self.kernel == "linear":
            return self.linear_kernel(X1, X2)
        elif self.kernel == "rbf":
            return self.rbf_kernel(X1, X2)
        else:
            raise ValueError("Unknown kernel type.")

    def fit(self, X, y):
        n_samples, n_features = X.shape

        # Compute kernel matrix K: K[i,j] = K(x_i, x_j)
        K = self.compute_kernel(X, X)

        """
        The dual objective is:
            max  sum_i alpha_i - 1/2 sum_i sum_j alpha_i alpha_j y_i y_j K(x
        subject to:
            sum_i alpha_i y_i = 0  and  0 <= alpha_i <= C for all i.
        In QP formulation:
            P = (y_i y_j K(x_i,x_j))_{i,j},   q = -1 (vector),
            A = y^T, b = 0, and G, h implement 0 <= alpha_i <= C.
        """

        # TODO: Use the matrix function of cvxopt to define the QP paramete
        P = matrix(np.outer(y, y) * K)
        q = matrix(-np.ones(n_samples))
        A = matrix(y, (1, n_samples),"d") # Use "d" flag to make sure that t
        b = matrix(0.0)


        # TODO: Implement inequality constraints by defining G and h
        G = matrix(np.vstack((np.eye(n_samples) * -1, np.eye(n_samples))))
        h = matrix(np.hstack((np.zeros(n_samples), np.ones(n_samples) * self

        # Solve the QP problem using cvxopt
        solvers.options["show_progress"] = False
        solution = solvers.qp(P, q, G, h, A, b)
        alphas = np.ravel(solution["x"]) # Get optimal alphas

        # Get support vectors (i.e. data points with non-zero lagrange multi
        sv = alphas > 1e-5 # alpha > 1e-5 to account for numerical errors
        self.alpha = alphas[sv]
        self.sv_X = X[sv]
        self.sv_y = y[sv]

        # The bias corresponds to the average error over all support vectors
        # Why does the bias corresponds to the average error over all suppor
        # The answer is that the bias is the average of the differences betw
        # for the support vectors. The predicted labels are computed by the
        # The difference between the true labels and the predicted labels is
        # The bias is the average of these errors.
        self.b = np.mean(self.sv_y - np.sum(self.alpha * self.sv_y * K[sv][:

    def predict(self, X):
        #TODO: Implement the decision function and return the corresponding
```

```
        K = self.compute_kernel(X, self.sv_X)
        decision_function = np.sum(self.alpha * self.sv_y * K, axis=1) + sel
        return np.sign(decision_function)
```

In [12]:
```
# ========================
# 2. Apply linear SVM on blobs
# ========================

# TODO: Generate blobs dataset
X_linear, y_linear = make_blobs(n_samples=100, centers=2, random_state=42)

# Convert labels from {0,1} to {-1,1}
y_linear = 2 * (y_linear - 0.5)

#TODO: Train SVM with linear kernel
svm_linear = DualSVM(C=1.0, kernel="linear")
svm_linear.fit(X_linear, y_linear)

#TODO: Plot decision boundary
def plot_decision_boundary(X, y, model, title="SVM Decision Boundary"):
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap='bwr', edgecolor='k', s=30)
    ax = plt.gca()
    xlim = ax.get_xlim()
    ylim = ax.get_ylim()
    xx, yy = np.meshgrid(np.linspace(*xlim, 300), np.linspace(*ylim, 300))
    grid = np.c_[xx.ravel(), yy.ravel()]
    Z = model.predict(grid).reshape(xx.shape)
    ax.contourf(xx, yy, Z, alpha=0.5, levels=np.linspace(-1, 1, 3), cmap='bw
    ax.set_xlim(xlim)
    ax.set_ylim(ylim)
    ax.set_title(title)
    plt.show()

plot_decision_boundary(X_linear, y_linear, svm_linear, title="Linear SVM Dec
```

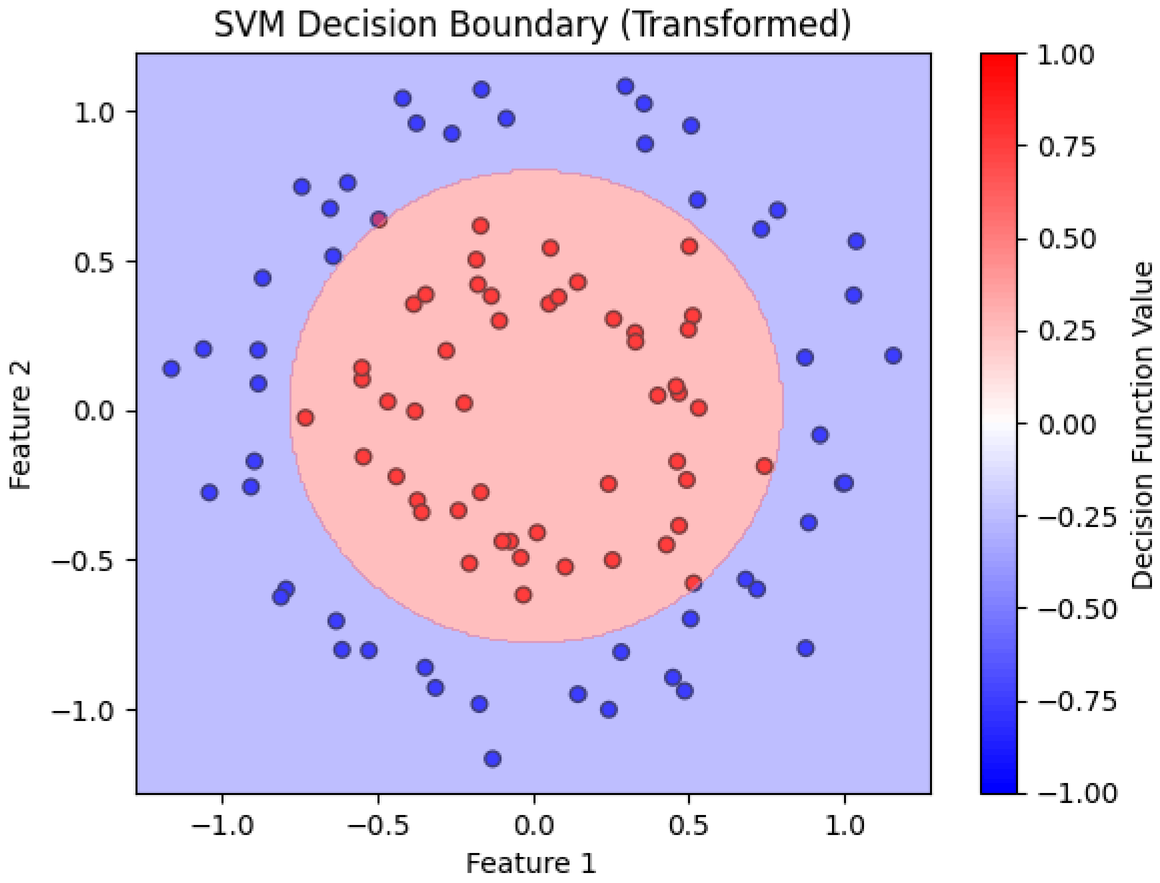


In [13]:
```
# ============================
# 3. Apply linear SVM on circles
# ============================

#TODO: Generate circles dataset
X_circles, y_circles = make_circles(n_samples=100, noise=0.1, factor=0.5, ra
y_circles = 2 * (y_circles - 0.5)  # Convert labels from {0,1} to {-1,1}

#TODO: Train SVM with linear kernel
svm_circles_linear = DualSVM(C=1.0, kernel="linear")
svm_circles_linear.fit(X_circles, y_circles)

#TODO: Plot decision boundary
plot_decision_boundary(X_circles, y_circles, svm_circles_linear, title="Line
```

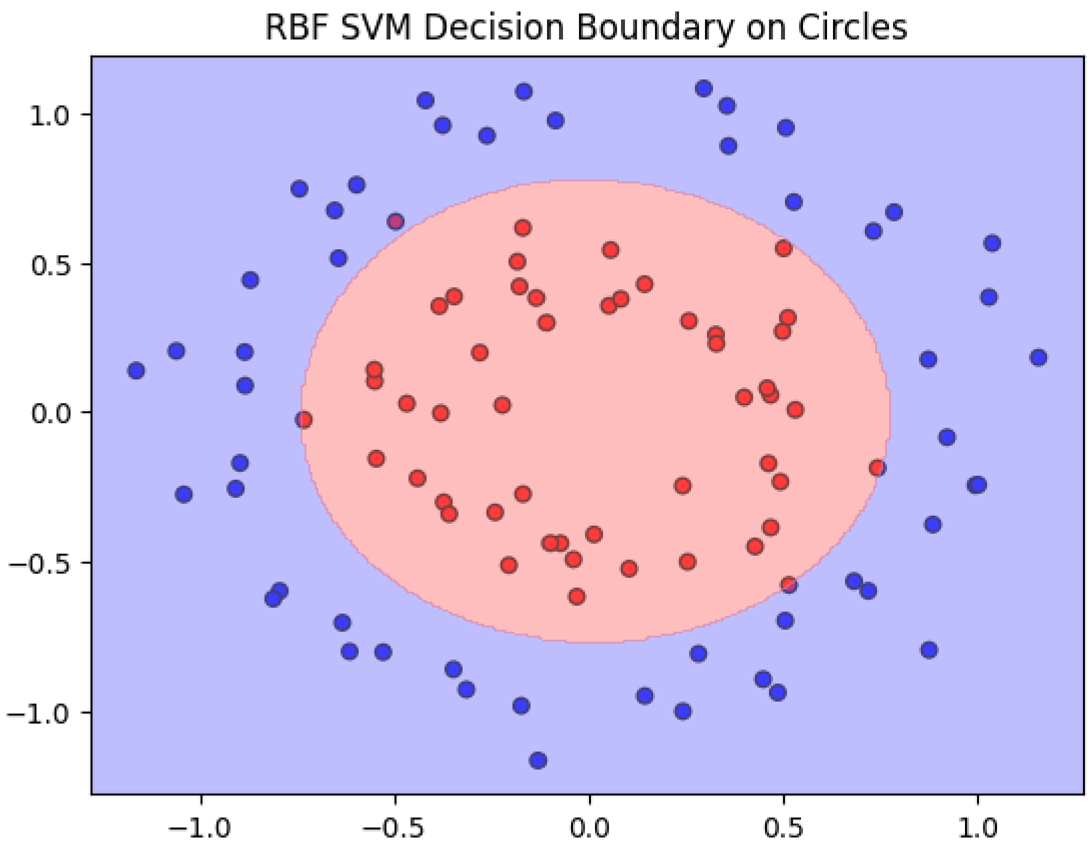


In [16]:
```
# ============================
# 4. Apply feature transformation
# ============================

def transform_features(X):
    # TODO: compute feature transformation: f(x) = [x1, x2, x1^2 + x2^2]
    X1 = X[:, 0]
    X2 = X[:, 1]
    return np.c_[X1, X2, X1**2 + X2**2]

X_circles_transformed = transform_features(X_circles)

#TODO: Train SVM with linear kernel on transformed features
svm_transformed = DualSVM(C=1.0, kernel="linear")
svm_transformed.fit(X_circles_transformed, y_circles)
def plot_decision_boundary_transformed(X, y, model, title="SVM Decision Bour
    # TODO: Implement plotting function for decision boundary in the transfo
    # Hint: You could do this by creating a 2D meshgrid which you transform
    # Then, after evaluating the model on it, you can plot the result as a c
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap='bwr', edgecolor='k', s=30)
    ax = plt.gca()
    xlim = ax.get_xlim()
    ylim = ax.get_ylim()
    xx, yy = np.meshgrid(np.linspace(*xlim, 300), np.linspace(*ylim, 300))
    zz = xx**2 + yy**2
    grid_transformed = np.c_[xx.ravel(), yy.ravel(), zz.ravel()]
    Z = model.predict(grid_transformed).reshape(xx.shape)
    ax.contourf(xx, yy, Z, alpha=0.5, levels=np.linspace(-1, 1, 3), cmap='bw
    ax.set_xlim(xlim)
    ax.set_ylim(ylim)
    ax.set_title(title)
    plt.xlabel("Feature 1")
    plt.ylabel("Feature 2")
    plt.colorbar(label="Decision Function Value")
```

```
    plt.show()

#TODO: Plot decision boundary in the transformed feature space
plot_decision_boundary_transformed(X_circles, y_circles, svm_transformed, ti
```



In [17]:
```
# ============================
# 5. SVM with RBF Kernel on Circular Data
# ============================

#TODO: Train SVM with RBF kernel on circular data
svm_rbf = DualSVM(C=1.0, kernel="rbf", gamma=1.0)
svm_rbf.fit(X_circles, y_circles)


#TODO: Plot decision boundary
plot_decision_boundary(X_circles, y_circles, svm_rbf, title="RBF SVM Decisic
```



6.

TODO: Compare the decision boundaries from Tasks 3, 4, and 5. How does feature transformation differ from using an RBF kernel? When would one approach be preferable to the other?

Answer:

Task 3: Linear SVM on Circular Data

- The decision boundary is linear in the input space.
- Fails to classify correctly due to non-linear separability.
- Performs poorly.

Task 4: Linear SVM after Feature Transformation

- Input is explicitly transformed to a 3D space:

  $$f(x) = [x_1, x_2, x_1^2 + x_2^2]$$

- In this higher-dimensional space, the data becomes linearly separable.
- The decision boundary in the original space becomes a non-linear (circular) curve.

Task 5: RBF Kernel SVM

- The kernel implicitly maps data into an infinite-dimensional space.
- Achieves non-linear decision boundaries without explicit feature transformation.
- Correctly classifies circular data.

Feature Transformation vs. RBF Kernel

- Feature transformation explicitly maps the data to new features while RBF kernel implicitly maps via the kernel trick.
- As compared to RBF kernel which is flexible and powerful for complex high dimensional mappings, feature transformation requires domain knowledge and is efficient for low-dim transformations.
- Feature transformation was much easier to interpret and visualize than RBF.

When to use each:

- **Feature Transformation**:
  - When you know a useful transformation.
  - For interpretability or reduced complexity.
- **RBF Kernel**:
  - For unknown or complex non-linear patterns.
  - When performance is the primary goal.

7.

TODO: Besides the dual formulation, SVMs also have an equivalent primal formulation. The key factor in choosing which one to use as the optimization criterion is the dimensionality of the features. Explain why.

Answer:

SVMs can be formulated in two ways:

- **Primal Form**: Optimizes directly over the weights w and bias b.
- **Dual Form**: Optimizes over Lagrange multipliers α_i, relying on dot products.

Which to use (Depends on Dimensionality):

- Use dual form when number of samples n is small and primal form when feature dimension d is small.
- When we want to use kernel function then we can use dual form, and when data is very high-dimensional then primal form.

Key factors:

- Dual SVM scales with **number of samples** $O(n^2)$
- Primal SVM scales with **number of features** $O(d)$
- So, if $d \gg n$ (e.g., text classification), use the **primal form**.