In [80]:
```
# Import necessary libraries
from sklearn.datasets import load_breast_cancer
# Load the breast cancer dataset
data = load_breast_cancer()
```

In [81]:
```
# Split the data into 60% training, 20% testing and 20% validation sets
from sklearn.model_selection import train_test_split
# Split the data into training and temporary sets (60% train, 40% temp)
X_train, X_temp, y_train, y_temp = train_test_split(data.data, data.target,
# Split the temporary set into testing and validation sets (20% test, 20% va
X_test, X_val, y_test, y_val = train_test_split(X_temp, y_temp, test_size=0.
```

Setup of models and variables

Exercise-1: Data Preparation

1.1 Load the dataset, inspect feature names and target distribution. Comment on dataset imbalance.

In [82]:
```
import pandas as pd
# Inspect feature names
print("Feature names:\n")
# one item per line
for feature in data.feature_names:
    print(feature)
# Inspect target names
print("\nTarget names:\n")
print(data.target_names)
# Inspect datset imbalance
print("\nTarget distribution:\n")
# Create a DataFrame
y = data.target
df = pd.DataFrame({"target": y})
# Map numeric values to names
df["target_name"] = df["target"].map({0: "malignant", 1: "benign"})
print(df["target_name"].value_counts())
# Percentage distribution
print(df["target_name"].value_counts(normalize=True))
```

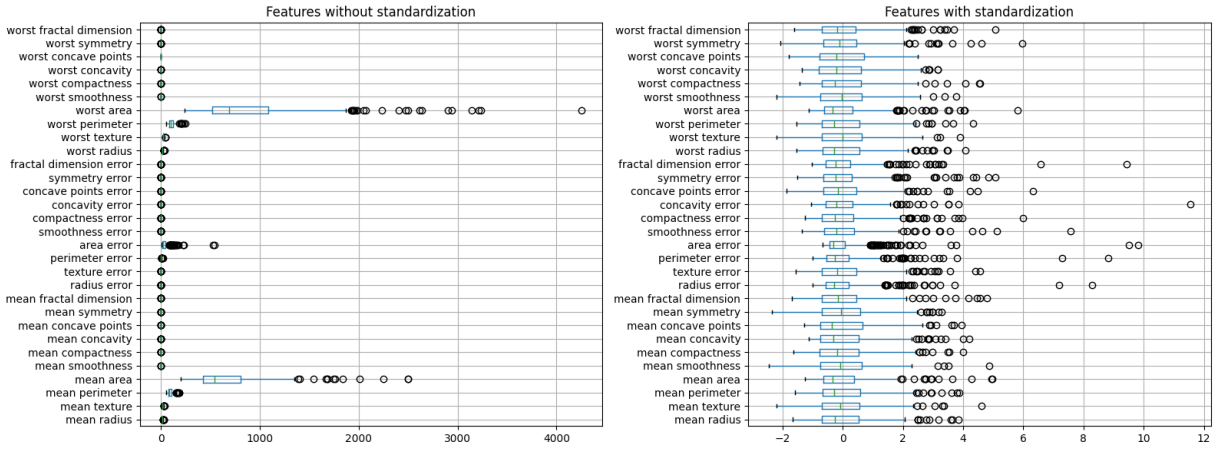
```
Feature names:

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

Target names:

['malignant' 'benign']

Target distribution:

target_name
benign       357
malignant    212
Name: count, dtype: int64
target_name
benign       0.627417
malignant    0.372583
Name: proportion, dtype: float64
```

Benign cases make up about 63% of the dataset (357 samples), while **malignant** cases make up about 37% (212 samples). This indicates a moderate class imbalance.

1.2 Analyze all features with and without standardization (i.e., zero mean and unit variance). Plot the feature analysis with and without

standardization and decide which version is more suitable.

In [83]:
```
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler

# Standardize the features
scaler = StandardScaler()
# X_train_scaled = (X_train - mean) / std
X_train_scaled = scaler.fit_transform(X_train)

# Create DataFrames for easier plotting
df = pd.DataFrame(X_train, columns=data.feature_names)
df_scaled = pd.DataFrame(X_train_scaled, columns=data.feature_names)

plt.figure(figsize=(16, 6))

# Raw features
plt.subplot(1, 2, 1)
df.boxplot(vert=False)
plt.title("Features without standardization")

# Standardized features
plt.subplot(1, 2, 2)
df_scaled.boxplot(vert=False)
plt.title("Features with standardization")

plt.tight_layout()

plt.show()
```



Standardization is the process of centering each feature by removing its mean and scaling it to unit variance. Without standardization, features with larger scales (e.g., mean area, worst area) dominate the analysis, since their values are much higher than those of smaller-scale features. After standardization, all features are on the same scale, so no single feature dominates purely because of its magnitude.

- Most suitable: **with standarization**

1.3 Comment on importance of three way split with respect to hyperparameter search and robustness of any learned model.

The three way split refers to the splitting of the dataset into train, test and validate.

- **Training set**: this is used to fit the model (learn the weights/parameters)
- **Validate set**: Used to tune the hyperparameters (a hyperparameter is parameter that can be set during a models learning process)
- **Test set**: Used only to evalute the model, with an unseen part of the dataset

Hyperparameter search discovers an optimal set of hyperparameters that produces the best model performance. This is done with the **Validate set**. The test set purpose is to evalute the model on unseen data. This is crucial for assessing the robustness of the model. If not for the test there would be greater risk of overfitting the hyperparameters to the test data.

Exercise-2 Decision trees

2.1 Train a Decision Tree classifier using default parameters. Evaluate it on validation sets from original splits (report accuracy mean and std).

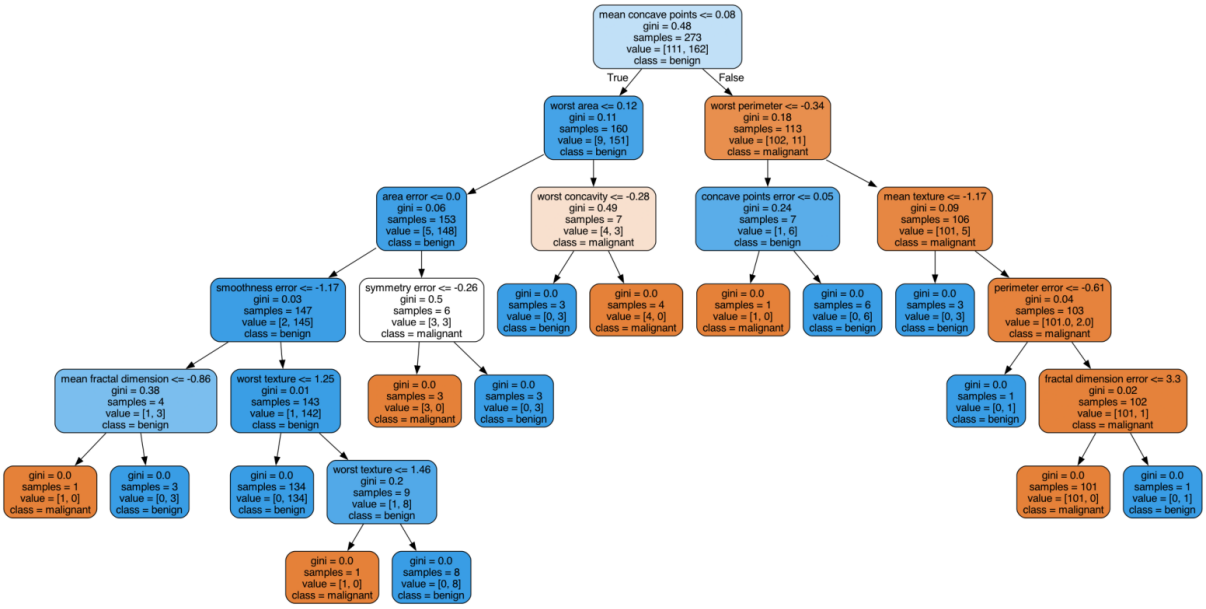
In [84]:
```
from sklearn.tree import DecisionTreeClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.model_selection import KFold, cross_val_score
from sklearn.metrics import accuracy_score

# Build a pipeline with scaling and decision tree classifier
dt_model = Pipeline([
    ('scaler', StandardScaler()),
    ('clf', DecisionTreeClassifier(random_state=42)),
])

# Fit on training and evaluate on validation set
dt_model.fit(X_train, y_train)
```

```
y_val_pred = dt_model.predict(X_val)
val_accuracy = accuracy_score(y_val, y_val_pred)
# How well does the model perform on the validation set
print(f"Validation Accuracy on the validation set: {val_accuracy:.4f}")

# Define CV strategy (5 folds, shuffled for randomness)
kf = KFold(n_splits=5, shuffle=True, random_state=42)
# Run cross-validation on training data
cv_scores_dt_model = cross_val_score(dt_model, X_train, y_train, cv=kf, scor
# Print results
print("Cross-validation scores:", cv_scores_dt_model)
print(f"Mean accuracy: {cv_scores_dt_model.mean():.4f}")
print(f"Std accuracy: {cv_scores_dt_model.std():.4f}")
```

```
Validation Accuracy on the validation set: 0.9123
Cross-validation scores: [0.95652174 0.89705882 0.85294118 0.89705882 0.9117
6471]
Mean accuracy: 0.9031
Std accuracy: 0.0332
```

In [85]:

```
# Visualize the decision tree
from sklearn.tree import export_graphviz

clf = dt_model.named_steps['clf']
export_graphviz(clf, out_file="tree.dot",
                feature_names=data.feature_names,
                class_names=data.target_names,
                rounded=True, proportion=False,
                precision=2, filled=True)
# To convert the .dot file to a .png file, use the command:
# dot -Tpng tree.dot -o tree.png

# Load image
import matplotlib.image as mpimg
img = mpimg.imread('tree.png')
plt.figure(figsize=(20, 20))
plt.imshow(img)
plt.axis('off')  # Turn off axis numbers and ticks
plt.show()
```



2.2 From the trained model, comment on feature importance values and identify the top 3 features from your model.

In [86]:
```
# Feature importance
import numpy as np
importances = clf.feature_importances_
# top 3 features
indices = np.argsort(importances)[::-1][:3]
# Print feature ranking
print("Feature ranking:")
for rank, idx in enumerate(indices):
    print(f"{rank + 1}. Feature: {data.feature_names[idx]}, Importance: {imp
```

```
Feature ranking:
1. Feature: mean concave points, Importance: 0.7064
2. Feature: worst concavity, Importance: 0.0846
3. Feature: worst perimeter, Importance: 0.0390
```

The features: mean concave points (~70%), worst concavity (~8%) and worst perimeter (~4%)are the most important features. The features with high importance dominates the model decision making. A higher importance means a stronger predictor.

2.3 Vary the max depth parameter (e.g., depth 2-10). Use validation accuracy (mean ± std from cross-validation on the training set) to choose the best depth. Provide performance for each chosen depth (at least 5 to be reported) and discuss the aspects of overfitting vs. underfitting.

In [87]:
```
from sklearn.tree import DecisionTreeClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.model_selection import KFold, cross_val_score

max_depths = range(2, 11)
results = []
# Evaluate each max_depth using cross-validation
for max_depth in max_depths:
    dt_model = Pipeline([
        ('scaler', StandardScaler()),
        ('clf', DecisionTreeClassifier(max_depth=max_depth, random_state=42)
    ])
    # 5-fold cross-validation on the training set
    kf = KFold(n_splits=5, shuffle=True, random_state=42)
    cv_scores_dt_model = cross_val_score(dt_model, X_train, y_train, cv=kf,
    results.append((max_depth, cv_scores_dt_model.mean(), cv_scores_dt_model

# Print results
print("Max Depth  | Mean Accuracy| Std Dev")
for max_depth, mean_acc, std_acc in results:
    print(f"{max_depth:9}  | {mean_acc:.4f}       | {std_acc:.4f}")
# Choose the best max_depth based on mean accuracy
best_depth = max(results, key=lambda x: x[1])[0]
print(f"\nBest max_depth based on mean accuracy: {best_depth}")
```

```
Max Depth  | Mean Accuracy| Std Dev
        2  | 0.9150       | 0.0058
        3  | 0.9208       | 0.0074
        4  | 0.9119       | 0.0164
        5  | 0.9149       | 0.0147
        6  | 0.9060       | 0.0290
        7  | 0.9031       | 0.0332
        8  | 0.9031       | 0.0332
        9  | 0.9031       | 0.0332
       10  | 0.9031       | 0.0332

Best max_depth based on mean accuracy: 3
```

Overfitting (the model is too complex)

After some depth, we can observe that the models accuracy starts dropping and that the standard deviation increases.

Underfitting (the model is too simple)

In general underfitting happens when the model is too simple and cannot capture all the patterns in the data. On the second level the model slightly underfits, since it improves on level deeper.

2.4 Repeat previous exercise with different min_samples_leaf values. Which setting generalizes best according to the validation set?

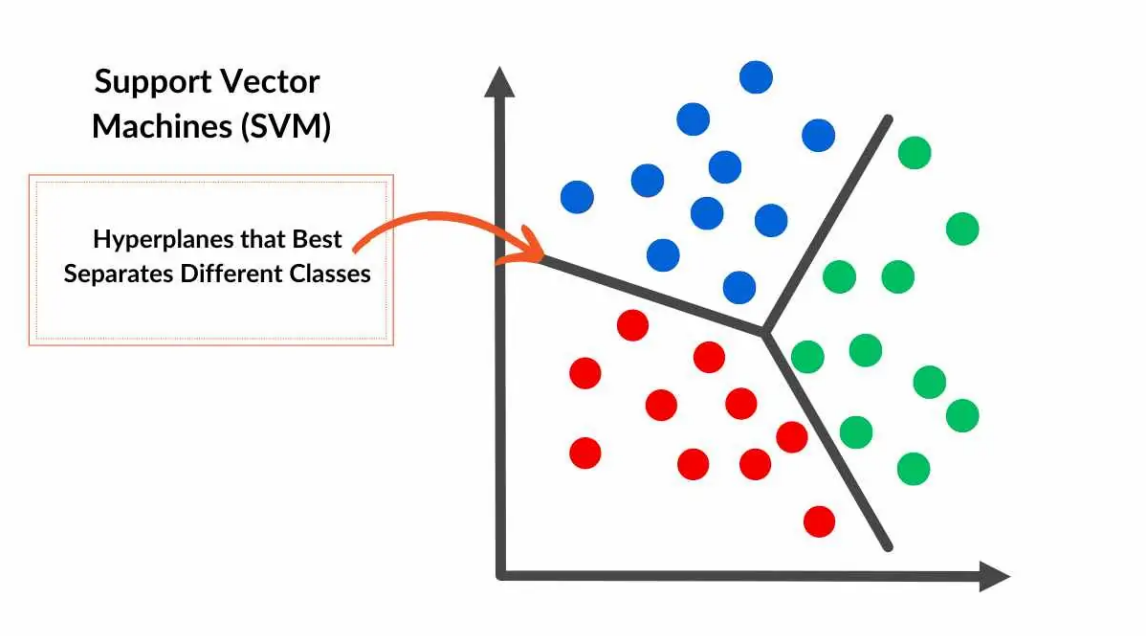
In [88]:
```
from sklearn.tree import DecisionTreeClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline

# Helper function to create a decision tree model with scaling
def make_tree(min_samples_leaf):
    return Pipeline([
        ('scaler', StandardScaler()),
        ('clf', DecisionTreeClassifier(random_state=42, min_samples_leaf=min
    ])

# Repeat previous exercise with different min_samples_leaf values. Which set
min_samples_leafs = [1, 2, 4, 6, 8, 10]
results = []
# Evaluate each min_samples_leaf using cross-validation
for min_samples_leaf in min_samples_leafs:
    dt_model = make_tree(min_samples_leaf)
    # 5-fold cross-validation on the training set
    kf = KFold(n_splits=5, shuffle=True, random_state=42)
    cv_scores_dt_model = cross_val_score(dt_model, X_train, y_train, cv=kf,
    results.append((min_samples_leaf, cv_scores_dt_model.mean(), cv_scores_d

# Print results
print("Min Samples Leaf | Mean Accuracy | Std Dev")
for min_samples_leaf, mean_acc, std_acc in results:
    print(f"{min_samples_leaf:16} | {mean_acc:.4f}       | {std_acc:.4f}")
# Choose the best min_samples_leaf based on mean accuracy
best_leaf = max(results, key=lambda x: x[1])[0]
print(f"\nBest min_samples_leaf based on mean accuracy: {best_leaf}")
```

```
Min Samples Leaf | Mean Accuracy | Std Dev
               1 | 0.9031       | 0.0332
               2 | 0.9149       | 0.0147
               4 | 0.8884       | 0.0381
               6 | 0.9061       | 0.0200
               8 | 0.9091       | 0.0060
              10 | 0.9033       | 0.0112

Best min_samples_leaf based on mean accuracy: 2
```

Exercise-3 Support Vector Machines (SVM)

3.1 Train a linear SVM (kernel="linear") and evaluate on validation sets (use the original 5 splits that was created in previous exercise). Report accuracy mean, std and plot the ROC for each split

A SVM is a supervised machine learning algorithm used for both classification and regression tasks. It has a goal of finding the optimal hyperplane (a line in 2D, or a plane in higher dimensions) that best separates data points into different classes. It identifys "support vectors," which are the closest data points to the hyperplane.



In [89]:
```
from sklearn import svm

# Train a linear SVM
linear_svm = svm.SVC(kernel="linear")
linear_svm.fit(X_train, y_train)

# Evaluate on validation set
y_val_pred = linear_svm.predict(X_val)
val_accuracy = accuracy_score(y_val, y_val_pred)
print(f"Validation Accuracy with linear SVM: {val_accuracy:.4f}")
# Accuracy means and standard deviation across folds
kf = KFold(n_splits=5, shuffle=True, random_state=42)
cv_scores_linear_svm = cross_val_score(linear_svm, X_train, y_train, cv=kf,
print("Cross-validation scores with linear SVM (without scaling):", cv_score
print(f"Mean accuracy: {cv_scores_linear_svm.mean():.4f}")
print(f"Standard deviation: {cv_scores_linear_svm.std():.4f}")
```

```
Validation Accuracy with linear SVM: 0.9386
Cross-validation scores with linear SVM (without scaling): [0.95652174 0.955
88235 0.91176471 0.94117647 0.98529412]
Mean accuracy: 0.9501
Standard deviation: 0.0239
```

In [90]:
```python
from sklearn import svm

# Build a pipeline with scaling and SVM classifier
linear_svm = Pipeline([
    ('scaler', StandardScaler()),
    ('clf', svm.SVC(kernel='linear',probability=True, random_state=42)),
])
# Fit on training and evaluate on validation set
linear_svm.fit(X_train, y_train)
y_val_pred = linear_svm.predict(X_val)
val_accuracy_linear = accuracy_score(y_val, y_val_pred)

print(f"Validation Accuracy with linear SVM: {val_accuracy_linear:.4f}")
# Accuracy means and standard deviation across folds
kf = KFold(n_splits=5, shuffle=True, random_state=42)
cv_scores_linear_svm = cross_val_score(linear_svm, X_train, y_train, cv=kf,
print("Cross-validation scores with linear SVM (with scaling):", cv_scores_l
print(f"Mean accuracy: {cv_scores_linear_svm.mean():.4f}")
print(f"Standard deviation: {cv_scores_linear_svm.std():.4f}")
```

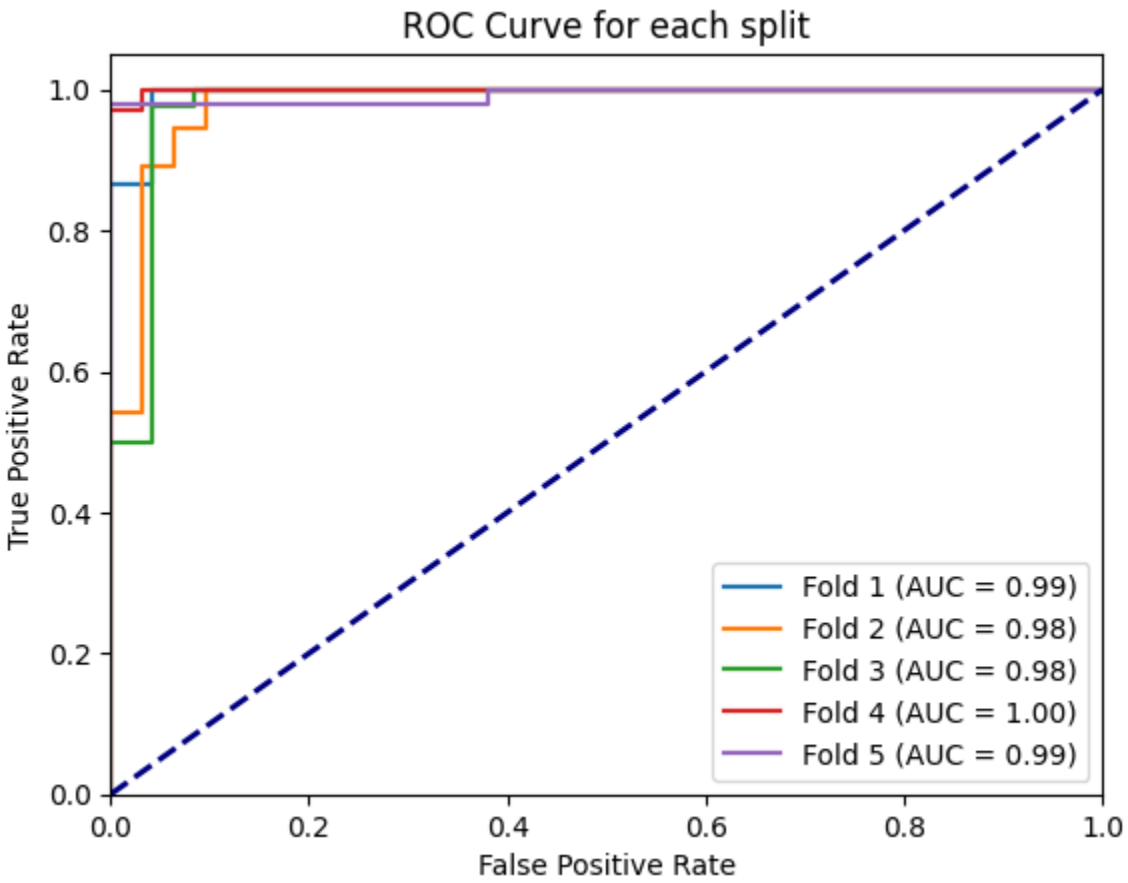
```
Validation Accuracy with linear SVM: 0.9561
Cross-validation scores with linear SVM (with scaling): [0.95652174 0.941176
47 0.95588235 0.98529412 0.97058824]
Mean accuracy: 0.9619
Standard deviation: 0.0149
```

ROC curve (receiver operating characteristic)

- The ROC curve is the plot of the true positive rate (TPR) against the false positive rate (FPR) at each threshold setting.
- A classifier with the random performance level always shows a straight line from the origin (0.0, 0.0) to the top right corner (1.0, 1.0), this is the random classifier line. Everything above is considered "good" and under is poor"

In [91]:
```python
from sklearn.metrics import roc_curve, auc
import matplotlib.pyplot as plt

svm = linear_svm
# Prepare to collect accuracy and AUC (area under the curve) for each fold
accs, aucs = [], []

# function to set up ROC plot
def plot_roc_curve(title="ROC Curve"):
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title(title)
    plt.legend(loc="lower right")
```

```
    # random classifier line
    plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')

plt.figure()

for i, (tr_idx, va_idx) in enumerate(kf.split(X_train, y_train), start=1):
    X_tr, X_va = X_train[tr_idx], X_train[va_idx]
    y_tr, y_va = y_train[tr_idx], y_train[va_idx]

    svm.fit(X_tr, y_tr)
    y_pred = svm.predict(X_va)
    y_prob = svm.predict_proba(X_va)[:, 1]

    # accuracy and AUC per split
    accs.append(accuracy_score(y_va, y_pred))
    linear_fpr, linear_tpr, _ = roc_curve(y_va, y_prob)
    roc_auc = auc(linear_fpr, linear_tpr)
    aucs.append(roc_auc)

    # plot ROC for this split
    plt.plot(linear_fpr, linear_tpr, lw=1.6, label=f'Fold {i} (AUC = {roc_au

# Plot ROC curve for each split
plot_roc_curve(title="ROC Curve for each split")

plt.show()
```



3.2 Train an Radial Basis Function (RBF) kernel SVM.

Compare its performance to the linear kernel using validation accuracy. Plot ROC for these models.

It is used to transform the input data into a higher-dimensional space to find a hyperplane that separates the data points.

The RBF kernel function is defined as:

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

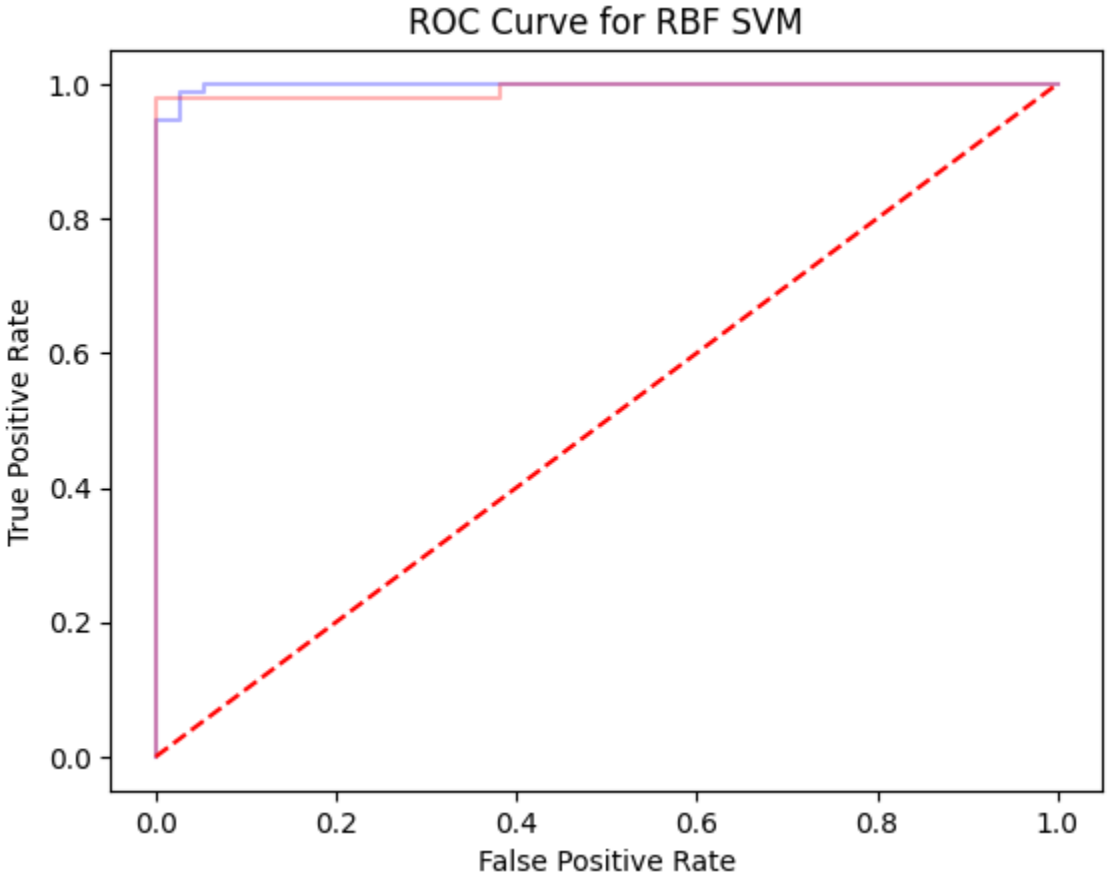
Where x and x_i are input vectors, $\|x - x_i\|^2$ is the squared Euclidean distance between the two vectors, and gamma is the kernel parameter.

The RBF kernel function has several advantages. It is a universal kernel, meaning that it can approximate any continuous function to arbitrary precision given sufficient data. It is also computationally efficient because it can be implemented using only inner products, and is relatively simple to use because it only requires the choice of a single parameter, gamma.

In [92]:
```
# Train an Radial Basis Function (RBF) kernel SVM. Compare its performance t
from sklearn import svm
import matplotlib.pyplot as plt


rbf_model = Pipeline([
    ('scaler', StandardScaler()),
    ('clf', svm.SVC(kernel='rbf', probability=True, random_state=42)),
])

# Fit on training and evaluate on validation set
rbf_model.fit(X_train, y_train)
y_val_pred = rbf_model.predict(X_val)
val_accuracy_rbf = accuracy_score(y_val, y_val_pred)

# Accuracy means and standard deviation across folds
kf = KFold(n_splits=5, shuffle=True, random_state=42)
cv_scores_rbf_svm = cross_val_score(rbf_model, X_train, y_train, cv=kf, scor

print(f"Validation Accuracy with RBF SVM (uses standard scaler): {val_accura
# Linear SVM for comparison
print(f"Validation Accuracy with linear SVM (uses standard scaler): {val_acc

# Plot ROC curve for RBF SVM
y_val_prob = rbf_model.predict_proba(X_val)[:, 1]
plt.figure()
fpr, tpr, _ = roc_curve(y_val, y_val_prob)

# Blue for RBF SVM, red for linear SVM
plt.plot(fpr, tpr, color='blue', alpha=0.3)
plt.plot(linear_fpr, linear_tpr, color='red', alpha=0.3)

# Random classifier line
```

```
plt.plot([0, 1], [0, 1], color='red', linestyle='--')
plt.title("ROC Curve for RBF SVM")
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.show()
```

```
Validation Accuracy with RBF SVM (uses standard scaler): 0.9737
Validation Accuracy with linear SVM (uses standard scaler): 0.9561
```



3.3 Experiment with different values of C (regularization strength). Use the validation set to select the best C. Report results as a plot of accuracy vs. C.

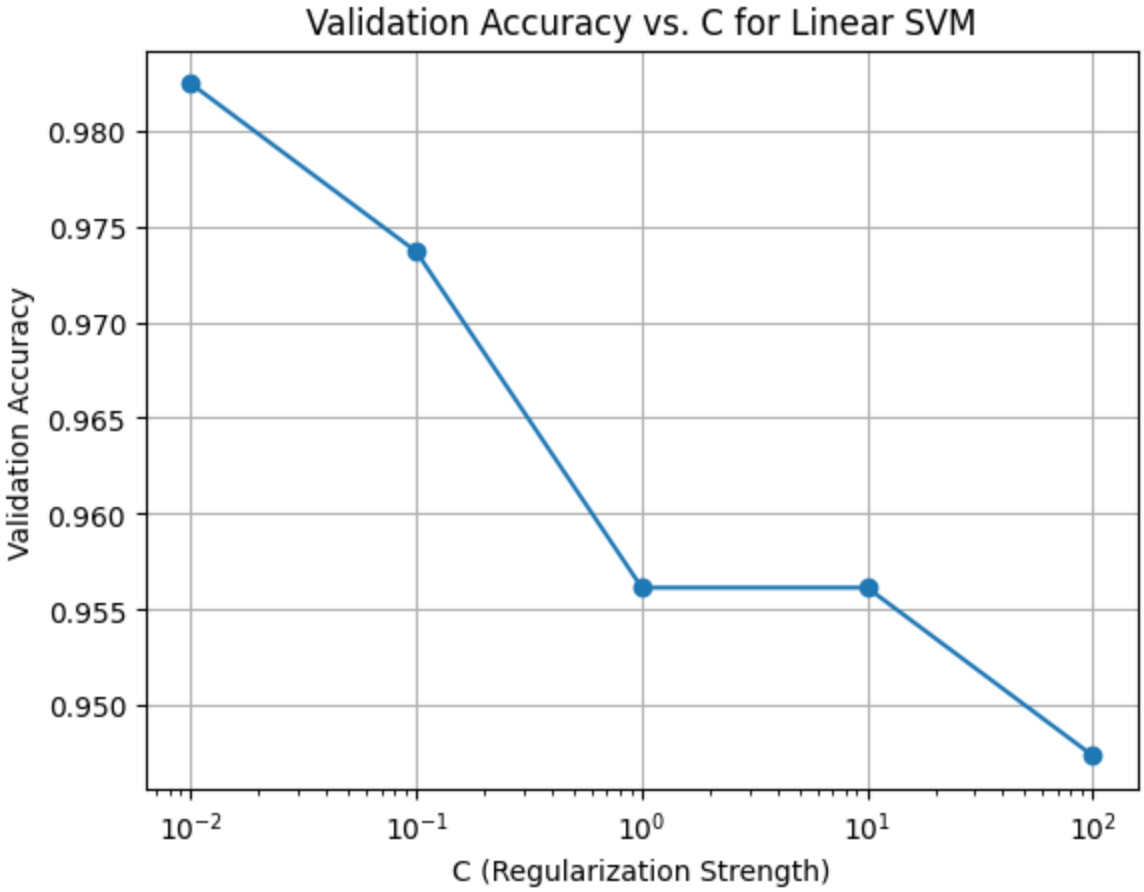
A regularization rate (lambda) controls the strength of regularization, with higher values leading to simpler models and lower values increasing the risk of overfitting.

- Higher values = simpler models (underfitting)
- Lower values = complex models (overfitting)

In [93]:
```
# Regularization strength C values to try
C_values = [0.01, 0.1, 1, 10, 100]
results = []
# Evaluate each C using cross-validation
for C in C_values:
    linear_svm = Pipeline([
```

```
        ('scaler', StandardScaler()),
        ('clf', svm.SVC(kernel='linear', C=C, probability=True, random_state
    ])
    # Fit the model
    linear_svm.fit(X_train, y_train)
    # Evaluate on validation set
    y_val_pred = linear_svm.predict(X_val)
    val_accuracy = accuracy_score(y_val, y_val_pred)
    results.append((C, val_accuracy))
    print(f"Validation Accuracy with linear SVM (C={C}): {val_accuracy:.4f}"

# sort results by accuracy
results.sort(key=lambda x: x[1], reverse=True)
# Print results
print("\nC Value | Validation Accuracy")
for C, val_acc in results:
    print(f"{C:7} | {val_acc:.4f}")

# Plot accuracy vs. C
C_vals, accuracies = zip(*results)
plt.figure()
plt.semilogx(C_vals, accuracies, marker='o')
plt.xlabel("C (Regularization Strength)")
plt.ylabel("Validation Accuracy")
plt.title("Validation Accuracy vs. C for Linear SVM")
plt.grid(True)
plt.show()
```

```
Validation Accuracy with linear SVM (C=0.01): 0.9825
Validation Accuracy with linear SVM (C=0.1): 0.9737
Validation Accuracy with linear SVM (C=1): 0.9561
Validation Accuracy with linear SVM (C=10): 0.9561
Validation Accuracy with linear SVM (C=100): 0.9474

C Value | Validation Accuracy
   0.01 | 0.9825
    0.1 | 0.9737
      1 | 0.9561
     10 | 0.9561
    100 | 0.9474
```

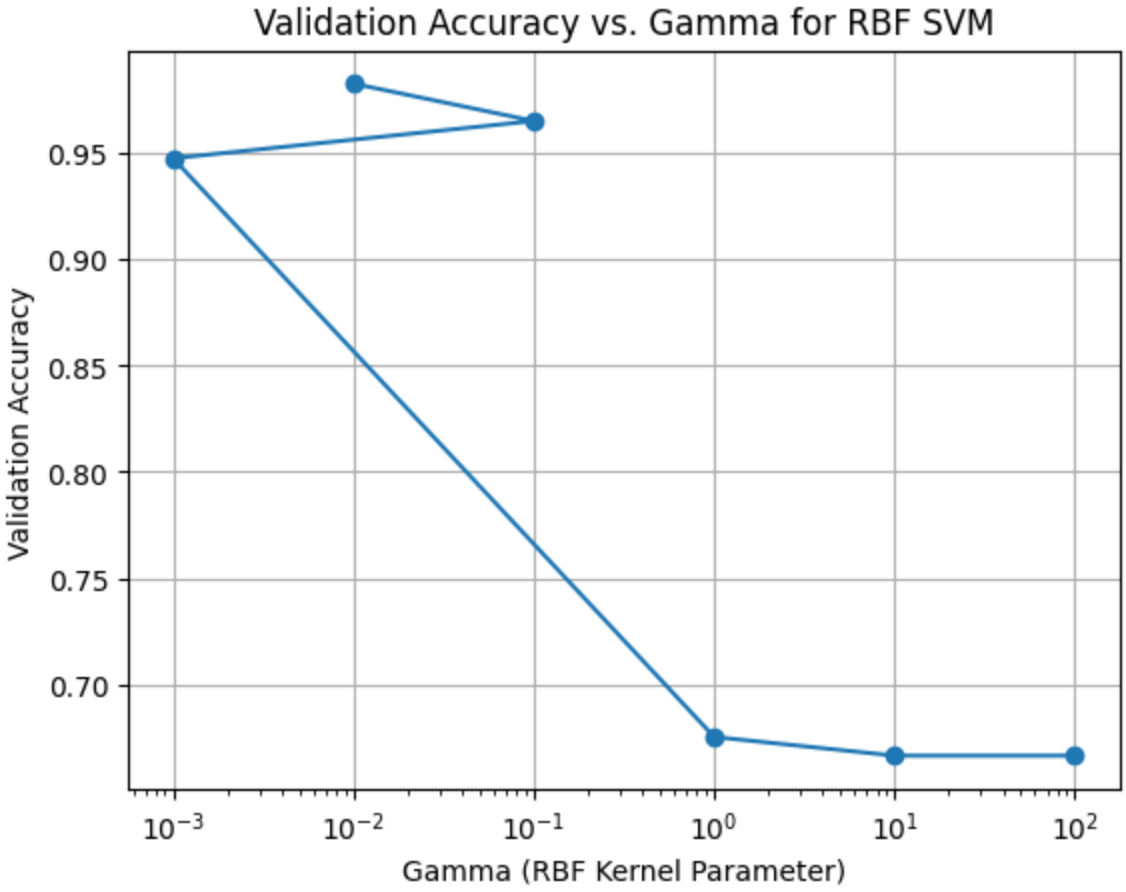


3.4 Experiment with different γ values for the RBF kernel. Discuss the effect on bias-variance trade-off for all experimented values. Select the best γ using the validation set and report the performance on validation set.

In [94]:
```
# Testing different gamma values for RBF kernel SVM
gamma_values = [0.001, 0.01, 0.1, 1, 10, 100]
results = []
# Evaluate each gamma using cross-validation
for gamma in gamma_values:
    rbf_svm = Pipeline([
        ('scaler', StandardScaler()),
        ('clf', svm.SVC(kernel='rbf', gamma=gamma, probability=True, random_
    ])
    # Fit the model
    rbf_svm.fit(X_train, y_train)
    # Evaluate on validation set
    y_val_pred = rbf_svm.predict(X_val)
    val_accuracy_rbf = accuracy_score(y_val, y_val_pred)
    results.append((gamma, val_accuracy_rbf))
    print(f"Validation Accuracy with RBF SVM (gamma={gamma}): {val_accuracy_

# sort results by accuracy
```

```
results.sort(key=lambda x: x[1], reverse=True)
# Print results
print("\nGamma Value | Validation Accuracy")
for gamma, val_acc in results:
    print(f"{gamma:11} | {val_acc:.4f}")

# Plot accuracy vs. gamma
gamma_vals, accuracies = zip(*results)
plt.figure()
plt.semilogx(gamma_vals, accuracies, marker='o')
plt.xlabel("Gamma (RBF Kernel Parameter)")
plt.ylabel("Validation Accuracy")
plt.title("Validation Accuracy vs. Gamma for RBF SVM")
plt.grid(True)
plt.show()
```

```
Validation Accuracy with RBF SVM (gamma=0.001): 0.9474
Validation Accuracy with RBF SVM (gamma=0.01): 0.9825
Validation Accuracy with RBF SVM (gamma=0.1): 0.9649
Validation Accuracy with RBF SVM (gamma=1): 0.6754
Validation Accuracy with RBF SVM (gamma=10): 0.6667
Validation Accuracy with RBF SVM (gamma=100): 0.6667

Gamma Value | Validation Accuracy
       0.01 | 0.9825
        0.1 | 0.9649
      0.001 | 0.9474
          1 | 0.6754
         10 | 0.6667
        100 | 0.6667
```



Exercise-4: Model Comparison

4.1 Compare Decision Tree and SVM results from the training set (cross-validation mean ± std) and validation set. Plot the performance comparison plots (e.g., scatter plots, ROC curves). Which model generalizes better?

In [100…

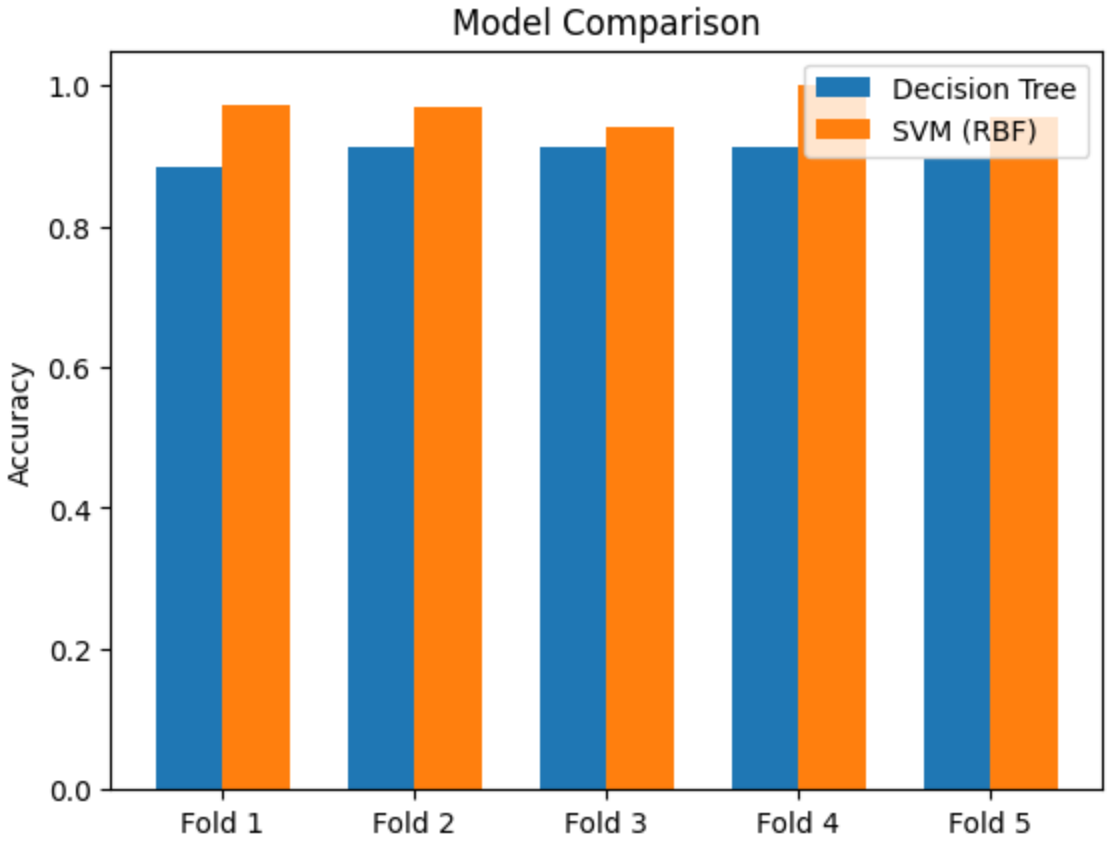
```
# Decision Tree vs. SVM comparison
dt_model
svm_model = rbf_model

# Fit on training and evaluate on validation set
dt_model.fit(X_train, y_train)
svm_model.fit(X_train, y_train)
# Evaluate on validation set
# Decision Tree
y_val_pred_dt = dt_model.predict(X_val)
val_accuracy_dt = accuracy_score(y_val, y_val_pred_dt)
# SVM
y_val_pred_svm = svm_model.predict(X_val)
val_accuracy_svm = accuracy_score(y_val, y_val_pred_svm)
# Print validation accuracies
```

```
print(f"Validation Accuracy with Decision Tree: {val_accuracy_dt:.4f}")
# SVM (from previous exercise)
print(f"Validation Accuracy with SVM: {val_accuracy_svm:.4f}")

# Print results
print("\nDecision Tree Results:")
print(f"Validation Accuracy: {val_accuracy_dt:.4f}")
print("Cross-validation scores:", cv_scores_dt_model)
print(f"Mean accuracy: {cv_scores_dt_model.mean():.4f}")
print(f"Std accuracy: {cv_scores_dt_model.std():.4f}")

print("\nSupport Vector Machine Results:")
print(f"Validation Accuracy: {val_accuracy_svm:.4f}")
print("Cross-validation scores:", cv_scores_rbf_svm)
print(f"Mean accuracy: {cv_scores_rbf_svm.mean():.4f}")
print(f"Std accuracy: {cv_scores_rbf_svm.std():.4f}")
```

```
Validation Accuracy with Decision Tree: 0.9474
Validation Accuracy with SVM: 0.9737

Decision Tree Results:
Validation Accuracy: 0.9474
Cross-validation scores: [0.88405797 0.91176471 0.91176471 0.91176471 0.8970
5882]
Mean accuracy: 0.9033
Std accuracy: 0.0112

Support Vector Machine Results:
Validation Accuracy: 0.9737
Cross-validation scores: [0.97101449 0.97058824 0.94117647 1.         0.9558
8235]
Mean accuracy: 0.9677
Std accuracy: 0.0195
```

In [99]:

```
# Performance comparison plots
import matplotlib.pyplot as plt

# Plotting
labels = ['Fold 1', 'Fold 2', 'Fold 3', 'Fold 4', 'Fold 5']
x = range(len(labels)) # the label locations
width = 0.35  # the width of the bars

fig, ax = plt.subplots()
bars1 = ax.bar(x, cv_scores_dt_model, width, label='Decision Tree')
bars2 = ax.bar([p + width for p in x], cv_scores_rbf_svm, width, label='SVM

# Add some text for labels, title and custom x-axis tick labels, etc.
ax.set_ylabel('Accuracy')
ax.set_title('Model Comparison')
ax.set_xticks([p + width / 2 for p in x])
ax.set_xticklabels(labels)
ax.legend()

plt.show()
```



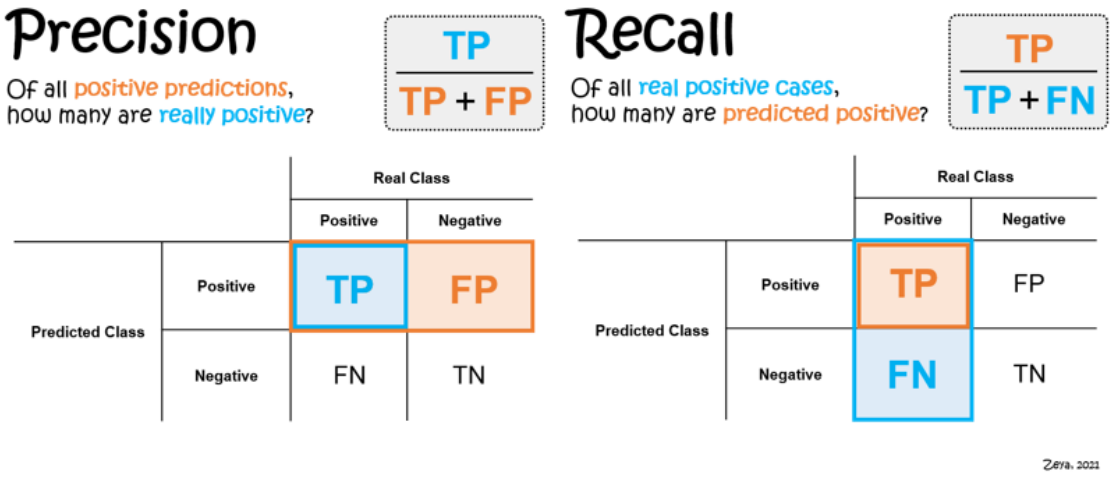
The SVM Model generalized better

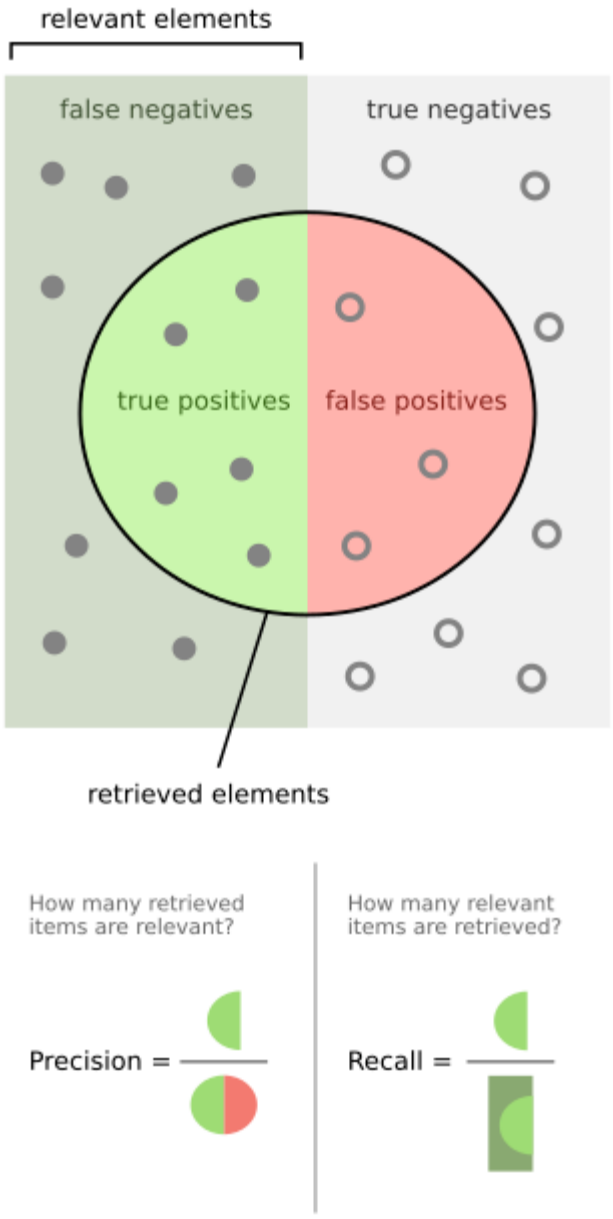
4.2 Discuss the trade-off between usability and accuracy for this dataset. Which model would you recommend for a medical decision-support system, and why? (Hint - Make use of F_β score analysis)

$$F_\beta = (1 + \beta^2) \cdot \frac{(\text{precision} \cdot \text{recall})}{(\beta^2 \cdot \text{precision}) + \text{recall}}$$

In [97]:
```
from sklearn.metrics import fbeta_score
# Calculate F2 score for both models on validation set

def calculate_f2_score(y_validation,beta):
    f2_dt = fbeta_score(y_validation, y_val_pred_dt, beta=beta)
    f2_svm = fbeta_score(y_validation, y_val_pred_svm, beta=beta)
    print(f"F2 Score with Decision Tree: {f2_dt:.4f}")
    print(f"F2 Score with SVM: {f2_svm:.4f}")

# Emphasize recall (beta > 1)
weighted_beta = 2 # A weight of 2 emphasizes recall twice as much as precisi
calculate_f2_score(y_val, weighted_beta)
```

```
F2 Score with Decision Tree: 0.9605
F2 Score with SVM: 0.9763
```





Explanation

When developing a medical decision-support system based on the **Breast Cancer Wisconsin Diagnostic** dataset, there is a distinct trade-off between **usability** and **accuracy**. In this context, *accuracy* refers to how well the model correctly classifies malignant and benign tumors, while *usability* reflects how practical the system is for clinicians — particularly in terms of how often it raises false alarms.

It is more important for the system to detect as many **true malignant cases** as possible, even if this reduces precision and increases the number of false positives. A lower precision may affect usability, as doctors might need to review more non-cancerous cases, but the consequences of missing a malignant tumor are far more serious.

Therefore, the recommended approach is to select the model with the **highest** F_β score using **β > 1**, which emphasizes **recall** over precision. This ensures the model identifies as many malignant cases as possible — an essential property for a reliable medical decision-support system.

I would suggest using the suppert vector machine model.

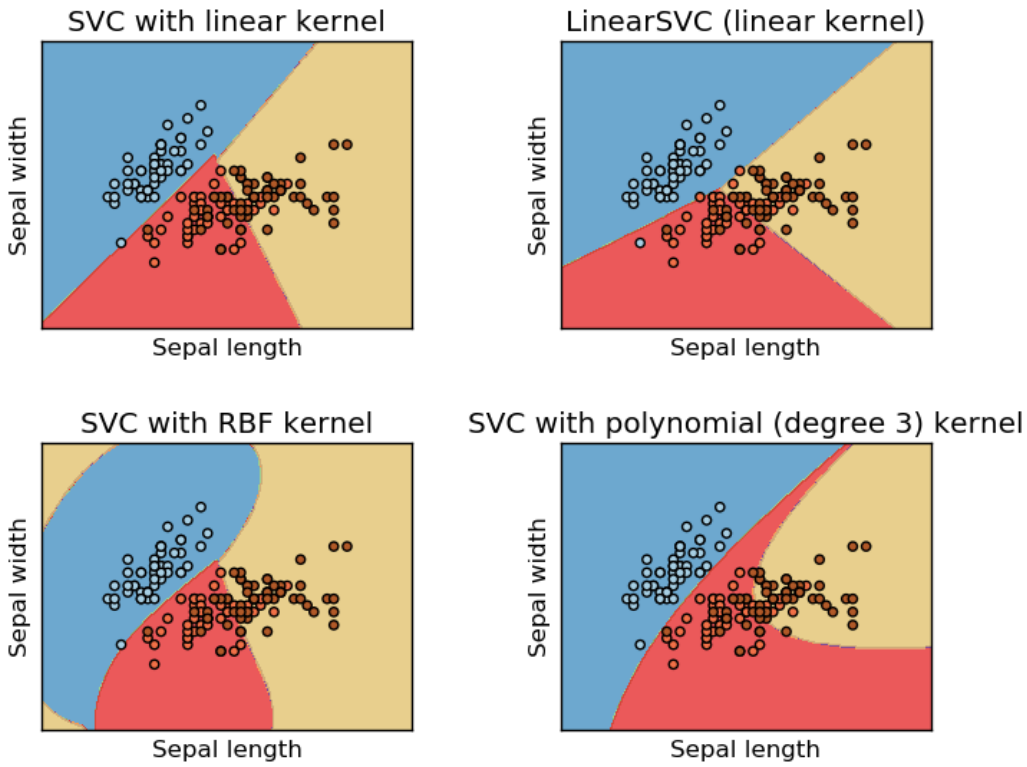
4.4 Compare the final test set accuracy of the best Decision Tree and best SVM. Which model performs better in practice?

In [98]:
```
best_dt = dt_model
best_svm = rbf_model

# Test the models on the test set
y_test_pred_dt = best_dt.predict(X_test)
y_test_pred_svm = best_svm.predict(X_test)
# Evaluate test accuracy
test_accuracy_dt = accuracy_score(y_test, y_test_pred_dt)
test_accuracy_svm = accuracy_score(y_test, y_test_pred_svm)
# Print test accuracies
print(f"Test Accuracy with Decision Tree: {test_accuracy_dt:.4f}")
print(f"Test Accuracy with SVM: {test_accuracy_svm:.4f}")
```

```
Test Accuracy with Decision Tree: 0.9561
Test Accuracy with SVM: 0.9737
```

In practice the support vector model using the RBF kernel performs better in practice

4.5 The Breast Cancer Wisconsin dataset has 30 continuous features, many of which are correlated and not linearly separable. Explain why a linear SVM might fail to capture complex patterns in this dataset. How does using an RBF kernel help in this case? Discuss your answer in terms of the dataset's feature space and the geometry of the decision boundary.



The Breast Cancer Wisconsin dataset contains 30 continuous, often correlated features that are not linearly separable. A linear SVM creates a flat hyperplane as its decision boundary, which cannot capture the complex relationships between these features. In contrast, an SVM with an RBF kernel maps the data into a higher-dimensional feature space, where nonlinear patterns can become linearly separable. The RBF kernel allows the model to capture the complex geometry of the dataset more effectively than a linear SVM.

- The linear SVM produces a flat hyperplane (simple geometry).
- The RBF kernel SVM produces a flexible, curved boundary that can adapt to the data's structure.