# Problem 1:

1. The log-likelihood function for logistic regression is given by:

$$l(\beta) = \sum_{i=1}^{n} \left[ y_i \log(p_i) + (1 - y_i) \log(1 - p_i) \right]$$

where $p_i = \frac{1}{1 + \exp(-\beta^T X_i)}$ .

To show concavity, we need to calculate the second derivative of the log-likelihood function. For logistic regression, the Hessian matrix (second derivative) of the log-likelihood is negative semi-definite, which implies concavity. Specifically, the Hessian is:

$$H = -X^T W X$$

where $W$ is a diagonal matrix with entries $p_i (1 - p_i)$ , and since $p_i(1 - p_i) \geq 0$ , the Hessian is negative semi-definite, confirming concavity.

2. Quasi-Newton Method. This method is a generalization of the Newton algorithm but avoids calculating the Hessian matrix directly, which can be computationally expensive, especially for large datasets. Instead, it builds an approximation to the Hessian matrix using gradient information over iterations.

Update rule: The algorithm iteratively updates the parameters using:

$$\theta_{k+1} = \theta_k - H_k^{-1} \nabla L(\theta_k)$$

where $\nabla L(\theta_k)$ is the gradient of the likelihood function, and $H_k^{-1}$ is an approximation of the inverse Hessian matrix.

Pros of Quasi-Newton Method:

1. No Hessian calculation: Unlike the Newton algorithm, Quasi-Newton methods do not require the explicit computation of the Hessian matrix, which is computationally cheaper for large datasets.
2. Faster convergence: It often converges faster than gradient descent since it uses second-order information (the Hessian approximation), leading to more informed parameter updates.
3. Stable updates: Compared to Newton's method, where poor initial estimates can lead to large, unstable steps, Quasi-Newton methods often provide more stable steps.

Cons of Quasi-Newton Method:

1. More memory usAge: It requires storing the Hessian approximation, which takes more memory compared to simple gradient descent.
2. Still more expensive than gradient descent: While it avoids direct Hessian calculation, Quasi-Newton methods

still involve higher computational complexity per iteration compared to first-order methods like gradient descent.
3. Less accurate than Newton's method: Since the Hessian is only approximated, it may not converge as precisely as Newton's method, especially in cases where the true Hessian has special properties (e.g., when it's close to singular).

3. The coefficient $\beta_i$ in a logistic regression model represents the log-odds change associated with a one-unit increase in the predictor variable $X_i$ . Specifically:

$$log\left(\frac{P(Y=1|X)}{P(Y=0|X)}\right) = \beta_0 + \beta_i X_i$$

4. Pseudocode:

```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import roc_curve, roc_auc_score
import matplotlib.pyplot as plt

# Train logistic regression model
model = LogisticRegression()
model.fit(X_train, y_train)

# Predict probabilities
y_prob = model.predict_proba(X_train)[:, 1]

# Compute ROC curve
fpr, tpr, thresholds = roc_curve(y_train, y_prob)

# Plot ROC curve
plt.plot(fpr, tpr, label="Logistic Regression ROC")
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Curve")
plt.show()
```

5. • Logistic Regression is a discriminative model that directly models the conditional probability $P(y|X)$ without assuming any distribution for the predictors.

• Linear Discriminant Analysis (LDA) is a generative model that assumes the predictors $X$ are normally distributed within each class. LDA estimates the parameters of the distribution and then uses Bayes' theorem to compute the posterior probabilities $P(y|X)$.

The key difference lies in the underlying assumptions about the data distribution: logistic regression makes fewer assumptions, whereas LDA assumes normally distributed features within each class.

6. In Linear Discriminant Analysis (LDA), we assume that the feature vector x follows a Gaussian distribution $N(\mu_k,\Sigma)$ for each class k, where $\mu_k$ is the class mean and $\Sigma$ is the shared covariance matrix. Using Bayes' Theorem, the posterior probability can be written as:

$$P(y=k|x)=\frac{P(x|y=k)P(y=k)}{P(x)}$$

Since $P(x)$ is the same across all classes, we maximize the log-posterior:

$$\delta_k(x)=log P(x|y=k)+log P(y=k)$$

Given that $P(x|y=k)$ is a Gaussian distribution, we have:

$$\delta_k(x) = \mu_k^T \Sigma^{-1} x - \frac{1}{2} \mu_k^T \Sigma^{-1} \mu_k + \log(\pi_k)$$

7. The pooled sample covariance matrix in LDA is calculated by combining the within-class sample covariance matrices:

$$\hat{\Sigma} = \frac{1}{n - K} \sum_{k=1}^{K} \sum_{i:y_i=k} (x_i - \mu_k)(x_i - \mu_k)^T$$

The sample covariance matrix for each class k is:

$$\hat{\Sigma}_k = \frac{1}{n_k - 1} \sum_{i:y_i = k} (x_i - \mu_k)(x_i - \mu_k)^T$$

where $n_k$ is the number of data points in class k .

Since the classes are assumed to have the same covariance matrix $\Sigma$ , the pooled sample covariance matrix $\hat{\Sigma}$ combines the individual class sample covariance matrices and adjusts for the total number of samples:

$$\hat{\Sigma} = \frac{1}{n - K} \sum_{k=1}^{K} (n_k - 1) \hat{\Sigma}_k$$

Since the individual class sample covariance matrices $\hat{\Sigma}_k$ are unbiased estimators of $\Sigma$ , we have:

$$E(\hat{\Sigma}_k) = \Sigma$$

Therefore, the expected value of the pooled covariance matrix is:

$$E(\hat{\Sigma}) = \frac{1}{n - K} \sum_{k=1}^{K} (n_k - 1) E(\hat{\Sigma}k) = \frac{1}{n - K} \sum_{k=1}^{K} (n_k - 1) \Sigma = \Sigma$$

Thus, the expected value of the pooled sample covariance matrix $\hat{\Sigma}$ is equal to the population covariance matrix $\Sigma$ , proving that $\hat{\Sigma}$ is an unbiased estimator.

8. Assume Class-Specific Covariance Matrices: • $ \mathbf{x} | Y = k \sim N(\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) $

Compute Log of Posterior (ignoring constants):

Express Log-Likelihood:

$$ \log P(\mathbf{x} | Y = k) = -\frac{1}{2} (\mathbf{x} - \boldsymbol{\mu}_k)^\top \boldsymbol{\Sigma}_k^{-1} (\mathbf{x} - \boldsymbol{\mu}_k) - \frac{1}{2} \log |\boldsymbol{\Sigma}_k| $$

Combine Terms: $$ \delta_k(\mathbf{x}) = -\frac{1}{2} \left[ (\mathbf{x} - \boldsymbol{\mu}_k)^\top \boldsymbol{\Sigma}_k^{-1} (\mathbf{x} - \boldsymbol{\mu}_k) + \log |\boldsymbol{\Sigma}_k| \right] + \log P(Y = k) $$

Resulting Discriminant Function: • The discriminant function is quadratic in due to the term $ (\mathbf{x} - \boldsymbol{\mu}_k)^\top \boldsymbol{\Sigma}_k^{-1} (\mathbf{x} - \boldsymbol{\mu}_k) $.

9. • Confusion Matrix: For multiclass classification, the confusion matrix is extended to a  matrix, where  is the number of classes. Each row represents the actual class, and each column represents the predicted class. The entry indicates the number of instances where the true class is  and the predicted class is . • ROC Curve: In multiclass problems, one common approach is to treat each class as the positive class and all other classes as the negative class (one-vs-rest approach), then plot a separate ROC curve for each class. An alternative is to use an averAged ROC curve across all classes.

10.

• Variance: As K increases, each training set used in the cross-validation process becomes closer to the full dataset, leading to lower variance in the evaluation metrics. When $K=n$ (i.e., leave-one-out cross-validation), the variance is minimized.

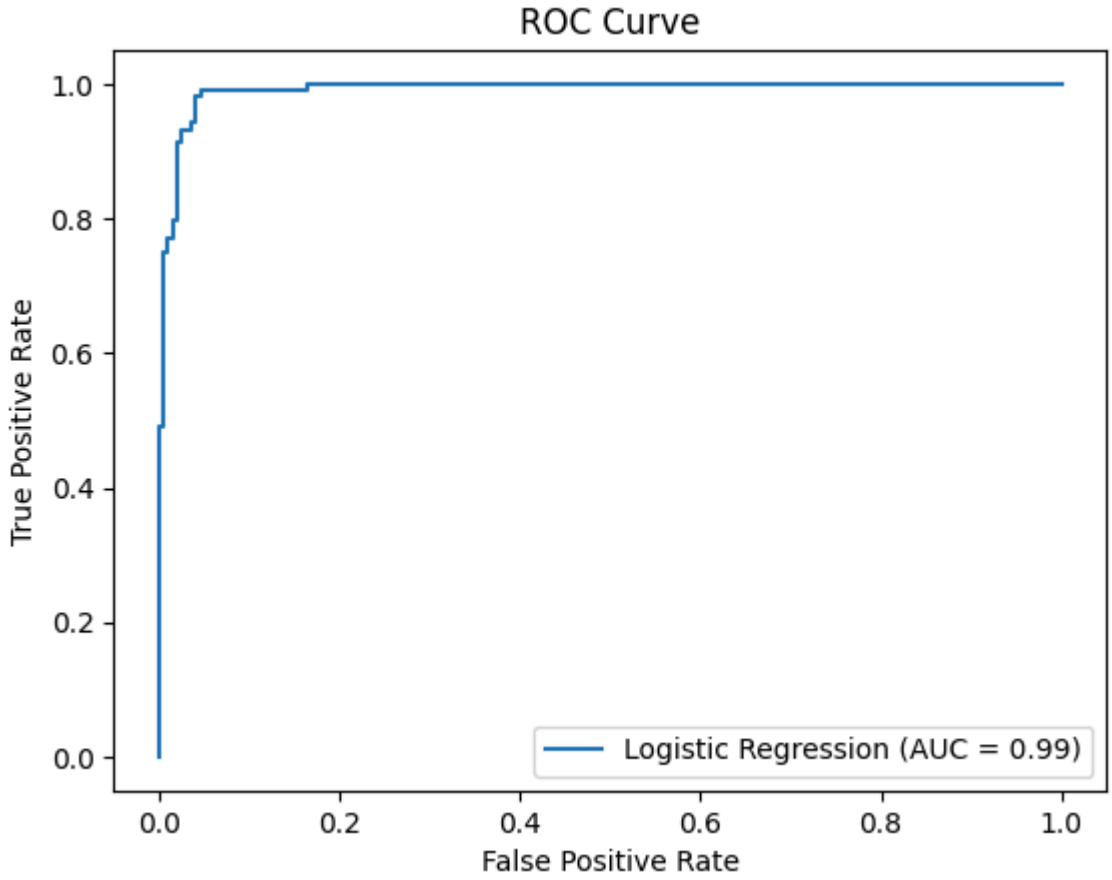
• Bias: As K increases, the training sets become larger, which reduces the bias in the model's performance estimate. However, if K is too large, it might lead to overfitting because the model will train on nearly the entire dataset in each fold.

Typically, K values like 5 or 10 are used to balance bias and variance effectively.

# Problem2

In [1]:
```
import pandas as pd
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import roc_curve, roc_auc_score
import matplotlib.pyplot as plt
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler

train_data = pd.read_csv('BreastCancer_train.csv')
test_data = pd.read_csv('BreastCancer_test.csv')

# Load the datasets
train_data['Class'] = train_data['Class'].map({'benign': 0, 'malignant':
```

```
test_data['Class'] = test_data['Class'].map({'benign': 0, 'malignant': 1}

# Separate predictors and target
X_train = train_data.drop(columns=['Id', 'Class'])
y_train = train_data['Class']
X_test = test_data.drop(columns=['Id', 'Class'])
y_test = test_data['Class']

# Handle missing values (imputation)
imputer = SimpleImputer(strategy='mean')
X_train = imputer.fit_transform(X_train)
X_test = imputer.transform(X_test)

# Standardize the features (optional)
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Logistic Regression Model
model = LogisticRegression()
model.fit(X_train, y_train)

# Predict probabilities on the test set
y_prob = model.predict_proba(X_test)[:, 1]

# Calculate ROC curve and AUC
fpr, tpr, thresholds = roc_curve(y_test, y_prob)
auc = roc_auc_score(y_test, y_prob)

# Plot ROC curve
plt.plot(fpr, tpr, label=f'Logistic Regression (AUC = {auc:.2f})')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve')
plt.legend()
plt.show()
```




In [2]:
```
train_data
test_data
```

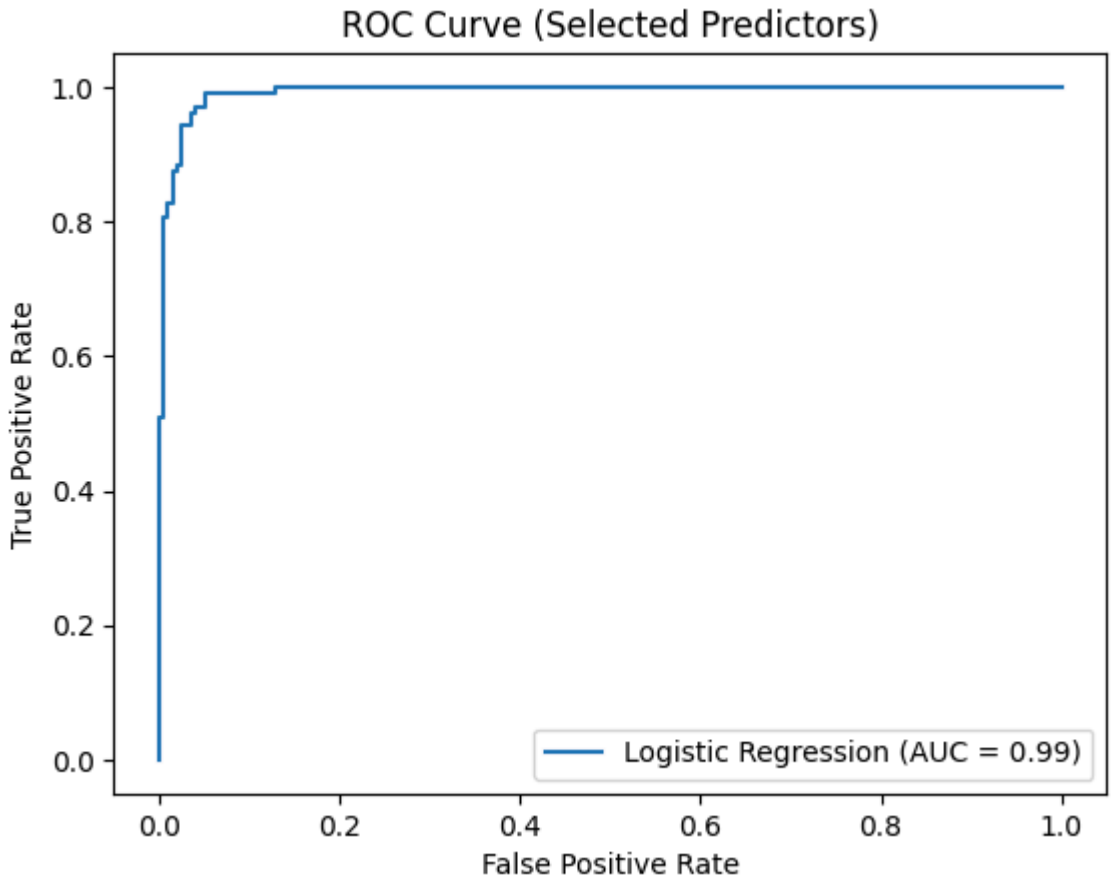
Out[2]:

| | Id | Cl.thickness | Cell.size | Cell.shape | Marg.adhesion | Epith.c.size | Bar |
|---|---|---|---|---|---|---|---|
| **0** | 1002945 | 5 | 4 | 4 | 5 | 7 | |
| **1** | 1017122 | 8 | 10 | 10 | 8 | 7 | |
| **2** | 1018099 | 1 | 1 | 1 | 1 | 2 | |
| **3** | 1033078 | 4 | 2 | 1 | 1 | 2 | |
| **4** | 1041801 | 5 | 3 | 3 | 3 | 2 | |
| **...** | ... | ... | ... | ... | ... | ... | |
| **294** | 1371026 | 5 | 10 | 10 | 10 | 4 | |
| **295** | 1371920 | 5 | 1 | 1 | 1 | 2 | |
| **296** | 466906 | 1 | 1 | 1 | 1 | 2 | |
| **297** | 466906 | 1 | 1 | 1 | 1 | 2 | |
| **298** | 536708 | 1 | 1 | 1 | 1 | 2 | |

299 rows × 11 columns

In [3]:
```
selected_features = ['Cl.thickness', 'Cell.shape', 'Marg.adhesion', 'Bare

X_train_selected = train_data[selected_features]
X_test_selected = test_data[selected_features]
```

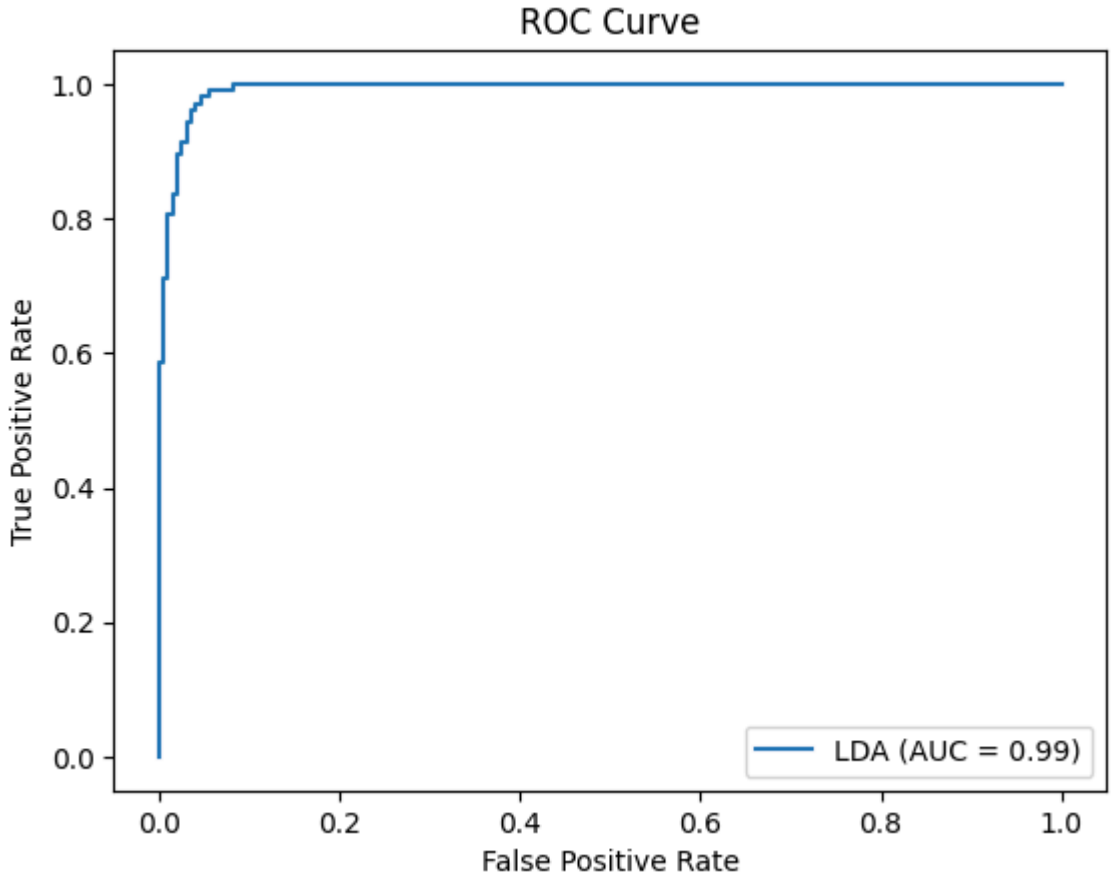
```
# Handle missing values and standardize
X_train_selected = imputer.fit_transform(X_train_selected)
X_test_selected = imputer.transform(X_test_selected)

X_train_selected = scaler.fit_transform(X_train_selected)
X_test_selected = scaler.transform(X_test_selected)

# Logistic Regression Model with selected predictors
model = LogisticRegression()
model.fit(X_train_selected, y_train)

# Predict probabilities
y_prob = model.predict_proba(X_test_selected)[:, 1]

# ROC curve and AUC
fpr, tpr, thresholds = roc_curve(y_test, y_prob)
auc = roc_auc_score(y_test, y_prob)

# Plot ROC curve
plt.plot(fpr, tpr, label=f'Logistic Regression (AUC = {auc:.2f})')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve (Selected Predictors)')
plt.legend()
plt.show()
```



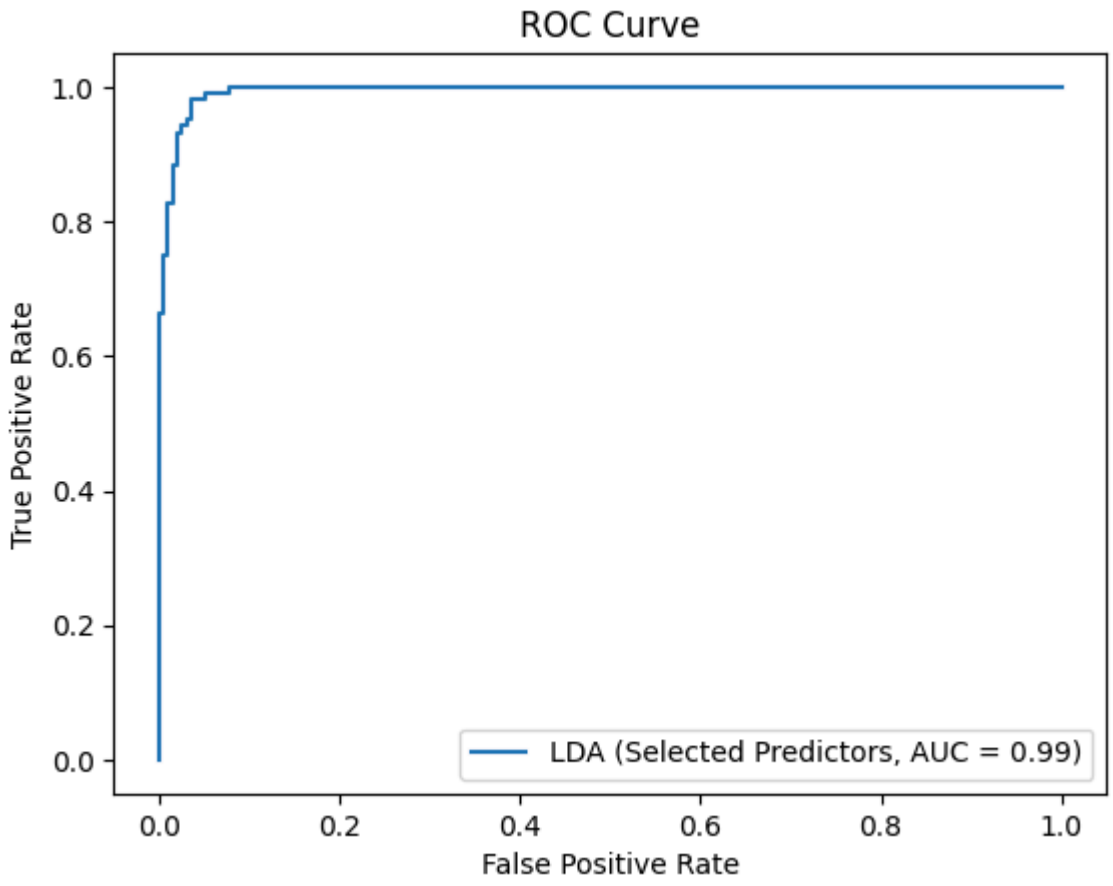

In [4]:

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

lda_model = LinearDiscriminantAnalysis()
lda_model.fit(X_train, y_train)

# Predict probabilities
```

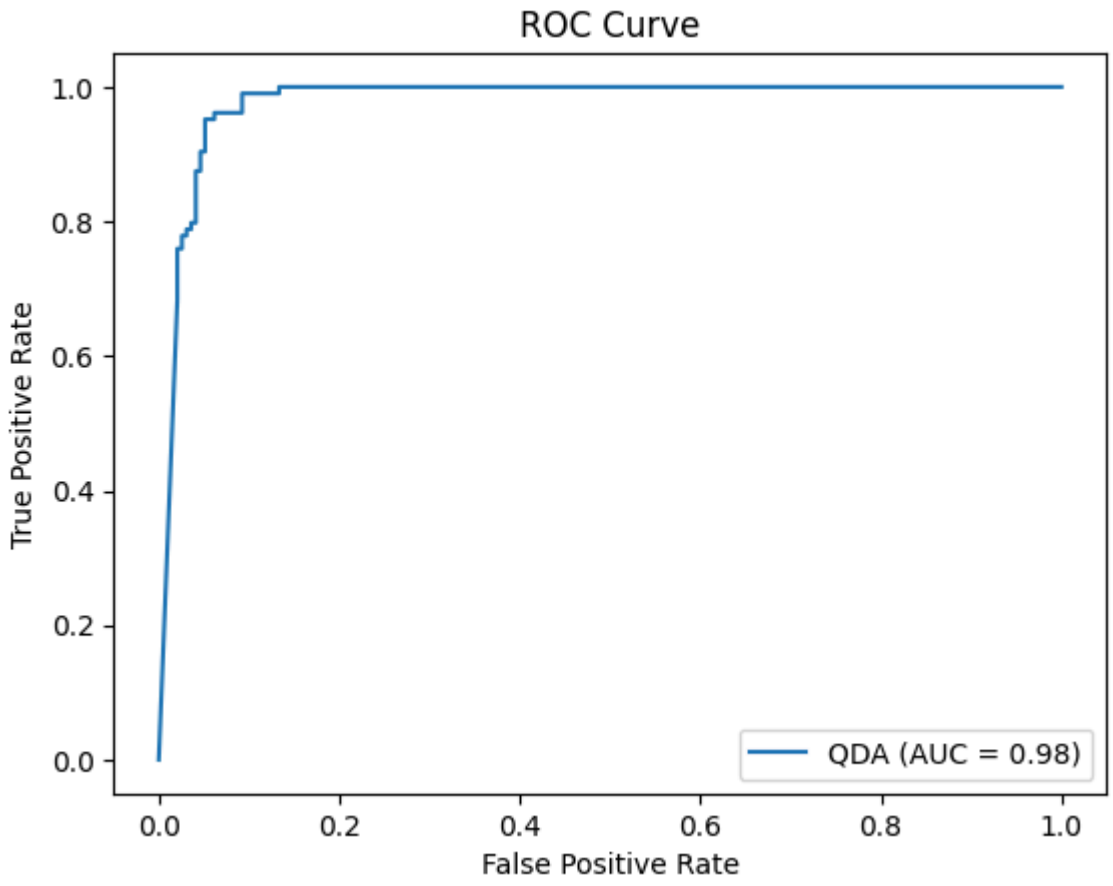
```
y_prob_lda = lda_model.predict_proba(X_test)[:, 1]

# ROC curve and AUC
fpr, tpr, thresholds = roc_curve(y_test, y_prob_lda)
auc = roc_auc_score(y_test, y_prob_lda)

# Plot ROC curve
plt.plot(fpr, tpr, label=f'LDA (AUC = {auc:.2f})')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve')
plt.legend()
plt.show()
```




In [5]:
```
lda_model_selected = LinearDiscriminantAnalysis()
lda_model_selected.fit(X_train_selected, y_train)

# Predict probabilities
y_prob_lda_selected = lda_model_selected.predict_proba(X_test_selected)[:

# ROC curve and AUC
fpr, tpr, thresholds = roc_curve(y_test, y_prob_lda_selected)
auc = roc_auc_score(y_test, y_prob_lda_selected)

# Plot ROC curve
plt.plot(fpr, tpr, label=f'LDA (Selected Predictors, AUC = {auc:.2f})')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve')
plt.legend()
plt.show()
```



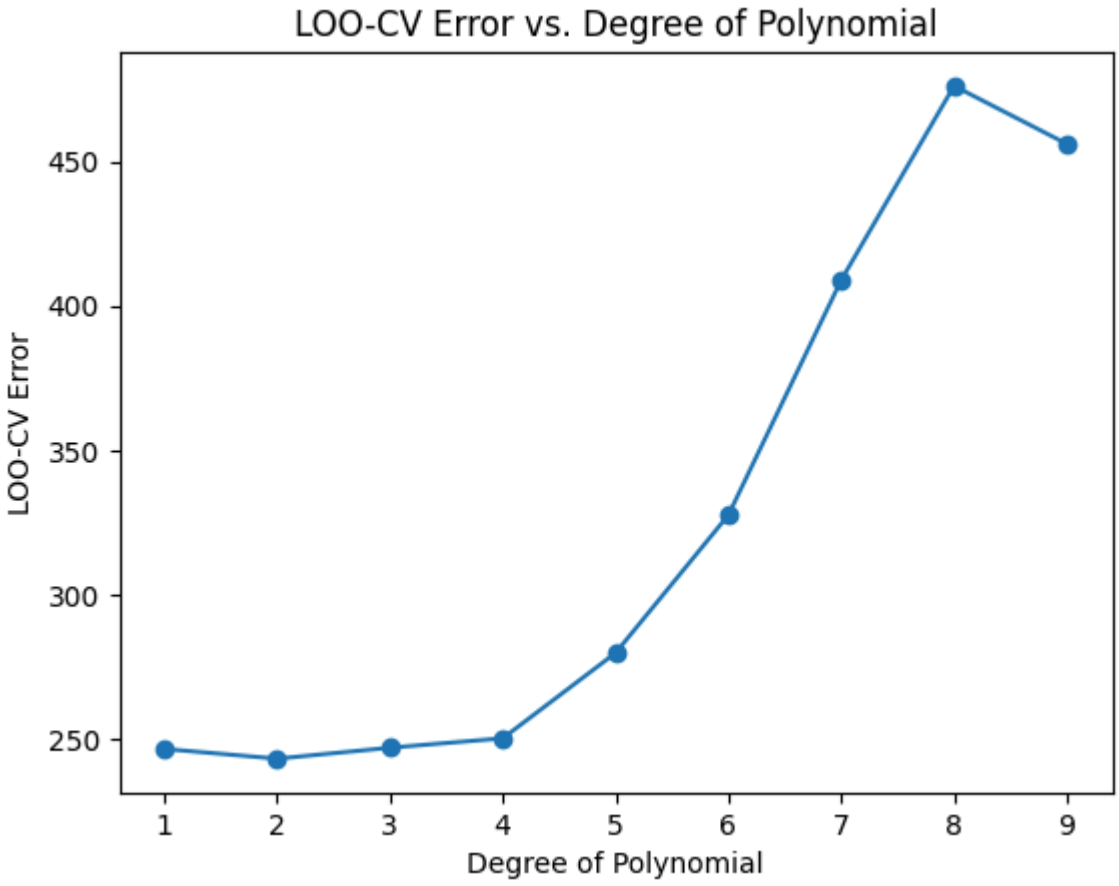

In [6]:
```
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis

qda_model = QuadraticDiscriminantAnalysis()
qda_model.fit(X_train, y_train)

# Predict probabilities
y_prob_qda = qda_model.predict_proba(X_test)[:, 1]

# ROC curve and AUC
fpr, tpr, thresholds = roc_curve(y_test, y_prob_qda)
auc = roc_auc_score(y_test, y_prob_qda)

# Plot ROC curve
plt.plot(fpr, tpr, label=f'QDA (AUC = {auc:.2f})')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve')
plt.legend()
plt.show()
```




# Problem 3

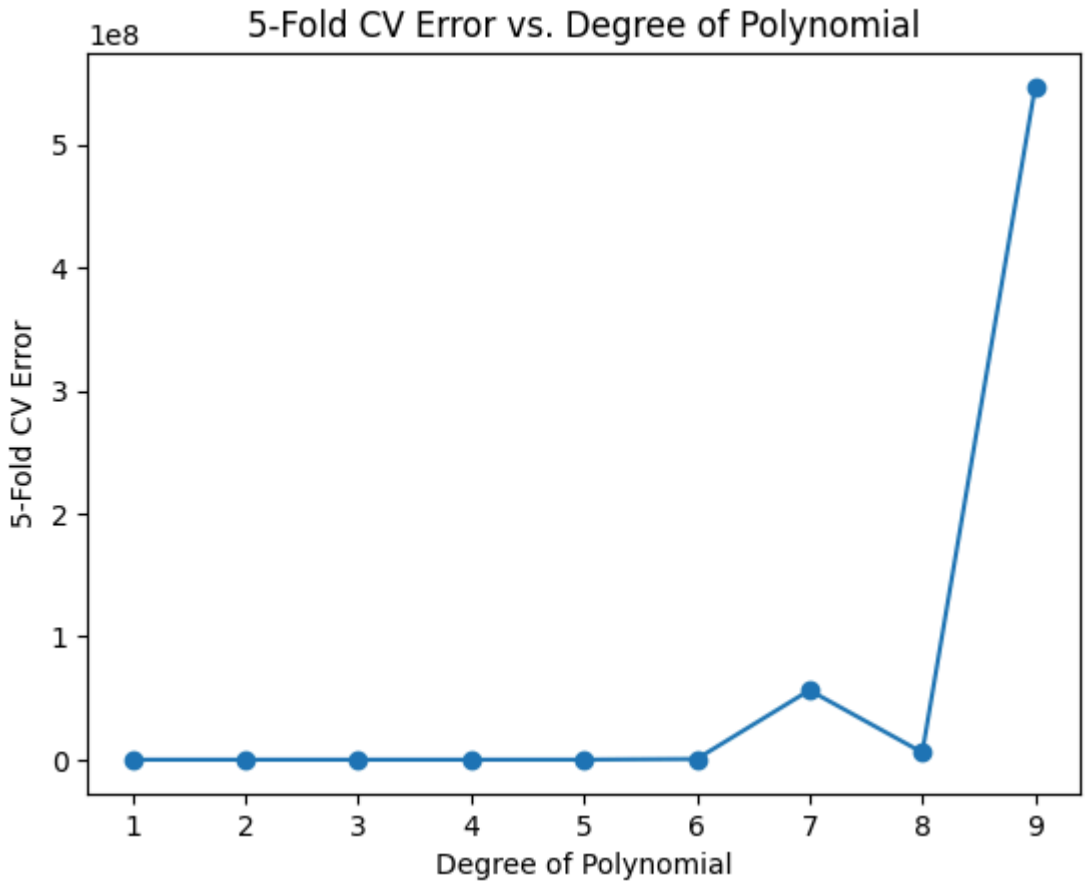
In [7]:
```
import pandas as pd
import numpy as np
from sklearn.model_selection import LeaveOneOut
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt

# Load the dataset
data = pd.read_csv('Cars.csv')

# Separate the features and target variable
X = data[['speed']].values
y = data['dist'].values

# Initialize Leave-One-Out Cross-Validation
loo = LeaveOneOut()

# Initialize list to store CV errors
cv_errors = []

# Loop over polynomial degrees
degrees = list(range(1, 10))
for degree in degrees:
    poly = PolynomialFeatures(degree)
    X_poly = poly.fit_transform(X)

    errors = []
```

```
    # Perform LOO-CV
    for train_idx, test_idx in loo.split(X_poly):
        X_train, X_test = X_poly[train_idx], X_poly[test_idx]
        y_train, y_test = y[train_idx], y[test_idx]

        model = LinearRegression()
        model.fit(X_train, y_train)
        y_pred = model.predict(X_test)
        errors.append(mean_squared_error(y_test, y_pred))

    # Store the averAge error
    cv_errors.append(np.mean(errors))

# Plot the CV errors versus degree of polynomial
plt.plot(degrees, cv_errors, marker='o')
plt.xlabel('Degree of Polynomial')
plt.ylabel('LOO-CV Error')
plt.title('LOO-CV Error vs. Degree of Polynomial')
plt.show()
```




In [8]:
```
from sklearn.model_selection import cross_val_score

# Initialize list to store CV errors for 5-Fold CV
cv_errors_5fold = []

# Loop over polynomial degrees
for degree in degrees:
    poly = PolynomialFeatures(degree)
    X_poly = poly.fit_transform(X)

    model = LinearRegression()
```

```
    mse = -cross_val_score(model, X_poly, y, cv=5, scoring='neg_mean_squa

    # Store the averAge error across the 5 folds
    cv_errors_5fold.append(mse.mean())

# Plot the 5-fold CV errors versus degree of polynomial
plt.plot(degrees, cv_errors_5fold, marker='o')
plt.xlabel('Degree of Polynomial')
plt.ylabel('5-Fold CV Error')
plt.title('5-Fold CV Error vs. Degree of Polynomial')
plt.show()
```




In [9]:
```
from sklearn.neighbors import KNeighborsRegressor
from sklearn.model_selection import GridSearchCV

# Define a range of bandwidths (h values)
h_values = np.linspace(0.1, 1.0, 10)

# Use GridSearchCV to find the best h using LOO-CV and 5-fold CV
knn = KNeighborsRegressor(weights='distance', metric='euclidean')

# Grid search with Leave-One-Out CV
grid_loo = GridSearchCV(knn, param_grid={'n_neighbors': range(1, 10)}, cv
grid_loo.fit(X, y)
loo_best_h = grid_loo.best_params_

# Grid search with 5-Fold CV
grid_5fold = GridSearchCV(knn, param_grid={'n_neighbors': range(1, 10)},
grid_5fold.fit(X, y)
fold5_best_h = grid_5fold.best_params_
```

```
print(f"Best h (bandwidth) with LOO-CV: {loo_best_h}")
print(f"Best h (bandwidth) with 5-Fold CV: {fold5_best_h}")
```

```
Best h (bandwidth) with LOO-CV: {'n_neighbors': 9}
Best h (bandwidth) with 5-Fold CV: {'n_neighbors': 9}
```

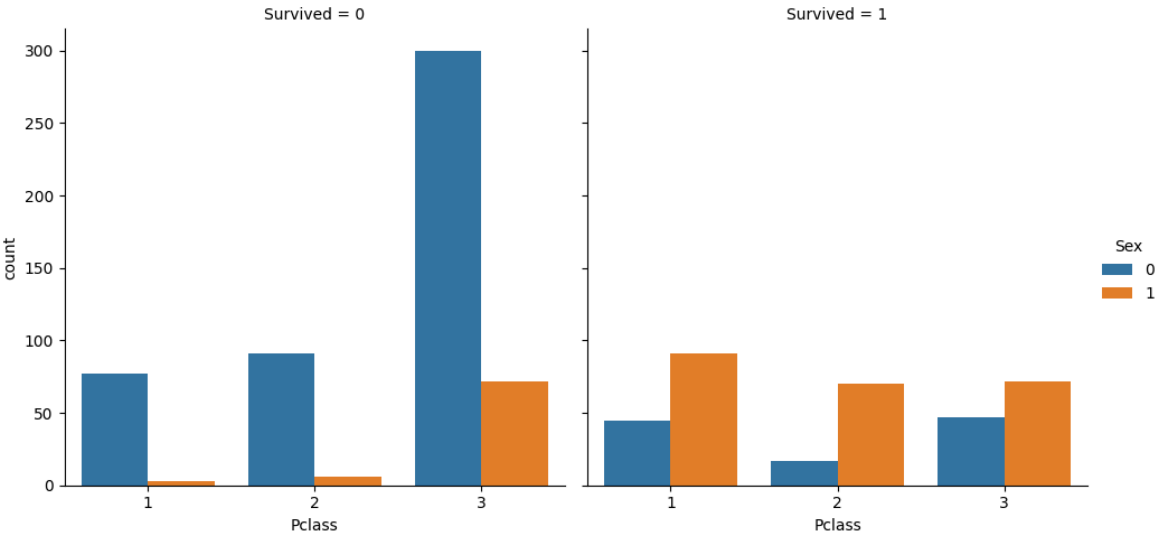
# Problem4

In [10]:

```
# Step 1: Import required libraries
import pandas as pd
import numpy as np
from sklearn.linear_model import LogisticRegression
from sklearn.utils import resample
from sklearn.preprocessing import StandardScaler
import statsmodels.api as sm
import seaborn as sns
import matplotlib.pyplot as plt

# Step 2: Load the Titanic dataset (replace with the correct path to your
df = pd.read_csv('titanic.csv')

# Step 3: Data Preprocessing
# Convert 'Sex' to numerical and handle missing values (if any)
df['Sex'] = df['Sex'].map({'male': 0, 'female': 1})
df['Age'] = df['Age'].fillna(df['Age'].mean())  # Fill missing Ages with

# Prepare the predictor variables (Pclass, Sex, Age, SibSp, Fare)
X = df[['Pclass', 'Sex', 'Age', 'SibSp', 'Fare']]
y = df['Survived']  # 'Survived' is the response variable

# Step 4: Standardize the features (to improve convergence)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Add constant for intercept in statsmodels (for logistic regression in s
X_sm = sm.add_constant(X_scaled)

# Step 5: Fit Logistic Regression Model with Statsmodels
logit_model = sm.Logit(y, X_sm)
result = logit_model.fit()
print(result.summary())  # This will give you the coefficients and confid

# Step 6: Bootstrap the model coefficients
def bootstrap_logit(X, y, n_iterations=1000):
    coefs = []
    for i in range(n_iterations):
        X_resample, y_resample = resample(X, y)
        model = sm.Logit(y_resample, sm.add_constant(X_resample)).fit(dis
        coefs.append(model.params)
    return pd.DataFrame(coefs)

# Apply bootstrapping
coefs = bootstrap_logit(X_scaled, y)
ci = coefs.quantile([0.025, 0.975])
print("Bootstrap Confidence Intervals:")
print(ci)

# Step 7: Exploratory Data Analysis (EDA)
```

```
# Plot survival rates by class and Sex
sns.catplot(x="Pclass", hue="Sex", col="Survived", data=df, kind="count")
plt.show()

# Step 8: Logistic Regression Prediction for the first observation using
# Remove the first observation and fit the model on the remaining data
X_train = X_scaled[1:]
y_train = y.iloc[1:]
X_test = X_scaled[0].reshape(1, -1)  # First observation for test

# Fit the Logistic Regression model using sklearn with increased max_iter
model = LogisticRegression(solver='lbfgs', max_iter=1000)
model.fit(X_train, y_train)

# Predict survival probability for the first observation
prob = model.predict_proba(X_test)[0, 1]
print(f"Predicted survival probability for the first observation: {prob}"

# Step 9: Bootstrap to estimate prediction interval for the first observa
boot_preds = []
for i in range(1000):
    X_resample, y_resample = resample(X_train, y_train)
    model.fit(X_resample, y_resample)
    boot_preds.append(model.predict_proba(X_test)[0, 1])

# 95% Prediction Interval
lower_bound = np.percentile(boot_preds, 2.5)
upper_bound = np.percentile(boot_preds, 97.5)
print(f"95% Prediction Interval: [{lower_bound}, {upper_bound}]")
```

```
Optimization terminated successfully.
         Current function value: 0.443127
         Iterations 6
                           Logit Regression Results
==============================================================================
====
Dep. Variable:                Survived   No. Observations:
891
Model:                           Logit   Df Residuals:
885
Method:                            MLE   Df Model:
5
Date:                Tue, 15 Oct 2024   Pseudo R-squ.:                   0.
3346
Time:                        23:55:40   Log-Likelihood:                -39
4.83
converged:                       True   LL-Null:                       -59
3.33
Covariance Type:            nonrobust   LLR p-value:                 1.312
e-83
==============================================================================
====
                 coef    std err          z      P>|z|      [0.025      0.
975]
------------------------------------------------------------------------------
----
const         -0.6499      0.091     -7.125      0.000      -0.829      -
0.471
x1            -0.9176      0.115     -7.960      0.000      -1.144      -
0.692
x2             1.3025      0.093     14.013      0.000       1.120
1.485
x3            -0.5118      0.101     -5.067      0.000      -0.710      -
0.314
x4            -0.4173      0.117     -3.565      0.000      -0.647      -
0.188
x5             0.1178      0.112      1.054      0.292      -0.101
0.337
==============================================================================
====
Bootstrap Confidence Intervals:
          const        x1        x2        x3        x4        x5
0.025 -0.848945 -1.156754  1.153973 -0.728982 -0.655827 -0.121660
0.975 -0.467461 -0.696008  1.502998 -0.296140 -0.208960  0.403789
```



```
Predicted survival probability for the first observation: 0.09274024824413
34
95% Prediction Interval: [0.06562679532088944, 0.12257934969802953]
```

In [ ]: