Machine Learning Lab Manual

[Course Code: CSP473]

Experiment-1: Understanding problem solving through Machine Learning and Problem identification.

Faculty Need to Do:

- Faculty need to explain the standard procedure to identification problem statement, how machine learning can solve those objectives.

Students Need to Do:

- Identify a real-world problem suitable for Machine Learning.
- Define the problem statement, objectives, and expected outcome.
- Determine the type of ML problem (classification, regression, clustering, etc.).
- Identify possible datasets, input features, and target variable.
- Present the proposed ML solution.

Note: This help student to prepare their project problem statement and concept. Students need to prepare and submit a review document.

Experiment-2: State of the art review for Tools used in Machine learning Experiments, Project development.

First Students Need to Do:

- Review the Tools which are commonly used in Machine learning Experiments, Project development. Also, need to prepare and submit the document.

Faculty Need to Do:

- Then faculty need to describe tools and libraries which aren't mentioned by the students in the report.

Experiment-3: Write a Program to load and view dataset file. [Iris dataset]

Sample Code:

```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
```

```
from sklearn.datasets import load_iris

# 1. Load the Iris dataset
iris = load_iris()
X = iris.data
y = iris.target

# 2. View the dataset
print("--- Iris Dataset Information ---")
print(f"Feature Names: {iris.feature_names}")
print(f"Target Names: {iris.target_names}")
print("\nFirst 5 rows of features (X):\n", X[:5])
print("\nFirst 5 rows of target (y):\n", y[:5])

# Create a Pandas DataFrame for easier visualization
iris_df = pd.DataFrame(data=X, columns=iris.feature_names)
iris_df['species'] = iris.target
iris_df['species'] = iris_df['species'].map(lambda x: iris.target_names[x]) # Map numerical target to names

print("\nDataFrame Head:\n", iris_df.head())

# 3. Visualize the dataset
print("\n--- Visualizing the Iris Dataset with Pair Plot ---")
sns.pairplot(iris_df, hue='species', palette='viridis')
plt.suptitle('Pair Plot of Iris Dataset (Features by Species)', y=1.02) # Adjust title position
plt.show()
```

Experiment-4: Write a program to implement simple linear regression using housing price prediction problem. [California Housing Dataset]

Sample Code:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score
```

```
# 1. Load the California Housing dataset
housing = fetch_california_housing(as_frame=True)

# For simple linear regression, select one feature (e.g., Median Income) and the target (Median House Value)
X = housing.data[['MedInc']]  # Independent variable (feature)
y = housing.target       # Dependent variable (target)

print("--- California Housing Dataset Information ---")
print(f"Features used: {X.columns.tolist()}")
print(f"Target variable: {housing.target_names[0]}")
print("\nFirst 5 rows of selected feature (X):\n", X.head())
print("\nFirst 5 rows of target (y):\n", y.head())

# 2. Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

print(f"\nTraining samples: {len(X_train)}")
print(f"Testing samples: {len(X_test)}")

# 3. Initialize and train the Simple Linear Regression model
model = LinearRegression()
model.fit(X_train, y_train)

# 4. Make predictions on the test set
y_pred = model.predict(X_test)

# 5. Evaluate the model
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print(f"\n--- Model Evaluation ---")
print(f"Intercept: {model.intercept_:.2f}")
print(f"Coefficient (MedInc): {model.coef_[0]:.2f}")
print(f"Mean Squared Error (MSE): {mse:.2f}")
print(f"R-squared (R2): {r2:.2f}")

# 6. Visualize the regression line
plt.figure(figsize=(10, 6))
sns.scatterplot(x=X_test['MedInc'], y=y_test, alpha=0.6, label='Actual Values')
plt.plot(X_test['MedInc'], y_pred, color='red', linewidth=2, label='Regression Line')
plt.title('Simple Linear Regression: Median Income vs. Median House Value')
plt.xlabel('Median Income (MedInc)')
plt.ylabel('Median House Value (MedHouseVal)')
```

```
plt.legend()
plt.grid(True)
plt.show()
```

Experiment-5: Write a program to implement binary logistic regression using cancer identification problem. [Breast Cancer Wisconsin Dataset]

Sample Code:

```
import numpy as np
import pandas as pd
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns

# 1. Load the Breast Cancer Wisconsin dataset
cancer = load_breast_cancer()
X = cancer.data
y = cancer.target

print("--- Breast Cancer Wisconsin Dataset Information ---")
print(f"Feature Names: {cancer.feature_names}")
print(f"Target Names: {cancer.target_names} (0: Malignant, 1: Benign)")
print(f"Shape of features (X): {X.shape}")
print(f"Shape of target (y): {y.shape}")

# 2. Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42, stratify=y)

print(f"\nTraining samples: {len(X_train)}")
print(f"Testing samples: {len(X_test)}")

# 3. Scale the features (important for Logistic Regression)
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# 4. Initialize and train the Logistic Regression model
model = LogisticRegression(random_state=42, solver='liblinear') # 'liblinear' is good for small datasets
```

```
model.fit(X_train_scaled, y_train)

# 5. Make predictions on the test set
y_pred = model.predict(X_test_scaled)

# 6. Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
report = classification_report(y_test, y_pred, target_names=cancer.target_names)
cm = confusion_matrix(y_test, y_pred)

print(f"\n--- Model Evaluation (Logistic Regression) ---")
print(f"Accuracy: {accuracy:.4f}")
print("\nClassification Report:")
print(report)

print("\nConfusion Matrix:")
print(cm)

# Visualize the Confusion Matrix
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
        xticklabels=cancer.target_names, yticklabels=cancer.target_names)
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Confusion Matrix for Breast Cancer Prediction')
plt.show()
```

Experiment-6: Write a program to Implement the Gradient Descent algorithm to optimize model parameters and visualize the convergence of the cost function. [California Housing Dataset]

Sample Code:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.datasets import fetch_california_housing
from sklearn.preprocessing import StandardScaler

# 1. Load the California Housing dataset
housing = fetch_california_housing(as_frame=True)

# For simple linear regression with Gradient Descent, we'll use one feature and add a bias term.
# Target: Median House Value (MedHouseVal)
```

```
# Feature: Median Income (MedInc)

X = housing.data[['MedInc']].values  # Convert to numpy array for matrix operations
y = housing.target.values.reshape(-1, 1) # Reshape y to a column vector

print("--- California Housing Dataset for Gradient Descent ---")
print(f"Feature used: {'MedInc'}")
print(f"Target variable: {housing.target_names[0]}")
print("\nFirst 5 rows of selected feature (X):\n", X[:5])
print("\nFirst 5 rows of target (y):\n", y[:5])

# 2. Data Preprocessing:
#    a. Normalize the feature (MedInc) for better Gradient Descent performance
scaler_X = StandardScaler()
X_scaled = scaler_X.fit_transform(X)

#    b. Add a bias (intercept) term to X_scaled
#       This adds a column of ones to X so that theta_0 (intercept) can be learned.
X_b = np.c_[np.ones((X_scaled.shape[0], 1)), X_scaled]

print(f"\nShape of X_b (features with bias): {X_b.shape}")
print("First 5 rows of X_b (bias + scaled MedInc):\n", X_b[:5])

# 3. Implement Gradient Descent
def gradient_descent(X, y, learning_rate=0.01, n_iterations=1000):
    m = len(y) # Number of training examples
    theta = np.random.randn(X.shape[1], 1) # Initialize parameters (weights) randomly
    cost_history = []

    for iteration in range(n_iterations):
        # Calculate predictions
        predictions = X.dot(theta)

        # Calculate error
        errors = predictions - y

        # Calculate cost (Mean Squared Error)
        cost = (1/(2*m)) * np.sum(errors**2)
        cost_history.append(cost)

        # Calculate gradients
        gradients = (1/m) * X.T.dot(errors)

        # Update parameters
        theta = theta - learning_rate * gradients
```

```
    return theta, cost_history

# Set hyperparameters for Gradient Descent
learning_rate = 0.01
n_iterations = 2000

# Run Gradient Descent
print(f"\n--- Running Gradient Descent (Learning Rate: {learning_rate}, Iterations: {n_iterations}) ---")
theta_optimized, cost_history = gradient_descent(X_b, y, learning_rate, n_iterations)

print(f"Optimized Parameters (Theta):\n{theta_optimized}")
print(f"Final Cost (MSE): {cost_history[-1]:.4f}")

# --- Additional Visualizations for better understanding ---

# 4. Visualize the convergence of the cost function
plt.figure(figsize=(10, 6))
plt.plot(range(n_iterations), cost_history, color='blue')
plt.xlabel('Number of Iterations')
plt.ylabel('Cost (MSE)')
plt.title('Convergence of Cost Function during Gradient Descent')
plt.grid(True)
plt.show()

# 5. Visualize the regression line found by Gradient Descent
# Generate points for the regression line using the optimized theta
X_plot = np.linspace(X_scaled.min(), X_scaled.max(), 100).reshape(-1, 1)
X_plot_b = np.c_[np.ones((100, 1)), X_plot]
y_pred_gd = X_plot_b.dot(theta_optimized)

plt.figure(figsize=(10, 6))
sns.scatterplot(x=X_scaled.flatten(), y=y.flatten(), alpha=0.6, label='Scaled Actual Values')
plt.plot(X_plot.flatten(), y_pred_gd.flatten(), color='red', linewidth=2, label='Gradient Descent Regression Line')
plt.title('Gradient Descent: Scaled Median Income vs. Median House Value')
plt.xlabel('Scaled Median Income (MedInc)')
plt.ylabel('Median House Value (MedHouseVal)')
plt.legend()
plt.grid(True)
plt.show()

# 6. Distribution of the Target Variable (MedHouseVal)
plt.figure(figsize=(10, 6))
sns.histplot(y.flatten(), kde=True, bins=50, color='purple')
plt.title('Distribution of Median House Value')
```

```
plt.xlabel('Median House Value')
plt.ylabel('Frequency')
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.show()

# 7. Actual vs. Predicted Values (from GD model)
# Need to make predictions for the whole dataset to compare with actuals for visualization clarity
all_predictions_gd = X_b.dot(theta_optimized)

plt.figure(figsize=(10, 6))
sns.scatterplot(x=y.flatten(), y=all_predictions_gd.flatten(), alpha=0.6, color='green')
plt.plot([y.min(), y.max()], [y.min(), y.max()], 'r--', lw=2, label='Perfect Prediction Line')
plt.title('Actual vs. Predicted Median House Values (Gradient Descent)')
plt.xlabel('Actual Median House Value')
plt.ylabel('Predicted Median House Value')
plt.legend()
plt.grid(True)
plt.show()

# 8. Residuals Plot (Errors vs. Predicted Values)
residuals = y - all_predictions_gd

plt.figure(figsize=(10, 6))
sns.scatterplot(x=all_predictions_gd.flatten(), y=residuals.flatten(), alpha=0.6, color='orange')
plt.axhline(y=0, color='red', linestyle='--', label='Zero Residual Line')
plt.title('Residuals Plot (Gradient Descent)')
plt.xlabel('Predicted Median House Value')
plt.ylabel('Residuals (Actual - Predicted)')
plt.legend()
plt.grid(True)
plt.show()
```

Experiment-7: Write a program to Investigate the effect of learning rate and number of iterations on Gradient Descent convergence. [California Housing Dataset]

Sample Code:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_california_housing
from sklearn.preprocessing import StandardScaler

# Re-load data and preprocess (as it was done in the previous cell)
# This ensures the cell is self-contained if run independently
```

```
housing = fetch_california_housing(as_frame=True)
X = housing.data[['MedInc']].values
y = housing.target.values.reshape(-1, 1)

scaler_X = StandardScaler()
X_scaled = scaler_X.fit_transform(X)
X_b = np.c_[np.ones((X_scaled.shape[0], 1)), X_scaled]

# Re-define the Gradient Descent function for this investigation
def gradient_descent_investigation(X, y, learning_rate, n_iterations):
    m = len(y)
    theta = np.random.randn(X.shape[1], 1)
    cost_history = []

    for iteration in range(n_iterations):
        predictions = X.dot(theta)
        errors = predictions - y
        cost = (1/(2*m)) * np.sum(errors**2)
        cost_history.append(cost)
        gradients = (1/m) * X.T.dot(errors)
        theta = theta - learning_rate * gradients

    return theta, cost_history

# Define different learning rates and iterations to investigate
learning_rates = [0.001, 0.01, 0.1]
n_iterations_list = [500, 2000, 5000]

results = {}

print("--- Investigating Gradient Descent Hyperparameters ---")

# Run Gradient Descent for each combination and store results
for lr in learning_rates:
    for n_iter in n_iterations_list:
        print(f"\nRunning GD with LR={lr}, Iterations={n_iter}...")
        theta, costs = gradient_descent_investigation(X_b, y, lr, n_iter)
        results[(lr, n_iter)] = costs
        print(f"  Final Cost: {costs[-1]:.4f}")

# Visualize the convergence of the cost function for each scenario
plt.figure(figsize=(12, 8))
colors = ['blue', 'green', 'red', 'cyan', 'magenta', 'yellow', 'black', 'purple', 'orange']
color_idx = 0

for (lr, n_iter), costs in results.items():
```

```
    plt.plot(range(len(costs)), costs, label=f'LR={lr}, Iter={n_iter}', color=colors[color_idx])
    color_idx = (color_idx + 1) % len(colors)

plt.xlabel('Number of Iterations')
plt.ylabel('Cost (MSE)')
plt.title('Effect of Learning Rate and Iterations on Gradient Descent Convergence')
plt.legend()
plt.grid(True)
plt.ylim(bottom=0) # Ensure cost doesn't go below 0 on the plot
plt.show()
```

Experiment-8: Write a program to implement Ridge and Lasso Regularized Linear Regression and compare their performance with standard Linear Regression. [California Housing Dataset] and [Breast Cancer Wisconsin Dataset]

Sample Code:

```
import numpy as np
import pandas as pd
from sklearn.datasets import fetch_california_housing, load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LinearRegression, Ridge, Lasso, LogisticRegression
from sklearn.metrics import mean_squared_error, r2_score, accuracy_score, classification_report
import matplotlib.pyplot as plt
import seaborn as sns

print("--- Comparing Standard, Ridge, and Lasso Linear Regression ---\n")

# --- California Housing Dataset (Regression Problem) ---
print("--- California Housing Dataset (Regression) ---")

# 1. Load Data
housing = fetch_california_housing(as_frame=True)
X_housing = housing.data
y_housing = housing.target

# 2. Split Data
X_train_h, X_test_h, y_train_h, y_test_h = train_test_split(X_housing, y_housing, test_size=0.2, random_state=42)

# 3. Scale Features
scaler_h = StandardScaler()
```

```
X_train_scaled_h = scaler_h.fit_transform(X_train_h)
X_test_scaled_h = scaler_h.transform(X_test_h)

# 4. Train and Evaluate Models

# Standard Linear Regression
lin_reg = LinearRegression()
lin_reg.fit(X_train_scaled_h, y_train_h)
y_pred_lin = lin_reg.predict(X_test_scaled_h)
mse_lin = mean_squared_error(y_test_h, y_pred_lin)
r2_lin = r2_score(y_test_h, y_pred_lin)
print(f"\nStandard Linear Regression:")
print(f"  MSE: {mse_lin:.4f}")
print(f"  R2 Score: {r2_lin:.4f}")

# Ridge Regression
# alpha is the regularization strength; larger values specify stronger regularization.
ridge_reg = Ridge(alpha=1.0, random_state=42)
ridge_reg.fit(X_train_scaled_h, y_train_h)
y_pred_ridge = ridge_reg.predict(X_test_scaled_h)
mse_ridge = mean_squared_error(y_test_h, y_pred_ridge)
r2_ridge = r2_score(y_test_h, y_pred_ridge)
print(f"\nRidge Regression (alpha=1.0):")
print(f"  MSE: {mse_ridge:.4f}")
print(f"  R2 Score: {r2_ridge:.4f}")

# Lasso Regression
# alpha is the regularization strength. For Lasso, it can lead to sparsity.
lasso_reg = Lasso(alpha=0.01, random_state=42, max_iter=2000) # Increased max_iter for convergence
lasso_reg.fit(X_train_scaled_h, y_train_h)
y_pred_lasso = lasso_reg.predict(X_test_scaled_h)
mse_lasso = mean_squared_error(y_test_h, y_pred_lasso)
r2_lasso = r2_score(y_test_h, y_pred_lasso)
print(f"\nLasso Regression (alpha=0.01):")
print(f"  MSE: {mse_lasso:.4f}")
print(f"  R2 Score: {r2_lasso:.4f}")

print("\n" + "="*60 + "\n")

# --- Breast Cancer Wisconsin Dataset (Classification Problem) ---
print("--- Breast Cancer Wisconsin Dataset (Classification) ---")
print("Note: For binary classification, Logistic Regression is the appropriate linear model. Its L1 and L2 penalties serve as the equivalent of Lasso and Ridge regularization.\n")

# 1. Load Data
```

```
cancer = load_breast_cancer()
X_cancer = cancer.data
y_cancer = cancer.target

# 2. Split Data
X_train_c, X_test_c, y_train_c, y_test_c = train_test_split(X_cancer, y_cancer, test_size=0.2,
random_state=42, stratify=y_cancer)

# 3. Scale Features
scaler_c = StandardScaler()
X_train_scaled_c = scaler_c.fit_transform(X_train_c)
X_test_scaled_c = scaler_c.transform(X_test_c)

# 4. Train and Evaluate Models (Logistic Regression with different penalties)

# Logistic Regression (default L2 penalty)
log_reg_default = LogisticRegression(random_state=42, solver='liblinear')
log_reg_default.fit(X_train_scaled_c, y_train_c)
y_pred_log_default = log_reg_default.predict(X_test_scaled_c)
acc_log_default = accuracy_score(y_test_c, y_pred_log_default)
print(f"Logistic Regression (Default L2 Regularization):")
print(f"  Accuracy: {acc_log_default:.4f}")
print("  Classification Report:\n", classification_report(y_test_c, y_pred_log_default,
target_names=cancer.target_names))

# Logistic Regression with L2 penalty (Ridge equivalent)
# C is the inverse of regularization strength; smaller C values specify stronger regularization.
log_reg_l2 = LogisticRegression(penalty='l2', C=1.0, random_state=42, solver='liblinear')
log_reg_l2.fit(X_train_scaled_c, y_train_c)
y_pred_log_l2 = log_reg_l2.predict(X_test_scaled_c)
acc_log_l2 = accuracy_score(y_test_c, y_pred_log_l2)
print(f"\nLogistic Regression (L2 Penalty, C=1.0 - Ridge equivalent):")
print(f"  Accuracy: {acc_log_l2:.4f}")
print("  Classification Report:\n", classification_report(y_test_c, y_pred_log_l2,
target_names=cancer.target_names))

# Logistic Regression with L1 penalty (Lasso equivalent)
# C is the inverse of regularization strength.
log_reg_l1 = LogisticRegression(penalty='l1', C=1.0, random_state=42, solver='liblinear')
log_reg_l1.fit(X_train_scaled_c, y_train_c)
y_pred_log_l1 = log_reg_l1.predict(X_test_scaled_c)
acc_log_l1 = accuracy_score(y_test_c, y_pred_log_l1)
print(f"\nLogistic Regression (L1 Penalty, C=1.0 - Lasso equivalent):")
print(f"  Accuracy: {acc_log_l1:.4f}")
print("  Classification Report:\n", classification_report(y_test_c, y_pred_log_l1,
target_names=cancer.target_names))
```

```
# --- Visual Comparison of Coefficients for California Housing (Optional, for deeper analysis)
---
print("\n" + "="*60 + "\n")
print("--- Visualizing Coefficients for California Housing Models ---")

coefficients = {
    'Linear Reg': lin_reg.coef_,
    'Ridge Reg': ridge_reg.coef_,
    'Lasso Reg': lasso_reg.coef_,
}

# Ensure all coefficients are 1D arrays for plotting if they are not already
for key in coefficients:
    if coefficients[key].ndim > 1:
        coefficients[key] = coefficients[key].flatten()

coef_df = pd.DataFrame({
    'Feature': X_housing.columns,
    'Linear Reg': coefficients['Linear Reg'],
    'Ridge Reg': coefficients['Ridge Reg'],
    'Lasso Reg': coefficients['Lasso Reg']
})

coef_df.set_index('Feature', inplace=True)

plt.figure(figsize=(12, 7))
sns.heatmap(coef_df.T, annot=True, cmap='coolwarm', fmt=".2f", linewidths=.5)
plt.title('Comparison of Model Coefficients for California Housing')
plt.ylabel('Model')
plt.show()
```

Experiment-9: Write a program to implement develop a Regularized Logistic Regression model and analyze the impact of regularization strength on classification accuracy. [California Housing Dataset] and [Breast Cancer Wisconsin Dataset]

Sample Code:

```
import numpy as np
import pandas as pd
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
```

```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt
import seaborn as sns

print("--- Regularized Logistic Regression on Breast Cancer Dataset ---")

# 1. Load the Breast Cancer Wisconsin dataset
cancer = load_breast_cancer()
X = cancer.data
y = cancer.target

# 2. Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42, stratify=y)

# 3. Scale the features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Define a range of C values (inverse of regularization strength)
c_values = [0.001, 0.01, 0.1, 1, 10, 100, 1000]

# Initialize lists to store results
accuracy_l1 = []
accuracy_l2 = []
coefficients_l1 = []
coefficients_l2 = []

print("\n--- Training Models with Varying Regularization Strengths ---")
for c in c_values:
    # Logistic Regression with L1 penalty (Lasso-like)
    # solver='liblinear' supports both L1 and L2 penalties and is good for smaller datasets
    log_reg_l1 = LogisticRegression(penalty='l1', C=c, solver='liblinear', random_state=42, max_iter=1000)
    log_reg_l1.fit(X_train_scaled, y_train)
    y_pred_l1 = log_reg_l1.predict(X_test_scaled)
    acc_l1 = accuracy_score(y_test, y_pred_l1)
    accuracy_l1.append(acc_l1)
    coefficients_l1.append(log_reg_l1.coef_.flatten())
    print(f"C={c:<7} (L1): Accuracy = {acc_l1:.4f}")

    # Logistic Regression with L2 penalty (Ridge-like)
    log_reg_l2 = LogisticRegression(penalty='l2', C=c, solver='liblinear', random_state=42, max_iter=1000)
```

```
    log_reg_l2.fit(X_train_scaled, y_train)
    y_pred_l2 = log_reg_l2.predict(X_test_scaled)
    acc_l2 = accuracy_score(y_test, y_pred_l2)
    accuracy_l2.append(acc_l2)
    coefficients_l2.append(log_reg_l2.coef_.flatten())
    print(f"C={c:<7} (L2): Accuracy = {acc_l2:.4f}")

# --- Visualization ---

# 1. Plot Accuracy vs. C (Regularization Strength)
plt.figure(figsize=(12, 6))
plt.plot(c_values, accuracy_l1, marker='o', label='L1 Regularization (Lasso)', linestyle='--')
plt.plot(c_values, accuracy_l2, marker='x', label='L2 Regularization (Ridge)')
plt.xscale('log') # Use log scale for C values as they span orders of magnitude
plt.xlabel('C (Inverse of Regularization Strength, log scale)')
plt.ylabel('Accuracy')
plt.title('Impact of Regularization Strength (C) on Logistic Regression Accuracy')
plt.legend()
plt.grid(True)
plt.show()

# 2. Plot Coefficients for different C values
# Convert coefficient lists to DataFrames for easier plotting
coef_df_l1 = pd.DataFrame(coefficients_l1, index=c_values, columns=cancer.feature_names)
coef_df_l2 = pd.DataFrame(coefficients_l2, index=c_values, columns=cancer.feature_names)

# Select a few C values to visualize the coefficients more clearly
selected_c_values = [0.01, 1, 100]

plt.figure(figsize=(15, 8))
plt.subplot(1, 2, 1)
sns.heatmap(coef_df_l1.loc[selected_c_values].T, annot=True, cmap='coolwarm', fmt=".2f",
linewidths=.5)
plt.title('L1 Regularization: Coefficients across selected C values')
plt.xlabel('C Value')
plt.ylabel('Feature')

plt.subplot(1, 2, 2)
sns.heatmap(coef_df_l2.loc[selected_c_values].T, annot=True, cmap='coolwarm', fmt=".2f",
linewidths=.5)
plt.title('L2 Regularization: Coefficients across selected C values')
plt.xlabel('C Value')
plt.ylabel('Feature')

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

```
print("\n--- Interpretation of Results ---")
print("The accuracy plot shows how C affects the model's performance. Generally, an optimal C value exists where the model generalizes best. Very small C values (strong regularization) might lead to underfitting, while very large C values (weak regularization) might lead to overfitting.")
print("The coefficient heatmaps illustrate the effect of L1 and L2 penalties:")
print("- L1 (Lasso) regularization tends to drive some coefficients to exactly zero (feature selection), especially with smaller C values.")
print("- L2 (Ridge) regularization shrinks coefficients towards zero but rarely makes them exactly zero. It spreads the effect more evenly among features.")
```

Experiment-10: Write a program to implement a Support Vector Regression (SVR) model using an appropriate dataset and compare different kernel functions. Evaluate the performance of Support Vector Regression using MAE, MSE, and R^2 score. [California Housing or UCI Diabetes Dataset]

Sample Code:

```
import numpy as np
import pandas as pd
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVR
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
import matplotlib.pyplot as plt
import seaborn as sns

print("--- Support Vector Regression (SVR) with Different Kernels on California Housing Dataset ---")

# 1. Load the California Housing dataset
housing = fetch_california_housing(as_frame=True)
X = housing.data
y = housing.target

# 2. Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# 3. Scale the features (important for SVR)
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

```
# Define different kernel functions to compare
kernels = ['linear', 'poly', 'rbf', 'sigmoid']

results = {}

print("\n--- Training and Evaluating SVR Models ---")
for kernel_name in kernels:
    print(f"\nTraining SVR with {kernel_name} kernel...")
    # Initialize SVR model with the current kernel
     # For 'poly' kernel, degree and gamma might need tuning. C and epsilon are general parameters.
    if kernel_name == 'poly':
        svr = SVR(kernel=kernel_name, C=100, epsilon=0.1, degree=2, gamma='scale')
    else:
        svr = SVR(kernel=kernel_name, C=100, epsilon=0.1, gamma='scale') # C: Regularization parameter, epsilon: epsilon-tube within which no penalty is associated in the training loss function

    # Train the model
    svr.fit(X_train_scaled, y_train)

    # Make predictions
    y_pred = svr.predict(X_test_scaled)

    # Evaluate performance
    mae = mean_absolute_error(y_test, y_pred)
    mse = mean_squared_error(y_test, y_pred)
    rmse = np.sqrt(mse) # Root Mean Squared Error
    r2 = r2_score(y_test, y_pred)

    results[kernel_name] = {'MAE': mae, 'MSE': mse, 'RMSE': rmse, 'R2': r2}

    print(f"  MAE: {mae:.4f}")
    print(f"  MSE: {mse:.4f}")
    print(f"  RMSE: {rmse:.4f}")
    print(f"  R2 Score: {r2:.4f}")

print("\n--- Comparison of SVR Model Performance ---")
results_df = pd.DataFrame(results).T
print(results_df)

# --- Visualization of Results (Optional: for one best performing kernel) ---
# Let's visualize the predictions for the kernel with the highest R2 score
best_kernel = results_df['R2'].idxmax()
print(f"\nVisualizing predictions for the best performing kernel: {best_kernel}")
```

```
# Retrain the best model to get its predictions
if best_kernel == 'poly':
    svr_best = SVR(kernel=best_kernel, C=100, epsilon=0.1, degree=2, gamma='scale')
else:
    svr_best = SVR(kernel=best_kernel, C=100, epsilon=0.1, gamma='scale')
svr_best.fit(X_train_scaled, y_train)
y_pred_best = svr_best.predict(X_test_scaled)

plt.figure(figsize=(10, 6))
sns.scatterplot(x=y_test, y=y_pred_best, alpha=0.6)
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], 'r--', lw=2, label='Perfect Prediction Line')
plt.xlabel('Actual Median House Value')
plt.ylabel('Predicted Median House Value')
plt.title(f'SVR ({best_kernel} kernel): Actual vs. Predicted Values')
plt.legend()
plt.grid(True)
plt.show()

print("\n--- Interpretation ---")
print("From the comparison table, you can observe how different kernel functions impact the SVR model's performance on the California Housing dataset.")
print("The 'rbf' (Radial Basis Function) kernel is often a good default choice and performs well in many cases due to its ability to handle non-linear relationships. 'Linear' kernel is suitable for linearly separable data. 'Poly' and 'sigmoid' kernels can also capture non-linearities but might require more tuning of their hyperparameters (e.g., degree for 'poly').")
print("A higher R2 score indicates a better fit, while lower MAE and MSE values indicate more accurate predictions.")
```

Experiment-11: Design and train an Artificial Neural Network using the Backpropagation algorithm on a suitable dataset. Also Analyze the effect of network architecture, activation functions, and learning rate on ANN performance. [MNIST (digits) or Fashion-MNIST Dataset]

Sample Code:

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

print("--- Artificial Neural Network (ANN) on MNIST Dataset ---")

# 1. Load the MNIST dataset
```

```
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()

# 2. Preprocess the data
# Normalize pixel values to be between 0 and 1
x_train = x_train.astype("float32") / 255
x_test = x_test.astype("float32") / 255

# Flatten images: MNIST images are 28x28, so flatten to 784
# The input layer will expect this flattened size
num_pixels = x_train.shape[1] * x_train.shape[2]
x_train_flattened = x_train.reshape(-1, num_pixels)
x_test_flattened = x_test.reshape(-1, num_pixels)

# One-hot encode the target labels (10 classes for digits 0-9)
num_classes = 10
y_train_encoded = keras.utils.to_categorical(y_train, num_classes)
y_test_encoded = keras.utils.to_categorical(y_test, num_classes)

print(f"\nShape of x_train_flattened: {x_train_flattened.shape}")
print(f"Shape of y_train_encoded: {y_train_encoded.shape}")
print(f"Shape of x_test_flattened: {x_test_flattened.shape}")
print(f"Shape of y_test_encoded: {y_test_encoded.shape}")

# 3. Function to build an ANN model with customizable architecture and activation
def build_ann_model(input_shape, num_classes, hidden_layers_config, activation='relu'):
    model = keras.Sequential()
    model.add(keras.Input(shape=input_shape))

    for neurons in hidden_layers_config:
        model.add(layers.Dense(neurons, activation=activation))

    # Output layer
    model.add(layers.Dense(num_classes, activation='softmax'))
    return model

# Parameters for experimentation
input_shape = (num_pixels,)

# --- Experiment 1: Network Architecture (Number of Layers and Neurons) ---
architecture_experiments = [
    {'name': '1-Hidden-Layer (128 neurons)', 'config': [128], 'activation': 'relu'},
    {'name': '2-Hidden-Layers (128, 64 neurons)', 'config': [128, 64], 'activation': 'relu'},
    {'name': '3-Hidden-Layers (256, 128, 64 neurons)', 'config': [256, 128, 64], 'activation': 'relu'},
]

# --- Experiment 2: Activation Functions (using a fixed architecture) ---
```

```
activation_experiments = [
    {'name': 'Activation: ReLU', 'activation': 'relu'},
    {'name': 'Activation: Tanh', 'activation': 'tanh'},
    {'name': 'Activation: Sigmoid', 'activation': 'sigmoid'},
]
# Using a common architecture for activation function comparison
fixed_architecture = [128, 64]

# --- Experiment 3: Learning Rate (using a fixed architecture and activation) ---
learning_rate_experiments = [
    {'name': 'LR: 0.1', 'learning_rate': 0.1},
    {'name': 'LR: 0.01', 'learning_rate': 0.01},
    {'name': 'LR: 0.001', 'learning_rate': 0.001},
    {'name': 'LR: 0.0001', 'learning_rate': 0.0001},
]
# Using a common architecture and activation for learning rate comparison
fixed_lr_architecture = [128, 64]
fixed_lr_activation = 'relu'

results = []
epochs = 10 # Number of epochs for training each model variant
batch_size = 128

print(f"\n--- Starting Architecture Experiments (Epochs: {epochs}) ---")
for exp in architecture_experiments:
    print(f"\nTraining Model: {exp['name']}")
            model = build_ann_model(input_shape, num_classes, exp['config'],
activation=exp['activation'])

    # Compile with a default learning rate for architecture comparison
    optimizer = keras.optimizers.Adam(learning_rate=0.001)
    model.compile(optimizer=optimizer, loss='categorical_crossentropy', metrics=['accuracy'])

            history = model.fit(x_train_flattened, y_train_encoded, epochs=epochs,
batch_size=batch_size, verbose=0, validation_split=0.2)
    loss, accuracy = model.evaluate(x_test_flattened, y_test_encoded, verbose=0)
    results.append({
        'experiment_type': 'architecture',
        'name': exp['name'],
        'architecture': exp['config'],
        'activation': exp['activation'],
        'learning_rate': 0.001,
        'test_accuracy': accuracy,
        'test_loss': loss,
        'history': history
    })
```

```
    print(f"  Test Accuracy: {accuracy:.4f}")

print(f"\n--- Starting Activation Function Experiments (Architecture: {fixed_architecture}, Epochs: {epochs}) ---")
for exp in activation_experiments:
    print(f"\nTraining Model: {exp['name']}")
            model = build_ann_model(input_shape, num_classes, fixed_architecture, activation=exp['activation'])

    # Compile with a default learning rate for activation comparison
    optimizer = keras.optimizers.Adam(learning_rate=0.001)
    model.compile(optimizer=optimizer, loss='categorical_crossentropy', metrics=['accuracy'])

            history = model.fit(x_train_flattened, y_train_encoded, epochs=epochs, batch_size=batch_size, verbose=0, validation_split=0.2)
    loss, accuracy = model.evaluate(x_test_flattened, y_test_encoded, verbose=0)
    results.append({
        'experiment_type': 'activation',
        'name': exp['name'],
        'architecture': fixed_architecture,
        'activation': exp['activation'],
        'learning_rate': 0.001,
        'test_accuracy': accuracy,
        'test_loss': loss,
        'history': history
    })
    print(f"  Test Accuracy: {accuracy:.4f}")

print(f"\n--- Starting Learning Rate Experiments (Architecture: {fixed_lr_architecture}, Activation: {fixed_lr_activation}, Epochs: {epochs}) ---")
for exp in learning_rate_experiments:
    print(f"\nTraining Model: {exp['name']}")
            model = build_ann_model(input_shape, num_classes, fixed_lr_architecture, activation=fixed_lr_activation)

    optimizer = keras.optimizers.Adam(learning_rate=exp['learning_rate'])
    model.compile(optimizer=optimizer, loss='categorical_crossentropy', metrics=['accuracy'])

            history = model.fit(x_train_flattened, y_train_encoded, epochs=epochs, batch_size=batch_size, verbose=0, validation_split=0.2)
    loss, accuracy = model.evaluate(x_test_flattened, y_test_encoded, verbose=0)
    results.append({
        'experiment_type': 'learning_rate',
        'name': exp['name'],
        'architecture': fixed_lr_architecture,
        'activation': fixed_lr_activation,
```

```
        'learning_rate': exp['learning_rate'],
        'test_accuracy': accuracy,
        'test_loss': loss,
        'history': history
    })
    print(f"  Test Accuracy: {accuracy:.4f}")

# --- Analysis and Visualization of Results ---

print("\n--- Analysis of Experiment Results ---")

# Architecture Experiment Analysis
print("\nArchitecture Experiment Summary:")
arch_results = [r for r in results if r['experiment_type'] == 'architecture']
for r in arch_results:
    print(f"  {r['name']}: Test Accuracy = {r['test_accuracy']:.4f}")

# Plotting Architecture results
plt.figure(figsize=(10, 6))
plt.bar([r['name'] for r in arch_results], [r['test_accuracy'] for r in arch_results], color='skyblue')
plt.xlabel('Network Architecture')
plt.ylabel('Test Accuracy')
plt.title('Effect of Network Architecture on ANN Performance')
plt.ylim(0.9, 1.0)
plt.xticks(rotation=45, ha='right')
plt.tight_layout()
plt.show()

# Activation Function Experiment Analysis
print("\nActivation Function Experiment Summary:")
act_results = [r for r in results if r['experiment_type'] == 'activation']
for r in act_results:
    print(f"  {r['name']}: Test Accuracy = {r['test_accuracy']:.4f}")

# Plotting Activation Function results
plt.figure(figsize=(10, 6))
plt.bar([r['name'] for r in act_results], [r['test_accuracy'] for r in act_results], color='lightcoral')
plt.xlabel('Activation Function')
plt.ylabel('Test Accuracy')
plt.title('Effect of Activation Function on ANN Performance')
plt.ylim(0.9, 1.0)
plt.xticks(rotation=45, ha='right')
plt.tight_layout()
plt.show()

# Learning Rate Experiment Analysis
```

```
print("\nLearning Rate Experiment Summary:")
lr_results = [r for r in results if r['experiment_type'] == 'learning_rate']
for r in lr_results:
    print(f"  {r['name']}: Test Accuracy = {r['test_accuracy']:.4f}")

# Plotting Learning Rate results
plt.figure(figsize=(10, 6))
plt.plot([str(r['learning_rate']) for r in lr_results], [r['test_accuracy'] for r in lr_results], marker='o', linestyle='-', color='mediumseagreen')
plt.xscale('log') # Log scale for learning rate for better visualization
plt.xlabel('Learning Rate (log scale)')
plt.ylabel('Test Accuracy')
plt.title('Effect of Learning Rate on ANN Performance')
plt.ylim(0.9, 1.0)
plt.grid(True, which="both", ls="--", c='0.7')
plt.tight_layout()
plt.show()

print("\n--- Conclusion ---")
print("From the experiments, we can observe how different architectural choices (number of layers/neurons), activation functions, and learning rates impact the final test accuracy of the ANN. \n")
print("- **Network Architecture**: Deeper or wider networks don't always guarantee better performance and might require more epochs to converge. Too simple networks might underfit.\n")
print("- **Activation Functions**: ReLU is often a good default choice for hidden layers due to its ability to mitigate vanishing gradients. Sigmoid can suffer from vanishing gradients, especially in deep networks. Tanh is an alternative that is zero-centered.\n")
print("- **Learning Rate**: A well-tuned learning rate is crucial. Too high can lead to unstable training and divergence, while too low can lead to very slow convergence. An optimal learning rate allows the model to converge efficiently to a good minimum.\n")
print("These experiments highlight the importance of hyperparameter tuning in achieving optimal ANN performance.")

# ### Training Progress Analysis

# Function to plot training history
def plot_history(history, title):
    plt.figure(figsize=(12, 5))

    # Plot training & validation accuracy values
    plt.subplot(1, 2, 1)
    plt.plot(history.history['accuracy'])
    plt.plot(history.history['val_accuracy'])
    plt.title(f'{title} - Model Accuracy')
    plt.ylabel('Accuracy')
```

```
    plt.xlabel('Epoch')
    plt.legend(['Train', 'Validation'], loc='upper left')
    plt.grid(True)

    # Plot training & validation loss values
    plt.subplot(1, 2, 2)
    plt.plot(history.history['loss'])
    plt.plot(history.history['val_loss'])
    plt.title(f'{title} - Model Loss')
    plt.ylabel('Loss')
    plt.xlabel('Epoch')
    plt.legend(['Train', 'Validation'], loc='upper left')
    plt.grid(True)

    plt.tight_layout()
    plt.show()

print("\n--- Visualizing Training History for Architecture Experiments ---")
for r in arch_results:
    plot_history(r['history'], f"Architecture: {r['name']}")

print("\n--- Visualizing Training History for Activation Function Experiments ---")
for r in act_results:
    plot_history(r['history'], f"Activation: {r['name']}")

print("\n--- Visualizing Training History for Learning Rate Experiments ---")
for r in lr_results:
    plot_history(r['history'], f"Learning Rate: {r['name']}")
```

Experiment-12: Write a program to demonstrate the working of the decision tree based ID3 algorithm. Use an appropriate data set for building the decision tree and apply this knowledge to classify a new sample. [Play Tennis or Iris Dataset]

Sample Code:

```
import pandas as pd
import numpy as np
from collections import Counter
from sklearn.datasets import load_iris

print("--- ID3 Decision Tree Algorithm Demonstration on Iris Dataset ---")

# 1. Load the Iris dataset
iris = load_iris()
```

```
X = iris.data
y = iris.target
feature_names = iris.feature_names
target_names = iris.target_names

# Create a DataFrame
iris_df = pd.DataFrame(data=X, columns=feature_names)
iris_df['species'] = y

print("\n--- Original Iris Dataset (first 5 rows) ---")
print(iris_df.head())

# 2. Discretize numerical features for ID3
# ID3 works with categorical data, so we'll convert numerical features into bins.

# For simplicity, we'll use 3 bins for each feature: 'low', 'medium', 'high'
# You could adjust bin edges for more nuanced categorization.
for col in feature_names:
    iris_df[col] = pd.cut(iris_df[col], bins=3, labels=['low', 'medium', 'high'])

# Convert target numerical labels to actual species names
iris_df['species'] = iris_df['species'].map(lambda x: target_names[x])

print("\n--- Discretized Iris Dataset (first 5 rows) ---")
print(iris_df.head())

# 3. Implement Entropy Calculation
def calculate_entropy(target_column):
    counts = Counter(target_column)
    total_count = len(target_column)

    entropy = 0
    for count in counts.values():
        probability = count / total_count
        if probability > 0: # Avoid log(0)
            entropy -= probability * np.log2(probability)
    return entropy

# 4. Implement Information Gain Calculation
def calculate_information_gain(data, feature, target):
    total_entropy = calculate_entropy(data[target])

    weighted_entropy = 0
    for value in data[feature].unique():
        subset = data[data[feature] == value]
        subset_entropy = calculate_entropy(subset[target])
```

```
        weighted_entropy += (len(subset) / len(data)) * subset_entropy

    information_gain = total_entropy - weighted_entropy
    return information_gain

# 5. Implement a simplified ID3 Decision Tree Builder
def build_tree(data, features, target_attribute_name):
    target_values = data[target_attribute_name].unique()

    if len(target_values) == 1:
        return target_values[0]

    if len(features) == 0 or data.empty:
        return data[target_attribute_name].mode()[0] if not data.empty else None

    gains = {feature: calculate_information_gain(data, feature, target_attribute_name) for feature
in features}
    # Handle cases where all gains might be 0 (e.g., no predictive features left or pure subset)
    if all(value == 0 for value in gains.values()):
        return data[target_attribute_name].mode()[0]

    best_feature = max(gains, key=gains.get)

    tree = {best_feature: {}}

    remaining_features = [f for f in features if f != best_feature]

    for value in data[best_feature].unique():
        subset = data[data[best_feature] == value]
        if not subset.empty:
          tree[best_feature][value] = build_tree(subset, remaining_features, target_attribute_name)
        else:
            # If subset is empty, assign the most common target from the parent node
            tree[best_feature][value] = data[target_attribute_name].mode()[0]

    return tree

# Build the Decision Tree
features_for_tree = [col for col in iris_df.columns if col != 'species']
id3_tree = build_tree(iris_df, features_for_tree, 'species')

print("\n--- Constructed ID3 Decision Tree ---")
import json

def convert_numpy_types(obj):
    """Recursively converts numpy types (like np.str_) to native Python types."""
```

```
    if isinstance(obj, np.generic): # Covers many numpy scalar types
        return obj.item()
    if isinstance(obj, dict):
        return {convert_numpy_types(k): convert_numpy_types(v) for k, v in obj.items()}
    if isinstance(obj, list):
        return [convert_numpy_types(elem) for elem in obj]
    return obj

# Convert the tree to be fully JSON serializable
serializable_tree = convert_numpy_types(id3_tree)
print(json.dumps(serializable_tree, indent=2))

# 6. Implement Classification for a New Sample
def classify_sample(sample, tree):
    if not isinstance(tree, dict):
        return tree

    feature = list(tree.keys())[0]
    value = sample[feature]

    if value in tree[feature]:
        next_level = tree[feature][value]
        return classify_sample(sample, next_level)
    else:
        # If a path for the value doesn't exist, this implies an unseen category
        # In practice, one might return the majority class of the parent node or 'Unknown'
        return "Unknown_Category_Path"

# 7. Apply to a new sample (must have discretized features)
# Example new samples, manually discretized to match the training data format
new_sample_1 = {
    'sepal length (cm)': 'low',
    'sepal width (cm)': 'medium',
    'petal length (cm)': 'low',
    'petal width (cm)': 'low'
}

new_sample_2 = {
    'sepal length (cm)': 'medium',
    'sepal width (cm)': 'low',
    'petal length (cm)': 'medium',
    'petal width (cm)': 'medium'
}

new_sample_3 = {
    'sepal length (cm)': 'high',
```

```
    'sepal width (cm)': 'medium',
    'petal length (cm)': 'high',
    'petal width (cm)': 'high'
}

print(f"\n--- Classifying New Sample 1: {new_sample_1} ---")
prediction_1 = classify_sample(new_sample_1, id3_tree)
print(f"Prediction for new sample 1: {prediction_1}")

print(f"\n--- Classifying New Sample 2: {new_sample_2} ---")
prediction_2 = classify_sample(new_sample_2, id3_tree)
print(f"Prediction for new sample 2: {prediction_2}")

print(f"\n--- Classifying New Sample 3: {new_sample_3} ---")
prediction_3 = classify_sample(new_sample_3, id3_tree)
print(f"Prediction for new sample 3: {prediction_3}")



#### visualize the ID3 decision tree using Graphviz

import graphviz

def add_nodes_edges(tree, dot, parent_name=None, edge_label=None):
    """Recursively adds nodes and edges to a Graphviz dot object."""
    if isinstance(tree, dict):
        feature_name = list(tree.keys())[0]
        current_node_name = f'node_{id(tree)}'
                    dot.node(current_node_name,     feature_name,     shape='box',     style='filled',
fillcolor='lightblue')

        if parent_name is not None:
            dot.edge(parent_name, current_node_name, label=str(edge_label))

        for value, subtree in tree[feature_name].items():
            add_nodes_edges(subtree, dot, current_node_name, value)
    else:
        # Leaf node
        leaf_node_name = f'leaf_{id(tree)}'
        dot.node(leaf_node_name, str(tree), shape='ellipse', style='filled', fillcolor='lightgreen')

        if parent_name is not None:
            dot.edge(parent_name, leaf_node_name, label=str(edge_label))

def visualize_tree(tree):
    """Creates and displays a Graphviz visualization of the decision tree."""
```

```
    dot = graphviz.Digraph(comment='ID3 Decision Tree', graph_attr={'rankdir': 'TB'})
    add_nodes_edges(tree, dot)
    return dot

print("\n--- Visualizing the Decision Tree with Graphviz ---")
# Ensure id3_tree is converted to be fully serializable before passing to visualize_tree
# (using the convert_numpy_types function defined earlier)
serializable_tree = convert_numpy_types(id3_tree)

# Create and render the graph
graph = visualize_tree(serializable_tree)
graph.render('id3_iris_tree', view=True, format='png', cleanup=True)
display(graph)
```

Experiment-13: Write a program to demonstrate the working of the Random Forest algorithm. Use an appropriate data set for classifying a new sample. [Wine or Breast Cancer Dataset]

Sample Code:

```
import numpy as np
import pandas as pd
from sklearn.datasets import load_wine, load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
import seaborn as sns

print("--- Random Forest Classifier Demonstration ---")

# --- Part 1: Wine Dataset ---
print("\n--- Using Wine Dataset ---")

# 1. Load the Wine dataset
wine = load_wine()
X_wine = wine.data
y_wine = wine.target
feature_names_wine = wine.feature_names
target_names_wine = wine.target_names

print(f"Number of samples: {X_wine.shape[0]}")
print(f"Number of features: {X_wine.shape[1]}")
print(f"Target classes: {target_names_wine}")
```

```
# 2. Split data into training and testing sets
X_train_wine, X_test_wine, y_train_wine, y_test_wine = train_test_split(X_wine, y_wine, test_size=0.3, random_state=42, stratify=y_wine)

# 3. Scale features (optional but good practice for many ML models)
scaler_wine = StandardScaler()
X_train_wine_scaled = scaler_wine.fit_transform(X_train_wine)
X_test_wine_scaled = scaler_wine.transform(X_test_wine)

# 4. Initialize and train the Random Forest Classifier
# n_estimators: number of trees in the forest
# random_state: for reproducibility
print("\nTraining Random Forest Classifier on Wine dataset...")
rf_classifier_wine = RandomForestClassifier(n_estimators=100, random_state=42)
rf_classifier_wine.fit(X_train_wine_scaled, y_train_wine)

# 5. Make predictions and evaluate the model
y_pred_wine = rf_classifier_wine.predict(X_test_wine_scaled)

accuracy_wine = accuracy_score(y_test_wine, y_pred_wine)
report_wine = classification_report(y_test_wine, y_pred_wine, target_names=target_names_wine)
cm_wine = confusion_matrix(y_test_wine, y_pred_wine)

print(f"\nWine Dataset - Model Evaluation:")
print(f"  Accuracy: {accuracy_wine:.4f}")
print("  Classification Report:\n", report_wine)
print("  Confusion Matrix:\n", cm_wine)

# Plot Confusion Matrix for Wine
plt.figure(figsize=(8, 6))
sns.heatmap(cm_wine, annot=True, fmt='d', cmap='Blues',
            xticklabels=target_names_wine, yticklabels=target_names_wine)
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Confusion Matrix - Wine Dataset')
plt.show()

# 6. Classify a new sample for Wine dataset
# Let's create a synthetic new sample (it needs to be scaled like the training data)
# For demonstration, we'll take the mean of the first class as a 'new sample'
new_sample_wine_raw = X_wine[y_wine == 0].mean(axis=0)
new_sample_wine_scaled = scaler_wine.transform(new_sample_wine_raw.reshape(1, -1))

predicted_class_wine = rf_classifier_wine.predict(new_sample_wine_scaled)[0]
```

```
predicted_class_name_wine = target_names_wine[predicted_class_wine]

print(f"\nClassifying a new sample (mean of class 0): {new_sample_wine_raw}")
print(f"Predicted class for new Wine sample: {predicted_class_name_wine}")

# --- Part 2: Breast Cancer Dataset ---
print("\n" + "="*60 + "\n")
print("--- Using Breast Cancer Dataset ---")

# 1. Load the Breast Cancer dataset
cancer = load_breast_cancer()
X_cancer = cancer.data
y_cancer = cancer.target
feature_names_cancer = cancer.feature_names
target_names_cancer = cancer.target_names # 0: malignant, 1: benign

print(f"Number of samples: {X_cancer.shape[0]}")
print(f"Number of features: {X_cancer.shape[1]}")
print(f"Target classes: {target_names_cancer}")

# 2. Split data into training and testing sets
X_train_cancer, X_test_cancer, y_train_cancer, y_test_cancer = train_test_split(X_cancer, y_cancer, test_size=0.3, random_state=42, stratify=y_cancer)

# 3. Scale features
scaler_cancer = StandardScaler()
X_train_cancer_scaled = scaler_cancer.fit_transform(X_train_cancer)
X_test_cancer_scaled = scaler_cancer.transform(X_test_cancer)

# 4. Initialize and train the Random Forest Classifier
print("\nTraining Random Forest Classifier on Breast Cancer dataset...")
rf_classifier_cancer = RandomForestClassifier(n_estimators=100, random_state=42)
rf_classifier_cancer.fit(X_train_cancer_scaled, y_train_cancer)

# 5. Make predictions and evaluate the model
y_pred_cancer = rf_classifier_cancer.predict(X_test_cancer_scaled)

accuracy_cancer = accuracy_score(y_test_cancer, y_pred_cancer)
report_cancer = classification_report(y_test_cancer, y_pred_cancer, target_names=target_names_cancer)
cm_cancer = confusion_matrix(y_test_cancer, y_pred_cancer)

print(f"\nBreast Cancer Dataset - Model Evaluation:")
print(f"  Accuracy: {accuracy_cancer:.4f}")
print("  Classification Report:\n", report_cancer)
print("  Confusion Matrix:\n", cm_cancer)
```

```
# Plot Confusion Matrix for Breast Cancer
plt.figure(figsize=(8, 6))
sns.heatmap(cm_cancer, annot=True, fmt='d', cmap='Blues',
            xticklabels=target_names_cancer, yticklabels=target_names_cancer)
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Confusion Matrix - Breast Cancer Dataset')
plt.show()

# 6. Classify a new sample for Breast Cancer dataset
# For demonstration, take the mean of the 'malignant' class (0) as a 'new sample'
new_sample_cancer_raw = X_cancer[y_cancer == 0].mean(axis=0)
new_sample_cancer_scaled = scaler_cancer.transform(new_sample_cancer_raw.reshape(1, -1))

predicted_class_cancer = rf_classifier_cancer.predict(new_sample_cancer_scaled)[0]
predicted_class_name_cancer = target_names_cancer[predicted_class_cancer]

print(f"\nClassifying a new sample (mean of malignant class): {new_sample_cancer_raw}")
print(f"Predicted class for new Breast Cancer sample: {predicted_class_name_cancer}")

print("\n--- Random Forest Demonstration Complete ---")
```

Experiment-14: Write a program to implement K-Means clustering algorithm using an appropriate dataset. [Iris or Mall Customers]

Sample Code:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.datasets import load_iris
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import KMeans

print("--- K-Means Clustering Algorithm Demonstration on Iris Dataset ---")

# 1. Load the Iris dataset
iris = load_iris()
X = iris.data
y = iris.target # We'll use this later for comparison, but K-Means is unsupervised
feature_names = iris.feature_names
target_names = iris.target_names
```

```
print(f"\nNumber of samples: {X.shape[0]}")
print(f"Number of features: {X.shape[1]}")
print(f"Actual target classes: {target_names}")

# Create a DataFrame for better inspection
iris_df = pd.DataFrame(data=X, columns=feature_names)
iris_df['target'] = y

print("\n--- Original Iris Dataset (first 5 rows) ---")
print(iris_df.head())

# 2. Preprocessing: Scale the features
# Scaling is crucial for K-Means as it is distance-based.
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

print("\n--- Scaled Iris Dataset (first 5 rows) ---")
print(pd.DataFrame(X_scaled, columns=feature_names).head())

# 3. Determine the optimal number of clusters (k) using the Elbow Method
# The inertia (sum of squared distances of samples to their closest cluster center) is plotted.
# The 'elbow' point in the graph suggests the optimal k.
wcss = [] # Within-cluster sum of squares
max_k = 10

print(f"\n--- Performing Elbow Method to find optimal k (1 to {max_k-1} clusters) ---")
for i in range(1, max_k):
        kmeans = KMeans(n_clusters=i, init='k-means++', max_iter=300, n_init=10,
random_state=42)
   kmeans.fit(X_scaled)
   wcss.append(kmeans.inertia_)

# Plotting the Elbow Method results
plt.figure(figsize=(10, 6))
plt.plot(range(1, max_k), wcss, marker='o', linestyle='--')
plt.title('Elbow Method for Optimal K')
plt.xlabel('Number of Clusters (K)')
plt.ylabel('Within-Cluster Sum of Squares (WCSS/Inertia)')
plt.xticks(range(1, max_k))
plt.grid(True)
plt.show()

print("Based on the Elbow Method, we can observe a significant bend (elbow) around K=3. This
suggests that 3 is an optimal number of clusters, which aligns with the known 3 species in the
Iris dataset.")
```

```
optimal_k = 3 # Choose k=3 based on the Elbow Method (or observation of Iris data)

# 4. Apply K-Means clustering with the chosen 'k'
print(f"\n--- Applying K-Means with optimal K = {optimal_k} ---")
kmeans_model = KMeans(n_clusters=optimal_k, init='k-means++', max_iter=300, n_init=10,
random_state=42)
clusters = kmeans_model.fit_predict(X_scaled)

# Add the cluster labels to the DataFrame
iris_df['cluster'] = clusters

print("\n--- Iris Dataset with K-Means Cluster Labels (first 5 rows) ---")
print(iris_df.head())

# 5. Visualize the clusters
# For visualization, we'll use two principal components if the feature count is > 2
# or directly use two features if the count is <= 2.

if X_scaled.shape[1] > 2:
    from sklearn.decomposition import PCA
    pca = PCA(n_components=2)
    X_pca = pca.fit_transform(X_scaled)
        vis_df = pd.DataFrame(data=X_pca, columns=['Principal Component 1', 'Principal
Component 2'])
    x_axis = 'Principal Component 1'
    y_axis = 'Principal Component 2'
else:
    vis_df = pd.DataFrame(data=X_scaled, columns=feature_names)
    x_axis = feature_names[0]
    y_axis = feature_names[1]

vis_df['cluster'] = clusters
vis_df['actual_target'] = iris_df['target'].map(lambda x: target_names[x]) # For comparison

plt.figure(figsize=(14, 6))

plt.subplot(1, 2, 1)
sns.scatterplot(x=x_axis, y=y_axis, hue='cluster', data=vis_df, palette='viridis', s=100,
alpha=0.8)
plt.title(f'K-Means Clusters (K={optimal_k})')
plt.xlabel(x_axis)
plt.ylabel(y_axis)
plt.grid(True)

plt.subplot(1, 2, 2)
```

```
sns.scatterplot(x=x_axis, y=y_axis, hue='actual_target', data=vis_df, palette='viridis', s=100, alpha=0.8)
plt.title('Actual Iris Species')
plt.xlabel(x_axis)
plt.ylabel(y_axis)
plt.grid(True)

plt.tight_layout()
plt.show()

print("\n--- K-Means Clustering Demonstration Complete ---")
print("The scatter plots above visualize the clusters found by K-Means compared to the actual species. You can observe how well the unsupervised K-Means algorithm was able to separate the different species based solely on their features.")

# Evaluation (optional, for comparing with ground truth)
from sklearn.metrics import adjusted_rand_score, silhouette_score

# Map clusters to original labels (this is an approximation as cluster labels are arbitrary)
# One way to do this is to find the most frequent true label in each cluster
cluster_to_label = {}
for cluster_id in np.unique(clusters):
    mask = (clusters == cluster_id)
    true_labels_in_cluster = y[mask]
    most_common_label = Counter(true_labels_in_cluster).most_common(1)[0][0]
    cluster_to_label[cluster_id] = most_common_label

predicted_labels_mapped = np.array([cluster_to_label[c] for c in clusters])

print(f"\n--- K-Means Performance Metrics (compared to true labels) ---")
print(f"Adjusted Rand Score: {adjusted_rand_score(y, predicted_labels_mapped):.4f}")
print(f"Silhouette Score: {silhouette_score(X_scaled, clusters):.4f}")
print("The Adjusted Rand Score measures the similarity between the true and predicted cluster assignments, correcting for chance. A score of 1.0 indicates perfect agreement.")
print("The Silhouette Score measures how similar an object is to its own cluster compared to other clusters. Higher values indicate better-defined clusters (range from -1 to 1).")
```

Experiment-15: Write a program to implement data split into training, cross validation and testing data. Evaluate model performance using Hold-Out Validation and k-Fold Cross Validation techniques. [Iris Dataset]

Sample Code:

```
import numpy as np
```

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split, StratifiedKFold
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, classification_report

print("--- Data Splitting and Model Evaluation Demonstration (Iris Dataset) ---")

# 1. Load the Iris dataset
iris = load_iris()
X = iris.data
y = iris.target
feature_names = iris.feature_names
target_names = iris.target_names

print(f"\nDataset shape (features): {X.shape}")
print(f"Dataset shape (target): {y.shape}")
print(f"Target classes: {target_names}")

# 2. Initial Train-Test Split
# We'll reserve a portion of the data (e.g., 20%) for final, unbiased testing.
# The remaining 80% will be used for training and validation (Hold-Out or K-Fold).

X_train_val, X_test, y_train_val, y_test = train_test_split(X, y, test_size=0.2, random_state=42,
stratify=y)

print(f"\nInitial split: Train/Validation set size: {X_train_val.shape[0]}, Test set size:
{X_test.shape[0]}")

# 3. Feature Scaling
# Scale the features based on the training/validation set. Apply the same scaling to the test set.
scaler = StandardScaler()
X_train_val_scaled = scaler.fit_transform(X_train_val)
X_test_scaled = scaler.transform(X_test)

print("Features scaled using StandardScaler (fitted on Train/Validation set).")

# Initialize a model (Logistic Regression for classification)
model = LogisticRegression(random_state=42, solver='liblinear')

# --- Hold-Out Validation ---
print("\n" + "="*60 + "\n--- Hold-Out Validation ---")
```

```
# Further split the train_val set into training and validation sets for hold-out
# e.g., 75% for training (from X_train_val), 25% for validation (from X_train_val)
X_train_ho, X_val_ho, y_train_ho, y_val_ho = train_test_split(
    X_train_val_scaled, y_train_val, test_size=0.25, random_state=42, stratify=y_train_val
)

print(f"Hold-Out Split: Training set size: {X_train_ho.shape[0]}, Validation set size: {X_val_ho.shape[0]}")

# Train model on Hold-Out Training set
model.fit(X_train_ho, y_train_ho)

# Evaluate on Hold-Out Validation set
y_pred_ho_val = model.predict(X_val_ho)
ho_val_accuracy = accuracy_score(y_val_ho, y_pred_ho_val)
print(f"Hold-Out Validation Accuracy: {ho_val_accuracy:.4f}")
print("Hold-Out Validation Classification Report:\n", classification_report(y_val_ho, y_pred_ho_val, target_names=target_names))

# --- k-Fold Cross-Validation ---
print("\n" + "="*60 + "\n--- k-Fold Cross-Validation ---")

n_splits = 5 # Number of folds
skf = StratifiedKFold(n_splits=n_splits, shuffle=True, random_state=42)

kfold_accuracies = []

print(f"{n_splits}-Fold Cross-Validation on the Train/Validation set (size: {X_train_val_scaled.shape[0]})\n")

# Iterate over each fold
for fold, (train_index, val_index) in enumerate(skf.split(X_train_val_scaled, y_train_val)):
    X_train_kfold, X_val_kfold = X_train_val_scaled[train_index], X_train_val_scaled[val_index]
    y_train_kfold, y_val_kfold = y_train_val[train_index], y_train_val[val_index]

    # Train a new model for each fold
    fold_model = LogisticRegression(random_state=42, solver='liblinear')
    fold_model.fit(X_train_kfold, y_train_kfold)

    # Evaluate on the validation part of the fold
    y_pred_kfold_val = fold_model.predict(X_val_kfold)
    fold_accuracy = accuracy_score(y_val_kfold, y_pred_kfold_val)
    kfold_accuracies.append(fold_accuracy)

    print(f"  Fold {fold+1} Accuracy: {fold_accuracy:.4f}")
```

```
mean_kfold_accuracy = np.mean(kfold_accuracies)
std_kfold_accuracy = np.std(kfold_accuracies)

print(f"\nAverage k-Fold CV Accuracy: {mean_kfold_accuracy:.4f} (+/- {std_kfold_accuracy:.4f})")

# --- Final Model Training and Evaluation on Test Set ---
print("\n" + "="*60 + "\n--- Final Model Evaluation on Unseen Test Set ---")

# Train the final model on the entire X_train_val_scaled dataset
final_model = LogisticRegression(random_state=42, solver='liblinear')
final_model.fit(X_train_val_scaled, y_train_val)

# Evaluate the final model on the completely unseen X_test_scaled data
y_pred_final_test = final_model.predict(X_test_scaled)
final_test_accuracy = accuracy_score(y_test, y_pred_final_test)

print(f"Final Model Test Accuracy: {final_test_accuracy:.4f}")
print("Final Model Test Classification Report:\n", classification_report(y_test, y_pred_final_test, target_names=target_names))

# --- Summary of Results ---
print("\n" + "="*60 + "\n--- Summary of Validation Results ---")
print(f"Hold-Out Validation Accuracy: {ho_val_accuracy:.4f}")
print(f"Average {n_splits}-Fold CV Accuracy: {mean_kfold_accuracy:.4f} (+/- {std_kfold_accuracy:.4f})")
print(f"Final Model Test Accuracy: {final_test_accuracy:.4f}")

# Visualization of results
metrics = ['Hold-Out Validation', f'{n_splits}-Fold CV (Mean)', 'Final Test']
accuracies = [ho_val_accuracy, mean_kfold_accuracy, final_test_accuracy]

plt.figure(figsize=(10, 6))
sns.barplot(x=metrics, y=accuracies, palette='viridis')
plt.ylim(min(accuracies) - 0.1, 1.05) # Adjusted y-axis limits
plt.ylabel('Accuracy')
plt.title('Comparison of Model Performance with Different Validation Techniques')
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.show()

print("\n--- Demonstration Complete ---")
```

Experiment-16: Implement an Ensemble Learning approach [Bagging, Boosting, or Stacking] by combining different models to solve time series-based prediction problem. [California Housing or UCI Bike Sharing Demand Dataset]

Sample Code:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error

print("--- Bagging Ensemble (Random Forest Regressor) on California Housing Dataset ---")

# 1. Load the California Housing dataset
housing = fetch_california_housing(as_frame=True)
X = housing.data
y = housing.target

print(f"\nDataset shape (features): {X.shape}")
print(f"Dataset shape (target): {y.shape}")
print("First 5 rows of features:\n", X.head())
print("First 5 rows of target:\n", y.head())

# 2. Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

print(f"\nTraining samples: {len(X_train)}")
print(f"Testing samples: {len(X_test)}")

# 3. Scale the features
# Scaling is important for many models, though less critical for tree-based models like Random Forest,
# it's good practice, especially if combining with other models later.
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Convert scaled arrays back to DataFrame for feature naming if needed, but for RF it's fine as arrays
# X_train_scaled_df = pd.DataFrame(X_train_scaled, columns=X.columns)
# X_test_scaled_df = pd.DataFrame(X_test_scaled, columns=X.columns)
```

```
# 4. Initialize and train the Random Forest Regressor (Bagging Ensemble)
# n_estimators: The number of trees in the forest.
# random_state: Controls the randomness of the bootstrapping of samples and features.
print("\nTraining Random Forest Regressor...")
rf_regressor = RandomForestRegressor(n_estimators=100, random_state=42, n_jobs=-1) # n_jobs=-1 uses all available CPU cores
rf_regressor.fit(X_train_scaled, y_train)

# 5. Make predictions on the test set
y_pred = rf_regressor.predict(X_test_scaled)

# 6. Evaluate the model
mse = mean_squared_error(y_test, y_pred)
rmse = np.sqrt(mse)
mae = mean_absolute_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print("\n--- Model Evaluation (Random Forest Regressor) ---")
print(f"Mean Squared Error (MSE): {mse:.4f}")
print(f"Root Mean Squared Error (RMSE): {rmse:.4f}")
print(f"Mean Absolute Error (MAE): {mae:.4f}")
print(f"R-squared (R2) Score: {r2:.4f}")

# 7. Visualize Actual vs. Predicted Values
plt.figure(figsize=(10, 6))
sns.scatterplot(x=y_test, y=y_pred, alpha=0.6)
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], 'r--', lw=2, label='Perfect Prediction Line')
plt.xlabel('Actual Median House Value')
plt.ylabel('Predicted Median House Value')
plt.title('Random Forest Regressor: Actual vs. Predicted Values')
plt.legend()
plt.grid(True)
plt.show()

# 8. Feature Importance (Specific to tree-based models like Random Forest)
feature_importances = pd.Series(rf_regressor.feature_importances_, index=X.columns).sort_values(ascending=False)

plt.figure(figsize=(10, 6))
sns.barplot(x=feature_importances.values, y=feature_importances.index, palette='viridis')
plt.title('Feature Importances from Random Forest Regressor')
plt.xlabel('Importance')
plt.ylabel('Feature')
plt.show()
```

```
print("\n--- Random Forest Regressor Demonstration Complete ---")
print("The R2 score indicates how well the model explains the variance in the target variable.")
print("Higher importance values for features suggest they contribute more significantly to the predictions.")
```

Experiment-17: Perform statistical hypothesis testing (t-test, Chi-square test, ANOVA, or Mann-Whitney U test) on an appropriate dataset. Interpret hypothesis testing results and draw statistical conclusions using Python statistical libraries. [seaborn - Tips, seaborn - Titanic, or Iris dataset]

Sample Code:

```
##ANOVA Test

import pandas as pd
import numpy as np
from sklearn.datasets import load_iris
from scipy.stats import f_oneway
import matplotlib.pyplot as plt
import seaborn as sns

print("--- ANOVA Test on Iris Dataset (Sepal Length by Species) ---")

# 1. Load the Iris dataset
iris = load_iris()
iris_df = pd.DataFrame(data=iris.data, columns=iris.feature_names)
iris_df['species'] = iris.target
iris_df['species_name'] = iris_df['species'].map(lambda x: iris.target_names[x])

print("\nFirst 5 rows of Iris Dataset:")
display(iris_df.head())

# 2. Extract 'sepal length (cm)' for each species
sepal_length_setosa = iris_df[iris_df['species_name'] == 'setosa']['sepal length (cm)']
sepal_length_versicolor = iris_df[iris_df['species_name'] == 'versicolor']['sepal length (cm)']
sepal_length_virginica = iris_df[iris_df['species_name'] == 'virginica']['sepal length (cm)']

print(f"\nMean Sepal Length for Setosa: {sepal_length_setosa.mean():.2f}")
print(f"Mean Sepal Length for Versicolor: {sepal_length_versicolor.mean():.2f}")
print(f"Mean Sepal Length for Virginica: {sepal_length_virginica.mean():.2f}")

# 3. Perform ANOVA test
# Null Hypothesis (H0): The means of 'sepal length (cm)' are equal across all three species.
# Alternative Hypothesis (H1): At least one species' mean 'sepal length (cm)' is different from the others.
```

```
f_statistic, p_value = f_oneway(sepal_length_setosa, sepal_length_versicolor, sepal_length_virginica)

print("\n--- ANOVA Test Results ---")
print(f"F-statistic: {f_statistic:.2f}")
print(f"P-value: {p_value:.3e}") # Display p-value in scientific notation for clarity

# 4. Interpret the results
alpha = 0.05
print(f"\nSignificance level (alpha): {alpha}")

if p_value < alpha:
    print("Conclusion: Since the p-value is less than the significance level (alpha),")
    print("we reject the null hypothesis.")
    print("This suggests that there is a statistically significant difference in the mean sepal length (cm) among the three Iris species.")
else:
    print("Conclusion: Since the p-value is greater than the significance level (alpha),")
    print("we fail to reject the null hypothesis.")
    print("This suggests that there is no statistically significant difference in the mean sepal length (cm) among the three Iris species.")

# 5. Visualize the distributions
plt.figure(figsize=(10, 6))
sns.boxplot(x='species_name', y='sepal length (cm)', data=iris_df, palette='viridis')
sns.swarmplot(x='species_name', y='sepal length (cm)', data=iris_df, color='black', alpha=0.5)
plt.title('Distribution of Sepal Length by Iris Species')
plt.xlabel('Iris Species')
plt.ylabel('Sepal Length (cm)')
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.show()

print("\n--- Statistical Conclusion Drawn --- ")
print("The box plot visually supports the ANOVA finding, showing clear differences in the central tendency and spread of sepal length across the species, especially between setosa and the other two.")
```

Chi-square test on Titanic dataset

```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from scipy.stats import chi2_contingency

print("--- Chi-square Test on Titanic Dataset (Sex vs. Survived) ---")
```

```
# 1. Load the Titanic dataset
titanic_df = sns.load_dataset('titanic')

print("\nFirst 5 rows of Titanic Dataset:")
display(titanic_df.head())

# 2. Create a contingency table of 'Sex' and 'Survived'
contingency_table = pd.crosstab(titanic_df['sex'], titanic_df['survived'])

print("\nContingency Table (Sex vs. Survived):")
display(contingency_table)

# 3. Perform Chi-square test
chi2, p_value, dof, expected = chi2_contingency(contingency_table)

print("\n--- Chi-square Test Results ---")
print(f"Chi-square Statistic: {chi2:.2f}")
print(f"P-value: {p_value:.3e}")
print(f"Degrees of Freedom (dof): {dof}")
# print("Expected Frequencies:\n", expected)

# 4. Interpret the results
alpha = 0.05
print(f"\nSignificance level (alpha): {alpha}")

if p_value < alpha:
    print("Conclusion: Since the p-value is less than the significance level (alpha),")
    print("we reject the null hypothesis.")
     print("This suggests that there is a statistically significant association between gender and survival on the Titanic.")
else:
    print("Conclusion: Since the p-value is greater than the significance level (alpha),")
    print("we fail to reject the null hypothesis.")
     print("This suggests that there is no statistically significant association between gender and survival on the Titanic.")

# 5. Visualize the relationship
plt.figure(figsize=(8, 6))
sns.countplot(data=titanic_df, x='sex', hue='survived', palette='viridis')
plt.title('Survival Count by Gender on Titanic')
plt.xlabel('Gender')
plt.ylabel('Number of Passengers')
plt.xticks(ticks=[0, 1], labels=['Male', 'Female']) # Assuming 0 and 1 correspond to categories in 'sex'
plt.legend(title='Survived', labels=['No', 'Yes'])
plt.grid(axis='y', linestyle='--', alpha=0.7)
```

```
plt.show()

print("\n--- Statistical Conclusion Drawn ---")
print("The countplot visually supports the Chi-square test finding, showing a clear difference in survival rates between males and females.")

### t-test on the Iris dataset

import pandas as pd
import numpy as np
from sklearn.datasets import load_iris
from scipy.stats import ttest_ind
import matplotlib.pyplot as plt
import seaborn as sns

print("--- Independent Samples t-test on Iris Dataset ---\n")

# 1. Load the Iris dataset (if not already loaded in the notebook context)
# Using the iris_df created in a previous cell, but re-creating for self-containment.
iris = load_iris()
iris_df = pd.DataFrame(data=iris.data, columns=iris.feature_names)
iris_df['species'] = iris.target
iris_df['species_name'] = iris_df['species'].map(lambda x: iris.target_names[x])

# 2. Extract 'sepal length (cm)' for the two species (setosa and versicolor)
sepal_length_setosa = iris_df[iris_df['species_name'] == 'setosa']['sepal length (cm)']
sepal_length_versicolor = iris_df[iris_df['species_name'] == 'versicolor']['sepal length (cm)']

print(f"Mean Sepal Length for Setosa: {sepal_length_setosa.mean():.2f}")
print(f"Mean Sepal Length for Versicolor: {sepal_length_versicolor.mean():.2f}")

# 3. Perform Independent Samples t-test
# equal_var=True (default) assumes equal population variances. If unequal, set to False.
t_statistic, p_value = ttest_ind(sepal_length_setosa, sepal_length_versicolor, equal_var=True)

print("\n--- t-test Results ---")
print(f"t-statistic: {t_statistic:.2f}")
print(f"P-value: {p_value:.3e}")

# 4. Interpret the results
alpha = 0.05
print(f"\nSignificance level (alpha): {alpha}")

if p_value < alpha:
    print("Conclusion: Since the p-value is less than the significance level (alpha),")
    print("we reject the null hypothesis.")
```

```
    print("This suggests that there is a statistically significant difference in the mean sepal length (cm) between Setosa and Versicolor species.")
else:
    print("Conclusion: Since the p-value is greater than the significance level (alpha),")
    print("we fail to reject the null hypothesis.")
    print("This suggests that there is no statistically significant difference in the mean sepal length (cm) between Setosa and Versicolor species.")

# 5. Visualize the distributions
plt.figure(figsize=(8, 6))
sns.boxplot(x='species_name', y='sepal length (cm)', data=iris_df[iris_df['species_name'].isin(['setosa', 'versicolor'])], palette='pastel')
sns.swarmplot(x='species_name', y='sepal length (cm)', data=iris_df[iris_df['species_name'].isin(['setosa', 'versicolor'])], color='black', alpha=0.5)
plt.title('Distribution of Sepal Length: Setosa vs. Versicolor')
plt.xlabel('Iris Species')
plt.ylabel('Sepal Length (cm)')
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.show()

print("\n--- Statistical Conclusion Drawn ---")
print("The box plot visually confirms the distinct separation and difference in sepal lengths between Setosa and Versicolor species.")


### ### Mann-Whitney U Test on Iris Dataset (Sepal Length between Setosa and Versicolor)

import pandas as pd
import numpy as np
from sklearn.datasets import load_iris
from scipy.stats import mannwhitneyu
import matplotlib.pyplot as plt
import seaborn as sns

print("--- Mann-Whitney U Test on Iris Dataset ---\n")

# 1. Load the Iris dataset (if not already loaded in the notebook context)
# Using the iris_df created in a previous cell, but re-creating for self-containment.
iris = load_iris()
iris_df = pd.DataFrame(data=iris.data, columns=iris.feature_names)
iris_df['species'] = iris.target
iris_df['species_name'] = iris_df['species'].map(lambda x: iris.target_names[x])

# 2. Extract 'sepal length (cm)' for the two species (setosa and versicolor)
sepal_length_setosa = iris_df[iris_df['species_name'] == 'setosa']['sepal length (cm)']
sepal_length_versicolor = iris_df[iris_df['species_name'] == 'versicolor']['sepal length (cm)']
```

```
print(f"Median Sepal Length for Setosa: {sepal_length_setosa.median():.2f}")
print(f"Median Sepal Length for Versicolor: {sepal_length_versicolor.median():.2f}")

# 3. Perform Mann-Whitney U test
statistic, p_value = mannwhitneyu(sepal_length_setosa, sepal_length_versicolor, alternative='two-sided')

print("\n--- Mann-Whitney U Test Results ---")
print(f"Statistic: {statistic:.2f}")
print(f"P-value: {p_value:.3e}")

# 4. Interpret the results
alpha = 0.05
print(f"\nSignificance level (alpha): {alpha}")

if p_value < alpha:
    print("Conclusion: Since the p-value is less than the significance level (alpha),")
    print("we reject the null hypothesis.")
    print("This suggests that there is a statistically significant difference in the distributions of sepal length (cm) between Setosa and Versicolor species.")
else:
    print("Conclusion: Since the p-value is greater than the significance level (alpha),")
    print("we fail to reject the null hypothesis.")
    print("This suggests that there is no statistically significant difference in the distributions of sepal length (cm) between Setosa and Versicolor species.")

# 5. Visualize the distributions (same as t-test for consistency)
plt.figure(figsize=(8, 6))
sns.boxplot(x='species_name', y='sepal length (cm)', data=iris_df[iris_df['species_name'].isin(['setosa', 'versicolor'])], palette='pastel')
sns.swarmplot(x='species_name', y='sepal length (cm)', data=iris_df[iris_df['species_name'].isin(['setosa', 'versicolor'])], color='black', alpha=0.5)
plt.title('Distribution of Sepal Length: Setosa vs. Versicolor')
plt.xlabel('Iris Species')
plt.ylabel('Sepal Length (cm)')
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.show()

print("\n--- Statistical Conclusion Drawn ---")
print("Similar to the t-test, the Mann-Whitney U test confirms a significant difference, indicating that the distributions of sepal length for Setosa and Versicolor are not the same. The visual representation also supports this finding.")
```