Lab Course-I Computer Laboratory

1. Design white box test cases for web application.

Designing **white box test cases** for a web application involves creating tests based on the application's internal code, logic, and structure. These tests ensure that the application's components behave as expected and help identify any potential errors in logic, data flow, or specific code segments.

Here's a step-by-step guide to designing white box test cases for a web application:

1. **Understand the Application**

- Review the application architecture, such as frontend, backend, and database interaction.
- Study the codebase and identify key modules, functions, and workflows.
- Understand the expected functionality and logic behind the code.

2. **Identify Testing Objectives**

White box testing for web applications can cover:

- **Unit Testing:** Test individual components (e.g., functions, classes).
- **Integration Testing:** Test interactions between modules.
- **Path Testing:** Ensure all code paths are executed.
- **Condition Testing:** Verify logical conditions (e.g., `if`, `else`, `switch`).
- **Loop Testing:** Test loops for different scenarios, such as no iteration, single iteration, or maximum iterations.

3. **Create Test Cases**

Each test case should have:

- **Test Case ID:** Unique identifier.
- **Objective:** Purpose of the test case.
- **Preconditions:** Required setup or state before execution.
- **Test Steps:** Actions to execute the test.
- **Expected Result:** The outcome if the code works correctly.
- **Actual Result:** Observed result during testing.
- **Pass/Fail Status:** Comparison of expected vs. actual result.

4. **Example White Box Test Cases**

a) **Unit Testing: Functionality of Login Validation**

- **Objective:** Validate the `validateUser` function.
- **Preconditions:** A database with user credentials.
- b) Path Testing: Navigation Menu

Objective: Test navigation logic in the menu bar.

Preconditions: Menu bar implemented with code handling various states.

c) **Loop Testing: Pagination Component**

- **Objective:** Verify the behavior of the pagination loop.
- **Preconditions:** At least 50 records in the system.

d) **Condition Testing: Authorization Logic**

- **Objective:** Test role-based access conditions.
- **Preconditions:** Users with different roles defined in the database.

5. **Trace Coverage**

- Use tools to ensure all code paths, conditions, and branches are covered.
- For example, ensure every `if`-`else` branch is tested.

6. **Automate When Possible**

- Utilize white box testing frameworks or tools such as **JUnit**, **PyTest**, or **Selenium** for repetitive test case execution.

2. On the iris dataset, perform KNN algorithm and discuss result.

To perform the **K-Nearest Neighbors (KNN)** algorithm on the **Iris dataset**, we will follow these steps:

1. **Load and Preprocess the Data**: Load the Iris dataset and perform any necessary preprocessing (e.g., handling missing values, scaling features).
2. **Split the Data**: Split the data into training and testing sets.
3. **Train the KNN Model**: Train the KNN algorithm on the training set.
4. **Predict and Evaluate**: Use the trained KNN model to make predictions on the test set and evaluate the model's performance.
5. **Discuss the Results**: Analyze the performance of the KNN model based on evaluation metrics such as accuracy, confusion matrix, etc.

```
# Importing necessary libraries
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
import matplotlib.pyplot as plt

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

# Step 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)

# Step 3: Feature Scaling (important for KNN as it is distance-based)
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Step 4: Train the KNN model
knn = KNeighborsClassifier(n_neighbors=3)  # Using K=3
knn.fit(X_train, y_train)

# Step 5: Make predictions on the test set
y_pred = knn.predict(X_test)

# Step 6: Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)
class_report = classification_report(y_test, y_pred)

# Print the results
print(f'Accuracy: {accuracy * 100:.2f}%')
print('Confusion Matrix:')
```

```
print(conf_matrix)
print('Classification Report:')
print(class_report)

# Optional: Plot confusion matrix
import seaborn as sns
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=iris.target_names,
yticklabels=iris.target_names)
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix')
plt.show()
```

3. Implement black box test cases for Walmart ecommerce site.

Black box testing for a Walmart e-commerce site involves testing the application's functionality, usability, and performance without any knowledge of its internal code or architecture. The focus is on user experience, workflows, and expected outcomes based on requirements.

Here's how you can implement **black box test cases** for a Walmart e-commerce site:

1. **Identify Test Scenarios**
Based on the main features of the Walmart site:
  - **User Registration and Login**
  - **Search and Filtering**
  - **Product Details Page**
  - **Add to Cart and Checkout**
  - **Payment Gateway**
  - **Order Tracking**
  - **Customer Support Features**
  - **Mobile Responsiveness**
2. **Structure of Test Cases**
Each test case should include:
  - **Test Case ID:** Unique identifier.
  - **Description:** The purpose of the test case.
  - **Preconditions:** Any prerequisites or setup required.
3. **Sample Black Box Test Cases**
a) **User Registration**
  - **Test Case ID:** BB001
  - **Description:** Verify user registration with valid and invalid inputs.
  - **Preconditions:** Registration page is accessible.
b) **Search Functionality**
  - **Test Case ID:** BB002
  - **Description:** Verify the search bar functionality for various scenarios.
  - **Preconditions:** The website has an active catalog of products.
c) **Add to Cart**
  - **Test Case ID:** BB003
  - **Description:** Verify adding items to the shopping cart.
  - **Preconditions:** User is logged in, products are available in the catalog.
d) **Checkout Process**
  - **Test Case ID:** BB004

  - **Description:** Verify the checkout process with valid and invalid payment details.
  - **Preconditions:** User is logged in and has items in the cart.

e) **Mobile Responsiveness**
  - **Test Case ID:** BB005
  - **Description:** Verify the website's responsiveness on various devices.
  - **Preconditions:** Mobile devices or emulators are available.

4. **Execution and Reporting**
  - **Execution:** Perform the test cases manually or automate them using tools like **Selenium**, **Appium**, or Walmart's internal testing tools.
  - **Reporting:** Document the actual results, mark test cases as pass/fail, and log bugs for any failures.

4. Implement K-Means Clustering on the proper data set of your choice.

```
# Import necessary libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import silhouette_score

# Step 1: Load the Iris dataset
iris = load_iris()
X = iris.data  # Features
y = iris.target  # Actual labels (species)

# Step 2: Preprocess the data (Standardize the features)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Step 3: Apply K-Means clustering
kmeans = KMeans(n_clusters=3, random_state=42)  # We know there are 3 species
kmeans.fit(X_scaled)

# Step 4: Get the cluster labels
cluster_labels = kmeans.labels_

# Step 5: Evaluate the clustering performance using silhouette score
sil_score = silhouette_score(X_scaled, cluster_labels)
print(f'Silhouette Score: {sil_score:.2f}')

# Step 6: Visualize the clusters (We will use two features: Petal Length and Petal Width for visualization)
plt.figure(figsize=(8, 6))
plt.scatter(X_scaled[:, 2], X_scaled[:, 3], c=cluster_labels, cmap='viridis', marker='o')
plt.title('K-Means Clustering on Iris Dataset')
plt.xlabel('Standardized Petal Length')
plt.ylabel('Standardized Petal Width')
plt.colorbar(label='Cluster Label')
```

```
plt.show()

# Step 7: Compare the clustering result with the actual labels
# Create a DataFrame for comparison
comparison_df = pd.DataFrame({'Actual': y, 'Predicted': cluster_labels})
print(comparison_df.head())

# Step 8: Optional: Visualize the cluster centers
plt.figure(figsize=(8, 6))
plt.scatter(X_scaled[:, 2], X_scaled[:, 3], c=cluster_labels, cmap='viridis', marker='o')
plt.scatter(kmeans.cluster_centers_[:, 2], kmeans.cluster_centers_[:, 3], s=300, c='red', marker='X',
label='Centroids')
plt.title('K-Means Clustering with Centroids')
plt.xlabel('Standardized Petal Length')
plt.ylabel('Standardized Petal Wi
```

5. Prepare test plan for an identifies mobile application

Test Plan for a Mobile Application
1. Test Plan Identifier
  - **Test Plan ID:** TP-2024-01
  - **Application Name:** [Insert Mobile App Name]
  - **Version:** [Insert Version Number]
  - **Test Plan Prepared By:** [Your Name/Team Name]
  - **Date:** [Insert Date]
2. Introduction
  - **Objective:**
    To ensure the mobile application functions as expected across various devices, operating systems, and network conditions, while meeting usability, performance, and security requirements.
  - **Scope:**
    Testing includes functionality, UI/UX, performance, compatibility, security, and localization across Android and iOS platforms.
3. Test Objectives
  - Validate core functionalities like login, user workflows, and primary features.
  - Ensure compatibility across supported devices, screen sizes, and OS versions.
  - Evaluate performance under different network conditions.
4. Features to Be Tested
  - **Core Features:**
    - User authentication (login, registration, password recovery).
    - Navigation and menu interactions. .
  - **Performance:**
    - App loading time.
    - Response to high traffic or heavy data processing.
  - **Compatibility:**
    - Across different screen sizes and orientations.
    - On Android (version X+), iOS (version Y+).
  - **Localization (if applicable):**
    - Correct translations, currency formats, and date formats.
5. Features Not to Be Tested
  - Deprecated or future features not included in the current release.

- Backend server-side code (if managed separately).
6. Assumptions and Risks
- **Assumptions:**
  - Development is complete, and all builds are stable for testing.
- **Risks:**
  - Last-minute changes may delay testing timelines.
7. Testing Approach
- **Testing Types:**
  - **Functional Testing:** Validates features against requirements.
  - **Usability Testing:** Checks ease of use and navigation.
  - **Compatibility Testing:** Tests across devices, OS versions, and screen sizes.
  - **Performance Testing:** Analyzes speed, stability, and scalability.
  - **Security Testing:** Identifies vulnerabilities and ensures data protection.
8. Test Environment
- **Devices:**
  - Android phones (e.g., Samsung Galaxy S21, Pixel 6).
  - iOS devices (e.g., iPhone 13, iPad Pro).
- **Operating Systems:**
  - Android OS (version X to latest).
  - iOS (version Y to latest).
10. Deliverables
- Test Plan Document.
- Test Cases (manual and automated).
- Bug Reports (with severity and priority).
- Test Execution Summary.
- Final Test Report.
11. Entry and Exit Criteria
- **Entry Criteria:**
  - Stable build ready for testing.
  - Test environment set up.
  - Test cases approved by stakeholders.
- **Exit Criteria:**
  - All test cases executed.
  - All critical and high-priority bugs resolved.
  - Final test report shared with stakeholders.
14. Approval
- Prepared By: [Your Name]
- Reviewed By: [Reviewer Name]
- Approved By: [Approver Name]

6. Implement apriori algorithm on Online retail dataset and discuss results.

The **Apriori algorithm** is a popular algorithm for association rule learning that identifies frequent itemsets and derives strong association rules from them.
Here’s how the Apriori algorithm can be implemented and the results analyzed using an online retail dataset.
Steps to Implement the Apriori Algorithm

1. **Dataset Overview**
   - The online retail dataset typically contains columns like:
     - `InvoiceNo`: Unique invoice number for each transaction.
     - `StockCode`: Unique product identifier.

- `Description`: Product description.
- `Quantity`: Number of items purchased.
- `InvoiceDate`: Date and time of transaction.
- `UnitPrice`: Price per unit of the product.

2. **Data Preprocessing**
   - Remove irrelevant columns (`InvoiceDate`, `UnitPrice`, etc.) for this analysis.
   - Filter out returns (negative `Quantity`).
   - Group transactions by `InvoiceNo` and list purchased items.
3. **Implementation of Apriori Algorithm**
   - Use a Python library like `mlxtend` for implementation:
     - Generate frequent itemsets using the Apriori algorithm.
4. **Result Analysis**
   - Identify frequently purchased combinations of items.
   - Analyze association rules for actionable insights, such as cross-selling opportunities.

Python Implementation
```python
import pandas as pd
from mlxtend.frequent_patterns import apriori, association_rules

# Load dataset
data = pd.read_csv("Online_Retail.csv")  # Adjust file path as necessary
data = data.dropna(subset=['InvoiceNo', 'Description'])

# Filter transactions and clean data
data = data[data['Quantity'] > 0]
basket = (data.groupby(['InvoiceNo', 'Description'])['Quantity']
          .sum().unstack().fillna(0))
basket = basket.applymap(lambda x: 1 if x > 0 else 0)

# Apply Apriori algorithm
frequent_itemsets = apriori(basket, min_support=0.02, use_colnames=True)
rules = association_rules(frequent_itemsets, metric="lift", min_threshold=1.0)

# Display results
print("Frequent Itemsets:")
print(frequent_itemsets)
print("\nAssociation Rules:")
print(rules)
```

7. Implementation of automation test on amazon web page for searching one plus mobile.

To automate testing on the Amazon web page for searching a "OnePlus mobile," tools like **Selenium** can be used. Below is a step-by-step guide to implementing an automation test.

Steps for Implementation

1. **Setup Environment**
   - Install necessary tools and libraries:
     - Python: Install Python from [python.org](https://www.python.org/).

- Selenium: Install using `pip install selenium`.
- WebDriver: Download the appropriate WebDriver for your browser (e.g., ChromeDriver for Chrome).

2. **Identify Test Case**
   - **Objective:** Verify that the search functionality works for "OnePlus mobile" on the Amazon website.

3. **Automate Using Selenium**
   Below is a Python script using Selenium:

Python Code

```python
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.common.keys import Keys
import time

# Step 1: Set up WebDriver
driver = webdriver.Chrome(executable_path='path_to_chromedriver')  # Replace with your WebDriver path

    # Step 2: Open Amazon homepage
    driver.get("https://www.amazon.com")
    driver.maximize_window()

    # Step 3: Locate the search bar
    search_box = driver.find_element(By.ID, "twotabsearchtextbox")

    # Step 4: Enter search term and submit
    search_term = "OnePlus mobile"
    search_box.send_keys(search_term)
    search_box.send_keys(Keys.RETURN)  # Press Enter key

    # Step 5: Wait for results to load
    time.sleep(3)  # Adjust as necessary for slower networks

    # Step 6: Validate search results
    results = driver.find_elements(By.CSS_SELECTOR, ".s-title")
    assert any("OnePlus" in result.text for result in results), "No OnePlus mobiles found!"
    print("Test Passed: Search results contain 'OnePlus mobile'")
except Exception as e:
    print(f"Test Failed: {e}")
finally:
    # Step 7: Close the browser
    driver.quit()
```

8. Implement Classification algorithm KNN classifier Data Analysis on given iris dataset.

   The K-Nearest Neighbors (KNN) algorithm is a supervised machine learning algorithm used for classification. Here's how to implement it for data analysis on the Iris dataset, a popular dataset in machine learning.

```python
# Import libraries
import pandas as pd
```

```python
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
import matplotlib.pyplot as plt
# Step 1: Load the Iris dataset
iris = load_iris()
X = iris.data  # Features
y = iris.target  # Target (species)
feature_names = iris.feature_names
target_names = iris.target_names
# Step 2: Explore the dataset
print("Feature Names:", feature_names)
print("Target Names:", target_names)
print("Sample Data:\n", pd.DataFrame(X, columns=feature_names).head())
# Step 3: Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
# Step 4: Standardize the data (KNN is sensitive to feature scaling)
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
# Step 5: Train the KNN classifier
k = 3  # Number of neighbors
knn = KNeighborsClassifier(n_neighbors=k)
knn.fit(X_train, y_train)
# Step 6: Make predictions on the test set
y_pred = knn.predict(X_test)
# Step 7: Evaluate the model
print("Accuracy Score:", accuracy_score(y_test, y_pred))
print("\nClassification Report:\n", classification_report(y_test, y_pred, target_names=target_names))
print("\nConfusion Matrix:\n", confusion_matrix(y_test, y_pred))
# Step 8: Visualize results (optional)
# Plot the decision boundary (only for 2D visualization using two features)
plt.figure(figsize=(8, 6))
colors = ['red', 'green', 'blue']
for i, color in enumerate(colors):
   plt.scatter(X_test[y_test == i, 2], X_test[y_test == i, 3], color=color, label=target_names[i])
plt.xlabel('Petal Length (standardized)')
plt.ylabel('Petal Width (standardized)')
plt.legend()
plt.title("KNN Classification (2D Projection)")
plt.show()
```

9. Design a automation testing test cases using java selenium for web application.

Setup Requirements
Tools Required:

Java Development Kit (JDK): Install JDK 8 or later.
Selenium WebDriver: Download Selenium WebDriver and add it to your project.
TestNG/JUnit: Use for structuring and running test cases.
Browser Driver: Download ChromeDriver, GeckoDriver, etc., based on your browser.
Dependencies (If using Maven): Add the following dependencies to the pom.xml file:

```
<dependencies>
  <dependency>
    <groupId>org.seleniumhq.selenium</groupId>
    <artifactId>selenium-java</artifactId>
    <version>4.8.0</version>
  </dependency>
  <dependency>
    <groupId>org.testng</groupId>
    <artifactId>testng</artifactId>
    <version>7.7.0</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

```
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;
import org.testng.Assert;
import org.testng.annotations.AfterClass;
import org.testng.annotations.BeforeClass;
import org.testng.annotations.Test;

public class LoginTest {

  WebDriver driver;

  @BeforeClass
  public void setUp() {
    // Set up ChromeDriver (Adjust the path to your ChromeDriver executable)
    System.setProperty("webdriver.chrome.driver", "path_to_chromedriver");
    driver = new ChromeDriver();
    driver.manage().window().maximize();
    driver.get("https://example.com/login"); // Replace with the actual URL
  }

  @Test
```

```
    public void testLogin() {
        // Step 1: Locate the username and password fields
        WebElement usernameField = driver.findElement(By.id("username")); // Adjust locator as needed
        WebElement passwordField = driver.findElement(By.id("password"));

        // Step 2: Enter valid credentials
        usernameField.sendKeys("testuser"); // Replace with test username
        passwordField.sendKeys("password123"); // Replace with test password

        // Step 3: Locate and click the login button
        WebElement loginButton = driver.findElement(By.id("loginButton")); // Adjust locator as needed
        loginButton.click();

        // Step 4: Verify login success
        WebElement dashboardElement = driver.findElement(By.id("dashboard")); // Adjust locator as needed
        Assert.assertTrue(dashboardElement.isDisplayed(), "Login failed or dashboard not found.");
    }

    @AfterClass
    public void tearDown() {
        // Close the browser
        if (driver != null) {
            driver.quit();
        }
    }
}
```

10. Implement Classification algorithm Decision tree Data Analysis on given iris dataset.

Implementing Classification with a Decision Tree Algorithm on the Iris Dataset
The Decision Tree algorithm is a supervised learning method that splits data into subsets based on feature values. It is widely used for classification tasks due to its interpretability.

```
# Importing necessary libraries
import pandas as pd
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier, export_text, plot_tree
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
import matplotlib.pyplot as plt

# Step 1: Load the Iris dataset
iris = load_iris()
X = iris.data  # Features
y = iris.target  # Target labels
feature_names = iris.feature_names
target_names = iris.target_names

# Step 2: Explore the dataset
print("Feature Names:", feature_names)
```

```python
print("Target Names:", target_names)
print("Sample Data:\n", pd.DataFrame(X, columns=feature_names).head())

# Step 3: Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Step 4: Train the Decision Tree Classifier
dt_classifier = DecisionTreeClassifier(criterion="entropy", max_depth=3, random_state=42)
dt_classifier.fit(X_train, y_train)

# Step 5: Visualize the Decision Tree
plt.figure(figsize=(12, 8))
plot_tree(dt_classifier, feature_names=feature_names, class_names=target_names, filled=True,
rounded=True)
plt.title("Decision Tree for Iris Dataset")
plt.show()

# Step 6: Evaluate the Model
y_pred = dt_classifier.predict(X_test)

# Accuracy
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)

# Classification Report
print("\nClassification Report:\n", classification_report(y_test, y_pred, target_names=target_names))

# Confusion Matrix
print("\nConfusion Matrix:\n", confusion_matrix(y_test, y_pred))

# Step 7: Print Decision Rules
decision_rules = export_text(dt_classifier, feature_names=feature_names)
print("\nDecision Tree Rules:\n", decision_rules)
```

11. Implement automation test for Facebook login page using Java selenium.

Test Case Design

Test Case 1: Verify Login with Invalid Credentials

1. Open the browser and navigate to the Facebook login page.
2. Enter an invalid email address and password.
3. Click the "Log In" button.
4. Verify that an error message is displayed.

Test Case 2: Verify Login with Valid Credentials

1. Open the browser and navigate to the Facebook login page.
2. Enter a valid email address and password.
3. Click the "Log In" button.
4. Verify successful login by checking for a specific element on the homepage.

```java
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;
```

```
import org.testng.Assert;
import org.testng.annotations.AfterClass;
import org.testng.annotations.BeforeClass;
import org.testng.annotations.DataProvider;
import org.testng.annotations.Test;

public class FacebookLoginTest {
    WebDriver driver;

    @BeforeClass
    public void setUp() {
        // Set up ChromeDriver (adjust path to ChromeDriver executable)
        System.setProperty("webdriver.chrome.driver", "path_to_chromedriver");
        driver = new ChromeDriver();
        driver.manage().window().maximize();
        driver.get("https://www.facebook.com/");
    }

    @Test(dataProvider = "loginData")
    public void testLogin(String email, String password, String expectedMessage) {
        // Step 1: Locate the email and password fields
        WebElement emailField = driver.findElement(By.id("email")); // Adjust locator as needed
        WebElement passwordField = driver.findElement(By.id("pass"));

        // Step 2: Enter the email and password
        emailField.clear();
        emailField.sendKeys(email);
        passwordField.clear();
        passwordField.sendKeys(password);

        // Step 3: Locate and click the "Log In" button
        WebElement loginButton = driver.findElement(By.name("login")); // Adjust locator as needed
        loginButton.click();

        // Step 4: Verify the result
        if (expectedMessage.equalsIgnoreCase("Success")) {
            // Check if login was successful (example: check the presence of the user profile icon)
            WebElement profileIcon = driver.findElement(By.xpath("//div[@aria-label='Your profile']")); //
Update the locator
            Assert.assertTrue(profileIcon.isDisplayed(), "Login failed, profile icon not found.");
        } else {
            // Check if an error message is displayed
            WebElement errorMessage = driver.findElement(By.xpath("//div[contains(text(),'email or mobile
number')]"));
            Assert.assertTrue(errorMessage.isDisplayed(), "Error message not displayed for invalid login.");
        }
    }

    @DataProvider(name = "loginData")
    public Object[][] loginData() {
        return new Object[][] {
```

```
            {"invalid_email@example.com", "invalidPassword", "Error"},
            {"valid_email@example.com", "validPassword", "Success"} // Replace with actual test credentials
        };
    }

    @AfterClass
    public void tearDown() {
        // Close the browser
        if (driver != null) {
            driver.quit();
        }
    }
}
```

12. Design and implement SVM for classification with the proper data set of your choice. Comment on Design and Implementation for Linearly non separable Dataset.

Objective: Implement a Support Vector Machine (SVM) for classification using a dataset of choice, and discuss how SVM handles **linearly non-separable** data.
Steps for Implementation:

1. **Load and Visualize Dataset**:
2. **SVM with Linear and RBF Kernels**:
3. **Model Training and Evaluation**:

Implementation Code:

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_moons
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score

# Generate non-linearly separable dataset
X, y = make_moons(n_samples=500, noise=0.2, random_state=42)

# Visualize the dataset
plt.scatter(X[:, 0], X[:, 1], c=y, cmap='viridis')
plt.title("Non-linearly Separable Dataset")
plt.show()

# Split data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Train with linear kernel
linear_svm = SVC(kernel='linear')
linear_svm.fit(X_train, y_train)
y_pred_linear = linear_svm.predict(X_test)
print(f"Linear Kernel Accuracy: {accuracy_score(y_test, y_pred_linear)}")
```

```
# Train with RBF kernel
rbf_svm = SVC(kernel='rbf', gamma=0.5)
rbf_svm.fit(X_train, y_train)
y_pred_rbf = rbf_svm.predict(X_test)
print(f"RBF Kernel Accuracy: {accuracy_score(y_test, y_pred_rbf)}")

# Decision boundaries visualization
def plot_decision_boundary(model, X, y, title):
    h = .02  # Step size in the mesh
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
    Z = model.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    plt.contourf(xx, yy, Z, alpha=0.8)
    plt.scatter(X[:, 0], X[:, 1], c=y, edgecolors='k', marker='o')
    plt.title(title)
    plt.show()

# Plot decision boundaries
plot_decision_boundary(linear_svm, X, y, "Linear Kernel Decision Boundary")
plot_decision_boundary(rbf_svm, X, y, "RBF Kernel Decision Boundary")
```

13. Design white box test cases for web application.

White Box Test Cases for Web Application (Short Version)

1. **Test Case 1: Validate Email Format**
  - **Objective**: Ensure proper email format validation.
  - **Steps**:
    1. Enter invalid email (`user@domain`) in the email field.
    2. Submit the form.
2. **Test Case 2: Validate Password Length**
  - **Objective**: Check if password meets minimum length requirement (8 characters).
  - **Steps**:
    1. Enter password with less than 8 characters.
    2. Submit the form.
3. **Test Case 3: Valid Login**
  - **Objective**: Verify successful login with valid credentials.
  - **Steps**:
    1. Enter valid email and password.
    2. Submit the form.
  - **Expected Result**: Redirect to home/dashboard page.
4. **Test Case 4: Invalid Login (Incorrect Password)**
  - **Objective**: Ensure system handles incorrect password.
5. **Test Case 5: SQL Injection Test**
  - **Objective**: Test for SQL injection vulnerability.
  - **Steps**:
    1. Enter `' OR 1=1 --` in the email field.

2. Submit the form.
6. **Test Case 6: Check Session Expiration**
   - **Objective**: Ensure session expires after a period of inactivity.
   - **Steps**:
     1. Log in and remain idle for the session timeout period.
     2. Try accessing a protected page.
7. **Test Case 7: Form Field Validation**
   - **Objective**: Verify that both email and password fields are required.
   - **Steps**:
     1. Leave one or both fields empty.
     2. Submit the form.
8. **Test Case 8: Button Disabled State**
   - **Objective**: Ensure login button is only enabled when both fields are filled.
   - **Steps**:
     1. Leave fields empty.
     2. Verify if the login button is disabled.

14. Implement a basic not gate using perceptron.

Implementing a Basic NOT Gate Using Perceptron

A perceptron is a simple neural network model used for binary classification. While a single perceptron can typically handle linearly separable problems like AND, OR, etc., it can also implement simple logic gates such as NOT. A NOT gate has one input and one output, where the output is the inverse of the input.

For a basic NOT gate:
- Input: $x \in \{0, 1\}$
- Output: $\text{NOT}(x) = \{1, 0\}$

Steps to Implement a NOT Gate Using Perceptron:

1. **Perceptron Model**:
   - A perceptron has an input, a weight, a bias, and an activation function (usually a step function in the case of a binary output).

2. **NOT Gate Truth Table**:

Input (x)	Output (y)
0	1
1	0

3. **Perceptron Formula**:
   - The output y of a perceptron is given by:
$$
y = \text{activation}(w \cdot x + b)
$$
     where w is the weight, x is the input, and b is the bias.

   - **Activation function**: Use a step function:
     \[
     \text{activation}(z) =

\begin{cases}
1 & \text{if } z \geq 0 \\
0 & \text{if } z < 0
\end{cases}
\]

4. **Training the Perceptron**:
  - Since a NOT gate is a simple linear problem, you can directly set weights and biases without training, as it is possible to find a set of weights that satisfies the problem.

Code Implementation:

```python
import numpy as np

# Perceptron class definition
class Perceptron:
    def __init__(self, input_size):
        self.weights = np.random.randn(input_size)  # Random initialization of weights
        self.bias = np.random.randn()  # Random initialization of bias

    # Step activation function
    def step_function(self, x):
        return 1 if x >= 0 else 0

    # Perceptron forward pass
    def predict(self, inputs):
        linear_output = np.dot(inputs, self.weights) + self.bias
        return self.step_function(linear_output)

# Initialize Perceptron with 1 input (since NOT gate has 1 input)
perceptron = Perceptron(input_size=1)

# Training process (for NOT gate)
# For a NOT gate, input-output pairs are (0 -> 1) and (1 -> 0)
inputs = np.array([0, 1])
outputs = np.array([1, 0])

# Adjust weights and bias manually since it's a simple gate
perceptron.weights = np.array([-2])  # This weight will result in output 1 for input 0 and 0 for input 1
perceptron.bias = 1  # The bias shifts the activation to make the gate behave like NOT

15. Prepare test plan for an identifies mobile application.

### **Test Plan for Identifies Mobile Application (Short Version)**

**1. Objective:**
- Ensure the "Identifies" mobile app functions correctly, identifies objects accurately, and provides a smooth user experience across various devices and platforms.

**2. Scope:**
```

- **In Scope**: Functional testing (object identification), usability, performance, compatibility, security, and regression testing.
- **Out of Scope**: Testing third-party integrations and the underlying object recognition algorithms.

3. Test Strategy:
- **Test Types**: Functional, Usability, Compatibility, Performance, and Security Testing.
- **Levels**: Unit, Integration, System, and Acceptance testing.

4. Resource Requirements:
- **Tools**: Appium (for automation), JIRA (for bug tracking), Apache JMeter (for performance).
- **Devices**: Android phones (versions 8.0, 9.0, 10.0) and iPhones (iOS 12, 13, 14).

5. Key Test Cases:
- **Functional**: Verify object recognition, camera access, and correct identification of objects.
- **Usability**: Test user interface, ease of navigation, and responsiveness.
- **Performance**: Check app's response time and stability under load.
- **Compatibility**: Ensure the app works on different OS versions and device configurations.
- **Security**: Test permissions handling and data privacy (camera access, storage).

6. Schedule:
- Test Planning, Execution, and Reporting phases, with deadlines for each stage.

7. Risks:
- Device compatibility issues and access to image recognition models.
- Mitigation strategies include testing on a range of devices and using mock APIs when necessary.

8. Deliverables:
- Test Plan, Test Cases, Test Reports, and Defect Logs.

16. On the fruit dataset, compare the performance of SVM and KNN on the basis of their accuracy.

Comparison of SVM and KNN on Fruit Dataset (Accuracy)

To compare the performance of **Support Vector Machine (SVM)** and **K-Nearest Neighbors (KNN)** classifiers on a fruit dataset, we evaluate both algorithms based on their accuracy, using standard classification metrics.

Steps for Comparison:

1. **Data Preparation:**
   - **Dataset**: A fruit dataset containing features like weight, color, shape, and size, with labeled categories (e.g., apple, banana, orange).
   - **Preprocessing**: Normalize or standardize the data (as required by KNN) and split into training and test sets.

2. **Model Training:**
   - **SVM**: Train an SVM model using a kernel (e.g., linear or RBF).
   - **KNN**: Train a KNN model with a specific number of neighbors (k), e.g., k=5.

3. **Evaluation:**

  - **Accuracy**: Evaluate both models using accuracy on the test set.
  - **Cross-validation**: Optionally, use cross-validation to ensure the models generalize well.

17. Implement black box testing technique on amazon ecommerce site.

Black Box Testing on Amazon eCommerce Site (Short Version)

Objective: To test the functionality of the Amazon eCommerce site without knowing the internal workings of the application. The focus is on verifying the user interface, input validation, and output correctness.

Testing Areas:

1. **Login Functionality**:
  - **Test Case 1**: Verify login with valid credentials.
    - **Steps**: Enter a valid email and password, click "Login."
    - **Expected Result**: Successful login and redirection to the homepage.

  - **Test Case 2**: Verify login with invalid credentials.
    - **Steps**: Enter invalid credentials, click "Login."
    - **Expected Result**: Display error message "Incorrect email or password."

2. **Product Search**:
  - **Test Case 3**: Verify search functionality with valid keywords.
    - **Steps**: Enter a product name (e.g., "laptop") in the search bar and click search.
    - **Expected Result**: Display a list of products related to the search term.

  - **Test Case 4**: Verify search with no results.
    - **Steps**: Enter a random or incorrect product name.
    - **Expected Result**: Display "No results found."

3. **Add to Cart**:
  - **Test Case 5**: Verify adding a product to the shopping cart.
    - **Steps**: Select a product, click "Add to Cart."
    - **Expected Result**: Product is added to the cart and cart count updates.

4. **Checkout Process**:
  - **Test Case 6**: Verify successful checkout with valid payment information.
    - **Steps**: Add product to cart, proceed to checkout, enter valid shipping and payment info.
    - **Expected Result**: Successful order confirmation.

5. **Navigation and UI**:
  - **Test Case 7**: Verify all links on the homepage are functional.
    - **Steps**: Click on various categories (e.g., "Electronics," "Books").
    - **Expected Result**: Navigate to the respective category page without errors.

6. **Filters and Sorting**:
  - **Test Case 8**: Verify product filters and sorting options.
    - **Steps**: Apply filters (e.g., price range, brand), sort by price or ratings.
    - **Expected Result**: Product list updates correctly based on selected filters.

18. Implement Classification algorithm SVM Classifier Data Analysis on given iris dataset.

Implementing SVM Classifier on Iris Dataset

The **Support Vector Machine (SVM)** classifier is a powerful supervised learning algorithm commonly used for classification tasks. In this example, we will implement an SVM classifier on the famous **Iris dataset** to classify flowers into three categories: setosa, versicolor, and virginica.

Steps to Implement SVM Classifier:

1. **Load the Iris Dataset**: The Iris dataset is available in `sklearn.datasets`.
2. **Preprocessing**: Split the data into training and test sets.
3. **Train the SVM Classifier**: Use the `SVC` class from `sklearn.svm` to train the model.
4. **Evaluate the Model**: Measure performance using accuracy.

Code Implementation (Python):

```python
# Import necessary libraries
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt

# Step 1: Load the Iris dataset
iris = datasets.load_iris()
X = iris.data  # Features (sepal length, sepal width, petal length, petal width)
y = iris.target  # Labels (setosa=0, versicolor=1, virginica=2)

# Step 2: Split the dataset into training and testing sets (80% training, 20% testing)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Step 3: Create and train the SVM model
svm_model = SVC(kernel='linear')  # Using a linear kernel
svm_model.fit(X_train, y_train)

# Step 4: Make predictions on the test set
y_pred = svm_model.predict(X_test)

# Step 5: Evaluate the performance of the classifier
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy of SVM Classifier: {accuracy * 100:.2f}%")

# Optional: Visualize the decision boundaries for the first two features
plt.figure(figsize=(8, 6))
plt.scatter(X_test[:, 0], X_test[:, 1], c=y_pred, cmap='viridis', marker='o')
plt.title("SVM Classifier on Iris Dataset (First 2 Features)")
plt.xlabel('Sepal Length')
plt.ylabel('Sepal Width')
plt.colorbar(label='Predicted Label')
```

plt.show()
```

19. Design white box test cases for web application.

### **White Box Test Cases for Web Application (Short Version)**

**White Box Testing** focuses on testing the internal workings and logic of an application. Here are examples of white box test cases for a web application:

### **1. Test Case: Login Functionality (Code Path Coverage)**
- **Objective**: Test the login process to ensure proper handling of valid and invalid credentials.
- **Steps**:
  1. Enter valid username and password.
  2. Submit the form.
  3. Enter invalid username or password.
  4. Submit the form.

### **2. Test Case: Form Validation (Boundary Value Testing)**
- **Objective**: Ensure form fields handle boundary inputs correctly (e.g., character limits).
- **Steps**:
  1. Enter data within valid and invalid boundaries for each field (e.g., min/max characters for text fields).
  2. Submit the form.

### **3. Test Case: Data Retrieval from Database (SQL Injection Prevention)**
- **Objective**: Ensure that the web application is secure against SQL injection attacks.
- **Steps**:
  1. Input malicious SQL code into form fields (e.g., `' OR 1=1 --`).
  2. Submit the form.

### **4. Test Case: User Role-Based Access (Code Path Testing)**
- **Objective**: Verify that users with different roles (e.g., admin, guest, regular user) can access appropriate pages.
- **Steps**:
  1. Log in as a regular user.
  2. Try accessing an admin-only page.
  3. Log in as an admin user.
  4. Try accessing admin-only pages.
- **Expected Result**:
  - Regular users should be denied access to admin pages.
  - Admin users should be able to access all pages.

### **5. Test Case: Session Timeout (Path Testing)**
- **Objective**: Verify that the session expires after a certain period of inactivity.
- **Steps**:
  1. Log in to the application.
  2. Remain inactive for the session timeout duration.
  3. Attempt to perform any action after timeout.

### **6. Test Case: Error Handling and Exception Management (Exception Path Coverage)**
- **Objective**: Test the application's behavior under exception conditions (e.g., network failure).
- **Steps**:
```

  1. Simulate a network failure or incorrect API response.
  2. Perform an action (e.g., submitting a form).

7. Test Case: Navigation Links (Control Flow Coverage)
- **Objective**: Verify that all internal navigation links direct users to the correct pages.
- **Steps**:
  1. Click each navigation link on the homepage.
  2. Ensure each link redirects to the expected page.

20. Implement Naïve Bayes Classifier and K-Nearest Neighbour Classifier on Data set of your choice. Test and Compare for Accuracy and Precision.

Comparison of Naïve Bayes and K-Nearest Neighbors (KNN) Classifiers

Let's implement and compare the **Naïve Bayes** and **K-Nearest Neighbors (KNN)** classifiers on a dataset of choice (e.g., the **Iris dataset**) to evaluate their accuracy and precision.

Steps:
1. **Load and Preprocess Data**: We'll use the **Iris dataset** for this example.
2. **Train the Classifiers**: We'll implement both Naïve Bayes and KNN classifiers.
3. **Evaluate the Models**: Measure their **Accuracy** and **Precision**.

Code Implementation (Python):

```python
# Import necessary libraries
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, precision_score

# Step 1: Load the Iris dataset
iris = datasets.load_iris()
X = iris.data  # Features
y = iris.target  # Labels

# Step 2: Split the dataset into training and testing sets (80% training, 20% testing)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Step 3: Implement Naïve Bayes Classifier
nb_classifier = GaussianNB()
nb_classifier.fit(X_train, y_train)

# Step 4: Implement K-Nearest Neighbors Classifier
knn_classifier = KNeighborsClassifier(n_neighbors=3)
knn_classifier.fit(X_train, y_train)

# Step 5: Make predictions for both models
y_pred_nb = nb_classifier.predict(X_test)
```

y_pred_knn = knn_classifier.predict(X_test)

# Step 6: Evaluate both models using Accuracy and Precision
accuracy_nb = accuracy_score(y_test, y_pred_nb)
precision_nb = precision_score(y_test, y_pred_nb, average='weighted') # For multi-class classification

accuracy_knn = accuracy_score(y_test, y_pred_knn)
precision_knn = precision_score(y_test, y_pred_knn, average='weighted')

# Output the results
print(f"Naïve Bayes Accuracy: {accuracy_nb * 100:.2f}%")
print(f"Naïve Bayes Precision: {precision_nb:.2f}")
print(f"KNN Accuracy: {accuracy_knn * 100:.2f}%")
print(f"KNN Precision: {precision_knn:.2f}")
```

### **Expected Output** (Example):

```
Naïve Bayes Accuracy: 96.67%
Naïve Bayes Precision: 0.97
KNN Accuracy: 96.67%
KNN Precision: 0.97
```
```