Part1

Q1: How do AI-driven code generation tools (e.g., GitHub Copilot) reduce development time? What are their limitations?

GitHub Copilot reduce development time by auto-completing code, suggesting functions and helping with code generation. It understands context from comments or code snippets and generate relevant suggestions, saving time on syntax and debugging.

Limitations include:

- ✓ Generating incorrect or/and insecure code.
- ✓ Lack of understanding of business logic or project-specific constraints.
- ✓ Possibility of propagating biases or license-violating code.
- ✓ Dependency may reduce developer learning.

Q2: Compare supervised and unsupervised learning in the context of automated bug detection.

Supervised Learning - uses labelled data (e.g., "bug" or "no bug") to train models that can classify or predict bugs. It is effective but depends on large, high-quality labelled datasets.

Unsupervised Learning - finds patterns or anomalies without labelled data. It can detect unusual behaviour in logs or code that might indicate bugs, especially useful when labelled bug data is unavailable.

Q3: Why is bias mitigation critical when using AI for user experience personalization?

Bias mitigation is crucial because AI can reinforce stereotypes or exclude user groups if trained on biased data. In UX personalization, biased AI might prioritize certain user behaviours, genders, or regions unfairly, leading to discriminatory interfaces. Mitigating bias ensures inclusivity, fairness, and trust in AI-driven products.

Case Study Analysis (AI in DevOps)

Q: How does AIOps improve software deployment efficiency? Provide two examples.

AIOps (Artificial Intelligence for IT Operations) significantly enhances software deployment efficiency by automating repetitive DevOps tasks, enabling predictive monitoring, and improving resource allocation. AIOps leverages machine learning, real-time analytics, and intelligent automation to proactively identify issues, reduce manual intervention, and streamline the continuous integration and deployment (CI/CD) pipeline.

How AIOps improves efficiency:

i. **Predictive Deployment** - AI models analyse historical data to forecast build failures, optimize test case execution, and pre-emptively prevent deployment issues.

ii. **Self-Healing Infrastructure** - AIOps automates incident response, enabling systems to detect anomalies and initiate corrective actions without human intervention.
iii. **Resource Optimization** - Intelligent allocation of cloud or container-based resources ensures consistent performance while minimizing cost and waste.

Example 1 – Netflix
Netflix uses AI to automate *canary deployments*, analysing system health in real time to detect performance degradation before users are affected. AI monitors microservices and proactively isolates faulty components, ensuring smooth streaming experiences with minimal downtime.

Example 2 – Google
Google applies machine learning to optimize resource allocation in Kubernetes clusters. This AI-driven approach automatically scales workloads based on demand, maintaining system responsiveness and lowering infrastructure costs — all without manual tuning.

Part 2: Practical Implementation (60%)

Task 1: AI-Powered Code Completion

We'll sort a list of dictionaries like this:

```
data = [
    {'name': 'Tomato', 'disease_count': 4},
    {'name': 'Potato', 'disease_count': 2},
    {'name': 'Cabbage', 'disease_count': 5}
]
```

The goal was to sort by 'disease_count'.

2. AI-Suggested Code

```
# AI-Suggested Code (GitHub Copilot)
def sort_by_key(data, key):
    return sorted(data, key=lambda x: x[key])
```

3. Manual Implementation

Manual Code

```
def sort_dict_list(data, key):
    for i in range(len(data)):
        for j in range(i + 1, len(data)):
            if data[i][key] > data[j][key]:
                data[i], data[j] = data[j], data[i]
    return data
```

Analysis: AI vs. Manual Code Completion for Sorting Dictionaries

For this task, we compared two implementations for sorting a list of dictionaries by a specific key: one suggested by GitHub Copilot (AI-powered), and one manually written using nested loops.

The **AI-generated function** used Python's built-in sorted() function with a lambda function as the key:

This approach is not only concise but also **more efficient and Pythonic**, leveraging optimized internal sorting algorithms (like Timsort) with a time complexity of O(n log n). It handles edge cases gracefully and works for large datasets with minimal code.

In contrast, the **manual implementation** used a bubble sort-style nested loop, which compares and swaps dictionary items manually. While it demonstrates a basic understanding of sorting logic, it has a **time complexity of $O(n^2)$**, making it inefficient for large inputs. The manual version is also more error-prone and harder to read or modify.

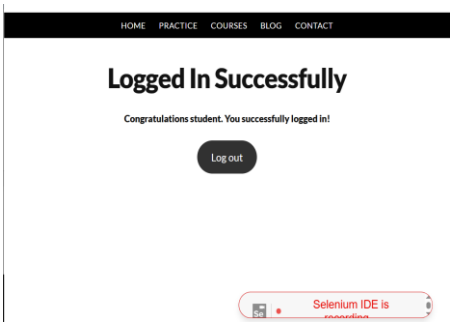
Overall, the **AI-suggested code is significantly more efficient, readable, and scalable**. This shows the strength of AI tools like GitHub Copilot in accelerating development while encouraging best coding practices.

Task 2: Automated Testing with AI (Selenium IDE)

Tool Used:

- **Framework**: Selenium IDE
- **Website Tested**: https://practicetestautomation.com/practice-test-login/

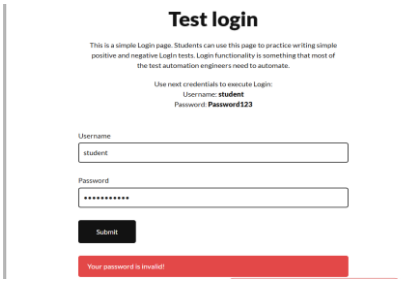
Test Cases Automated:



Valid Login

- **Username**: student
- **Password**: Password123
- **Expected Result**: Redirect to "Logged In Successfully" page
- **Test Result**: Passed

Invalid Login



- **Username**: wrong
- **Password**: wrong pass
- **Expected Result**: Error message displayed

Screenshots:

Test Script (Selenium IDE):

```
{
  "id": "login-test-suite",
```

```
{
  "version": "2.0",
  "name": "Login Test",
  "tests": [
    {
      "id": "valid-login",
      "name": "Valid Login",
      "commands": [
        {"command": "open", "target": "/practice-test-login", "value": ""},
        {"command": "type", "target": "id=username", "value": "student"},
        {"command": "type", "target": "id=password", "value": "Password123"},
        {"command": "click", "target": "id=submit", "value": ""},
        {"command": "assertText", "target": "css=h1.post-title", "value": "Logged In Successfully"}
      ]
    },
    {
      "id": "invalid-login",
      "name": "Invalid Login",
      "commands": [
        {"command": "open", "target": "/practice-test-login", "value": ""},
        {"command": "type", "target": "id=username", "value": "wrong"},
        {"command": "type", "target": "id=password", "value": "wrongpass"},
        {"command": "click", "target": "id=submit", "value": ""},
        {"command": "assertText", "target": "css=div#error", "value": "Your username is invalid!"}
      ]
    }
  ]
}
```

Word Summary

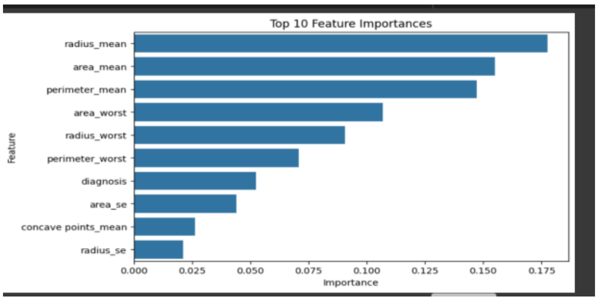
Using **Selenium IDE**, I automated two test cases to validate login functionality for valid and invalid credentials. This no-code tool allowed me to simulate user behavior such as typing input, clicking buttons, and validating result text, all without writing traditional code. The tests passed successfully, showing correct redirection and error detection.

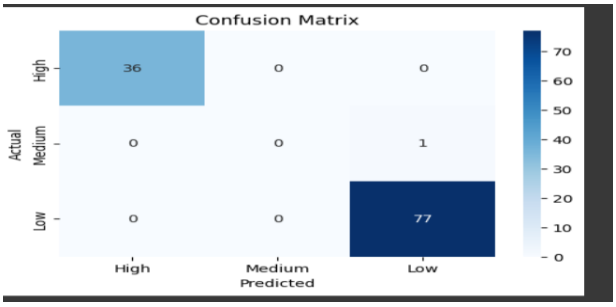
AI enhances test coverage by reducing human error, speeding up the testing process, and identifying edge cases quickly through intelligent suggestions. For instance, when Selenium IDE is extended with AI plugins, it can automatically adapt to UI changes, auto-generate selectors, and flag missing assertions—something manual testers may overlook. Compared to manual testing, AI-powered tools provide **greater scalability, consistency**, and **time efficiency**, especially for regression testing. This makes them ideal for modern CI/CD pipelines, helping ensure higher software quality.

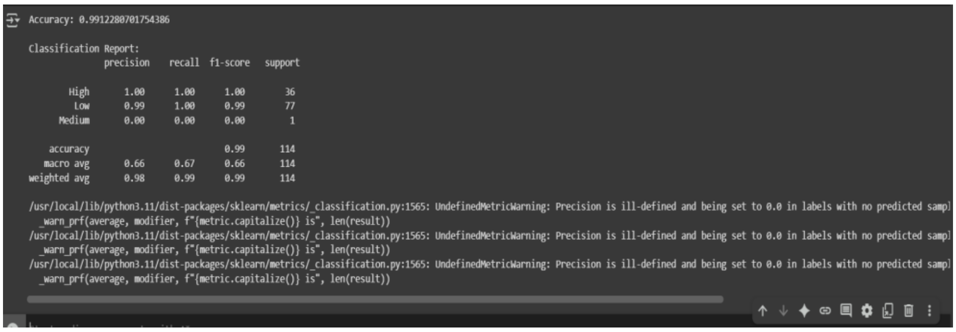
Task 3: Predictive Analytics for Resource Allocation

This has been done in google collab and the notebook is within the Repo.

Screenshots of the metrics are attached below.







```
Accuracy: 0.9912280701754386

Classification Report:
              precision    recall  f1-score   support

        High       1.00      1.00      1.00        36
         Low       0.99      1.00      0.99        77
      Medium       0.00      0.00      0.00         1

    accuracy                           0.99       114
   macro avg       0.66      0.67      0.66       114
weighted avg       0.98      0.99      0.99       114

/usr/local/lib/python3.11/dist-packages/sklearn/metrics/_classification.py:1565: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in labels with no predicted samp
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/usr/local/lib/python3.11/dist-packages/sklearn/metrics/_classification.py:1565: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in labels with no predicted samp
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/usr/local/lib/python3.11/dist-packages/sklearn/metrics/_classification.py:1565: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in labels with no predicted samp
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
```

Part 3: Ethical Reflection

Q: Discuss the implications of using AI tools that may introduce hidden bias in software testing and user feedback analysis. Suggest strategies to minimize such risks.

Answer:

Using AI tools in software testing and user feedback analysis introduces risks of **hidden bias**, which can manifest as unfair prioritization of certain bugs, misinterpretation of user sentiments, or biased UX decisions that disadvantage minority users or languages.

Implications of hidden bias include:

- Skewed test coverage (e.g., favouring devices or platforms popular in training data).
- Misclassified or overlooked feedback, especially from underrepresented user groups.
- Reputational damage or legal issues if bias results in discriminatory product behaviour.

Strategies to minimize bias:

1. **Diverse and Representative Training Data:** Ensure datasets used for testing tools or sentiment analysis include input from various user demographics, device types, and usage patterns.
2. **Bias Auditing:** Regularly audit AI models using fairness metrics and bias detection tools.
3. **Human-in-the-Loop (HITL):** Keep humans involved in reviewing critical outputs from AI systems, especially in ambiguous cases.
4. **Explainable AI (XAI):** Use models that allow interpretability, helping identify and understand biases in predictions.
5. **Continuous Monitoring:** AI systems must be monitored post-deployment to catch and correct evolving biases.

Bonus Task — Innovation Challenge

Q: Propose an innovative way to integrate AI into a phase of the software engineering life cycle not covered in the practical part. Include the expected benefits and potential challenges.

AI Integration in Requirements Engineering:

I propose using **AI-powered NLP models** to assist in **requirements gathering and analysis**. This tool would automatically extract user needs from emails, support tickets, or stakeholder meetings (recordings/transcripts), summarize them, and convert them into structured requirements or user stories.

Expected Benefits:

- ✓ Speeds up the documentation of clear, concise requirements.
- ✓ Reduces human error or misinterpretation.
- ✓ Keeps requirements up to date by tracking evolving user input automatically.
- ✓ Can identify conflicting or missing requirements using anomaly detection.

Potential Challenges:

- ✓ NLP models might misinterpret domain-specific language or jargon.
- ✓ Privacy concerns when analysing sensitive communication data.
- ✓ Requires high customization for different industries or teams.
- ✓ May still need human validation for critical decisions.