Part 1: Theoretical Analysis

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

How they save time:

- ***Boilerplate Automation:*** They auto-generate repetitive code (like CRUD operations), so you don't have to type everything manually.
- ***Quick Prototyping*:** You can get a working code snippet just by describing what you want in a comment.
- ***Learning Aid:*** They help you explore new libraries or syntax you're not familiar with, reducing time spent on docs.
- ***Context Awareness:*** They use your current file and project structure to suggest relevant code.

Limitations:

- ***Security Risks:*** Can suggest code with vulnerabilities if the training data included flawed examples.
- ***Lack of Understanding***: Doesn't truly "understand" your project's business logic or architecture.
- ***Code Quality Issues:*** May produce inefficient, outdated, or non-optimized code.
- ***Over-reliance Risk:*** Developers might stop thinking critically, leading to skill erosion.

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

	Supervised Learning	**Unsupervised Learning**
How it works	Trained on labelled datasets (e.g., code snippets labelled as "buggy" or "clean")	Finds patterns or anomalies without pre-labelled data (e.g., clustering similar code patterns)
Use case	Classifying new code as bug-prone based on past examples	Detecting unusual code patterns that might indicate a bug
Pros	Highly accurate if you have good labelled data	Doesn't need labelled data; can find new, unknown bug types
Cons	Requires a lot of labelled data, which can be time-consuming to create	May produce more false positives; harder to interpret results

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

- ***Filter Bubbles:*** AI might show users only what they like, limiting exposure to new ideas.
- ***Discrimination:*** Personalization can unfairly exclude certain groups (e.g., job ads shown only to specific demographics).
- ***Reinforcing Stereotypes:*** If training data is biased, the AI will learn and amplify those biases.
- ***User Trust:*** Biased systems lead to poor user experience and loss of trust in the platform.

Case Study Analysis: *AI in DevOps: Automating Deployment Pipelines*

How does AIOps improve software deployment efficiency?

AIOps (Artificial Intelligence for IT Operations) significantly enhances software deployment efficiency by introducing **proactive intelligence** and **automation** into the deployment pipeline. It moves beyond reactive, human-dependent processes to systems that can predict, prevent, and automatically resolve issues.

Two key examples include:

1. **Intelligent Test Optimization:** AI analyzes historical test data to identify which tests are most likely to fail for a given code change. Instead of running the entire, time-consuming test suite, it prioritizes these high-risk tests. This gives developers faster feedback on critical failures, drastically reducing the wait time in the CI/CD pipeline and allowing for quicker iterations.

2. **Automated, Predictive Rollbacks:** AI continuously monitors application performance metrics (like error rates and latency) in real-time after a deployment. If it detects an anomaly that indicates a failing deployment, it can **automatically trigger a rollback** to the previous stable version without waiting for human intervention. This minimizes system downtime and user impact, making the deployment process more robust and reliable.

Part 2: Practical Implementation

TASK 1 AI vs Manual Code Showdown

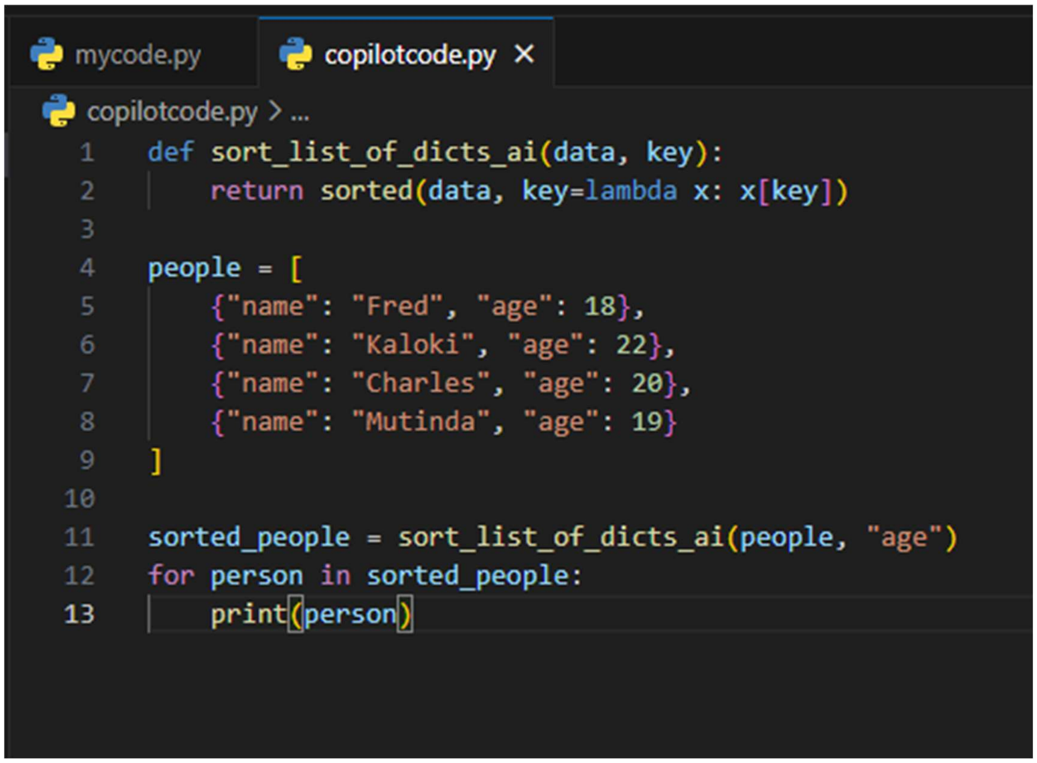
So, I tried both approaches - letting AI do the work and coding it manually - and the results were pretty interesting!

What Happened?

Both versions successfully sorted my list of people by age, from youngest to oldest. The AI used Python's built-in sorted () function with a lambda, while I implemented a bubble sort algorithm from scratch.

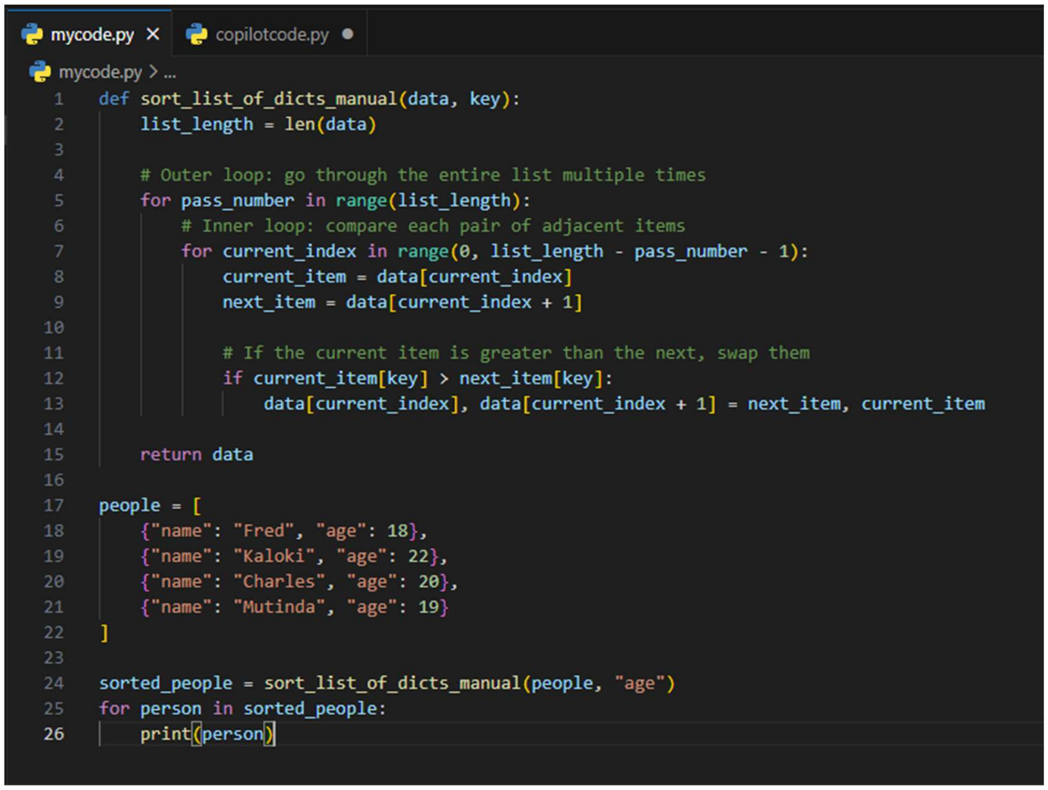
The Speed Difference Was Wild

- **AI**: Literally 2 seconds to write the whole function



```python
def sort_list_of_dicts_ai(data, key):
    return sorted(data, key=lambda x: x[key])

people = [
    {"name": "Fred", "age": 18},
    {"name": "Kaloki", "age": 22},
    {"name": "Charles", "age": 20},
    {"name": "Mutinda", "age": 19}
]

sorted_people = sort_list_of_dicts_ai(people, "age")
for person in sorted_people:
    print(person)
```

- **Me**: About 10-15 minutes of thinking, coding, and debugging



```python
def sort_list_of_dicts_manual(data, key):
    list_length = len(data)

    # Outer loop: go through the entire list multiple times
    for pass_number in range(list_length):
        # Inner loop: compare each pair of adjacent items
        for current_index in range(0, list_length - pass_number - 1):
            current_item = data[current_index]
            next_item = data[current_index + 1]

            # If the current item is greater than the next, swap them
            if current_item[key] > next_item[key]:
                data[current_index], data[current_index + 1] = next_item, current_item

    return data

people = [
    {"name": "Fred", "age": 18},
    {"name": "Kaloki", "age": 22},
    {"name": "Charles", "age": 20},
    {"name": "Mutinda", "age": 19}
]

sorted_people = sort_list_of_dicts_manual(people, "age")
for person in sorted_people:
    print(person)
```

The AI clearly wins on development speed for this common task. It's like having a coding buddy who instantly knows all the Python tricks.

But Here's Where It Gets Interesting...

My manual version (bubble sort) has O(n^2) time complexity, which is way less efficient than the AI's O (n log n) approach. The AI actually chose a **better algorithm** than I did!

For small lists like our 4 people, you wouldn't notice the difference. But if we had to sort 1,000 records? My version would be painfully slow while the AI's would still be speedy.

The Trade-offs

AI Pros:

- Lightning fast to implement
- Uses optimized, built-in functions
- Less code = less potential for bugs

AI Cons:

- Doesn't explain *why* it chose that approach
- If I just copy-pasted without understanding, I wouldn't learn anything
- Might not handle edge cases without specific prompting

Manual Pros:

- I actually understand how the sorting works under the hood
- Good learning exercise for algorithm fundamentals
- Full control over every aspect

Manual Cons:

- Way more time-consuming
- Higher chance of introducing bugs
- Less efficient algorithm choice

The Verdict

For real-world work, I'd 100% use the AI version - it's cleaner, faster, and more efficient. But for learning purposes, writing it manually was valuable because now I truly understand how list sorting works.

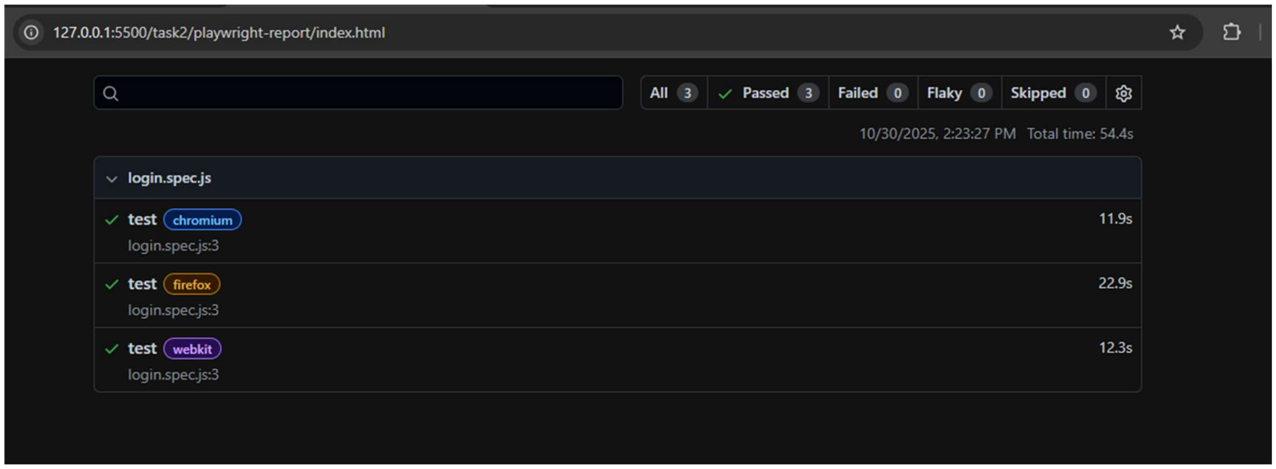
The AI is like a super-smart assistant: amazing for productivity, but you still need to understand the fundamentals to use it effectively and catch when it might be wrong!

TASK 2: AI-Augmented Test Automation

The integration of artificial intelligence within test automation frameworks, such as Playwright's code generation capability, substantially enhances software verification processes. By transcribing manual user interactions into executable test scripts in real-time, AI eliminates the traditional overhead of manual test script composition. This automation accelerates test development and ensures consistent replication of user workflows.

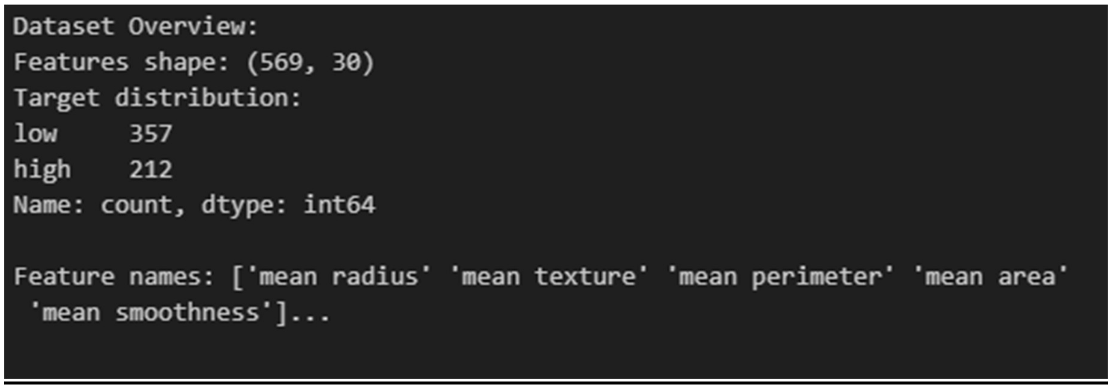
Compared to purely manual testing, AI-driven automation provides superior test coverage through systematic execution of both valid and invalid input scenarios. The AI intelligently employs robust locator strategies, such as role-based selection, which improve test resilience against user interface modifications. This method reduces human error and enables frequent regression testing, which is often impractical with manual efforts.

Consequently, AI in test automation represents a significant evolution in quality assurance, shifting testing from a labor-intensive, repetitive task to an efficient, scalable, and reliable engineering practice. It enables development teams to maintain high software quality while optimizing resource allocation.

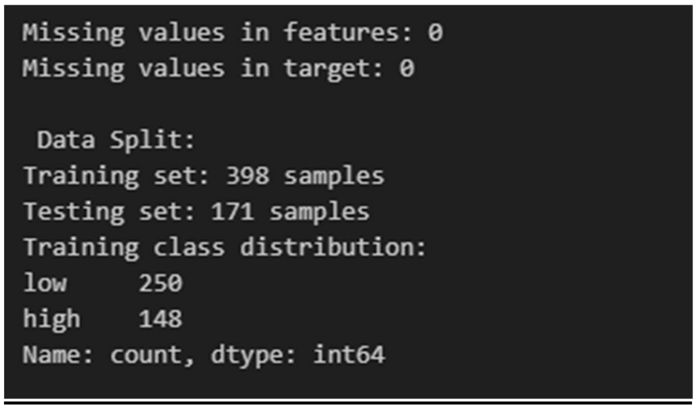


```
127.0.0.1:5500/task2/playwright-report/index.html

All 3   Passed 3   Failed 0   Flaky 0   Skipped 0
10/30/2025, 2:23:27 PM   Total time: 54.4s

login.spec.js
✓ test chromium                                   11.9s
  login.spec.js:3
✓ test firefox                                    22.9s
  login.spec.js:3
✓ test webkit                                     12.3s
  login.spec.js:3
```

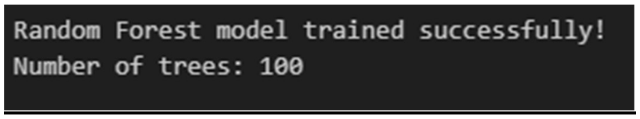
TASK 3 – Breast Cancer Prediction



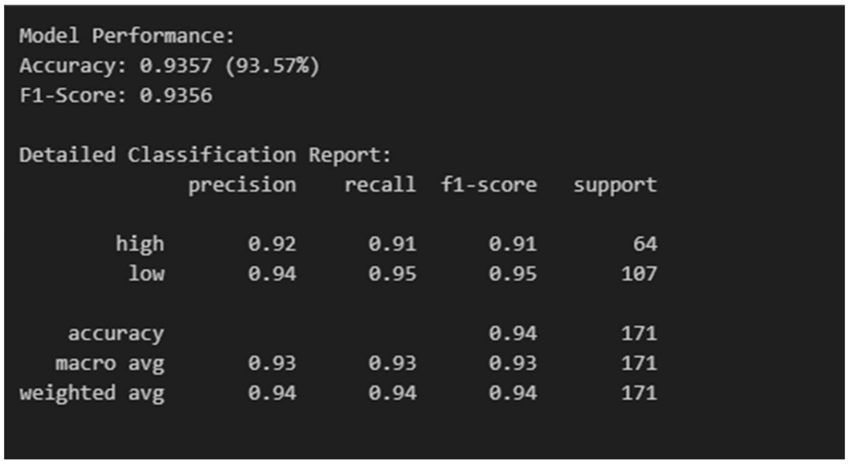
```
Dataset Overview:
Features shape: (569, 30)
Target distribution:
low     357
high    212
Name: count, dtype: int64

Feature names: ['mean radius' 'mean texture' 'mean perimeter' 'mean area'
 'mean smoothness']...
```



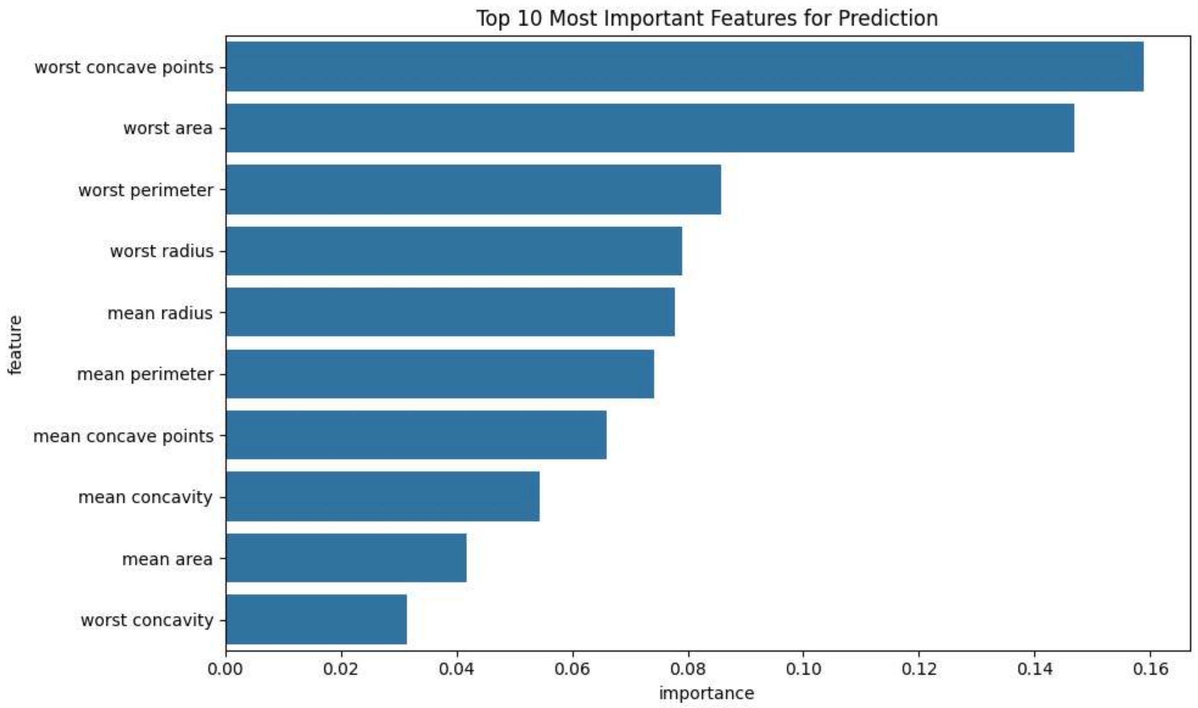
```
Missing values in features: 0
Missing values in target: 0

 Data Split:
Training set: 398 samples
Testing set: 171 samples
Training class distribution:
low     250
high    148
Name: count, dtype: int64
```

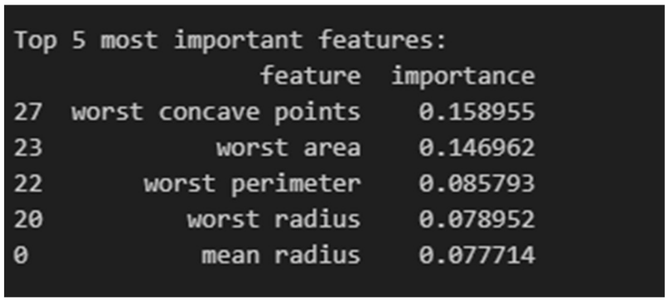


```
Random Forest model trained successfully!
Number of trees: 100
```



```
Model Performance:
Accuracy: 0.9357 (93.57%)
F1-Score: 0.9356

Detailed Classification Report:
              precision    recall  f1-score   support

        high       0.92      0.91      0.91        64
         low       0.94      0.95      0.95       107

    accuracy                           0.94       171
   macro avg       0.93      0.93      0.93       171
weighted avg       0.94      0.94      0.94       171
```





```
Top 5 most important features:
                 feature  importance
27  worst concave points    0.158955
23            worst area    0.146962
22       worst perimeter    0.085793
20          worst radius    0.078952
0            mean radius    0.077714
```

Part 3: Ethical Reflection

Based from the predictive model from Task 3

While our breast cancer prediction model achieved 93.57% accuracy, several ethical considerations emerge when considering real-world deployment. The dataset may underrepresent certain demographic groups - such as specific age ranges, ethnicities, or geographic regions - which could lead to biased predictions for underrepresented populations. Our model heavily weights features like 'worst concave points' and 'worst area,' but if these measurements systematically vary across demographic groups, the model could perpetuate healthcare disparities.

Tools like IBM AI Fairness 360 could help address these concerns by quantifying bias through metrics like demographic parity and equal opportunity difference. For instance, AIF360 could reweight the training data to ensure balanced representation or apply preprocessing techniques to remove biased patterns. In a clinical setting, such fairness checks are crucial to ensure the model doesn't disadvantage already marginalized patient groups.

Ultimately, high accuracy alone isn't sufficient; we must ensure our AI systems promote equitable healthcare outcomes across all patient demographics.