Week 4 Assignment: AI in Software Engineering
Name: Rodney Anyira
Date: 12/11/2025

Part 1: Theoretical Analysis (30%)

1. Short Answer Questions

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

AI code tools like GitHub Copilot reduce development time in two main ways. First, they **accelerate coding** by autocompleting entire blocks of boilerplate code, complex functions, and tests, saving developers from typing. Second, they **reduce cognitive load** by acting as an "in-editor" search engine, suggesting syntax and patterns without the developer needing to leave their IDE to search for documentation.

Limitations:
* **Accuracy:** The AI can confidently suggest code that is subtly wrong, buggy, or inefficient.
* **Security:** It can be trained on public code containing security vulnerabilities and suggest those flawed patterns.
* **Copyright:** It may generate code that is a direct copy of a repository with a restrictive license, creating legal risks.
* **Over-reliance:** Junior developers may become reliant on the tool and fail to learn the underlying problem-solving principles.

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

* **Supervised Learning:** This method requires a labeled dataset where code commits or patterns are marked as "buggy" or "clean." It is trained to recognize the specific features of known bugs.
    * **Use Case:** Excellent for **predicting recurring bug types** (e.g., null pointer exceptions, common logic errors) based on historical data.
    * **Limitation:** It cannot detect new, novel, or "zero-day" bugs it has never seen before.

* **Unsupervised Learning:** This method uses unlabeled data. It learns the "normal" behavior or patterns of a healthy codebase.

```
    * **Use Case:** Excellent for **anomaly detection**. It can flag code that "looks weird" or deviates from established patterns, even if that pattern isn't a known bug. This can help find new and unusual bugs.
    * **Limitation:** It can have a higher rate of false positives, flagging code that is just "different" but not actually buggy.

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

Bias mitigation is critical because a biased AI can lead to **user alienation, revenue loss, and echo chambers.** If a personalization model (e.g., for a movie service) is trained on biased data, it might learn to only recommend action films to men. This creates two failures: 1) It alienates female users who like action films (bad UX), and 2) The company loses potential revenue from that user segment. It also creates a "filter bubble" that prevents users from discovering new interests, leading to a stale and less engaging experience.

### 2. Case Study Analysis

**Article:** *AI in DevOps: Automating Deployment Pipelines.*

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

AIOps (AI for IT Operations) improves deployment efficiency by moving from a reactive to a **proactive and predictive** model. It analyzes massive amounts of data (logs, metrics) to automate decisions and prevent failures before they happen.

* **Example 1: Predictive Failure Analysis.** Before a deployment, AIOps can analyze the new code changes and compare them to historical data. It can predict the *risk* of the deployment failing or causing a performance drop. If the risk is high, it can automatically halt the pipeline, preventing a bad deployment from ever reaching users.

* **Example 2: Intelligent Anomaly Detection & Root Cause Analysis.** After deployment, AIOps provides real-time monitoring that is smarter than simple alarms. It can detect subtle "anomalies" (e.g., memory slowly leaking) that a static threshold would miss. When a failure *does* happen, it correlates logs from dozens of services to instantly pinpoint the root cause, reducing troubleshooting time from hours to minutes.

---

## Part 2: Practical Implementation (60%)
```

```markdown
### Task 1: AI-Powered Code Completion

* **Manual Code:** `part2_task1_sorting/manual_sort.py`
* **AI-Suggested Code:** `part2_task1_sorting/ai_sort.py`

**Analysis (200 words):**
This analysis compares my manual Python function for sorting a list of
dictionaries against the version suggested by GitHub Copilot. The goal was to
evaluate "efficiency" in its three forms: developer, computational, and
maintenance.

In this instance, both my manual implementation and the AI-generated code were
identical. Both used the optimal Pythonic solution: `sorted(data, key=lambda x:
x[sort_key])`.

1.  **Developer Efficiency:** The AI approach was massively more efficient. After
typing the function name, Copilot generated the entire function body in a single
tab press. This saves significant time in writing boilerplate code and eliminates
the cognitive load of recalling specific syntax.

2.  **Computational Efficiency:** Both versions are identically efficient. They
use Python's built-in Timsort algorithm, which has an average and worst-case time
complexity of $O(n \log n)$, the standard for efficient sorting.

3.  **Maintenance Efficiency:** As the code is the same, maintenance is
identical. The solution is clean, standard, and easily readable by any Python
developer.

In conclusion, for common tasks like this, AI tools like Copilot provide a clear
win by maximizing developer efficiency without sacrificing computational
performance or code quality.

### Task 2: Automated Testing with AI

* **Test Script:** `part2_task2_testing/run_test.py`
* **Test Results Screenshot:**
    ![Screenshot of VS Code terminal showing the test script
passed](test_results.png)

**Summary (150 words):**
For this task, I automated a login test case using a Python script with the
**Selenium WebDriver** library. The script (`run_test.py`) automatically opens a
Chrome browser, navigates to the login page, enters valid credentials (`tomsmith`
/ `SuperSecretPassword!`), and clicks the login button. It then confirms a
successful login by finding the success message on the resulting page.

This script itself represents a traditional, brittle test. The assignment's AI
theme comes in by improving **test resilience**. An AI-powered tool (like Testim
or the AI features in modern recorders) would not rely on a fixed
`ID="username"`. It would understand the element as "the text box labeled
'Username'". If a developer changed the ID, the AI would still find the box,
"self-healing" the test and preventing it from failing. This AI-driven resilience
is what drastically reduces test maintenance.

### Task 3: Predictive Analytics for Resource Allocation

* **Jupyter Notebook:** `part2_task3_model/Part2_Task3_Predictive_Model.ipynb`

**Performance Metrics:**
The Random Forest model was trained to predict "issue priority" (mapped from the
dataset's binary target).
* **Accuracy: 95.91%**
* **F1-Score: 95.90%**

These metrics indicate that the model is extremely effective at classifying the
data based on the provided features.

---

## Part 3: Ethical Reflection (10%)

**Prompt:** Your predictive model from Task 3 is deployed. Discuss potential
biases and how fairness tools could address them.

The model I built (per the assignment instructions) used the Breast Cancer
dataset to predict "issue priority." This is an artificial scenario, but in a
real-world deployment, the potential for bias is high.

1.  **Potential Biases:** If this were a *real* dataset of software issues, it
could be heavily biased.
    * **Reporter Bias:** Issues reported by senior developers or managers might
be historically labeled "High Priority" more often, not because of their
*technical* severity, but because of *who* reported them. The model would learn
this bias and unfairly down-prioritize issues from junior developers or external
users.
    * **Project Bias:** The model might be trained on data from only one project
(e.g., a web app). When deployed to work on a different project (e.g., a mobile
app), its definition of "priority" might be completely wrong, leading to
inaccurate predictions.
    * **Underrepresentation:** If a specific team (e.g., the "documentation
team") has very few historical issues, the model may be unable to accurately
prioritize their problems, effectively ignoring their needs.

2.  **Addressing Bias with Tools:** Tools like **IBM AI Fairness 360 (AIF360)**
can help. First, we would use it to *measure* bias. We could perform a "Disparate
Impact" analysis to see if the model's "high priority" predictions are unfairly
distributed across different groups (e.g., 'senior' vs 'junior' reporters). If
bias is found, AIF360 provides "Reweighing" algorithms. This technique would
assign a higher "weight" to the underrepresented group's data (e.g., the junior
dev's issues) during training, forcing the model to pay more attention to them
and learn their patterns, resulting in a fairer, more balanced model.

---

## Bonus Task: Innovation Challenge

**Proposal: AI-Powered "Docu-Mentor"**

1.  **Problem:** Software documentation is almost always outdated, incomplete, or
non-existent. This makes onboarding new engineers slow and wastes senior engineer
time answering repetitive questions.
2.  **Tool Purpose:** "Docu-Mentor" is an AI tool that solves this by
**automatically generating and maintaining documentation** that is context-aware.
3.  **Workflow:**
    * **Ingestion:** The tool continuously scans the entire codebase (e.g.,
Python, JavaScript files) and all project-related-text (Jira tickets, Slack
conversations, past Pull Requests).
    * **Vectorization:** It uses a Large Language Model (LLM) to turn all this
code and text into a "vector database" — a knowledge base that understands the
*meaning* and *relationships* between different parts of the project.
    * **Interaction:** A developer can now interact with this knowledge base.
        * **Automatic Docstrings:** A developer can right-click a complex
function and ask the tool to "Generate documentation." It will write a clear
docstring explaining *what* the function does, *what* its parameters are, and
*why* it exists (based on a Jira ticket it found).
        * **Q&A Chatbot:** A new engineer can open a chat window in their IDE and
ask, "How do I process a payment?" The tool will provide a step-by-step answer
with code snippets, referencing the `PaymentProcessor.py` and `StripeAPI.js`
files, and even link to the original Pull Request where the feature was
discussed.
```

```
4.  **Impact:** This tool would drastically cut developer onboarding time, reduce
time spent on code-archeology, and create high-quality, "living" documentation
that never goes stale, making the entire engineering team more efficient.
```