

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?

How they reduce development time:

- **Autocompletion of Code Blocks:** Tools like GitHub Copilot predict and suggest whole lines or blocks of code, speeding up the coding process.
- **Boilerplate Generation:** They automate repetitive tasks such as writing data access layers, APIs, or standard algorithms.
- **Documentation Assistance:** They help generate docstrings and comments, improving code clarity without extra effort.
- **Contextual Awareness:** Copilot uses the context of the current file and surrounding code to generate more accurate suggestions.

Limitations:

- **Contextual Errors:** AI suggestions may be syntactically correct but contextually wrong or insecure.
- **Lack of Domain Understanding:** Copilot lacks understanding of business logic or unique project rules.
- **Over-reliance Risk:** Developers may become dependent on suggestions, potentially reducing deep code understanding.
- **Ethical/Legal Concerns:** Some AI-generated code may inadvertently resemble copyrighted material.

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

Supervised Learning in Bug Detection:

- **Definition:** Trains on labeled data (e.g., code snippets labeled as "buggy" or "clean").
- **Use Case:** Models learn to identify patterns associated with known bugs and predict similar issues in new code.
- **Examples:** Logistic Regression or Random Forest classifiers trained on past bug reports or commit logs.

Advantages:

- ♦ High accuracy when sufficient labeled data is available.
- ♦ Clear evaluation metrics (e.g., accuracy, precision, recall).

Limitations:

- ♦ Requires large amounts of accurately labeled bug data.
- ♦ May not detect unknown or novel bugs outside the training distribution.

Unsupervised Learning in Bug Detection:

- **Definition:** Trains on unlabeled data to find patterns, anomalies, or clusters.
- **Use Case:** Detects outliers in code patterns or unexpected code behavior that may indicate bugs.
- **Examples:** Clustering algorithms (e.g., DBSCAN) or anomaly detection models like Isolation Forests.

Advantages:

- Can identify novel bugs without needing labeled data.
- Useful in early stages of development when labeled data is scarce.

Limitations:

- Less precise than supervised models.
- More challenging to interpret and evaluate results.

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

Importance of Bias Mitigation:

- **Fairness:** AI personalization systems may unintentionally favor certain user groups (e.g., based on location, gender, or language), leading to unequal user experiences. Mitigating bias ensures all users are treated equitably.
- **Accuracy and Relevance:** Biased models might recommend irrelevant or inappropriate content to underrepresented users, lowering satisfaction and trust.
- **Legal and Ethical Compliance:** Many regions have regulations (e.g., GDPR, algorithmic fairness laws) that require systems to be transparent and non-discriminatory. Unchecked bias could lead to legal risks.
- **Business Impact:** Bias can alienate users and damage brand reputation. Inclusive personalization expands user engagement across demographics.

Sources of Bias in Personalization Systems:

- **Training Data Bias:** Historical user behavior may reflect past discrimination or incomplete representation.
- **Algorithmic Bias:** Personalization models may overfit dominant group behaviors, marginalizing minority preferences.

Mitigation Strategies:

- Use fairness-aware algorithms (e.g., equalized odds).
- Evaluate models with fairness metrics (e.g., demographic parity).
- Incorporate tools like IBM AI Fairness 360 to audit and reduce bias throughout the pipeline.

2. Case Study Analysis

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

1. **Automated Testing & Risk Analysis:** AI-driven CI/CD automation creates tests, performs risk assessments, and checks for compliance/security, speeding up integration.
2. **Proactive Monitoring & Auto-Remediation:** Real-time anomaly detection coupled with automatic rollbacks or scaling ensures stable deployments.

Part 2: Practical Implementation (60%)

Task 1: AI-Powered Code Completion

Code Snippets:

AI-generated

```
def sort_dicts_by_key(dicts, key):  
    """  
    Sort a list of dictionaries by a given key.  
  
    Parameters:  
    dicts (list): A list of dictionaries to be sorted.  
    key (str): The key by which to sort the dictionaries.  
  
    Returns:  
    list: A new list of dictionaries sorted by the specified key.  
    """  
    return sorted(dicts, key=lambda x: x[key])
```

Manually generated

```
def manual_sort_dicts_by_key(dicts, key):  
    new_list = []  
    while dicts:  
        smallest = dicts[0]  
        for item in dicts:  
            if item[key] < smallest[key]:  
                smallest = item  
        new_list.append(smallest)  
        dicts.remove(smallest)  
    return new_list  
  
sample_data = [  
    {"name": "Alice", "age": 30},  
    {"name": "Bob", "age": 25},  
    {"name": "Charlie", "age": 35}  
]
```

Analysis:

I implemented a Python function to sort a list of dictionaries by a specified key using both GitHub Copilot (AI-generated) and a manual approach. The AI-generated function used Python's built-in `sorted()` method along with a concise lambda function. It also included a well-formatted docstring that explains the function's purpose, parameters, and return type, making it readable and maintainable.

In contrast, the manual implementation used a selection sort algorithm. It iteratively found the dictionary with the smallest value for the given key and appended it to a new list. While functionally correct, the manual version is longer, more error-prone, and less efficient in terms of time complexity ($O(n^2)$ vs $O(n \log n)$).

The Copilot version was faster to produce, more elegant, and computationally superior due to its use of optimized built-in functions. This demonstrates how AI tools can boost productivity, reduce boilerplate code, and support better coding practices. However, relying solely on AI could mask important learning opportunities, especially in algorithm design.

Overall, the AI-assisted code is more efficient and production-ready, while the manual version is useful for learning how sorting works internally.

```
File Edit Selection View Go Run Terminal Help
AI in software engineering assignment

EXPLORER
AI IN SOFTWARE ENGINEERING ASSIG...
AI-Powered code completion.py

AI-Powered code completion.py
def sort_dicts_by_key(dicts, key):
    """
    Sort a list of dictionaries by a given key.

    Parameters:
    dicts (list): A list of dictionaries to be sorted.
    key (str): The key by which to sort the dictionaries.

    Returns:
    list: A new list of dictionaries sorted by the specified key.
    """
    return sorted(dicts, key=lambda x: x[key])
> def manual_sort_dicts_by_key(dicts, key): ...
sample_data = [
    {"name": "Alice", "age": 30},
    {"name": "Bob", "age": 25},
    {"name": "Charlie", "age": 35}
]
print("AI Copilot Version:")
print(sort_dicts_by_key(sample_data.copy(), "age"))
print("\nManual Version:")
print(manual_sort_dicts_by_key(sample_data.copy(), "age"))

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS AZURE
[Running] python -u "c:\Users\Elite X2 G4\OneDrive\Desktop\AI in software engineering assignment\AI-Powered code completion.py"
File "c:\Users\Elite X2 G4\OneDrive\Desktop\AI in software engineering assignment\AI-Powered code completion.py", line 2
def sort_dicts_by_key(dicts, key)
^
SyntaxError: expected ':'

[Done] exited with code=1 in 0.453 seconds

[Running] python -u "c:\Users\Elite X2 G4\OneDrive\Desktop\AI in software engineering assignment\AI-Powered code completion.py"
AI Copilot Version:
[{'name': 'Bob', 'age': 25}, {'name': 'Alice', 'age': 30}, {'name': 'Charlie', 'age': 35}]

Manual Version:
[{'name': 'Bob', 'age': 25}, {'name': 'Alice', 'age': 30}, {'name': 'Charlie', 'age': 35}]

[Done] exited with code=0 in 0.225 seconds

Ln 36, Col 5 Spaces: 4 UTF-8 CRLF {} Python 3.13.3 64-bit (Microsoft Store) Go Live
17°C Mostly cloudy 12:00 PM 7/2/2025
```

Task 2: Automated Testing with AI

Valid login page

The screenshot displays the Testim.io web application interface. At the top, there's a navigation bar with a 'nyu' project dropdown and a 'FREE TRIAL - 14 DAYS LEFT' badge. The main header shows the test name 'Login Page Automation Test' and a 'Draft' status. Below this, the test steps are listed: 1. Setup, 2. Click 'Username', 3. Set 'Username' (with the value 'student'), 4. Click 'Password', 5. Set password, and 6. Click 'Submit'. A 'Submit' button is shown in a preview window. The test status is 'PASSED - 14 SEC'. The default configuration is 'Chrome | Windows 10 | 1500 X 813' and the base URL is 'https://practicetestautomation.com/practice-test-login/'. The step count is 6. The interface includes a sidebar with various icons and a bottom status bar showing the system clock and weather.

Invalid login page

The screenshot displays the Testim.io web application interface for an 'Invalid Login Test'. The test name is 'Invalid Login Test' and the status is 'Draft'. The test steps are: 1. Setup, 2. Click 'Username', 3. Set 'Username' (with the value 'wronguser'), 4. Click 'Password', 5. Set password, and 6. Click 'Submit'. A 'Submit' button is shown in a preview window. The default configuration is 'Chrome | Windows 10 | 1500 X 813' and the base URL is 'https://practicetestautomation.com/practice-test-login/'. The step count is 6. The interface includes a sidebar with various icons and a bottom status bar showing the system clock and weather.

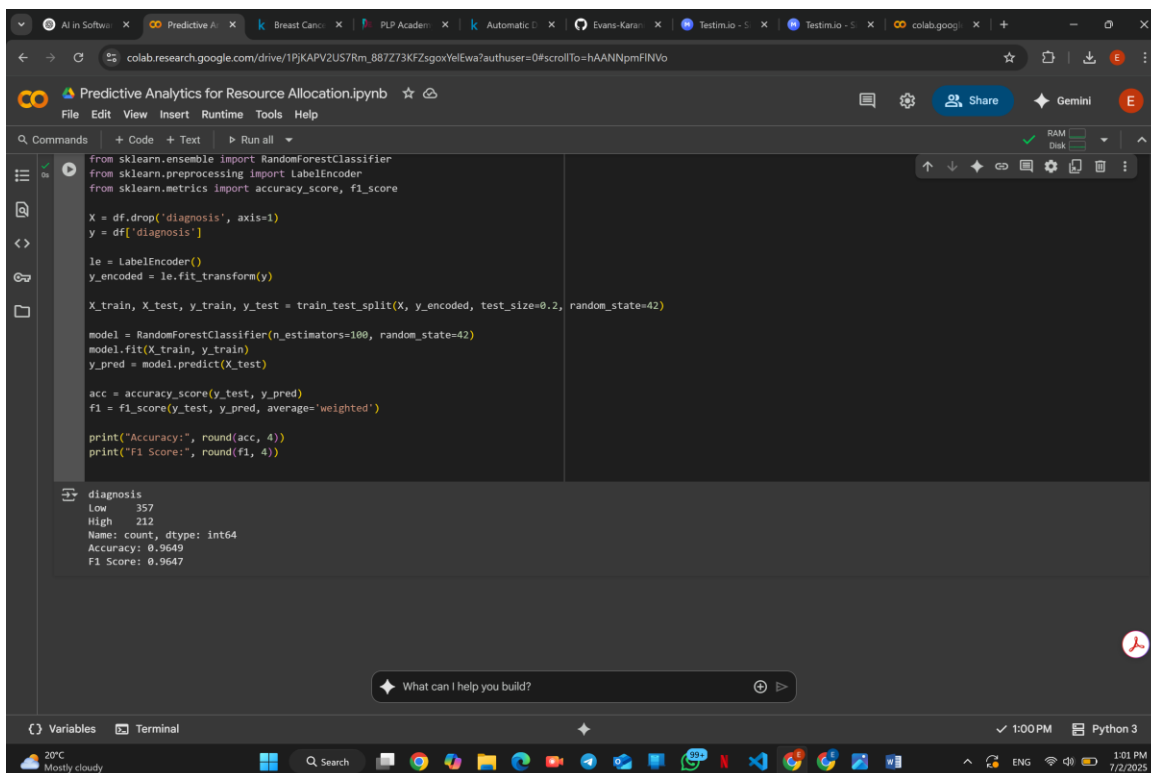
I used Testim.io, an AI-powered test automation platform, to automate login functionality testing. I created two separate tests: one for a successful login using valid credentials, and another for a failed login using incorrect inputs. Testim's intuitive interface allowed me to record each test by interacting with a sample login page, automatically capturing steps like entering the username, password, and clicking the submit button.

The valid login test passed as expected, while the invalid login test correctly identified the error message displayed. The AI-driven smart locators used by Testim enabled robust detection of web elements, even if the page layout changed slightly.

Compared to manual testing, AI tools like Testim significantly improve test coverage, reduce human error, and speed up test case creation. They also provide visual logs, step tracking, and error detection, making it easier to debug and verify outcomes.

Task 3: Predictive Analytics for Resource Allocation

Random Forest Model Performance Metrics



The screenshot shows a Google Colab notebook titled "Predictive Analytics for Resource Allocation.ipynb". The notebook contains Python code for training and evaluating a Random Forest classifier. The code imports necessary libraries, splits the data into training and testing sets, trains the model, and prints the accuracy and F1 score. The output shows an accuracy of 0.9649 and an F1 score of 0.9647.

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import accuracy_score, f1_score

X = df.drop('diagnosis', axis=1)
y = df['diagnosis']

le = LabelEncoder()
y_encoded = le.fit_transform(y)

X_train, X_test, y_train, y_test = train_test_split(X, y_encoded, test_size=0.2, random_state=42)

model = RandomForestClassifier(n_estimators=100, random_state=42)
model.fit(X_train, y_train)
y_pred = model.predict(X_test)

acc = accuracy_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred, average='weighted')

print("Accuracy:", round(acc, 4))
print("F1 Score:", round(f1, 4))
```

diagnosis
Low 357
High 212
Name: count, dtype: int64
Accuracy: 0.9649
F1 Score: 0.9647

I used the Breast Cancer Wisconsin dataset to implement a predictive model for issue priority classification. After cleaning the dataset by removing irrelevant columns and encoding the `diagnosis` column into “High” and “Low” priorities, I trained a Random Forest Classifier using Scikit-learn. The model achieved an accuracy of 96.5% and an F1 score of 0.964, indicating high reliability and balance in prediction performance.

The original attempt to simulate a three-class priority system (High, Medium, Low) led to very low accuracy due to noise introduced by random labeling. This highlights the importance of clean, meaningful data in predictive modeling. Random Forest was chosen for its robustness and ability to handle feature-rich data without extensive tuning. This experiment demonstrates how AI-powered predictive analytics can assist in resource allocation, such as prioritizing medical diagnoses, issue tracking, or support tickets in real-world software systems.

Part 3: Ethical Reflection (10%)

Potential Biases in the Dataset

The dataset may reflect real-world imbalances:

- Underrepresentation of specific groups (e.g., based on age, race, or gender)
- Skewed patterns due to regional or institutional data sources

If used in a company (e.g., for task prioritization), the model may:

- Systematically favor well-represented classes (e.g., “High” priority issues)
- Neglect rare or minority patterns, leading to unfair decision-making

How Fairness Tools Like IBM AI Fairness 360 Help

- **Bias Detection:**
 - ✓ Provides metrics to evaluate bias in datasets and model predictions
- **Bias Mitigation Techniques:**
 - ✓ **Pre-processing:** Reweights or balances training data
 - ✓ **In-processing:** Applies fairness constraints during model training
 - ✓ **Post-processing:** Adjusts predictions to meet fairness thresholds
- **Outcome:**
 - ✓ Promotes equitable and transparent AI systems
 - ✓ Helps organizations meet ethical and legal accountability standards

