task1_2

July 14, 2025

[1]:

```
Sorted by age (ascending): [{'name': 'Bob', 'age': 25, 'city': 'London'},
{'name': 'Alice', 'age': 30, 'city': 'New York'}, {'name': 'Charlie', 'age': 35,
'city': 'Paris'}]
Sorted by name (descending): [{'name': 'Charlie', 'age': 35, 'city': 'Paris'},
{'name': 'Bob', 'age': 25, 'city': 'London'}, {'name': 'Alice', 'age': 30,
'city': 'New York'}]
```

[ ]:

```
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-powered code generation tools like GitHub Copilot assist developers by␣
 ↪auto-suggesting entire code blocks, functions, or syntax based on context.␣
 ↪They help reduce development time by:

Completing repetitive code faster.

Reducing time spent on boilerplate or common patterns.

Suggesting alternative approaches or syntax for tasks.

Assisting with unfamiliar libraries or functions.

However, limitations include:

Lack of understanding of project context: Suggestions may not align with␣
 ↪project architecture or logic.

Security and quality risks: Copilot may suggest outdated or vulnerable code␣
 ↪from its training data.

Code bias: Generated suggestions can reflect biased or suboptimal patterns from␣
 ↪public repositories.
```

```
Over-reliance: Developers might accept suggestions without fully understanding␣
 ↪the underlying logic, affecting code quality.

Q2: Compare supervised and unsupervised learning in the context of automated␣
 ↪bug detection?
In automated bug detection:

Supervised Learning:

Trained on labeled datasets (e.g., bug vs. clean code).

Learns to classify new code based on previously seen patterns.

Strengths: High accuracy for known bug types.

Limitations: Requires large labeled datasets and cannot detect new/unseen bugs.

Unsupervised Learning:

Finds anomalies or clusters in code without labels.

Can detect outliers that may represent unknown bugs.

Strengths: Works with unlabeled data; can flag unusual patterns.

Limitations: Higher false positives; harder to interpret results.

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

Bias in AI personalization can:

Reinforce stereotypes (e.g., showing different features based on gender or␣
 ↪race).

Exclude minority groups from certain experiences or features.

Distort recommendations by overrepresenting popular trends while ignoring niche␣
 ↪interests.

Bias mitigation ensures:

Inclusive experiences across diverse users.

Fair recommendations based on behavior, not demographics.

Legal and ethical compliance (e.g., GDPR, anti-discrimination laws).
```

```
Tools like fairness indicators, diverse training data, and ethical audits help␣
 ↪minimize such biases.

2 Case Study Analysis

Read the article: AI in DevOps: Automating Deployment Pipelines.
Answer: How does AIOps improve software deployment efficiency? Provide two␣
 ↪examples?.


AIOps (Artificial Intelligence for IT Operations) enhances deployment␣
 ↪efficiency by:

Monitoring and analyzing logs, metrics, and traces using machine learning.

Automating decision-making to detect and fix issues faster.

Examples:

Automated Failure Detection: AIOps systems detect anomalies in real-time logs␣
 ↪during deployment and trigger rollback automatically, preventing downtime.

Predictive Scaling: AIOps predicts peak load times and auto-scales␣
 ↪infrastructure to meet demand without manual intervention.

These capabilities reduce downtime, improve user satisfaction, and increase␣
 ↪developer productivity.
```

[1]:
```
def sort_list_of_dictionaries(list_of_dicts, key_to_sort_by, reverse=False):
    """
    Sorts a list of dictionaries by a specific key.

    Args:
        list_of_dicts (list): The list of dictionaries to be sorted.
        key_to_sort_by (str): The key within each dictionary to sort by.
        reverse (bool, optional): If True, sort in descending order. Defaults␣
 ↪to False.

    Returns:
        list: A new sorted list of dictionaries.
    """
    return sorted(list_of_dicts, key=lambda d: d[key_to_sort_by],␣
 ↪reverse=reverse)
```

```
# Example Usage:
data = [
    {"name": "Alice", "age": 30, "city": "New York"},
    {"name": "Bob", "age": 25, "city": "London"},
    {"name": "Charlie", "age": 35, "city": "Paris"}
]

# Sort by 'age' in ascending order
sorted_by_age = sort_list_of_dictionaries(data, "age")
print(f"Sorted by age (ascending): {sorted_by_age}")

# Sort by 'name' in descending order
sorted_by_name_desc = sort_list_of_dictionaries(data, "name", reverse=True)
print(f"Sorted by name (descending): {sorted_by_name_desc}")
```

```
Sorted by age (ascending): [{'name': 'Bob', 'age': 25, 'city': 'London'},
{'name': 'Alice', 'age': 30, 'city': 'New York'}, {'name': 'Charlie', 'age': 35,
'city': 'Paris'}]
Sorted by name (descending): [{'name': 'Charlie', 'age': 35, 'city': 'Paris'},
{'name': 'Bob', 'age': 25, 'city': 'London'}, {'name': 'Alice', 'age': 30,
'city': 'New York'}]
```

[2]:
```
def sort_dicts_by_key(dict_list, sort_key, reverse=False):
    """
    Sorts a list of dictionaries by a specific key.

    Args:
        dict_list (list): List of dictionaries to sort.
        sort_key (str): The key to sort the dictionaries by.
        reverse (bool): If True, sorts in descending order. Default is False.

    Returns:
        list: A new list of dictionaries sorted by the specified key.
    """
    return sorted(dict_list, key=lambda x: x.get(sort_key, None),␣
 ↪reverse=reverse)
```

[3]:
```
data = [
    {'name': 'Alice', 'age': 30},
    {'name': 'Bob', 'age': 25},
    {'name': 'Charlie', 'age': 35}
]

sorted_data = sort_dicts_by_key(data, 'age')
print(sorted_data)
# Output: [{'name': 'Bob', 'age': 25}, {'name': 'Alice', 'age': 30}, {'name':␣
 ↪'Charlie', 'age': 35}]
```

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

[ ]:
```
Characteristics of manual:

Uses lambda d: d[key_to_sort_by] as the sort key.
Assumes every dictionary contains the key key_to_sort_by.
Will raise a KeyError if a dictionary is missing the key.

Characteristics of codepilot suggested ai:

Uses lambda x: x.get(sort_key, None) as the sort key.
Handles missing keys gracefully by defaulting to None.
No KeyError if a dictionary is missing the sort key.

Efficiency Analysis
Performance:
Both versions use Python's built-in sorted() with a lambda function as the key.␣
 ↪The only difference is the method of accessing the dictionary value:

d[key_to_sort_by] (attached code): Direct dictionary access.
x.get(sort_key, None) (suggested code): Uses dict.get, which adds a tiny␣
 ↪overhead for the method call and default handling.
In practice:

The direct access d[key_to_sort_by] is marginally faster since it does not␣
 ↪check for a default.
d.get(key, None) is safer as it avoids KeyError but is infinitesimally slower␣
 ↪due to the method call (the difference is negligible for most use cases).
Example Timing (for large lists):

The difference is typically less than 1% in most real-world scenarios.

Robustness and Usability
Manual code is more efficient (faster), but less robust: it breaks if a key is␣
 ↪missing.
Codepilot code is slightly less efficient (slower by a negligible amount) but␣
 ↪more robust: it safely handles missing keys.
Conclusion & Recommendation
If you are certain every dictionary will always contain the sort key, manual␣
 ↪code is marginally more efficient.
If there is any chance a dictionary may be missing the sort key, or you want␣
 ↪safer code, my version is preferable.

Documentation Summary
```

```
While the output is similar, the benefit of Copilot is evident in speed-it␣
 ↪suggested the full implementation instantly after typing the function name.␣
 ↪For beginner programmers, this can significantly reduce coding time.
However, Copilot lacks context about the broader application. If error handling␣
 ↪or custom logic is needed, a human touch is still essential. The manual␣
 ↪implementation required deeper understanding and provided learning value.
In conclusion, Copilot improves productivity in repetitive or simple tasks, but␣
 ↪it's best used alongside manual coding and review to ensure robustness and␣
 ↪correctness.

The manual code is marginally more efficient due to direct key access, but␣
 ↪lacks robustness as it raises an error for missing keys. The suggested code␣
 ↪is slightly less efficient but more robust, safely handling missing keys by␣
 ↪using .get(). For most use cases, the performance difference is negligible,␣
 ↪so robustness should typically be preferred.
```

[ ]: