Ht no:2403A52154 Batch:06

Ai Assisted coding

Lab exam\_3

Set E5

Q1:  
Scenario: In the Finance sector, a company faces a challenge related to data structures with ai.  
Task: Use AI-assisted tools to solve a problem involving data structures with ai in this context.  
Deliverables: Submit the source code, explanation of AI assistance used, and sample output.

**Concise Explanation of the Solution**

The solution uses an **AI-inspired data structure** to solve a common financial problem: **Portfolio Optimization** by efficiently identifying the top $K$ investment strategies based on their **Sharpe Ratio** (risk-adjusted return).

The core data structure used is a **Max-Heap (Priority Queue)**, which is ideal for finding the "Top $K$" elements in $\mathcal{O}(N \log K)$ time, far more efficient than a full sort ($\mathcal{O}(N \log N)$) when dealing with a large number of potential strategies ($N$).

1. **InvestmentStrategy Class:** Defines a strategy with key metrics (return, volatility, cost) and includes a method to calculate the **Sharpe Ratio**.
2. **Max-Heap Simulation:** Python's heapq is a Min-Heap. To simulate a Max-Heap (to prioritize the largest Sharpe Ratios), the algorithm **negates** the Sharpe Ratio before pushing it onto the heap. This ensures the smallest element in the heap (the root) is the one with the highest *positive* Sharpe Ratio.
3. **Efficiency:** The function iterates through all strategies, maintaining a heap of size $K$. If a new strategy has a better Sharpe Ratio than the worst one currently in the heap, the worst is replaced, keeping the overall computation fast.

Code:

import heapq

from typing import List, Tuple

class InvestmentStrategy:

def \_\_init\_\_(self, name: str, expected\_return: float, volatility: float, cost: float):

self.name = name

self.expected\_return = expected\_return

self.volatility = volatility

self.cost = cost

def calculate\_sharpe\_ratio(self, risk\_free\_rate: float = 0.02) -> float:

if self.volatility == 0:

return float('inf')

return (self.expected\_return - risk\_free\_rate) / self.volatility

# Used for direct comparison in the heap (though the main function uses a tuple for priority)

def \_\_lt\_\_(self, other):

return self.calculate\_sharpe\_ratio() > other.calculate\_sharpe\_ratio()

def find\_top\_k\_strategies(strategies: List[InvestmentStrategy], k: int) -> List[Tuple[str, float]]:

# Min-Heap to store (-Sharpe\_Ratio, Strategy\_Object)

top\_k\_heap = []

for strategy in strategies:

sharpe = strategy.calculate\_sharpe\_ratio()

neg\_sharpe = -sharpe

if len(top\_k\_heap) < k:

# If heap isn't full, add it

heapq.heappush(top\_k\_heap, (neg\_sharpe, strategy))

else:

# Check if current strategy is better than the worst one in the heap (root)

worst\_neg\_sharpe\_in\_heap = top\_k\_heap[0][0]

if neg\_sharpe < worst\_neg\_sharpe\_in\_heap:

# Replace the worst with the better one

heapq.heapreplace(top\_k\_heap, (neg\_sharpe, strategy))

# Extract results

results = []

for neg\_sharpe, strategy in top\_k\_heap:

results.append((strategy.name, -neg\_sharpe))

results.sort(key=lambda x: x[1], reverse=True)

return results

# Example usage (input generation omitted for conciseness)

# top\_5 = find\_top\_k\_strategies(all\_strategies, 5)  
  
output:

--- Portfolio Optimization: Top 5 Strategies by Sharpe Ratio ---

--------------------------------------------------

Risk-Free Rate: 0.02

Strategy Name | Sharpe Ratio

--------------------------------------------------

Strategy\_33 | 2.1981

Strategy\_1 | 1.0972

Strategy\_4 | 0.9909

Strategy\_2 | 0.9308

Strategy\_3 | 0.9152

Q2:  
Scenario: In the Agriculture sector, a company faces a challenge related to data structures with ai.  
Task: Use AI-assisted tools to solve a problem involving data structures with ai in this context.  
Deliverables: Submit the source code, explanation of AI assistance used, and sample output.

**Concise Explanation of AI-Assisted Data Structure Solution 🌳**

The solution leverages **two specialized data structures** to handle the dual query criteria (Space and Time):

1. **Temporal Indexing (Pandas DatetimeIndex):** The data is indexed by **timestamp** using the highly optimized structures within the **pandas** library. This allows for near-instantaneous (logarithmic time complexity, $O(\log N)$) retrieval of all sensor readings within a specific time window.
2. **Spatial Indexing (BallTree/k-d tree concept):** The location data (Latitude/Longitude) is conceptually indexed using a tree structure (like a **BallTree** from scikit-learn). These multi-dimensional data structures are standard tools in AI/ML for efficient geometric searching. Although a bounding-box query is slightly simplified here, the use of this index avoids a slow linear scan of every data point to find the desired geographical area.

By applying these two efficient filters sequentially—first time, then space—the system quickly curates a small, relevant dataset for the downstream AI model to analyze crop health trends (e.g., mean soil moisture).

Code:

import pandas as pd

import numpy as np

from sklearn.neighbors import BallTree

np.random.seed(42)

N\_SENSORS = 500

START\_DATE = '2025-01-01'

END\_DATE = '2025-01-31'

latitudes = np.random.uniform(30.0, 31.0, N\_SENSORS)

longitudes = np.random.uniform(-100.0, -99.0, N\_SENSORS)

date\_range = pd.date\_range(start=START\_DATE, end=END\_DATE, freq='H')

data = []

for i in range(N\_SENSORS):

sensor\_data = pd.DataFrame({

'timestamp': date\_range,

'latitude': latitudes[i],

'longitude': longitudes[i],

'soil\_moisture': np.clip(

np.sin(np.arange(len(date\_range)) / 24) \* 15 + np.random.normal(50, 5, len(date\_range)),

30, 70

).astype(int)

})

data.append(sensor\_data)

df\_raw = pd.concat(data, ignore\_index=True)

# CREATE SPATIAL INDEX (BallTree)

coords\_rad = np.deg2rad(df\_raw[['latitude', 'longitude']].values)

spatial\_index = BallTree(coords\_rad, metric='haversine')

# CREATE TEMPORAL INDEX (Pandas DatetimeIndex)

df\_indexed = df\_raw.set\_index('timestamp').sort\_index()

# QUERY PARAMETERS

TARGET\_LAT\_MIN, TARGET\_LAT\_MAX = 30.5, 30.7

TARGET\_LON\_MIN, TARGET\_LON\_MAX = -99.5, -99.3

TARGET\_START\_TIME = pd.to\_datetime('2025-01-15')

TARGET\_END\_TIME = pd.to\_datetime('2025-01-20')

# STEP 1: FAST TEMPORAL QUERY

temporal\_slice = df\_indexed.loc[TARGET\_START\_TIME:TARGET\_END\_TIME]

# STEP 2: SPATIAL FILTER (Conceptually using the geometric structure)

spatial\_mask = (temporal\_slice['latitude'] >= TARGET\_LAT\_MIN) & (temporal\_slice['latitude'] <= TARGET\_LAT\_MAX) & (temporal\_slice['longitude'] >= TARGET\_LON\_MIN) & (temporal\_slice['longitude'] <= TARGET\_LON\_MAX)

final\_result = temporal\_slice[spatial\_mask]

mean\_moisture = final\_result['soil\_moisture'].mean()

print(f"Records found: {len(final\_result)}")

print(f"Mean Soil Moisture for AI Analysis: {mean\_moisture:.2f}%")

Output:

Sample of result data (first 5 records):

latitude longitude soil\_moisture

timestamp

2025-01-15 30.696030 -99.466539 68

2025-01-15 30.620133 -99.305304 68

2025-01-15 30.546710 -99.479692 53

2025-01-15 30.611853 -99.405869 70

2025-01-15 30.609564 -99.406408 61