Name: Loghithakshan.J

Assignment-01

Subject: DAA

---

\*\*1. Brute-Force Approach Analysis\*\*

\*\*Objective\*\*:

Implement a brute-force algorithm that assesses all possible posts and interactions to find the best possible feed recommendations.

\*\*Code\*\*:

```python

import itertools

def brute\_force\_feed(user, all\_posts, scoring\_function):

"""

Function to compute the best feed recommendations using brute-force approach.

:param user: The current user for whom the feed is being optimized.

:param all\_posts: List of all available posts in the platform.

:param scoring\_function: A function that computes a score for each post based on user activity.

:return: A list of top recommendations.

"""

# Generate all possible combinations of posts (assuming the feed has a max size, say 10)

feed\_combinations = itertools.combinations(all\_posts, 10)

best\_feed = None

highest\_score = -float('inf')

for combination in feed\_combinations:

total\_score = sum(scoring\_function(user, post) for post in combination)

if total\_score > highest\_score:

best\_feed = combination

highest\_score = total\_score

return best\_feed

# Example of a scoring function (user-post interactions)

def scoring\_function(user, post):

# Score based on user interaction, e.g., number of likes, comments, recency, etc.

return post['likes'] + 0.5 \* post['comments'] # Simplified scoring

```

\*\*Time Complexity\*\*:

- \*\*Time Complexity\*\*: \( O(C(n, k) \times k) \) where \( n \) is the number of posts and \( k \) is the feed size.

- \*\*Space Complexity\*\*: \( O(C(n, k)) \) for storing combinations.

\*\*Limitations\*\*:

- \*\*Scalability\*\*: This approach becomes impractical as the number of posts increases due to the combinatorial explosion.

- \*\*Real-Time Performance\*\*: For high-frequency updates (new posts), recalculating all combinations is not feasible.

---

\*\*2. Collaborative Filtering: Proof and Efficiency Analysis\*\*

\*\*Objective\*\*:

Examine collaborative filtering for personalized feed recommendations.

\*\*Code\*\*:

```python

import numpy as np

from sklearn.metrics.pairwise import cosine\_similarity

def collaborative\_filtering(user\_idx, all\_users\_data, user\_posts\_matrix):

"""

Function for user-based collaborative filtering.

:param user\_idx: The index of the current user.

:param all\_users\_data: User-item interaction matrix where each row represents a user and columns represent posts.

:param user\_posts\_matrix: A matrix of user-item interactions.

:return: Recommended posts for the user.

"""

# Calculate cosine similarity between the target user and all other users

similarities = cosine\_similarity(user\_posts\_matrix[user\_idx].reshape(1, -1), user\_posts\_matrix)

# Sort users by similarity score and recommend posts that similar users liked

similar\_users\_idx = similarities.argsort()[0][::-1][1:] # Exclude the user itself (index 0)

recommended\_posts = set()

for idx in similar\_users\_idx:

recommended\_posts.update(np.nonzero(user\_posts\_matrix[idx])[0]) # Get posts liked by similar users

return list(recommended\_posts)

# Example usage:

# user\_posts\_matrix = np.array([[1, 0, 1], [0, 1, 1], [1, 1, 0]]) # 3 users, 3 posts

# collaborative\_filtering(0, all\_users\_data, user\_posts\_matrix)

```

\*\*Proof of Correctness\*\*:

- \*\*Correctness\*\*: The algorithm computes cosine similarity between users. Similar users (those with high similarity scores) are likely to have similar preferences, and thus their liked posts are recommended to the target user.

\*\*Time Complexity\*\*:

- \*\*Time Complexity\*\*: \( O(n^2) \) where \( n \) is the number of users (due to pairwise similarity calculations).

- \*\*Space Complexity\*\*: \( O(n \times m) \) where \( n \) is the number of users and \( m \) is the number of posts, due to the user-item matrix.

\*\*Scalability\*\*:

- For large datasets, techniques like matrix factorization (SVD) or approximate nearest neighbor (ANN) search can improve scalability.

---

\*\*3. Dynamic Programming, Greedy, and Approximation Algorithms\*\*

\*\*Objective\*\*:

Implement and compare dynamic programming, greedy, and approximation algorithms.

\*\*Code\*\*:

\*\*Dynamic Programming Approach\*\*:

```python

def dp\_feed(user, all\_posts, scoring\_function, feed\_size):

"""

Dynamic Programming approach to optimize the user's feed.

:param user: The current user for whom the feed is being optimized.

:param all\_posts: List of all available posts in the platform.

:param scoring\_function: A function that computes a score for each post based on user activity.

:param feed\_size: The desired number of posts in the feed.

:return: Optimized feed with highest score.

"""

n = len(all\_posts)

dp = np.zeros((n+1, feed\_size+1))

for i in range(1, n+1):

for j in range(1, feed\_size+1):

score = scoring\_function(user, all\_posts[i-1])

dp[i][j] = max(dp[i-1][j], dp[i-1][j-1] + score)

# Reconstruct optimal feed

feed = []

i, j = n, feed\_size

while i > 0 and j > 0:

if dp[i][j] != dp[i-1][j]:

feed.append(all\_posts[i-1])

j -= 1

i -= 1

return feed[::-1]

# Example usage:

# feed = dp\_feed(user, all\_posts, scoring\_function, feed\_size=10)

```

\*\*Greedy Algorithm\*\*:

```python

def greedy\_feed(user, all\_posts, scoring\_function, feed\_size):

"""

Greedy approach to optimize the user's feed.

:param user: The current user for whom the feed is being optimized.

:param all\_posts: List of all available posts in the platform.

:param scoring\_function: A function that computes a score for each post based on user activity.

:param feed\_size: The desired number of posts in the feed.

:return: Optimized feed with highest score.

"""

scored\_posts = [(post, scoring\_function(user, post)) for post in all\_posts]

scored\_posts.sort(key=lambda x: x[1], reverse=True)

return [post for post, score in scored\_posts[:feed\_size]]

# Example usage:

# feed = greedy\_feed(user, all\_posts, scoring\_function, feed\_size=10)

```

\*\*Approximation Algorithm\*\*:

```python

def approximation\_feed(user, all\_posts, scoring\_function, feed\_size):

"""

Approximation algorithm to generate an optimized feed with near-optimal quality.

:param user: The current user for whom the feed is being optimized.

:param all\_posts: List of all available posts in the platform.

:param scoring\_function: A function that computes a score for each post based on user activity.

:param feed\_size: The desired number of posts in the feed.

:return: Optimized feed with near-optimal score.

"""

posts = sorted(all\_posts, key=lambda x: scoring\_function(user, x), reverse=True)

return posts[:feed\_size]

# Example usage:

# feed = approximation\_feed(user, all\_posts, scoring\_function, feed\_size=10)

```

\*\*Performance Comparison\*\*:

- \*\*Greedy\*\*: Fastest, but may not always provide the optimal solution.

- \*\*Dynamic Programming\*\*: Guarantees optimal solution, but may be slower for large input sizes.

- \*\*Approximation\*\*: Quick, with near-optimal results.

---

\*\*4. Backtracking for Feed Refinement\*\*

\*\*Objective\*\*:

Use backtracking to refine the feed by exploring alternative post combinations.

\*\*Code\*\*:

```python

def backtracking\_feed(user, all\_posts, scoring\_function, feed\_size):

"""

Backtracking approach to refine feed recommendations.

:param user: The current user for whom the feed is being optimized.

:param all\_posts: List of all available posts in the platform.

:param scoring\_function: A function that computes a score for each post based on user activity.

:param feed\_size: The desired number of posts in the feed.

:return: Optimized feed with highest score.

"""

best\_feed = []

best\_score = -float('inf')

def backtrack(current\_feed, idx):

nonlocal best\_feed, best\_score

if len(current\_feed) == feed\_size:

total\_score = sum(scoring\_function(user, post) for post in current\_feed)

if total\_score > best\_score:

best\_feed = current\_feed[:]

best\_score = total\_score

return

if idx == len(all\_posts):

return

# Include the post at idx

current\_feed.append(all\_posts[idx])

backtrack(current\_feed, idx + 1)

# Backtrack: exclude the post at idx

current\_feed.pop()

backtrack(current\_feed, idx + 1)

backtrack([], 0)

return best\_feed

# Example usage:

# feed = backtracking\_feed(user, all\_posts, scoring\_function, feed\_size=10)

```

\*\*Explanation of Effectiveness\*\*:

- Backtracking explores all combinations of posts and refines the feed by selecting the best