Tiktok project - Stochastic dynamic matching

September 2023

1 DRAFT 1

Notes summarised fromStochastic dynamic matching.

1.1 Graph

1.1.1 Vertices

- Consist of $V_{m1}, V_{m2}, ..., V_{mk}$ nodes for k moderators
- Consist of $V_{c1}, V_{c2}, ..., V_{cp}$ nodes for p classes of contents

1.1.2 Edges

- $E = e_1, e_2, ..., e_m$ denotes set of edges. Edges only exist between moderator nodes and content nodes, do not exist within moderators, or contents.
- **To Try**: Assign weights to edges to determine the compatibility between moderator and content
  - Look at: A Framework for Dynamic Matching in Weighted Graphs.
  - We get a **bipartite graph**

1.1.3 Independent Sets

Exists only 2 independent sets: Moderators and Contents

1.2 Arrival Process

- Poisson process with rate λ_i for each $i \in V$
- Treat moderators and contents separately:
  - $\lambda_m = \lambda_{m1}, ..., \lambda_{mk}$ for the k classes of moderators
  - $\lambda_c = \lambda_{c1}, ..., \lambda_{cp}$ for p classes of contents
- Use unit normalisation so that: $\sum_{i \in m} \lambda_i = 1$ and $\sum_{j \in c} \lambda_j = 1$

- $I_m = (I_{mt}, t \in N)$ is i.i.d such that I_{mt} is the moderator who is free at time $t+1$. Then $P(I_{mt} = i) = \frac{\lambda_i}{\sum_{\lambda_m}}$
- $I_c = (I_{ct}, t \in N)$ is i.i.d such that I_{ct} is the class of content that comes in at time $t+1$. Then $P(I_{ct} = j) = \frac{\lambda_j}{\sum_c \lambda_c}$.

1.2.1 How to take into account score and waiting time

Sort the queue in each node, with order as a function of score and waiting time?

2 DRAFT 2

2.1 Problem Description

We are given a bipartite graph that is divided into two independent sets: moderator nodes and content nodes. Each moderator node can potentially be matched with any content node. The goal is to find the optimal pairing between moderators and contents based on dynamically changing weights.

- **Moderator Node**: Represented by a pair (score, vector).
- **Content Node**: Represented by a pair (score, vector), and length of content. Additionally, it has an attribute to track the time since it was last paired with a moderator node.
- **Weight Calculation**: The weight between a moderator and a content node is determined by a function that takes into account:
  - Time since the content node was last matched.
  - Score difference between the moderator and the content.
  - Cosine similarity between the vectors of the moderator and the content.
- **Evaluation Time**: When a moderator is paired with content, it takes some time to evaluate the content. This time follows a Beta distribution, which considers the score of the moderator and the length of the content.

2.2 Algorithm

The algorithm works in iterative time steps:

1. Increase the time since last match for each unmatched content node.
2. Pair unmatched moderators and contents using maximum weight matching.
3. Update the matched status for the paired nodes.

4. For each matched pair, use the Beta distribution to decide if the evaluation has completed. If completed, unpair them and replace the content node if there's more content in the queue.

5. Continue to the next time step.

2.3 Python Code

```
import numpy as np
from scipy.optimize import linear_sum_assignment
from scipy.stats import beta
from scipy.spatial.distance import cosine

class Moderator:
    def __init__(self, score, vector):
        self.score = score
        self.vector = vector
        self.matched = False

class Content:
    def __init__(self, score, vector, length):
        self.score = score
        self.vector = vector
        self.length = length
        self.time_since_empty = 0
        self.matched = False

def weight_function(moderator, content):
    time_since_last_match_weight = content.
        ↪ time_since_empty
    score_diff_weight = 1 / (1 + abs(moderator.score -
        ↪  content.score))
    cosine_similarity_weight = 1 - cosine(moderator.
        ↪ vector, content.vector)

    # You can adjust the coefficients based on the
        ↪ importance you assign to each factor
    return time_since_last_match_weight +
        ↪ score_diff_weight + cosine_similarity_weight

def maximum_weight_matching(moderators, contents):
    weights = -np.inf * np.ones((len(moderators), len(
        ↪ contents)))
    for i, mod in enumerate(moderators):
        if not mod.matched:
            for j, con in enumerate(contents):
```

```
                if not con.matched:
                    weights[i, j] = weight_function(
                        ↪ mod, con)
    row_ind, col_ind = linear_sum_assignment(-weights)
    return row_ind, col_ind

m = 5
moderators = [Moderator(np.random.rand(), np.random.
    ↪ rand(3)) for _ in range(m)]
content_queue = [Content(np.random.rand(), np.random.
    ↪ rand(3), np.random.randint(1, 5)) for _ in range
    ↪ (20)]
contents = [content_queue.pop(0) for _ in range(2 * m)
    ↪ ]

for _ in range(100):
    # Increase the time_since_empty for unmatched
        ↪ content nodes
    for content in contents:
        if not content.matched:
            content.time_since_empty += 1

    # Pair unmatched moderators and contents
    row_ind, col_ind = maximum_weight_matching(
        ↪ moderators, contents)

    # Mark them as matched
    for mod_idx, con_idx in zip(row_ind, col_ind):
        moderators[mod_idx].matched = True
        contents[con_idx].matched = True
        contents[con_idx].time_since_empty = 0  #
            ↪ Reset since it's now matched

    # Check if each matched moderator-content pair
        ↪ gets unmatched using Beta distribution
    for mod_idx, con_idx in zip(row_ind, col_ind):
        a_val = 1 + 4 * moderators[mod_idx].score  #
            ↪ Linear mapping for score
        b_val = 1 + 4 * contents[con_idx].length  #
            ↪ Linear mapping for length

        prob_complete = beta.cdf(1, a_val, b_val)  #
            ↪ Probability of completion

        if np.random.rand() < prob_complete:  #
            ↪ unmatch with this probability
```

```
            moderators[mod_idx].matched = False
            contents[con_idx].matched = False
            if content_queue:  # refill the content
                ↪ node if there's content in the queue
                contents[con_idx] = content_queue.pop
                    ↪ (0)

    # Display pairings for the current time step
    print(f"Time Step { + 1}:")
    for mod_idx, con_idx in zip(row_ind, col_ind):
        print(f"Moderator {mod_idx} paired with
            ↪ Content {con_idx}")
    print("-" * 50)
```

https://chat.openai.com/share/58528edb-d114-417a-b508-4df473d83612

3 DRAFT 3

To edit on draft 2:

- Replace content queue with priority queue, order contents based on their score.
- Don't wait for moderators to finish task at hand before pairing, each time, select k moderators with least time left for evaluation, and below a certain threshold.
- Use genetic algorithm (or similar stochastic algorithm) to do maximum weight pairing.
- Design loss function that takes into account priority of content, score/vector matching between content and moderator, and total time taken to moderate all contents.

4 DRAFT 4

4.1 Problem formation

Assign $\sim$ 40000 contents to $\sim$ 1000 moderators.
Contents have the following attributes:

- Score (Priority)
- Country
- Complexity (Measure of time needed)

Moderators have the following attributes:

- Score (Ability)
- Country (A list/vector)
- Productivity (Measure of efficiency)
- Length of tasks at hand (waiting time for next task)

4.2 Approach

Split Moderators into 3 categories: High score, average score, low score; Contents into 3 categories: High priority, average priority, low priority. The number contents in each category should be proportional to the number of workers in the corresponding category.

In a given category of contents, the contents are ordered by their priority score, so that contents with higher priority will be processed first

Given ONE category of moderator and content:

- Perform assignment in batches
- Size of batch = number of moderators in that category
- Perform assignment using algorithm such as the Hungarian algorithm, aiming to minimize the loss function.
- After assignment of each batch, update the length of tasks at hand attribute for moderators

4.3 Loss Function

Depends on: Score difference between content and moderator, compatibility of countries, length of moderator's queue after assignment

$$L(moderator, content) = w_1 L_1 + w_2 L_2 + w_3 L_3, w_2 < 0$$
$$L_1 = |moderator[score] - content[score]|$$

$$L_2 = \max_i compatibility\_score(content[country], moderator[contry_i])$$

$$L_3 = moderator[lengthoftaskathand] + time\_needed(moderator, content$$

Note:

- time_needed(moderator, content) will be a function to compute estimated amount of needed for moderator to process the content.