IASC-INSA-NASI SUMMER RESEARCH FELLOWSHIP PROGRAM- 2024

Eight-week Report

Date: 11th July, 2024

Summer Research Fellow:
Aishini Bhattacharjee
Application no.: ENGS1577

Guide:
Dr. Sudarshan Iyengar
Department of Computer Science and Engineering
Indian Institute of Technology Ropar

Project Problem Statement

The aim of this project is to construct a concept positioning/recommender system. Students and learners are often faced with difficulty and confusion regarding which concept /chapter they should be diving into, given that they know certain concepts. Our attempt to solve this problem encompasses various approaches as given below:

Data Collection and Processing: Naive Approach

Initially, we had considered mining textbooks and MOOC datasets to construct a hierarchical relationship between StackOverflow tags, and that was seen to be useful to some extent. We took the help of BERT, a language model, and fine-tuned it on the corpus of textbooks on data structures and algorithms to capture semantic relations between tags and construct embeddings accordingly.
For context: BERT (Bi-directional Encoder Representations from Transformers) stands as an open-source machine learning framework designed for the realm of natural language processing (NLP). It employs an encoder-only architecture. Traditional language models process text sequentially, either from left to right or right to left. This method limits the model's awareness to the immediate context preceding the target word. BERT uses a bi-directional approach considering both the left and right context of words in a sentence, instead of analyzing the text sequentially, BERT looks at all the words in a sentence simultaneously.

Here are the detailed steps taken:

1. Initial Setup and Data Loading: We began our project by importing essential libraries and loading our data. We used pandas, a powerful data manipulation library in Python, to read our 'Questions.csv' file into a structure called a DataFrame. This DataFrame is like a table that allows us to easily work with our data.

2. Tag Parsing: Our dataset contained tags in a special format, enclosed in angle brackets. We created a function to extract these tags, transforming them from a string like "<tag1><tag2>" into a list ["tag1", "tag2"]. This made the tags easier to work with in our subsequent analyses.

3. Data Limitation: To manage our computational resources effectively, we decided to limit our analysis to the first 500 rows of our dataset. This allowed us to develop and test our methods more quickly.

4. Text Preprocessing: We then moved on to clean our text data. We used the Natural Language Toolkit (NLTK), a leading platform for building Python programs to work with human language data. We removed common words called "stopwords" (like "the", "is", "at") that don't carry much meaning, as well as HTML tags and links. This process helped us focus on the most important words in our text.

5. Entity Extraction: After cleaning our text, we extracted what we call "entities". In our case, we used a simple method of considering each word as an entity. In more advanced applications, entities might be more specific things like names, places, or organizations.

6. Similarity Calculation: To understand how similar different pieces of text were to each other, we used a technique called TF-IDF (Term Frequency-Inverse Document Frequency) vectorization. This converts text into numbers in a way that reflects how important words are to a document in a collection of documents. We then calculated the cosine similarity between these vectors, which gives us a measure of how similar two pieces of text are.

7. Determining Related Tags: Using our similarity scores, we determined which items in our dataset were related. We considered two items related if their similarity score was above 0.7, a threshold we chose based on our understanding of the data.

8. Graph Creation: We then represented our data as a graph, which is a mathematical structure consisting of nodes (points) and edges (lines connecting the points). In our case, each node represented a tag, and we connected nodes with edges if their tags appeared together in the same item or if they were from related items.

9. Graph Analysis: To understand our graph better, we created a function to show us all the tags (nodes) connected to a given tag. This helped us see the relationships between different tags.

10. Graph Visualization: We used matplotlib, a plotting library, to create a visual representation of our graph. This visualization helped us see patterns and relationships that might not be obvious from looking at the data in a table.

11. Graph Data Extraction: We extracted some key information from our graph, including the adjacency list (which shows which nodes are connected to each node) and the edge list (which lists all the connections in the graph).

12. BERT Embedding Generation: We then used BERT to generate embeddings for each of our tags. An embedding is a way of representing words or phrases as vectors of real numbers.

13. Similarity Matrix Creation: Finally, we created a similarity matrix using our embeddings. This matrix shows how similar each tag is to every other tag, based on their representations. We used cosine similarity again for this comparison.

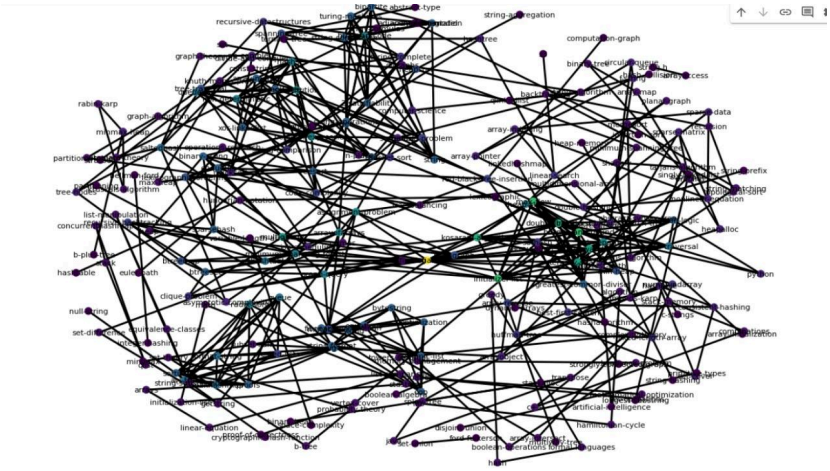


Figure: Graph obtained, representing relationships between tags

From the graph and similarity matrix, we queried ChatGPT and attempted to obtain relative difficulties of various concepts on a scale of -5 to 5. The matrix was constructed in the following way:
The value at the ith row and jth column depicted the relative difficulty of concept j in comparison to concept i. A value of 5 indicated that concept j was much more difficult than concept i, a value of -5 indicated that concept i was much more difficult than concept j, and 0 indicated equivalent difficulties.

Limitations of this approach: While we did use transitive properties to assign absolute difficulty scores to all concepts, it did not work well because ChatGPT tended to give erroneous answers in some cases, hence we could not vouch for the hierarchy obtained.

Note: The algorithm that I used to assign absolute difficulty ratings to concepts was the Analytical Hierarchical Process (AHP), detailed below (I'll call the pairwise comparison matrix A):

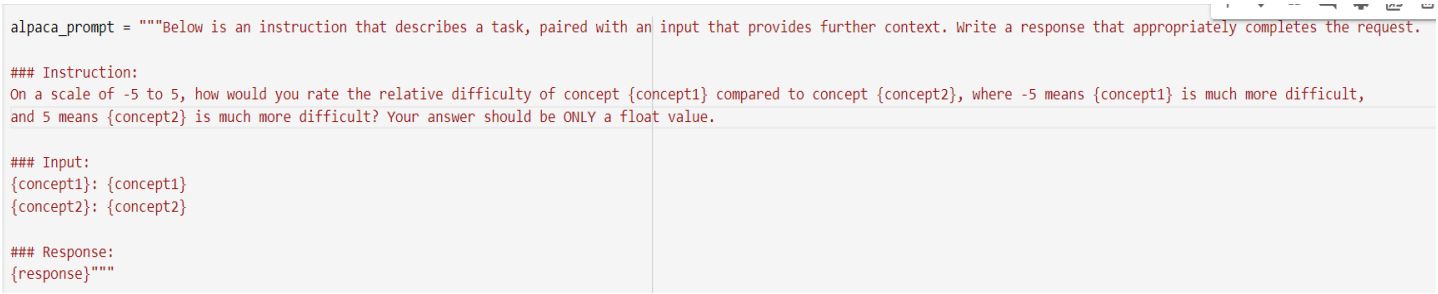
1. Normalization: I adjust this matrix so each column sums to 1. This creates a special type of matrix that's useful for the next step.

2. Priority Vector Computation: I look for the eigenvector associated with the largest eigenvalue of my normalized matrix. This eigenvector becomes my priority vector, giving each concept a single score that reflects its difficulty relative to all other concepts.

3. Consistency Check: I use the largest eigenvalue to check if my original comparisons are consistent. If this eigenvalue is close to the number of concepts I'm comparing, it suggests my comparisons are fairly consistent.

4. Absolute Rating Assignment:
   - I map the priority vector to an absolute scale, initially [0, 10].
   - For each concept i: rating_i = 0 + (w_i - min(w)) * (10 - 0) / (max(w) - min(w))
   - This linear transformation preserves the relative differences in the priority vector.

5. Rating Normalization:
   - I apply another linear transformation to ensure my ratings span the full [0, 10] range.
   - normalized_rating_i = 0 + (rating_i - min(rating)) * (10 - 0) / (max(rating) - min(rating))



```
alpaca_prompt = """Below is an instruction that describes a task, paired with an input that provides further context. Write a response that appropriately completes the request.

### Instruction:
On a scale of -5 to 5, how would you rate the relative difficulty of concept {concept1} compared to concept {concept2}, where -5 means {concept1} is much more difficult,
and 5 means {concept2} is much more difficult? Your answer should be ONLY a float value.

### Input:
{concept1}: {concept1}
{concept2}: {concept2}

### Response:
{response}"""
```

Figure: Custom prompt used for obtaining relative ratings

Note: When we were attempting to construct a hierarchy graph, I had thought of implementing a path-planning algorithm, which would be able to direct a learner from a concept s/he knows to a concept that s/he wishes to learn in a directed graph, but at that moment results were inaccurate, probably because of relatively incorrect relative difficulty ratings assigned by ChatGPT. Here is the algorithm:

1. The graph contained edges directed from a concept with lower absolute difficulty rating to higher absolute difficulty rating.

2. To traverse from a node with lower difficulty to a node with higher difficulty, I would wish to traverse as few nodes as possible, but it should not result in traversing between nodes of widely varying difficulty.
   Eg. In the graph, there is a directed edge from ‘python’ to ‘machine-learning’. But the variance in the difficulty of these two concepts is very high. Even if the shortest path

algorithm takes the direct edge, that's not an accurate representation of anyone's learning path.

3. If I considered only the absolute difference in difficulty ratings as weights, there was a high chance of possibly encountering many redundant nodes in the graph, which wouldn't necessarily be related but would be pulled into the path because they technically won't be adding to the path length.

4. I had considered using a weight metric which would lead to a balance between the 'slope' of the path and the number of nodes traversed, ie, the absolute difference in rating between the source node and the node traversed at any point, divided by the number of nodes encountered.

Effective Matrix Construction (Work on this was done by my teammates)

The challenge was to construct a user-tag matrix which would depict user-tag interactions in a way such that helpful inferences could be drawn from it. We took all the data from the well-known community-based website StackOverflow, and we used the sub-domain of data structures and algorithms for experimenting. While analysis of tags showed a few useful trends, there were a lot of redundant programming language-based tags like 'C', 'Java' and other preliminary concepts which weren't really relevant to the domain of data structures and algorithms from a learner's perspective. We were provided with a set of 346 tags, which we pruned and brought down to 231, by eliminating irrelevant tags.

Furthermore, StackOverflow metadata consisted of various metrics which we had to take into consideration while building the matrix. There were attributes like QuestionScore, AnswerScore, time of questioning, time of answering, extent of user interaction in the website in general and user reputation. Taking into consideration all these factors, we decided to go with the answer scores of a particular user while constructing the user tag matrix. The answer score of a user for a particular tag is the sum of all upvotes that all of the user's answers, to questions corresponding to particular tags, have received. Here are the metrics which we considered, but didn't go forward with:

1. User Reputation: While this did seem like a great metric for gauging a user's experience overall in the domain, it failed to evaluate a user's proficiency in specific tags. In the future, we could aim to construct a metric which would combine a user's reputation and answer score.

2. Question Score: Even though it was considered in the beginning, it ended up not being accepted because again, it didn't really give us an idea of how good a user was at something, rather it gauged how good questions framed by users are, which is not a direct indication of one's expertise in a subject.

3. Time Stamp: It was considered as a means to figure out how every user's learning curve should progress, but eventually after experimenting, we failed to find a meaningful sequence of user interaction, which could explain the learning curve of any learner.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1		OwnerUse	proof	probability	finite-auto	np-comple	mathemat	linear-pro	satisfiabili	stable-mai	np	complexity	boolean-lo	boolean
2	0	1	0	0	0	0	0	0	0	0	0	0	0	0
3	1	3	0	0	0	0	0	0	0	0	0	0	0	0
4	2	4	0	0	0	0	0	0	0	0	0	0	0	0
5	3	5	0	0	0	0	0	0	0	0	0	0	0	0
6	4	13	0	0	0	0	0	0	0	0	0	105	0	116
7	5	17	0	0	0	0	0	0	0	0	0	0	0	18
8	6	20	0	0	0	0	0	0	0	0	0	0	0	0
9	7	22	0	0	0	0	0	0	0	0	0	0	0	0
10	8	23	0	0	0	0	0	0	0	0	0	0	0	0
11	9	25	0	0	0	0	0	0	0	0	0	0	0	0
12	10	26	0	0	0	0	0	0	0	0	0	0	0	0
13	11	29	0	0	0	0	0	0	0	0	0	0	0	13

Figure: A sample of the user-tag matrix obtained from the filtered tags.

Construction of Challenge Dataset

We were assigned the task of constructing a challenge dataset for a competition to be hosted on Kaggle. A variety of methods were employed for this purpose. I will elaborate on a probabilistic approach that I employed, in an attempt to generate synthetic user sequences, reflecting the distribution of various tags involved.

Part A: Sequence Construction

Here is how user tags were flattened to form sequences, with one sequence for each user:

1. calculate_time_score function:
   - I take two timestamps as input: the current timestamp and the oldest timestamp.
   - If either is None, I return 0.
   - I calculate the time difference in seconds between these timestamps.
   - This time difference becomes the time score.

2. calculate_combined_score function:
   - I take several inputs: time_score, actual_score, su (sum of scores), and optional weights.
   - I normalize the time_score by dividing it by the maximum of itself and 1.
   - I normalize the actual_score by dividing it by the maximum of su and 1.
   - I then compute a weighted sum of these normalized scores:
   combined_score = (time_weight * normalized_time_score) + (score_weight * normalized_actual_score)

3. process_row function: This is the main function that processes each row of data. Here's what I do after preprocessing the data:

a. Score Calculation:
- I find the oldest timestamp among the valid data.
- I calculate time scores for each timestamp using calculate_time_score.

- I calculate combined scores using calculate_combined_score.

b. Tag Sorting and Deduplication:
- I sort the tags based on their combined scores in descending order.
- I flatten the sorted list of tag lists into a single list.
- I create a final list of unique tags, preserving the order of first appearance.

Part B: Data Generation

1. analyze_sequences function: This function performs statistical analysis on a set of tag sequences.
    a. Data Collection: I iterate through each sequence, collecting: Unique tags (all_tags set), Tag frequencies (tag_counts Counter), Normalized positions of each tag (tag_positions dictionary), Sequence lengths
    b. Statistical Calculations: For each tag, I calculate:
    Count: The frequency of the tag
    Mean Position: $\mu = \Sigma(xi) / n$, where xi are the normalized positions
    Standard Deviation of Position: $\sigma = \sqrt{(\Sigma(xi - \mu)^2 / n)}$
    Probability: P(tag) = count(tag) / total_count
    c. Length Distribution: I use numpy's histogram function to create a discrete probability distribution of sequence lengths.

2. generate_synthetic_sequence function: This function generates a synthetic sequence based on the analyzed statistics.
    a. Length Determination: I sample a length from the empirical length distribution using numpy's random.choice.
    b. Tag Selection: For each position in the sequence: I calculate a probability for each tag using a Gaussian-like function:
    $P(tag|position) \propto P(tag) * \exp(-(position - \mu)^2 / (2\sigma^2))$
    Where μ is the mean position and σ is the standard deviation of position for that tag. I normalize these probabilities and sample a tag using numpy's random.choice.

Part C: Introducing some randomness to reduce overlap

It was seen that while processing smaller chunks of the dataset, considerable portions of the sequences generated overlapped with those already present. To mitigate this, I carried out the following:

1. While calculating the probability of occurrence of a given tag at a given position in the sequence, I introduced a uniform random factor between 0.5 and 1.5
    $P'(tag|position) = P(tag|position) * U(0.5, 1.5)$, where U is a uniform distribution.

2. I initially generated double the number of sequences required to be generated.

3. I calculated the initial overlap: overlap_ratio = overlap_count / num_synthetic_initial

4. I filtered unique sequences and added back a controlled number of potentially overlapping sequences to achieve 20% overlap.

```
Comparison of original vs synthetic statistics:
Original - Mean length: 2.39, Std length: 3.49
Synthetic - Mean length: 2.28, Std length: 3.18
```

Figure: Comparison of mean lengths and standard deviation of sequence lengths in original and synthesized data

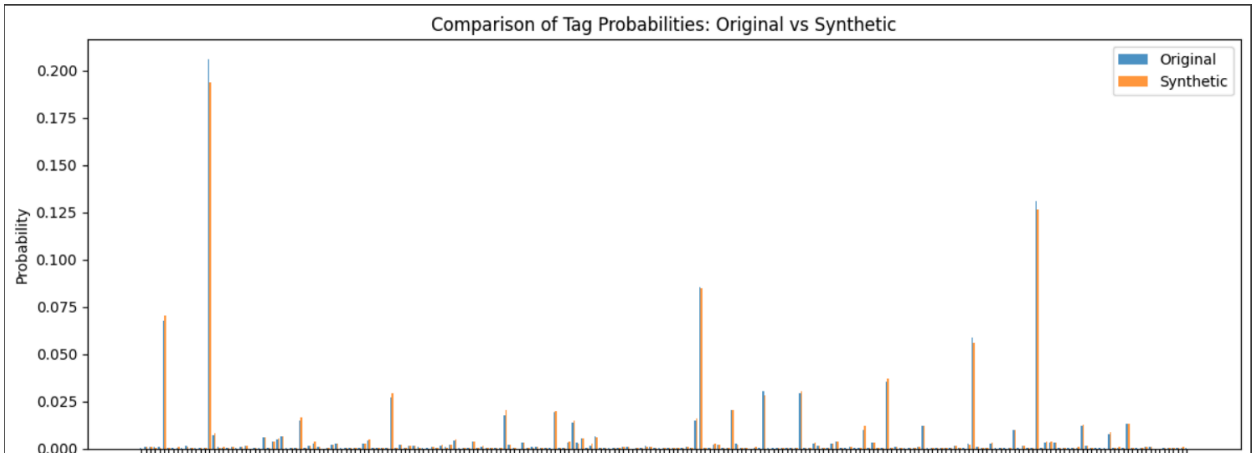


Figure: Tag Distributions in original and synthetic data

```
"['arrays', 'string', 'boolean', 'list', 'doubly-linked-list', 'fibonacci', 'list', 'stack-memory', 'arrays', 'tree', 'string']"
['list']
"['arrays', 'arrays', 'sorting', 'arrays', 'list', 'data-structures', 'data-structures', 'algorithm', 'list', 'optimization', 'string-comparison', 'linear-search'
"['sorting', 'big-o']"
['arrays']
"['optimization', 'string', 'arrays', 'optimization', 'arrays', 'list', 'boolean', 'substring']"
"['recursion', 'sorting']"
['string']
"['arrays', 'boolean', 'memory-management']"
['string']
"['list', 'memory-management']"
"['string', 'singly-linked-list']"
['recursion']
"['list', 'insertion-sort', 'multidimensional-array']"
"['string', 'memory-management', 'arrays']"
['list']
['memory-management']
"['string', 'stack', 'stack']"
['arrays']
"['string', 'linked-list', 'optimization', 'sorting', 'tree-traversal']"
"['algorithm', 'recursion']"
['string']
['arrays']
"['string', 'optimization', 'list']"
['graph']
['stack']
"['string', 'arrays', 'set', 'mathematical-optimization', 'list', 'fibonacci']"
['hash']
['arrays']
"['arrays', 'arrays', 'graph', 'arrays', 'algorithm']"
"['memory-management', 'multidimensional-array']"
```

Figure: A sample of the synthetic sequences generated

Recommender Systems
Based on the user-tag matrix obtained (as shown in previous pages), we decided to try out various approaches for building an effective recommender system for recommending unknown tags. I decided to try out neural collaborative filtering and Restricted Boltzmann Machine.

Related Concepts

Neural Collaborative Filtering

In traditional collaborative filtering, we recommend items based on user and item similarities. For example, if two users have similar preferences, we can recommend items liked by one user to the other. NCF enhances this by using neural networks to learn more complex patterns in the data.

Key Components

1. Users and Items: Let's say we have a set of users and a set of items. Each user has interacted with some items (e.g., watched movies or bought products).
2. Interactions: These interactions are usually represented in a matrix where rows are users, columns are items, and entries indicate interactions (like ratings or clicks).

Neural Network Structure
NCF uses a neural network to learn the interaction patterns between users and items. Here's a simplified structure:

1. Input Layer: Takes user ID and item ID as inputs.
2. Embedding Layer: Converts user ID and item ID into dense vectors (called embeddings) that capture their features.
3. Hidden Layers: Several layers of neurons that process the embeddings to learn complex interaction patterns.
4. Output Layer: Produces a prediction score indicating how likely the user is to interact with the item.

How It Works

1. Learning Embeddings: The model learns embeddings for users and items during training. These embeddings capture latent factors (hidden features) representing user preferences and item characteristics.
2. Combining Embeddings: The embeddings are combined and passed through hidden layers to model user-item interactions.
3. Prediction: The final layer outputs a prediction score. A high score means the model predicts that the user will likely interact with the item.

In the final model, pre-trained weights of both matrix factorization and multi-layer perceptron are embedded.

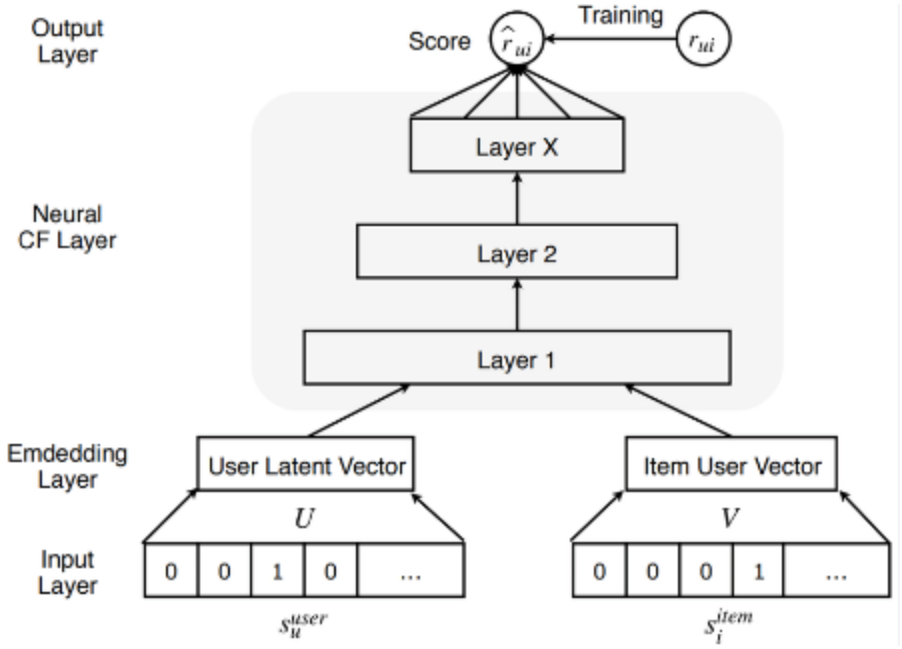


Figure: Basic Structure of an NCF model (taken from ResearchGate: link)

Restricted Boltzmann Machine

Structure:
Visible Layer: Represents the items. Each neuron corresponds to an item, and its state indicates whether the user has interacted with the item (e.g., watched a movie or rated a product).
Hidden Layer: Captures the hidden features or preferences of users. Each neuron corresponds to a hidden feature that the RBM learns from the data.

Connections:
Bidirectional Connections: Neurons in the visible layer are connected to neurons in the hidden layer with weights.
No Intra-Layer Connections: Neurons within the same layer are not connected to each other.

Working:

1. We start by assigning random weights between visible and hidden layers. The training data is the user-tag matrix.
2. Forward pass: For a given user, the items they have interacted with activate the visible layer neurons. These activations are passed to the hidden layer.
3. Backward pass: The activations of the hidden neurons are passed back to the visible layer. Each visible neuron updates its state based on the activations from the hidden layer and the weights.

Results and Evaluation

We considered a few evaluation metrics for evaluating the recommender.

Precision
Precision@k measures the accuracy of the top-k recommendations.

- **Definition**: The proportion of recommended items in the top-k set that are relevant.
- **Formula**: Precision@k = (Number of relevant items in the top-k recommendations) / k.

Recall
Recall@k measures the coverage of the top-k recommendations.

- **Definition**: The proportion of relevant items that are included in the top-k recommendations.
- **Formula**: Recall@k = (Number of relevant items in the top-k recommendations) / (Total number of relevant items).

NDCG
NDCG@k measures the ranking quality of the top-k recommendations, taking into account the position of relevant items.

- **Definition**: A measure that considers the position of relevant items in the recommendations, with higher relevance and position getting higher scores.
- **Formula**: NDCG@k = (DCG@k) / (IDCG@k), where DCG@k is the Discounted Cumulative Gain and IDCG@k is the Ideal DCG.
    - DCG@k = Sum of (relevance of item / log2(position + 1)) for the top-k items.
    - IDCG@k is the DCG value for the ideal (perfect) ranking.

MAP
MAP@k measures the average precision across all users and recommendations up to k items.

- **Definition**: The mean of the Average Precision scores for all users, where Average Precision considers both precision and the position of relevant items.
- **Formula**: MAP@k = (Sum of Average Precision@k for all users) / (Number of users).
    - Average Precision@k = (Sum of Precision@i for each relevant item i in the top-k) / (Number of relevant items in the top-k).

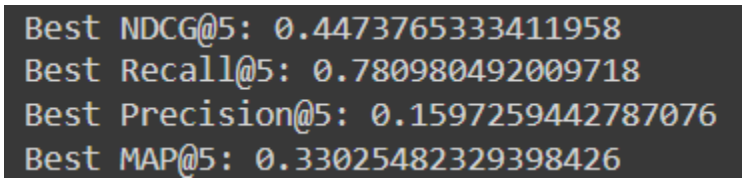


Figure: Training Results for Restricted Boltzmann Machine for the Original Dataset

['algorithm', 'optimization', 'list']	boolean	memory-manage	recursion	substring	hash
['boolean', 'boolean-expression', 'optimization', 'string', 'hash', 'arrays', 'memory-management', 'queue', 'sorting', 'list']	partitioning	fifo	code-complexity	heuristics	set-theory
['string']	algorithm	arrays	optimization	list	boolean
['algorithm', 'optimization', 'string', 'hash', 'multidimensional-array', 'arrays', 'graph', 'combinatorics', 'sorting', 'list']	heuristics	probability	graph-theory	code-complexity	mathematical-optimization
['string', 'arrays', 'recursion']	set	multidimensional	boolean	queue	graph
['algorithm', 'string', 'clique-problem', 'graph-theory']	clique	bellman-ford	undirected-graph	ford-fulkerson	heap-size
['optimization', 'memory-management', 'recursion']	stack	multidimensional	set	hashtable	boolean
['algorithm', 'string', 'arrays', 'sorting']	optimization	data-structures	recursion	boolean	memory-management
['heuristics', 'algorithm', 'string', 'data-structures', 'arrays', 'recursion', 'sorting']	linear-programm	greedy	shortest-path	sparse-matrix	code-complexity
['string']	algorithm	arrays	optimization	sorting	list
['recursion']	multidimensiona	string-concatenat	stack	substring	set
['string']	arrays	algorithm	optimization	list	hash
['hashtable', 'string', 'data-structures', 'arrays', 'sorting', 'list']	linked-list	tree	stack	multidimensional-arr	substring
['boolean', 'optimization', 'substring', 'string', 'tree', 'arrays', 'heap-memory', 'stack', 'memory-management', 'recursion', 'list']	binary-tree	boolean-logic	queue	binary-search	priority-queue
['heap-memory', 'memory-management']	hashmap	stack	hashtable	linked-list	stack-memory
['arrays']	algorithm	optimization	list	sorting	hash
[]	algorithm	string	arrays	optimization	data-structures
['algorithm', 'string', 'graph', 'tail-recursion', 'fibonacci', 'recursion']	heuristics	finite-automata	boolean-logic	proof	linear-programming
['algorithm']	optimization	data-structures	sorting	memory-managemer	tree
['algorithm', 'string', 'data-structures', 'arrays', 'queue', 'recursion', 'list']	partitioning	graph	string-matching	string-comparison	code-complexity
['optimization', 'hash']	recursion	stack	multidimensional-ar	substring	boolean
['string', 'arrays', 'memory-management']	hash	substring	stack	recursion	hashtable

Figure: Sample recommendations obtained from the neural collaborative filtering recommender

Future work:

1. Fine tuning of the neural collaborative filtering model is being slightly difficult. I aim to carry it out in an efficient way and give firm evaluation metrics for it to be compared with other recommenders tested.
2. I wish to develop an ensemble model which would be able to make recommendations according to the form of input, the recommender would be personalized to user profiles and should also be able to address cold start problems based on user profiles or other data.