Data 301 Project Final Report **Ryan Teape 68855096**

Please note:

1) **AI**: For all report text and code comments, I wrote the initial content myself and then used generative AI tools to perform spelling checks and improve readability. The graphs in this report were also generated with AI from the test data I gave it. I've highlighted in light grey the parts of the report that were ran through AI for spell check and readability (i.e. all of it) to comply with the Data 301 generative AI policy / to indicate the use of AI.

2) I changed the primary accuracy metric from precision at K (as stated in my proposal) to recall at K, since recall better measures the recovery of ground truths.

Abstract / Summary

My proposed project is to create a user-based collaborative content-recommendation system using the Twitch dataset. The research question is "How well does user similarity predict the accuracy of Twitch-streamer recommendations, as measured by recall at K?" Algorithms used include divide-and-conquer, map, foldby, etc., cosine similarity; and potentially hashing, clustering, and Dask-ML k-nearest neighbours. I suspect user similarity will correlate highly with streamer recommendation accuracy, if so, we can conclude that user similarity is a valid basis for a content-recommendation system in the context of Twitch.

Introduction

Motivation

I'm interested in this research question because I'm interested more generally in using data to make more optimal decisions. I find inferring user preferences from something as simple as Twitch watch history quite interesting. Also, people my age consume a lot of online content and engage with such content-recommendation systems in day-to day life, so understanding how they work is both useful and relevant.

Background

Twitch is an online video live-streaming platform primarily focused on gaming-based content. The Twitch dataset (found here: https://mcauleylab.ucsd.edu/public_datasets/gdrive/twitch/) contains 15.5 million records of user watch history in the form of User ID, Stream ID, Streamer username, Time start, Time stop. The dataset therefore implicitly provides watch time per user per streamer (end time – start time).

The key metric for user similarity will be the cosine similarity between user watch-time matrices. Cosine similarity is appropriate here because it measures the angle between vectors in n-dimensional space and is more accurate for sparse vectors compared to other similarity measures.

K-nearest neighbours allows us to find the k closest other users for a given input user (here, "closest" means highest cosine similarity). Therefore, nearest neighbours will be our most similar users.

Research Question

"How well does user similarity predict the accuracy of Twitch-streamer recommendations, as measured by recall at K?"

More detail:

- User similarity = cosine similarity between vector matrices for each user, where each entry in the matrix represents watch time for a given streamer as a percentage of total watch time for that user.
- Recall@K values for my content-recommendation system will be compared with Recall@K values for both random recommendations and popularity based recommendations in order to evaluate the content recommendation system.
- This question is relevant to the Twitch dataset because it contains large amounts of historical user watch-time data for many streamers. By applying the

algorithms/steps outlined in 5), we should be able to build a functioning

collaborative-filtering-based content-recommendation system.

Experimental Design and Methods

The project implements a collaborative filtering recommendation system, designed to suggest streamers to a target user based on the viewing habits of similar users. The algorithm proceeds through several distinct steps, leveraging Dask for distributed processing of a large dataset. Initially, the system downloads a 100k CSV file of Twitch viewing data. This raw data undergoes preprocessing: each line is parsed into a list, watch times are calculated (end time - start time), and irrelevant stream IDs are discarded. An important step involves mapping streamer names to unique Streamer IDs and then grouping user-streamer watch data to sum total watch time per user per streamer. To evaluate the model's performance, up to four streamers for the target user are deliberately omitted, creating "ground truths" for later comparison.

The core of the recommendation algorithm involves creating a numerical representation of each user's viewing preferences. This is achieved by generating NumPy arrays where each index corresponds to a streamer ID, and the value at that index represents the total watch time for that streamer. These arrays are then normalized to represent percentages of total watch time, effectively forming a watch preference matrix. Cosine similarity is then calculated between the target user's normalized watch preference array and every other user's array to identify the most similar users (nearest neighbours). The top 5 nearest neighbours are selected, and their individual watch times are normalized and then weighted by their cosine similarity to the target user. These weighted watch times are summed per streamer across all nearest neighbours, resulting in a prediction score for each streamer. Finally, the top predicted streamers (excluding those already watched by the target user) are presented as recommendations. The model then evaluates its accuracy by comparing its recommendations against the initially omitted ground truth data, calculating Recall@K.

The following code and libraries were used to implement the methods:

- dask.bag (db): Used for parallel processing of large datasets, enabling efficient manipulation and transformation of the raw CSV data.

- numpy (np): Crucial for numerical operations, particularly for creating and manipulating watch preference arrays and for efficient vector operations.

- scipy.spatial.distance.cosine: Employed to calculate the cosine distance (and thus cosine similarity) between user watch preference vectors, a core component of the collaborative filtering approach.

- dask.distributed.Client, LocalCluster: Used to set up and manage a local Dask cluster, allowing for distributed computation and scaling the number of workers.

- urllib.request: Utilized for downloading the 100k_a.csv dataset from a specified URL.

- time: Used to measure the execution time of the model.

1) run_model(UserIDofInterest, problemSize, numberOfWorkers, k) (function): The main function encapsulating the entire recommendation system logic, from data download and preprocessing to prediction and evaluation.

2) processIntoLists(line) (function): A helper function to parse each line of the CSV into a list.

3) doWatchTimeprocessing(line) (function): A helper function to calculate the watch time from start and end times in each record.

4)discardStreamID(line) (function): A helper function to remove specific stream ID fields from the data.

5)preProcessing(inputBag) (function): A high-level function that orchestrates the initial data cleaning and transformation using the previously mentioned helper functions.

6) createlookupTable(inputBag) (function): Creates a dictionary to map streamer names to unique integer IDs.

7) mapStreamerNametoStreamerID(line) (function): Maps streamer names in the dataset to their corresponding integer IDs.

8) combinedKey(x) (function): A lambda-like function used as a key for grouping data by UserID and StreamerID.

9) reformatData(x) (function): Reformats grouped data into a consistent (UserID, StreamerID, WatchTime) tuple.

10) predicitonTestingFunc(line, UserIDofInterest) (function): Omits specific data points for the target user to create a ground truth for testing.

11) mappingFunct2(line) (function): Transforms raw watch data into a numerical NumPy array representing user watch preferences.

12) datanormalisationfunct(line) (function): Normalizes user watch preference arrays into a percentage-based distribution.

13) downCastArrayDtype(line) (function): Reduces the memory footprint of NumPy arrays by downcasting their data type.

14) getUserInputArray(UserIDofInterest) (function): Retrieves the normalized watch preference array specifically for the target user.

15) cosineSimilaritys2(givenUser) (function): Calculates the cosine similarity between the target user and another given user.

16) computeCosineSimilarities(inputUserArray, datanormalisation) (function): Applies the cosineSimilaritys2 function across all users to compute similarities.

17) upsize(input) (function): Upsizes the data type of arrays for improved precision in subsequent calculations.

18) normaliseCosineSimilarityValues(line, totalCosineSimilarity) (function): Normalizes the calculated cosine similarity values of nearest neighbors.

19) reformat(line) (function): A general reformatting function for cleaning up joined Dask Bag elements.

20) normaliseWatchTimes(line) (function): Normalizes watch times within each user's data to percentages.

21) weightByCosineSimilarity(line) (function): Weights streamer watch times by the cosine similarity of the contributing neighbor.

22) Print Predictions: A sorted list of all streamer names with prediction values above the threshold. Each entry is (Streamer Name, Prediction Score), where a higher score indicates the model's stronger belief that the target user will like that streamer. Note that this list may include streamers the user already watches.

23) Print Input Data: The model's observed data for the target user, formatted as (Streamer Name, Total Watch Time). This represents exactly what the model "saw" during training.

24) Print Recommendations: From the Predictions list, filter out any streamers already in the Input Data. The remaining entries—formatted as (Streamer Name, Prediction Score)—constitute the actual recommendation output.

25) Print Omitted Data: The list of streamer names withheld for testing. These are the "ground truth" we expect the model to recover.

26) Print Correct Recommendations: The intersection of Recommendations and Omitted Data. These are the streamers the model correctly predicted despite them being removed from the training input.

Results

Scalability Graphs and Data

below is the data used to make the scalability graphs

Runtime (Seconds)	Problem Size (# of records)	# of CPUs	Work per CPU	Efficiency (%)	Runtime Increase (%)
13.643573522567749	5,000	1	5,000	100.0%	- (baseline)
14.297307968139648	10,000	2	5,000	95.4%	+4.8%
14.180954217910767	20,000	4	5,000	96.2%	+3.9%
15.697008848190308	30,000	6	5,000	86.9%	+15.0%
16.902909517288208	40,000	8	5,000	80.7%	+23.9%

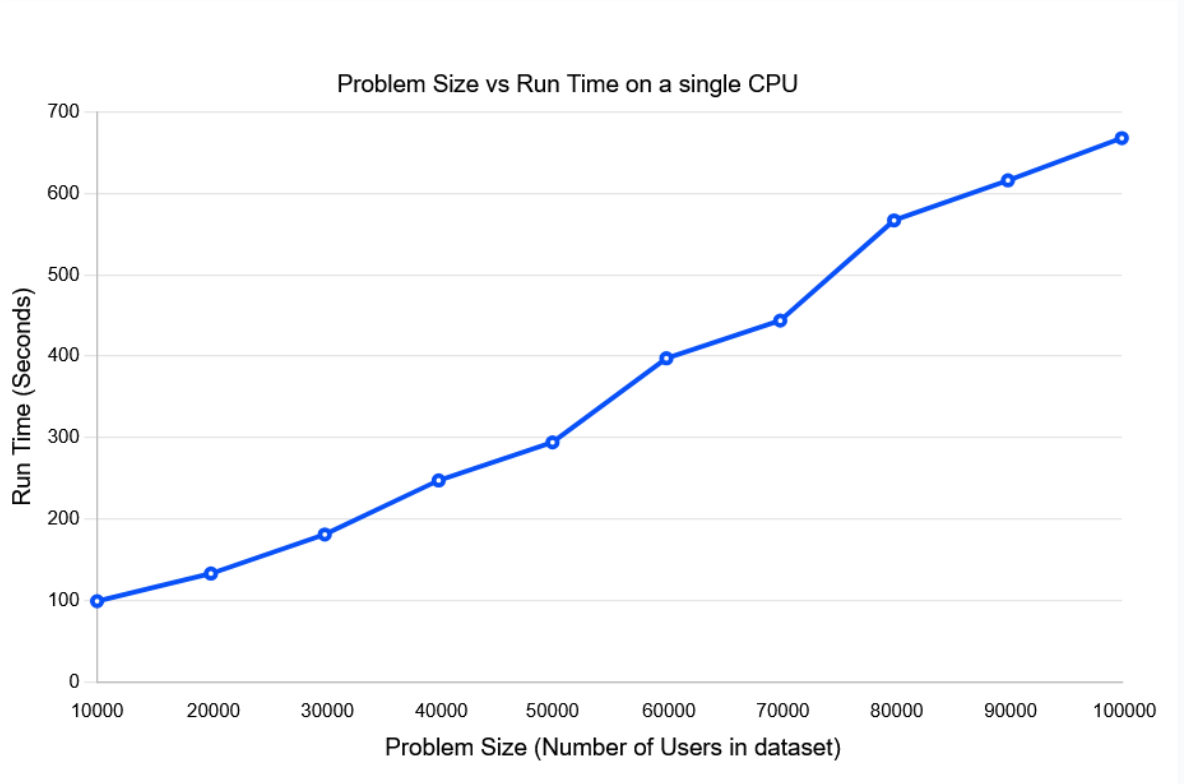
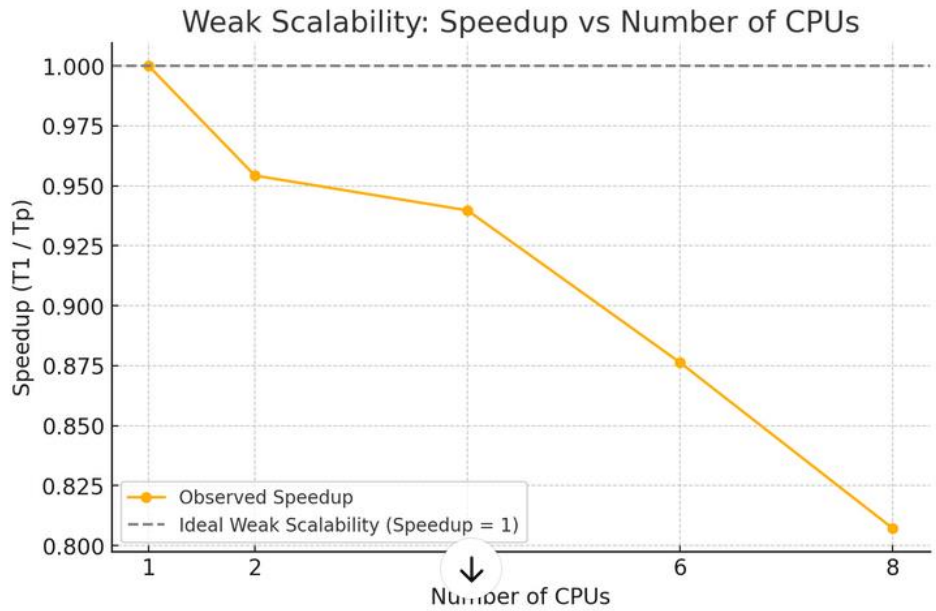
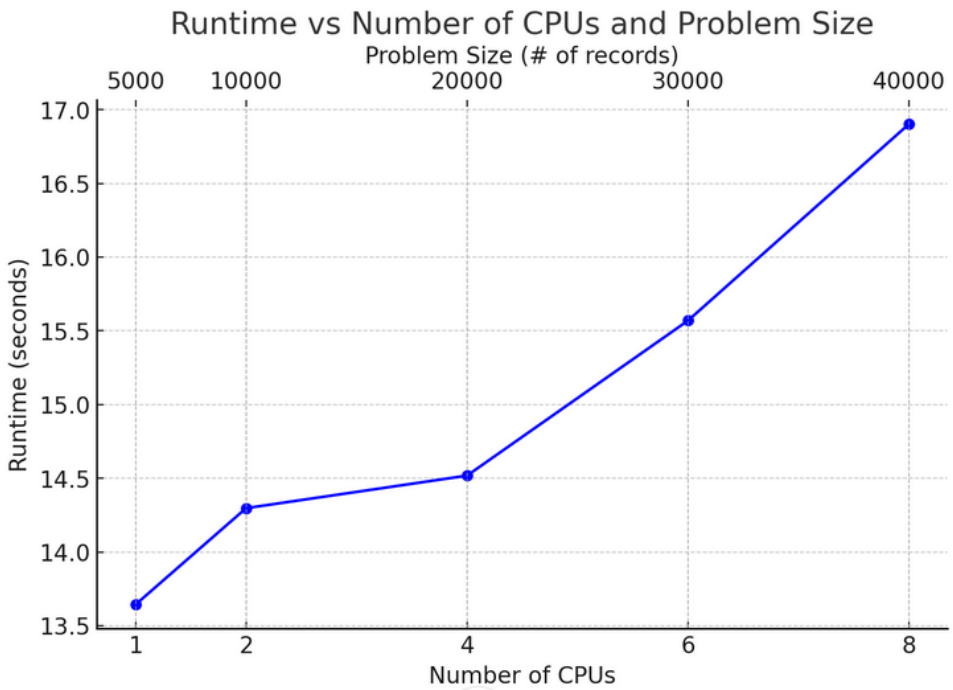


Figure 1.1

The above graph (figure 1.1) shows Run time of my project (seconds) vs problem size (in number of users / # of records). As you can see on a single CPU Run time increases roughly linearly as problem size increases.



The graph above shows the weak scalability of the model by plotting speed-up versus the number of CPUs while keeping the workload per CPU constant. As the number of CPUs increases, the speed-up falls significantly below the ideal weak-scalability line. This indicates that the program does not run fully in parallel and/or is somewhat poorly optimised. From testing, I know that the sequential creation of the streamer-ID to streamer-name lookup dictionary is a main culprit as this part of the program does not scale well.



The above graph shows runtime (Seconds vs # of CPUs and Problem Size (# of records) and is basically just the inverse of the Weak scalability graph.

Verification of model prediction accuracy

UserID	Recall at K	Random Recall at K
1	0.50	0.00
637	0.50	0.00
1469	0.66	0.00
710	0.50	0.00
1025	0.50	0.00

Figure 2.2

The figures in the table above are the result of running the model with five randomly selected user IDs, a problem size of 100,000 records, and K = 20 (that is, the model is allowed to make 20 streamer recommendations). As you can see, the recall at K from the content recommendation model is significantly better than the recall at K from 20 purely random guesses. This verifies that recommendations based on cosine similarity of watch preferences among nearest neighbours, which is the collaborative filtering approach my model used, yields much better results in terms of accuracy (measured by recall at K) than streamer recommendations based on randomness. In other words, the model works. Note that the recall at K is essentially just the percentage of how many of the held-out ground truths from the input data for the target user the model was able to correctly guess.

What affects recall at K (Model Accuracy)?

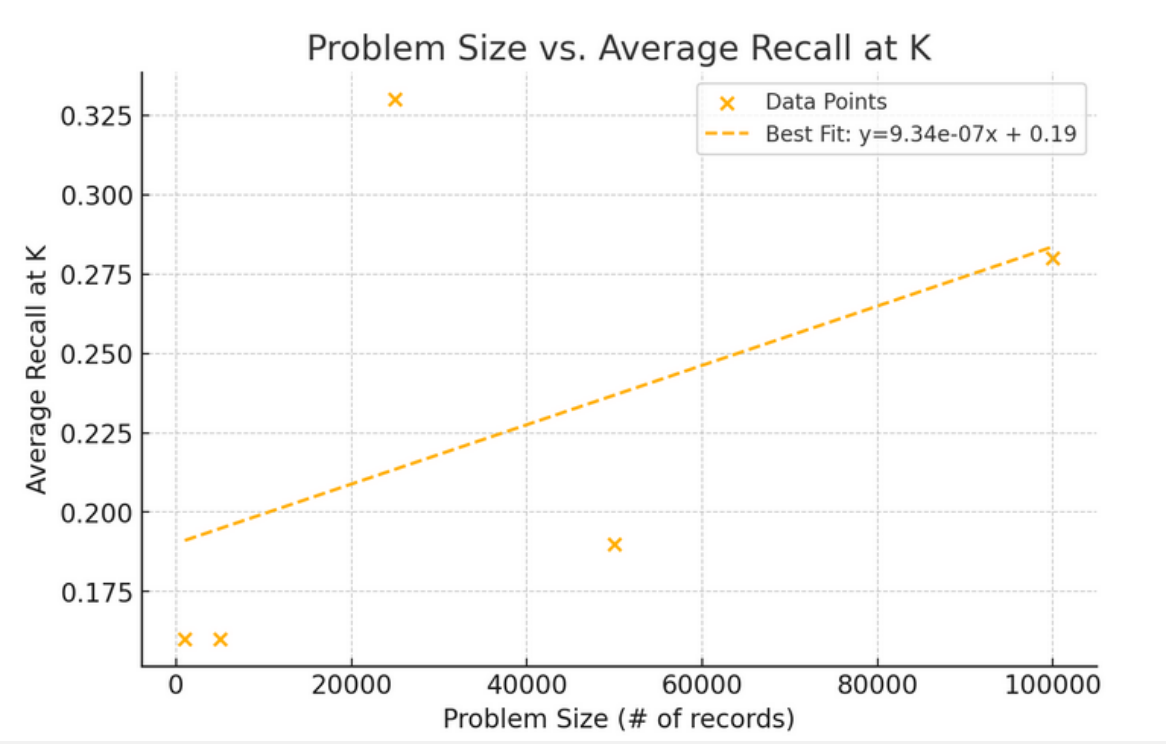
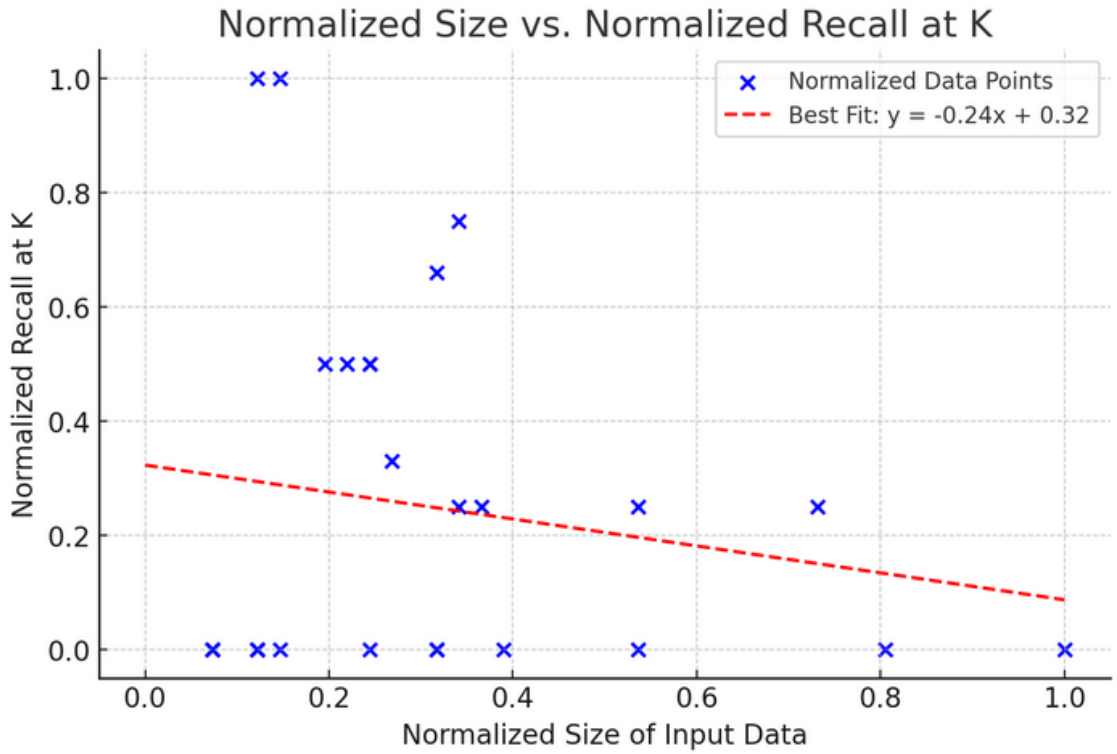


Figure 3

Figure 2 shows problem size (number of records) versus average Recall at K. As you can see, there is a clear positive correlation between problem size and Recall at K. Intuitively this makes sense, the more data the model has, the more close nearest neighbours it can find, and the more accurate its recommendations become.



The above graph shows normalized recall at K versus normalized size of input data, where K is 20. From the graph, there is a slight negative relationship between the amount of target user-related data the model receives and the recall-at-K value. This is somewhat surprising, because intuitively I would have thought that recovering ground truths / improving recall would become easier as the model receives more data about the user. However, this increase in data has little effect.

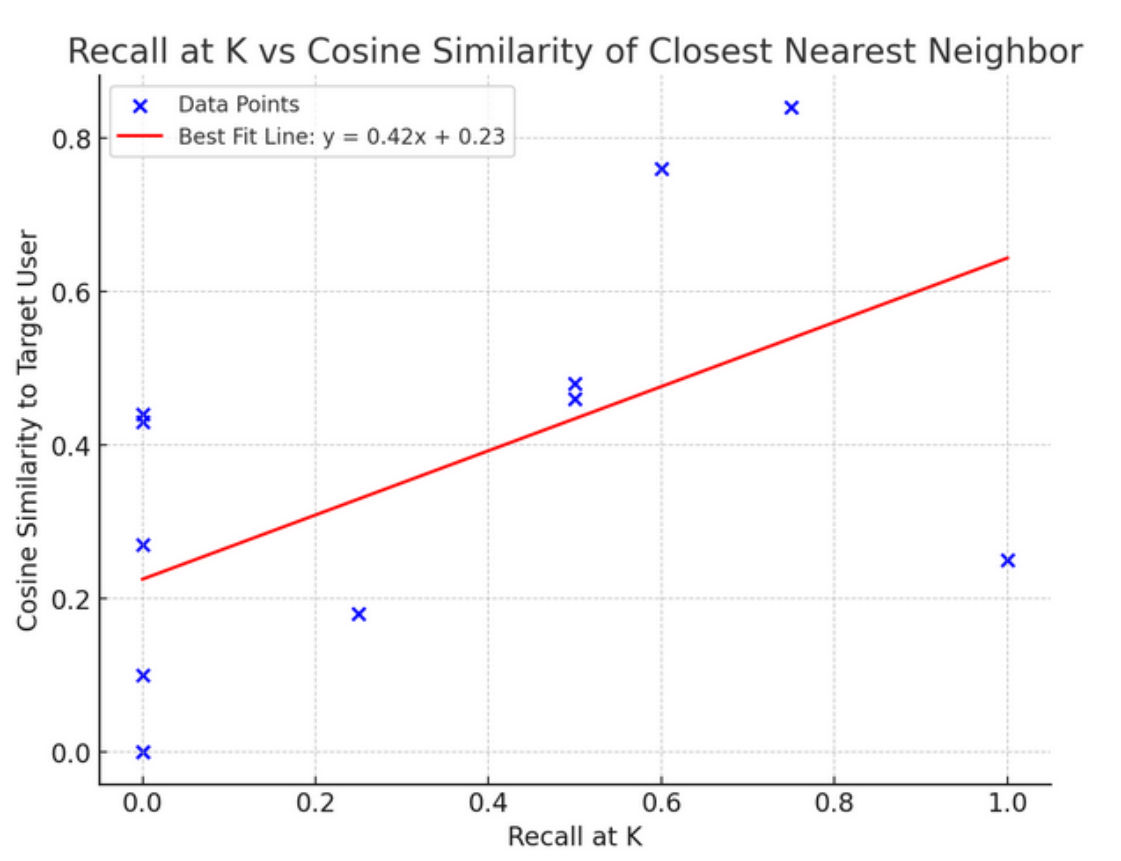


Figure 3: The above graph shows Recall at K versus the cosine similarity of the closest nearest neighbour to the target user where K = 20. There is a strong positive correlation between these two variables. This result makes sense because the model relies on collaborative filtering based on nearest neighbours. In other words, when the target user has very similar nearest neighbours, the model's recommendations become more accurate.

Discussion of research question:

“How well does user similarity predict the accuracy of Twitch-streamer recommendations, as measured by recall at K?”

Based on the data in the graphs above, we can answer this question: yes. Increasing user similarity to the target user does improve the accuracy of Twitch-streamer recommendations as measured by recall at K. The primary evidence for this is Figure 3, which shows a strong positive correlation between recall at K and the cosine similarity of the closest nearest neighbour to the target user. In other words, as the recommendation system receives target users whose nearest neighbours have higher cosine similarity, the accuracy (recall at K) of its recommendations increases. This assumption lies at the heart of the collaborative-filtering approach used in my recommender system, and it turns out to be correct.

Figure 1 also confirms this hypothesis, since recall at K values are much higher for recommendations generated by collaborative filtering than for those based on random guesses.

Conclusion

Were you able to answer your hypothesis / research questions? Explain how and why (or why not)?

Yes, I was able to answer my hypothesis / research questions. Based on the data in the graphs, increasing user similarity to the target user does improve the accuracy of Twitch-streamer recommendations as measured by recall at K. Figure 3 shows a clear positive correlation between recall at K and the cosine similarity of the closest nearest neighbour to the target user. In other words, when the recommendation system receives target users whose nearest neighbours have higher cosine similarity, the accuracy (as measured by recall at K) of its recommendations increases. This confirms the core assumption of the collaborative-filtering approach used in my recommender system. Figure 2.2 further supports this, since recall at K values are substantially higher for collaborative-filtering recommendations than for those based on random guessing. Because both figures align with the hypothesis, I was able to answer my research question affirmatively.

What implications do your results have?

The primary implications of my results are twofold. First, the project serves as a proof of concept for a collaborative-filtering recommender system for Twitch streamers. The fact that my system produces accurate recommendations and recovers ground truths at a much higher rate than random guesses shows that this approach is a valid basis for a recommender system. Second, the closeness of the nearest neighbours is crucial. As shown in Figure 3, there is a strong correlation between recall at K and the cosine similarity of the closest nearest neighbour to the target user. This intuitively makes sense, and it also highlights the importance of finding many close neighbours when building these types of recommender systems. It then follows that optimizing the content recommendation system to process more data is paramount. Improving the parallelization of the code would allow more user data to be handled, resulting in closer nearest neighbours and increasing recommendation accuracy. From a commercial perspective, more accurate streamer suggestions will boost watch time for Twitch customers and enhance the experience for both Twitch users and streamers. This would have positive implications for all parties involved: Twitch users, streamers, and Twitch itself.

What future questions or directions would you take with your project?

In the future, I would expand my project in two ways to increase the accuracy of the content recommender system. First, I would fully optimize the code's scalability so that the system can process more data more efficiently. This is the project's key weakness: the limited input data volume means fewer users are processed, fewer close nearest neighbours are found, and ultimately the recommendations are less accurate. Addressing this issue has massive room for improvement. Second, I would seek advice on tuning the recommender system, such as determining the optimal number of nearest neighbours to use and applying more rigorous statistical methods. The specific values I used in this project (for example, five nearest neighbours) were largely the result of guesswork, some research and trial and error. Although the recommendations appear reasonably accurate, I suspect the system is not optimally tuned. In the future, I would also be interested in implementing a hybrid content and collaborative filtering approach in my recommender system. This would generate more accurate suggestions for users with very few nearest neighbours, addressing a key weakness of the current system.

Critique of Design and Project

A key part of my design and methods that could have been better implemented is the creation of the streamer name–to–streamer ID lookup dictionary. From testing with timing

print statements, I know this is a major bottleneck in my code. This approach is overly sequential, does not parallelize well, and is not scalable. It limits performance, especially as the number of unique streamer names grows. If I were to rewrite the project, I would explore using Dask methods to work directly with streamer name strings during group-by and fold operations, potentially eliminating the need for an explicit lookup dictionary.

Another major flaw in the project design is the use of large NumPy arrays to store watch preferences for every user. A sparse vector representation, such as a Compressed Sparse Row (CSR) matrix, or even Dask arrays, would have been far more efficient. These alternatives would greatly reduce memory pressure on the workers and the recommendation system. Implementing both changes would increase scalability and allow the system to process more data, resulting in more accurate streamer recommendations

Reflection

useful concepts from the course for this project:

- Cosine similarity
- Collaborative Filtering
- Dask / big data processing methods
- Lab 4 (the collaborative filtering–based recommender for movies)
- NumPy methods and arrays
- Parallelisation
- Cloud computing

From this project, I gained a deeper appreciation for the power of big data. It was fascinating to see how accurate recommendations emerge by processing large datasets with Dask. The project reinforced the importance of optimizing code for scale and writing clean, bug-free code. Above all, it highlighted how essential it is to design code to run in parallel when working with large-scale applications, this was the main limitation in my implementation of the collaborative-filtering recommender system. Time management was also a major factor. I spent too much time on inconsequential tasks, which ultimately left me with less room to optimize the code. I also should have asked for help sooner, figuring out things like setting up Google Cloud buckets ended up being an unnecessary time sink.

References

Data Set: UCSD MCauley Lab. (n.d.). Twitch dataset. Retrieved June 2, 2025, from https://mcauleylab.ucsd.edu/public_datasets/gdrive/twitch/

Rappaz, J., McAuley, J., & Aberer, K. (2021). Recommendation on live-streamingplatforms: Dynamic availability and repeat consumption

Leskovec, J., Rajaraman, A., & Ullman, J. D. (2020). Mining of Massive Datasets (3rd ed.).Cambridge University Press. http://www.mmds.org/

Wikipedia contributors. (2025, April 30). Recommender system. In Wikipedia, The FreeEncyclopedia. https://en.wikipedia.org/wiki/Recommender_system

GeeksforGeeks. (n.d.). *Sparse Matrix and its representations | Set 1 (Using arrays and linked lists)*. Retrieved June 2, 2025, from https://www.geeksforgeeks.org/sparse-matrix-representation/

Optimization techniques for Big Data Systems [Video]. (n.d.). YouTube. https://www.youtube.com/watch?v=3e-adUYxrb8

PyParis 2017. (2017). Collaborative filtering for recommendation systems in Python [Video]. YouTube. https://www.youtube.com/watch?v=z0dx-YckFko

Collaborative filtering. (n.d.). In *Wikipedia*. Retrieved June 2, 2025, from https://en.wikipedia.org/wiki/Collaborative_filtering

Other students worked with: None