

Team Members:
Keertana Kappuram
Shweta Nalluri
Sreetama Chowdhury

DSC 202 : FINAL REPORT

PROBLEM STATEMENT

Twitch is an American live-streaming service, most popular for video games and broadcasts of esports (competitive professional gaming) competitions but also used as, among other things, a teaching and learning platform. Users can watch content in real time (as it's being streamed) or on demand after the fact. As of January 2025, Twitch ranks as the 30th most visited website globally and stands as the leading live-streaming video service overall. While this puts it a step ahead of competitors in terms of the streaming aspect, our goals for this project from a business perspective are to allow it to take further steps into the social networking potential of the site. Twitch users can subscribe to, follow, and "friend" each other, and our goal is to capitalize on the increasing prevalence of online fan communities and attempt to facilitate those kinds of social interactions on Twitch by making it easier for users to find people with similar areas of interest. We accomplish this by using user data to recommend other users to follow/befriend – an easier task if it's already known that the two have certain things in common.

DATASET AND DATABASE

The datasets used in this project were collected by the Stanford Network Analysis Platform (SNAP) in 2018 and consist of Twitch user-user networks representing gamers who stream in specific languages. In these networks, nodes represent individual users, and links represent mutual friendships between them. For our analysis, we utilized the English-language dataset, which comprises over 7,000 nodes and 35,000 edges. Each node in the dataset contains detailed information about the user, including their unique user ID, the number of days they have been active on the platform, whether they stream mature content, their total number of views, and whether they are a Twitch partner (a designation given to high-achieving, monetized streamers). To effectively manage and analyze this data, we imported it into two database systems: PostgreSQL and Neo4j. PostgreSQL was used to host the feature-based aspects of the data, such as user attributes, while Neo4j was employed to handle the relational aspects, particularly the friendships and connections between users. This dual-database approach allowed us to leverage the strengths of each system for different components of our analysis.

METHODOLOGY

I) Software Used:

For this project, the main libraries and databases that we used are:

1. PostgreSQL: Relation Database to store and query user profile data.
2. Neo4j: Graph database to store and query graph data.
3. Python: Programming language to preprocess data, load it into PostgreSQL and implement the recommendation logic.
4. Pandas: This is a python library we have used for data manipulation and analysis
5. Py2neo: This is another python library that was used for interacting with Neo4j.
6. Psycopg2: This is a PostgreSQL adapter for python to interact with the database.

As mentioned above, we have implemented this project using 2 different databases, namely Neo4j and PostgreSQL. This section describes how we used these two databases and what we accomplished through these.

II) Neo4j - Collaborative Filtering

Neo4j is a graph database that is useful in storing and processing data. It accomplishes this through nodes, relationships and properties. Cypher is the query language used in Neo4j to perform operations like complex pattern matching or do any analysis of data. It is widely used on real-time data in social network analysis, knowledge graph or recommendation system like the one we implemented.

In our project, Neo4j is used to perform **Collaborative filtering** for recommendation. We used Neo4j for storing user interactions as a graph structure. This enabled us to analyse user connections and find similar users based on their following patterns.

In Neo4j, we loaded the file that represents the connections between the users. The graph structure was created by specifying each user as a node. Each user is assigned with their properties and attributes like the number of days a user has been active, the number of views the user has and other such attributes. The connections were established between the user. This was a '**FOLLOWS**' relationship indicating the connection between the users. This relationship is not symmetrical which implies that if user1 follows user2 does not imply that user2 must also be user1.

One of the most important libraries used was the Neo4j's Graph Data Science(GDS) library. This library is useful for running advanced graph algorithms and performing similarity calculations, link predictions and other such tasks.

A. Node Similarity Algorithm

Through Neo4j, we implemented graph algorithms to analyse the user relationships and recommend similar users based on their 'FOLLOWS' patterns. The graph algorithm we used was Node similarity Algorithm. The aim of Collaborative filtering was to recommend users to out target users that are mutually connected to them and not already being followed by the target user.

Node Similarity algorithm in Neo4j's Graph Data Science (GDS) library is ideal for identifying similar nodes in a graph based on their relationships. This algorithm initially compares two nodes and then computes a similarity score based on the number of connections they share.

The two main steps followed by the algorithm in the project are:

- **Pairwise Comparison:** For every target node, it is compared with every other user node based on the 'FOLLOWS' relationship.
- **Similarity Score:** The similarity between the nodes is computed based on the extent of overlap in the follow list.

The core metric we have used to measure the similarity between nodes based on their follow list in **Jaccard Similarity**. This is a metric that is used by default in the Node Similarity Algorithm. This is a widely used metric for measuring similarity. It is computed as follows:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

The numerator indicates the number of common users both user A and user B follow. The denominator represents the total number of unique user both user A and user B follow.

A high Jaccard similarity score shows that the similarity is high.

III) PostgreSQL - Context Based Filtering

The other database that we have used in this project is PostgreSQL. This is a relational database management system that is very efficient in handling complex queries and advanced data processing. It is also very good at handling a growing dataset without any degradation in performance.

In this project, we use PostgreSQL to store the data we have in the files and query the data which basically is the features of the user as these are important for the context based filtering recommendation.

We store the user information which are the attributes of each user. We stored information like views, partner status, the number of days they have been on twitch, maturity in a table named 'users'. The recommendation is done based on the similarity in these attributes of the users.

The similarity between the target user and every other user in the database is computed by finding the cosine similarity on their features. Cosine Similarity is calculated as follows:

- **Dot product:** The first step we did was to get the dot product of the feature vector of the target user with that of all the other users. It is calculated as:

$$\text{dot}(A,B) = A_i B_i$$

- **Magnitude:** We next found the magnitude of the feature vector of all the users that we have in the database

$$\text{Magnitude}_A = \sqrt{A_i^2}$$

- **Cosine Similarity:** After calculating the Dot product and Magnitude, we calculated the Cosine Similarity between the target user and every other user.

$$\text{Cosine}(A,B) = \frac{\text{dot}(A,B)}{\text{Magnitude}_A * \text{Magnitude}_B}$$

The value of the cosine similarity shows how similar two users are based with respect to their attributes. Cosine similarity ranges between 0 to 1, with a score of 1 indicating high similarity.

IV) Final Recommendation Score

For a target user, after obtaining the similar users based on Collaborative filtering approach and Context-based filtering approach, we hybridized these two outputs.

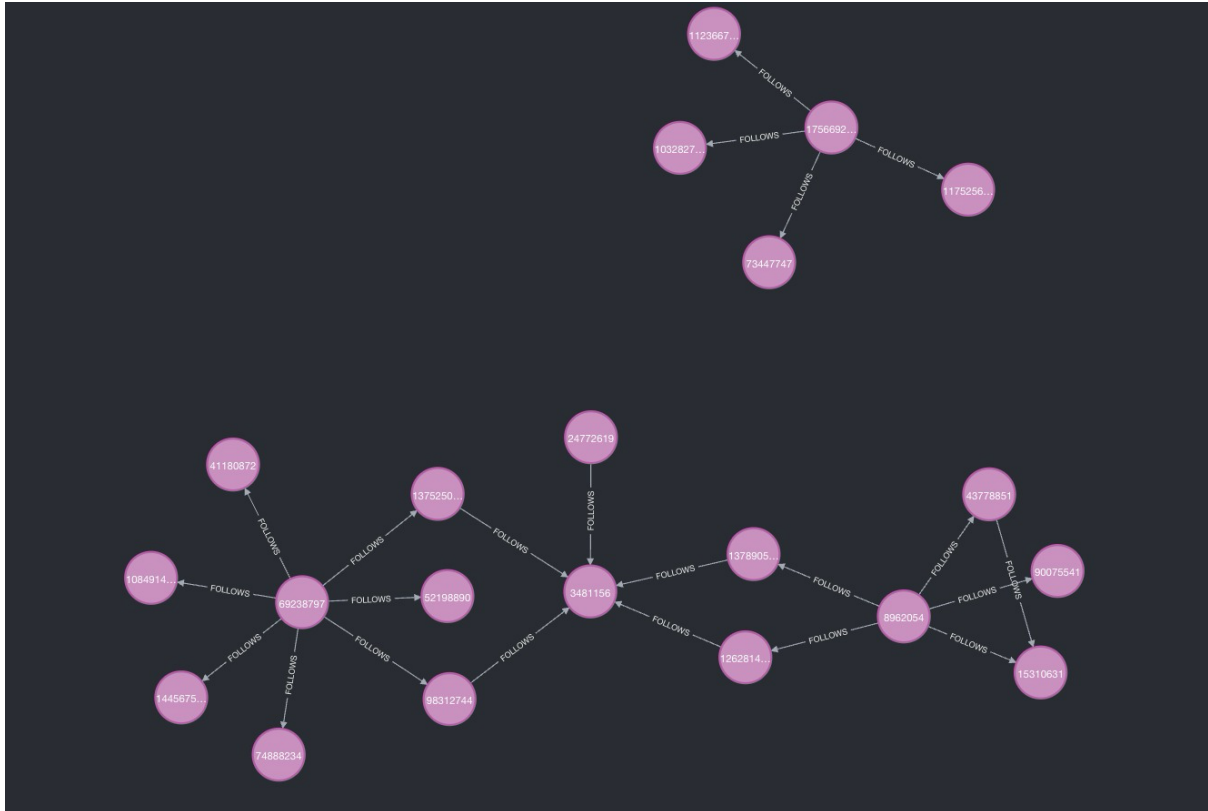
To hybridize, we joined the recommendations from both the models on the user and the recommended IDs ensuring we are comparing the same pairs across both the models. The final recommendation score was calculated by taking the weighted average of the similarity scores of both the approaches. Each approach was weighed at 50%. After obtaining the final recommendation scores, we ranked the scores in descending order and returned top 5 most relevant recommendations.

This hybridization gives us a more personalized recommendation as it allows us to consider both user interaction patterns and the profile-based similarity between users.

CODE EXPLANATION AND RESULTS

I) Neo4j:

In Neo4j, we created a local DBMS. We then imported the required data source file 'musae_ENGB_edges.csv' into Neo4j. The graphical representation of the data is as follows:



We continue by importing the Graph Data Science Library plugin. After this, we start loading the 'musae_ENGB_edges.csv' file into the actual database which has the relationships between each user that follows another user. We load the csv file containing the connection between the users. By using the MERGE command, we create unique nodes for each user in the database. We established a connection between each pair of users by creating a 'FOLLOWS' relationship between them. This is how we structured the given data of the csv file as a graph.

Code snippet:

```
LOAD CSV WITH HEADERS FROM 'file:///musae_ENGB_edges.csv' AS row
MERGE (u1:User {new_id: toInteger(row.from)})
MERGE (u2:User {new_id: toInteger(row.to)})
CREATE (u1)-[:FOLLOWS]->(u2);
```

We created an in-memory representation of the graph by creating a graph projection using the Graph Data Science library. We are basically projecting the users as nodes and their 'FOLLOWS' relationship. The orientation was specified as "NATURAL" so the relationships direction does not change through the projection.

Code snippet:

```
CALL gds.graph.project(  
  'userGraph', // Name of the graph projection  
  'User', // Node label  
  { FOLLOWS: { type: 'FOLLOWS', orientation: 'NATURAL' } } // Relationship type  
  and direction);
```

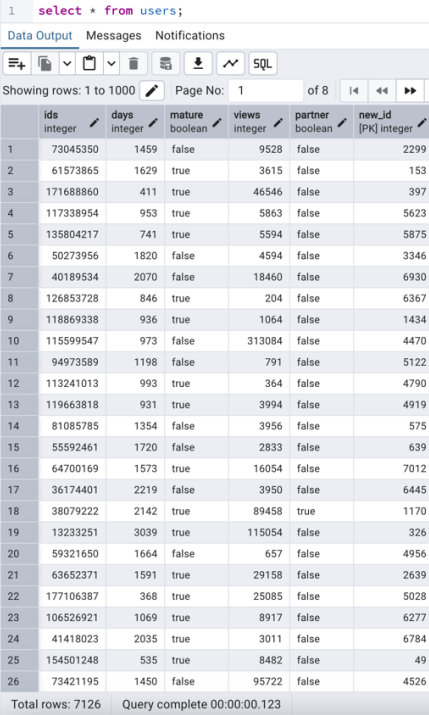
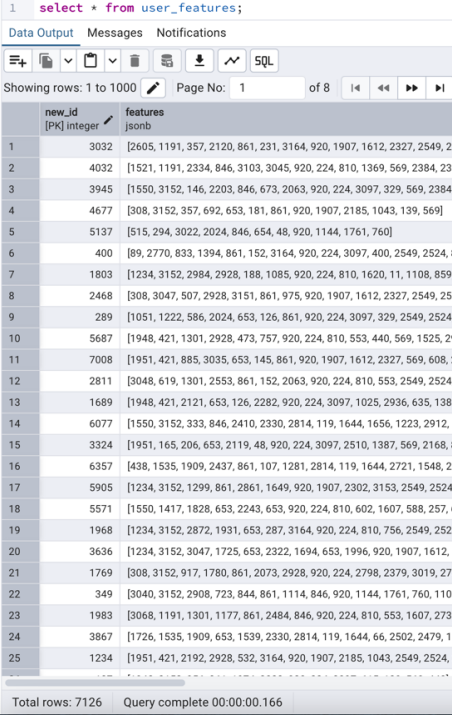
Output:

	nodeProjection	relationshipProjection	graphName	nodeCount	relationshipCount	projectMillis
1	{ "User": { "label": "User", "properties": { } } } }	{ "FOLLOWS": { "aggregation": "DEFAULT", "orientation": "NATURAL", "indexInverse": false, "properties": { } }, "type": "FOLLOWS" } }	"userGraph"	7126	105972	96

Started streaming 1 records after 4 ms and completed after 121 ms.

II) PostgreSQL:

Following are the tables we have created and used for our recommendations on PostgreSQL:

user_features(musae_ENGB_features.json)	"users" (musae_ENGB_target.csv)
	

Insertion of data into these tables has happened using Python- We first connect to the PostgreSQL database using the library 'psycopg2' and then loaded the required json file/csv file into tables created beforehand in PostgreSQL.

Code snippet:

Creation of table-

```
CREATE TABLE user_features (  
    new_id SERIAL PRIMARY KEY,  
    features JSONB NOT NULL  
);
```

Insertion of data into table-

```
import psycopg2  
import json  
  
# Database connection  
conn = psycopg2.connect("dbname=twitch_recommendations user=postgres  
password=postgres host=localhost port=5433")  
cursor = conn.cursor()  
  
# Load JSON file  
with open("/Users/keertanakappuram/musae_ENGB_features.json", "r") as f:  
    user_features = json.load(f)  
  
# Insert data  
for user_id, features in user_features.items():  
    cursor.execute("""  
        INSERT INTO user_features (new_id, features)  
        VALUES (%s, %s)  
        ON CONFLICT (new_id) DO NOTHING;  
        """, (user_id, json.dumps(features)))  
  
# Commit & close  
conn.commit()  
cursor.close()  
conn.close()
```

III) Database Connection:

Code snippet-

```
# Function to get the database connection
def get_db_connection():
    conn = psycopg2.connect(
        host="localhost",
        dbname="twitch_recommendations",
        user="postgres",
        password="postgres",
        port="5433" )
    return conn
```

We use this code to connect our PostgreSQL database to our python code, using parameter values like hostname, database name, username, password and port number.

Code snippet-

```
URI = "bolt://localhost:7687"
AUTH = ("neo4j", "postgres")

driver = GraphDatabase.driver(URI, auth=AUTH)

def fetch_neo4j_results(query, parameters=None):
    with driver.session() as session:
        result = session.run(query, parameters)
        return [dict(record) for record in result]
```

This code is used to connect Neo4j to Python and fetch_neo4j_results function is used to execute Cypher queries and return results conveniently.

IV) Recommendation Logic:

1. **Content/Feature based recommendations:** We are making feature-based recommendations on the basis of the cosine similarity score between one user and every other user in the database, in order to understand which 2 users have the highest cosine similarity score, so as to recommend such users to each other.

The main logic of this type of recommendation is in the function called "get_cosine_similarity_recommendations"

In order to achieve this, we created 3 Common Table Expressions (CTEs), which are as follows:

- (a) user_features_final : This combines all the features used to make predictions from both tables.
- (b) dot_product: As the cosine similarity formula's numerator requires us to find the dot product of features between 2 users, we are calculating that here.

(c) magnitude: As the cosine similarity formula's denominator requires us to find the magnitude of each user's features, we are calculating that here.

Code snippet for calculating dot product:

```
With dot_product AS (  
  SELECT  
    uf1.new_id AS user_id_1,  
    uf2.new_id AS user_id_2,  
    (  
      CAST(uf1.views AS FLOAT8) * CAST(uf2.views AS FLOAT8) +  
      CAST(uf1.partner AS FLOAT8) * CAST(uf2.partner AS FLOAT8) +  
      CAST(uf1.days AS FLOAT8) * CAST(uf2.days AS FLOAT8) +  
      CAST(uf1.mature AS FLOAT8) * CAST(uf2.mature AS FLOAT8) +  
      COALESCE(  
        SELECT SUM(f1::FLOAT8 * f2::FLOAT8)  
        FROM jsonb_array_elements(uf1.features) WITH ORDINALITY AS  
t1(f1, ord1)  
        JOIN jsonb_array_elements(uf2.features) WITH ORDINALITY AS t2(f2,  
ord2)  
        ON t1.ord1 = t2.ord2  
      ), 0)  
    ) AS dot_prod  
  FROM user_features_final uf1  
  JOIN user_features_final uf2  
    ON uf1.new_id != uf2.new_id  
  WHERE uf1.new_id = {target_user_id}  
)
```

Once these calculations are done, the function returns the recommended user's id and cosine similarity value.

Code snippet:

```
SELECT  
  dp.user_id_2 AS recommended_user,  
  dp.dot_prod / (m1.magnitude * m2.magnitude) AS cosine_similarity  
FROM dot_product dp  
JOIN magnitude m1 ON dp.user_id_1 = m1.new_id  
JOIN magnitude m2 ON dp.user_id_2 = m2.new_id  
WHERE dp.user_id_1 != dp.user_id_2  
ORDER BY cosine_similarity DESC;
```

2. Inserting the feature-based recommendations back into the database:

We are inserting the recommendations and their similarity scores back into the database in a table called 'recommendations_content' to ensure they are reusable. For this, we are using 'psycopg2' again.

Code Snippet:

```
# Function to insert recommendations into the database
def insert_recommendations(user_id, recommendations):
    cursor = conn.cursor()

    # Create recommendations table if it doesn't exist
    cursor.execute("""
    CREATE TABLE IF NOT EXISTS recommendations_content (
        user_id INT,
        recommended_user INT,
        cosine_similarity FLOAT8
    );
    """)

    # Prepare data for insertion (convert NumPy floats to Python floats)
    data_to_insert = [(user_id, int(row['recommended_user']),
float(row['cosine_similarity'])) for _, row in recommendations.iterrows()]

    #insertion query
    insert_query = """
    INSERT INTO recommendations_content (user_id, recommended_user,
cosine_similarity)
    VALUES (%s, %s, %s);
    """

    # Example usage:
    target_user_id = 6194
    recommendations = get_cosine_similarity_recommendations(target_user_id)

    # Insert recommendations into the database
    insert_recommendations(target_user_id, recommendations)
```

```
1 select * from recommendations_content;
```

Data Output Messages Notifications

	user_id integer	recommended_user integer	cosine_similarity double precision
1	6194	176	0.9879848951133936
2	6194	1659	0.9879660181922227
3	6194	6642	0.9878655564631159
4	6194	2252	0.9873921222391336
5	6194	6020	0.9870874108932917
6	6194	2175	0.9870467953308458
7	6194	4901	0.9864241245011703
8	6194	2248	0.9863311538540516
9	6194	156	0.986284236124971

3. Collaboration based recommendations:

We are making collaboration-based recommendations on the basis of the metric-Jaccard similarity. The logic lies in the function called “get_user_recommendations”. The code snippet for this logic is as follows-

```
def get_user_recommendations(user_id, similarity_threshold=0.0):
    query = """
    CALL gds.nodeSimilarity.stream('userGraph')
    YIELD node1, node2, similarity
    WITH gds.util.asNode(node1) AS user1, gds.util.asNode(node2) AS user2,
    similarity
    WHERE user1.new_id = $user_id AND similarity > $threshold
    AND NOT EXISTS { MATCH (user1)-[:FOLLOWS]->(user2) }
    RETURN user2.new_id AS recommended_user, similarity
    ORDER BY similarity DESC
    """

    parameters = {"user_id": user_id, "threshold": similarity_threshold}
    return fetch_neo4j_results(query, parameters)
```

4. Inserting the collaboration-based recommendations back into the database:

Like the results we inserted for feature-based recommendations, we are inserting the collaboration-based recommendations and their similarity scores back into the database in a table called ‘recommendations_collab’ to ensure they are reusable.

```
1 select * from recommendations_collab;
```

	user_id [PK] integer	recommended_user [PK] integer	similarity double precision
1	6194	2756	0.1
2	6194	708	0.1
3	6194	3596	0.09090909090909091
4	6194	3532	0.07692307692307693
5	6194	1695	0.07692307692307693
6	6194	1133	0.07692307692307693
7	6194	2444	0.05555555555555555
8	6194	1501	0.05555555555555555
9	6194	2669	0.037037037037037035

5. Final score calculation:

The final score calculation is done by giving 50% weightage to the collaboration-based recommendation jaccard similarity score and 50% weightage to the feature-based recommendation cosine similarity score, to calculate the final score. The logic behind this code is in the function called "get_combined_recommendations". The following query returns the result which is the user_id, recommended user_ids, Collaboration based recommendation similarity score, Feature/content-based recommendation similarity score and final score. The results are ordered in the descending order of the final score as the highest score is the best recommendation and we are giving the top 5 recommendations.

Code snippet:

```
SELECT
  rc.user_id,
  rc.recommended_user,
  rc.similarity AS similarity_score_collab,
  rcon.cosine_similarity AS similarity_score_context,
  (0.5 * rc.similarity + 0.5 * rcon.cosine_similarity) AS final_score
FROM recommendations_collab rc
JOIN recommendations_content rcon
ON rc.user_id = rcon.user_id
AND rc.recommended_user = rcon.recommended_user
GROUP BY rc.user_id, rc.recommended_user, rc.similarity, rcon.cosine_similarity
ORDER BY final_score DESC
LIMIT 5;
```

V) OUTPUT-

Following are the recommendations for user 6194 :

	user_id	recommended_user	similarity_score_collab	similarity_score_context	final_score
0	6194	3596	0.090909	0.961712	0.526310
1	6194	2756	0.100000	0.934258	0.517129
2	6194	1501	0.055556	0.961471	0.508513
3	6194	2444	0.055556	0.959891	0.507723
4	6194	2669	0.037037	0.960227	0.498632

LESSONS LEARNED

This project highlighted the importance of selecting the right database for different tasks. Neo4j was exceptionally effective in managing and querying graph-based relationships, such as friendships, while PostgreSQL proved to be well-suited for handling structured, feature-based data like user attributes. Through the project, we gained a deeper understanding of the power of graph-based algorithms in recommendation systems, as Neo4j allowed for efficient traversal of user networks to identify potential connections. This experience was particularly educational because, while we had some familiarity with PostgreSQL, working with graph databases and understanding their capabilities was entirely new to us.

FUTURE WORK

In the future, we plan to expand our work with the SNAP dataset by including data beyond just English-language sets and incorporating more recent data. We also envision extending the project to incorporate additional user attributes, such as favorite game genres and real-time data like live comments on streams they watch. Moreover, by adding more types of relationships among users, we hope to tackle more complex problem statements and offer recommendations based on a wider variety of criteria, which will allow for more personalized and accurate suggestions.