Progetto di New Generation Data Models and DBMSs

Alessia Cecere

1 Introduzione

Il progetto che andrò ad esporre riguarda l'implementazione di un sistema di raccomandazione per film, tramite l'utilizzo di un database a grafo in Neo4j [6].

Un sistema di raccomandazione è, brevemente, un software di filtraggio di contenuti che ha l'obiettivo di creare raccomandazioni personalizzate per l'utente: il suo principale scopo è, dato un utente e un item che non conosce, prevedere quanto quest'ultimo possa essere interessante per l'utente.

Esistono due principali tecniche per la raccomandazione di contenuti.

- **Collaborative filtering**: sfrutta il comportamento passato degli utenti, assumendo che si protragga anche nel presente; viene utilizzato principalmente in due varianti.
  - **User-based**: dato un utente, viene identificato un insieme di *utenti simili* (persone che hanno dato rating simili o hanno visualizzato gli stessi item). L'apprezzamento da parte di questi utenti per il contenuto viene utilizzato per prevedere quello dell'utente d'interesse.
  - **Item-based**: la similarità viene calcolata tra gli item (assumendo che item simili ricevano rating simili/vengano visualizzati dagli stessi utenti). A un utente vengono consigliati item simili a quelli che ha valutato positivamente.
- **Content-based filtering**: viene creato un *profilo dell'utente* tramite gli item che ha valutato, mentre gli item vengono rappresentati mediante loro specifiche caratteristiche. All'utente vengono raccomandati item sulla base della similarità tra il suo profilo e la rappresentazione degli item.

Come possiamo vedere, la raccomandazione dipende in primo luogo dalle relazioni tra gli oggetti: quelle tra utenti e quelle tra un utente e ciò che ha comprato/valutato. Sappiamo che è possibile modellare questo dominio anche in un database relazionale, ma per attraversare le sue relazioni dovremmo ricorrere a query SQL che praticano join computazionalmente dispendiosi. Le performance peggiorano mano a mano che i join, gli utenti e gli item aumentano, rendendo questa soluzione poco pratica per grosse quantità di dati. I database

a grafo, al contrario, si caratterizzano per la velocità di attraversamento delle relazioni, e anche per la loro capacità di caratterizzarle: utilizzando un grafo etichettato, è facile dare peso maggiore o minore alla connessione tra gli oggetti, in modo da percorrere i path in maniera significativa e trovare dei pattern.

Un'altra struttura dati frequentemente utilizzata per il problema della raccomandazione è la *user-item matrix*, ovvero una matrice che ha per righe gli utenti e per colonne gli item, e contiene i rating degli utenti per gli item. Un problema notevole di tale rappresentazione è che tende ad essere sparsa (con molti zeri, perché per ogni utente ci sono molti item che non ha valutato e viceversa), e dunque inutilmente dispendiosa in termini di memoria. La rappresentazione a grafo, al contrario, è densa d'informazione, dal momento che due oggetti sono connessi solamente se sono effettivamente in una relazione significativa tra di loro.

Infine, i database a grafo sono adatti ai sistemi di raccomandazione online (in opposizione a quelli pre-calcolati periodicamente offline): il ricalcolo di una raccomandazione in real time sulla base, ad esempio, della creazione di nuove relazioni, è in grado di dare all'utente un'esperienza migliore e più personalizzata.

2 Dataset

Come dataset di partenza per il mio sistema di raccomandazione, ho utilizzato il *MovieLens 25M Dataset* [5], contenente 25 milioni di rating, 62.000 film e 162.000 utenti, per un totale di circa 1200 MB di dati.

I CSV che ho usato per il progetto sono i seguenti.

- `movies.csv`: elenco di film con i loro `movieId`, `title`, e `genres` (una stringa in cui i generi sono separati dal simbolo |, ad esempio "Comedy | Romance").
- `ratings.csv`: insieme di rating, con le colonne `userId`, `movieId`, `rating` e `timestamp`.
- `genome-tags.csv`: insieme di categorie con il loro `tagId` e `name`. Le categorie provengono dal modello *tag genome* [13], che codifica l'intensità con cui i film sono associati ad alcune proprietà (*atmospheric*, *thought-provoking*, *realistic*, eccetera). Il *tag genome* è stato computato con un algoritmo di machine learning su contenuti creati da utenti, che includevano tag, rating e recensioni testuali.
- `genome-scores.csv`: tabella che contiene le associazioni tra tag e film, con le colonne `movieId`, `tagId` e `relevance`.

3 Diagramma UML

A partire dal dominio del dataset sopra riportato, ho costruito il diagramma UML in Figura 1.

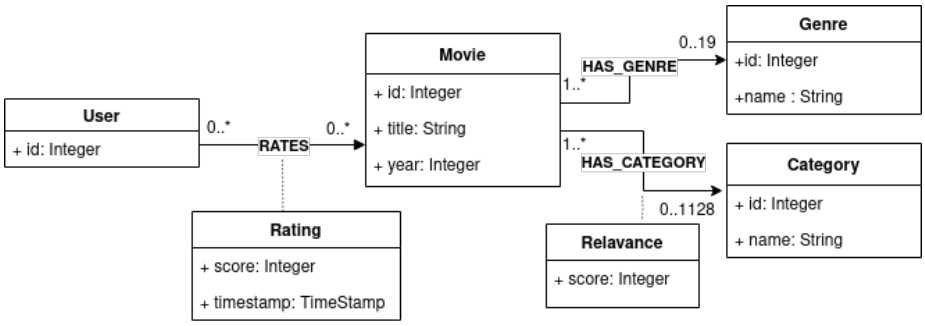

Figura 1: Diagramma UML

3.1 Vincoli

Ho individuato i seguenti vincoli.

- Possono essere presenti **User** che non hanno dato **nessun rating** ai film. Nella pratica, nel mio progetto non ci saranno, perché i dati vengono dal dump, e nel dump un utente è presente solo se presente nella tabella `rating`. Ho pensato, però, di includere questa possibilità nel diagramma UML perché in un sistema reale sarebbe frequente avere nuovi utenti. Per questi utenti esiste un noto problema, nel mondo della raccomandazione, chiamato **cold start problem** (non sappiamo cosa raccomandare loro, dal momento che non hanno effettuato nessuna azione né espresso alcun giudizio). Solitamente, si raccomandano gli item più popolari (query inclusa nel worflow, a cui farò riferimento successivamente).

- Ho seguito lo stesso criterio per i **Movie**, che, soprattutto in un'ottica collaborativa, vengono aggiunti frequentemente.

- L'attributo **score** nell' **entità-relazione rating** è un valore intero compreso tra **1** e **5**. Ho scelto di rappresentare il rating in questo modo perché si tratta di un'entità che vive solamente nel contesto della relazione tra utente e film. Con questa rappresentazione, non vi possono essere più rating dati dallo stesso utente per lo stesso film. Credo che il vincolo abbia senso, perché non si sta parlando di recensione ma di giudizio complessivo relativo al film: nel caso cambiasse, rappresenta il fatto che il vecchio giudizio cessi di esistere e venga sostituito dal nuovo.

- Il **genre name** può assumere solo i seguenti valori:

  [Action, Adventure, Animation, Children's, Comedy, Crime, Documentary, Drama, Fantasy, Film-Noir, Horror, Musical, Mystery, Romance, Sci-Fi, Thriller, War, Western, IMAX].

  Un Movie può non essere associato a nessun Genre, ma un Genre esiste solo se relazionato ad almeno un Movie (in quanto sua descrizione). Un Movie

può essere in relazione con al massimo **19 Genre**, ovvero la cardinalità del loro insieme.

- Il **category id** e il **category name** possono assumere come valori solo i **1128 genome tags** salvati nel dataset. Nel dataset ogni Movie è associato a ogni categoria e viceversa (dunque le cardinalità sono 1128 e 62.000); per ottimizzare i tempi di computazione, e dal momento che siamo interessati solo ai collegamenti rilevanti, ho deciso di mantenere la relazione tra Category e Movie solo se caratterizzata da una **relevance** $\geq$ **0.4**.

  La **relevance** è una classe associazione nella relazione tra **Movie** e **Category** (per le stesse motivazioni e con gli stessi vincoli di quanto avveniva per rating); il suo attributo **score** può assumere valori floating point tra **0** e **1**.

3.2 Verso delle relazioni

Il verso delle relazioni nel diagramma UML è stato pensato per ottimizzare il seguente **workflow**.

1. Dato un **User**, trovare i suoi **top k Genres**
2. Dato un **User**, trovare le sue **top k Categories**
3. Dato un **Genre**, trovare i suoi **top k Movies**
4. Data una **Category**, trovare i suoi **top k Movies**
5. Dato un **User**, trovare gli **User simili**
6. Dato un **User**, raccomandare **Movies** sulla base degli **User simili** (*collaborative filtering*)
7. Dato un **Movie**, trovare **Movies** simili
8. Dato un **User**, raccomandare **Movies** sulla base della **similarità** con quelli che ha guardato (*content-based*)

- **Relazione tra User e Movie**: abbiamo bisogno di navigarla in entrambi i versi, esattamente per lo stesso numero di query (nel senso da **User** a **Movie** per le query **1** e **2**, in quello opposto per le query **3** e **4**, mentre per le altre servono entrambe le direzioni). Ho dunque deciso di riportarla da **User** a **Movie** nel diagramma UML sia per una ragione di numero di riferimenti per entità (ci aspettiamo di avere meno film per utente che viceversa), sia per una ragione di significato (è l'**User** che dà il **rating** al **Movie**).
- **Relazione tra Movie e Genre**: il verso da **Genre** a **Movie** viene navigato per rispondere alla query **3**, mentre quello opposto è utilizzato

dalla query **1** e anche per calcolare la similarità tra **Movies** per le query **7** e **8** (il **Genre** è una delle caratteristiche principali che contraddistingono il **Movie**). Oltre ad essere il più utilizzato nel workflow, quest'ultimo verso della relazione è anche quello che permette di specificare il numero minore di associazioni per entità: come abbiamo espresso prima nei vincoli, un **Movie** può essere in relazione con al massimo **19** Genres, mentre un **Genre** potrebbe potenzialmente essere associato a tutti i 62.000 Movies.

- **Relazione tra Movie e Category**: valgono gli stessi ragionamenti fatti precedentemente per il genere, prendendo in considerazione le query **4** e **2** (al posto di 3 e 1). Sebbene la soglia sulla **relevance** riduca notevolmente il numero di relazioni, c'è comunque una notevole differenza: un **Movie** ha associate molte meno **Categories** di quanti **Movies** abbia associati una **Category**.

4 Modello logico

Per tradurre il modello UML in un un database a grafo, ho individuato la seguente struttura.

Nodi

- **User**
  properties: id (string)
- **Movie**
  properties: id (string), title (string), year (int)
- **Genre**
  properties: id (string), name (string)
- **Category**
  properties: id (string), name (string)

La principale novità è l'introduzione degli **id** come stringhe, al posto degli interi che si trovavano nel dataset originale. Generalmente viene sconsigliato di utilizzare come identificatori gli id di default di Neo4j, perché questi sono progressivi, ma nel momento in cui un nodo viene cancellato possono essere riassegnati. Questo fenomeno fa sì che si abbia poco controllo su di essi al momento della creazione di un nodo, col rischio di non riuscire a tracciarlo correttamente. Gli id presenti nel database, tuttavia, erano interi progressivi che partivano da 1 (e questo per ogni tabella): dal momento che, pur non utilizzando gli id di default, volevo essere in grado di mantenere l'unicità dell'id per ogni elemento del database, ho deciso di utilizzare degli **UUID** (stringhe di 16 byte con 32

caratteri esadecimali, visualizzati in cinque gruppi separati da trattini, nella forma 8-4-4-4-12).[4]

Relazioni

- **RATES** (da User a Movie)
  properties: rating (int)
- **HAS_GENRE** (da Movie a Genre)
- **HAS_CATEGORY** (da Movie a Category)
  properties: relevance (float)

Grazie al modo in cui Neo4j organizza i dati, la velocità di attraversamento delle relazioni non dipende dalla loro direzione: l'analisi fatta precedentemente è stata tuttavia molto utile per decidere come rappresentarle, in modo da ottimizzare il numero di relazioni a partire da un nodo e la chiarezza del modello a grafo.

In Figura 2 un esempio di istanziazione del modello sopra descritto.

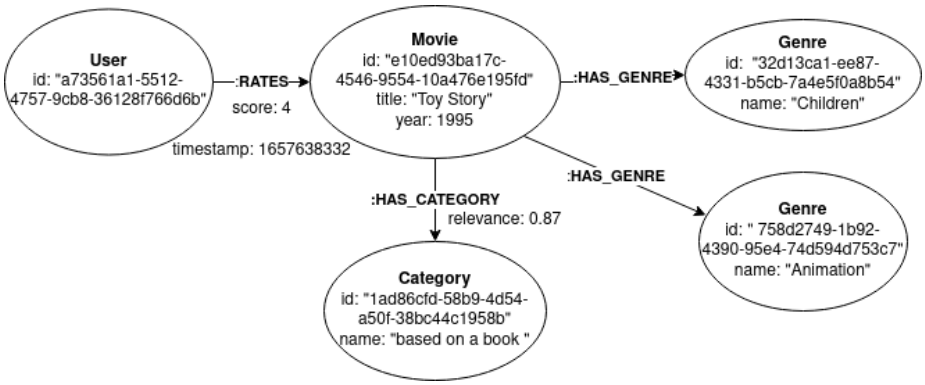

Figura 2: Istanziazione modello a grafo

5 Script di creazione del database

Per integrare Neo4j all'interno di script Python, ho utilizzato la libreria *py2neo* [11]. Lo script `populate_db.py` assume che esista già, sulla macchina che lo esegue (la connessione avviene a *localhost*), un database Neo4j e che questo sia attivo. Successivamente lo script chiede *username*, *password* e *porta* su cui il database d'interesse sta ascoltando, effettuando la connessione tramite questi valori.

5.1 Pre-processing

Una parte dello script svolge un lavoro di preprocessing sui CSV contenuti nel dataset: la funzione `preprocess_movies` elimina alcuni caratteri speciali, che causano conflitti nell'inserimento, e aggiunge una nuova colonna *year*, estraendola dal titolo (precedentemente nella forma *Movie(year)*), mentre `preprocess-ratings` trasforma in `int` lo score di un utente per un film (precedentemente espresso con un `float` a cifre decimali 0).

5.2 Gestione degli UUIDs

Un'altra componente si occupa di tradurre gli id presenti nei CSV in UUIDs, come precedentemente descritto: la funzione `create uuid_associations` crea una dizionario Python che ha come chiavi gli id contenuti nel dataset originale e come valori i corrispettivi UUID, contestualmente generati tramite il modulo Python *uuid*. [12]

Questa funzione viene applicata agli **users** (lista di elementi unici del `userId` della tabella `rating.csv`), ai **Genres** (lista di stringhe che ne rappresentano i nomi, precedentemente salvata nella costante `GENRES`), alle **Categories** (i `tagId`) e ai **Movies** (`movieId`).

Dopo essere stati generati, tutti i dizionari vengono serializzati in file `pickle` [8], così da garantire che si possa tenere traccia delle corrispondenze.

5.3 Creazione dei nodi

Tutti i nodi sono stati creati tramite la funzione `create_node`, che riceve in input la *label* associata al nodo e un dizionario di *properties*: attraverso l'iterazione su tale dizionario, la funzione wrappa un'operazione `CREATE` di Neo4j, integrata tramite la funzione di py2neo `graph.run`. Gli id vengono inseriti nel dataset scorrendo la tabelle CSV e ricavando dai dizionari precedentemente creati i rispettivi UUIDs.

5.4 Creazione delle relazioni

A causa delle molteplici associazioni dei nodi di tipo **Movie** e della sostituzione degli UUIDs, le relazioni sono state inserite dopo i nodi, facendo prima una query per identificarli e solo creando l'associazione. Questo approccio ha portato all'introduzione di **indici** sugli id di **Movie**, **Genre**, **User** e **Category**, tramite la funzione `create_node_index`, per garantire una creazione più veloce. La relazione tra **Movie** e **Genre**, meno dispendiosa in termini di numero di associazioni (107.245), è stata creata tramite la funzione `create_relationship`, che wrappa una `MATCH` seguita da una `CREATE` in Neo4j, ricevendo in input i nodi di partenza e fine - con le rispettive label - e il nome della relazione da creare.

Per le relazioni tra **User** e **Movie** e tra **Movie** e **Category** è stato impossibile procedere come sopra, a causa del loro elevato numero (25.000.094 e

963.219, rispettivamente). Ho, invece, utilizzato le **bulk operations** [9] messe a diposizione da py2neo, ovvero trasferimenti contemporanei di grosse quantità di dati, elaborati insieme nel database in modo da massimizzare le performance (fino al 90%).

La funzione `create_bulk_relationships` prende in input una lista di tuple rappresentative della relazione - in formato *(start node id, relationship property, end node id)* e creata dai CSV tramite la funzione `create_bulk_data-`, il nome del tipo di relazione da creare, due tuple che contengono la *label* e il nome dei campi inseriti come identificatori dei nodi nel primo argomento, e infine una lista di nomi delle proprietà della relazione; ha un decoratore che la chiama su batch da 10.000 tuple alla volta.

5.5 Riproducibilità

Dal momento che per inserire nodi e relazioni sono state utilizzate delle `CREATE` (e non delle `MERGE`, che avrebbero richiesto molto più tempo di computazione), se lo script viene rieseguito tutti i dati vengono duplicati nel database. Per questo motivo, all'inizio dello script (dietro conferma dell'utente) viene chiamata la funzione `empty_graph`, che elimina relazioni e nodi.

5.6 Performance

In Tabella 1 un riassunto dei tempi di creazione dello script, sulla mia macchina con CPU AMD Ryzen 7 5700U, 8 GB di RAM e 16 GB di memoria SWAP.

componente	**tempo**	**tempo medio**
User	11m 28s	0.004s
Movie	5m 23s	0.005s
Genre	10s	0.5s
Category	30s	0.02s
RATES	28m	0.00006s
HAS_GENRE	11m 51s	0.006s
HAS_CATEGORY	6m 38s	0.003s

Tabella 1: Tempi richiesti dallo script per inserire nel database i rispettivi componenti.

La principale discriminante sul tempo di creazione totale è chiaramente il numero di nodi e relazioni. Possiamo notare che l'aggiunta di *properties* al nodo e le operazioni di pre-processing tendono a non modificare particolarmente il tempo di creazione (ad esempio, guardando il tempo medio di User e Movie). Diventa anche evidente come sia pivotale l'utilizzo **bulk operations**: il tempo medio di creazione della relazione RATES è di ordini di grandezza inferiore rispetto a quello della relazione HAS_GENRE, per cui non sono state utilizzate.

Un altro dato interessante è la quantità di RAM utilizzata: nello script sono state inserite molte `delete` di variabili e chiamate al `garbage collector`: la

quantità di dati su cui stavo lavorando era notevole, e senza queste cancellazioni graduali la memoria si sarebbe saturata dopo poche operazioni, rendendo lo script estremamente più lento (perché costretto a ricorrere frequentemente alla memoria SWAP). In Figura 3 un grafico rappresentativo del riempimento durante l'esecuzione dello scipt: si può notare che questo cresce velocemente al caricare delle variabili e viene abbattuto grazie alle periodiche `delete`. Le performance sarebbero state sicuramente migliori se lo script fosse stato eseguito su una macchina con 16GB o più di RAM.

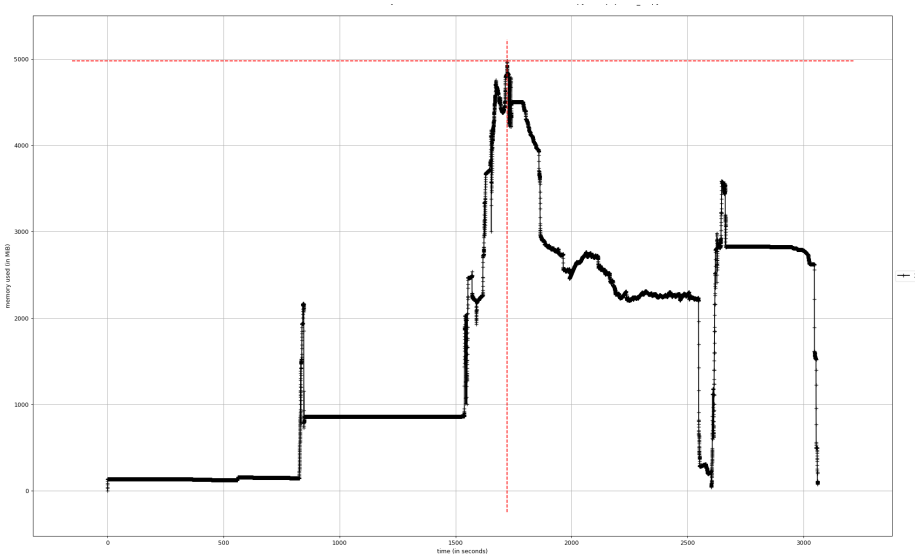

Figura 3: Grafico rappresentativo dello stato della memoria durante l'esecuzione dello script: sull'asse delle x i secondi, sull'asse delle y i MiB.

6 Queries

Nel file `datasetnalaysis.ipynb` si trovano alcune statistiche sul database Neo4j creato precedentemente, con commenti che meglio aiutano a comprendere le differenze tra le performance delle diverse query. Nel file `queries.ipynb` si trova invece il codice delle query che rispondono al **worflow** specificato precedentemente, con i loro risultati e tempi di computazione.

Query 1: Dato un User, trovare i suoi top k Genres

Per questa query ho seguito due approcci. Entrambi trovano i **Movies** a cui l'utente ha dato un **rating** e ne identificano il **Genre**, raggruppando per esso: il primo approccio ordina i **Genre** per **media** dei rating dati a **Movies** di quel Genre, mentre il secondo per **numero** di film per Genre, moltiplicato per il **massimo** dei rating dati a film di quel Genre.

Il primo metodo restituisce una valutazione d'insieme, ma corre il rischio di non tenere conto del numero di film che effettivamente l'utente ha guardato nel genere di riferimento e di come questo influenzi la media. Il secondo metodo tiene invece conto di questo fattore, ma potrebbe dare troppo peso ad outlier nei rating (considerando che ordina per il massimo), oltre che favorire generi che nel dataset sono più popolari (come *Drama* e *Comedy*), dal momento che il massimo viene moltiplicato per numero di Movies guardati. Le statistiche sulle performance sono state calcolate sul primo metodo, in quanto ritenuto - tramite una valutazione empirica dei risultati - più efficace per rispondere al quesito posto.

Query 2: Dato un User, trovare le sue top k Categories

Anche per questa seconda query ho esplorato due metodi, che prendono i **Movies** visti dall'utente e li raggruppano per **Category**. Il primo approccio ordina le categorie trovate per media della **relevance** dei **Movies** per quella categoria, moltiplicata per la media dello **score** assegnato dall'utente per i film di quella categoria; il secondo metodo, invece, ordina sempre per media della **relevance**, ma la moltiplica per il **massimo** e per il **numero** degli **score** dell'utente per quella categoria. Per considerazioni analoghe a quelle fatte per la **query 1**, il primo metodo è stato ritenuto più efficace per ottenere una visione d'insieme e personalizzata per l'utente.

Query 3: Dato un Genre, trovare i suoi top k Movies

Per rispondere a questa query ho considerato tutti i **Movies** con associato il determinato **Genre**, e tutti i rating dati loro dagli utenti; li ho poi ordinati utilizzando i due metodi descritti per le precedenti query (**media** degli **score** e **massimo** moltiplicato per **numero** degli **score** del film). In questo caso ho ritenuto che il **secondo metodo** fosse più utile per rispondere alla query del workflow: non stiamo, infatti, cercando una raccomandazione per un utente

specifico, quanto più un indice di popolarità per film appartenenti a un determinato genere, da consigliare ad esempio per mediare il **cold start problem**. In questi contesti il numero di utenti che fruiscono un contenuto tende a essere più indicativo del rating medio che gli viene assegnato, che ad esempio è facile sia più elevato per un contenuto di nicchia.

Query 4: Data una Category, trovare i suoi top k Movies

La query può essere intesa in due modi diversi: può richiedere i **Movies** che più **rispecchiano** una determinata **Category**, oppure quelli più **popolari** al suo interno. Ho risposto alla prima interpretazione trovando i Movies associati alla categoria e ordinandoli per **relevance**, mentre per la seconda occorre accedere anche a tutti gli utenti che hanno dato un rating ai Movies selezionati: una volta fatto ciò, ho raggruppato per Movie e ordinato per il prodotto tra **media** dei **rating** e **relevance** del Movie. Questo secondo approccio ha il vantaggio di tenere conto della popolarità dei Movie, ma lo svantaggio di dover accedere a molti più nodi, con l'aumento della complessità che ne deriva: le successive considerazioni sulle performance riguarderanno questo metodo.

Raccomandazione

Dopo una breve analisi preliminare, è stato subito chiaro che rispondere alle query successive lavorando direttamente sul grafo avrebbe richiesto un tempo eccessivo. Se si volesse, ad esempio, seguire un approccio di **Collaborative Filtering user-based** - e dunque per prima cosa trovare utenti simili ad un dato Utente sulla base dei Movies che ha visto - occorrerebbe innanzitutto trovare tutti i Movies a cui ha dato un rating (in media 153), e per ognuno di questi Movies risalire ai loro utenti (in media 432) e ai film che a loro volta hanno guardato: già solo lo step iniziale ci richierebbe, dunque, di visitare in media più di 10 milioni di nodi. Ci troveremmo, inoltre, con una lista molto lunga di utenti che non sapremmo bene come discriminare.

Per rispondere alle query successive ho quindi fatto ricorso alla libreria *Neo4j Graph Data Science (GDS)* [10], che contiene l'implementazione efficiente e parallela di diversi algoritmi su grafo, compresi quelli più utilizzati nel contesto della raccomandazione (creazione di **embedding** e **K Nearest Neighbors**).

Il file `gds_recommendation.py` contiene le funzioni che ho utilizzato per lavorare con GDS: brevemente, gli algoritmi non possono essere direttamente applicati al grafo di Neo4j, ma vanno create delle **projection**, ovvero oggetti che rappresentano porzioni di grafo su cui la libreria lavora, che possono essere modificate tramite le funzioni `mutate` e riscritte (totalmente o in parte) nel database originale tramite le funzioni `write`. L'interazione tra Python e la libreria, che è scritta nella forma di query Neo4j, avviene sempre tramite *py2neo*.

Collaborative Filtering

Ho applicato la tecnica di raccomandazione collaborative filtering in entrambe le varianti **user-based** e **item-based**.

Nel caso **user-based** ho inizialmente utilizzato l'algoritmo **Fast Random Projection** [2] per creare un embedding - ovvero un vettore rappresentativo - degli User; l'algoritmo è stato applicato su una porzione del grafo che includeva solo i nodi di tipo Movie e User e le loro relazioni **rates**, ed è in questo senso puramente collaborativo. L'idea è dunque stata quella di rappresentare gli User sulla base della relazione coi Movie, in modo da potere confrontare due utenti tra loro solo tramite la proprietà **embedding**, che ho riscritto sul grafo originale. Ho poi utilizzato una proiezione con i soli nodi User (e la loro nuova proprietà embedding) per applicare l'algoritmo **K Nearest Neighbors** [3], e dunque trovare gli **User simili**: ho infine considerato per ogni utente i **K** utenti più simili (con $k = 5$), con i quali ho creato la relazione **similar** (che ha come attributo **score** il valore di similarità calcolato tramite **KNN**), scrivendola sul grafo di partenza.

Per la tecnica **item-based** ho proceduto allo stesso modo, ma creando gli **embedding** per i Movies e la relazione **users also liked** (per distiguerla dalla relazione che ho creato per la tecnica **content-based**, di cui parlerò in seguito).

Le nuove relazioni aggiunte al grafo semplificano di molto le risposte alle **query 5 e 6**.

Query 5: Dato un User, trovare gli User simili

In questo caso, si tratta semplicemente, dato un utente, di trovare gli utenti con cui ha la relazione **similar**.

Query 6: Dato un User, raccomandare Movies sulla base degli User simili

Nel caso **user-based**, dato un utente ho ricavato i suoi utenti simili, dopodiché i film a cui questi hanno dato un rating (**escludendo** quelli già visti dall'utente): dopo aver raggruppato per film, ho ordinato per **media** dei **rating** moltiplicata per la media degli **score** dell'associazione similar.

Nel caso **item-based**, dato un utente ho ricavato suoi film, poi ho considerato i film simili (secondo la relazione **users also liked**, e sempre escludendo quelli già visti) e li ho ordinati tramite la similarità moltiplicata per il rating dato dall'utente al film originale.

Content-based

In questo caso, ho creato gli **embedding** dei **Movies** tramite una proiezione del grafo che conteneva solamente i nodi di tipo **Movie**, **Genre** e **Category**, con le loro relazioni. Ho poi applicato l'algoritmo **KNN** e scritto una nuova relazione **similar**, questa volta tra **movies**, che ho utilizzato per rispondere alle queries **7** e **8**.

Query 7: Dato un Movie, trovare Movies simili

Come prima, per rispondere a questa query è sufficiente percorrere le relazioni **similar** tra **Movies**.

Query 8: Dato un User, raccomandare Movies sulla base della similarità con quelli che ha guardato

In questo caso ho proceduto in maniera del tutto analoga a quanto fatto per la **query 6** versione **content-based**, ma sfruttando la nuova la relazione **similar**. Un approccio puramente **content-based** avrebbe creato un profilo dell'utente (ovvero una media degli embedding dei film visti) e selezionato i Movies a partire da questo: si tratta, tuttavia, di una strada poco praticabile navigando le relazioni, che mi ha portata a decidere di utilizzare l'approccio appena descritto. Si potrebbe definire un approccio **ibrido**, in quanto la rappresentazione dei Movies è creata sulla base delle loro proprietà (approccio **content-based**), ma la loro selezione avviene con un criterio di **collaborative filtering**.

6.1 Performance

In Tabella 2 un riassunto dei tempi richiesti per la computazione di ogni query del workflow. Possiamo notare che la **query 1** richiede molto più tempo della **query 2**: questo avviene perché la query 1, a partire dai film guardati da un utente ne ricava i generi - che sono in media 1.48 per film -, mentre la query 2 ne ricava le categorie, il cui numero medio è 62; la query 1 richiede, perciò, di navigare meno relazioni rispetto alla query 2, e contestualmente di calcolare la media di meno rating associati.

query	tempo totale	tempo medio
1	17m 48s	0.0065s
2	1h 23m	0.03s
3	6m	19s
4	3h 20s	10s
5	22m 6s	0.008s
6	41m 41s	0.015s
7	5m 24s	0.005s
8	34m 14s	0.012s

Tabella 2: Tempi di computazione per le query del workflow per $k = 5$.

Il tempo medio per rispondere alla **query 3** è decisamente più alto che per le precedenti due; il tempo totale risulta minore per il solo fatto che questa deve essere eseguita per 19 Genres, invece che per ogni User. Il costo di questa query è maggiore perché coinvolge tutti i Movies di un determinato genere (in media 5644, contro la media di 153 Movies per User), e perché per ognuno di questi ricava ogni rating (in media, vi sono 423 utenti per film): si tratta, dunque, di una query computazionalmente molto onerosa, che richiede di visitare mediamente 2

387 412 nodi. Dal momento che ci aspettiamo che questo genere di dato (ovvero il film più popolare dato un genere) tenda a cambiare molto più raramente di quanto viene richiesto, sarebbe utile fare uso del *precomputed pattern* [1], inserendo dei collegamenti di tipo **top movie** tra **Genre** e **Movie**. Lo stesso si può dire riguardo alla **query 4**, che va inoltre eseguita per tutte le 1128 categorie, con un tempo totale notevole: i nodi che devono essere visitati sono in media 371 055 (853 Movies per Category, moltiplicato per 435 Users per Movie); se non tramite il *precomputed pattern* (aggiunta di una relazione **top movies** da Genre a Movie), questo problema potrebbe essere risolto non considerando la popolarità ma solamente la rilevanza, come descritto precedentemente.

Grazie all'utilizzo della libreria GDS, le query **5**, **6**, **7** e **8** richiedono tempi medi piuttosto bassi. Il tempo aggiuntivo per la creazione delle proiezioni dei grafi (da svolgere periodicamente alla luce di nuovi dati) e per l'applicazione degli algoritmi è di circa 4 minuti, ma si deve anche considerare l'aumento di spazio che deriva dall'aggiunta delle proprietà embedding e delle nuove relazioni (circa 1 milione e mezzo).

7 Note per l'esecuzione

- Sono necessarie alcune librerie Python per l'esecuzione dei diversi script: per installarle, è sufficiente eseguire i seguenti comandi per le crazione di un *virtual enviroment*.

```
virtualenv venv
source ./venv/bin/activate
pip install -r requirements.txt
```

- Come già accennato, lo script di creazione del dataset `populate_db.py` assume che esista già un database Neo4j sulla macchina locale e che questo sia attivo e in ascolto a una determinata porta richiesta in input dallo script. Le versioni non enterprise di Neo4j consentono di avere un solo database attivo alla volta: se non si vuole utilizzare il database di default `neo4j`, se ne può creare un altro e attivarlo seguendo questa procedura [7].

- Risulta consigliabile, per un tempo di esecuzione ragionevole, che lo script di creazione `populate_db.py` venga eseguito su una macchina con almeno 8GB di RAM

8 Conclusioni

Questo progetto mi ha permesso di approfondire e ragionare su aspetti molto diversi tra loro, dalla gestione delle **risorse** di tempo e spazio nei database a grafo alla loro applicazione nel contesto dei sistemi di raccomandazione.

Come sviluppi futuri mi piacerebbe realizzare una vera e propria fase di **tuning** dei parametri utilizzati per i diversi algoritmi di GDS (al momento impostati ai parametri default) e una valutazione sistematica dei risultati ottenuti, tramite la suddivisione in grafi di **training** e **test**.

Inoltre, tramite i metodi utilizzati sono riuscita a ottenere un notevole miglioramento delle **performance**, di cui però non sono ancora pienamente soddisfatta: credo che, per esempio, si potrebbe lavorare ancora sulla **parallelizzazione** delle operazioni, oppure trasferire Neo4j su un **server** non locale con risorse migliori.

Riferimenti bibliografici

[1] *Building With Patterns: The Computed Pattern*. Last accessed 10 August 2022. URL: https://www.mongodb.com/blog/post/building-with-patterns-the-computed-pattern.

[2] *Fast Random Projection*. Last accessed 11 August 2022. URL: https://neo4j.com/docs/graph-data-science/current/machine-learning/node-embeddings/fastrp/.

[3] *K-Nearest Neighbors*. Last accessed 11 August 2022. URL: https://neo4j.com/docs/graph-data-science/current/algorithms/knn/.

[4] Paul J. Leach, Rich Salz e Michael H. Mealling. *A Universally Unique IDentifier (UUID) URN Namespace*. RFC 4122. Lug. 2005. DOI: 10.17487/RFC4122. URL: https://www.rfc-editor.org/info/rfc4122.

[5] *MovieLens 25M Dataset*. Last accessed 29 July 2022. 2019. URL: https://grouplens.org/datasets/movielens/25m/.

[6] *Neo4j*. Last accessed 12 August 2022. URL: https://neo4j.com/.

[7] *New database activation*. Last accessed 12 August 2022. URL: https://stackoverflow.com/a/45802452.

[8] *pickle — Python object serialization*. Last accessed 31 July 2022. URL: https://docs.python.org/3/library/pickle.html.

[9] *py2neo.bulk – Bulk data operations*. Last accessed 31 July 2022. URL: https://py2neo.org/2021.0/bulk/index.html.

[10] *The Neo4j Graph Data Science Library Manual v2.1*. Last accessed 11 August 2022. URL: The%20Neo4j%20Graph%20Data%20Science%20Library%20Manual%20v2.1.

[11] *The Py2neo Handbook*. Last accessed 31 July 2022. URL: https://py2neo.org/2021.1/.

[12] *uuid — UUID objects according to RFC 4122*. Last accessed 31 July 2022. URL: https://docs.python.org/3/library/uuid.html.

[13] Jesse Vig, Shilad Sen e John Riedl. "The Tag Genome: Encoding Community Knowledge to Support Novel Interaction". In: *ACM Trans. Interact. Intell. Syst.* 2.3 (set. 2012). ISSN: 2160-6455. DOI: 10.1145/2362394.2362395. URL: https://doi.org/10.1145/2362394.2362395.