UNIVERSITA' DEGLI STUDI DI
NAPOLI FEDERICO II

Facoltà di Ingegneria

Corso di Studi in Ingegneria Informatica

Elaborato d'esame

Big Data Analytics and Business Intelligence

Anno Accademico 2020/2021

professore

Giancarlo Sperlì

studenti

Carmine Cesarano matr. M63/948

Roberto Allocca matr. M63/1045

Alessandro Castaldo matr. M63/946

Sommario

1 Introduzione del problema 3
2 Caratterizzazione del dataset 5
3 Architettura proposta 5
3.1 Data management layer: MongoDB 6
3.1.1 Import dei dati e trasformazioni iniziali 6
3.1.2 Creazione degli indici 7
3.2 Data processing layer: Apache Spark 8
3.2.1 Data Modeling 8
3.2.2 Collaborative filtering e ALS 10
3.2.3 Architettura complessiva del recommendation system 14
3.3 Presentation layer: PowerBI 20
3.3.1 Import e modellazione dei dati 20
3.3.2 Implementazione delle dashboard 21
4 Future improvements 26

1 Introduzione del problema

L'obiettivo fondamentale di un **sistema di raccomandazione** è di migliorare la user experience fornendo delle raccomandazioni personalizzate per gli utenti e calcolate sulla base dei loro precedenti feedback impliciti o espliciti. Questi sistemi, infatti, passivamente o attivamente, tracciano i diversi tipi di comportamenti degli utenti, con lo scopo di modellare le preferenze dell'utente stesso. Questa modellazione sarà sicuramente utile per supportare l'utente, con dei consigli personalizzati, nella scelta tra le migliaia di opzioni proposte.

Formalmente:

- U: *set di tutti gli utenti*
- I: *set di tutti i possibili prodotti da consigliare*

Definendo **p** una funzione che misura l'utilità dell'item i per l'utente u, la raccomandazione è calcolata come:

$$p: UxI \rightarrow R$$

Dove R è il set ordinato dei prodotti consigliati.

Dal punto di vista applicativo, il sistema deve trovare l'espressione della funzione di utilità per consigliare prodotti con i quali l'utente non ha ancora interagito. Tali raccomandazioni possono essere generate adoperando differenti approcci:

- **Popularity-based**: il sistema consiglia i prodotti più popolari tra tutti gli altri utenti.

- **Content-based**: questo approccio permette di creare un profilo per ogni utente, caratterizzato dalle features che lui ha preferito in passato. Le raccomandazioni dunque sono generate sulla base della somiglianza tra il profilo creato e le features proprie dell'item.

- **Collaborative-filtering:** questo approccio non crea un profilo esplicito ma è basato sulle interazioni che gli utenti hanno avuto con gli items. Le raccomandazioni vengono calcolate cercando di trovare similarità tra i modelli di preferenza e interazione dei diversi utenti. Dunque all'utente attivo verranno consigliati item con i quali utenti simili hanno interagito allo stesso modo.

Collaborative filtering: vantaggi e svantaggi

Uno dei vantaggi del collaborative filtering è il fatto che rappresenta un approccio 'self-generating'. Gli utenti creano in modo naturale i dati sui quali l'algoritmo lavorerà per trovare le raccomandazioni, semplicemente interagendo con gli items nel sistema. Le uniche informazioni necessarie sono le interazioni users-items e questo può essere un vantaggio specialmente nei casi in cui risulti difficile ottenere delle features di alta qualità dagli items. Un altro vantaggio di questo approccio è che aiuta gli utenti a scoprire nuovi items che sono fuori dal sottospazio definito dalle loro preferenze storiche.

Tuttavia ci sono alcuni inconvenienti, come ad esempio il problema del cold start, che si configura quando il sistema parte e deve inferire le preferenze quando non sono ancora a disposizione per l'utente specifico sufficienti interazioni. Inoltre, è difficile per il CF raccomandare item nuovi, introdotti nel sistema, con i quali non c'è stata ancora alcuna interazione.

Content-based: vantaggi e svantaggi

I sistemi content-based invece si affidano alle features dell'item, e questo rende l'approccio molto più robusto al popularity bias e al cold start problem; infatti permettono facilmente di consigliare anche nuovi item basandosi sulle features precedentemente preferite dall'utente.

Lo svantaggio fondamentale nell'uso di questo approccio, invece, è che, non considerando affatto le interazioni utenti-film, verranno consigliati soltanto items che appartengono al sottospazio delle preferenze dell'utente attivo, e inoltre, potrebbero essere consigliati anche item che hanno ottenuto un rating basso da altri utenti.

Creando un ibrido tra i due approcci è possibile progettare un sistema che consigli items tra utenti che hanno un profilo di valutazione simile, pur continuando a fornire consigli basati sulle caratteristiche specifiche preferite dall'utente.

2 Caratterizzazione del dataset

In particolare, il problema delle raccomandazioni, in questo lavoro, è contestualizzato nell'ambito della filmografia e il sistema da progettare si propone di generare delle raccomandazioni di film affini ad ogni utente.

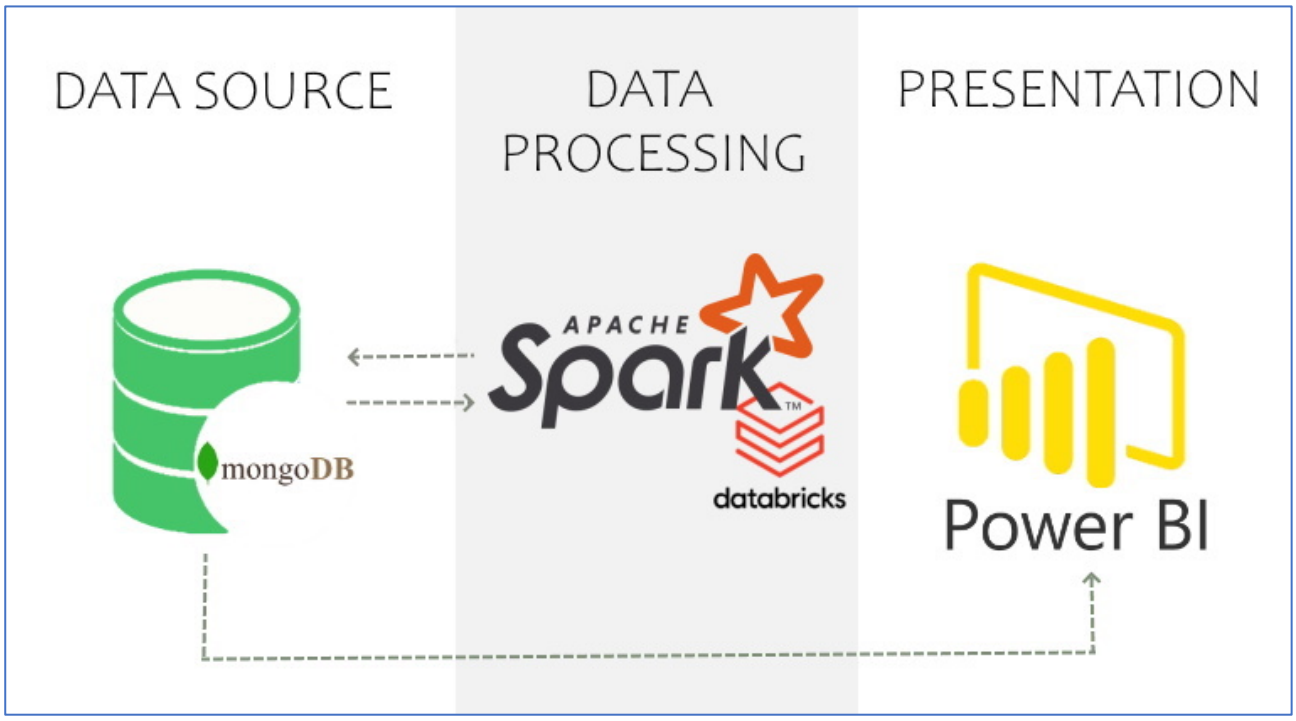
Il dataset utilizzato è opensource ed è fornito dal sito **'grouplens.org'** nella versione 20M. E' costituito, in particolare, dai seguenti file in formato csv:

- *ratings.csv*: 27M di ratings
- *tags.csv*: 1,1M di tags
- *movies.csv* : 58K film
- *genome_scores*.csv: tag relevance scores per film
- *genome_tags*.csv: tag descriptions per i tag ID

3 Architettura proposta

Il sistema complessivo progettato ed implementato per la risoluzione del problema proposto, consta della seguente architettura:

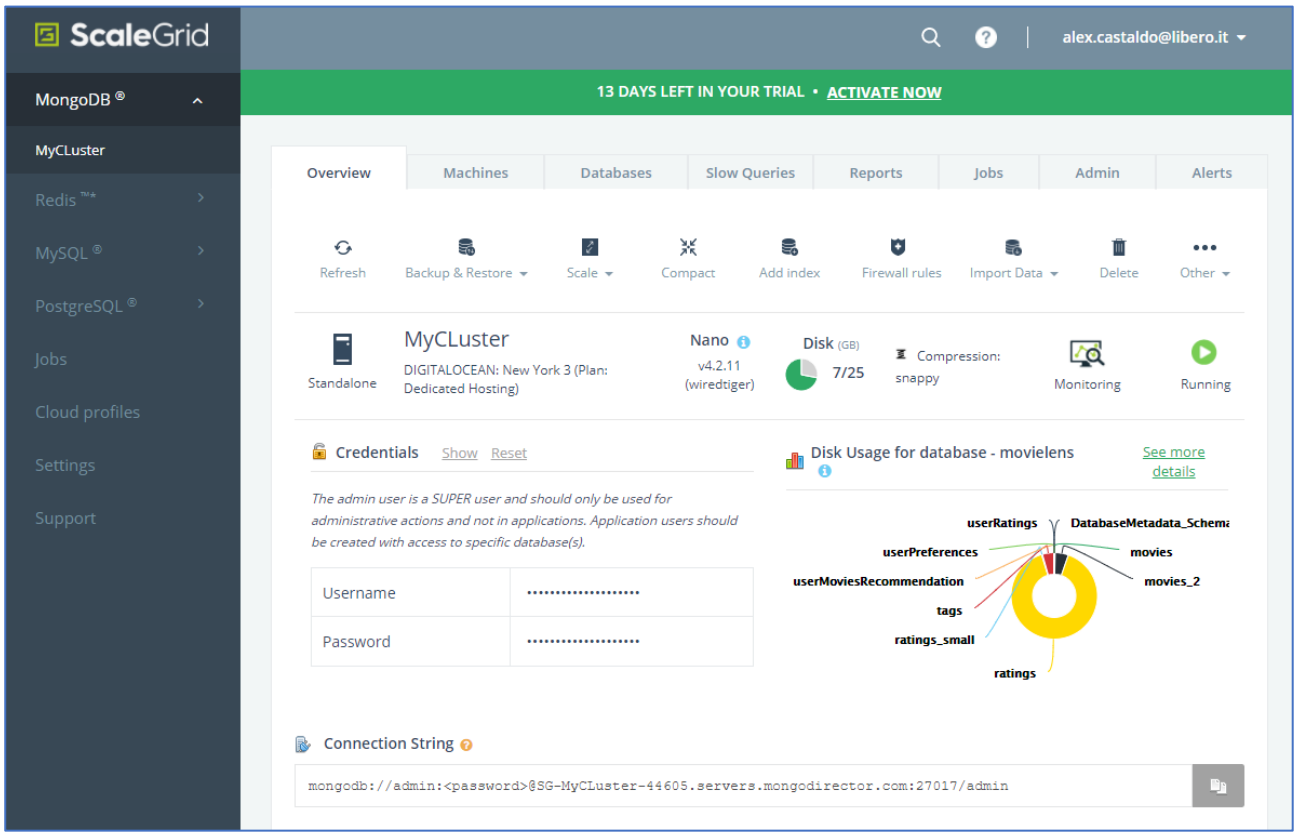
- Data management layer
- Data processing layer
- Presentation layer



3.1 Data management layer: MongoDB

Come si evince dalla definizione del problema, questo tipo di sistema deve essere fortemente basato sull'interazione con le informazioni pregresse sugli utenti, le quali consistono in moli enormi di dati.

Data la necessità di dover processare, gestire e memorizzare milioni di entries, i sistemi di storage di tipo relazionale non risultano essere la soluzione ideale per questo problema. A supporto del processo di data storage e managment infatti, nell'ambito di questo lavoro, verranno sfruttati sistemi di tipo NoSQL, e in particolare sarà usata un'istanza di database document-based MongoDB, adoperando il servizio '*scalegrid.io*' un database-as-a-service.



3.1.1 Import dei dati e trasformazioni iniziali

Prima della fase di import dei dati nel database è necessario effettuare delle operazioni di data preparation e data transformation. Sia per la fase di import dei dati che per il pre-processing, viene utilizzata la libreria '*PySpark*', in esecuzione sulla piattaforma databricks, in modo da effettuare queste operazioni più efficientemente sfruttando il clustering computing.

Una delle trasformazioni che può essere applicata a questo livello, può essere la seguente, che opera sul dataset '*movies.csv*':

```
csvMovies = "dbfs:/FileStore/tables/movielens/movies.csv"

datiMoviesRDD = spark.read.option("inferSchema", "True")
                          .option("header", "True").csv(csvMovies).rdd
                         .map(lambda x: (str(x[0]),x[1],x[2].split("|")))
```

Il file *csv* viene caricato nella struttura dati RDD (*Resilient Distributed Dataset*), alla quale possono essere applicate tutte le funzioni della libreria Spark che sfruttano il processing distribuito. Tramite una '**map**' quindi, si effettua lo split sul carattere '|' che permette di spacchettare la stringa dei generi associati ad un film, in un vettore di generi.

$$1, Toy\ Story\ (1995), Adventure|Animation|Children|Comedy|Fantasy$$

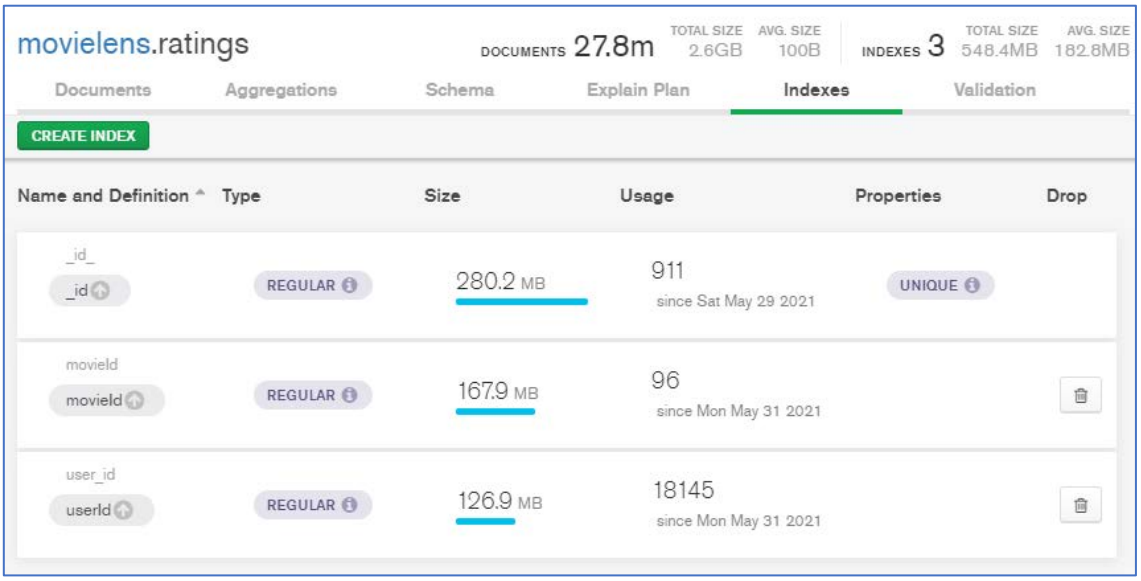
Successivamente, per operare l'import in MongoDB è necessario aprire una connessione con il server, trasformare l'RDD in una **lista di dizionari** (che ha formattazione equivalente ad un file JSON) e utilizzare la funzione **insert_many**, che carica i dati manipolati nell'RDD all'interno di una collection di documents in MongoDB.

```
client = pymongo.MongoClient('mongodb://admin:password@myserverIP', 27017)
db = client['movielens']

movieKeys = ["movieId", "title", "genres"]
moviesList = []

for lista in datiMoviesRDD.collect():
  yourdic = dict(zip(movieKeys, lista))
  moviesList.append(yourdic)

movies_collection.insert_many(moviesList)
```

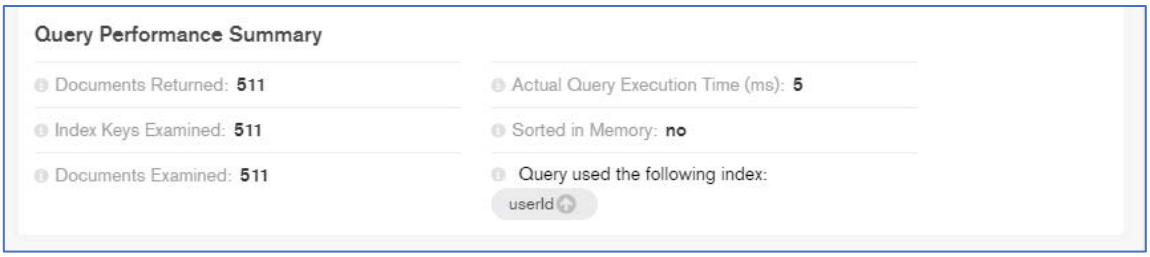
3.1.2 Creazione degli indici

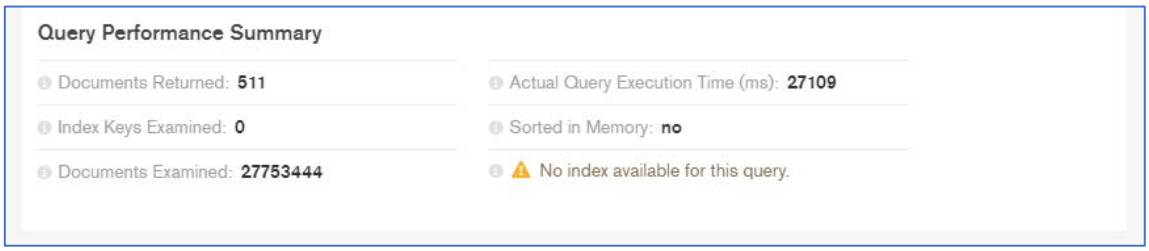
Nell'ambito del processo di data management, inoltre, è opportuno, dopo aver caricato i dati, progettare degli appositi indici per ciascuna delle collection. Questa operazione favorirà l'efficienza nell'accesso ai documents, per le successive fasi.

A titolo d'esempio vengono mostrati gli indici che sono stati creati per la collection '**ratings**':



Di seguito invece, tramite il tool interno al client **MongoDB Compass**, si mostrano le prestazioni dell'esecuzione di una query, in termini di tempo di esecuzione, utilizzando o meno gli indici:



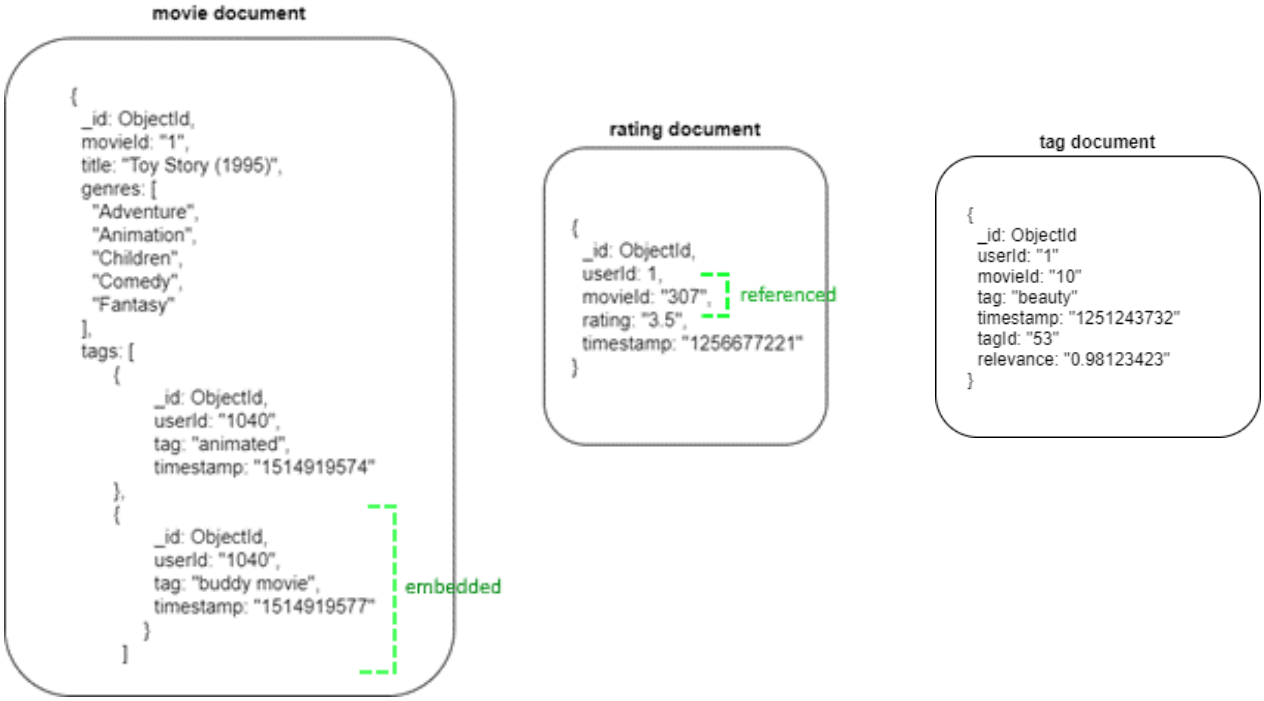


3.2 Data processing layer: Apache Spark

Per gli stessi motivi menzionati in precedenza, la fase di processing dei dati non può essere operata con codice di tipo sequenziale eseguito su un'unica macchina, ma deve essere sfruttata la potenza di calcolo di un cluster di nodi che eseguono le operazioni sui dati in modo distribuito. A supporto di questa fase viene utilizzato il framework **Apache Spark** in esecuzione sull'ambiente cloud **Databricks** che fornisce supporto per il clustering computing.

3.2.1 Data Modeling

La prima operazione di processing fondamentale è quella che permette la creazione di un modello dei dati che sarà utile nel layer applicativo per identificare più facilmente le relazioni tra le entità del dominio. In particolare viene utilizzato un modello ibrido che sfrutta sia relazioni **referenced** che **embedded**, a seconda delle necessità e delle caratteristiche delle particolari collection.



Di seguito è riportato il codice Spark che di effettuare le operazioni di modellazione sulle collection e permette di ottenere la struttura dei documents come riportata precedentemente. In particolare, tramite il framework Aggregation i document relativi ai film, vengono incorporati con i relativi documenti inerenti i tags che sono stati dati dagli utenti a quel film. Inoltre, ai documents relativi ai tags viene aggiunto l'attributo relevance preso dalla collection genome scores.

```
client = pymongo.MongoClient('mongodb://admin:nUOSRJ3kasFEspIb@sg-mycluster-
44605.servers.mongodirector.com', 27017)
db = client['movielens']

movies_join_collection = db.movies_2

db.movies.aggregate([
        {
           '$lookup':
           {
             'from': 'tags',
             'localField': 'movieId',
             'foreignField': 'movieId',
             'as': 'tags'
           }
        },{
          '$project': {
           'tags.movieId': 0
          }
        }, {
          '$out': "movies_2"
        }
    ]);

db.tags.aggregate([
      {"$addFields": {"tag": { "$toLower": "$tag" }}},
      {"$lookup": {
         'from': 'genome_tags',
         'localField': 'tag',
         'foreignField': 'tag',
         'as': 'tag_doc'
      }},
      {"$addFields": {"tagId": {"$arrayElemAt": [ "$tag_doc.tagId", 0 ] }}},
      {"$project": {"tag_doc": 0}},
      {"$lookup": {
         'from': 'genome_scores',
         'let' : {'movieId': '$movieId', 'tagId':'$tagId'},
         'pipeline': [
                     { "$match" :
                      { "$expr" :
                       { "$and" : [
                          { "$eq": ["$movieId", "$$movieId"] },
                          {"$eq": ["$tagId", "$$tagId"]}
                        ]
                       }
                      }
                     }
                ],
          'as': 'relevance'
         }
      },
      {"$addFields": {"relevance": {"$arrayElemAt": [ "$relevance.relevance", 0 ] }}},
      {"$out": "tags_relevance"}
])
```

3.2.2 Collaborative filtering e ALS

Nella progettazione del layer di processing deve essere inclusa anche tutta la pipeline che si occuperà del calcolo delle raccomandazioni per gli utenti. Il core del sistema d'esame è rappresentato da un algoritmo basato sull'approccio collaborative filtering, messo insieme ad ulteriori filtri in un'architettura più complessa che verrà successivamente dettagliata.

Utilizzando un sistema basato sul **collaborative filtering**, l'assunzione fondamentale da fare è che ogni singolo utente che ha mostrato un certo insieme di preferenze continuerà a mostrarle in futuro.

Questi sistemi possono operare secondo due strategie:

- User-based
- Item-based

In particolare l'approccio utilizzato è quello **user-based**, il quale opera in due passaggi fondamentali:

1. Cerca gli utenti che condividono gli stessi modelli di valutazione con l'utente attivo (quello a cui è destinata la predizione).
2. Usa le valutazioni di quegli utenti che condividono lo stesso modello per calcolare una predizione.

Nel problema in esame, il dataset ('*ratings collection*') può essere rappresentato come una matrice d'interazione R tra utenti e film, in cui in ogni cella è riportato un rating attribuito da un utente ad un film. Analiticamente quindi, il problema delle raccomandazioni, prevede, nella prima fase, il calcolo delle predizioni dei valori mancanti per la matrice sparsa R.

	Movie 1	Movie 2	Movie ...	Movie N
User 1	1	BLANK	BLANK	3
User 2	BLANK	5	BLANK	3
User 3	BLANK	BLANK	1	BLANK
User 4	2	3	BLANK	BLANK
User 5	BLANK	BLANK	1	BLANK
User 6	4	BLANK	5	BLANK
User 7	BLANK	4	BLANK	BLANK
User ...	BLANK	3	BLANK	BLANK
User m	BLANK	BLANK	BLANK	4

ALS

	Movie 1	Movie 2	Movie ...	Movie N
User 1	1	4	2	3
User 2	1	5	3	3
User 3	2.5	2.8	1	3.5
User 4	2	3	2	3.5
User 5	2.5	2.8	1	3.1
User 6	4	1.2	5	1.4
User 7	1	4	2.5	3
User ...	2	3	2	3
User m	1	4	2	4

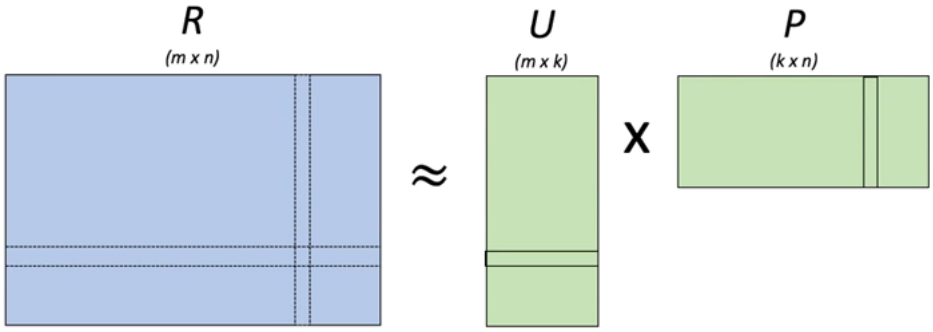
Tuttavia, con il crescere del numero di utenti e di items, le interazioni potrebbero crescere esponenzialmente e il problema potrebbe diventare poco scalabile. Per risolvere questo tipo di problemi, viene utilizzato un algoritmo di **Matrix Factorization** detto **Alternating Least Square** che rappresenta lo stato dell'arte per gli **sparse data problems**.

L'algoritmo ALS per la matrix factorization assume che esista una serie di attributi, comuni sia agli items (movies) che agli utenti, che avranno un certo peso in relazione al preciso film o all'utente a cui sono legati. Quindi è ragionevole presumere che ogni film sia legato a uno a più attributi con una certa importanza, e ad ogni utente possano piacere uno o più attributi con una certa importanza. Ciò significa che ciascuno di questi attributi influenza la precisa interazione contenuta nella matrice R, e quindi la valutazione user-movie dovrebbe dipendere dalla somiglianza tra le caratteristiche dell'utente e le caratteristiche del film.

Questi attributi sono definiti fattori latenti (o nascosti), perchè non forniti e, nel caso di un film, possono rappresentare, ad esempio, genere, cast di attori, regista, anno di produzione, etc.

Analiticamente, l'algoritmo fattorizza la matrice R di interazioni user-movie nel prodotto di due matrici, U e P, aventi rispettivamente un numero di colonne e un numero di righe, pari a k, dove k è il numero di fattori nascosti scelti per risolvere il problema. La fattorizzazione sarà tale che:

$$R \cong U\ x\ P^T$$



Questo vuol dire che inizialmente l'algoritmo riempie le matrici U e P con valori randomici e, iterativamente, calcolando l'errore commesso ed utilizzando un approccio simile al gradient descent, modifica i valori delle matrici in modo che il loro prodotto approssimi sempre meglio la matrice R, minimizzando iterativamente la seguente funzione di costo:

$$J = \|R - UxP^T\|_2 + \lambda(\|U\|_2 + \|P\|_2)$$

Il primo termine misura la distanza dell'errore quadratico medio (MSE) tra la matrice di valutazione originale R e la sua approssimazione $U\ x\ P^T$, mentre il secondo è un termine di regolarizzazione per evitare l'overfitting dovuto ad eventuale rumore locale sui dati.

Chiaramente, per il valutare l'errore durante l'addestramento, i ratings predetti dall'algoritmo devono essere confrontati con quelli a disposizione in un dataset di ground truth.

Inoltre, come per tutti gli algoritmi di machine learning, anche per ALS devono essere ottimizzati degli iperparametri, sfruttando tecniche di hold-out o cross validation:

- maxIter: il numero massimo d'iterazioni da eseguire;
- rank: il numero di fattori latenti del modello;
- regParam: il parametro di regolarizzazione λ (L2 regularization).

Selezione degli iperparametri

Tramite il framework Spark, e in particolare adoperando la libreria **MLlib**, è stato possibile addestrare un'istanza dell'algoritmo ALS sui dati dei ratings. Per operare la fase di selezione degli iperparametri e valutazione dell'errore di generalizzazione, il dataset dei ratings viene splittato in tre porzioni (60-20-20):

```
# Splitting del dataset
trainingRDD_GT, validationRDD_GT, testRDD_GT = ratingsRDD.randomSplit([6, 2, 2])

# Creazione degli RDD da usare per le predizioni (viene eliminato il rating reale)
validationRDD = validationRDD_GT.map(lambda x: (x[0], x[1]))
testRDD = testRDD_GT.map(lambda x: (x[0], x[1]))
```

Successivamente vengono eseguiti iterativamente differenti addestramenti del modello, al variare del parametro *maxIter* e *rank*, tenendo fisso il parametro *regParam = 0.1*:

```
# Grid search degli iperparametri
iterazioni = [3, 5, 10]
ranks = [4, 8 ,16]
regul = 0.1
errMin = float('inf')
bestIter = -1
bestRank = -1

# Training della rete ALS
for iter in iterazioni:
 for rank in ranks:
  model = ALS.train(trainingRDD_GT, rank, iterations=iter, lambda_=regul)
  preds = model.predictAll(validationRDD).map(lambda r: ((r[0], r[1]), r[2]))
  ratesPreds = validationRDD_GT.map(lambda r: ((int(r[0]), int(r[1])), float(r[2]))).join(preds)
  error = math.sqrt(ratesPreds.map(lambda r: (r[1][0] - r[1][1])**2).mean())
  print('Rank %s, Iterazione %s -> RMSE = %s' % (rank, iteration, error))

  if error < errMin:
     errMin = error
     bestIter = rank
     bestRank = iter

print('il modello migliore è stato addestrato con: rank %s e iterazione %s' % (bestRank,
bestIter))
```

Valutazione delle prestazioni del modello

Dopo aver addestrato il modello con un training set (60% del dataset) viene valutata la bontà degli iperparametri scelti nella specifica iterazione, calcolando l'errore di classificazione commesso su un validation set (20% del dataset). La funzione **predictAll(validationRDD)** calcola i ratings per ciascun incrocio tra tutti gli utenti e i film contenuti all'interno del *validation set*.

Il validation set, dal punto di vista analitico, risulta essere una matrice sparsa di ratings utente-film e quindi la ground truth, con la quale valutare l'errore di classificazione delle predizioni, non contiene tutte le possibili interazioni.

Per questo motivo, a supporto del calcolo dell'errore di classificazione, deve essere effettuato un **inner join** tra l'RDD relativo alle predizioni e l'RDD relativo al validation set, in maniera tale da mantenere in un RDD soltanto le predizioni per quelle interazioni presenti nella matrice sparsa iniziale.

In seguito alla scelta degli iperparametri migliori, è possibile riaddestrare un modello mettendo insieme training e validation (60% + 20%), per poi calcolare l'errore di generalizzazione utilizzando un test set completamente disgiunto (20%):

```
# Calcolo dell'errore di generalizzazione sul test set
training_validationRDD = sc.union([trainingRDD_GT, validationRDD_GT])
model = ALS.train(training_validationRDD, bestRank, bestIter, lambda_= regul)
predictions = model.predictAll(testRDD).map(lambda r: ((r[0], r[1]), r[2]))
ratesPreds = testRDD_GT.map(lambda r: ((int(r[0]), int(r[1])), float(r[2]))).join(predictions)
error = math.sqrt(ratesPreds.map(lambda r: (r[1][0] - r[1][1])**2).mean())
```

L'errore di classificazione in fase di selezione degli iperparametri, così come l'errore di generalizzazione, è stato misurato in termini di RMSE, metrica che valuta la radice delle differenze dei quadrati tra i ratings predetti e quelli reali. Come conseguenza, errori grandi potrebbero influire drammaticamente sul RMSE e questo rende questa metrica ideale quando non sono desiderabili errori grandi.

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(d_i - \widehat{d}_i\right)^2}$$

- d_i *rating reale*
- $\widehat{d}_i$ *rating predetto*
- n *numero di ratings*

Di seguito si mostrano i risultati ottenuti durante l'addestramento:

```
Rank 4, Iteration 3 -> RMSE = 0.9262262803333907
Rank 8, Iteration 3 -> RMSE = 0.9431901175959143
Rank 16, Iteration 3 -> RMSE = 0.9293336358593488
Rank 4, Iteration 5 -> RMSE = 0.9116391177279807
Rank 8, Iteration 5 -> RMSE = 0.9275789725595872
Rank 16, Iteration 5 -> RMSE = 0.9192877556188399
Rank 4, Iteration 10 -> RMSE = 0.903592390085454
Rank 8, Iteration 10 -> RMSE = 0.9166057975556416
Rank 16, Iteration 10 -> RMSE = 0.9162789483651548
Il modello migliore è stato addestrato con: rank 4 e iteration 10
Errore di generalizzazione sui dati di test utilizzando il modello migliore: RMSE = 0.8849531190582499
```

Per esigenze computazionali, nella fase di selezione degli iperparametri, viene utilizzato il dataset '*ratings_small.csv*' che consta di 100K ratings, al posto di '*ratings.csv',* che ne contiene 27M.

Possiamo essere abbastanza confidenti del fatto che i migliori iperparametri trovati saranno gli stessi di quelli eventualmente ricercati utilizzando il dataset completo, essendo il contributo di bias dell'errore lo stesso per entrambe le versioni del dataset.

Training del modello completo

Successivamente quindi, utilizzando gli iperparametri migliori trovati, viene addestrato un modello finale, utilizzando il dataset completo (27M di ratings):

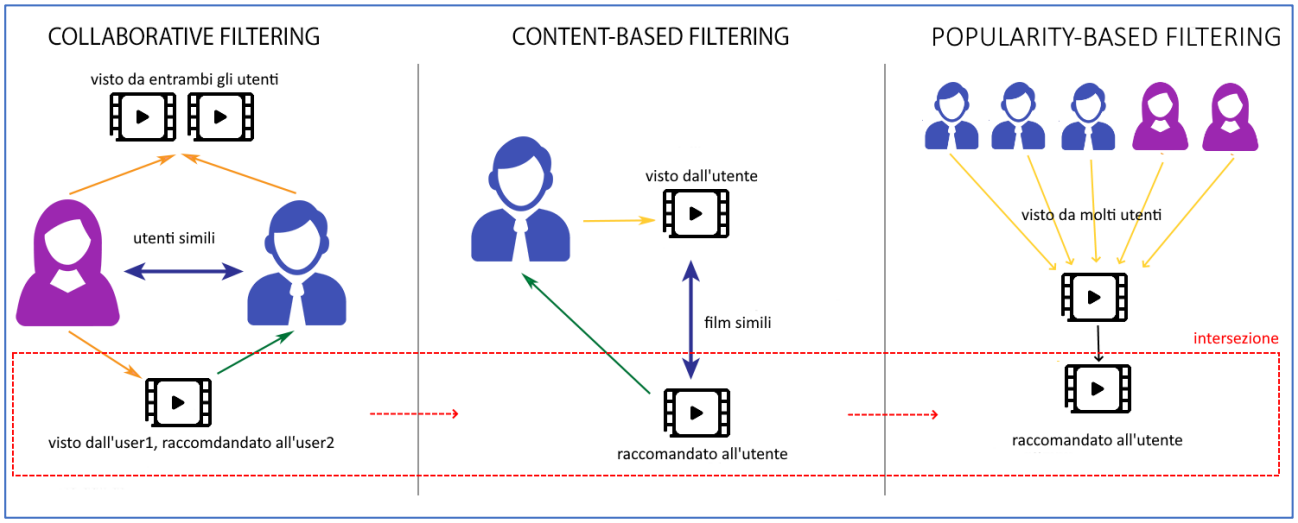
```
# Splitting del dataset
completeTrainingRDD_GT, completeTestRDD_GT = ratings_completeRDD.randomSplit([7.5, 2.5])
completeTestRDD = completeTestRDD_GT.map(lambda x: (x[0], x[1]))

# Training del modello finale
model = ALS.train(completeTrainingRDD_GT, bestRank, bestIter, lambda_=regul)

# Calcolo dell'errore di generalizzazione sul test set
preds = model.predictAll(completeTestRDD).map(lambda r: ((r[0], r[1]), r[2]))
ratesPreds = completeTestRDD_GT.map(lambda r: ((int(r[0]), int(r[1])), float(r[2]))).join(preds)
complete_error = math.sqrt(ratesPreds.map(lambda r: (r[1][0] - r[1][1])**2).mean())

print ('Errore sui dati di test:  RMSE = %s' % (complete_error))

# Salvataggio in locale del modello addestrato sul cluster databricks
model_path = os.path.join('/', 'models', 'movie_lens')
complete_model.save(sc, model_path)
```

Si ottengono i seguenti risultati in termini di RMSE:

```
Numero ratings nel dataset completo: 27753444
Errore sui dati di test:  RMSE = 0.831005003207503
```

3.2.3 Architettura complessiva del recommendation system

Con l'obiettivo di fornire delle raccomandazioni più affini alle preferenze degli utenti, si è scelto di combinare all'approccio collaborative filtering user-based esposto in precedenza, quello di tipo content-based, oltre che posporre un filtraggio di tipo popularity-based.



Il sistema di tipo collaborative, da solo, permette di ottenere, per ciascun utente, la lista di tutti i film ancora non visti e consigliati, mostrati in ordine decrescente sulla base del rating predetto.

L'approccio ibrido realizzato, invece, permette di ottenere raccomandazioni qualitativamente migliori, incrociando i risultati del collaborative filtering con caratteristiche content-based dello specifico item consigliato per l'utente attivo. Ciò è utile per filtrare tutti i film consigliati sulla base di features per le quali l'utente attivo ha già espresso gradimento in precedenza (ad esempio: generi preferiti).

Può essere utile, inoltre, un filtraggio basato sulla popularity dei film consigliati e questo permette di eliminare dai consigliati quei film che, seppur individuati dall'algoritmo come compatibili con l'utente, sono caratterizzati da una scarsa affidabilità nella predizione dato l'esiguo numero di visioni registrate. In particolare, vengono filtrati tutti i film visti e votati meno di 30 volte.

In sintesi, l'approccio adottato prevede complessivamente i seguenti filtri:

- Collaborative Filtering: fornisce la lista di *tutti i film non ancora visti*, in ordine di compatibilità con l'utente.
- Content-based Filtering: filtra i risultati precedenti sulla base dei *top 5 generi preferiti* dall'utente.
- Popularity-based Filtering: filtra dai risultati precedenti i *film visti meno di 30 volte*.

Di seguito viene mostrato come l'approccio è stato implementato a livello applicativo in ambiente Spark. Per esigenze computazionali, le raccomandazioni sono generate soltanto relativamente ai primi 100 utenti.

I **top 5 generi preferiti** dall'utente vengono determinati definendo, in primo luogo, la lista di generi relativi a tutti e soli i film che l'utente ha visto, e successivamente attribuendo ad ognuno dei generi un rating calcolato come la media dei ratings attribuiti dall'utente a tutti i film visti di quel genere specifico. Di questa lista verranno considerati solo i primi 5 elementi con migliore rating.

La seguente funzione permette di valutare quanto descritto per il singolo utente:

```
# Sostituisce all'id del film, il rating assegnato dall'utente a quel film
def substitute_movieID_withRating(genre, ratings_by_user):
    substituted = []
    for movie in genre['rating']:
        for rating in ratings_by_user:
            if rating['movieId'] == movie:
                substituted.append(float(rating['rating']))
    result_per_genre['rating'] = substituted
    return result_per_genre

# Trova il rating medio dei film votati dall'utente per ogni genere
def average_rating_for_genreViewByUser(db, ID):
    # tutti i ratings dell'utente
    ratings_by_user = list(db.ratings.find({'userId': ID}))

    # tutti gli id dei film votati dall'utente
    movie_ids_rated_by_user = list(db.ratings.distinct('movieId', {'userId': ID}))

    # crea la lista di generi visti dall'utente, ognuno con il relativo array di movieId
    pipeline = [
        {"$match": {"movieId": {"$in": movie_ids_rated_by_user}}},
        {"$unwind": "$genres"},
        {"$group": {"_id": "$genres", "rating": {"$addToSet": "$movieId"}}},
        {"$addFields": {"genre": "$_id"}},
        {"$project": {"_id": 0}}
        ]
    genre_rated_by_user = list(db.movies.aggregate(pipeline, cursor={}))
    {'rating': ['1591'], 'genre': 'Adventure'},
    {'rating': ['2134'], 'genre': 'Fantasy'},
    {'rating': ['1257', '3893'], 'genre': 'Romance'},

    preferences = {}
    preferences['userId'] = str(ID)
    preferences['genre'] = []
    for genre in genre_rated_by_user:
        # Sostituisce all'id del film, il rating assegnato dall'utente a quel film
        genre = substitute_movieID_withRating(genre, ratings_by_user)
        {'rating': [1.5], 'genre': 'Adventure'}
        {'rating': [4.5], 'genre': 'Fantasy'}
        {'rating': [4.5, 3.5], 'genre': 'Romance'}

        # calcola la media di tutti i ratings relativi ad un genere
        genre['rating'] = round((sum(genre['rating'])/len(genre['rating'])),2)
        {'rating': 1.5, 'genre': 'Adventure'}
        {'rating': 4.5, 'genre': 'Fantasy'}
        {'rating': 4.0, 'genre': 'Romance'}

        preferences['genre'].append(genre)
    return preferences
```

Chiamando la funzione esposta in precedenza in un loop è possibile generare una collection su MongoDB, contenente un documento, per ognuno degli utenti, che riporta il rating medio per ogni genere:

```
preferencesList = []
all_users = list(ratings_collection.distinct('userId'))
all_users = sorted([int(i) for i in all_users])
for user in all_users[:100]:
    preferencesList.append(average_rating_for_genreViewByUser(db,str(user)))

user_preferences_collection = db.userPreferences
user_preferences_collection.insert_many(preferencesList)
```

Con la seguente funzione invece è possibile recuperare i Top 5 generi per ogni utente:

```
def top5_user_genre_preferences(userId,db):
  preferences_user = db.userPreferences.find({'userId': userId})
  preferences_user = list(preferences_user)[0]['genre']
  preferences_user = sorted(preferences_user, key = lambda i: i['rating'], reverse=True)
  top_preferences_user = preferences_user[:5]
  top_preferences_user = [d['genre'] for d in top_preferences_user]
  return top_preferences_user
```

Per quanto riguarda il **filtro basato sulla popolarità**, è possibile implementarlo semplicemente adoperando il framework *Aggregation* di MongoDB e contando il numero di ratings ottenuti da ognuno dei film nella collection relativa:

```
# Conteggio dei ratings per ogni film
def computeMoviesRatingCount(db):
    rating_count = db.ratings.aggregate([
        {
            '$match': {
                'movieId': { '$not': {'$size': 0} }
            }
        },
        { '$unwind': "$movieId" },
        {
            '$group': {
                '_id': {'$toLower': '$movieId'},
                'count': { '$sum': 1 }
            }
        }
    ]);
    jsonString = json.dumps(list(rating_count))
    dbutils.fs.rm("/data/data.json")
    dbutils.fs.put("/data/data.json",jsonString)
    moviesRatingsCountsRDD = spark.read.json('/data/data.json',multiLine=True).cache().rdd
                                    .map(lambda x: (x[0],x[1]))
    return moviesRatingsCountsRDD

moviesRatingsCountsRDD = computeMoviesRatingCount(db)
```

Succesivamente è possibile caricare il modello di ALS pre-addestrato nella fase precedente e calcolare le predizioni per ognuno degli utenti.

```
# Carica il modello ALS addestrato precedentemente e salvato in locale
def load_model_ALS():
  model_path = os.path.join('/', 'models', 'movie_lens')
  return MatrixFactorizationModel.load(sc, model_path)

# Predizioni del rating per ogni film ancora non visto dall'utente
def predict_ratings_for_unrated_movies(db, userId, model):

    # tutti gli id dei film votati dall'utente
    userRatingsMovieIds = list(db.ratings.distinct('movieId', {'userId': userId}))

    # film ancora non votati dall'utente
    userUnratedRDD = moviesRDD.filter(lambda x: x[0] not in userRatingsMovieIds)
                              .map(lambda x: (userId, x[0]))
    # predict ratings for unrated movies
    return model.predictAll(userUnratedRDD)
```

In un loop sui primi 100 utenti, vengono chiamate tutte le funzioni descritte in precedenza per:

- utilizzare il modello ALS per le predizioni
- filtrare i film sulla base dei generi preferiti del preciso utente
- scartare, i film aventi meno di 30 recensioni.

Infine, i risultati delle raccomandazioni vengono salvati in una collection che sarà utilizzata durante la successiva fase di analytics

```
# Load del modello ALS già addestrato
model = load_model_ALS()

# Conteggio dei ratings per ogni film
moviesRatingsCountsRDD = computeMoviesRatingCount(db)

# Calcolo delle recommendation per i primi 100 utenti
for user in all_users[:100]:

    userId = str(user)

    # Predizioni del rating per ogni film ancora non visto dall'utente
    myuserRecommendationsPredictionsRDD = predict_ratings_for_unrated_movies(db, userId, model)

    # Join tra rating predetto, conteggio rating e generi del film
    myuserRecommendationsRDD = myuserRecommendationsPredictionsRDD
                                .map(lambda x : (str(x.product), x.rating))\
                                .join(moviesRDD.map(lambda x: (str(x[0]),[x[1],x[2]])))\
                                .join(moviesRatingsCountsRDD.map(lambda x: (str(x[0]),x[1])))\
                            .map(lambda x:(x[0],x[1][0][1][0],x[1][0][0],x[1][1],x[1][0][1][1]))

    # Top 5 generi preferiti dall'utente
    genres_user_preferences = top5_user_genre_preferences(userId,db)

    # Filtraggio dei film con almeno 30 recensioni, e genere tra i generi preferiti
    myuserTopRecommendationRDD = myuserRecommendationsRDD
                                        .filter(lambda x: x[3]>=30)
                                        .filter(lambda x: anyElem_List1_in_List2(x[4],
                                                          genres_user_preferences))

    # Filtraggio dei top 10 aventi rating più alto
    myuserTopRecommendationRDD = myuserTopRecommendationRDD
                                                .map(lambda x: (x[0],x[1],x[2],x[3]))
                                                .takeOrdered(10, key=lambda x: -x[2])

    # Convert RDD to dict and persist recommendation
    moviesList = []
    keys = ["movieId", "title", "rating", "ratings_count"]

    for lista in myuserTopRecommendationRDD:
      yourdic = dict(zip(keys, lista))
      moviesList.append(yourdic)

    userRecommendations = {}
    userRecommendations['user'] = userId
    userRecommendations['movies'] = moviesList

    user_movies_recommendation_collection.insert_one(userRecommendations)
```

3.3 Presentation layer: PowerBI

Tutti i risultati precedentemente ottenuti, oltre che ulteriori analytics operate sui dati, vengono fornite all'utente mediante un layer di presentazione, implementato tramite il software **PowerBI**. Quest'ultimo permette la progettazione di **dashboard**, che consentono in modo efficace l'analisi e l'estrazione di conoscenza dai dati.

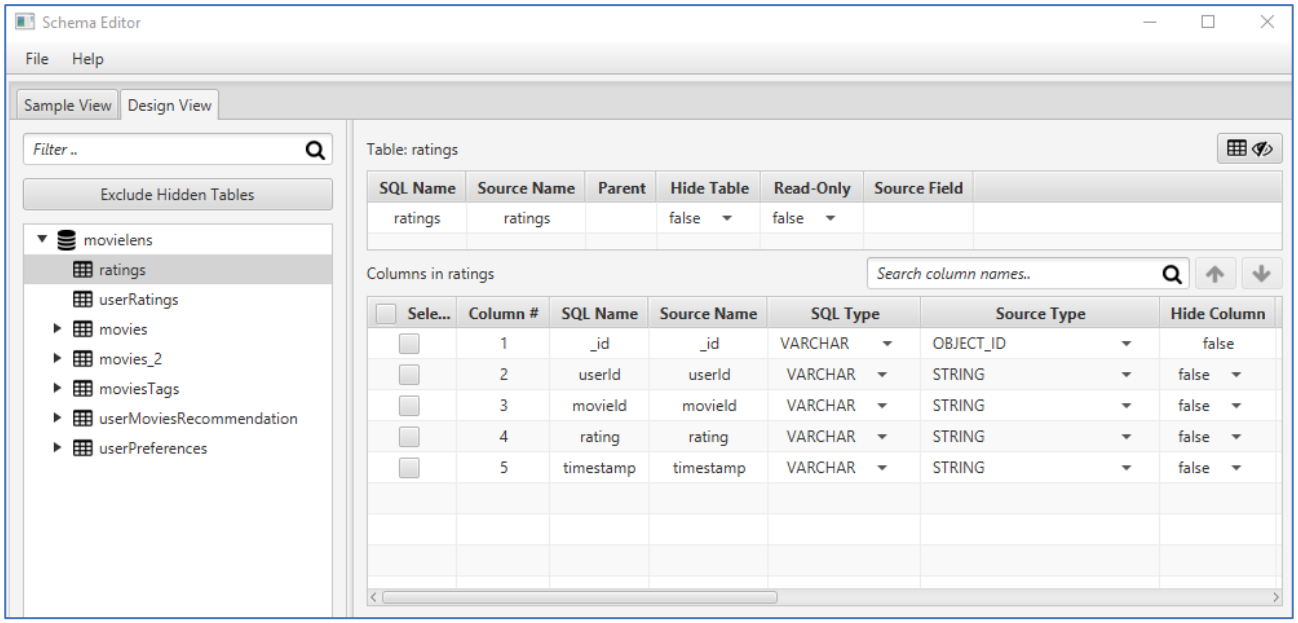
È necessario osservare che tutte le analytics successive, per esigenze di tipo computazionale, sono basate su un sottoinsieme dei dati risultanti da un filtraggio sui primi 100 utenti. Di seguito viene mostrato lo snippet di codice *PySpark* che permette tale operazione e salva i risultati in una collection mongoDB:

```
pipeline = [
      {"$addFields": {"userId": { "$toInt": "$userId" }}},
      {"$match": {"userId": {"$lt": 101}}},
      {"$out": "100userRatings" }
      ]
db.ratings.aggregate(pipeline, cursor={})
```

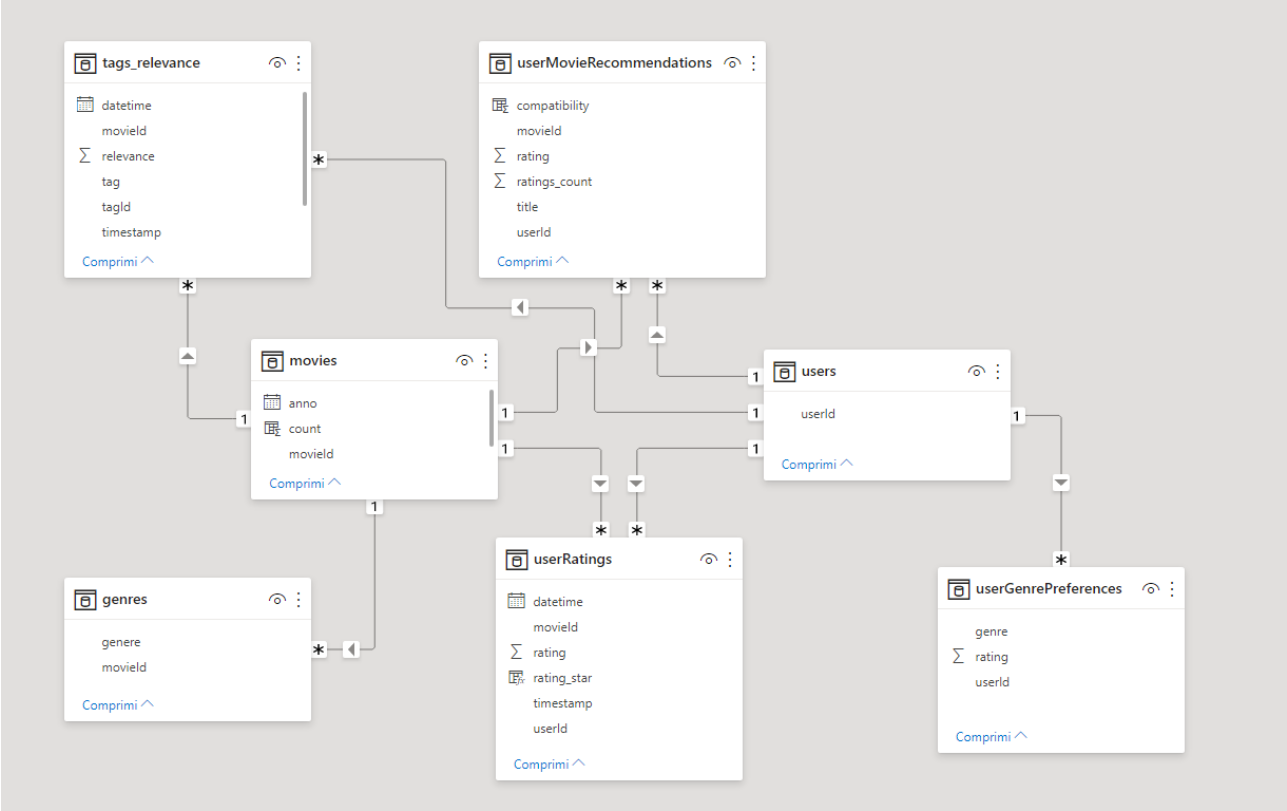
3.3.1 Import e modellazione dei dati

Per analizzare e visualizzare i dati memorizzati all'interno di MongoDB, con il tool PowerBI, è necessario utilizzare un connettore di tipo ODBC per stabilire la connessione diretta tra il layer di presentazione e il layer di data management.

In particolare, viene utilizzato il connettore di terze parti '*Simba MongoDB ODBC Driver*', che permette di importare le collection NoSQL mappandole in uno schema relazionale. Data la natura schemaless dei dati, infatti, prima di procedere con lo sviluppo delle dashboard è necessario modellare le entità del problema e le relazioni tra esse. Questa fase di mapping è utile inoltre per tradurre documents contenenti oggetti di tipo strutturato in entità disgiunte e relazionate tra loro.

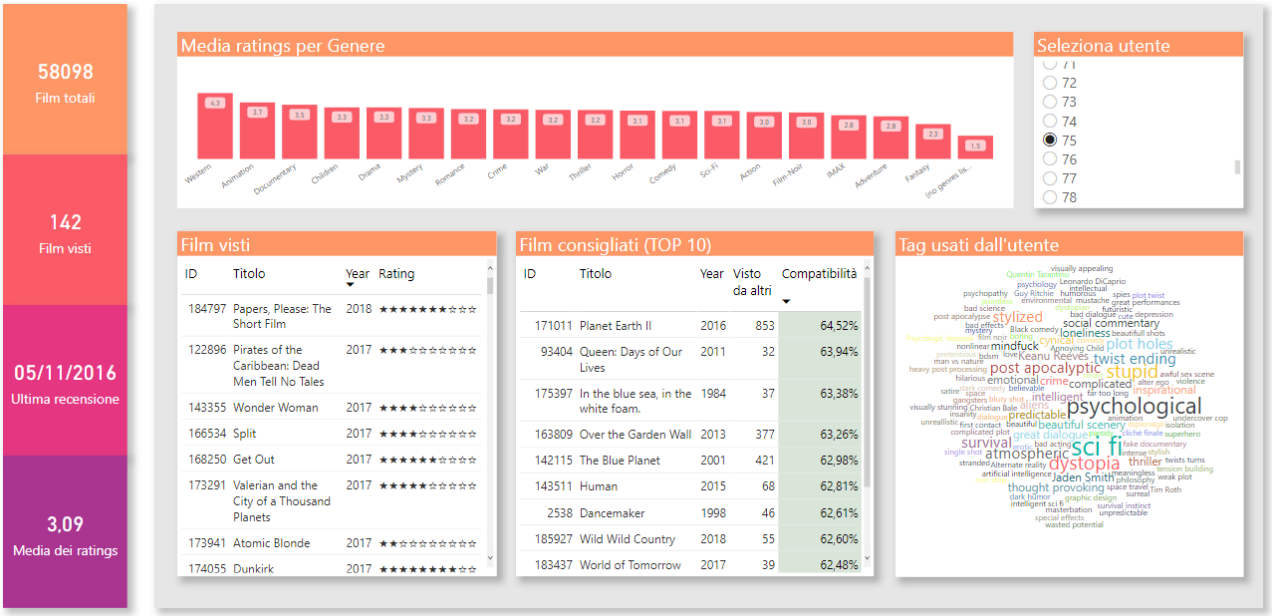


L'interfaccia di PowerBI permette intuitivamente di realizzare tale la modellazione. Lo specifico modello generato come output di questa fase è il seguente:



3.3.2 Implementazione delle dashboard

La **prima dashboard** implementata riguarda la presentazione dei risultati inerenti le raccomandazioni calcolate in precedenza, per ogni utente, oltre che altre analytics di interesse e che riguardano il singolo utente:



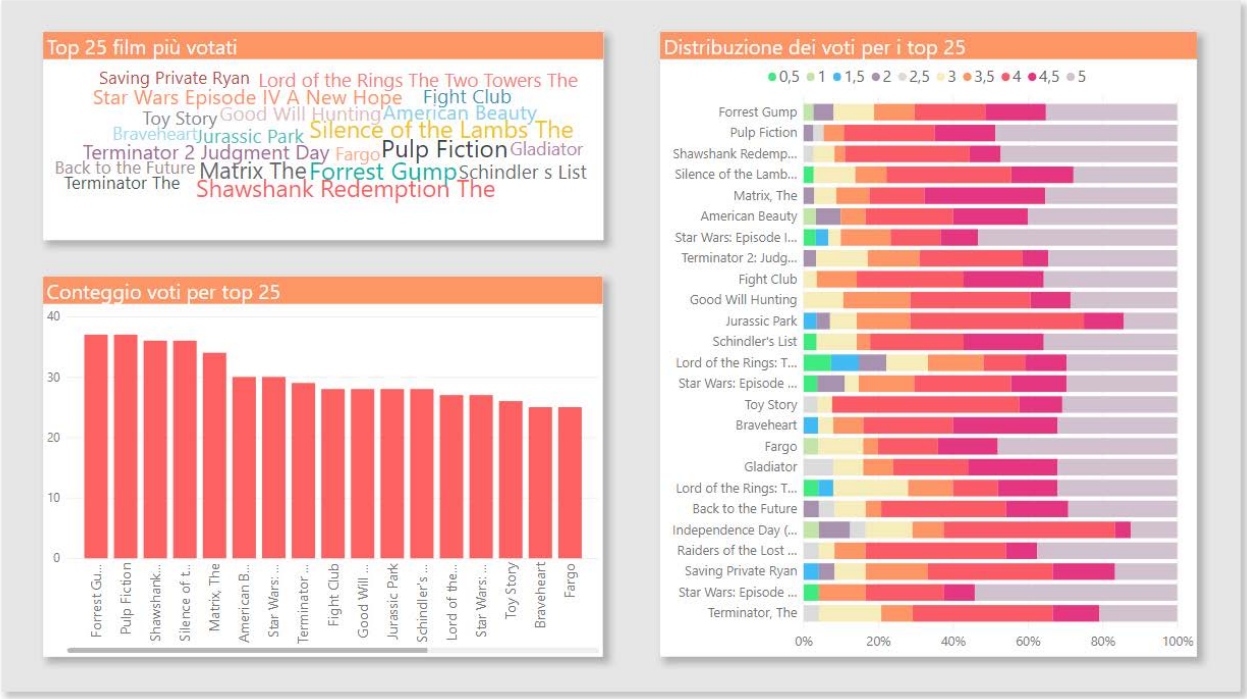
Selezionando l'utente dal filtro in alto a destra è possibile visualizzare le seguenti informazioni:

- Film visti dall'utente;
- Ultima recensione;
- Media dei ratings dell'utente;
- Media dei ratings dell'utente suddivisi per genere;
- Lista dei film visti, con relativo rating dato dall'utente;
- Word cloud di tutti i tags usati dall'utente nelle reviews;
- Lista dei top 10 film consigliati per l'utente, con relativo conteggio delle reviews ottenute finora e relativo grado di compatibilità con l'utente.

Dato che l'algoritmo ALS, per una precisa interazione user-film, fornisce una predizione numerica del rating che l'utente avrebbe dato a quel film, questa informazione può essere tradotta nel **grado di compatibilità** del film con l'utente, in modo molto semplice:

$$compatibility = \frac{predicted\ rating}{\max(predicted\ ratings)}$$

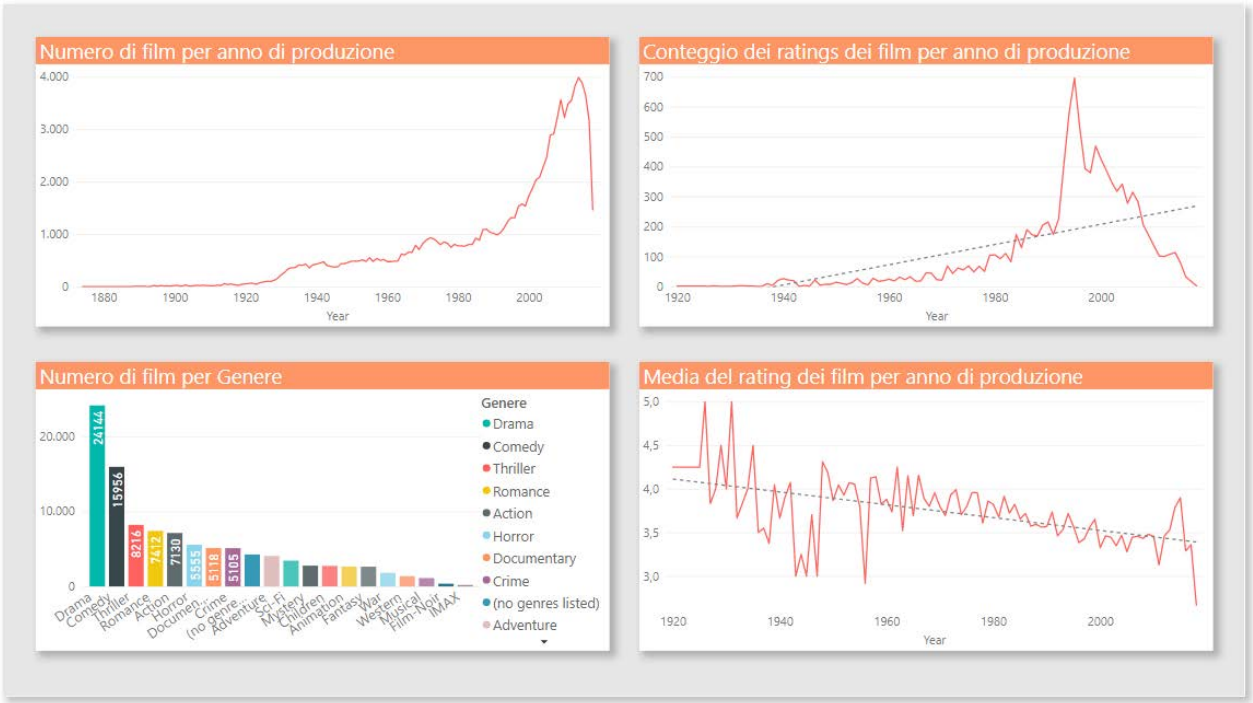
La **seconda dashboard** riguarda ulteriori analytics sull'intero dataset dei film, indipendentemente dall'utente.



Sono evidenziate le seguenti informazioni:

- Word cloud dei 25 film più votati finora;
- Conteggio dei voti ottenuti dai 25 film più votati;
- Distribuzione dei diversi voti ottenuti dai 25 film più votati.

La **terza dashboard** ha l'obiettivo di operare un'analytics sui film, che tenga conto anche dell'anno di produzione degli stessi:

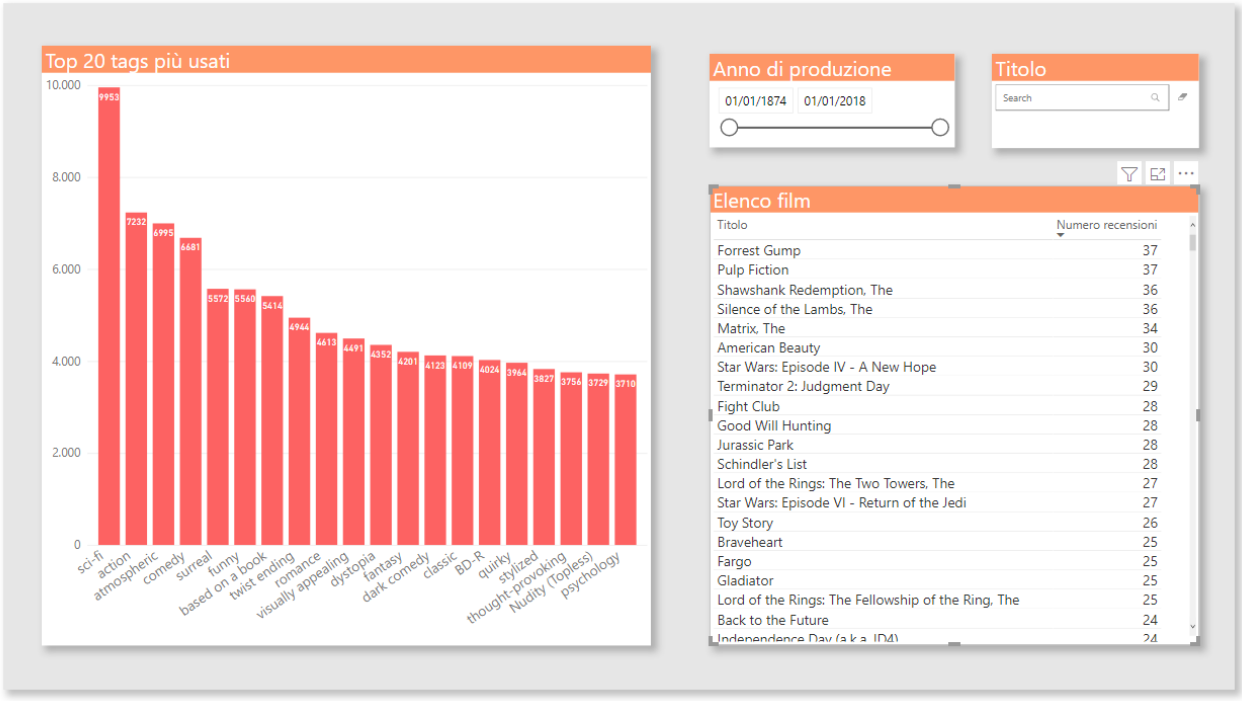


In particolare si visualizzano:

- Numero di film prodotti in relazione al genere;
- Numero di film prodotti per anno di produzione;
- Numero di ratings dei film per anno di produzione;
- Media dei ratings su tutti i film in funzione dell'anno di produzione.

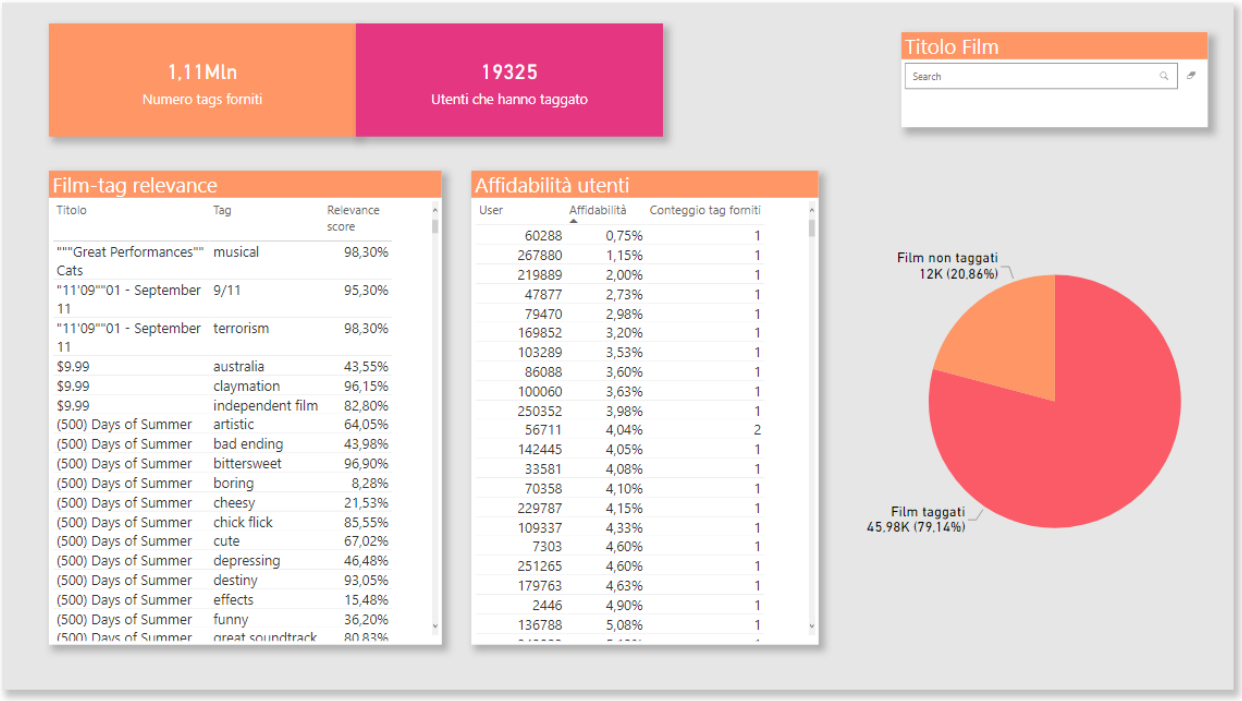
Analizzando graficamente questa dashboard è possibile mettere in relazione l'andamento del primo grafico con quello del quarto e constatare che, seppur il numero di film prodotti per anno sia aumentato notevolmente con il passare del tempo, il tasso di apprezzamento medio degli utenti invece presenta un trend negativo.

La **quarta dashboard**, invece, si focalizza sulle informazioni inerenti i tags attribuiti dagli utenti ai film:

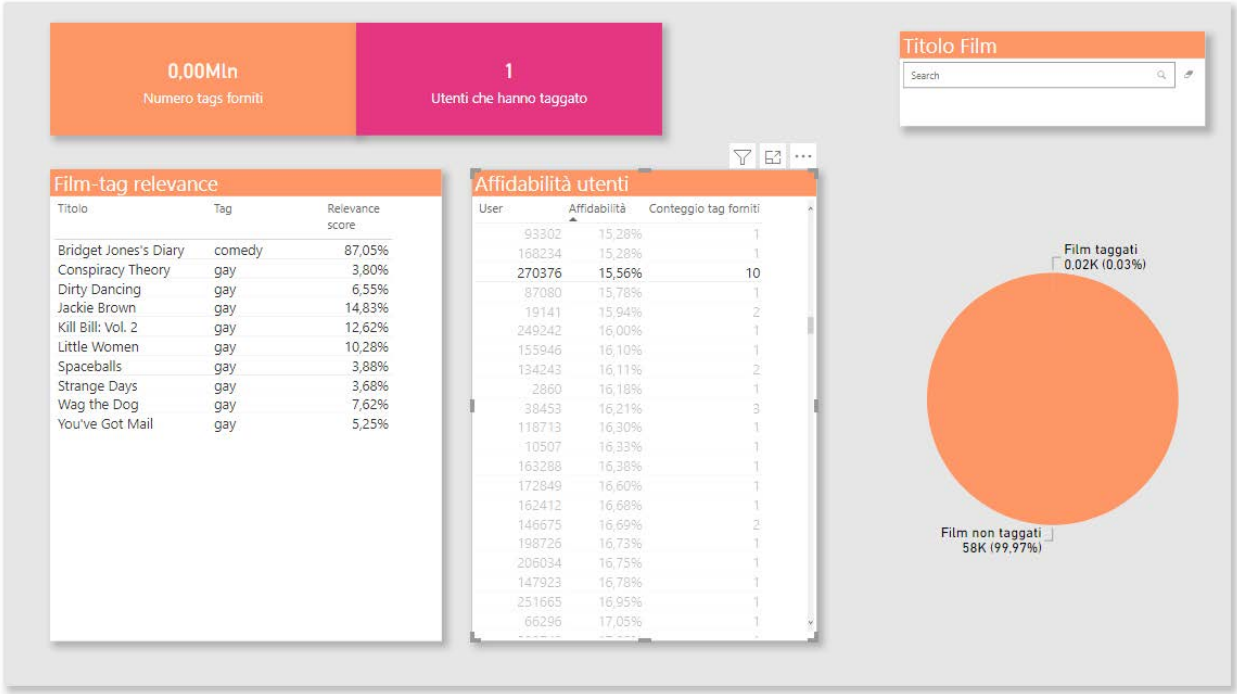


In particolare, vengono presentati i top 20 tags più usati dagli utenti, di ricercare tramite una barra di ricerca o selezionare tramite l'elenco, uno specifico film di cui si vogliono analizzare i tags attribuiti. Inoltre è fornita la possibilità di filtrare tutti i film per anno, per mostrare i tags più utilizzati in un dato periodo.

La **quinta dashboard** ha l'obiettivo di analizzare l'affidabilità degli utenti. Il dataset '*genome_score.csv*' fornisce, per ognuno dei tags, una metrica di **relevance** in relazione ad ogni film del dataset. Incrociando questa informazione con il dataset che contiene i tags attribuiti dall'utente ai film che ha visto, è possibile determinare se un preciso utente fornisce tags in modo affidabile.



L'idea è che si possa ritenere poco affidabile un utente che fornisce spesso tags poco rilevanti per i film votati, e quindi calcolando la media dei relevance score relativi ai tags forniti da un utente, è possibile effettuare un ranking sulla loro affidabilità.



Selezionando un preciso utente dalla lista "Affidabilità Utenti", è possibile ispezionare nella lista a sinistra tutti i film taggati dall'utente, con il relativo tag fornito, insieme al relevance score.

4 Future improvements

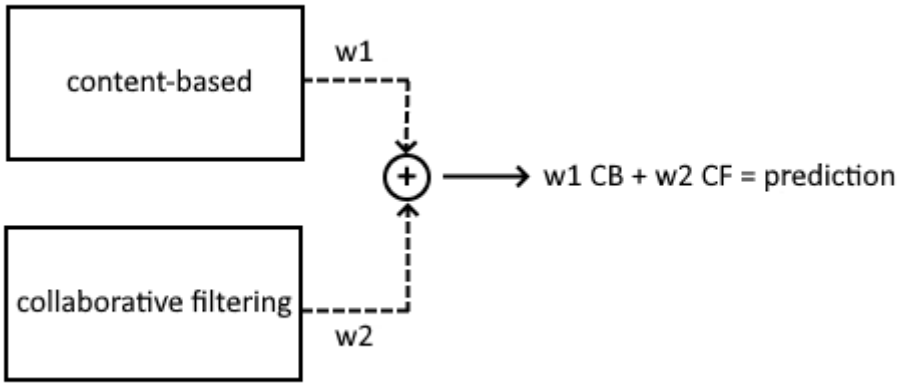
Per mettere in esercizio il sistema di raccomandazione implementato, si deve tener conto di ulteriori problematiche che non sono state affrontate durante la progettazione dell'engine di raccomandazione. La maggior parte delle problematiche deriva dal fatto che un sistema del genere difficilmente dovrà lavorare su dati statici e fornire soltanto delle analytics di tipo batch, ma sicuramente dovrà prevedere e supportare una certa mutabilità nei dati, derivante dall'interattività con gli utenti.

Cold Start Problem

In primo luogo, seppur considerato, non è stato risolto il **cold start problem**, inerente sia ad eventuali nuovi utenti registrati che eventuali nuovi film inseriti nel sistema.

Nuovo film: essendo il core dell'engine di raccomandazione basato sull'approccio collaborative filtering, se viene inserito un nuovo film nel sistema, e chiaramente non si hanno a disposizioni pregresse interazioni tra gli utenti e quel film, il sistema non lo includerà mai nei consigli.

Una possibile soluzione a questo problema è quello di inserire un core di raccomandazione content-based all'interno del sistema allo stesso livello di quello collaborative e quindi generare una raccomandazione effettuando una somma pesata dei risultati dei due sistemi, piuttosto che usarli in cascata, come si verifica nella soluzione proposta.



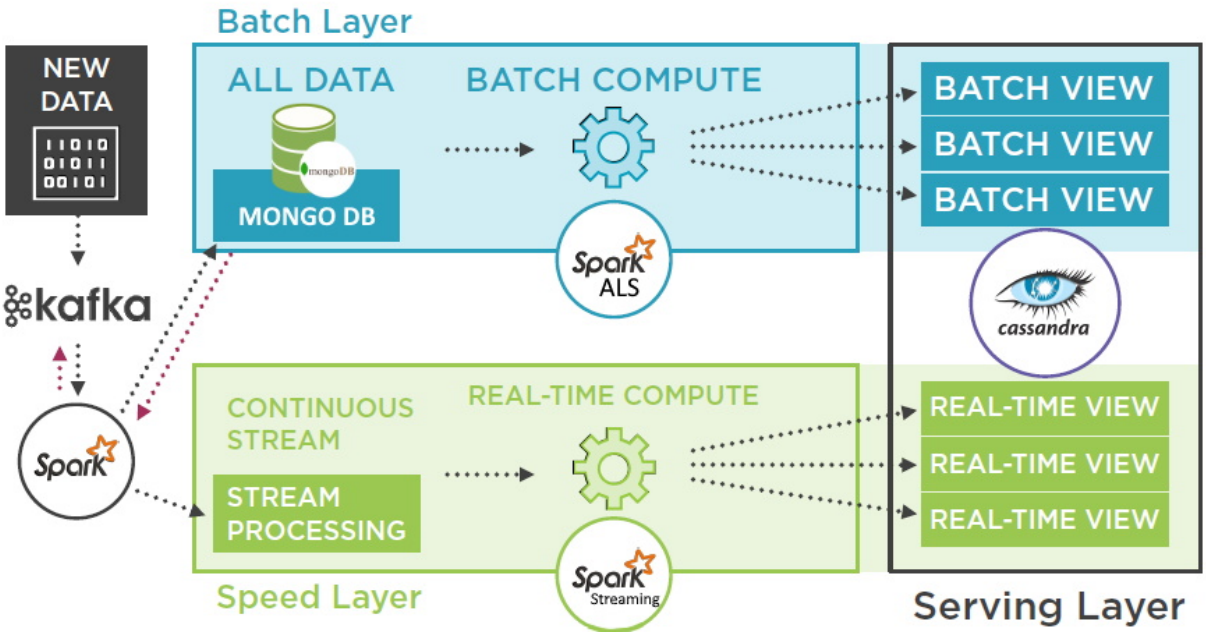
Nuovo utente: analogamente, quando nel sistema viene inserito un nuovo utente, l'approccio collaborative non riesce a trovare similarità con gli altri utenti, se non ha a disposizione alcuna interazione per quell'utente.

Una possibile soluzione, in questo caso, potrebbe essere quella di prevedere una fase iniziale in cui vengono chieste al nuovo utente le sue preferenze, in modo da avere sufficienti dati per il corretto funzionamento del suggeritore.

New user-movies interactions

Ulteriore problematica che dovrebbe essere affrontata è fornire il supporto per l'analisi non solo dei dati storici ma anche dei nuovi feedback forniti real-time durante l'esercizio della piattaforma. Questo permetterebbe al sistema di mettere insieme la conoscenza pregressa e i nuovi dati in streaming, per ricalcolare iterativamente le preferenze dell'utente e quindi fornire raccomandazioni sempre più affini a lui.

Una possibile soluzione può essere individuata nell'utilizzo di un'**architettura di tipo lambda** che consta dei seguenti layers:



- Un **batch layer**, per l'analisi dei dati storici dell'utente, che può essere rappresentato dalla esatta architettura esposta in precedenza, e che fornisce una batch view delle top K raccomandazioni basate sui ratings storici dell'utente.
- Uno **speed layer** che si occupa dell'analisi dello streaming dei dati di feedback forniti real-time dagli utenti, e potrebbe fornire, ad esempio, una view sui **top k trending film** che correntemente, in tempo reale, sono visti maggiormente dagli altri utenti.
- Il **serving layer**, si dovrebbe occupare semplicemente del merge dei risultati forniti dalle due views, per proporre agli utenti film basati allo stesso tempo sull'analisi batch e sull'analisi dello streaming dei feedback.