Definizione ed Implementazione di un Sistema di Raccomandazione Distribuito per film e Modellazione di Eventi Complessi

Prof. Ing. Tommaso di Noia
Prof.ssa Marina Mongiello
Mauro Losciale
Pietro Tedeschi

Logica e Intelligenza Artificiale
Ingegneria del Software Avanzata
Laurea Magistrale in Ingegneria Informatica
Politecnico di Bari
A.A 2015 - 2016

Indice

1 Introduzione **1**

2 Stato dell'arte **1**
2.1 Introduzione ai sistemi CEP 1
2.2 Sistemi di Raccomandazione 2
2.2.1 Metriche di Valutazione 5
2.2.2 Introduzione alla Matrix Factorization 6
2.3 Introduzione al Data Stream Processing 6
2.4 Il paradigma Publish-Subscribe 7
2.5 Il pattern Facade 7
2.6 Il pattern Singleton 8
2.7 Il pattern Model-View-Controller (MVC) 9
2.8 La tecnologia WebSocket 10

3 Analisi del progetto **10**

4 Soluzione proposta **10**
4.1 La libreria Spark 10
4.1.1 Spark Streaming 11
4.1.2 Spark MLlib 12
4.1.3 Spark SQL 12
4.2 Apache Kafka 12
4.2.1 Integrazione con Spark Streaming 13
4.3 Il framework Node.js 14
4.3.1 Angular.js 14
4.4 La libreria socket.IO 15

5 Conclusioni e sviluppi futuri **15**

Bibliografia **16**

6 Sorgenti **17**

Appendice A Sorgenti Spark **17**

Appendice B Sorgenti Node.js **22**

1 Introduzione

2 Stato dell'arte

2.1 Introduzione ai sistemi CEP

L'incremento dei dispositivi interconnessi e delle applicazioni distribuite, richiede un'elaborazione continua del flusso dati. Esempi di tali applicazioni, vanno dal traffico generato dalle Wireless Sensor Networks (WSN) al flusso dati relativo agli indici finanziari, dal monitoraggio stradale alla Clickstream Analysis.

Un sistema ad eventi complessi, meglio conosciuto come *Complex Event Processing* (CEP), modella il flusso informativo dei dati, visualizzando gli elementi come notifiche di ciò che sta accadendo nel mondo esterno. I dati vengono rilevati e filtrati utilizzando dei pattern (oppure le *processing rules*), i quali hanno il compito di rappresentare il modello di riferimento con l'informazione da rilevare, per poi farla pervenire alle rispettive parti (ad esempio, i dispositivi che effettuano una sottoscrizione ad un determinato topic nel paradigma *publish-subscribe*). L'obiettivo di un sistema CEP consiste nell'identificare eventi significanti e rispondere ad essi nel più breve tempo possibile. Un pattern o una regola, può essere definita mediante un linguaggio basato su query, il cosiddetto **Event Query Language**.

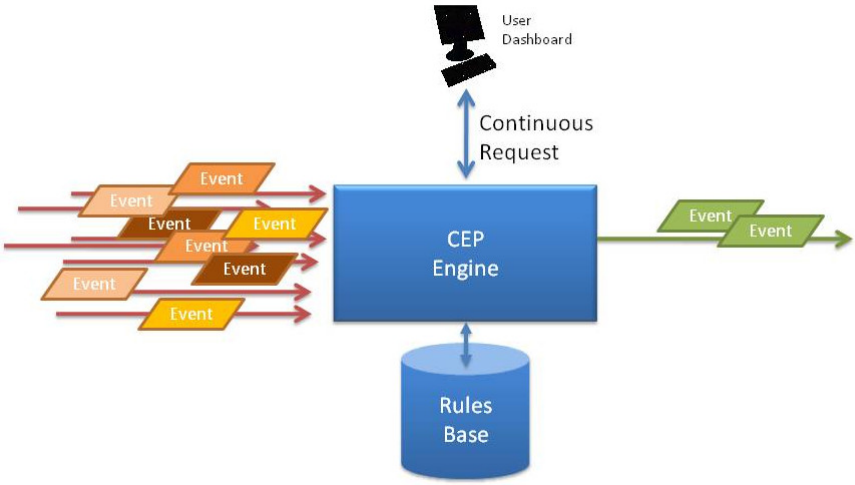

Figura 1: Architettura di un Sistema CEP [7]

Gli **Event Query Languages** possono essere raggruppati in tre categorie: Composition Operators, Data Stream Query Languages e Production Rules. I Composition Operators identificano gli eventi complessi partendo dalla composizione dei singoli eventi, utilizzando operatori quali congiunzione, negazione o di sequenza per la costruzione delle espressioni. Esempi rilevanti sono IBM Active Middleware Technology e ruleCore.

I **Data Stream Query Languages** sono basati sul linguaggio SQL; gli stream di dati sono semplicemente tuple convertite per database relazionali, in modo che si possano eseguire query SQL su di esse. E' utile citare i seguenti approcci: CQL, Coral8, StreamBase, Aleri, Esper e così via.

Le **Production Rules** specificano le azioni che devono essere eseguite quando il sistema si trova in determinati stati; non è un linguaggio ad eventi, ma costituisce un approccio importante nei sistemi CEP. Un esempio pratico è TIBCO Business Events.

Un altro fattore importante è il tempo. Sono due le parti da considerare quando si parla del tempo, il tempo della finestra ed il tempo dell'evento informativo. Il tempo della finestra mostra gli eventi che vengono esaminati in un determinato intervallo. Il tempo dell'evento invece, porta con se informazioni relative alla data, ora di rilevazione, tempo di transizione, ed intervallo di elaborazione.

I contributi relativi ai sistemi CEP, arrivano da diverse comunità, a partire da quelle che si occupano di sistemi distribuiti, automazione industriale, sistemi di controllo, monitoraggio delle reti, Internet of Things, e middleware in generale.

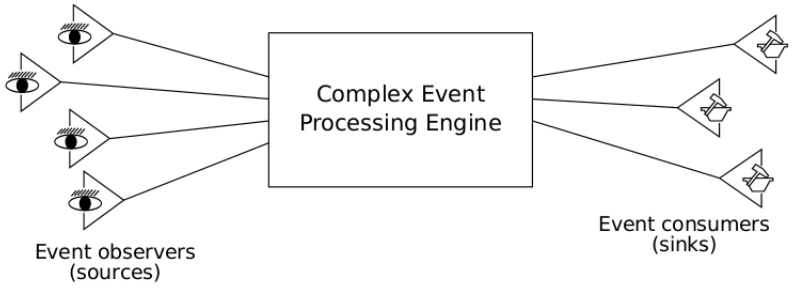

Figura 2: Funzionamento di un Sistema CEP (High-Level) [3]

Come possiamo vedere dalla Figura 2, viene associata una semantica dettagliata agli elementi informativi da processare. Da una parte abbiamo gli **Event Observer**, i quali rappresentano la sorgente dei dati e degli eventi da notificare; in seguito abbiamo il **CEP Engine**, responsabile del filtraggio e della notifica degli eventi ai nodi *sink*, identificati come **Event Consumers** [3,6].

2.2 Sistemi di Raccomandazione

I sistemi di raccomandazione, raccolgono informazioni sulle preferenze di utente in corrispondenza di un insieme di elementi (ad esempio, film, musica, libri, giochi, viaggi, siti web, applicazioni, gadget). L'informazione può essere acquisita in maniera esplicita (tipicamente ciò viene fatto acquisendo il voto di un utente) o implicitamente (analizzando il comportamento dell'utente, ad esempio musica ascoltata, applicazioni scaricate, siti web visitati, libri letti). Inoltre i sistemi di raccomandazione, possono tener conto anche delle caratteristiche demografiche dell'utente (ad esempio età, nazionalità, sesso); dei contenuti informativi presenti nel mondo del Web 2.0, ad esempio all'interno delle piattaforme di Social Networking, quali follower, followed, twit, like, post; dei dati provenienti dai dispositivi caratterizzanti l'Internet of Things (ad esempio, coordinate GPS, RFID, segnali medici inviati in real-time).

I sistemi di raccomandazione utilizzano diverse sorgenti informative al fine di fornire all'utente finale una migliore Quality of Experience relativa alla predizione ed alla raccomandazione degli elementi che potrebbero interessargli. La tecnica del Collaborative Filtering (CF), ha un ruolo fondamentale nella raccomandazione, sebbene viene spesso usata anche con altre tecniche di filtraggio basate sul contenuto o basate sulla conoscenza. Il CF si basa sulla cronologia decisionale dell'utente: oltre alle nostre esperienze, facciamo le nostre decisioni anche in base alla conoscenza che ci circonda.

Il processo con cui un sistema di raccomandazione generi una raccomandazione, è basato sulla combinazione delle seguenti considerazioni:

1. La tipologia di dati disponibili nel databse (votazioni, informazioni di registrazione dell'utente, caratteristiche peculiari del contenuto informativo, relazioni sociali)

2. L'algoritmo di filtraggio usato (demografico, basato sul contenuto, collaborativo, basato sulle relazioni sociali, dipendente dal contesto e ibrido).

3. Il modello scelto (basato sull'uso diretto dei dati: 'memory-based', oppure un modello generato usando tali dati: 'model-based').

4. Le tecniche impiegate: approccio probabilistico, reti Bayesiane, algoritmi di tipo nearest neighbors, algoritmi genetici, reti neurali, logica fuzzy.

5. Livello di dispersione del database e scalabilità desiderata.

6. Capacità di elaborazione del sistema (tempo di elaborazione e consumi di memoria).

7. L'obiettivo da raggiungere (predizioni e raccomandazioni)

8. La qualità del risultato desiderata (ad esempio la precisione).

La ricerca nell'ambito dei sistemi di raccomandazione, richiede che i dati siano di dominio pubblico, al fine di semplificare la ricerca sulle tecniche innovative relative all'analisi dei dati. Esempi di dataset pubblici presenti in letteratura, sono Last.Fm, Delicious, Netflix, MovieLens. Le funzionalità interne per i sistemi di raccomandazione, sono caratterizzate dagli algoritmi di filtraggio. Gli algoritmi di filtraggio vengono classificati nel modo seguente:

- Collaborative Filtering
- Demographic Filtering
- Content-Based Filtering
- Hybrid Filtering

Il **Content-Based Filtering** consente di creare raccomandazioni basate sulle scelte fatte in passato da un utente (ad esempio, in un sito E-Commerce, se l'utente ha acquistato una fiction cinematografica, probabilmente il sistema di raccomandazione gli consiglierà una fiction recente, che non ha ancora acquistato sul sito). La tecnica consente inoltre di generare la raccomandazione utilizzando il contenuto dell'oggetto, ad esempio il testo, le immagini, l'audio.

Il **Demographic Filtering** si basa sul principio che gli individui con caratteristiche personali comuni, quali età, sesso, luogo di residenza e così via, avranno le stesse preferenze.

Il **Collaborative Filtering** consente agli utenti di attribuire un voto ad un insieme di elementi (filmati, canzoni, film, libri, all'interno di una piattaforma web) salvando le proprie preferenze all'interno di un database, e consentendo di creare una raccomandazione specifica per ogni utente. I voti degli utenti possono essere anche acquisiti in maniera implicita (ad esempio il numero delle che viene ascoltata una canzone, il numero delle consultazioni relative ad una risorsa). L'algoritmo utilizzato maggiormente per il Collaborative Filtering è il k Nearest Neighbors (kNN).

Nella versione "*user to user*", il KNN esegue i seguenti task per generare la raccomandazione:

1. Determinare i k utenti vicini all'utente corrente.

2. Implementare un approccio che tenga conto degli elementi "*vicini*" non ancora votati dall'utente corrente.

3. Estrarre le predizioni dal passo 2 e selezionare le N raccomandazioni.

La similarità può essere distinta in **item-item**: due elementi sono simili se tendono ad ottenere lo stesso rate dagli utenti; oppure **user-user**: due utenti sono simili se tendono a dare lo stesso rate agli elementi.

Uno dei problemi più noti del filtro collaborativo è il "*Cold Start*", che si potrebbe verificare nel caso in cui si considerano elementi che non sono stati votati in precedenza.

L'**Hybrid Filtering** è una combinazione di Collaborative Filtering e Demographic Filtering, oppure una combinazione tra Collaborative Filtering e Content-Based Filtering che sfrutta i pregi di ciascuna di queste tecniche. Il metodo è basato su metodi probabilistici come gli algoritmi genetici, genetica fuzzy, reti neurali, reti Bayesiane, clustering.

Inoltre possiamo suddividere i metodi in *Memory-Based* e *Model-Based*:

I **Memory Based** conservano in memoria le informazioni associate ad ogni utente, item o voto all'interno del sistema. Queste informazioni costituiscono la *Knowledge Base* sulla quale lavora l'algoritmo di predizione. Idealmente i sistemi di tipo memory based devono essere in grado di generare l'insieme di predizioni in maniera efficiente, processando tutte le informazioni contenute nella matrice *Utenti/Item*.

La matrice *Utenti/Item* definita all'interno del sistema di raccomandazione contiene entry i, j che rappresentano un voto dell'utente *i-esimo* per l'elemento *j-esimo*; nel caso in cui dovesse mancare una preferenza per un determinato item, il valore viene posto a 0.

$$M = \begin{array}{c} \\ u_1 \\ u_2 \\ \dots \\ u_i \\ \dots \\ u_n \end{array} \begin{array}{c} \begin{array}{cccccc} i_1 & i_2 & \dots & i_j & \dots & i_m \end{array} \\ \begin{bmatrix} \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & r_{i,j} & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \end{bmatrix} \end{array}$$

Poiché i sistemi di raccomandazione sono caratterizzati da una elevata dimensione dello spazio degli utenti e degli item, il funzionamento degli algoritmi *memory-based* è stato definito sull'ipotesi di determinare un grado di similarità tra gli utenti, che permetta di estrapolare dalla matrice dei voti, le informazioni associate ai soli utenti simili all'utente attivo.

I **Model Based**, utilizzano l'insieme dei voti espressi dagli utenti per costruire un modello statistico di preferenze su cui generare le predizioni.

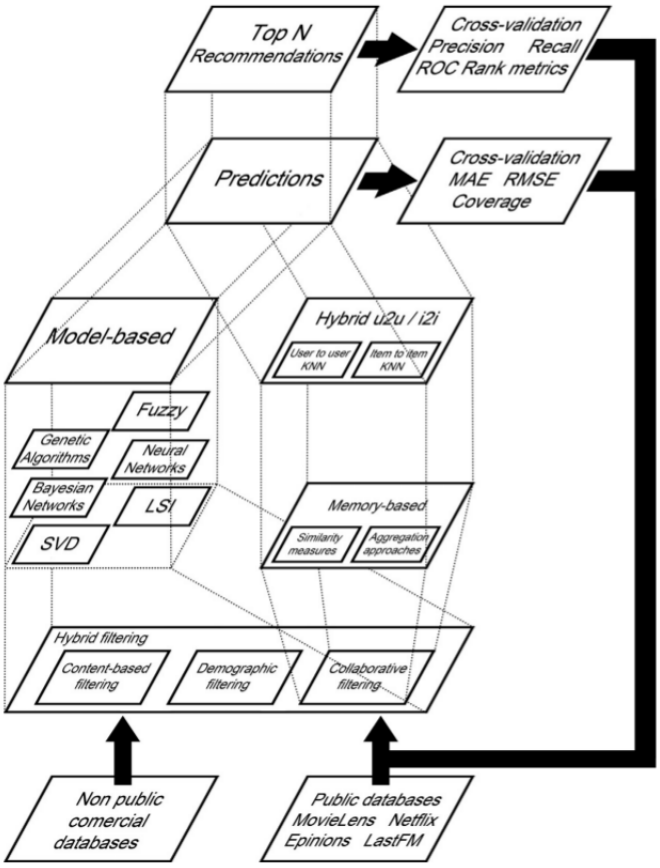

Figura 3: Modelli di Raccomandazione [2]

E' utile citare i metodi di riduzione basati sulla Matrix Factorization, la tecnica model-based Latent Semantic Index (LSI), il metodo di riduzione Singular Value Decomposition (SVD) e le tecniche di clustering.

Nella Figura 3 viene illustrato un diagramma con le tecniche principali e gli algoritmi di raccomandazione maggiormente utilizzati in letteratura [2].

2.2.1 Metriche di Valutazione

La qualità di un sistema di raccomandazione può essere valutata dai risultati forniti in output. Le tipologie di metriche usate, dipendono dal tipo di CF usato. Le metriche possono essere categorizzate in: Predictive Accuracy, come ad esempio il Mean Absolute Error (MAE) e sue varianti; Classification Accuracy metrics, come la precision, recall, F1-measure, e sensibilità ROC; Rank Accuracy, come il Mean Average Precision (MAP), e così via. Desideriamo concentrarci sulla metrica **Root Mean Squared Error** (RMSE). Infatti la radice quadrata dell'errore quadratico medio, è una metrica molto utilizzata per la raccomandazione di film.

$$RMSE = \sqrt{\frac{1}{n}\sum_{i,j}(p_{i,j} - r_{i,j})^2}$$

dove n è il numero totale dei voti assegnati dagli utenti,$p_{i,j}$ è il voto predetto per l'utente i sull'elemento j, ed $r_{i,j}$ è il voto attuale. L'RMSE consente di valutare in maniera precisa i contributi degli errori assoluti tra i valori predetti ed i valori reali [8].

2.2.2 Introduzione alla Matrix Factorization

I modelli di Matrix Factorization possono essere utilizzati per ricercare quali sono i fattori latenti che caratterizzano le interazioni tra due tipologie di entità., tale che le interazioni user-item siano modellate come prodotti interni nello spazio. Ad esempio, due utenti potrebbero votare un determinato film con un voto alto, se ad entrambi piacciono gli attori, o il genere del film, ecc. Quindi, se si riescono a scoprire quali sono i fattori latenti dietro la scelta di un utente, si riesce anche a predire un voto o una possibile preferenza di un altro film rispetto all'utente, perchè le feature associate all'utente dovrebbero convergere alle feature associate all'elemento.

Analizziamo la tecnica dal punto di vista matematico. Sia U l'insieme degli utenti, e sia D l'insieme degli elementi. Sia R la matrice dei voti assegnati dagli utenti agli utenti, di dimensione $|U| \times |D|$. Supponiamo di voler cercare i K fattori latenti. Innanzi tutto bisogna cercare le due matrici, P (una matrice $|U| \times K$) e Q (una matrice $|D| \times K$) tale che il loro prodotto sia una stima del tipo: $R \approx P \times Q^T = \hat{R}$. Ogni riga di P rappresenta il grado di associazione tra un utente e le feature, mentre ogni riga di Q tra un elemento e le feature. Al fine di poter calcolare la predizione di un rating su un elemento d_j votato dall'utente u_i, definiamo il prodotto interno tra i due vettori corrispondenti come:

$$\hat{r_{i,j}} = {p_i}^T q_j = \sum_{1}^{k} p_{ik} q_{kj}$$

Quindi un modo per ovviare a questo problema consiste nell'inizializzare le due matrici con alcuni valori, e calcolare la differenza tra i valori stimati ed i valori reali, e infine minimizzare questa differenza in maniera iterativa. Un metodo per minimizzare l'errore calcolato può essere l'algoritmo di discesa del gradiente stocastico oppure l'algoritmo **Alternating Least Squares** (ALS) [9].

In generale, mentre l'algoritmo della discesa del gradiente stocastico è più semplice e più veloce di ALS, ALS risulta essere preferibile in molteplici casi, tra cui la parallelizzazione computazionale. In ALS infatti, il sistema calcola ogni vettore d_j indipendentemente dagli altri elementi, così come accade per u_i. Ciò comporta un incremento della parallelismo a livello computazionale.

2.3 Introduzione al Data Stream Processing

Di recente, è nata una nuova classe di applicazioni *data-intensive*, nei quali i dati vengono modellati come relazioni transienti piuttosto che persistenti. Definiamo con il termine Data Stream, una sequenza di elementi codificati in digitale rappresentanti un'informazione. Esempi di applicazioni **Data Stream** vanno dal campo finanziario, monitoraggio del traffico di rete, applicazioni web, sicurezza, telecomunicazioni, gestione dati e reti di sensori. Nel modello Data Stream, i dati possono essere tuple relazionali, come ad esempio misurazioni di rete, registro chiamate, pagine web visitate, dati provenienti da sensori, e così via. Tuttavia, questi dati costituiscono un flusso multiplo e tempo variabile, tali da richiedere una nuova tecnologia al fine di poter esser gestiti.

Nei vari scenari applicativi sopra citati, non è efficiente e ne tanto meno semplice gestire i dati in arrivo all'interno di un Database Management System (DBMS) tradizionale. Infatti, i DBMS tradizionali non sono progettati per trattare in maniera rapida un flusso di dati contiguo. Di conseguenza sono nati i Data **Stream Management System** (DSMS) in grado di gestire un flusso dati continuo. I DSMS sono in grado di offrire un'elevata flessibilità nell'eseguire query continue sullo stream dati. Poichè i DSMS sono data-driven, ogni qualvolta viene eseguita una query, essa produrrà nuovi risultati non appena arrivano nuovi dati da processare.

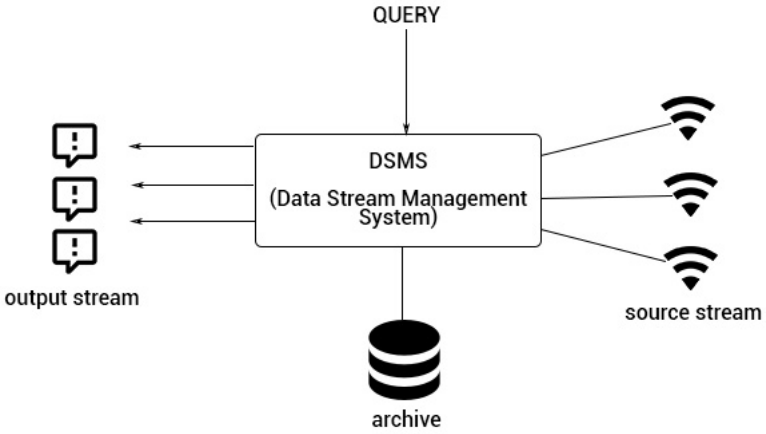

Figura 4: Architettura DSMS

I dati presenti all'interno di uno stream, arrivano online. Il sistema che elabora lo stream, non gestisce l'ordine di arrivo degli elementi da processare. Gli stream inoltre hanno la peculiarità di non avere una dimensione massima prefissata, ma sono altamente variabili. Dopo che il dato è stato processato, o viene scartato, oppure viene archiviato all'interno di uno spazio di storage [1].

2.4 Il paradigma Publish-Subscribe

Il **Publisher/Subscribe** (pub/sub) è un design pattern, o uno stile architetturale, organizzato in modo che le entità comunicano tra loro in maniera asincrona.

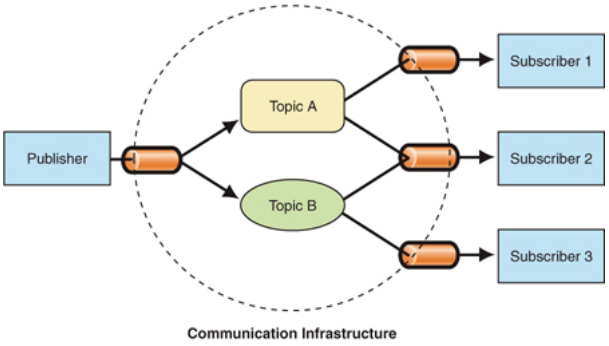

Figura 5: Publish/Subscribe

I publisher inviano i messaggi utilizzando un canale di comunicazione denominato topic. I publisher e di subscriber sono generalmente utenti che si sottoscrivono ad uno o più topic, ed in base alla loro posizione gerarchica pubblicheranno/riceveranno le informazioni relative al canale di interesse sottoscritto. Il sistema (vedi Figura 5) infatti ha il compito di distribuire i messaggi pubblicati dal publisher, a tutti i subscriber iscritti al topic. Infatti quando viene consegnato un nuovo messaggio su un determinato topic, ed un subscriber è iscritto a quest ultimo, il sub sarà notificato con un evento.

2.5 Il pattern Facade

Il design pattern **Facade**, è un pattern strutturale che consente di fornire un'interfaccia semplice ed unificata, che consente di accedere ai diversi sottosistemi dotati di interfacce complesse e

diverse tra loro.

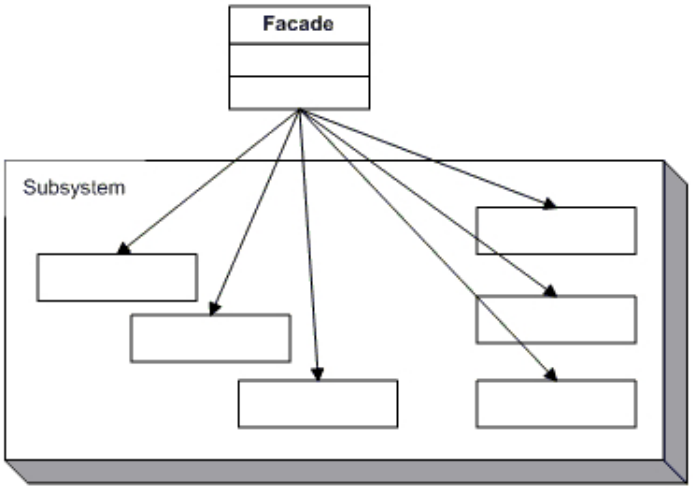

Figura 6: Diagramma UML del Pattern Facade

Le classi e gli oggetti definiti in questo pattern sono:

- **Facade**
  - Conosce le classi relative al sottosistema che sono responsabili di una richiesta.
  - Delega le richieste del client all'oggetto del sottosistema appropriato.
- **Classi del sottosistema**
  - Implementa le funzionalità di un sottosistema.
  - Gestisce i compiti assegnati dall'interfaccia *Facade*.
  - Nasconde la complessità ed i dettagli realizzativi con l'oggetto *Facade*.

2.6 Il pattern Singleton

Il **Singleton** è un design pattern creazionale che ha lo scopo di garantire che di una determinata classe venga creata una ed una sola istanza, e di fornire un punto di accesso globale a tale istanza.

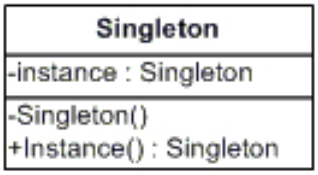

Figura 7: Diagramma UML del Pattern Singleton

Le classi e gli oggetti definiti in questo pattern sono:

- **Singleton**
  - Garantisce che un sistema istanzi un massimo di un oggetto di una classe.
  - Consente al client di accedere all'unica istanza creata.

2.7 Il pattern Model-View-Controller (MVC)

Il pattern MVC è un pattern architetturale in grado di separare i dati dell'applicazione (contenuti nel Modello) dai componenti per la presentazione grafica (Vista) e la logica per l'elaborazione dell'input (Controllore).

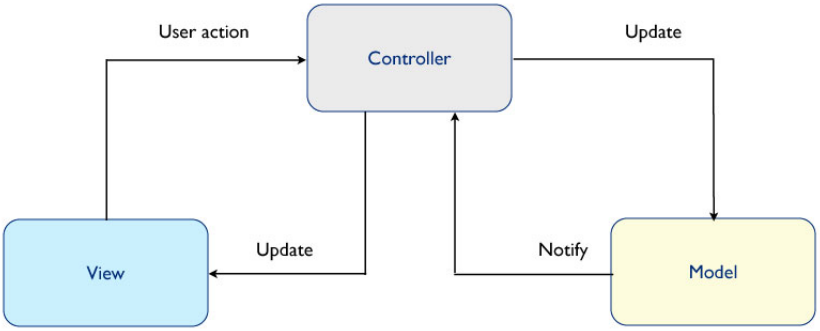

Figura 8: Il pattern MVC

- **Model**
  - Contiene le funzionalità di base ed i dati.
  - Incapsula lo stato dell'applicazione.
  - Mostra le funzionalità dell'applicazione.
  - Notifica alla View i cambiamenti.
  - Risponde alle query sullo stato.
- **View**
  - Mostra le informazioni all'utente.
  - Interpreta il Model.
  - Richiede aggiornamenti dal Model.
  - Consente al Controller di selezionare la View.
- **Controller**
  - Incapsula il comportamento dell'applicazione.
  - Mappa gli input dell'utente agli aggiornamenti del Model.
  - Seleziona la View dopo un input.

Separando il Model dalla View e dal Controller, si da la possibilità di associare più View allo stesso Model. Nel caso in cui l'utilizzatore modifica i dati contenuti nel Model mediante il Controller di una determinata View, tutte le altre View ricollegate a quel model ne propagano la variazione.

2.8 La tecnologia WebSocket

3 Analisi del progetto

4 Soluzione proposta

4.1 La libreria Spark

Apache Spark è un sistema di cluster computing di tipo general-purpose, scalabile e veloce. Dispone di API di alto livello in **Java**, **Scala**, **Python** ed **R**, e un engine ottimizzato che supporta grafi di esecuzione generici. Integra inoltre un ampio set di tool come **Spark SQL**, per structured data processing, **MLlib** per il machine learning e **Spark Streaming**, descritti nelle sezioni successive. Spark è eseguibile sia su sistemi Windows che UNIX-like (Linux, Mac OS) [5].

Una delle possibili configurazioni di un sistema Spark è la modalità *cluster*, mostrata in Figura 9.

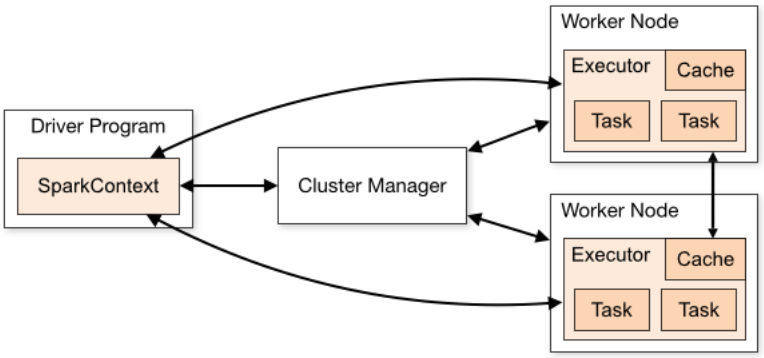

Figura 9: Configurazione in Spark di tipo Cluster Mode [5]

Le applicazioni Spark sono eseguite come un set di processi indipendenti sul cluster, coordinati dall'oggetto *SparkContext* del programma sorgente (detto **driver program**). Precisamente, il programma driver può connettersi su diversi tipi di *cluster managers* (ad esempio un cluster di tipo Standalone, Mesos o YARN), il quale alloca le risorse a disposizione delle applicazioni. Una volta connesso, Spark scansiona i nodi del cluster alla ricerca degli **executor** (detti anche *worker node*), i quali eseguono effettivamente i task e il data storage delle applicazioni. A questo punto il driver invia il codice dell'applicazione agli executor (tipicamente un file JAR o Python incluso nello SparkContext) e schedula i task per l'esecuzione parallela [5].

Alcune considerazioni riguardo tale architettura sono:

- Ogni applicazione gestisce i propri workers, i quali restano attivi durante tutto il ciclo di vita ed eseguono task multipli in thread multipli. Questo implica un isolamento tra le applicazioni, sia lato scheduling (ogni driver schedula i propri tasks) sia lato executor (tasks relativi ad applicazioni differenti risiedono in JVM differenti). Tuttavia ciò implica che non è possibile condividere nativamente i dati tra applicazioni diverse, a meno di utilizzare uno storage system esterno;

- Il driver deve poter gestire le connessioni con i workers durante l'intero ciclo di vita dell'applicazione. Per questo motivo dev'essere sempre garantita la visibilità a livello di rete tra driver e workers durante l'esecuzione;

- È necessario che driver e worker abbiano, a livello di rete, una distanza relativamente breve, preferibilmente nella stessa LAN, affinché lo scheduling sia rapidamente eseguito [5].

Il principio di funzionamento di Spark si basa sostanzialmente sul concetto di *Resilient Distributed Dataset* (**RDD**). Un RDD è una collezione di dati su cui è possibile operare parallelamente, ed è distribuita su tutti i nodi del cluster come file system Hadoop oppure è generata da una collezione esistente in Java o Scala [5].

Una seconda astrazione è rappresentata dalle variabili condivise (*shared variables*), utilizzate nelle computazioni parallele. Di default Spark tiene traccia delle variabili istanziate nei vari task, e consente se necessario di condividerle fra task o fra task e driver. Le variabili condivise possono essere di due tipi: di tipo *broadcast*, il cui valore viene salvato nella cache per ogni nodo, e di tipo *accumulatore*, per esempio contatori o sommatori [5].

4.1.1 Spark Streaming

Spark Streaming è un'estensione delle Core API di Spark per lo **stream processing** di live data streams ad alto throughput. Supporta molteplici sorgenti di data stream come **Kafka**, Flume, Twitter, ZeroMQ, Kinesis o socket TCP, i quali possono essere processati tramite direttive come *map*, *join*, *reduce* e *window*. Nel post processing è possibile salvare i data stream in un file system, in un database o visualizzarli in una live dashboard. Come ulteriore fase nella pipeline di operazione rientra anche il machine learning ed il graph processing. In Figura 10 viene riassunta l'architettura descritta [5].

Figura 10: Architettura di Spark Streaming [5]

Nello specifico, i data streams ricevuti vengono suddivisi in frammenti (*batches*), processati da Spark per generare lo stream finale risultante in batches, come mostrato in Figura 11 [5].

Figura 11: Spark Streaming Data Stream Processing [5]

A livello alto il flusso continuo di dati è rappresento da una struttura astratta detta *discretized stream* o *DStream*, il cui contenuto è rappresentato da tutte le sorgenti collegate eventualmente con Spark, o da stream risultati da altri DStream. Internamente, un DStream è rappresentato tramite una sequenza di RDD [5].

4.1.2 Spark MLlib

Spark MLlib è la libreria per il machine learning di Spark, ed il suo obiettivo è di rendere l'uso di tali funzionalità semplice e scalabile. Comprende i più comuni algoritmi di learning quali classificazione, regressione, clustering, filtro collaborativo, riduzione dello spazio delle features, etc. [5].

4.1.3 Spark SQL

Spark SQL è un modulo di Spark per il processing di dati strutturati. A differenza delle API RDD la sua interfaccia consente di descrivere in maniera dettagliata la struttura e le operazioni da eseguire sui dati, in stile SQL. Le strutture astratte di riferimento per questo modulo sono i **Dataframes** e i **Datasets** [5].

Un **Dataframe** è un collezione distribuita di informazioni organizzata in colonne e attributi. Concettualmente è equivalente ad una tabella in un database relazionale, con caratteristiche aggiuntive fornite da Spark. In accordo con tale definizione sono presenti, oltre alle classiche funzionalità SQL, la creazione di DataFrame a partire da un RDD o un oggetto JSON e viceversa [5].

Un **Dataset** invece è un'interfaccia sperimentale introdotta nella versione 1.6 di Spark, mirata all'integrazione delle API RDD con l'engine SQL. Attualmente il supporto è limitato alle API Java e Scala [5].

4.2 Apache Kafka

Apache Kafka è un sistema di messaggistica di tipo publish-subscribe, orientato alla distribuzione. La sua architettura consente ad un singolo cluster di agire da backbone centrale per i dati di grandi organizzazioni, e gli stream vengono partizionati e distribuiti lungo tutti i nodi del cluster, sfruttando la potenza di calcolo di ogni singola macchina. Di seguito si introduce la terminologia utilizzata in Kafka:

- Kafka organizza i flussi dei messaggi in categorie chiamate *topics*;
- I processi che si occupano di pubblicare i messaggi in Kafka sono chiamati *producers*;
- I processi che effettuano una sottoscrizione ad un topic ed elaborano i messaggi pubblicati sullo stesso sono chiamati *consumers*;
- I nodi all'interno del cluster(producers e consumers) sono coordinati da uno o più server chiamati *brokers* [4].

A livello concettuale il funzionamento di Kafka è riassunto in Figura 12. La comunicazione tra client e server avviene tramite protocollo TCP. Sono presenti varie implementazioni del client Kafka, disponibili in vari linguaggi tra cui Java, Javascript e PHP [4].

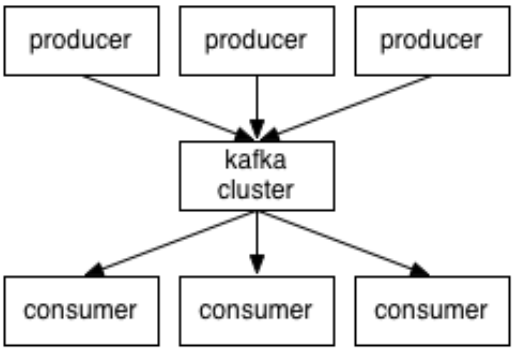

Figura 12: Architettura di Apache Kafka [4]

Di seguito viene mostrata nel dettaglio la struttura di un topic in Figura 13. Per ogni topic Kafka effettua un partizionamento dei messaggi in arrivo, tenendo traccia dell'ordine di arrivo con un sistema di log [4].

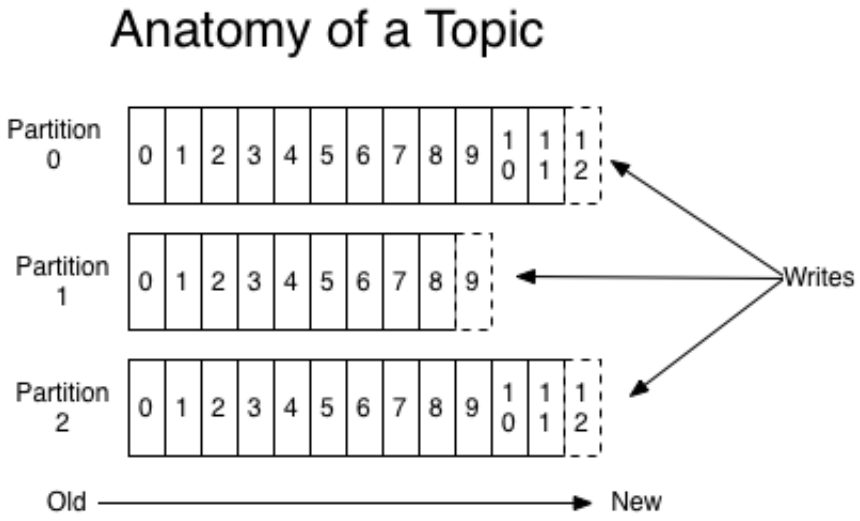

Figura 13: Struttura di un topic in Kafka [4]

Ogni partizione è una sequenza immutabile di messaggi, registrata e ordinata in un commit log. Ad ogni messaggio presente viene assegnato un id sequenziale, detto *offset*, che identifica univocamente un messaggio nella relativa partizione [4].

Il cluster di Kafka conserva in memoria tutti i messaggi pubblicati, letti o non dai consumers, per un periodo di tempo configurabile. In pratica, il server registra dei metadati relativi ad ogni posizione del consumer nel log, chiamato appunto offset. Tale offset è fissato dal consumer, il quale può leggere i messaggi nell'ordine preferito [4].

4.2.1 Integrazione con Spark Streaming

Le API di Spark Streaming offrono la possibilità di configurare Kafka come sorgente di data stream, attraverso due possibili approcci [5].

Il primo approccio è detto **Receiver-based**, e consiste nell'implementazione di un Receiver tramite le API di Kafka. I dati ricevuti attraverso tale oggetto vengono memorizzati e processati

nei worker node. Tuttavia, tale approccio potrebbe non garantire la consistenza in caso di data loss, ed è dunque necessario abilitare un meccanismo di integrità chiamato *Ahead Logs* [5].

Il secondo approccio, detto *Direct Approch* o *Receivers less*, non prevede l'uso di Receivers ma interroga periodicamente il server Kafka per rilevare il topic e la partizione aggiornata di recente, definendo in automatico la dimensione del batch di dati da processare. I vantaggi di questa tecnica sono:

- *Parallelismo semplificato:* Non è necessario creare ed unificare stream multipli di Kafka. Tramite la funzione *directStream*, Spark Streaming effettua un mapping tra le partizioni di Kafka in RDD, semplificandone l'elaborazione in parallelo;
- *Efficienza:* Il meccanismo Write Ahead Logs potrebbe generare problemi di duplicazione dei dati e relativi conflitti, nel momento in cui entrambe le istanze di Spark e Kafka tentino di risolvere una perdita di informazioni. Con un approccio diretto senza Receivers tale problema non sussiste [5].

4.3 Il framework Node.js

Node.js è un framework per la realizzazione di applicazioni web in Javascript lato server. La piattaforma è basata sul **Javascript Engine V8**, il runtime di Google utilizzato anche in Chrome, disponibile per le principali piattaforme, anche se maggiormente performante su sistemi operativi UNIX-like.

A differenza dei classici web server, che sfruttano un modello a processi concorrenti, Node.js utilizza una modalità di tipo *event-driven* per l'accesso alle risorse. Scrivere un'applicazione in Node.js dunque consiste nel definire una serie di eventi che il sistema operativo deve monitorare, e all'attivazione degli stessi eseguire una funzione associata a tale evento, detta **callback**. Tale approccio definisce processi non bloccanti, evita il verificarsi di deadlocks, e in generale le operazioni di I/O non avvengono in maniera diretta. Ciò garantisce una maggiore **scalabilità** e **flessibilità** dell'architettura web.

All'interno del framework si ha a disposizione il package manager **npm**, tramite il quale è possibile installare moduli e librerie, gestire le diverse release, gli update e le dipendenze.

Il modulo **kafka.node** (https://www.npmjs.com/package/kafka-node) mette a disposizione un client in Node.js per la connessione ad un server Kafka. È possibile implementare sia il Producer che il Consumer, gestire gli eventi, creare topics, etc.

4.3.1 Angular.js

L'interfaccia dell'applicazione è stata realizzata attraverso il framework AngularJS, creando un modello MVC che sfrutta il two-way data binding di AngularJS. Il front end è costruito grazie alla libreria Bootstrap, un framework che mira alla creazione di pagine web dinamiche, fluide e adattabili a qualsiasi dispositivo con differenti grandezze e risoluzione, che contiene una serie di modelli CSS, HTML e Javascript dei componenti web maggiormente utilizzati. Altre librerie utilizzate sono:

- **Angular Route** Una libreria che gestisce le view ed il routing in una macchina a stati.

Il layout contiene le seguenti directory:

- **/css** Fogli di stile per le pagine HTML;
- **/fonts** Web font;
- **/img** Logo e immagini;
- **/vendor** Librerie utilizzate dall'applicazione;
- **/js** File Javascript della dashboard;
  - **/controllers** Script dei controller di AngularJS;
  - **views** File HTML relativi alle view di AngularJS;
  - **/services** Direttive di AngularJS per gli eventi associati al websocket.
- **app.js** Script principale dell'applicazione. Al suo interno sono definiti gli stati con le relative view, le transizioni e le direttive associate ad ogni view per i relativi dati da visualizzare.

4.4 La libreria socket.IO

5 Conclusioni e sviluppi futuri

Riferimenti bibliografici

[1] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '02, pages 1–16, New York, NY, USA, 2002. ACM.

[2] J. Bobadilla, F. Ortega, A. Hernando, and A. GutiéRrez. Recommender systems survey. *Know.-Based Syst.*, 46:109–132, July 2013.

[3] Gianpaolo Cugola and Alessandro Margara. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.*, 44(3):15:1–15:62, June 2012.

[4] Apache™ Software Foundation. Kafka Official 0.9.0 Documentation. http://kafka.apache.org/documentation.html, 2016.

[5] Apache™ Software Foundation. Spark Official 1.6.1 Documentation. http://spark.apache.org/docs/latest/, 2016.

[6] Lajos Jeno Fülöp, Gabriella Tóth, Róbert Rácz, János Pánczél, Tamás Gergely, Arpád Beszédes, and Lóránt Farkas. Survey on complex event processing and predictive analytics. *Nokia Siemens Networks*, 2010.

[7] Patrick Gantet Patrick Gantet. Complex event processing.

[8] Xiaoyuan Su and Taghi M. Khoshgoftaar. A survey of collaborative filtering techniques. *Adv. in Artif. Intell.*, 2009:4:2–4:2, January 2009.

[9] Gábor Takács, István Pilászy, Bottyán Németh, and Domonkos Tikk. Matrix factorization and neighbor based algorithms for the netflix prize problem. In *Proceedings of the 2008 ACM Conference on Recommender Systems*, RecSys '08, pages 267–274, New York, NY, USA, 2008. ACM.

6 Sorgenti

A Sorgenti Spark

Listato 1: Main.scala

```
package ml

import cep.CEP
import commons.{FileParser, KafkaCommons, SparkCommons}
import org.apache.log4j.{Level, Logger}

/**
*
* @authors Mauro Losciale and Pietro Tedeschi.
*/
object Main {

def main(args: Array[String]): Unit ={

Logger.getLogger("org.apache.spark").setLevel(Level.ERROR)
Logger.getLogger("org.eclipse.jetty.server").setLevel(Level.OFF)

val ratings = FileParser.parseRatings()
val movies = FileParser.parseMovies()

val numRatings = ratings.count()
val numUsers = ratings.map(_._2.user).distinct().count()
val numMovies = ratings.map(_._2.product).distinct().count()

println("Got " + numRatings + " ratings from "
+ numUsers + " users on " + numMovies + " movies.")

val messages = KafkaCommons.kafkaDirectStreaming()

messages.foreachRDD { rdd =>
val votes = rdd.map(_._2)
val tableVotes = SparkCommons.sqlContext.read.option("header","true")
   .option("inferSchema","true")
   .json(votes).toDF().na.drop()

if (tableVotes.count() > 0) {

tableVotes.show()
tableVotes.registerTempTable("tableVotes")

val query = SparkCommons.sqlContext.sql("SELECT * FROM tableVotes WHERE rating > 3")
CEP.hasEnoughVotes(query, tableVotes, ratings, movies)

} else println("Nessun voto ricevuto")


}

SparkCommons.ssc.start()
```

```
SparkCommons.ssc.awaitTermination()

}

}
```

Listato 2: SparkCommons.scala

```
package commons

import org.apache.spark.sql.SQLContext
import org.apache.spark.streaming._
import org.apache.spark.{SparkConf, SparkContext}

/**
* Handles configuration, context and so
*
* @authors Mauro Losciale and Pietro Tedeschi.
*/
object SparkCommons {
//build the SparkConf object at once
lazy val conf = {
new SparkConf(false)
.setMaster("local[*]")
.setAppName("Collaborative Filtering with Kafka")
.set("spark.logConf", "true")
}

lazy val sc = SparkContext.getOrCreate(conf)
lazy val ssc = new StreamingContext(sc, Seconds(30))
lazy val sqlContext = new SQLContext(sc)

sc.addJar("target/scala-2.10/mlspark-assembly-1.0.jar")
}
```

Listato 3: KafkaCommons.scala

```
package commons

import java.util.Properties

import kafka.producer.{Producer, ProducerConfig}
import kafka.serializer.StringDecoder
import org.apache.spark.streaming.Seconds
import org.apache.spark.streaming.kafka.{KafkaUtils, OffsetRange}

/**
*
* @authors Mauro Losciale and Pietro Tedeschi.
*/
object KafkaCommons {

val kafkaTopics = "votes" // command separated list of topics
val kafkaBrokers = "makaveli-desktop:9092"
var offsetRanges = Array[OffsetRange]()
```

```
val kafkaParams = Map[String, String]("metadata.broker.list" -> kafkaBrokers
)
val topicsSet = kafkaTopics.split(",").toSet
val messages = KafkaUtils.createDirectStream[String, String, StringDecoder,
    StringDecoder](
SparkCommons.ssc, kafkaParams, topicsSet)

val props:Properties = new Properties()
props.put("metadata.broker.list", "192.168.0.4:9092")
props.put("serializer.class", "kafka.serializer.StringEncoder")

val config = new ProducerConfig(props)
val producer = new Producer[String, String](config)

def kafkaDirectStreaming() = {

KafkaUtils.createDirectStream[String, String, StringDecoder, StringDecoder](
SparkCommons.ssc, kafkaParams, topicsSet).window(Seconds(60), Seconds(30))
}

}
```

Listato 4: CollaborativeFiltering.scala

```
package ml

import commons.{KafkaCommons, SparkCommons}
import kafka.producer.{KeyedMessage, Producer}
import org.apache.spark.mllib.recommendation.{ALS, MatrixFactorizationModel,
    Rating}
import org.apache.spark.rdd._
import org.json4s.JsonDSL._
import org.json4s.jackson.JsonMethods._

/**
*
* @authors Mauro Losciale and Pietro Tedeschi.
*/
object CollaborativeFiltering {

def userCF(rdd: RDD[Rating], ratings: RDD[(Long, Rating)], movies: Map[Int,
    String]) = {

val myRatingsRDD = rdd

// split ratings into train (60%), validation (20%), and test (20%) based on the
// last digit of the timestamp, add myRatings to train, and cache them

val numPartitions = 4

val training = ratings.filter(x => x._1 < 6)
.values
.union(myRatingsRDD)
.repartition(numPartitions)
.cache()
```

```
val validation = ratings.filter(x => x._1 >= 6 && x._1 < 8)
.values
.repartition(numPartitions)
.cache()

val test = ratings.filter(x => x._1 >= 8).values.cache()

val numTraining = training.count()
val numValidation = validation.count()
val numTest = test.count()

println("Training: " + numTraining + ", validation: " + numValidation + ", test:
    " + numTest)

// train models and evaluate them on the validation set

val ranks = List(8, 12)
val lambdas = List(0.1, 10.0)
val numIters = List(10, 20)
var bestModel: Option[MatrixFactorizationModel] = None
var bestValidationRmse = Double.MaxValue
var bestRank = 0
var bestLambda = -1.0
var bestNumIter = -1
for (rank <- ranks; lambda <- lambdas; numIter <- numIters) {
val model = ALS.train(training, rank, numIter, lambda)
val validationRmse = computeRmse(model, validation, numValidation)
println("RMSE (validation) = " + validationRmse + " for the model trained with
    rank = "
+ rank + ", lambda = " + lambda + ", and numIter = " + numIter + ".")
if (validationRmse < bestValidationRmse) {
bestModel = Some(model)
bestValidationRmse = validationRmse
bestRank = rank
bestLambda = lambda
bestNumIter = numIter
}
}

// evaluate the best model on the test set

val testRmse = computeRmse(bestModel.get, test, numTest)

println("The best model was trained with rank = " + bestRank + " and lambda = "
    + bestLambda
+ ", and numIter = " + bestNumIter + ", and its RMSE on the test set is " +
    testRmse + ".")

// make personalized recommendations

val myRatedMovieIds = myRatingsRDD.map(_.product).collect().toSet
val candidates = SparkCommons.sc.parallelize(
   movies.keys.filter(!myRatedMovieIds.contains(_)).toSeq)
val recommendations = bestModel.get
.predict(candidates.map((0, _)))
```

```
    .collect()
    .sortBy(-_.rating)
    .take(9)

    out(recommendations, movies, KafkaCommons.producer)

    }

    /** Compute RMSE (Root Mean Squared Error). */
    def computeRmse(model: MatrixFactorizationModel, data: RDD[Rating], n: Long):
        Double = {
    val predictions: RDD[Rating] = model.predict(data.map(x => (x.user, x.product)))
    val predictionsAndRatings = predictions.map(x => ((x.user, x.product), x.rating))
    .join(data.map(x => ((x.user, x.product), x.rating)))
    .values
    math.sqrt(predictionsAndRatings.map(x => (x._1 - x._2) * (x._1 - x._2)).reduce(_
        + _) / n)
    }

    /** Write recommendations array to a JSON file **/
    def out(data: Array[Rating], film: Map[Int, String], producer: Producer[String,
        String]) = {

    println("Movies recommended for you:")

    data.foreach { r =>

    val jsonVote = ("movieid", r.product) ~("rating", r.rating.toInt) ~("title",
        film(r.product))
    producer.send(new KeyedMessage[String, String]("result",
        compact(render(jsonVote)).toString))
    }
    }
    }
```

Listato 5: CEP.scala

```
    package cep

    import ml.CollaborativeFiltering
    import org.apache.spark.mllib.recommendation.Rating
    import org.apache.spark.rdd.RDD
    import org.apache.spark.sql.DataFrame

    /**
    *
    * @authors Mauro Losciale and Pietro Tedeschi.
    */
    object CEP {

    def hasEnoughVotes(query: DataFrame, table: DataFrame, ratings:
        RDD[(Long,Rating)], movies: Map[Int, String]) = {

    query.count() match {
```

```
      case c if c == 0 => println("Nessun voto valido")
      case c if c > 3 =>
         val votesRating = table.map(v => Rating(0, v(0).toString.toInt,
             v(1).toString.toDouble))
         println("Make CF")
         CollaborativeFiltering.userCF(votesRating, ratings, movies)
      case c if c <= 3 => println("Voti validi inferiori a 4")
         }
      }

   }
```

B Sorgenti Node.js

Listato 6: app.js

```
var vote_server = require('./lib/vote_server');
var kafka_consumer = require('./lib/kafka_consumer');
var kafka_producer = require('./lib/kafka_producer');
var http = require('http');
var express = require('express');
var path = require('path');
var argv = require('minimist')(process.argv.slice(2));

var favicon = require('serve-favicon');
var logger = require('morgan');
var methodOverride = require('method-override');
var bodyParser = require('body-parser');
var multer = require('multer');
var errorHandler = require('errorhandler');

var app = express();

// all environments
app.set('port', process.env.PORT || 3000);
app.use(favicon(__dirname + '/public/favicon.ico'));
app.use(logger('dev'));
app.use(methodOverride());
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));
app.use(multer());
app.use(express.static(path.join(__dirname, 'public')));

app.get('/', function (req, res) {
res.sendfile(__dirname + '/public/index.html');
});

if ('development' == app.get('env')) {
app.use(errorHandler());
}

var server = http.createServer(app);
var port = (argv.port) ? argv.port : 3000;

server.listen(port);
voteServer = vote_server(server);
```

```
var groupId = (argv.groupid) ? argv.groupid : 'vote-server-group';
kafkaConsumer = kafka_consumer(voteServer, groupId);
kafkaProducer = kafka_producer(voteServer);

if (process.platform === "win32") {
var rl = require("readline").createInterface({
input: process.stdin,
output: process.stdout
});

rl.on("SIGINT", function () {
process.emit("SIGINT");
});
}

process.on("SIGINT", function () {
console.log('Shutting down');
kafkaConsumer.close();
kafkaProducer.close();
process.exit();
});
```