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

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**.

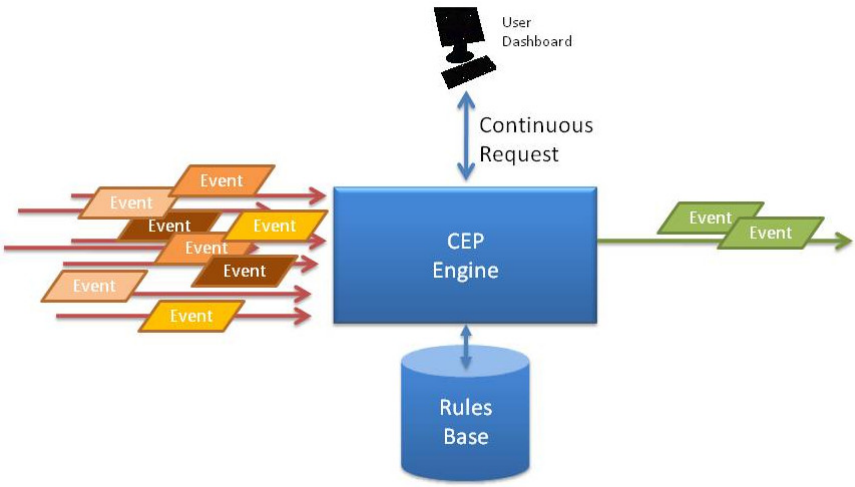

Figura 1: Architettura di un Sistema CEP [5]

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.

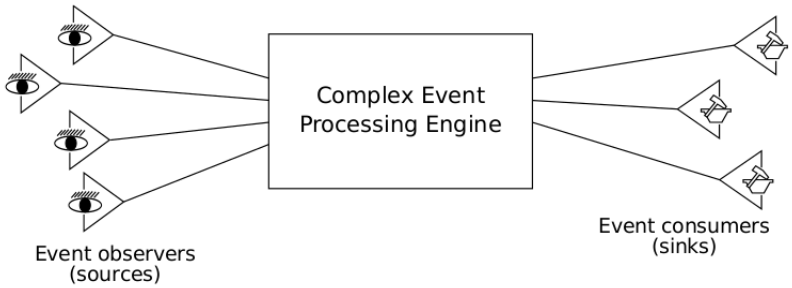

Figura 2: Funzionamento di un Sistema CEP (High-Level) [1]

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** [1,4].

2.2 Sistemi di Raccomandazione

2.2.1 Filtro collaborativo

2.3 Introduzione allo Stream Processing

2.3.1 Il paradigma Publish-Subscribe

2.4 Il pattern Facade

2.5 Il pattern Singleton

2.6 Il pattern Model-View-Controller (MVC)

2.7 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) [3].

Una delle possibili configurazioni di un sistema Spark è la modalità *cluster*, mostrata in Figura 3.

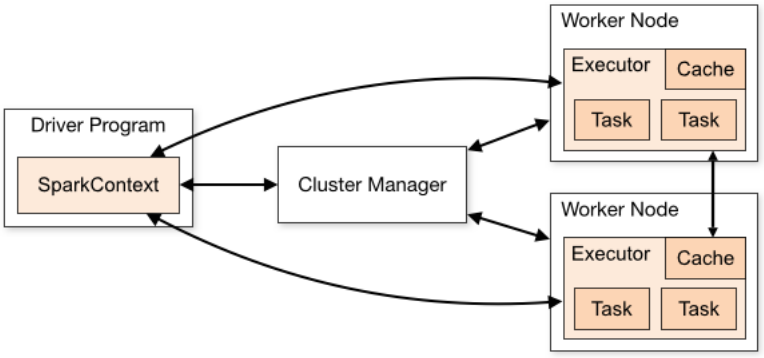

Figura 3: Configurazione in Spark di tipo Cluster Mode [3]

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 [3].

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 [3].

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 [3].

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 [3].

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 4 viene riassunta l'architettura descritta [3].

Figura 4: Architettura di Spark Streaming [3]

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 5 [3].

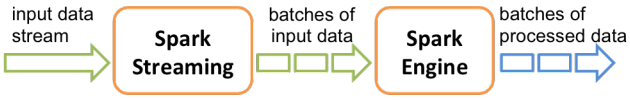

Figura 5: Spark Streaming Data Stream Processing [3]

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 [3].

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. [3].

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** [3].

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 [3].

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 [3].

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* [2].

A livello concettuale il funzionamento di Kafka è riassunto in Figura 6. 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 [2].

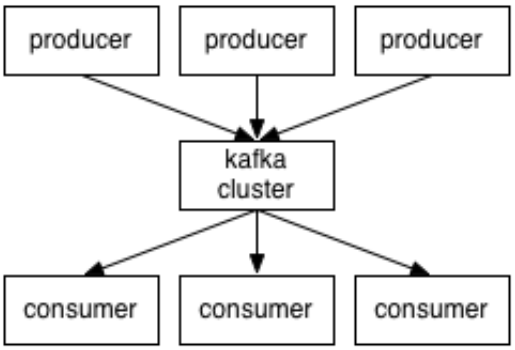

Figura 6: Architettura di Apache Kafka [2]

Di seguito viene mostrata nel dettaglio la struttura di un topic in Figura 7. Per ogni topic Kafka effettua un partizionamento dei messaggi in arrivo, tenendo traccia dell'ordine di arrivo con un sistema di log [2].

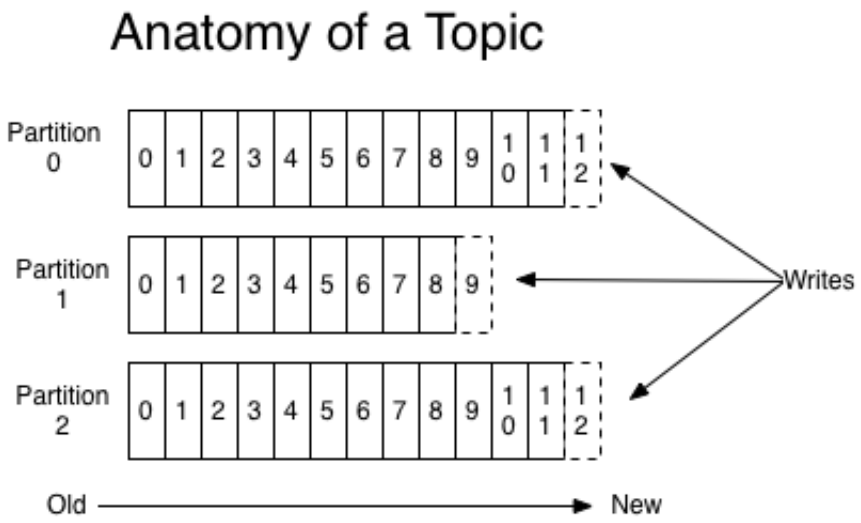

Figura 7: Struttura di un topic in Kafka [2]

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 [2].

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 [2].

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 [3].

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* [3].

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 [3].

Riferimenti bibliografici

[1] 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.

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

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

[4] 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.

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

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