POLITECNICO
MILANO 1863

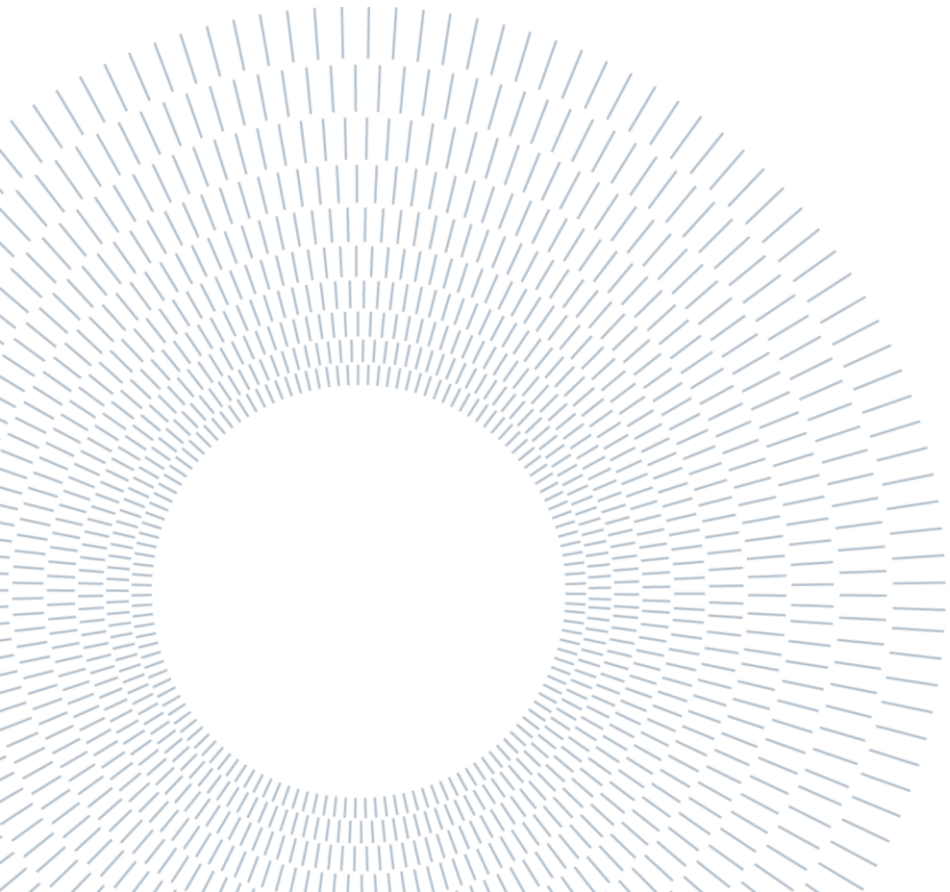
SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Real Time Recommendations with Clickstream Data

TESI DI LAUREA MAGISTRALE IN
COMPUTER SCIENCE AND ENGINEERING -
INGEGNERIA INFORMATICA

Author: **Alessandro Messori**

Student ID:	962606
Advisor:	Emanuele Della Valle
Academic Year:	2021-2022



Abstract

One of the most popular research threads and challenges of modern machine learning scientists and engineers is how to build models that can update themselves in real time as soon as new data is generated, without the need for periodically retraining the system from scratch. Streaming machine learning is the term used to describe models that work with real time data streams instead of traditional batch processes.
Recommender Systems greatly benefit from the use of stream processing, for many reasons: real time machine learning offers a very low latency in the update of recommendations, which can be extremely useful in highly dynamic context such as media consumption or e-commerce where users may change or update their tastes very frequently or perform sessions with a specific navigation target in mind. Thanks to real time recommendations, systems can immediately adapt to the sudden change in behavior and tastes of users and offer them exactly what they need as soon as they express a new interest.
This thesis has the objective of experimenting with different techniques and architectures for building real time recommender systems. This work focuses on a matrix factorization implementation of the popular collaborative filtering approach and proposes a way to integrate it with data stream inputs of user navigation events, called clickstreams, containing implicit item ratings. A model architecture and algorithm implementation will be proposed and explained, by using the distributed processing engine Spark and the message broker Kafka.
The proposed algorithms focus on working with many different categories of user to item interactions and integrate them in a single model, while also proposing a novel way of extracting common user behavioral patterns from clickstream sequences and exploiting them as a form of implicit item rating.
The potential and capabilities of the proposed approaches have then been proven by various experiments conducted in a local setting over multiple clickstream datasets.

Key-words: Stream Processing, Spark, Kafka, Collaborative Filtering, Incremental Learning

Abstract in lingua italiana

Uno dei campi di ricerca più diffusi e maggiore sfida dei moderni scienziati e ingegneri del machine learning è quelli di costruire modelli in grado di aggiornarsi in tempo reale non appena vengono generati nuovi dati, senza la necessità di allenare periodicamente il sistema da zero. Streaming machine learning è il termine utilizzato per descrivere i modelli che funzionano con stream di dati in tempo reale.

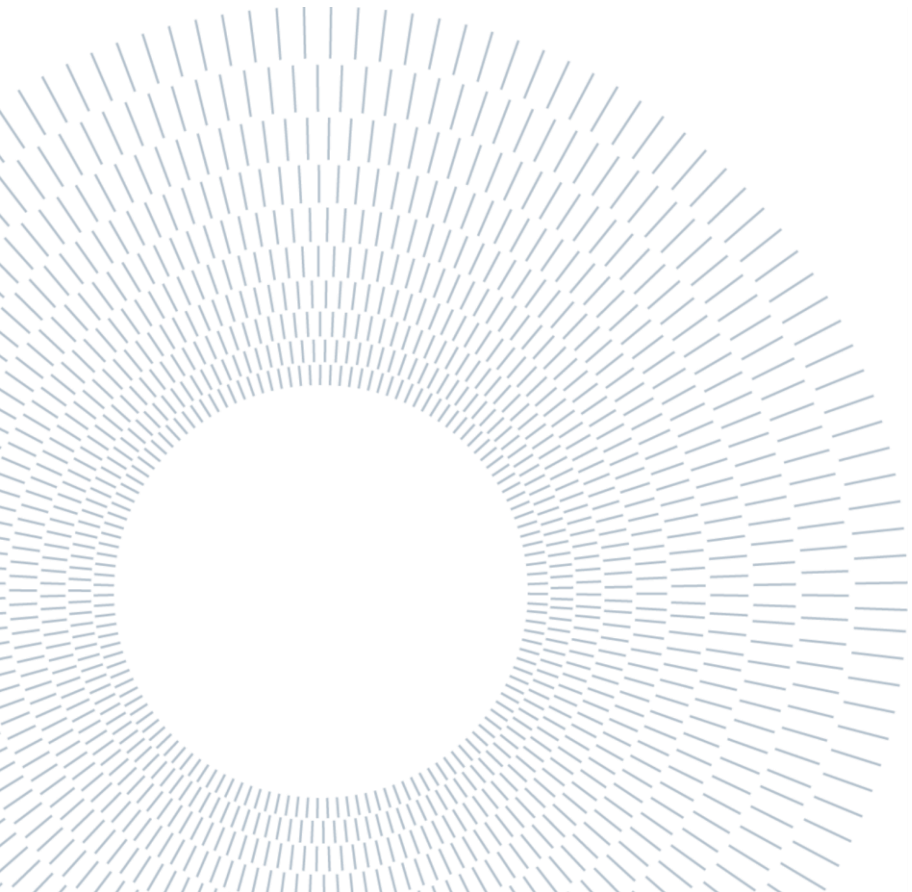
I sistemi di raccomandazione, in particolare, traggono grandi vantaggi dall'uso dell'elaborazione degli stream, per molte ragioni: il machine learning in tempo reale offre una latenza molto bassa nell'aggiornamento delle raccomandazioni, che può essere estremamente utile in contesti altamente dinamici come il consumo di media o l'e-commerce dove gli utenti possono cambiare i propri gusti molto frequentemente o eseguire sessioni con uno specifico target di navigazione in mente. Grazie alle raccomandazioni in tempo reale, i sistemi possono adattarsi immediatamente al cambiamento improvviso di comportamento e gusti degli utenti e offrire loro esattamente ciò di cui hanno bisogno non appena manifestano un nuovo interesse.

Questa tesi ha l'obiettivo di sperimentare diverse tecniche per la costruzione di sistemi di raccomandazione in tempo reale. Questo lavoro si concentra su un'implementazione dell'approccio collaborative filtering con fattorizzazione di matrice e propone un modo per integrarlo con gli input di stream di dati degli eventi di navigazione dell'utente, chiamati clickstream. Verrà proposta e spiegata un'architettura di modello e l'implementazione di un algoritmo, utilizzando Spark per l' elaborazione distribuita e Kafka come broker di messaggi.

Gli algoritmi proposti funzionano con diverse categorie di interazioni utente-elemento e le integrano in un unico modello, proponendo anche un nuovo modo di estrarre pattern comportamentali degli utenti dalle sequenze di clickstream e sfruttarli come una forma di valutazione implicita degli elementi.

Il potenziale e le capacità degli approcci proposti sono stati poi dimostrati da vari esperimenti condotti in un contesto locale su più set di dati clickstream.

Parole chiave: Stream Processing, Spark, Kafka, Collaborative Filtering, Incremental Learning



Contents

1. Problem Settings

1.1 Problem Definition

Traditional Recommendation algorithms work in an offline and non-distributed setting, meaning that all the processing is executed on a single machine and that the model is executed periodically.

This kind of approach works very well for use cases that don't require big amounts of data to process or in cases in which user interest doesn't evolve too frequently.

However, as applications and systems grow in size, there comes a point where one single machine can't hold all the data in memory anymore and doesn't have enough processing power to train the machine learning model in a reasonable time.

To overcome these limitations, machine learning models can be trained and deployed in distributed systems, that split the storage and processing of the big amounts of data needed for training.

Writing distributed counterpart of any algorithm can be quite difficult, it's necessary to consider how to distribute data evenly between all the nodes of the system, while keeping the amount of data shuffling to a minimum.

Apache Spark is currently the most popular and widespread project offering libraries enabling parallel computing and offering distributed implementation of the most popular machine learning algorithms.

The second big limitation of classical ml algorithms is that they typically work in an offline setting, this means that they are trained on a fixed dataset that was gathered at a certain point in time (usually extracted by a database), but in order to take into account the new data that is being continuously generated they need to be trained again from scratch, which introduces a lot of delay in recommendations context when users change their interests very quickly.

It is becoming more and more apparent the huge impact of real time prediction in data heavy apps and websites [14].

For example, being able to learn about the user's interest while he is navigating an app or website and update the next recommended item while the user is browsing content can have a huge impact on the success of a product.

Many prominent analysts attribute the recent Tik Tok success to their excellent real time recommendation algorithm [2], who keeps users engaged by suggesting videos who will be next in the viewing queue of the spectator.

Implementing a machine learning algorithm that works with data in real time (called streaming data) comes with many more challenges and hardships with respect to their offline version, and the streaming machine learning landscape is lagging a lot [15, 26].

Apache Spark for example, as of this moment only offers a working streaming implementation of a fraction of the models they have available offline.

Netflix has been famously struggling with implementing a real time version of their recommender engine and is still looking for an optimal way to discover what would be the best content by analyzing its real time behavior on their website.

This thesis proposes and explores a possible implementation of a distributed and real time version of famous algorithm for user-item recommendations (Collaborative Filtering with Matrix Factorization) and experiments with new ways of handling the input signal typically used in clickstream data analysis for recommendation.

Chapter 1 goes over the state of the art for all the main technologies and techniques used in the experiments presented in this thesis.

Chapter 2 analyzes the technological choices and libraries used for the experiments presented in this document, by comparing different tools and highlighting the tradeoff that came with each alternative.

Chapter 3 presents our approaches to tackling the problem defined in the first chapter and goes over the pseudocode of the developed algorithms.

Chapter 4 goes over the different experiments and dataset that were used to evaluate the approaches proposed in chapter 3.

Finally, chapter 5 comments on the results obtained in the previous chapter and talks about the limitations and possible future improvements of the work.

1.2 State of the Art

1.2.1 Distributed Computing

The basic idea behind distributed computing is simple: increase the time and space capabilities of a computer system by having the computations performed not by a single machine but by a cluster of nodes, by splitting the task into many subtasks that can be executed in parallel.

Every computing system that aspires to handle big amounts of data eventually needs to split the load of its traffic and divide processing into multiple machines (this technique is called horizontal scaling).

The paradigm that popularized the distributed model was MapReduce [28, 29] (alongside the Hadoop framework that offered all the tools to make it work on a cluster).

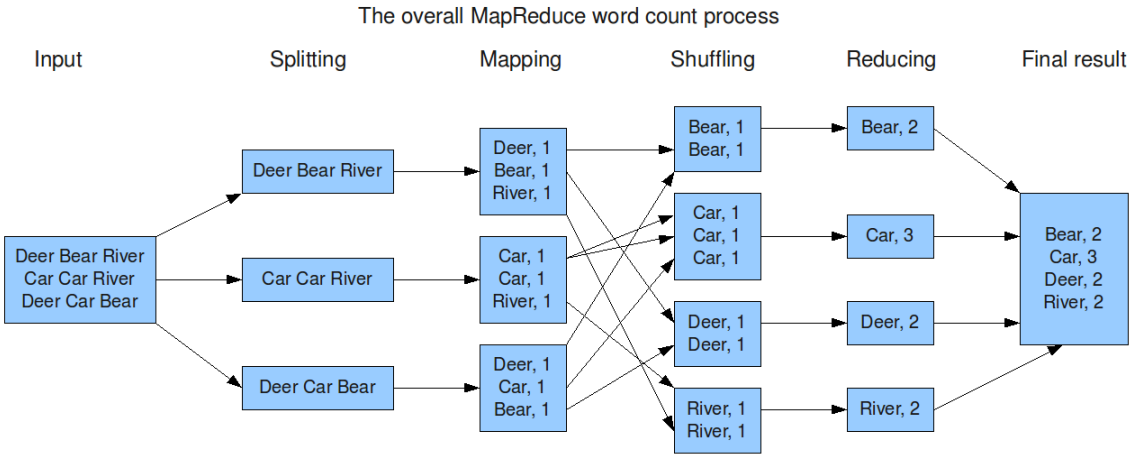


Figure 1.1: MapReduce Example [29]

MapReduce works by splitting the computation into steps that can be computed in parallel:

- Mapping, the input records get translated into a set of key-value pairs
- Shuffling, each pair generated in the mapping stage gets assigned and transmitted to a node in the cluster by following a partitioning policy (that should ideally keep the size of the partitions balanced)
- Reducing, all the data associated to the same key gets processed and transformed into a final output.

Hadoop and MapReduce in their original connotation have been replaced in the industry by more efficient and flexible distributed processing frameworks for a long time.

Apache Spark is one of these, it is based on the same concepts of Hadoop but offers 10x increases in performance thanks to its memory-based processing, compared to the disk-based system of Hadoop [37].

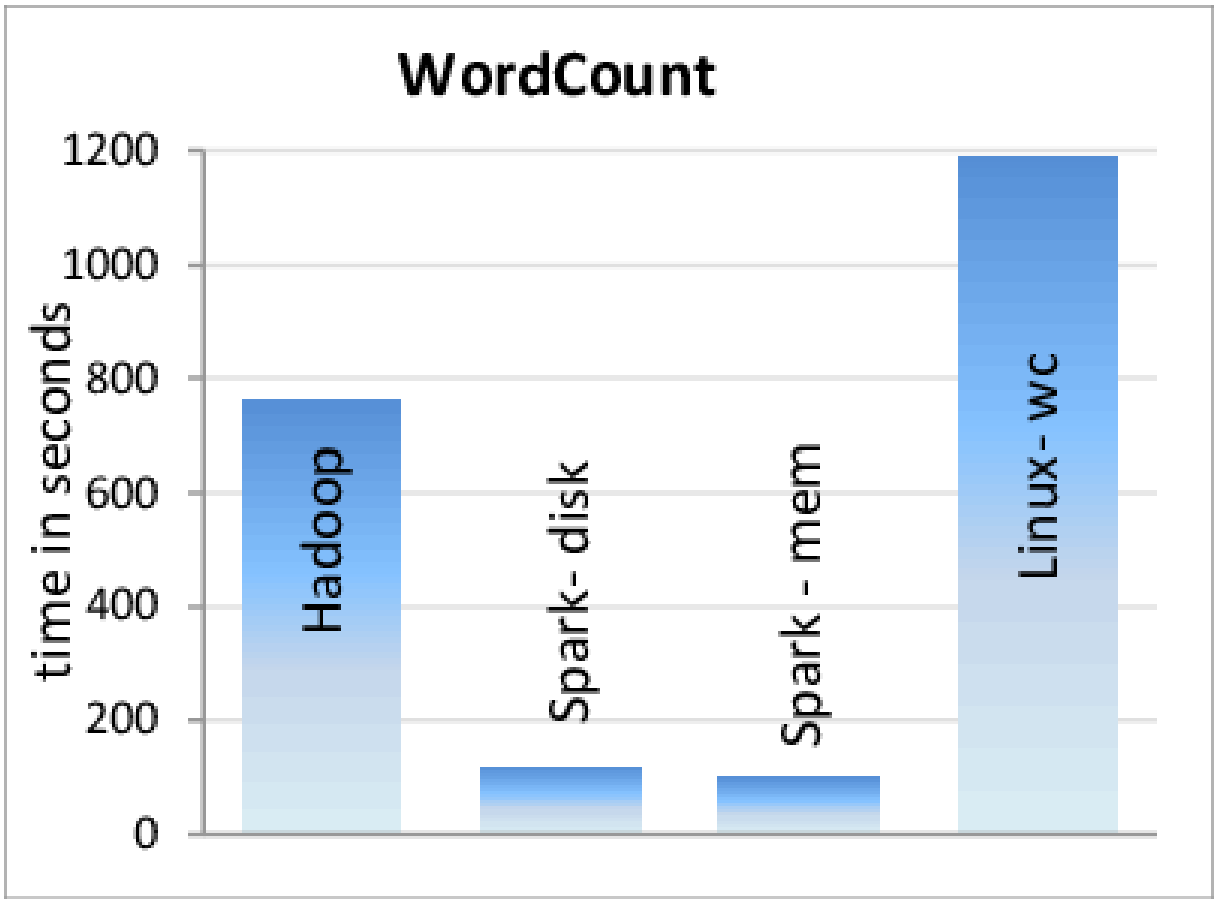


Figure 1.2: Hadoop vs Spark [37]

Spark offers a much more flexible and easier to use interface, based on the concept of RDDs (Resilient Distributed Datasets), immutable and distributed collections that can be treated in a similar way to an array.

RDDs are the basis for more complex data structures such as Block Matrixes and DataFrames, which are core aspects of Spark MlLib, a library which contains all the

necessary tools and models to implement machine learning at scale in a distributed setting.

Among the models offered by MlLib we have a distributed implementation of the Collaborative Filtering algorithm with ALS, which will be a great point of reference for our work.

1.2.2 Stream Processing

Stream processing allows for computation on data as soon as it's generated (or with minimal latencies), this allows to have a huge reduction in response time and can deliver a great impact on businesses that can benefit from delivering analysis as soon as possible to their customers.

Data streaming, even though it has been experiencing a big growth in popularity in recent years, remain somewhat of a niche field in the data analysis landscape, and the great majority of data scientists and engineers still uses offline/batch computations for all their needs [14].

This phenomenon happens mainly for 2 different reasons, first many still have the perception of online processing as a technology that is only needed for very specific tasks and don't fully realize the huge impact that very low latency analyses can have on a business.

The other reason is that writing streaming algorithms is generally much more complicated than writing offline ones. Streaming introduces in fact a lot of complexities and limitations into the computational model:

- Streaming data has unbound size, since it represents an ever-growing sequence of events
- Working with data coming in real time means that the system also needs to have a mechanism to handle record coming in late

Different processing engines tackle these challenges in different ways, Spark Structured Streaming for example adopts a microbatching approach, adjuvated by techniques like windowing, stateful processing and watermarking [32].

Windows are ways of grouping streaming data and perform computations on related data, the tree most used types of windows are:

- Tumbling Windows: fixed size, there is no overlapping among the events of each window
- Sliding Windows: fixed size, a new window is created each time a new event is generated

- Session Windows: dynamic size, a window is built around the concept of session

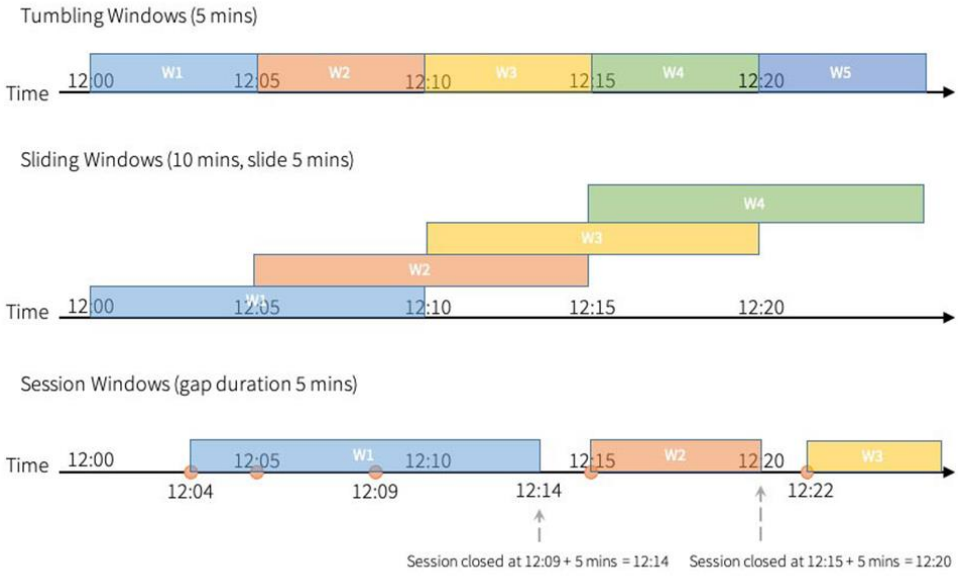


Figure 1.3: Windowing in Spark

Stream processing is typically used for computing small statistics or KPIs in real time, but it's also possible to use it for more complex analyses, such as training machine learning models and using them to make predictions.

As of right now there has been a lot of experimentation with streaming machine learning [12, 25], but online models represent only a tiny fraction of their offline counterpart: Apache Spark for example, allows to make predictions on streaming data with all their offline ml models, but only offers online training (which is the most effective form of real-time machine learning, and the one this thesis is focusing on) for a very small subset of them (currently k-means, linear regression and logistic regression).

One of the main objectives of this thesis is to propose and test an implementation of a streaming version of the collaborative filtering algorithm for recommendations with online training.

1.2.3 Collaborative Filtering

Collaborative filtering is probably the most popular and effective machine learning algorithm used for generating items to user recommendations in content-based applications [30, 34].

Unlike the content-based approaches, that work by analyzing the content of the item liked in the past by the users and finding similar products to recommend, collaborative filtering leverages only user past interactions and preferences and suggests items that were liked by other users who have had similar patterns in the past.

It works by storing the interactions of user to item inside an M by N rating matrix (where M represents the number of users and N the number of items) and the cell m,n contains the preferences of the user m for the item n.

The values inside the rating matrix are the user to item ratings, which can either be explicit or implicit [1, 19, 21].

Explicit recommendations have traditionally been used in collaborative filtering systems, and they are integer values (usually ranging from 1 to 5) that represent a preference a used has expressed over an item (websites often ask their users to leave these kinds of ratings after interacting with a certain item, for example reproducing a video on YouTube or purchasing a product on Amazon).

Explicit ratings guarantee a high degree of accuracy and reliability on the user preference, they are however often not enough for building an effective recommender system since they are not always available, and even if they are they usually on a smaller amount than their implicit counterpart (users typically interact with many more items than they leave a review for).

That's why the use of implicit rating has gained a lot of popularity in recent years; an implicit rating is derived from an interaction between a user and an item from which we can derive with some level of confidence how a user feels about an item.

Examples of implicit interactions could be the amount of time a user has spent watching a reel on TikTok, or how many times a customer has purchased a certain product on Amazon.

Figure 1.2: Rating Matrix Example in Collaborative Filtering [30]

The great majority of recommender systems only consider one single type of implicit interaction at the same time to estimate the ratings.

While there has been some research around it [9, 17, 18, 22], the use of multimodal inputs as implicit ratings is still for the most part an uncharted territory, and in this thesis, there will be an attempt of exploiting this kind of data to increase the recommender system accuracy.

1.2.4 Real time recommendations

As for any other machine learning models, recommender systems still work only with batch predictions in almost all their implementations, and for a good reason, since many websites/applications don't have this kind of low latency needs.

Usually, recommender systems get updated once a day, so users will start to receive recommendations with the updated preferences discovered during their navigation one day after displaying them, which is fine for most use cases.

There are also many situations however, in which user preferences gets updated very frequently, think for example about the eCommerce / Video sharing users that log-in to the website one day and starts looking for a new topic/product they have just hear about.

That's why in recent years a new thread of research was born, focusing on bringing user recommendations updated with the preferences they had displayed in that very same session [5, 16, 17, 18, 25].

Many real time recommendation systems are based on the concept of user similarity (often computed with cosine similarity) combined with clustering techniques such as k-nearest neighbors.

These kinds of approaches can offer good predictions (even though not as accurate as the one produced by collaborative filtering) and work well in an online setting given the limited computation time required to compute similarities.

More recent approaches leverage LHS blocking and Hamming Distance to compute in real time topic recommendations [3], these methods are quite flexible as they allow us to work with both explicit and implicit user preferences.

Finally, there are also proofs of real time recommender systems developed by Tencent based on item similarity and pair count and implicit collaborative filtering implemented in Apache Storm, that have the very interesting aspect of handling the processing of the data stream in a pipelined manner [38].

This thesis approach on the other hand will focus on an couple of key aspects that are often overlooked in the field, which are the sequence of user interactions inside a single website rather than a single user-item interaction, and an attempt to try to include many different kinds of actions into the implicit preference computation, instead of choosing a single kind of action as a proxy for explicit ratings (single-modal vs multi-modal approach).

The proposed collaborative filtering method is based on a lightweight variant of ALS matrix factorization, that can map our interaction sequences into preferences well.

2. Technology Choices

2.1 Stream Processing Engine

2.1.1 Alternatives

There is currently a plethora of possible choices when it comes to stream processing engines. The options were evaluated based on how well they could satisfy the project requirements, which are:

- High throughput distributed processing, with latency in the range of tens of seconds
- Low level algebraic distributed operations (mainly basic matrix operations, such as dot product, inverse and transpose)
- Support for running machine learning models at scale, both in a streaming and batch environment
- Some kind of stateful processing over streams

The three candidates that covered most of these aspects were Apache Spark, Apache Storm and Flink.

2.1.2 Apache Spark over Flink

Apache Spark and Flink have 2 fundamentally different execution models [32]:

The former simulates real time processing with a micro batching execution model while the latter offers an actual streaming engine.

This means that for some use cases, where the application requires millisecond levels of latency, Spark can't offer the same guarantees as Flink. On the other hand, Spark compensates with an increased throughput.

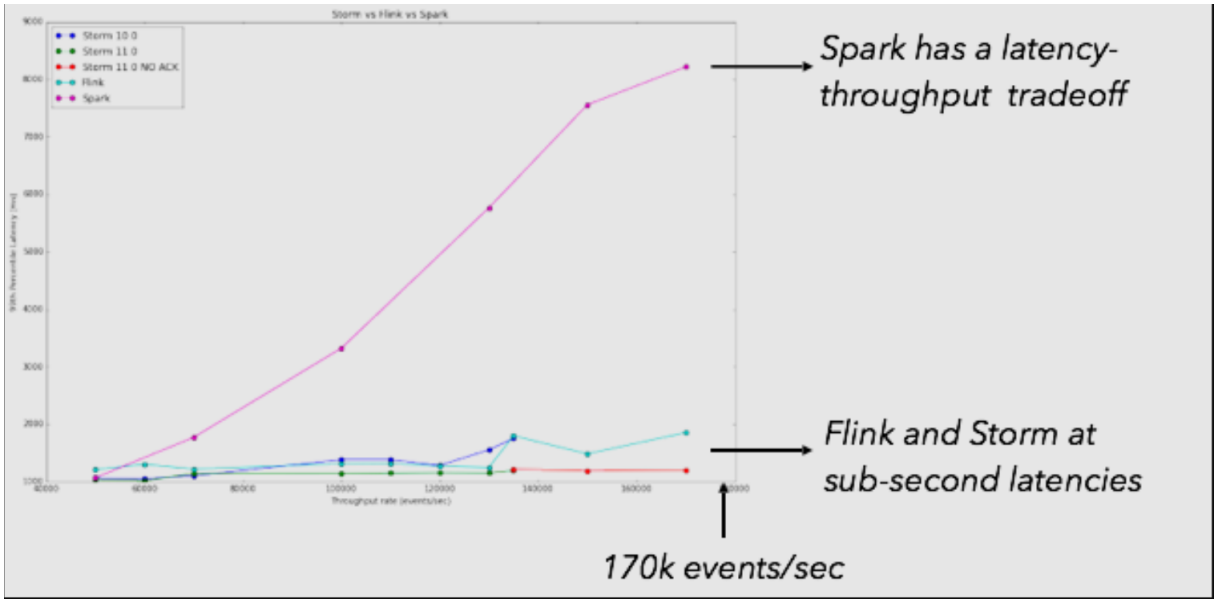


Figure 2.1: Processing Engines performance comparison [32]

While having such a millisecond level latency can seem very attractive for any real time use case, in our specific project the time scope of the recommendations is to the concept of user sessions, which on average operate on the scale of minutes (requirement that can be easily fulfilled by Spark Structured Streaming).

On the other hand, implementing a real time Matrix Factorization algorithm can be very demanding on the throughput side, so Spark has an edge over Flink on this aspect.

Moreover, both frameworks contain a machine learning library and toolset offering many of the most common models, but only Spark offers access to low level distributed matrixes, which are crucial for our implementation.

Due to all these reasons, Spark appeared as a better fit for the thesis needs.

2.1.3 Apache Spark over Apache Storm

The comparison among Spark and Storm is similar to the previous one with Flink [32]:

Storm too offers a real streaming processing model with milliseconds levels of latency at the cost of a reduced throughput and lacks in support with respect to distributed data structures for low level algebraic operations and high-level machine learning models.

Furthermore, Storm's community is less extended and established than Spark's, which, even though it doesn't really constitute a technical limitation, can make development more difficult.

Considering all the aforementioned points, Apache Spark was chosen as the most fitting tool for implementing our distribute, real time model.

2.2 Message Broker

To make our system more efficient and reliable, it's necessary to add a mediator component that will receive all the streaming data from our data sources, store it safely for a limited time, and then deliver each message exactly once to our stream processing cluster with the lowest possible latency.

The 2 technologies that were considered for this task were Apache Kafka and RabbitMQ, as they both satisfied the requirements.

While both these message brokers could provide good performances for this use case, the final choice landed on Kafka, that can offer higher horizontal scalability and guarantees a higher throughput for analytical applications such as this, while RabbitMQ performs better with low latency transactional systems.

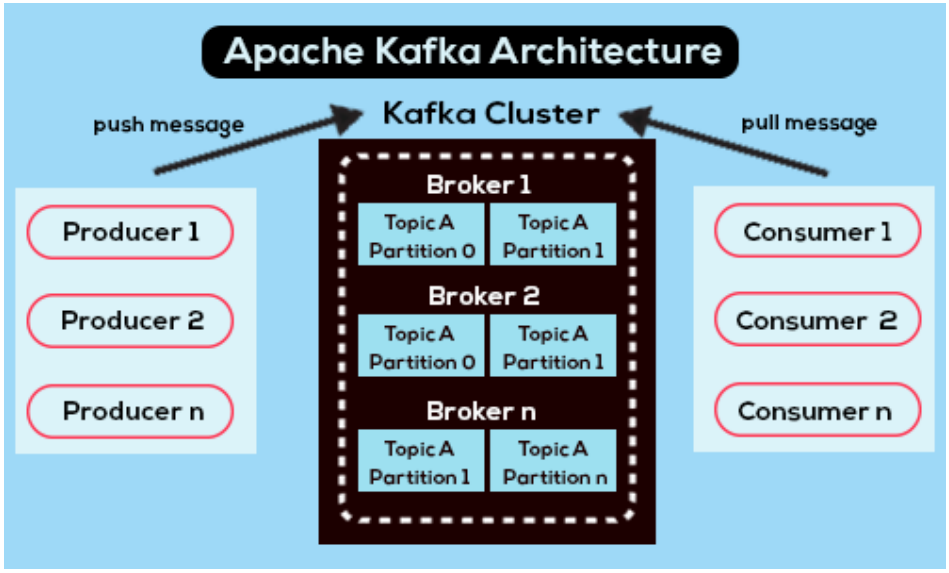


Figure 2.2: Kafka Architecture [31]

What makes Kafka so fast and efficient is its architecture: messages are published to a topic by one or more producers, and the data is stored in the Kafka brokers, which are multiple nodes that form a cluster and allow for partitioning and replication of the

data [31]. When the system needs to increase its speed or data processing capabilities, new brokers can be added to the cluster, allowing for horizontal scaling.

The streaming data can then be read in parallel by consumers (in our case the Spark clusters), which can be grouped into consumers groups to further increase throughput.

2.3 Programming Language

Having chosen Spark as our distributed streaming processing engine, the possible alternatives for programming languages were the ones officially supported by the Spark API: Scala, Java and Python.

The first option to be ruled out was Java, since it offers the same functionalities of the Scala API, while having an older interface and limited support to functional programming.

Python, on the other hand, while not offering support to some of the latest features of the Spark library, was a much more compelling option, mainly thanks to its ease of use compared to Scala and the wide community support and native compatibility with most of the tools and libraries used in the data and machine learning field.

Ultimately, the language chosen for implementing the model and conducting the experiments was Scala, for the following reasons:

- A good chunk of the work consists in building a recommender algorithm from scratch, which involves performing mathematical operations with complex data structures such as distributed matrixes, in this context having a strongly typed language can make it easier to structure the code in a modular, well defined style and can speed up development by allowing to find many errors at compile time rather than run time
- The Scala API offers more functionalities with respect to the Python one, particularly the Dataset interface and the stateful processing capabilities.
- The core apache Spark and MlLib frameworks are open source and built with Scala, and it can be easier to replicate patterns and best practices from the original library when working with the same language.
- This thesis will make heavy use of the User Defined Functions in Spark (UDF), which can be quite inefficient if not programmed properly, using a native JVM language with access to low level operations can help write more optimized UDFs.

3. Proposed Approach

3.1 Distributed Collaborative Filtering

3.1.1 Theory Overview

In this chapter we will dive deeper into the latent factor models for recommendations, and propose a version of this algorithm that could work in our distributed, real-time environment.

As we mentioned in chapter 1, collaborative filtering algorithms work by storing the user-product preferences in a m x n matrix called rating matrix.

Since on almost every big website each user interacts with a very small fraction of the products/items, the rating matrix will be very sparse, which means that most of the preferences will be unknown.

In this context, generating recommendations comes down to solving 2 fundamental issues:

- How do we deal with a matrix that can potentially have millions of rows and millions of columns? Is there any way to reduce its dimensionality?
- How do we find the items that will most likely be appreciated by a user inside this huge problem space?

Latent factor models are a solution that has proven to be very effective for solving this complex challenge.

This class of algorithms works by trying to factorize the rating matrix by using much expressing it as a dot product of user and item factors.

The matrix R, of dimension m x n, will then be estimated as the dot product of the item factors, of size m x k, and of the transposed of the item factors, of size n x k, where k << m,n. This operation not only allows us to work with data structures with a greatly

reduced dimensionality, but can also be used for estimating the rating of unknown user-item pairs, and thus matrix factorization becomes a great way to compute user recommendations.

How can this work? The idea behind every factorization method is that dimensionality gets reduced by expressing each product and each user as a linear combination of k fundamental tastes. To express it in a human readable way we could say that for example Harry Potter is composed of 40% adventure, 30% fantasy, 20% drama and 10% comedy.

The main objective of the latent factor algorithm is then to minimize the estimation error, which is the difference between the actual rating matrix and the estimated matrix obtained from the factor multiplication:

$$\ell_{ALS} = \sum_{x,y \in \mathcal{T}} \left(\mathrm{R}_{x,y} - \widehat{\mathrm{R}}_{x,y}\right)^2 + \lambda(\| \mathrm{U} \|^2 + \| \mathrm{P} \|^2)$$

Where the rating matrixes are expressed as R and U and P are respectively the user and item factors, while λ is a regularization parameter.

It's possible to find the factor values that minimize the loss by computing the partial derivatives and setting them to zero:

$$\begin{cases} \dfrac{\partial l}{\partial Ux} = 0 \\ \dfrac{\partial l}{\partial P_y} = 0 \end{cases}$$

This implementation is very efficient and works very well in an offline setting but doesn't respect the latency requirements of a real time application.

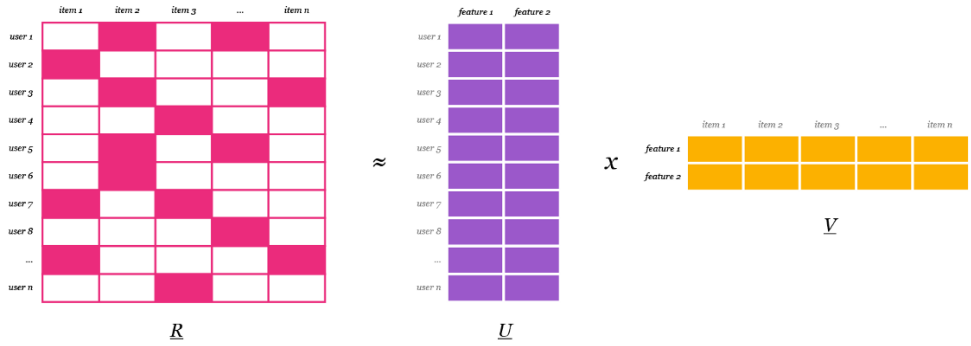


Figure 3.1: Matrix Factorization [29]

The objective of this thesis is to apply this approach in a distributed and real-time setting, let's focus on each problem at a time and then we'll see how to integrate them.

3.1.2 Spark Implementation

Apache Spark will be our main tool to tackle the distributed portion of this problem; MLlib implements a distributed offline version of the ALS algorithms itself, our approach will be an alternative implementation, adapted to our streaming use-case.

In order to implement a distributed matrix factorization algorithm, we need a data structure that represents a distributed matrix, supporting basic algebraic operations such as dot product, matrix inverse and transpose and efficient selection of specific rows and cells [33].

Apache Spark represents this concept with its abstract interface DistributedMatrix, implemented by two data structures that are very fitting for this use case:

- BlockMatrix, which represents a simple matrix divided into partitions and stored in distributed worker nodes. It offers interfaces to all the matrix operations we need (matrix sum, dot product, matrix transpose). The dot product implementation is particularly efficient, as it breaks down the matrix multiplication into blocks, minimizing the data shuffling among nodes. One big limitation of this data structure is that it doesn't allow for efficient selection of a specific row or cell.
- IndexedRowMatrix, an alternate implementation in which each Row as an index associated with it, which can be exploited to efficiently select a single row or cell. This class doesn't support basic algebraic operations but it's possible to efficiently convert it to a BlockMatrix to handle these kinds of operations.

By leveraging these 2 interfaces alongside the standard Spark RDDs we have everything we need to start working on our distributed Matrix Factorization implementation efficiently.

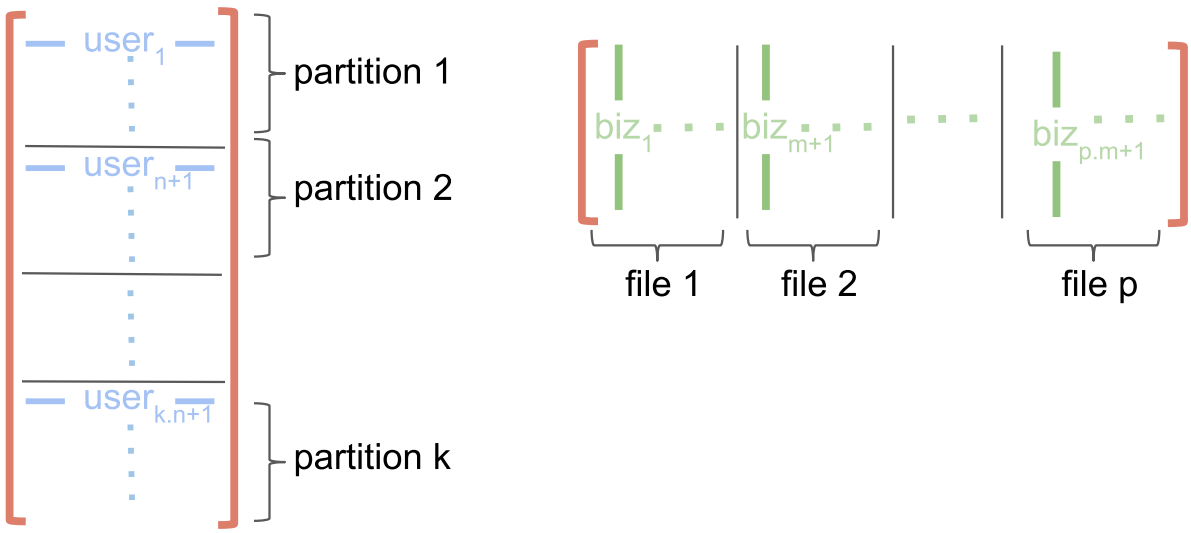


Figure 3.2: Distributed Matrix Multiplication [33]

Let's focus now on the challenges introduced by the real time requirement of the algorithm:

Having to process the data in real time gives a lot of limitations to our processing times, this means that we can't rely on the standard training processes since it would take too much time to factorize the whole rating matrix each time a new interaction (or a new minibatch of interactions) is generated.

The solution to this problem is not factorizing the whole matrix each time but using Stochastic Gradient Descent to update the factors with only one observation at the same time. [4 ,5]

This technique will come with a slight decrease in the factorization accuracy, but this is a necessary tradeoff for gaining the huge advantages of near real time latency we have already discussed about.

We will also introduce the concept of biases into the learning algorithm to represent how much different users rate the same products (this helps compensate for the sparser factorization method).

The estimated ratings formula with biases becomes:

$$\widehat{\mathrm{R}}_{x,y} = b_{x,y} + U_x^T \times P_y$$

Therefore, the loss function for B-SGD (Bias Stochastic Gradient Descent) is:

$$\ell_{SGD} = \sum_{x,y \in \mathcal{T}} \left(\mathrm{R}_{x,y} - \widehat{\mathrm{R}}_{x,y}\right)^2 + \lambda\left(\| \mathrm{U} \|^2 + \| \mathrm{P} \|^2 + b_x^2 + b_y^2\right)$$

As mentioned before, we won't be computing the full gradient for this loss function as it's unfeasible in an online scenario, but from this formula we can derive the B-SDG terms update equations, if we define the estimation error as:

$$\varepsilon_{x,y} = \mathrm{R}_{x,y} - \widehat{\mathrm{R}}_{x,y}$$

Then the update equations become:

$$\begin{aligned} b_x &\leftarrow b_x + \gamma\left(\epsilon_{x,y} - \lambda_x b_x\right) \\ b_y &\leftarrow b_y + \gamma\left(\epsilon_{x,y} - \lambda_y b_y\right) \\ \mathrm{U}_x &\leftarrow \mathrm{U}_x + \gamma\left(\epsilon_{x,y}\mathrm{P}_y - \lambda'_x \mathrm{U}_x\right) \\ \mathrm{P}_y &\leftarrow \mathrm{P}_y + \gamma\left(\epsilon_{x,y}\mathrm{U}_x - \lambda'_y \mathrm{P}_y\right) \end{aligned}$$

Where γ is the learning rate hyperparameter.

The algorithm is implemented using Spark Structured Streaming, all the necessary processing (data cleaning, wrong record eliminations, weighted multimodal interaction computation) will be executed using the Stream Dataframes API, and the model updates will be performed on minibatches of data aggregated by user id, implemented with a User Defined Function.

3.1.3 Pseudocode

Let's now dive deeper into the code implementation of the algorithm, the main variables involved are:

Variable Name	Variable Type	Description
userProductMatrix	BlockMatrix[Double]	Matrix containing the ratings currently processed by the real time model
userFactors	BlockMatrix[Double]	Matrix with the user factors
productFactors	BlockMatrix[Double]	Matrix with the product factors
userBiases	BlockMatrix[Double]	Matrix with the current user biases
productBiases	BlockMatrix[Double]	Matrix with the current product biases
userProductUpdateMatrix	BlockMatrix[Double]	Matrix containing the implicit ratings updates from the current mini batch, all the other values in the matrix are set to 0
estimatedUserProductMatrix	BlockMatrix[Double]	Matrix representing the estimate of the rating matrix obtained by

		performing dot product among user and product factors
estimationError	Double	Difference between the current factor estimation and the actual rating
ratingBatch	Array[Ratings]	Array containing the current ratings mini batch
currentUserRow	Array[Double]	Row of the actual rating matrix with the data of the current user
estimatedUserRow	Array[Double]	Row of the estimated rating matrix with the data of the current user
currentUserFactors	Array[Double]	Array with the factor row of the user being currently updated
currentProductFactors	Array[Double]	Array with the factor row of the item being currently updated
currentUserBiases	Array[Double]	Array with the bias row of the user being currently updated
currentProductBiases	Array[Double]	Array with the bias row of the item being currently updated

Table 3.1: Streaming Matrix Factorization Variables

While the models hyperparameters are:

Hyperparam Name	**Hyperparam Type**	**Description**
N_FACTORS	Integer	number of of factors used in the matrix factorization algorithm
N_USERS	Integer	Current number of users considered by the model
N_PRODUCTS	Integer	current number of items considered by the model
GAMMA	Double	Learning Rate
ALPHA	Double	SGD Hyperparameter
LAMBDA	Double	SGD Hyperparameter

Table 3.2: Streaming Matrix Factorization Hyperparameters

Here's the pseudocode for our streaming, distributed Matrix Factorization implementation in Spark.

Some clarifications:

- The BlockMatrix type is the Spark data type discussed in the previous chapter, all the matrix operations cited in the pseudocode are part of its API and can be executed efficiently and in parallel
- The toBlockMatrix() function is a helper tool that I wrote to efficiently convert standard Scala arrays in BlockMatrixes
- The toIndexedRowMatrix() function converts a Spark Block Matrix to a Spark IndexedRowMatrix, that are needed to extract specific rows from the matrixes

Algorithm 1 Distributed Streaming Matrix Factorization, Model Minibatch Update

```
1: userProductUpdateMatrix ← ratingBatch.toBlockMatrix()

   // updates the rating matrix with the new events
2: userProductMatrix ← userProductMatrix + userProductUpdateMatrix

   // computes the new estimatedProductMatrix
3: estimatedUserProductMatrix ←
     userFactors.multiply(productFactors.transpose)

4: for rating in ratingBatch do

5:   estimatedUserRow ←                  // extracts estimated user row from
       estimatedUserProductMatrix        // estimated rating matrix
```

```
        .toIndexedRowMatrix()
        .filter( rowId == rating.userId )

6:    currentUserRow ←                    // extracts actual user row from
        userProductMatrix                 // actual rating matrix
        .toIndexedRowMatrix()
        .filter( rowId == rating.userId )

7:    estimationError ← currentUserRow – estimatedUserRow

8:    currentUserFactors ←                // extracts current user factor row
        userFactors
        .toIndexedRowMatrix()
        .filter( rowId == rating.userId )

9:    currentProductFactors ←             // extracts current product factor row
        productFactors
        .toIndexedRowMatrix()
        .filter( rowId == rating.productId )

10:   userFactorsUpdate ←                 // computes the factors update with SGD
        currentUserFactors zip currentProductFactors
        .map( x, y  =>  GAMMA * (estimationError * y - LAMBDA * x))

11:   productFactorsUpdate ←              // computes the factors update with SGD
        currentUserFactors zip currentProductFactors
        .map( x, y  =>  GAMMA * (estimationError * x - LAMBDA * y))

12:   // adds update increment to user factors
      userFactors ← userFactors + userFactorsUpdate.toBlockMatrix()

      // adds update increment to product factors
13:   productFactors ←
        productFactors + productFactorsUpdate.toBlockMatrix()
```

```
14:   currentUserBias ←                    // extracts current user bias row
        userBias
        .toIndexedRowMatrix()
        .filter( rowId == rating.userId )

15:   currentProductBias ←                 // extracts current product bias row
        productBias
        .toIndexedRowMatrix()
        .filter( rowId == rating.productId )

      // computes the user bias update with SGD
16:   userBiasUpdate ←
        GAMMA* (estimationError - LAMBDA * currentUserBias)

      // computes the product bias update with SGD
17:   productBiasUpdate ←
        GAMMA * (estimationError - LAMBDA * currentProductBias)

      // adds update increment to user biases
18:   userBiases ← userFactors + userBiasUpdate.toBlockMatrix()

      // adds update increment to product biases
19:   productBiases ←
        productFactors + productBiasUpdate.toBlockMatrix()

20:  endfor
```

3.2 Implicit Multimodal Collaborative Filtering

3.2.1 Theory Overview

The approaches we have explored up until now were conducted in the classical setting of a recommender systems with explicit rating. As we mentioned before, the majority of real-life systems don't have access to explicit rating but have to estimate user preferences by using many different types of user-item interactions.

The use of implicit ratings for generating recommendations has been around for many years now [1, 9, 19, 21, 29], and it's become a popular solution for building recommender systems when explicit ratings are unavailable or not enough (which is by far the more common scenario).

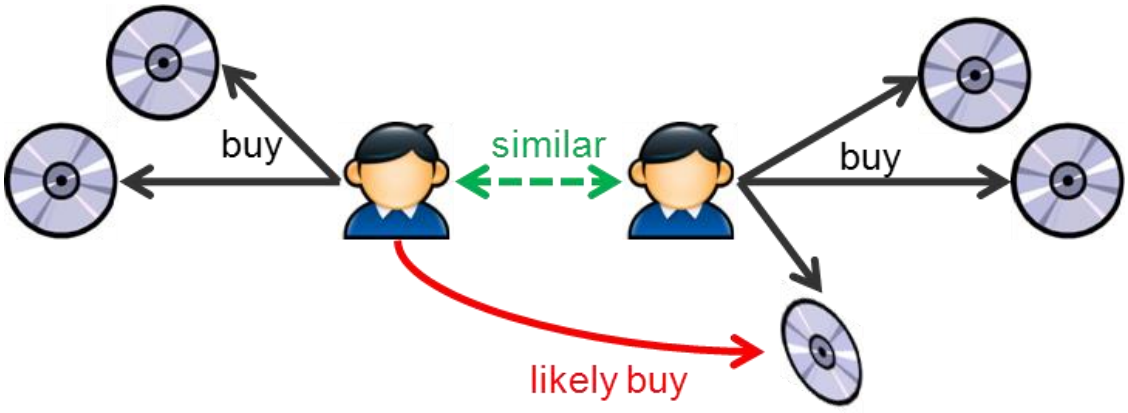


Figure 3.3: Collaborative Filtering [34]

This can be obtained by introducing the concept of preference (which is set to 1 if there's been any interaction among the user and the item) and confidence, derived from the implicit user-item interactions and representing how much we are sure of the rating we have estimated from the user interactions.

$$p_{ui} = \begin{cases} 1 & r_{ui} > 0 \\ 0 & r_{ui} = 0 \end{cases}$$

There are many ways for computing confidence from the implicit rating, this formula is very simple yet effective:

$$c_{ui} = 1 + \alpha r_{ui}$$

With these new concepts introduced into our recommendations systems, we can update the loss function as follows:

$$\mathrm{l} = \sum_{u,i} c_{ui}(p_{ui} - x_u^T y_i)^2 + \lambda \left(\sum_u \|x_u\|^2 + \sum_i \|y_i\|^2 \right)$$

This approach allows the system to function with implicit ratings, but we still don't have a way to take into account different types of interaction in the same recommendation model.

A very basic solution that incorporates this concept into the preference estimation is the weighted importance of interactions: each event will be used as an indicator of preference between the user performing the action and the item receiving it, and the amount of confidence increase will depend on the type of interaction.

For this approach to work, it's necessary to manually define the value of each interaction kind, in an e-commerce website a possible interaction-weight mapping could be:

- Item Search → 1
- Item Page View → 2
- Item Added to Favorites → 3
- Item Added to Cart → 4
- Item Purchase → 5

This method has the disadvantage of introducing more hyperparameters in the model that will have to be fine-tuned, but it allows for more accurate recommendations thanks to the more complete and diverse interaction set.

3.2.2 Pseudocode

Here's the pseudocode for our streaming, distributed Matrix Factorization implementation in Spark.

The pseudocode is similar to algorithm 1, with a couple of differences used to integrate implicit ratings and weighted importance:

- The weightedImportanceRatings is a function that given in input a batch of implicit ratings computes the updateMatrix taking into account the weights of each interaction kind
- The estimation error is computed using the implicit ratings methodology, using rating preference and confidence

Algorithm 2: Distributed Streaming Matrix Factorization, Model Minibatch Update with Multimodal Implicit Ratings

```
1:  userProductUpdateMatrix ← weightedImportanceRatings(ratingBatch)

    // updates the rating matrix with the new events
2:  userProductMatrix ← userProductMatrix + userProductUpdateMatrix

    // computes the new estimatedProductMatrix
3:  estimatedUserProductMatrix ←
      userFactors.multiply(productFactors.transpose)

4:  for rating in ratingBatch  do

5:    estimatedUserRow ←                  // extracts estimated user row from
        estimatedUserProductMatrix        // estimated rating matrix
       .toIndexedRowMatrix()
       .filter( rowId == rating.userId )
```

```
6:   currentUserRow ←                    // extracts actual user row from
       userProductMatrix                 // actual rating matrix
       .toIndexedRowMatrix()
       .filter( rowId == rating.userId )

7:   estimationError ← c( p(currentUserRow) – estimatedUserRow)

8:   currentUserFactors ←                 // extracts current user factor row
       userFactors
       .toIndexedRowMatrix()
       .filter( rowId == rating.userId )

9:   currentProductFactors ←              // extracts current product factor row
       productFactors
       .toIndexedRowMatrix()
       .filter( rowId == rating.productId )

10:  userFactorsUpdate ←                  // computes the factors update with SGD
       currentUserFactors zip currentProductFactors
       .map( x, y  =>  GAMMA * (estimationError * y - LAMBDA * x))

11:  productFactorsUpdate ←               // computes the factors update with SGD
       currentUserFactors zip currentProductFactors
       .map( x, y  =>  GAMMA * (estimationError * x - LAMBDA * y))

12:  // adds update increment to user factors
     userFactors ← userFactors + userFactorsUpdate.toBlockMatrix()

     // adds update increment to product factors
13:  productFactors ←
       productFactors + productFactorsUpdate.toBlockMatrix()

14:  currentUserBias ←                    // extracts current user bias row
       userBias
```

```
            .toIndexedRowMatrix()
            .filter( rowId == rating.userId )


15:     currentProductBias ←              // extracts current product bias row
            productBias
            .toIndexedRowMatrix()
            .filter( rowId == rating.productId )

        // computes the user bias update with SGD
16:     userBiasUpdate ←
            GAMMA* (estimationError - LAMBDA * currentUserBias)

        // computes the product bias update with SGD
17:     productBiasUpdate ←
            GAMMA * (estimationError - LAMBDA * currentProductBias)

        // adds update increment to user biases
18:     userBiases ← userFactors + userBiasUpdate.toBlockMatrix()

       // adds update increment to product biases
19:     productBiases ←
            productFactors + productBiasUpdate.toBlockMatrix()

20:  endfor
```

3.3 Association Rule Matching

3.3.1 Theory Overview

The weighted importance approach to multimodal, while being a viable and functioning solution, comes with many limitations and inefficiencies:

- The weight of each type of interaction must be defined manually, and there's no guarantee that the decided value will be the optimal ones
- Whenever a new kind of interaction gets added to the system, the weight configuration needs to be updated
- Each action is still considered as a standalone event, we never consider the meaning of action sequences when computing user preferences

A better way to handle multimodal interaction would be focusing more on the patterns and sequences of interaction that each user performs on the website and try to derive some information about their future behavior given this information.

A very common and effective way to discover common patterns inside sequences of events is the data mining technique called association rules mining [35].

Association Rules mining takes all possible itemsets in a list of transactions (an itemset is a subset of all the possible sequences of transactions) and computes 2 key metrics:

- Support: number of times an itemset appears in a transaction with respect to the total number of transactions.

$$Support(Itemset) = \frac{Number\ of\ Transactions\ Containing\ Itemset}{Total\ Number\ of\ Transactions}$$

- Confidence: likeliness of an association rules, meaning how probable is that a pattern X in the user navigation will lead to a pattern Y in the future.

$$Confidence(X \rightarrow Y) = \frac{Support(X \cup Y)}{Support(X)}$$

There are many different algorithms for finding association rules from a transaction set and for computing support and confidence of the itemset, one of the most recent and efficient approaches is the FPGrowh algorithm [40, 41].

3.3.2 Spark Implementation

Spark offers an efficient distributed implementation of the FPGrowth algorithm for Association Rules mining, that can be used to compute all the transaction sets that display a support or confidence exceeding a certain threshold.

In our specific use case, we could use transactions to compute with how much confidence we can say that a user will perform an action after displaying a certain interaction pattern. We could for example discover that in our ecommerce website, after a user view an item A for 10 seconds, clicks on an item B and scrolls through the item B description we could say with 80% confidence that he will purchase item A.

The concept of confidence in association rules is similar to the one used in collaborative filtering with implicit preferences, so it seems reasonable that there could be a way to integrate these 2 approaches.

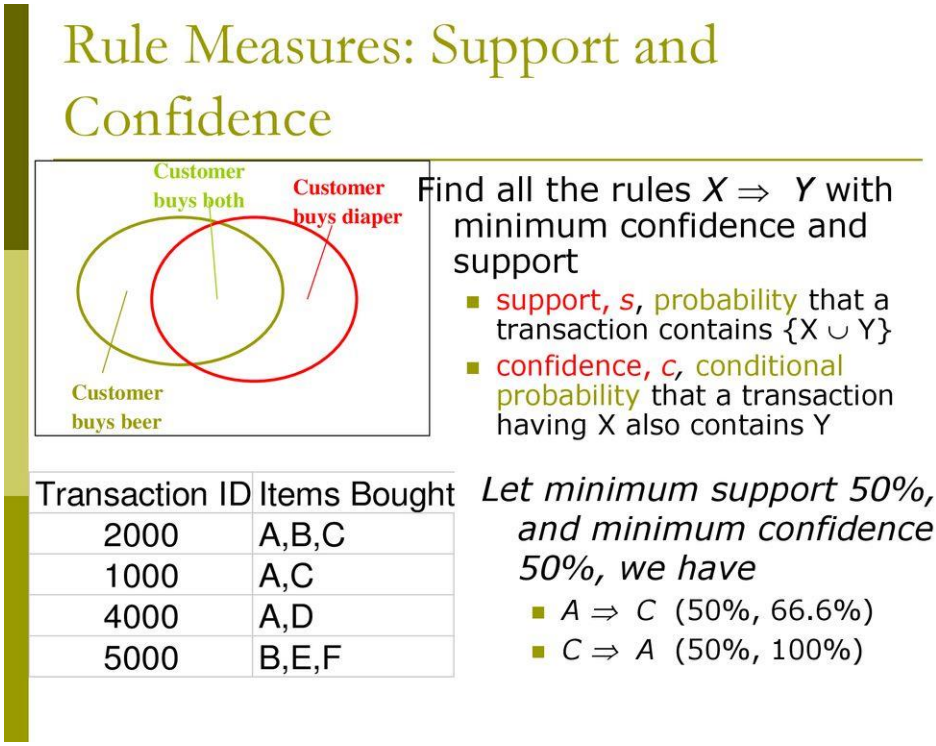


Figure 3.4: Support and Confidence in Association Rules [35]

3.3.3 Spark Implementation

A good approach could be to perform association rules mining with the FPGrowth algorithm on a subset of the clickstream dataset, and then use them to compute the ratings value on the remaining clickstream real time events.

The basic idea is that people that share similar behavioral patterns would also share similar tastes, so whenever the stream processing engine finds a match between the dataset of popular patterns and the sequence of live events of a certain user, we can increase the confidence for the rating of the item that pattern leads to, in a proportional manner with the confidence value of the association rule.

Pattern matching on live clickstream can be performed with the Spark Streaming SQL engine and using a 2-minute sliding window function, which will concatenate all the events descriptors (view_on_item_x, addToCart_on_item_y…) and aggregate the by user id.

This Stream will then be joined with the static Association Rules dataset and for every match found the rule confidence will be extracted and added to the next model update UDF minibatch.

3.3.4 Pattern Matching Spark SQL Implementation

Here's the Spark code used for generating the pattern detection and matching on live clickstream data:

```
val sequenceByUser = lines  // concatenates sequences of events by 50 seconds window and by user
  .withColumn("actionToItem", concat($"eventType", lit("_"), $"productId"))
  .withWatermark("unixTimestamp", "30 seconds")
  .groupBy(
    window($"unixTimestamp", "50 seconds"),
    $"userId",
  )
  .agg(concat_ws(",", collect_list($"actionToItem")).alias("pattern"))

val patternsMatch = ratingsByUser // matches live events sequences with the popular association rules patterns
  .join(associations, ratingsByUser("pattern") === associations("seq_pattern"), "inner")
  .select("userId","seq_pattern","confidence") //extracts implicit rating from user pattern
```

Figure 3.5: Pattern Matching on Clickstream Spark SQL Query

4. Experiments and Evaluation

4.1 Datasets

4.1.1 Cloudera Clickstream

The objective of this thesis is to explore the potential of distributed, real time recommender systems under different scenarios.

This means that the structure of the conducted experiments will be hierarchical, starting with simple datasets and algorithms and then moving on with more and more sophisticated simulations as we move one.

The requirements for the first experiments are just to test our recommender system in a distributed and streaming scenario, so the first iteration will be performed on a very basic clickstream dataset representing user navigation [39], taken from Cloudera Omniture

Here's an extract of the dataset loaded as a Spark Dataframe, after being preprocessed and selecting only the relevant columns for this experiment:

timestampCast	ip	url	product_id	swid	user_id
2012-03-01 00:00:27	147.222.227.200	http://www.acme.com/SH559040/VD55175948	27	{AF8A0FDF-B1F8-474C-8CD7-8CA06A8E435B}	8621
2012-03-01 00:00:59	71.217.29.209	http://www.acme.com/SH559040/VD55175948	27	{6FE1CB72-95C9-47F9-A1CB-7295C927F916}	5147
2012-03-01 00:01:21	147.222.227.200	http://www.acme.com/SH559040/VD55175948	27	{AF8A0FDF-B1F8-474C-8CD7-8CA06A8E435B}	8621
2012-03-01 00:01:31	68.5.184.133	http://www.acme.com/SH55126545/VD55173061	7	{60C8049D-C1A2-41C2-B503-6C1200424C49}	4473
2012-03-01 00:01:45	69.114.3.205	http://www.acme.com/SH559040/VD55175948	27	{F761B842-9DDA-42CC-9F28-A6359B6C7219}	11978
2012-03-01 00:02:00	67.212.209.115	http://www.acme.com/SH559040/VD55175948	27	{339A8DBD-0ABC-43CF-B417-110BB9BBEC2C}	2397
2012-03-01 00:03:10	70.187.162.17	http://www.acme.com/SH55126545/VD55165149	4	{88FC5287-907D-48FF-82A3-134F2EBC6351}	6238
2012-03-01 00:03:29	67.183.219.15	http://www.acme.com/SH5580165/VD55173281	13	{C9F1CFC4-ED7B-4DE1-9C56-A25CE86B2F6C}	9871
2012-03-01 00:04:13	99.102.31.167	http://www.acme.com/SH55126545/VD55173061	7	{7A7EC1B7-BAFE-431A-A4B5-0378A4CDE8CB}	5629
2012-03-01 00:04:46	96.228.252.94	http://www.acme.com/SH559040/VD55175948	27	{E68BAEE7-C23D-48B5-956B-64EECE0BF344}	11223
2012-03-01 00:05:12	24.9.63.79	http://www.acme.com/SH55126545/VD55149415	2	{DEADBDB9-274E-4119-8184-DA17A059D6CE}	10877
2012-03-01 00:05:32	24.9.63.79	http://www.acme.com/SH55126545/VD55149415	2	{DEADBDB9-274E-4119-8184-DA17A059D6CE}	10877
2012-03-01 00:05:37	68.52.198.113	http://www.acme.com/SH55126545/VD55149415	2	{DA55421E-31B5-45E5-9542-1E31B575E5CC}	10662
2012-03-01 00:05:48	170.223.207.33	http://www.acme.com/SH5580165/VD55173281	13	{1415FEC5-B602-96AA-BD43-6FCB08E16293}	898
2012-03-01 00:07:55	67.224.130.63	http://www.acme.com/SH55126545/VD55170364	6	{6375FC70-8A3D-444A-B5FC-708A3DF44AF7}	4594
2012-03-01 00:08:06	71.217.29.209	http://www.acme.com/SH559040/VD55175948	27	{6FE1CB72-95C9-47F9-A1CB-7295C927F916}	5147
2012-03-01 00:08:42	68.5.184.133	http://www.acme.com/SH55126545/VD55173061	7	{60C8049D-C1A2-41C2-B503-6C1200424C49}	4473
2012-03-01 00:09:26	98.228.34.12	http://www.acme.com/SH55126545/VD55173061	7	{D956528D-3CAC-4035-BB1C-79A8D4F4996A}	10615
2012-03-01 00:09:34	98.228.34.12	http://www.acme.com/SH55126545/VD55173061	7	{D956528D-3CAC-4035-BB1C-79A8D4F4996A}	10615
2012-03-01 00:09:58	207.62.158.62	http://www.acme.com/SH55126545/VD55163347	3	{E1D2DE32-6F67-4A50-ABC1-102555658531}	11008

Figure 4.1: Cloudera Clickstream dataset

- Timestamp, used to build the event order in our real time interaction simulator
- The ip address, necessary to differentiate between different user session and build the stream aggregation windows
- User and Item identifiers, essential to the Matrix Factorization algorithm.
- User and Item name, helpful during the testing and evaluation phases

The preference rating proxy will be unimodal and represented by the user to item url navigation.

4.1.2 Retailrocket Recommender System Dataset

In order to progress with the more complex experiments and leverage multimodal clickstream input, it is necessary to exploit a richer dataset, offering more details into the user navigation patterns.

The Retailrocket dataset, used also in a 2017 Kaggle challenge [40], is a very good fit for these requirements.

timestamp	visitorid	event	itemid	transactionid
1433221332117	257597	view	355908	null
1433224214164	992329	view	248676	null
1433221999827	111016	view	318965	null
1433221955914	483717	view	253185	null
1433221337106	951259	view	367447	null
1433224086234	972639	view	22556	null
1433221923240	810725	view	443030	null
1433223291897	794181	view	439202	null
1433220899221	824915	view	428805	null
1433221204592	339335	view	82389	null
1433222162373	176446	view	10572	null
1433221701252	929206	view	410676	null
1433224229496	15795	view	44872	null
1433223697356	598426	view	156489	null
1433224078165	223343	view	402625	null
1433222531378	57036	view	334662	null
1433223239808	1377281	view	251467	null
1433223236124	287857	addtocart	5206	null
1433224244282	1370216	view	176721	null
1433221078505	158090	addtocart	10572	null

Figure 4.2: Retailrocket Dataset

As before, we have all the necessary features to perform real time recommendations on clickstream data (timestamp, userID and productID), with the additional information of the type of interaction, which can be:

- View
- AddToCart
- Transaction

Having a richer set of information on the clickstream action makes it possible to apply the weighted importance and association rules-based approaches to this dataset, we will have then a chance to compare these 2 methods with the unimodal one.

4.2 Simulating the Real Time Interactions

Since we are dealing with streaming data, the data ingestion phase assumes critical importance in the overall functioning of the system.

In an offline setting it would be enough to just collect all the most recent user preferences/ratings and feed them to our recommendation model for the training phase, without paying too much attention to the generation time of each record.

In our case, having realistic ingestion times is necessary for simulating what the behavior of the model would be in a real-life setting.

The next best thing is writing a Kafka Producer that would produce a semi-realistic simulation of the interaction streams generated by the user navigation.

Of course, to make the simulation feasible with our limited resources and in our limited time scope, it was necessary to make a few simplifications and assumptions while designing the producer, that would allow our experiment to function but at the same time not impact the overall model behavior:

- One single producer acting as multiple users conducting clickstream events on a website
- Simple loop going through the clickstream datasets periodically and publishing event data in a Kafka topic
- Data is published in Avro format with the following schema:

Column Name	Type	Description
Timestamp	Unix Timestamp	Unix Timestamp of the record time of the clickstream
RemoteHost	String	Client device info
User	String	Name of the user producing the event
UserId	String	Id of the user producing the event

Product	String	Name of the item involved in the interaction
ProductId	String	Id of the item involved in the interaction
EventType	String	Category of the interaction event

Table4.1: Clickstream message schema

And this is a diagram explaining the application logic of the simulation:

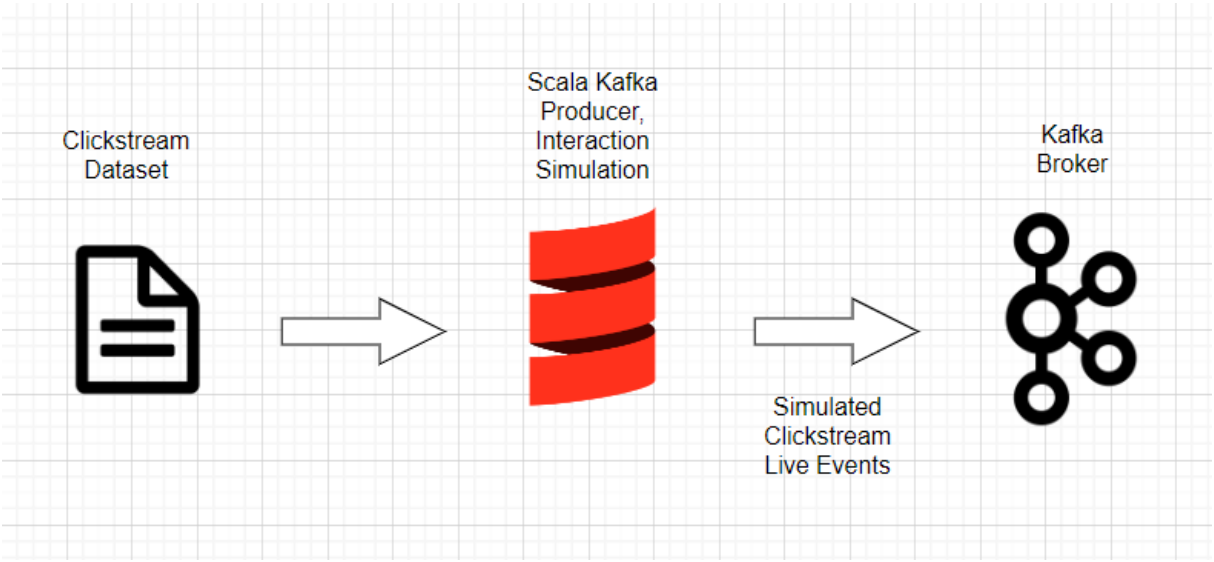


Figure 4.3: Simulation Architecture

4.3 Architecture

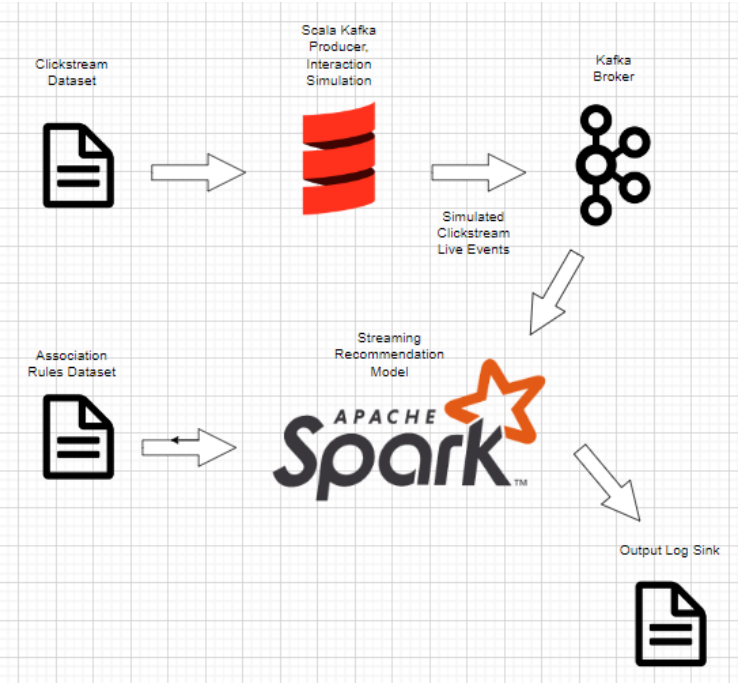
The architecture of the system will remain substantially the same among all experiments, with 3 main processing components, 1 data source and 1 data sink:

- Input Datasets with Clickstream data and Association Rules
- Scala Interaction Simulator
- Kafka Cluster and Brokers + Zookeeper
- Spark Streaming processing cluster
- Ouput File Sink

All the experiments will be run on a single laptop with an octa-core Intel i5 processor, 8 GB of RAM and SSD storage inside a standard Ubuntu Docker container.

The system will have 2 Kafka brokers deployed, 1 core will be occupied by the interaction simulator and another 1 by the Zookeeper server, while the Spark cluster will have 3 nodes with 1 core each.

Figure 4.4: Experimental Architecture



4.4 Testing the Architecture

The very first experiment that was conducted was meant to test out the processing capabilities of the system and simulate realistic real-time clickstream data generation, ingestion, and processing.

The basic idea of this initial experiment is to simulate in order to have a general-purpose testing environment to familiarize and perform the first real time transformations on clickstream data, as a starting point to conduct progressively more complex analyses.

The idea is to run a local e-commerce website from an open-source frontend application, write customized clickstream tracking scripts in Javascript to collect different kind of user to item interactions, record them all in a clickstream collector and then feed them as an input to our analytical architecture to test out its functioning capabilities.

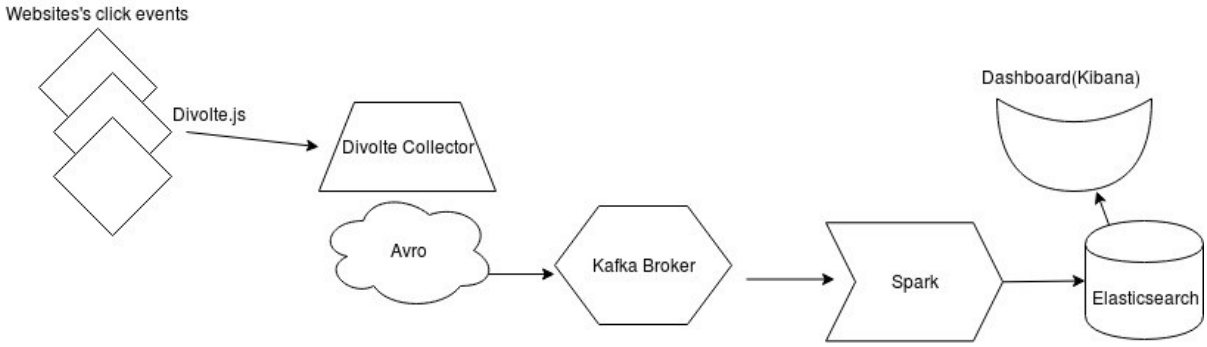


Figure 4.5: Divolte example data collection architecture [43]

An additional component that was introduced during the architecture in this step was the clickstream collector Divolte: this component allows to easily collect and customize and kind of user interaction and behavior in a website through asynchronous browser event tracking, collect all this data in a scalable way and then channel all this data a variety of sinks, in our case a Kafka cluster, where it can be used for analytical purposes.

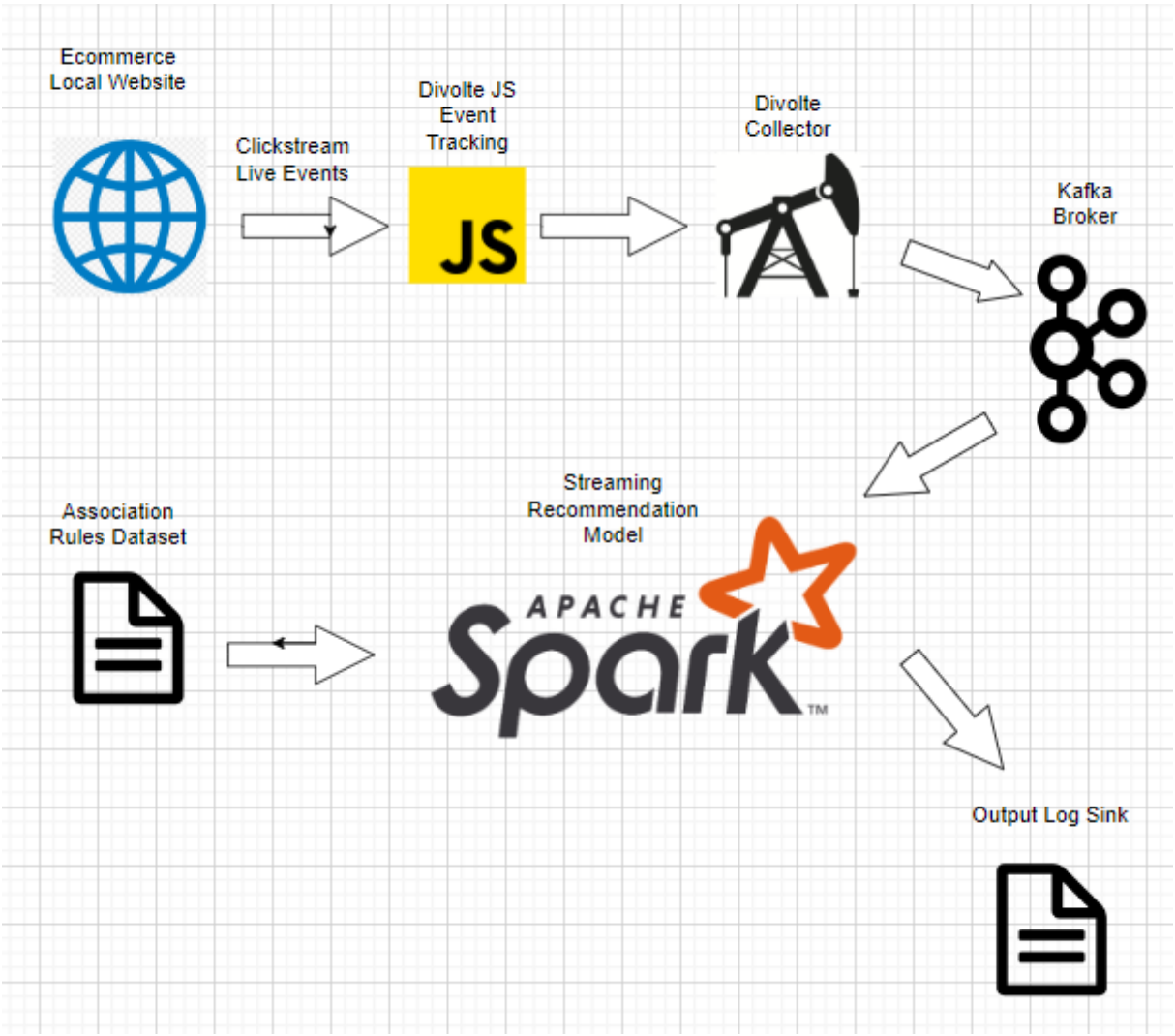


Figure 4.6: Divolte testing environment

This local environment was used to test the basic data flow of the system and experiment with spark queries and real time model capabilities, before moving on to more complex and extensive experiments.

This kind of architecture is also very descriptive because it can be a pretty good approximation of an architecture for a real-world deployment of a real time recommender system for a large scale website.

4.5 Implicit Single Modal

4.5.1 Introduction and Objective Definition

As mentioned in the previous paragraph, the experiments will be conducted in increasing order of complexity, in order to test out the different aspects of the proposed algorithms and evaluate and compare the different approaches.

The experiments began with a classic unimodal input based on the Cloudera Omniverse Dataset, the proxy for the implicit rating was the only interaction provided by the dataset, the page view.

The objective of this first experiment is to test out the distributed and real time capabilities of the algorithm in a traditional recommendation setup, without focusing too much on possible optimization that can be done in a streaming environment.

4.5.2 Data Preprocessing

The data contained in the dataset, by themselves, are not enough to be made into a proper input for our recommendation model. The dataset provides is in fact for each interaction the URL of the item and a string identifier for the user performing the action, but it lacks the fundamental information of an integer identifier for the users and items.

This issue can be solved quite easily by enriching the dataset, association a unique id to each item URL and to each SWID.

The implementation of this ID generator was done as a standard Spark script, the core of the implementation is this SQL query run within the Spark query engine:

```
select timestampCast,
          |       ip,
          |       url,
          |       dense_rank() over(partition by 1 order by url) as product_id,
          |       swid,
          |       dense_rank() over(partition by 1 order by swid) as user_id
          |   from ClickStream
          |   order by timestampCast
```

Figure 4.7: Data Preprocessing Spark SQL Query

timestampCast is the converted unix timestamp of the interaction read from the raw dataset, while the product_id and user_id were built by leveraging the dense_rank function, which provides an unique index to ordered elements inside a given window (in our case the windows were the url and swid columns respectively).

Here's how the dataset looks after this ID generation process:

timestampCast	ip	url	product_id	swid	user_id
2012-03-01 00:00:27	147.222.227.200	http://www.acme.com/SH559040/VD55175948	27	{AF8A0FDF-B1F8-474C-8CD7-8CA06A8E435B}	8621
2012-03-01 00:00:59	71.217.29.209	http://www.acme.com/SH559040/VD55175948	27	{6FE1CB72-95C9-47F9-A1CB-7295C927F916}	5147
2012-03-01 00:01:21	147.222.227.200	http://www.acme.com/SH559040/VD55175948	27	{AF8A0FDF-B1F8-474C-8CD7-8CA06A8E435B}	8621
2012-03-01 00:01:31	68.5.184.133	http://www.acme.com/SH55126545/VD55173061	7	{60C8049D-C1A2-41C2-B503-6C1200424C49}	4473
2012-03-01 00:01:45	69.114.3.205	http://www.acme.com/SH559040/VD55175948	27	{F761B842-9DDA-42CC-9F28-A6359B6C7219}	11978
2012-03-01 00:02:00	67.212.209.115	http://www.acme.com/SH559040/VD55175948	27	{339A8DBD-0ABC-43CF-B417-110BB9BBEC2C}	2397
2012-03-01 00:03:10	70.187.162.17	http://www.acme.com/SH55126545/VD55165149	4	{88FC5287-907D-48FF-82A3-134F2EBC6351}	6238
2012-03-01 00:03:29	67.183.219.15	http://www.acme.com/SH5580165/VD55173281	13	{C9F1CFC4-ED7B-4DE1-9C56-A25CE86B2F6C}	9871
2012-03-01 00:04:13	99.102.31.167	http://www.acme.com/SH55126545/VD55173061	7	{7A7EC1B7-BAFE-431A-A4B5-0378A4CDE8CB}	5629
2012-03-01 00:04:46	96.228.252.94	http://www.acme.com/SH559040/VD55175948	27	{E68BAEE7-C23D-48B5-956B-64EECE0BF344}	11223
2012-03-01 00:05:12	24.9.63.79	http://www.acme.com/SH55126545/VD55149415	2	{DEADBDB9-274E-4119-8184-DA17A059D6CE}	10877
2012-03-01 00:05:32	24.9.63.79	http://www.acme.com/SH55126545/VD55149415	2	{DEADBDB9-274E-4119-8184-DA17A059D6CE}	10877
2012-03-01 00:05:37	68.52.198.113	http://www.acme.com/SH55126545/VD55149415	2	{DA55421E-31B5-45E5-9542-1E31B575E5CC}	10662
2012-03-01 00:05:48	170.223.207.33	http://www.acme.com/SH5580165/VD55173281	13	{1415FEC5-B602-96AA-BD43-6FCB08E16293}	898
2012-03-01 00:07:55	67.224.130.63	http://www.acme.com/SH55126545/VD55170364	6	{6375FC70-8A3D-444A-B5FC-708A3DF44AF7}	4594
2012-03-01 00:08:06	71.217.29.209	http://www.acme.com/SH559040/VD55175948	27	{6FE1CB72-95C9-47F9-A1CB-7295C927F916}	5147
2012-03-01 00:08:42	68.5.184.133	http://www.acme.com/SH55126545/VD55173061	7	{60C8049D-C1A2-41C2-B503-6C1200424C49}	4473
2012-03-01 00:09:26	98.228.34.12	http://www.acme.com/SH55126545/VD55173061	7	{D956528D-3CAC-4035-BB1C-79A8D4F4996A}	10615
2012-03-01 00:09:34	98.228.34.12	http://www.acme.com/SH55126545/VD55173061	7	{D956528D-3CAC-4035-BB1C-79A8D4F4996A}	10615
2012-03-01 00:09:58	207.62.158.62	http://www.acme.com/SH55126545/VD55163347	3	{E1D2DE32-6F67-4A50-ABC1-102555658531}	11008

Figure 4.8: Preprocessed Clickstream Dataframe

Now the data is finally ready to be fed as an input of our Streaming Recommendation Model.

4.5.3 Experiment Setup

The experimental setup had to be modified with respect to the initial testing architecture; the main challenge of this new set of experiments is that it's necessary to generate a great amount of clickstream data, by simulating in a realistic way a stream of user interactions generated in real time while at the same time working inside all the limitations of a small-scale research environment with a limited-time scope.

The approach used to tackle this problem was a simulation script that would act as a Kafka producer and try to recreate as closely as possible the real time data generation dynamics that would occur in a real-world clickstream collection system.

The simulation script will load the clickstream data contained in the datasets, order them by timestamp, and publish them into the Kafka cluster, effectively replacing the role that would be be taken by a Divolte collector in a real-world deployment.

4.6 Implicit Multimodal

4.6.1 Introduction and Objective Definition

The second experiment will focus on adding the multimodal input management into the recommendation system model, we still won't be focusing on applying additional pattern detection techniques to the processing pipeline, this will be done in the next paragraph with the association rules mining based approach.

In order to add this complexity layer to our system, it will be necessary to move to the Retailrocket dataset, since the Cloudera one doesn't offer information on the user-item interaction time.

4.6.2 Weighted Importance

The approach to the multimodal input used is the weighed importance method, that requires a weight definition to be associated to each different interaction type, according to how meaningful that interaction type is considered to be in the preference estimation.

A good metric to help define the weights value is the cardinality of the interactions in the dataset, for the Retailrocket dataset we have:

- View → 97%
- AddToCart → 2-3%
- Transaction → <1%

A very reasonable assumption that can be made is that the rarer an event is, the more indicative of user preference it is. If we apply this reasoning to our dataset, it can be understood that a view event is very common and doesn't carry much significance to the preference estimation, while an AddToCart or Transaction event carries a very strong indication that the user likes a certain product.

We can then propose a very simple but reasonable value set for the weight definition, which is:

Interaction Type	Event Frequency	Weight Importance
View	97.%	0.5
AddToCart	2.3%	2
Transaction	0.7%	3

Table 4.2: Interaction types

4.6 Implicit Multimodal with Association Rules

The next step in the roadmap of the work is trying to apply pattern detection as an implicit way of measuring user ratings and interests.

The basic idea behind collaborative filtering is that a user who has expressed similar ratings on certain items to other people will likely also appreciate other content liked by those similar users. The proposed approach is based on the idea that ratings on items can not only be estimated by user interactions, but they can also be extracted from repeated and common user patterns inside a website, by mining clickstream data

Association rules can be a way to discover pattern and common behaviors in navigation data, we will be using them to extract the sequence of actions that are more commonly executed before performing an interaction (view, addToCart or transaction) on an item.

For this experiment, even more than in the previous one, where we tested the weighted importance approach, it is very significant to use a clickstream dataset offering multimodal user-item interactions, so the choice landed once again on the Retailrocket dataset. When mining user patterns in fact, the richer the action set is the more accurate and precise our behavior estimation will be.

The proposed approach to simulate a real time deployment of this algorithm is the following:

- Order the Retailrocket dataset by timestamp and split it in two subsets, the first one will be used to extract the association rules in an offline setting, while the second one will be used to simulate a real time clickstream ingestion and recommendation in an ecommerce website
- The computed association rules will be loaded in a Spark Dataframe, and the incoming interaction streams will be aggregated by user id with a 50 seconds sliding windows in order to compute the real time patterns and then they will be joined with the association rules static dataset.
- Each time a new real time pattern is detected by the stream processing engine (aka a join is found with the association rules dataset) we consider this event as an implicit rating towards the item u pointed in the matched association rule. The value of the rating will be proportional to the confidence value of the rule
- As in previous experiments, rating events are batched together in minibatches and the training streaming UDF function is called to perform matrix factorization and update our model with stochastic gradient descent

```
[view_381941,view_241555=>view_325310 ] 0.4634146341463415
[view_408907,view_19648,view_158498,view_165030,view_90219,view_411373=>view_373403 ] 1.0
[view_408907,view_19648,view_158498,view_165030,view_90219,view_411373=>view_242548 ] 0.9836065573770492
[view_408907,view_19648,view_158498,view_165030,view_90219,view_411373=>view_96904 ] 1.0
[view_19648,view_242548=>view_373403 ] 1.0
[view_19648,view_242548=>view_408907 ] 1.0
[view_19648,view_242548=>view_158498 ] 0.9142857142857143
[view_19648,view_242548=>view_165030 ] 0.9428571428571428
[view_19648,view_242548=>view_90219 ] 0.9571428571428572
[view_19648,view_242548=>view_96904 ] 0.9285714285714286
[view_19648,view_242548=>view_411373 ] 0.9571428571428572
[view_373403,view_408907,view_158498,view_242548,view_411373=>view_165030 ] 0.9836065573770492
[view_373403,view_408907,view_158498,view_242548,view_411373=>view_90219 ] 1.0
[view_373403,view_408907,view_158498,view_242548,view_411373=>view_96904 ] 1.0
[view_373403,view_408907,view_158498,view_242548,view_411373=>view_19648 ] 1.0
[view_180751=>view_262258 ] 0.11869436201780416
[view_180751=>addtocart_180751 ] 0.06528189910979229
[view_180751=>view_298009 ] 0.5548961424332344
[view_180751=>view_63119 ] 0.0712166172106825
[view_180751=>view_115183 ] 0.05637982195845697
[view_272527=>view_93941 ] 0.08888888888888889
[view_56696=>view_215715 ] 0.21929824561403508
[view_56696=>view_79572 ] 0.18421052631578946
[view_372188=>view_85914 ] 0.09698996655518395
[view_372188=>addtocart_372188 ] 0.07357859531772576
[view_372188=>view_272324 ] 0.06354515050167224
[view_46443=>view_102306 ] 0.20300751879699247
```

Figure 4.9: Association Rules extracted from the Retailrocket Dataset

The association rules were computed on 80% of the dataset, while the real time ingestion simulation was performed using a subset of the remaining 20% containing a fixed number of users and items, in order to cope with the very limited processing power at our disposal for the model testing.

4.7 Evaluation Methodology

The online setting of the problem makes evaluation more difficult to handle than in a classical offline machine learning scenario.

The most popular methods used in literature for evaluating machine learning models in a streaming environment are all based on the prequential evaluation technique [10, , 11, 12, 13].

Prequential evaluation operates with a test and learn system, each time a new interaction is collected by the system, the following operations are executed:

- Make a recommendation to the user performing the interaction for the top n elements
- Score the newly generated recommendations
- Update the model with the current event

This type of evaluation fits well for many stream learning scenarios, since works in a similar incremental fashion, and it offers many advantages over classical offline methods such as real time monitoring and metrics computation, while offering the chance to adjust model parameters online as well.

However, the limited processing power at or disposal makes it difficult to perform both model training and prequential evaluation (which requires to generate recommendations each time a new event is registered) on a single machine, and the key focus of this method is predicting the next action for each user, which isn't exactly the scope of this thesis's research.

Moreover, the online nature of prequential evaluation makes it so that this method can only really evaluate accurately a recommendation system when it is deployed on the website at the same time that the evaluation occurs, in order to actually test out the effect that the recommendations generated in real time have on the users behaviors. In this particular use case, the model was not deployed in a real system but the real time interactions were recreated a posteriori using a simulation, so there is no way of seeing the actual impact of the model on the interactions.

For all these reasons, prequential evaluation was ruled out as an evaluation methodology for this algorithm, in favor of the other popular methodology for evaluating stream learning algorithms: holdout evaluation of an independent test set.

This method is more similar to the more traditional batch learning ones [13, 21] (which is also an advantage since it makes it easier to compare offline recommenders with their online counterparts, as we will soon see), but it presents a few alterations to make it suitable to a stream machine learning system.

This method works by splitting the dataset into two independents subsets, each independent from each other, one used for training the model and the other one used for testing it. The model is then deployed in the online learning scenario and trained incrementally on the training set (simulating a real time ingestion of the user-item interactions as discussed in the previous chapter). Finally, the user and item factors trained in real time are periodically extracted from the online model and used for evaluation in an offline environment on the test set.

This method basically treats the stream model as a sequence of batch learning events and performs the evaluation at different points in time as if it were an offline environment.

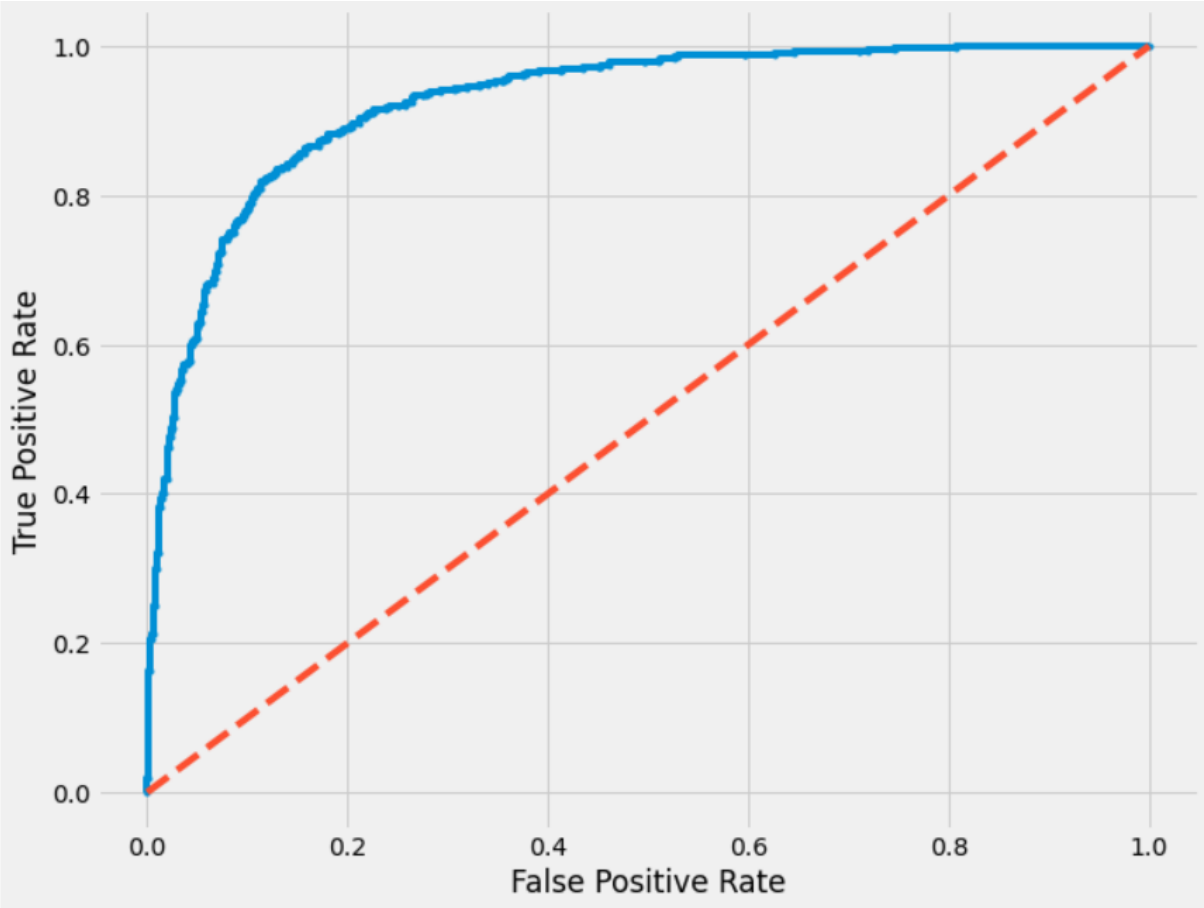


Figure 4.10: ROC curve example [36]

For each evaluation batch we will then have a fixed size rating matrix created with the ratings collected up to that specific point in time and the matrix factorization factors extracted from the online model.

A popular procedure used in similar use cases is the dataset mask, in which a percentage of the ratings dataset gets hidden (usually 20%), then the model generates recommendations for the users and compares the ranking of the suggested items with the future purchases/masked ratings of the users.

The chosen metric for the evaluation is the Area Under the ROC curve (AUC), which is a figure usually used for classifiers, but that can also be very effectively used for evaluating recommendations [36].

The greater the AUC, the closer our recommendations will be to the actual purchases or ratings.

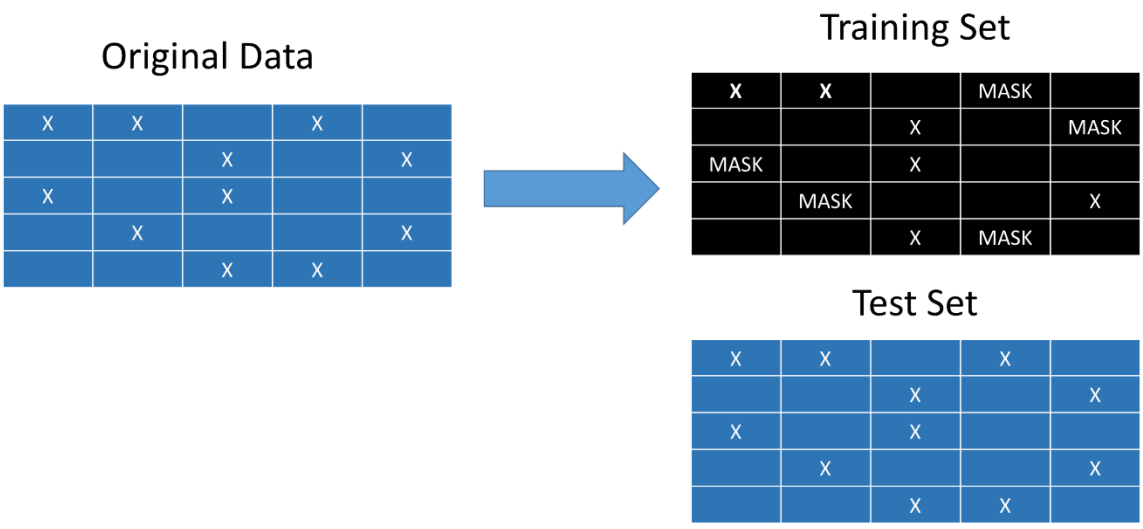


Figure 4.11: Masked dataset technique [21]

4.8 Experimental Results

In this section we will go over the results of the experiments and give some more details on the experimentation procedure.

The 3 main threads of experiments performed were the ones presented in the previous sections of this chapter:

- Real time recommendations with implicit ratings and unimodal events performed on the Cloudera Omniture dataset
- Real time recommendations with implicit ratings and multimodal events (3 categories) using the weighted importance technique over the Retailrocket clickstream dataset.
- Real time recommendations with implicit ratings and multimodal events (3 categories) using the pattern mining/ association rules technique over the Retailrocket clickstream dataset.

All these experiments were conducted on a subset of the whole datasets with a focus on a fixed number of users and items, to cope with the very limited resources available for testing the system.

Since we are dealing with an incremental machine learning model, the performance evolution over time is a very significant metric to track, so the results have been measured at different stages of the simulation execution, with respect to the processed percentage of the input dataset (25%, 50%, 100%).

Finally, to give more context and get more insights into the model performances, the results have been compared with simple recommendations generated by an offline popular item model, which works by simply recommending to each user the most popular items at any given point in time.

When interpreting the results, it's important to remember that streaming machine learning models are meant to sacrifice some accuracy with respect to offline ones in order to produce such low latency predictions, so it's expected to obtain a lower AUC score with respect to offline models and this is accepted because in highly dynamic context recommendation speed is more important than accuracy.

All the results in the table below refer to values obtained by the best performing model, obtained after finetuning the matrix factorization hyperparameters and obtaining their best values, which are:

- Learning rate Gamma: 0.1
- Lambda: 1
- Alpha: 4

	Omniverse Dataset (Unimodal)	**Retailtrocket Dataset (Multimodal WI)**	**Retailtrocket Dataset (Multimodal AS)**
AUC 25%	0.684	0.842	0.814
POP AUC 25%	0.952	0.921	0.921
AUC 50%	0.398	0.867	0.890
POP AUC 50%	0.939	0.944	0.944
AUC 100%	0.572	0.808	0.833
POP AUC 100%	0.911	0.853	0.853

Table 4.3: Experimental Results

5. Conclusion and Future Developments

5.1.1 Results Discussion

Let's have a few words regarding the outcomes of the experiments presented in the last chapter:

- The main objective of this thesis was proving the feasibility of real time matrix factorization models using clickstream data and explore pattern mining as a way to produce implicit feedback in recommender systems, both of these objectives have been completely reached in the experiments.
- The outcomes of the experiments were aligned with the expectations: real time recommendations with clickstream data are possible, it's possible to obtain accurate predictions (even though not as precise as their offline counterpart), and the more detailed and enriched the input data is (i.e. how many types of user to item interaction are available in the clickstream tracking), the more effective the recommender system will be, as proven by the fact that our model performed better on the Retailrocket multimodal dataset with respect to the Cloudera Omniture single-modal one.
- Multimodal feedback for recommendations is a very promising field of research and with yet much possible room for improvement, the richer event types set offers the possibility of improving the recommendations performance by conducting enrichment analyses on the clickstream sequences, such as association rules mining in our case.
- The whole rationale behind stream processing and real time machine learning is to sacrifice a bit of accuracy to have a huge gain in latency time, which could more than compensate for the loss of performance. Therefore, the results obtained in the experiments are very significant even if may not seem like it at first glance, given that they provide a lower accuracy than the offline method.

5.1.2 Large-Scale Testing and Deployment

Every system designed to work with a large user base needs to handle scaling and guarantee a high level of latency and/or throughput.

While designing the algorithms presented in this thesis and developing the system architecture, they were tested on making sure that they could scale and work in a real life was a priority requirement, and all the choices like adopting an horizontally scalable real time messaging system like Kafka and run the algorithm with a distributed processing framework were meant to enable deployment in high scale scenarios.

However, the limited resources at the disposal for this thesis made it impossible to test the scaling capabilities of the system in a big, distributed cluster with a large number of nodes, allowing only for a local deployment with a few nodes and testing only on a subset of the whole datasets and simulating the real time interactions with a stub.

An interesting possibility for future work would be having the chance of deploying the system in a real-world, large-scale system connecting it with actual users generating data in real time and testing the actual effect that the online recommendations have on the website applying techniques such as A/B testing.

5.1.3 Increase the Amount of Interaction Categories

In this thesis's experiments we have initially tested the recommender systems with a unimodal interaction dataset, and then we moved on to a multimodal with 3 different categories of interactions.

When doing this kind of machine learning research work one of the most difficult challenges is looking for and finding rich and suitable datasets that can be used to properly exhaust and properly analyze the potential of the algorithm being proposed, since the necessary kind of data is usually possessed only by large corporations who don't grant open access to it.

This case was no exception: even though a great amount of time was spent in researching and looking for clickstream datasets with multimodal interactions, the best candidate found was the Retailrocket one, who offers only 3 different kinds of interaction.

Even though 3 interaction types still enabled us to test and verify the capabilities of the proposed approach, one of the main purposes of this thesis was to map and match complex event sequences into user behaviors to find similar navigation pattern to feed to our recommender systems, and these user behavior representation gets more and more accurate when increasing the amount and scope of interactions considered [9].

A possible future progression of this work could focus on repeating the experiments done on a bigger and richer dataset and analyze the impact the number of inputs can have on the effectiveness of the system.

5.1.4 Real Time navigation pattern mining

In this experiment the focus was on verifying and testing the potential of pattern matching techniques applied to collaborative filtering systems working with real time data.

To cope with the limited time and scope of this thesis, during the experiments the assumption was made that the user behaviors and pattern represented in the system with the association rules are static and don't evolve over time.

This assumption allowed to simplify the data architecture and rely on stream (real time user navigation) to static dataset (precomputed association rules) joins, which still allowed to reach the objective of testing pattern-based recommendations in a real time scenario.

A more realistic and effective approach could however focus on handling also the pattern recognition section of the architecture in a real-time fashion. This would make sense since not only singular user interests, but also navigational patterns are also always continuously evolving and should be updated to have the most realistic model of current website usage.

A simple solution to this problem would be to periodically (at least once a day, more effectively a few times a day) recompute the association rules with the updated clickstream data, and then integrate the updated pattern dataset into the data architecture.

This approach would allow to have updated pattern data while still relying on the association rules technique for detecting user navigation patterns.

At the moment there aren't many models for computing association rules or even performing more general pattern detection with streaming data, if in the future there will be any development in this sense it could be very promising to integrate real-time pattern detection in the recommender system architecture proposed in this thesis.

5.1.5 Real Time User-Item List

Another simplification that was necessary to cope with the limited resources allowed for the project was selecting a fixed subset of user and items to be considered during the online model training and recommendation generation.

This not only reduced the scale of the data making it possible for all processing to be executed on a single multi-core machine, but it also simplified the algorithm itself since the rating matrix could be considered having a fixed size and it wasn't necessary to take into account the newly generated users and items that would be continuously added in a real-word production website.

There are a few different possible approaches to deal with the real-time integration of new user and items into the system, that could be exploited to make the research performed in this thesis closer to a real-word ready system:

- Rating Matrix and Factors with Dynamic size: it would be possible to increase the size of the operational matrixes in real time each time a new user starts interacting or an item is interacted with for the first time, initializing the new cells with values at random or derived with local functions from their neighbors.
  This solution would probably be expensive to be done in Spark Streaming, since RDDs are by nature immutable, so more specific optimization tailored to the specific use-cases would be necessary (like request micro batching).

- Rolling Feature Matrix Techniques: another very promising research thread that could provide a great and effective solution to this problem is the rollup matrix technique [20], which works by keeping a fixed size matrix in main memory that stores only the most recently requested item inside while the rest of the data is kept on a slower storage, implementing a sort of caching system that allows to have a great performance and latency while at the same time keeping the data size manageable.

These are just a couple of examples of possible techniques that could apply the core methodology of the proposed algorithms in the thesis to make them compatible with an ever-growing user and itemset, required for a deployment in a real website.

5.1.6 User Interest Drift

An important aspect in recommender systems that was out of the scope for this thesis was the user interest drift concept.

User preferences are not static and evolve over time, this concept is already natively included the system design of the approach proposed in this thesis since interests are up updated progressively in real time as soon as each user performs a new interaction with an item, but many existing systems [6, 8, 23] also introduce a rating decay factor, meaning that older values will get less and less significant in the preference estimation over times while newer inputs will hold a greater weight.

In recent years many different approaches have been proposed to introduce this concept in recommender systems, more simple and classical systems leverage a factor decay functions such as the exponential rating decay function, a classical solution works by giving each rating a fixed number of days after which that rating no longer holds any importance in the preference estimation. An implementation of this solution is the half-life decaying function, which works by defining a parameter h corresponding to the number of days after which a rating will have half of its original weight [7] :

$$w_{u,i} = \exp\left(\frac{\ln 0.5}{h}\left(t_{\mathrm{n}} - t_{u,i}\right)\right)$$

Where $w_{u,i}$ is the weight of the rating of the user u to the item i, t_{n} is the current time and $t_{u,i}$ is the production time of the rating.

More recent research threads have focused on integrating time dependency into collaborative filtering methods, and work by building a time series of rating matrixes and latent vectors, building in this way effectively a temporal matrix factorization algorithm.

Integrating this temporal dynamic into our proposed real time matrix factorization algorithm would indeed be the most relevant and natural progression of the research performed in this thesis, in this paragraph we have gone over the most recent and promising approaches for tackling this issue, all of which could possibly be adapted to a streaming processing scenario. The temporal matrix factorization model in particular appears to be very closely related to our use case, and integration with the algorithm proposed in this thesis could possibly be a very promising research thread for improving the state of real-time distributed systems for recommendations.

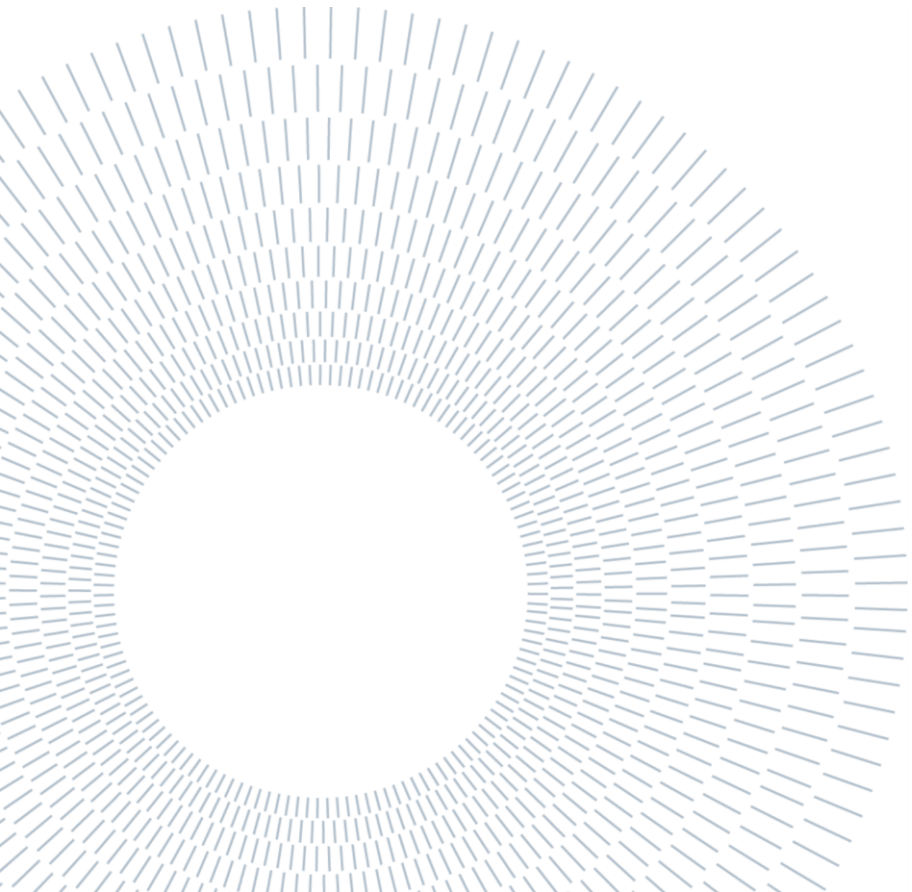
5.1.7 Conclusion

The main objective of the thesis was to propose an alternative approach to handle real time training for distributed collaborative filtering systems and test out innovative approaches to include multimodal user-item interactions and pattern matching in the process.

All the innovative processes were deployed in a local environment using datasets taken from real life data sources and tested using stubs that simulate a realistic data ingestion environment.

While there are still many limitations and possible improvements to be made that were explained in this final chapter, the main objective of the thesis has been reached as it has been proven that these new approaches yield good results and can offer accurate recommendations with a near real time latency.

The conclusion of this thesis can then set a new improvement in this field of research, and it can be a good starting point for future advancements in this area.



Bibliography

[1] Hu, Yifan & Koren, Yehuda & Volinsky, Chris. (2008). Collaborative Filtering for Implicit Feedback Datasets. Proceedings - IEEE International Conference on Data Mining, ICDM. 263-272. 10.1109/ICDM.2008.22.

[2] Liu, Zhuoran, et al. "Monolith: Real Time Recommendation System With Collisionless Embedding Table." arXiv preprint arXiv:2209.07663 (2022).

[3] Nkandu, Jeff. "Building and Evaluating an Adaptive Real-time Recommender System." (2014).

[4] Manuel Pozo, Raja Chiky. An implementation of a Distributed Stochastic Gradient Descent for Recommender Systems based on Map-Reduce. INTERNATIONAL WORKSHOP ON COMPUTATIONAL INTELLIGENCE FOR MULTIMEDIA UNDERSTANDING (IWCIM), Oct 2015, Prague, Czech Republic. pp.1-5, ff10.1109/IWCIM.2015.7347074ff. ffhal-01314906f.

[5] R. Vieira, A Streaming ALS Implementation, https://ruivieira.dev/a-streaming-als-implementation.html#27WaEIe-TX:0:U

[6] Y. -Y. Lo, W. Liao, C. -S. Chang and Y. -C. Lee, "Temporal Matrix Factorization for Tracking Concept Drift in Individual User Preferences," in IEEE Transactions on Computational Social Systems, vol. 5, no. 1, pp. 156-168, March 2018, doi: 10.1109/TCSS.2017.2772295.

[7] Ardagelou, Panagiotis & Arampatzis, Avi. (2017). A Half-Life Decaying Model for Recommender Systems with Matrix Factorization. 10.13140/RG.2.2.28296.93449.

[8] Ma, Shanle et al. "A Recommender System with Interest-Drifting." WISE (2007).

[9] Peska, Ladislav & Vojtáš, Peter. (2012). Evaluating Various Implicit Factors in E-commerce. CEUR Workshop Proceedings. 910.

[10] Vinagre, João & Jorge, Alípio & Gama, João. (2014). Evaluation of recommender systems in streaming environments. 10.13140/2.1.4381.5367.

[11] Gama, João & Sebastião, Raquel & Rodrigues, Pedro. (2013). On evaluating stream learning algorithms. Machine Learning. 90. 317-346. 10.1007/s10994-012-5320-9.

[12] Shankar, Shreya & Herman, Bernease & Parameswaran, Aditya. (2022). Rethinking Streaming Machine Learning Evaluation. 10.48550/arXiv.2205.11473.

[13] Gama, João & Sebastian, Raquel & Rodrigues, Pedro. (2009). Issues in evaluation of stream learning algorithms. Issues in evaluation of stream learnings algorithms. 329-338. 10.1145/1557019.1557060.

[14] Huyen, Chip. Machine Learning is going real-time. https://huyenchip.com/2020/12/27/real-time-machine-learning.html

[15] Huyen, Chip. Real-time machine learning: challenges and solutions. https://huyenchip.com/2022/01/02/real-time-machine-learning-challenges-and-solutions.html

[16] Dang, Tran & Nguyen, Quang & Nguyen, Sinh. (2019). Evaluating Session-Based Recommendation Approaches on Datasets from Different Domains. 10.1007/978-3-030-35653-8_37.

[17] Pal, Gautam & Atkinson, Katie & Li, Gangmin. (2021). Real-time user clickstream behavior analysis based on apache storm streaming. Electronic Commerce Research. 10.1007/s10660-021-09518-4.

[18] Wang, Shoujin & Cao, Longbing & Wang, Yan. (2019). A Survey on Session-based Recommender Systems.

[19] Baltrunas, Linas & Amatriain, Xavier. (2009). Towards time-dependant recommendation based on implicit feedback. Proceedings of the Third ACM Conference on Recommender Systems.

[20] Gregory Arefyev, Real-time Recommendation System: Rolling Feature Matrix, https://towardsdatascience.com/real-time-recommendation-system-rolling-feature-matrix-f5ca701439df

[21] Jesse Steinweg-Woods, A Gentle Introduction to Recommender Systems with Implicit Feedback, https://nbviewer.org/github/jmsteinw/Notebooks/blob/master/RecEngine_NB.ipynb

[22] Akib, Md. Tanjim-Al-Akib & Ashik, Lutfullahil & Shaiket, Hosne Al Walid & Chowdhury, Krishanu. (2016). User-modeling and recommendation based on mouse-tracking for e-commerce websites. 517-523. 10.1109/ICCITECHN.2016.7860252.

[23] C. Wangwatcharakul and S. Wongthanavasu, "Dynamic Collaborative Filtering Based on User Preference Drift and Topic Evolution," in IEEE Access, vol. 8, pp. 86433-86447, 2020, doi: 10.1109/ACCESS.2020.2993289.

[24] Bianchi, Federico & Tagliabue, Jacopo & Yu, Bingqing & Bigon, Luca & Greco, Ciro. (2020). Fantastic Embeddings and How to Align Them: Zero-Shot Inference in a Multi-Shop Scenario.

[25] Heidy Hazem, Ahmed Awad, Ahmed Hassan Yousef, A distributed real-time recommender system for big data streams, Ain Shams Engineering Journal, 2022.

[26] Benczúr, A.A., Kocsis, L., Pálovics, R. (2019). Online Machine Learning Algorithms over Data Streams. In: Sakr, S., Zomaya, A.Y. (eds) Encyclopedia of Big Data Technologies. Springer, Cham. https://doi.org/10.1007/978-3-319-77525-8_329

[27] Rawat, Mayank & Goyal, Neha & Singh, Soumya. (2017). Advancement of recommender system based on clickstream data using gradient boosting and random forest classifiers. 1-6. 10.1109/ICCCNT.2017.8204029.

[28] Lăpușan, Tudor, Hadoop MapReduce deep diving and tuning, https://www.todaysoftmag.com/article/1358/hadoop-mapreduce-deep-diving-and-tuning

[29] ALS Implicit Collaborative Filtering, https://medium.com/radon-dev/als-implicit-collaborative-filtering-5ed653ba39fe

[30] Sharma, Nikita. (2019). Recommender Systems with Python — Part III: Collaborative Filtering (Singular Value Decomposition).

https://heartbeat.comet.ml/recommender-systems-with-python-part-iii-collaborative-filtering-singular-value-decomposition-5b5dcb3f242b

[31] Apache Kafka Architecture and Its Components-The A-Z Guide. https://www.projectpro.io/article/apache-kafka-architecture-/442

[32] Tzoumas, Kostas. (2016). Counting in streams: A hierarchy of needs. https://www.ververica.com/blog/counting-in-streams-a-hierarchy-of-needs

[33] Bashar, Shafi. Gilmor, Alex. (2018). Scaling Collaborative Filtering with PySpark. https://engineeringblog.yelp.com/2018/05/scaling-collaborative-filtering-with-pyspark.html

[34] Luo, Shuyu. (2010). Introduction to Recommender Systems. https://towardsdatascience.com/intro-to-recommender-system-collaborative-filtering-64a238194a26

[35] Terzi, Evimaria. Mining Association Rules in Large Databased.

https://cs-people.bu.edu/evimaria/cs565-11/lect2.pdf

[36] The Beginners' Guide to the ROC Curve and AUC.

https://towardsai.net/p/machine-learning/the-beginners-guide-to-the-roc-curve-and-auc

[37] Shahrivari, Saeed. (2014). Beyond Batch Processing: Towards Real-Time and Streaming Big Data. Computers. 3. 10.3390/computers3040117.

[38] Huang, Y., Cui, B., Zhang, W., Jiang, J., & Xu, Y. (2015). TencentRec: Real-time Stream Recommendation in Practice. Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data.

[39] Cloudera Omniture Log dataset. https://github.com/iflubber/clickstream-analysis

[40] Retailrocket recommender system dataset. https://www.kaggle.com/datasets/retailrocket/ecommerce-dataset

[41] Han, J., Pei, J., Yin, Y. et al. Mining Frequent Patterns without Candidate Generation: A Frequent-Pattern Tree Approach. Data Mining and Knowledge Discovery 8, 53–87 (2004). https://doi.org/10.1023/B:DAMI.0000005258.31418.83

[42] Borgelt, Christian. (2010). An Implementation of the FP-growth Algorithm. Proceedings of the 1st International Workshop on Open Source Data Mining: Frequent Pattern Mining Implementations. 10.1145/1133905.1133907.

[43] Sharma, Anuj. How to store and real time analysis of clickstream data? (2010).

https://medium.com/analytics-vidhya/how-to-store-and-real-time-analysis-of-clickstream-data-e8460467aa88

List of Figures

List of Tables

List of Algorithms

2. Acknowledgements

My biggest gratitude goes to all the people who have been on my side during this thesis and during my study years.

Especially, I would like to thank with all my heart my parents and my sister, who have always proven themselves to be the best family anyone could ever ask for. I have lost count of how many times they have gone above and beyond to support me.

I also owe a huge debt of gratitude to my friends Federico and Jian, who have been a constant presence in my life during these last 5 years and probably have no idea how much help they have given me in going through the difficult challenges every university student needs to face in his academic career.

Finally, I would also like to sincerely thank everyone who helped me during the completion of this thesis with technical advice, especially Professor Della Valle and my colleagues in JAKALA.

I am infinitely happy and proud to have reached the conclusion of this long growing and learning path and I couldn't have made it without you.

Thank you,

Alessandro Messori