TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

JONNE PETTERI PIHLANEN

BUILDING A RECOMMENDATION ENGINE WITH APACHE SPARK

Master of Science thesis

Examiner: ????
Examiner and topic approved by the
Faculty Council of the Faculty of
xxxx
on 1st September 2014

ABSTRACT

JONNE PETTERI PIHLANEN: Building a Recommendation Engine with Apache Spark
Tampere University of Technology
Master of Science thesis, xx pages
September 2016
Master's Degree Program in Signal Processing
Major: Data Engineering
Examiner: ????
Keywords:

The amount of recommendation engines around the Internet is constantly growing.

This paper studies the usage of Apache Spark when building a recommendation engine.

TIIVISTELMÄ

JONNE PETTERI PIHLANEN: Suosittelijajärjestelmän rakentaminen Apache Sparkilla
Tampereen teknillinen yliopisto
Diplomityö, xx sivua
syyskuu 2016
Signaalinkäsittelyn koulutusohjelma
Pääaine: Data Engineering
Tarkastajat: ????
Avainsanat:

PREFACE

ASDASDASDASDASD

Tampere,

Jonne Pihlanen

TABLE OF CONTENTS

LIST OF ABBREVIATIONS AND SYMBOLS

Recommendation Engine	System that tries to predict the items that a user woud like
Collaborative	Users collaborate with each other to recommend items
Spark	Fast and general engine for large-scale data processing
Information retrieval (IR)	Activity of obtaining relevant information resources from a collection of information resources.
SDK	Software Development Kit

1. INTRODUCTION

Recommender systems have been successfully utilized to aid customers in decision making. In fact, they are constantly present in our everyday life. Whether a customer is shopping online, watching a movie from Netflix, browsing the Facebook or simply reading the news. All of these tasks involve a presence of a recommendation engine. Basically all parts of our daily life include recommendations of some sorts. However, the most basic type of recommendation is the one from human to human and happens completely without computers. However, humans can only recommend effectively those items they have personally experienced. This is where recommendation systems (RSs) become useful as they can potentially offer recommendations from thousands of different items.

Recommendation can be divided into two major categories: item-based recommendation and user-based recommendation. In item-based recommendation the idea is to search similar items, since the user could prefer same kind of items also in the future. In user-based recommendation the user is thought to be interested in items purchased by similar users thus trying to finds similar users to be able to offer items bought by them.

Apache Spark is a framework for building distributed programs. A distributed program denotes that the execution of the program is divided between a number of processing nodes. Recommendation problem can be modeled as an distributed program in which two matrices, users and items, are processed with an iterative algorithm that can be run in parallel.

Spark is built with Scala, a general-purpose, multi paradigm programming language that provides support for functional programming and a strong static type system. We will be using Scala in the implementation so a short introduction on the programming language will also be provided.

This thesis is structured as follows. Chapter 2 describes recommendation systems.

Chapter 3 discusses Apache Spark, an open source framework for building distributed programs. Chapter 4 presents the implementation. In Chapter 5, we go through the results. Finally, in Chapter 6 the evaluation is presented along with conclusions.

2. RECOMMENDATION SYSTEMS

Recommendation denotes a task in which items are recommended to users. The most simple version of recommendation is from peer to peer, completely without computers. However, humans can only recommend effectively those items they have personally experienced. This is where recommendation systems (RSs) become useful as they can potentially offer recommendations from thousands of different items. Recommender systems are a set of techniques and software tools that provide suggestions to users about potentially useful items. Item denotes the general subject that the system recommends to users. Recommendation systems are usually intended to recommend only specific kind of items such as books or movies. [10]

Recommendation systems are often targeted to aid customers in decision making when the amount of all items is overwhelming. Usually the recommendations are personalized, which implies that the recommendations are different for different users or groups of users. Recommendations can also be non-personalized and usually these are much simpler to generate. A simple ranked top-ten list of items is an example of non-personalized recommendation. Ranking is done by predicting the most suitable items based on the user's preferences and constraints. To complete this task, RS needs to collect these preferences from users. They might be explicitly expressed ratings for products or interpreted from user actions such as clicks and page views. For example, RS could consider navigation to a product page as an implicit evidence of preference for the items in that page. [10]

Development of RSs initiated from a rather simple observation: people usually tend to trust recommendations provided by others in making routine decisions. For example, it is common to rely on peer feedback when selecting a book to read or rely on the reviews the movie critics have writtenFirst RSs tried to mimic this behavior by applying algorithms to leverage recommendations from the community to an active user looking for recommendations. Recommendations were for users with similar tastes. This approach is essentially collaborative-filtering and the idea

behind it is that if users liked similar items in the past, they are likely to consider recommendations from these same users relevant. [10]

Along with the development of e-commerce sites, a need for recommendations emerged to limit the amount of alternatives. Users were finding it more and more difficult to find the correct choices from the vast range of items the sites were offering. As the variety of information in the Web has explosively grown constantly overwhelmed users, it has led them to make poor decisions. The availability of choices had begun to decrease users well-being, instead of producing benefit. While choice is good, more choice is not always better. [10]

Recently, RSs have proved to be an effective cure for the information overload problem at hand. RSs addresses this phenomenon by offering new, previously unknown items that are likely to be relevant to the current task of the user. When a user requests for recommendations, RSs generate recommendations by using knowledge and data about the users, available items and previous transactions present in the recommender database. After browsing the provided recommendations, she may accept them or not thus providing implicit and explicit feedback to the recommender. This new information can later on be used when generating new recommendations upon the next user-system interactions. [10]

Compared to research in the classical fields of information systems such as databases and search engines, study of recommender systems is relatively new. They became an independent research area in the mid-1990s. Recently, the interest towards recommendation systems has dramatically increased. Evidence can be found for example from such highly rated Internet sites as Amazon.com, YouTube, Netflix and IMDB in which RSs are playing an important role. Another example are conferences and journal issues dedicated to RS research and development such as RecSys and AI Communications (2008), consecutively. [10]

Two primary purposes exist for a RS: first is to aid the service provider somehow and the second is to produce value to the user of the service. Thus, a RS must balance between the needs of service provider and the customer [10]. For example, a service provider might introduce a RS to get help in tasks such as increasing the number of items sold, selling more diverse items, increasing user satisfaction, increasing user loyalty or understanding better what the user wants [10]. In addition, a number of different functions exist that the user might expect a recommendation system to offer. Some of these are similar to the ones mentioned above, thus considered as

core functions. For instance, a user may want a RS to recommend a sequence of items, aid in browsing or enable user to influence others users. Influencing might also be malicious. [10]

GroupLens, BookLens and MovieLens were pioneers in RSs. Furthermore, they also released data sets which, aside of recommendation, was also pioneering in the field. Data sets were not that common for benchmarking or just trying out new technologies. [2]

2.1 Recommendation techniques

Recommendation system must have some sort of understanding about the items to be able to recommend something. To achieve its goal, the system must be able to predict the usefulness or at least compare the utility of some items and then decide the ones to recommend. The prediction step of the recommender can be illustrated by, for example, a simple non-personalized, recommendation algorithm that recommends only the most popular movies. The reasoning behind this approach is that when lacking more precise information about the user's preferences, a movie liked by others is probably also liked by a general user, at least more than a randomly selected movie. Thus, the utility of these popular songs can be seen reasonably high for a generic user. [10]

The utility of the user u for the item i can be modeled as a real valued function $R(u, i)$ as is normally done in collaborative filtering by considering the ratings of items given by the users. Thus, the fundamental task of the collaborative filtering recommender is to predict the value of R over pairs of users and items to compute an estimation for the true function R. Consequently, by computing this prediction for the user u on a set of items, the system will recommend items with the largest predicted degree of utility. The amount of predicted items is usually much smaller than the whole amount of items, thus the RS is filtering the items that are recommended to users. [10]

RSs vary in terms of the addressed domain, the knowledge used and especially how the recommendations are made, denoting the recommendation algorithm [10]. This thesis will concentrate only on one class of recommendation techniques, collaborative filtering, since that is the one used in Apache Spark's MLlib.

Collaborative Filtering Recommendation Systems are based on the collaboration of

users. They aim at identifying patterns of user interests in order to make targeted recommendations [1]. The original implementation of this approach recommends to the active user those items that other similar users in the sense of tastes have liked in the past [10]. First a user provides ratings for items. Next the method will find recommendations based on other users that have purchased similar items or based on items that are the most similar to the user's purchases. Collaborative filtering can be divided into two sub categories which are item-based collaborative filtering and user-based collaborative filtering. Collaborative filtering has been studied the most thus being the most popular and widely implemented technique in recommendation systems [6] [10] [3].

Collaborative filtering analyzes relationships between users and interdependencies among products to identify new user-item associations. [7] For example, deciding that two users may both like the same song because they play many other same songs is an example of collaborative filtering. [11]

Collaborative filtering algorithms suffer from the new user and new item problems [6]. This originates to the fact that the recommendation is based only on user's recommendations on items. If user has not given any reviews, the algorithm is not able to produce any recommendations either. Other issues of collaborative filtering algorithms are cold start and sparsity. Cold start denotes that a relatively large amount of data is required in order to be able to provide accurate recommendations for a user. Sparsity means that the number of items typically exceeds the number of users. This makes the relations extremely sparse since most users have rated or purchased only a small subset of the total items. [1]

2.1.1 Memory-based collaborative filtering

Memory-based or neighborhood collaborative filtering

In memory-based methods the user-item ratings stored in the system are directly accessed to predict ratings for new items. This can be done in two ways known as user-based or item-based recommendation.

The following sections describe user-based collaborative filtering and item-based collaborative filtering.

Item-based collaborative filtering

Item-based collaborative filtering (IBCF) starts by finding similar items from the user's purchases [6]. Next step is to model the preferences of a user to an item based on ratings of similar items by the same user [10]. The following snippet presents the idea in IBCF for every new user.

Program 2.1 *Item-Based Collaborative Filtering algorithm [6]*

```
1. For each two items, measure how similar they are in terms of having received
    similar ratings by similar users

val similarItems = items.foreach { item1 =>
items.foreach { item2 =>
val similarity = cosineSimilarity(item1, item2);
}
}

2. For each item, identify the k−most similar items

val itemsSorted = sort(similarItems)

3. For each user, identify the items that are most similar to the user's
    purchases

users.foreach { user =>
user.purchases.foreach { purchase =>
val mostSimilar = findSimilarItem(purchase)
}
}
```

Amazon.com, the biggest Internet retailer in the United States, has previously been using item-to-item collaborative filtering method. In their implementation the algorithm builds a table containing similar items by finding ones that users tend to purchase together. The algorithm then finds items similar to each of the user's purchases and ratings, combines those items, and returns the most popular or correlated items. [8]

User-based collaborative filtering

User-based collaborative filtering (UBCF) starts by finding the most similar users, rate items purchased by similar users, pick top rated items. The similarity in taste of two users is calculated based on the similarity in the rating history of the users [10].

The steps for every new user in user-based collaborative filtering are as follows:

Program 2.2 *User-Based Collaborative Filtering algorithm [6]*

```
1. Measure how similar each user is to the new one. Like IBCF, popular
    similarity measures are correlation and cosine.

case class Similarity(userId1: Int, userId2: Int, score: Int)

val newUser: User = User("Adam", 31, purchases)
val similarities = users.map { user =>
Similarity(newUser.id, user.id, cosineSimilarity(user, newUser)
}

2. Identify the most similar users. The options are: Take account of the top k
    users (k-nearest_neighbors) Take account of the users whose similarity is
    above a defined threshold

val mostSimilarUsers = similarities. filter (_.score > 0.8)

3. Rate the items purchased by the most similar users. The rating is the average
     rating among similar users and the approaches are:
Average rating
Weighted average rating, using the similarities as weights

val ratedItems = mostSimilarUsers.map { user =>
user.purchases.map { purchase =>
val purchases = mostSimilarUsers.map { usr =>
usr.purchases. filter (_.id === purchase.id)
}
```

```
purchases.sum() / purchases.size
}
}

4. Pick the top−rated items.

val topRatedItems = ratedItems.take(10)
```

2.1.2 Model-based collaborative filtering

In contrast to memory-based systems, which use the stored ratings directly in the prediction, model-based approaches use these ratings to learn a predictive model. The general idea is to model the user-item interactions with factors representing latent characteristics of the users and items in the system, like the preference class of users and the category class of items. This model is then trained using the available data, and later used to predict ratings of users for new items. [10]

Alternating Least Squares is an example of model-based collaborative filtering algorithm, it is presented in the next chapter.

3. APACHE SPARK

Apache Spark is an open source framework that combines an engine for distributing programs across clusters of machines with an elegant model for writing programs. [11] Spark provides high-level APIs in Java, Scala, Python and R.

At a high level, every Spark application consists of a driver and one or more executors. Driver is a program that runs the user's main function and executes various parallel operations on a cluster. An executor is one machine in a cluster.

Spark can be introduced by describing its predecessor, MapReduce, and the advantages it offers. MapReduce offered a simple model for writing programs that could execute in parallel across hundreds of machines. MapReduce achieves nearly linear scalability as the data size increases. The execution time is maintained by adding more computers to handle the task.

Spark preserves MapReduce's linear scalability and fault tolerance while extending it in three important ways. In MapReduce the intermediate results between the map and reduce tasks must be written into memory where as Spark is able to pass the results directly to the next step in the pipeline. Spark also treats the developers better by offering a rich set of transformations which enables users to represent complex pipelines in a few lines of code. (EXAMPLE?) Spark also introduces in-memory processing by introducing Resilient Distributed Dataset (RDD) abstraction which offers a way for developers to materialize any step in a processing pipeline and store it into memory. This means that future steps do not need to calculate the previous results again. Previously this kind of feature has not been available within distributed processing engines. [11]

Spark programs can be written using Java, Scala, Python or R. However, using Spark with Scala instead of Java, Python or R has a couple of advantages to it. Performance overhead is reduced, since tasks such as transferring data across different layers or performing transformations for data may result in weaker performance.

Spark is written with Scala, which denotes that user has always access to latest and greatest features of the framework. Spark philosophy is easier to understand when Spark is used with the language it was built with. There is still one, maybe the biggest benefit of using Scala with Spark, and it is the developing experience that comes with the fact that user is using the same language for everything. Importing data from database, data manipulation, shipping the code into clusters. [11]

Spark is shipped with a read eval print loop (REPL), which enables developers to test out things quickly in the console, without having to make the application self contained from the begin. Usually when a application developed in REPL has matured enough, it is a good idea to move it into a compiled library (JAR). This way it is possible to prevent code and results from disappearing.

3.1 Resilient Distributed Dataset (RDD)

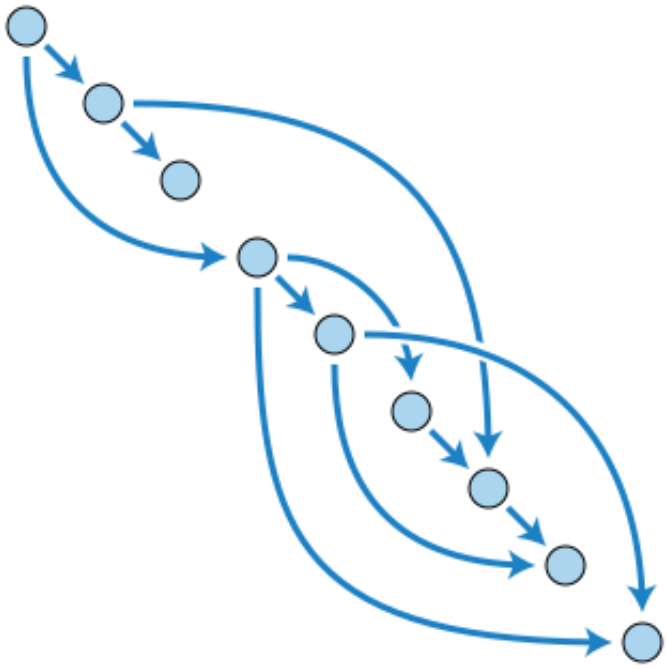
A Resilient Distributed Dataset (RDD), is the main abstraction in Spark. Essentially it is an immutable, partitioned collection of elements that can be distributed across multiple machines in a cluster. [16]

An important context to understand about RDDs is that they are lazy by nature. When a new RDD is created, nothing is actually done, it means that spark knows where the data is when the time comes to do something with it.

RDD can be created in two ways: parallelizing an existing Scala collection in the driver program or referencing an external dataset in an external storage system, such as HDFS, HBase or any file system supported by Hadoop [14].

RDDs can be persisted into memory, which allows programmer to reuse them efficiently across parallel operations. RDDs are able to recover from node failures automatically, using Directed Acyclic Graph (DAG) engine. DAG supports cyclic data flow. Each Spark job creates a DAG of task stages to be performed on the cluster. Compared to MapReduce, which creates a DAG with two predefined stages - Map and Reduce, DAGs created by Spark can contain any number of stages. This allows some jobs to complete faster than they would in MapReduce, with simpler jobs completing after just one stage, and more complex tasks completing a single run of many stages, rather than having to be split into multiple jobs. [18]

Figure 3.1 *Directed Acyclic Graph [4]*



3.2 Dataset API

Dataset, DS, is the replacement for RDD in Spark. It is a strongly typed collection of domain-specific objects that can be transformed in parallel using functional or relational operations. Operations available on Datasets are divided into transformations and actions. Transformations are operations such as map, filter, select and aggregate that produce new Datasets. Actions are operations such as count, show, or writing data out to file systems that trigger computation and return results. [15]

Datasets are lazy by nature, which refers to that computations are only triggered when an action is invoked. A Dataset essentially represents a logical plan that describes the computation required to produce the data. Upon an action invocation, the query optimizer of Spark optimizes the logical plan and generates a physical plan for efficient execution in a parallel and distributed manner. The logical plan as well as the optimized physical plan can be explored by using the explain function. [15]

To efficiently support domain-specific objects, an Encoder is required. The encoder maps the domain specific type T to Spark's internal type system. For example, given a class Person with two fields, name (string) and age (int), an encoder is used to tell Spark to generate code at runtime to serialize the Person object into a binary

structure. This binary structure often has much lower memory footprint as well as are optimized for efficiency in data processing (e.g. in a columnar format). To understand the internal binary representation for data, use the schema function.

Two ways typically exist to create a Dataset. The most common way is to make use of the read function provided by SparkSession and point Spark to some files on storage systems:

Program 3.1 *Creating a new Dataset by using read function*

```
val people = spark.read.json("./people.json").as[Person]
```

where *Person* would be a Scala case class, for example:

Program 3.2 *Definition of case class Person*

```
case class Person(id: BigInt, firstName: String, lastName: String)
```

Case classes are normal Scala classes that are:

- Immutable by default
- Decomposable through pattern matching
- Compared by structural equality instead of by reference
- Succinct to instantiate and operate on

If we would omit the casting with keyword *as* then we would end up creating a DataFrame and the schema of the created object would be guessed by Spark.

Program 3.3 *Creating a SparkSession*

```
val spark = SparkSession
  .builder
  .appName("MovieLensALS")
  .config("spark.executor.memory", "2g")
  .getOrCreate()
```

SparkSession is the entry point to programming Spark with the Dataset and DataFrame API. In the above snippet we create a *SparkSession* by chaining calls to the builder method, which creates a SparkSession.Builder object for constructing a SparkSession. Datasets can also be created through transformations available on existing Datasets:

Program 3.4 *Creating a new Dataset through a transformation*

```
val names = people.map(_.name)
```

[15]

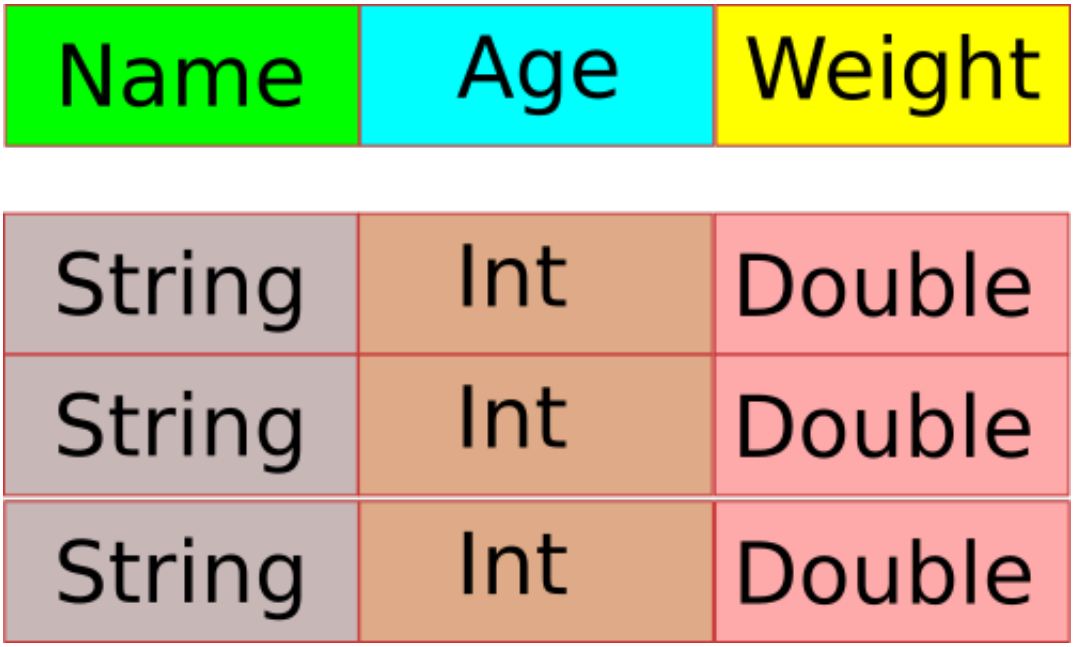
Datasets are similar to RDDs as they also provide strong typing and the ability to use powerful lambda functions [17]. These are accompanied with the benefits of Spark SQL's optimized execution engine [17]. However, instead of using standard serialization like Java serialization they use a specialized Encoder to serialize the objects. Serialization denotes a task in which an object is turned into bytes thus reducing the memory footprint of the object. In general, serialization is needed for processing or transmitting over the network. While both encoders and standard serialization are responsible for turning an object into bytes, encoders are generated dynamically by code. They are using a format that allows Spark to perform many operations such as filtering, sorting and hashing without the need of deserializing the bytes back into an object. [14]

3.3 DataFrame API

Picture about rdd vs dataset vs dataframe ?

A DataFrame is essentially a Dataset that is organized into named columns. In Scala, a DataFrame is represented by a Dataset of Rows. It is conceptually equivalent to a table in a relational database or a data frame in R/Python, but it has richer optimizations under the hood. DataFrames can be constructed from a range of sources such as structured data files, tables in Hive, external databases, or existing RDDs. [17]

Figure 3.2 *DataFrame*



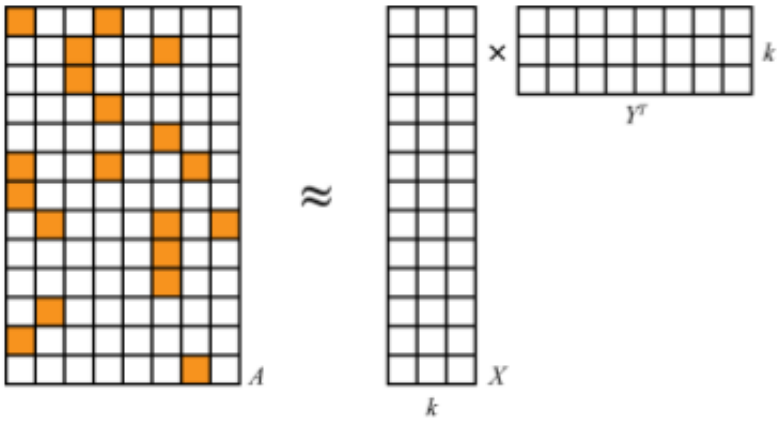
3.4 Matrix Factorization

Matrix factorization denotes a task in which a matrix is decomposed into a product of matrices. There are many different matrix decompositions. The following chapter will describe matrix factorization in general and the Alternating Least Squares algorithm which is the matrix factorization algorithm that is implemented in Spark. It is based on same idea as Netflix prize winner, matrix factorization models.

Matrix factorization belongs to a vast class of algorithms called latent-factor models. Latent-factor models try to explain observed interactions between a large number of users and products through a relatively small number of unobserved, underlying reasons. For example, they can try to explain why people would buy a particular album out of endless possibilities by describing users and albums in terms of tastes which are not directly available as data. [11] A latent factor is not available for direct observation. For example health of a human being is a latent factor. Health can not be observed as a variable such as blood pressure.

Matrix factorization algorithms treat the user and product data as if it was a large matrix A. Each entry in row i and column j represents a rating the user has given to a specific product. [11]

Figure 3.3 *Matrix factorization [11]*



Usually A is sparse, which denotes that most of the entries of A are 0. This is due to the fact that usually only a few of all the possible user-product combinations exist.

Matrix factorization models factor A as the matrix product of two smaller matrices, X and Y, which are quite tiny. Since A has many rows and columns, both of them have many rows, but both have just a few columns (k). The k columns match to the latent factors that are being used to explain the interactions of the data. The factorization can only be approximate because k is small. [11]

The standard approach to matrix factorization based collaborative filtering treats the entries in the user-product matrix as explicit preferences given by the user to the product, for example users giving ratings to movies. Implicit data denotes for example page views or a value representing if a user has listened to a artist. Explicit data means actual ratings that a user has given to a product. Spark ALS can handle both implicit and explicit data. [13] [11]

Usually many real-world use cases have access only to implicit feedback data such as views, clicks, purchases, likes or shares. However, instead of trying to model the matrix of ratings directly, the approach in Spark MLlib treats the data as numbers representing the strength of the observations such as the number of clicks, or the cumulative duration someone spent viewing a movie. Instead of explicit ratings, these numbers are related to the level of confidence in observed user preferences. Based on this data, the model tries to find latent factors that can be used to predict the expected preference of a user for an item. [13]

Sometimes these algorithms are referred to as matrix completion algorithms. This

is because the original matrix A may be sparse while the product XY^T is dense. Hence, the product is only an approximation of A. [11]

3.4.1 Alternating Least Squares (ALS)

Collaborative filtering is commonly used for recommender systems. These techniques aim to fill in the missing entries of a user-item association matrix. Spark MLlib currently supports model-based collaborative filtering, in which users and products are described by a small set of latent factors that can be used to predict missing entries. Spark MLlib uses the Alternating Least Squares (ALS) algorithm to learn these latent factors. [13]

Spark ALS attempts to estimate the ratings matrix A as the product of two lower-rank matrices, X and Y. [12]

$$A = XY^T \tag{3.1}$$

Typically these approximations are referred to as factor matrices. The general approach is iterative. During each iteration, one of the factor matrices is held constant, while the other is solved for using least squares. The newly-solved factor matrix is then held constant while solving for the other factor matrix. [12] Spark ALS enables massive parallelization since it can be done separately, it can be done in parallel which is an excellent feature for a large-scale computation algorithm. [11]

Spark ALS is a blocked implementation of the ALS factorization algorithm. Idea is to group the two sets of factors, referred to as *users* and *products*, into blocks. Grouping is followed by reducing communication by only sending one copy of each user vector to each product block on each iteration. Only those user feature vectors are sent that are needed by the the product blocks. Reduced communication is achieved by precomputing some information about the ratings matrix to determine the out-links of each user and in-links of each product. Out-link denotes those blocks of products that the user will contribute to. In-link refers to the feature vectors that each product receives from each user block they depend on. This allows to send only an array of feature vectors between each user block and product block. Consequently the product block will find the users' ratings and update the products based on these messages. [12]

Essentially, instead of finding the low-rank approximations to the rating matrix A, it finds the approximations for a preference matrix P where the elements of P are 1 when $r > 0$ and 0 when $r <= 0$. The ratings then act as confidence values related to strength of indicated user preferences rather than explicit ratings given to items. [12]

$$A_i Y (Y^T Y)^{-1} = X_i \tag{3.2}$$

Alternating Least Squares operates by rotating between fixing one of the unknowns u_i or v_j. While the other is fixed the other can be computed by solving the least-squares problem. This approach is useful because it turns the previous non-convex problem into a quadratic that can be solved optimally [1]. A general description of the algorithm for ALS for collaborative filtering taken from [1] is as follows:

Program 3.5 *Alternating Least Squares algorithm [1]*

1. Initialize matrix V by assigning the average rating **for** that movie

2. Fix V, solve U by minimizing the RMSE function.

3. Fix U, solve V by minimizing the RMSE function.

4. Repeat Steps 2 and 3 until convergence.

Minimizing the Root Mean Square Error RMSE function denotes a task in which line is plotted. EXPLAIN RMSE.

4. IMPLEMENTATION

- Scala? What is Scala?

GroupLens Research has collected and made available rating data sets from the MovieLens web site. The data sets were collected over various periods of time, depending on the size of the set. MovieLens ml-latest-small dataset is a collection of 100,000 ratings to 9,000 movies by 700 users. The downside of these datasets is that they will change over time thus they are not suitable for reporting research results. The current version, released 10/2016 is available in the project repository of this thesis. This dataset was chosen in order to be able to review movies that we have actually seen. MovieLens ml-latest-small dataset consists of *movies.csv* and *ratings.csv* files.

movieId	title	genres
1	Toy Story (1995)	Adventure|Animation|Children
2	Jumanji (1995)	Adventure|Children|Fantasy
3	Grumpier Old Men (1995)	Comedy|Romance
4	Waiting to Exhale (1995)	Comedy|Drama|Romance
5	Father of the Bride Part II (1995)	Comedy
6	Heat (1995)	Action|Crime|Thriller
7	Sabrina (1995)	Comedy|Romance
8	Tom and Huck (1995)	Adventure|Children
9	Sudden Death (1995)	Action
10	GoldenEye (1995)	Action|Adventure|Thriller

userId	movieId	rating	timestamp
1	31	2.5	1260759144
1	1029	3.0	1260759179
1	1061	3.0	1260759182
1	1129	2.0	1260759185
1	1172	4.0	1260759205
1	1263	2.0	1260759151
1	1287	2.0	1260759187
1	1293	2.0	1260759148
1	1339	3.5	1260759125

We used RDD based API since dataset API is not yet fully functional in collaborative filtering tasks. Loading data can be done with dataset API but the recommendation with ALS needs to be done still with RDD based API. Dataset API brings numerous improvements such as easier data loading.

4.1 MovieLensRecommendation.scala

First task, when writing a self contained spark application is to make a correct project structure and have a $< PROJECT_NAME >$ *.sbt* file which describes the dependencies of the application. A self contained spark application refers to a shippable jar (Java ARchive) file that can be distributed to a spark cluster and it contains your code and all the dependencies.

Applications can be launched on a cluster with the spark-submit. It can use all of Spark's supported cluster managers through a uniform interface so you don't have to configure your application specially for each one.

Program 4.1 *Creating assembly jar with sbt*

```
sbt package
```

Program 4.2 *Launch an application on a cluster*

```
spark-submit --class "MovieLensALS" --master local[4] movielens-recom
```

Program 4.3 *Loading ratings with RDD API*

```
val ratings = sc.textFile("ml-latest-small/ratings.csv")
  .filter(x => !isHeader("userId", x))
```

```
  .map { line =>
    val fields = line.split(",")
    (fields(3).toLong % 10, Rating(fields(0).toInt, fields(1).toInt,
  }
```

Program 4.4 *Loading ratings with Dataset API*

```
val ratings = spark.read.csv("ml-latest-small/ratings.csv")
  .filter(arr => arr(0) != "userId")
  .map { fields =>
        Rating(fields(0).asInstanceOf[String].toInt, fields(1).asInst
  }
```

5. RESULT

userId	movieId	movieName	rating
0	112897	The Expendables 3 (2014)	4.0
0	116887	Exodus: Gods and Kings (2014)	4.0
0	117529	Jurassic World (2015)	4.0
0	118696	The Hobbit: The Battle of the Five Armies (2014)	4.5
0	128520	The Wedding Ringer (2015)	4.5
0	122882	Mad Max: Fury Road (2015)	4.0
0	122886	Star Wars: Episode VII - The Force Awakens (2015)	4.5
0	131013	Get Hard (2015)	4.0
0	132796	San Andreas (2015)	3.0
0	136305	Sharknado 3: Oh Hell No! (2015)	1.0
0	136598	Vacation (2015)	4.0
0	137595	Magic Mike XXL (2015)	1.0
0	138208	The Walk (2015)	2.0
0	140523	"Visit, The (2015)"	3.5
0	146656	Creed (2015)	4.0
0	148626	"Big Short, The (2015)"	4.5
0	149532	Marco Polo: One Hundred Eyes (2015)	4.5
0	150548	Sherlock: The Abominable Bride (2016)	4.5
0	156609	Neighbors 2: Sorority Rising (2016)	3.5
0	159093	Now You See Me 2 (2016)	4.0
0	160271	Central Intelligence (2016)	4.0

Program 5.1 *Recommended movies*

```
1: Am Ende eiens viel zu kurzen Tages (Death of a superhero) (2011)
2: Prisoner of the Mountains (Kavkazsky plennik) (1996)
3: Funeral in Berlin (1966)
4: Caveman (1981)
5: Dream With the Fishes (1997)
```

6: Erik the Viking (1989)
7: Dead Man's Shoes (2004)
8: Excision (2012)
9: Mifune's Last Song (Mifunes sidste sang) (1999)
10: MaelstrÃ¶m (2000)

6. EVALUATION

We are going to evaluate our model against a non personalized recommendation built on Apache Pig. Apache Pig is a platform for analyzing large data sets. It consists of a high-level language for expressing data analysis programs coupled with infrastructure for evaluating these programs. A significant property of Pig programs is that their structure is able to handle substantial parallelization, which in turns enables them to handle very large data sets. [5]

At the present time, Pig's infrastructure layer consists of a compiler that produces sequences of Map-Reduce programs, for which large-scale parallel implementations already exist. Pig's language layer currently consists of a textual language called Pig Latin, which has the following key properties:

Ease of programming. It is trivial to achieve parallel execution of simple, "embarrassingly parallel" data analysis tasks. Complex tasks comprised of multiple interrelated data transformations are explicitly encoded as data flow sequences, making them easy to write, understand, and maintain. Optimization opportunities. The way in which tasks are encoded permits the system to optimize their execution automatically, allowing the user to focus on semantics rather than efficiency. Extensibility. Users can create their own functions to do special-purpose processing. [5]

The implementation is ported to a newer version of the MovieLens dataset but otherwise the code is reused as it is.

Using bigger dataset, providing more personal ratings could help.

6.1 Conclusion

There are a number of possible implementations for a recommendation engine. This was selected because Apache Spark could be a good tool to know in future and in addition learning Scala programming was another thing that was considered.

Existing recommendation or analytic engines should be evaluated before making a decision about the recommendation engine.

In the end the most difficult thing was to find the right approach for this task. By trial and error the right combination of technologies and an actual working example was found.

6.2 Future work

Actual parallelization?

Study MLlib again when Dataset API can be used with MLlib. -> Was studied already and Dataset API is much better. Something was failing and could not get to work. -> Could still provide examples of changed code?

Lately, a number of service providers, like Telegram and Microsoft, have started to introduce bot frameworks for their services. A bot is a web service that uses a conversational format to interact with users. Users can start conversations with the bot from any channel the bot is configured to work on. Conversations can be designed to be freeform, natural language interactions or more guided ones where the user is provided choices or actions. It is possible to utilize simple text strings or something more complex such as rich cards that contain text, images, and action buttons. [9]

Already for a long time, companies have had some sort of SMS that have been accepting feedback from customer or ordering a new data package for you mobile subscription. IRC channel bots have been around even longer. Idea is not new but now there are popular platforms for the bots. As any other idea, also a recommendation engine could be implemented in a way that it can be used via a bot. For example ElasticSearch forms a RESTful API so a bot could simply place queries against the API.

BIBLIOGRAPHY

[1] C. Aberger, "Recommender: An analysis of collaborative filtering techniques," 2014. [Online]. Available: http://cs229.stanford.edu/proj2014/Christopher%20Aberger,%20Recommender.pdf

[2] C. C. Aggarwal, *Recommender Systems*. Springer International Publishing, 2016.

[3] R. Burke, "Hybrid recommender systems: Survey and experiments," *User Modeling and User-Adapted Interaction*, vol. 12, no. 4, pp. 331–370. [Online]. Available: http://dx.doi.org/10.1023/A:1021240730564

[4] D. Eppstein. Directed acyclic graph. [Online]. Available: https://en.wikipedia.org/wiki/Directed_acyclic_graph#/media/File:Topological_Ordering.svg

[5] A. S. Foundation. (2017) Apache pig. [Online]. Available: https://pig.apache.org/

[6] S. K. Gorakala and M. Usuelli, *Building a Recommendation Engine with R*, 1st ed. Packt Publishing, 2015.

[7] Y. Koren, R. Bell, and C. Volinsky, "Matrix factorization techniques for recommender systems," 2009. [Online]. Available: https://datajobs.com/data-science-repo/Recommender-Systems-[Netflix].pdf

[8] G. Linden, B. Smith, and J. York, "Amazon.com recommendations," *IEEE INTERNET COMPUTING*, pp. 76–79, 2003. [Online]. Available: http://www.cin.ufpe.br/~idal/rs/Amazon-Recommendations.pdf

[9] Microsoft, "Bots." [Online]. Available: https://docs.botframework.com/en-us/

[10] F. Ricci, L. Rokach, B. Shapira, and P. B. Kanto, *Recommender Systems Handbook*, 1st ed. Springer, 2011.

[11] S. Ryza, U. Laserson, S. Owen, and J. Wills, *Advanced Analytics with Spark*. O'Reilly Media, Inc., 2015.

[12] Spark. (2014) ALS. [Online]. Available: http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.mllib.recommendation.ALS

[13] ——. (2014) Collaborative filtering - rdd-based api. [Online]. Available: http://spark.apache.org/docs/latest/mllib-collaborative-filtering.html

[14] ——. (2014) Spark programming guide. [Online]. Available: http://spark.apache.org/docs/latest/programming-guide.html

[15] ——. (2016) Dataset. [Online]. Available: https://spark.apache.org/docs/2.1.0/api/java/org/apache/spark/sql/Dataset.html

[16] ——. (2016) Rdd. [Online]. Available: https://spark.apache.org/docs/2.1.0/api/scala/index.html#org.apache.spark.rdd.RDD

[17] ——. (2016) Spark sql programming guide. [Online]. Available: http://spark.apache.org/docs/latest/sql-programming-guide.html

[18] M. Technologies. Apache spark. [Online]. Available: https://mapr.com/products/product-overview/apache-spark/