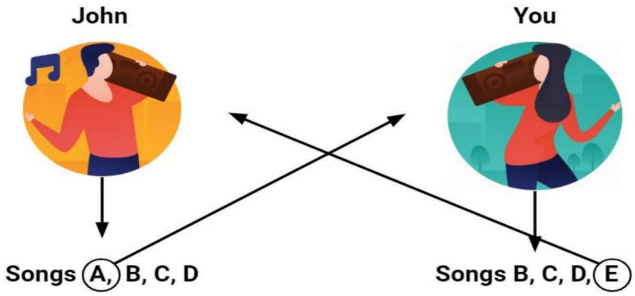
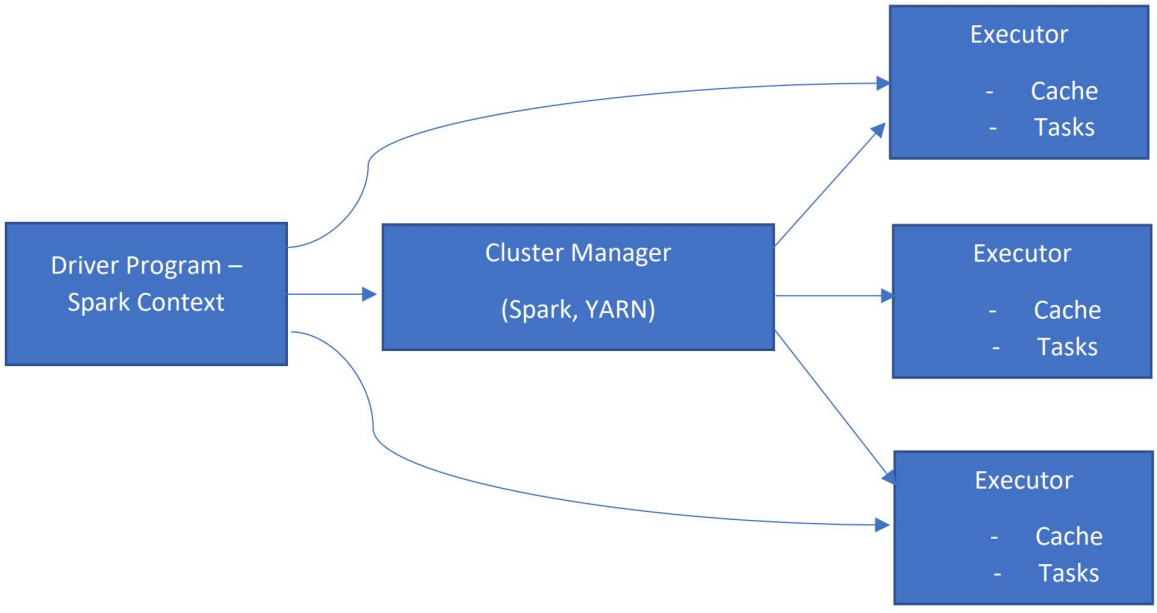
MUSIC RECOMMENDER SYSTEM WITH PYSPARK



Before diving into the project details, let us first discuss Spark.

What is Spark: according to Apache on their Web site, they say Spark is a fast and general engine for large-scale data processing. If we have a massive data set that can represent anything, weblogs, genomics data, etc. Spark can slice and dice that data up, and it can distribute the processing amongst a huge cluster of computers and take a data analysis problem that is just too big to run on one machine and divide it and conquer it by splitting it up amongst multiple machines.

Scalability: The way that it scales, it can run on top of a cluster manager, so our actual Spark scripts are just everyday scripts written in Python or Java or Scala, and they behave just like any other script, our driver program is what we call it, and it will run on our desktop or on one master node of our cluster, and it behaves just like any other script, but under the hood when we run it, Spark takes care of the work and actually farm it out to different computers on our cluster or different CPUs on our same machine even. So, Spark can run on top of different cluster managers, it has its own built-in cluster manager that we can use by default, but if we have access to a Hadoop cluster, there is a component of Hadoop called Yarn that Spark can also run-on top of to distribute work amongst a huge Hadoop cluster if we have one available



And that cluster manager will split up the work and coordinate among various executors. So, Spark will split up and create multiple executors per machine; ideally, we want one per CPU core, and it can do all the coordination using a cluster manager and our driver program itself to farm out work and distribute it to different nodes and give us fault tolerance, so if one of our executors goes down, it can recover without stopping entire job and making us start it all over.

So, the beauty of it is that it scales out to entire clusters of computers; it gives us horizontal partitioning, horizontal scalability with basically the sky is the limit. But from a user standpoint, from a developer standpoint, it is all just one simple little program running on one computer that feels a lot like writing any other script, so it is kind of a nice aspect of Spark.

Speed:

- Run programs up to 100x faster than Hadoop MapReduce in memory or 10x faster on disk.
- DAG Engine (directed acyclic graph) optimizes workflows

Why do people use Spark? Well, it has a lot in common with MapReduce really; it solves the same sorts of problems so, why are people using Spark instead of MapReduce? MapReduce has been around a lot longer, and many people know the ecosystem surrounding it, the tools surrounding it are more mature at this point. Well, one of the main reasons is that Spark is fast, and on the Apache website, they claim that Hadoop MapReduce is 100 times slower than Spark in some cases and ten times faster on disk, now that is a little bit of hyperbole, to be honest, I mean that is in a very contrived example. In many experiments done by different people, if we compare some of the same tasks that we run on Spark to the same tests we run using MapReduce, it is not a hundred times faster; it may be two to three times faster, but it is faster, so it has that going for it. And the way that it achieves that performance is by using what it calls a directed acyclic graph engine. The fancy thing about Spark is that it does not do anything until we ask it to deliver results, and at that point, it creates this graph of all the different steps that it needs to put together to achieve the results we want. And it does that in an optimal manner so it can wait until it knows what we are asking for and then figure out the optimal path to answering the question that we want.

Easy to code:

- Code in Python, Java, or Scala
- Built around one main concept: The Resilient Distributed Dataset (RDD)

The beautiful thing about Spark is it is not that hard. It allows us to code in Python, Java, or Scala. If we are already familiar with Python, we do not have to learn a new language, and from a developer standpoint, it is built around one main concept: The Resilient Distributed Dataset. There is one main kind of object that we do repeatedly encounter, the RDD, and various operations on that Resilient Distributed Data set let us slice and dice and carve up that data and do what we want with it. So, what would take many lines of code and different functions in a MapReduce job can often be done in just one line and much more quickly and efficiently using Spark.

The Resilient Distributed Dataset -

Let us go into a little bit more depth about how Spark works under the hood. The Resilient Distributed Data set object is sort of the core object that everything in Spark revolves around, even for the libraries built on top of Spark, like Sparks SQL or MLLlib. We are also using RDDs under the hood or extensions to the RDD objects to sort of make them look like something a little bit more structured. But if we understand what an RDD is in Spark, we are like 90 percent of the way there in understanding Spark. Fundamentally it is a data set, and it is an abstraction for a giant set of data, and that is the main thing we need to know as a developer, we know, what we are going to do is setting up RDD objects and loading them up with big data sets and then calling various methods on the RDD object to distribute the processing of that data. Now the beauty is that, although RDDs are both distributed and resilient, you know they can be spread out across an entire cluster of computers that may or may not be running locally, and they can also handle the failure of specific executor nodes in our cluster automatically and keep on going even if one node shuts down and redistribute the work as needed when that occurs. But we do not have to think about those things; that is what Spark does for us, and that is what our cluster manager does for us. So even though an RDD is distributed and resilient, which is a very powerful thing, we do not have to worry about the specifics about how that works. All we need to really know as the developer is that it represents a big data set, and we can use the RDD object to transform that data set from one set of data to another or to perform actions on that data set to get the results we want from it. Now from a programming standpoint, we are going to be given a Spark Context object.

The SparkContext –

- Created by the driver program
- Is responsible for making RDDs resilient and distributed!
- Creates RDDs
- The Spark shell creates an 'sc' object for us

Transforming RDDs - One common thing we are going to do once we have an RDD has transformed it in some way, shape, or form, and these are some of the basic operations we can do on RDDs, it is not a complete list, but these are all the most common operations we can do on an RDD

- map
- flatmap
- filter
- distinct
- sample
- union, intersection, subtract, cartesian

RDD Actions - In addition to transforming one RDD into another, we can also perform actions onto an RDD. Once we have the data that we want into an RDD dataset, we can then perform an action on it to get a result out of it. The most common actions are

- collect
- count
- countByValue

- take
- top
- reduce
- …and more…

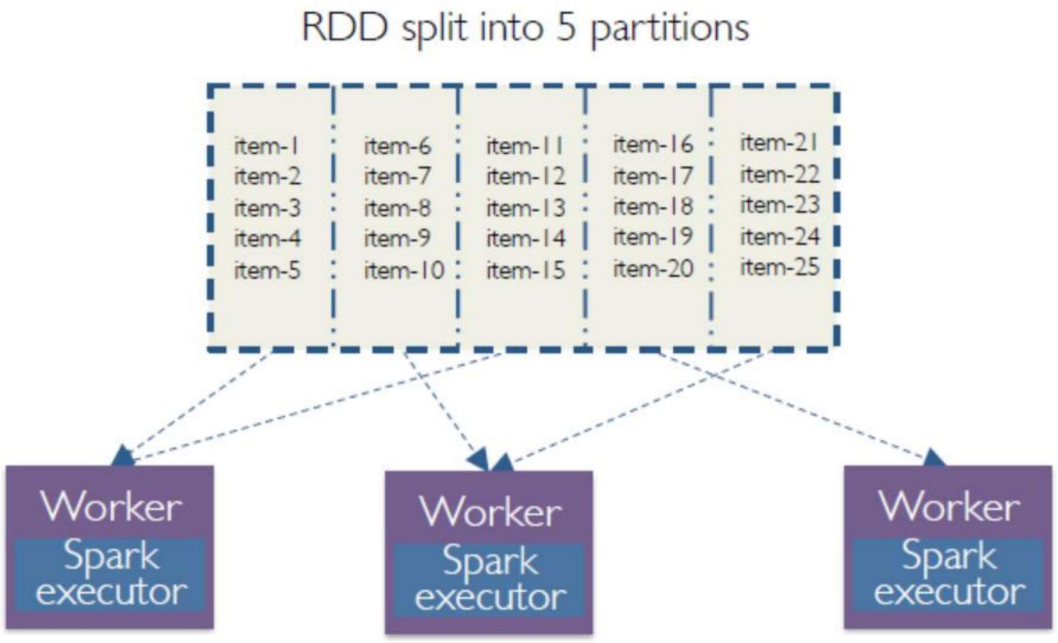
And another thing to understand with RDD is nothing happens until we call an action, and we talked earlier about how Spark is so fast because it constructs this directed acyclic graph as soon as we ask for an action to happen because at that point, it knows what needs to be done to get the results that we want, and it can compute the most optimal path to make that happen. So it is important when we are writing our Spark driver scripts that our script is not actually going to do anything until we call one of these action methods, and at that point, it will actually start farming things out to our cluster or run it on our own computer, whatever you told it to do, and actually start executing that program so we will not actually have any results or any processing whatsoever until one of these action methods are actually called within our script.

Optimizing running on a cluster: Partitioning

- Spark is not totally magic: we need to think about how our data is partitioned
- Set partition while creating an RDD or use .partitionBy() on an RDD before running a large operation that benefits from partitioning like
  - Join(), cogroup(), groupWith(), join(), leftOuterJoin(), rightOuterJoin(), groupByKey(), reduceByKey(), combileByKey(), and lookup()
  - Those operations will preserve our partitioning in their results too.

Choosing a Partition Size - How do I choose the argument for partition by how many partitions is the right partition size. If we have too few partitions, then we will take full advantage of our cluster, it cannot spread it out effectively, but if we have too many, we know we end up shuffling data around and breaking things up into two small chunks, and there is some overhead associated with running an individual executive job. We do not want too many executives either, but we want at least as many partitions as we have caused in our cluster or as many will fit in your available memory.

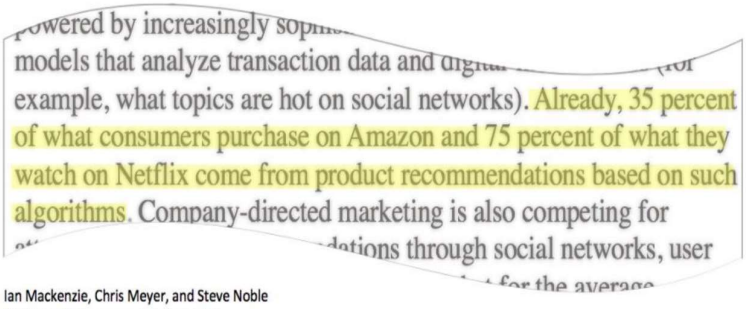
- Too few partitions will not take full advantage of our cluster
- Too many results in too much overhead from shuffling data
- At least as many partitions as we have cores or executors that fit within available memory.



Visualization of RDD being partitioned (photo credit: @medium.com)

Introduction to Music Recommender System Project:

You are probably aware of some recommendation engines where a website tells you something along the lines of, "If you like that, then you'll probably like this". You have likely seen these types of recommendations on your favorite retail or media streaming websites. These recommendations are generated through different data types that you as a user or customer provide directly or indirectly. When purchasing something online, watching a movie, or even reading an article, you are often given a chance to rate that item on a scale of 1 to 5 stars, a thumbs up or thumbs down, or some rating scale. Based on your feedback from these rating systems, companies can learn a lot about your preferences and offer recommendations based on similar users.


powered by increasingly sophi
models that analyze transaction data and digital
example, what topics are hot on social networks). Already, 35 percent of what consumers purchase on Amazon and 75 percent of what they watch on Netflix come from product recommendations based on such algorithms. Company-directed marketing is also competing for
dations through social networks, user
for the average

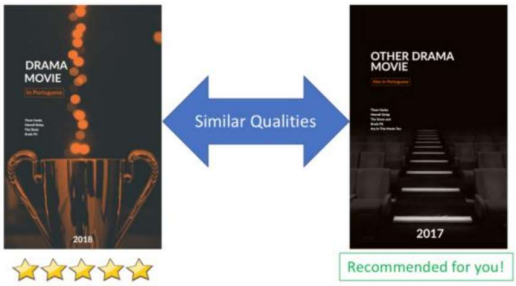
Ian Mackenzie, Chris Meyer, and Steve Noble
McKinsey & Company, October 2013

An article published by McKinsey & Company in October of 2013 stated that 35% of what customers buy on Amazon and 75% of what they watch on Netflix come from product recommendations based on algorithms such as the one going to implement in this project.

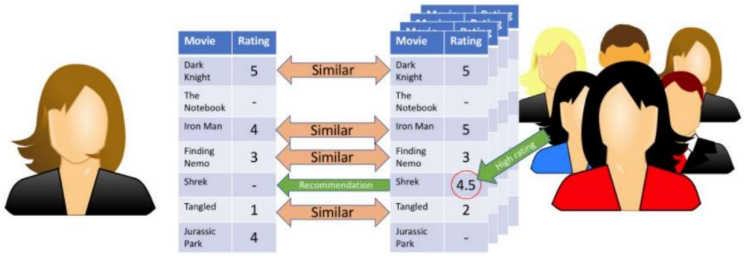
Types of recommendation engines:

In the world of recommendation engines, there are two basic types.

1. Collaborative-filtering engines
2. Content-based filtering engines.



An example of Content-based Filtering – based on features of items



An example Collaborative Filtering – based on similar user preferences

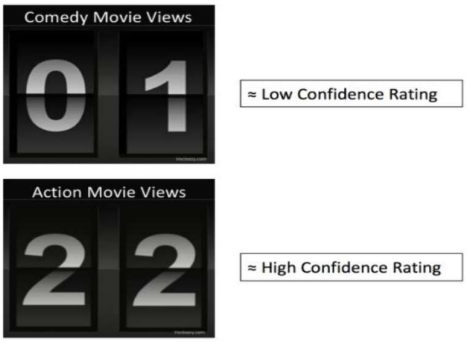
Both aim to offer meaningful recommendations, but they do so in slightly different ways. As the name suggests, content-based filtering tries to understand the content or features of the items and makes recommendations based on your preferences for those specific features. Collaborative filtering is a little bit different. Collaborative filtering is based on user similarity. However, unlike content-based filtering, manually created tags are not necessary. The features and groupings are created mathematically from patterns in the ratings provided by users. When you provide ratings for a product or item, whether it be a thumbs up or thumbs down, or even if you just watch a video without even giving it a rating, you are providing meaningful insight about your preferences. From this behavior, the Alternating Least Squares (ALS) algorithm can mathematically group you with similar users, predict your behavior, and help you have a more effective customer experience. While ALS can have content-based applications, this project implements its application to collaborative filtering, but many of the principles of collaborative filtering can be applied to content-based applications.

Types of ratings:

There are two types of ratings

1. Explicit ratings
2. Implicit ratings

An example of Explicit ratings



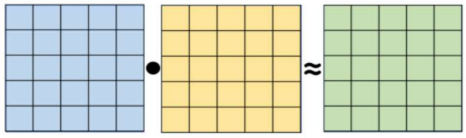
An example of Implicit ratings

Some stars or something like a thumbs up or thumbs down are explicit ratings because users explicitly state how much they like or dislike something. Implicit ratings are a little bit different. They are based on the passive tracking of your behavior, like the number of movies you have seen in different genres. Fundamentally, implicit ratings are generated from the frequency of your actions. For example, if you watch 30 movies, and of those 30 movies, 22 are action movies, and only 1 is a comedy, the low number of comedy views will be converted into low confidence that you like comedies, and the high number of action movie views will be converted into high confidence that you like action movies. These probabilities are then used as ratings. The logic behind this is, in essence, the more you carry out behavior, the higher the likelihood that you like it, and thus a higher rating.

Additionally, in some cases, you may not have access to behavior counts like this. A simpler form of ratings that still works with the ALS algorithm is simple binary ratings. Rather than having a count of user actions, binary ratings just show whether a user has done something like watched a comedy, represented by a 1 or not watched a comedy, represented by a 0. These types of ratings are not nearly as rich, but they still can provide meaningful insight and still work perfectly fine with the ALS algorithm. This project implements the algorithm based on implicit ratings.

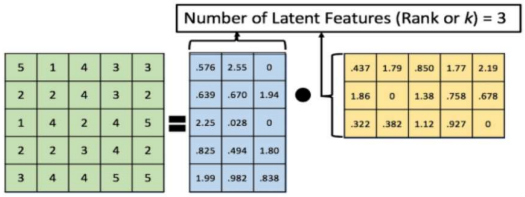
Matrix Factorization:

Matrix factorization, or matrix decomposition, is essentially the opposite of matrix multiplication. Rather than multiplying two matrices together to get one new matrix, matrix factorization splits a matrix into two or more matrices which, when multiplied back together, produce an approximation of the original matrix. There are several different mathematical approaches for this, each of which has a different application. The below diagram explains the factorization that ALS performs.



Matrix Factorization

Used in the context of collaborative filtering, ALS uses a factorization called non-negative matrix factorization. Because matrix factorization generally returns only approximations of the original matrix, in some cases, they can return negative values in the factor matrices, even when attempting to predict positive values. When predicting what rating a user will give to an item, negative values do not really make sense. Neither do they make sense in the context of latent features. For this reason, the version of ALS that we will use will require that the factorization return only positive values. Let us look at some sample factorizations.



The rank of factor matrices

Here is a sample matrix of possible item ratings. There are 5 rows and 5 columns. And here is one factorization of that matrix called the LU factorization. Notice that the factor matrices are the same dimensions or rank as the original matrix. Also, notice that some of the values in this factorization are negative. Using this type of factorization could result in negative predictions that would not make sense in our context. Here is another factorization. In this case, all the values are positive, meaning the resulting product of these factor matrices is guaranteed to be positive. This is closer to what we need for our purposes. Notice here that the dimensions of the factor matrices are such that the first-factor matrix has the same number of rows as the original matrix but a different number of columns. Also, the second-factor matrix has the same number of columns as the original matrix but a different number of rows.

The dimensions of the factor matrices that do not match the original matrix are called the "rank" or number of latent features. In this case, we have chosen the "rank" of the factor matrices to be 3. What that means is that the number of latent features of the factor matrices is 3. When doing these types of factorizations, you get to choose the rank or number of latent features the factor matrices will have. Now, look at this matrix. Not all cells have numbers in them. Despite this, we can still factor the values in the matrix. Also notice that because there is at least one value in every row and at least one value in every column that each of the factor matrices are totally full. Because of this, factoring a sparse matrix into two factor matrices gives us the means to not only approximate the original values that existed in the matrix to begin with, but to also provide predictions for the cells that were originally blank. And because the factorization is based on the values that existed previously, the blank cells are filled in based on those already-existing patterns. So, when we do this with user ratings, the blanks are filled in with values that reflect the individual user behavior and the behavior of users similar to them. That is why this method is called collaborative filtering.

About Dataset:

The dataset this time comes from the Million Songs Dataset available from Kaggle (Million Song Dataset Challenge | Kaggle). I am going to be using one file of this dataset titled "kaggle_visible_evaluation_triplets.txt." It contains information on over 1 million users, including the number of times they have played nearly 400,000 songs. This is more data than we can use for this project, so we will only be using a portion of it. We will first examine the data, get summary statistics, and then build and evaluate our model. One thing to note here is that because the use of implicit ratings causes ALS to calculate a level of confidence that a user likes a song based on the number of times they've played it, the matrix will need to include zeros for the songs that each user has not yet listened to.

Preprocessing: The given data has more than 1 million users and songs; creating recommendations requires many resources. Many inactive users in user's data even did not listen to a single song, and there are many songs that did not hear by anyone. Therefore, we pick only active users who listen songs at least 50 times. And, only chose those songs which are played at least 150 times altogether. After this, we do the cross join between users and song columns and fills all null values with zeros. This cross join creates millions of rows in the spark data frame. The Resulting data frame's columns need to be converted to index since Alternating Least Square(ALS) algorithm can only be applied on specific columns of type integer which holds respective index values. Pipeplie is used to convert string to the index for the columns "user" and "song".

Evaluating Model:

We have an implicit rating dataset, let us discuss these types of models. The first thing you should know is that the implicit rating model has an additional hyperparameter called alpha. Alpha is an integer value that tells the Spark how much each additional song play should add to the model's confidence that a user likes a song. Like the other hyperparameters, this will need to be tuned through cross-validation. The challenge of these models is the evaluation. With explicit ratings, we use the RMSE. It made sense in that situation because we could match predictions back to a true measure of user preference.

userId	movieId	num_plays	implicit rating prediction
1	2112	16	1.755
1	303	3	.88
2	5	1	.01
2	77	2	.5
3	913	1	.08
3	44	21	1.98
3	6	4	.98

Different metrics.
RMSE doesn't make sense here.

An example - why RMSE does not work in the case of Implicit Ratings

In the case of implicit ratings, however, we don't have a true measure of user preference. We only have the number of times a user listened to a song and a measure of how confident our model is that that they like that song. These aren't the same thing, and calculating an RMSE between them doesn't make sense. However, using a test set, we can see if our model gives high predictions to the songs that users have listened to. The logic being that if our model is returning a high prediction for a song that the respective user has actually listened to, then the predictions make sense, especially if they've listened to it more than once. We can measure this using this Rank Order Error Metric

(ROEM). Code Implementation of ROEM is taken from Building Recommendation Engines with Pyspark - Notes by Louisa (gitbook.io).

$$\mathrm{ROEM} = \frac{\sum_{u,i} r^t_{u,i} \mathrm{rank}_{u,i}}{\sum_{u,i} r^t_{u,i}}$$

Code Implementations and testing:

With DataFrame:

Since the purpose of the Big Data and Parallel Processing project is to determine the significance of this technology in improving the efficiency and performance of bigdata using PySpark. Therefore, we need to run and test the performance by comparing the results with a single CPU machine versus multiple workers. Please refer to the files attached with this project report. Time taken in model building and actions like the show and take calls on spark data frame is shown in jupyter notebook output cells.

1. music_rcs_local.ipynb
2. music_rcs_colab_gpu.ipynb
3. music_rcs_gcp_single.ipynb
4. music_rcs_gcp_multi.ipynb

One file as named below is used to find the best hyperparameters using cross validation with ROEM evaluation method. Since it takes time in hours to find the parameters, I did not add the time decorator to find the time taken to run this code. This file is run on GCP multicore setup.

1. music_rcs_cv_gcp_multi_worker.ipynb

Hyperparameters tried with are as below:

1. rank, k: number of latent features
2. maxIter: number of iterations
3. regParam: Lambda; regularization parameter, term added to error matrix to avoid overfitting the training data
4. alpha: Only used with implicit ratings. How much (in integer) should add to the model's confidence that a user likes the movie/song.

Links to datasets in csv format:

1. users.csv (contains a list of users)
2. songs.csv (contains a list of songs)
3. triplets.csv (contains rows of user, song, and number of time that song played by the user)

Comparisons in tabular form:

All the entries in the table are in seconds.

	Local Machine	Colab with GPU	GCP with Single Master	GCP with Multiple Workers (2 nodes)
model building	998.4110400676727	422.138795375824	23.938844442367554	19.361108779907227
executing action '.show()'	8.02993893623352	6.436712980270386	7.12470006942749	6.179260492324829
executing action '.take()'	8.538794994354248	4.780577659606934	4.251834392547607	3.908982753753662

Time taken on various resources in seconds

With RDD:

Instead of Spark ML, RDD uses Spark MLlib library. Spark's MLlib is focused on providing a core set of algorithms for people to use, while largely leaving the data pipeline, cleaning, preparation, and feature selection problems up to the user. In this project, I use StringIndexer for data pre-processing since MLlib does not directly support indexing. After the required pre-processing, the file is stored in CSV format using the below code:

df_r.repartition(1).write.csv(<file_path>, sep='|')

This table gives the comparison with different numbers of partitions in seconds.

	On Local machine			On GCP		
partitions	2	4	6	2	4	6
File reading	0.046998023986816406	0.05300092697143555	0.050998687744140625	0.03315377235412598	0.042401790618896484	0.04363512992858887
Model Building	32.213173627853394	33.734523773193336	33.530789613723755	28.789074420928955	28.563907146453857	30.538069009780884
Specific user's songs recommendations	0.08245134353637695	0.09696340560913086	0.08882761001586914	0.27993226051330566	0.19551539421081543	0.24622130393981934
.take(20) run time	0.7343583106994629	1.2869985103607178	1.3754560947418213	0.1340489387512207	0.16965603828430176	0.1913614273071289

Time taken on various resources in seconds

List of files with RDD implementations:

1. music_rcs_gcp_2_partitions.ipynb
2. music_rcs_gcp_4_partitions.ipynb
3. music_rcs_gcp_6_partitions.ipynb
4. music_rcs_local_2_partitions.ipynb
5. music_rcs_local_4_partitions.ipynb
6. music_rcs_local_6_partitions.ipynb

Link to pre-processed file:

1. rdd_preprocessed_triplets.csv

Limitations:

There are three significant limitations with this project

1. It is not including all users and songs for recommendations
2. It can not do Cross-validation for all scenarios.
3. Since the StringIndexer pipeline can't be created in this dataset using MLlib (deprecated), indexing is done with the help of a modern ML library.

Both the limitations are because of resource constraints. Inclusion of all users and songs requires more than 100 GB of memory. That's why I decided to filter data-frame with the filters on users and songs. Hyperparameter tuning with Cross-Validation takes too much time with ROEM evolution method. For this project, I decided not to cross-validation on low resources devices like single-node machines.

Conclusion:

The Alternating Least Square algorithm is a specialized implicit collaborative algorithm implemented in Apache Spark. In this project, we verify that parallel processing on big data gives a considerable boost in performance and efficiency. Code runs on a single node and on the cloud has a significant difference in terms of efficiency. A better model can be picked after hyperparameter tuning with cross-validation. It is not feasible to work on Big data with a high-end local PC. Cloud services are essential to building a good model since it requires resources like cores, memory, and disks. Machine Learning technologies combined with cloud computing for high performance are continuously evolving, and the future looks bright.