# Distributed machine learning algorithms on radio datasets

## Contents

## Abstract

The purpose of this project was to look into different techniques that the radio datasets could be used in order to predict recommendations for the users or the number of times that a user would be interested in listening to an artist. Collaborative filtering algorithms, machine learning algorithms and graph algorithms were used.

Graph algorithms were used as alternative ways of providing recommendations.

Random Forest Regression (RF) was used on the combined datasets of the user plays and user profile dataset to predict the total plays of each user, given their background (gender, country, age and the artists they listen to). Spark MLlib and ML libraries were used for RF and the performances were compared by reviewing the runtime and mean squared error.

Alternating Least Squares was used to recommend artists to users based on implicit feedback data. We conducted analyses on hyperparameters that define the model and made evaluations on the recommended artists using Mean Percentile Ranking, Mean Recall and number of correct artists recommended.

Stochastic Gradient Descent was used in a distributed way to predict the number of times that a user would be interested in listening to an artist. The project focused on the best ways that the algorithm can perform and it was shown that a small number of latent factors was enough for a good performance and the number of partitions should be equal to the number of workers in the working cluster.

## 1. Introduction: Choice and description of data

The datasets used in this project were collected from (Last.fm, 2010). This data

## User-Artist plays Dataset

The one is named ‘usersha1-artmbid-artname-plays.tsv’ and includes links between users of a radio station and artists that the users listen to. The dataset also includes the number of times that a user listened to the artist. There are 17,332,990 rows in the dataset, with 185,677 artists and 359,337 users. The schema of the dataset can be seen here:

```
play_df.printSchema()
```

```
root
 |-- user_id: string (nullable = true)
 |-- mus_artist_id: string (nullable = true)
 |-- artist_name: string (nullable = true)
 |-- plays: long (nullable = true)
```

## User Profile Dataset

The second dataset is named ‘usersha1-artmbid-artname-plays.tsv’ and includes attributes of the users, as seen from its schema below.

```
profile_df.printSchema()
```

```
root
 |-- user_id: string (nullable = true)
 |-- gender: string (nullable = true)
 |-- age: long (nullable = true)
 |-- country: string (nullable = true)
 |-- signup: string (nullable = true)
```

The user profile dataset has a total of 267,382 users, with no duplicates. There were 111 distinct values of the age category, which clearly indicated that there were outliers. The minimum and maximum age were found to be -1337 and 666 respectively. A distribution of the age was plotted, showing that this was indeed the case, and that most users were aged between 10 and 80. Therefore, this range was used for future calculations.

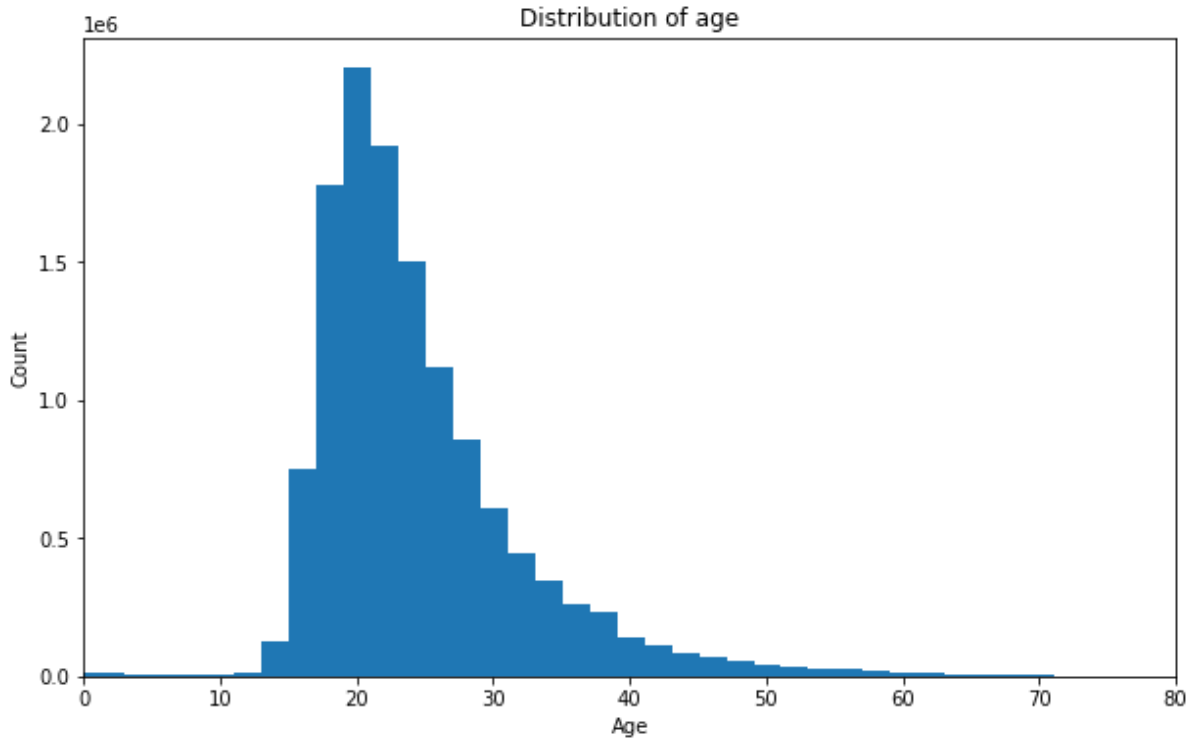



*Figure 1.1 Distribution of age: Most users in the dataset are in the teenage years and 20s*

Most users were found to be from the United States, having a count of 2.4 million users. Followed by Germany and the United Kingdom with 1.1m and 1.0m respectively. The table below shows the top 20 countries with the highest users. Furthermore, there are 239 distinct countries recorded in the dataset.

*Table 1.1 Top five countries with highest plays*

| **Country** | **Count** |
|---|---|
| United States | 2407884 |
| Germany | 1122323 |
| United Kingdom | 1046677 |
| Poland | 793690 |
| Russian Federation | 689921 |

*Table 1.2 Distinct count of each feature*

| | Country | Age | Gender | Artist |
|---|---|---|---|---|
| **Distinct count** | 239 | 69 | 2 | 185678 |

## Graph Analysis

To facilitate the data description process and to suggest some ways on recommending artists, a direct graph was built. The vertices of the graph are the users and the artists and the edges are formed when a user listens to an artist. Two graphs were formed, one weighted and with user attributes and one un-weighted. The weights on the weighted graph are the number of times that a user listened to the artist (plays). This graph was built using the Spark package, Graphframes.

The figure below shows a prototype network, as it would look if a user 'A' listens to two artists, a user 'B' to 1 artist, and they have an artist in common.

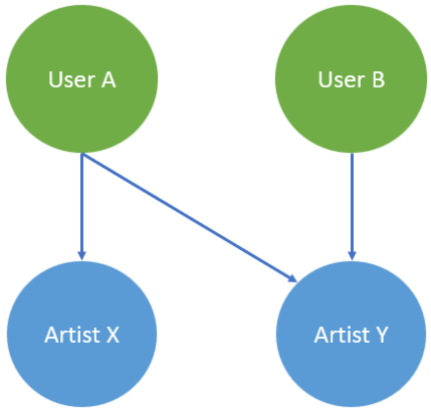



*Figure 2: Prototype example of the graph*

### Number of users per artist

The cumulative distribution of the number of users per artist has been calculated. It can be seen that most artists have less than 100 users in their audience, but there are a few that are the most famous ones and have thousand users.

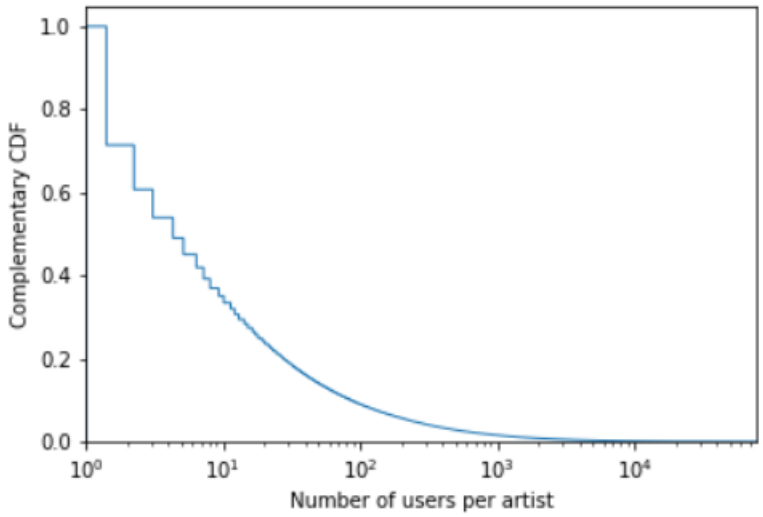



*Figure 3: Cumulative distribution of number of users per artist*

## Famous artists

If we ignore each user's preference and want to recommend an artist to a user, a naïve way would be to calculate the most popular artists and recommend those to all users. Calculating the in-degree of the graph we can get the top 10 artists that are listened to by users, hence the most famous artists. So, the table below shows us that Radiohead are listened to by 77,348 users.

*Table 3: Top 10 artists with highest in-degree*

| id | inDegree |
|---|---|
| radiohead | 77348 |
| the beatles | 76339 |
| coldplay | 66738 |
| red hot chili peppers | 48989 |
| muse | 47015 |
| metallica | 45301 |
| pink floyd | 44506 |
| the killers | 41280 |
| linkin park | 39833 |
| nirvana | 39534 |

## Most influential artists - PageRank

Another approach, would be to calculate the PageRank of each artist. PageRank would inform us about the most influencial artist. PageRank can be calculate based on iterative computations, and

hence graph parallel computation can be used. The iterative nature comes from the fact that the algorithm updates the vertex propertiess, based on its neighbour vertex property (Gonzalez et. Al., 2014).

Based on PageRank algorithm, these are the artists with the most influence. These artists are related to the most famous artists, but are also the ones that are listened from users that listen to many other well-influenced and popular artists.

Page Rank has been calculated for both weighted and un-weighted network. We see that the exact same artists come up as the most influential ones, but with a smaller page rank value when looking into the weighted network.

*Table 4: Top 10 artists with highest PageRank value for the un-weighted graph*

| id | pagerank |
|---|---|
| radiohead | 863.0589461767735 |
| the beatles | 855.9524952068926 |
| coldplay | 755.7580597175711 |
| red hot chili peppers | 556.7589781864452 |
| muse | 530.9934376172165 |
| metallica | 515.1416222927016 |
| pink floyd | 497.9394656264087 |
| linkin park | 467.48710300398074 |
| the killers | 462.8817883282724 |
| nirvana | 449.06797599335584 |

*Table 5: Top 10 artists with highest PageRank value for the weighted graph*

| id | pagerank |
|---|---|
| radiohead | 659.6347388760202 |
| the beatles | 658.2181833634426 |
| coldplay | 586.3668110778707 |
| red hot chili peppers | 446.7630069366572 |
| muse | 431.9242738521465 |
| metallica | 421.1938022314983 |
| pink floyd | 389.37384710286847 |
| linkin park | 386.13745919559597 |
| the killers | 365.67722483988706 |
| nirvana | 356.2079528102664 |

## Similar taste artists

We wanted to learn which are the most common pairs of artists that are usually listed by a user. In other words, which are the artists that are similar with each other regarding the taste of the users.

To achieve that we used the graph with the user attributes and created a relevant motif. To zoom a bit more into specific features of the data, we also calculated the most common pairs of artists for users that are from Italy and from Sweden. When looking into the pairs for each country it is interesting to see that some artists come at the top of the list for both countries, but with a different pair. This can be useful for cross-country recommendations. For example, we see that Italians who listen to Radiohead, often listen to Coldplay too. Swedish people who listen to Coldplay, often listen to Kent too. Therefore, depending on the purpose of this analysis, we could consider an algorithm that suggests Radiohead to Swedish users who listen to Coldplay.

*Table 6: Similar taste pair of artists for Italy*

| grp_artists | grp_cnt |
|---|---|
| [the beatles, radiohead] | 582 |
| [radiohead, coldplay] | 575 |
| [the beatles, pink floyd] | 568 |
| [radiohead, afterhours] | 526 |
| [radiohead, pink floyd] | 525 |
| [franco battiato, fabrizio de andré] | 520 |
| [the beatles, fabrizio de andré] | 512 |
| [pink floyd, fabrizio de andré] | 489 |
| [radiohead, fabrizio de andré] | 488 |
| [radiohead, muse] | 449 |

*Table 7: Similar taste pair of artists for Sweden*

| grp_artists | grp_cnt |
|---|---|
| [kent, coldplay] | 1082 |
| [the killers, coldplay] | 993 |
| [kent, håkan hellström] | 869 |
| [lars winnerbäck, kent] | 833 |
| [the killers, kent] | 794 |
| [metallica, in flames] | 733 |
| [håkan hellström, coldplay] | 729 |
| [lars winnerbäck, coldplay] | 725 |
| [the beatles, bob dylan] | 657 |
| [lars winnerbäck, håkan hellström] | 652 |

Connected components algorithm can be used for identifying bigger groups of similar artists, in a more computationally efficient way (Gonzalez et. Al., 2014). However, as this is not the main focus of the project, it has not been implemented here.

# 2. Stochastic Gradient Descent Algorithm (SGD)

## Methodology

One algorithm that was used to find artist recommendations for the users, is the Stochastic Gradient Descent. It is a latent factor model that uses matrix factorization (Baalbaki, 2016). The Gradient Descent algorithm was not chosen, as it would have been computationally expensive for big data, since it processes all the training data, by calculating the gradient of their loss function and summing it up for all of them. SGD, trains only a random sample of the data. This means that it computes a noisy estimate of the gradient. The algorithm then performs the updates according to gradient descent, using the gradient estimates instead of the true values.

In detail, the goal is to approximate the ratings matrix V, considering as ratings the number of times that each user listened to an artist (number of plays), by a low rank matrix M. M matrix is considered as the product of 2 matrices W and H, where Wi is a user parameter vector and Hj an artist parameter vector.

$$M_{ij} = Wi^T * H_j$$

Computing the gradient for all the training data and summing them up would have been as follows:

$$\nabla f(V; W, H) = \sum_{i=1}^{m} \nabla fi(V_{ij}; W_{i*}, H_{*j})$$

Instead, the SGD calculate the stochastic gradient vector $g_t$ at t iteration, for a random sample of k parameters.

$$g_t(V) = \frac{1}{k} \sum_{i \varepsilon St} \nabla fi(V_{ij}; W_{i*}, H_{*j})$$

Where $S_t$ is a random sample of k observations from the dataset.

The gradient is calculated on the l2 Loss function and hence the aim is to minimize that after a number of iterations.

In particular, the matrices W and H are updated one after the other, in every iteration, so that the algorithm can make use of their convex property. So, first the gradient of the l2 Loss function with respect to matrix W is calculated and then the l2 Loss function with respect to the matrix H.

SGD in this project was implemented in a distributed way, by finding blocks of data in the dataset that have no data in common. The dataset is then divided into these blocks and hence the blocks can be updated simultaneously. The blocks are distributed matrices, small in order to fit in memory on a single machine and they are based on an RDD (Bosagh Zadeh et. Al, 2016). The convergence was set into a maximum number of iterations.

The pseudo logic of the algorithm that was used can be seen below:

Require: $Z$, $W_0$, $H_0$, cluster size $d$
W$\leftarrow W_0$ and $H \leftarrow H_0$
Block $Z/W/H$ into $d \times d/d \times 1/1 \times d$ blocks
while not converged do $/ * epoch^* /$
Pick step size $\in$
for $s = 1, \ldots, d$ do $/ * subepoch^* /$
Pick $d$ blocks $\{Z^{1j_1}, \ldots, Z^{dj_d}\}$ to form a stratum
for $b = 1, \ldots, d$ do $/ * in\ paralle\ l^* /$
Run SGD on the training points in $Z^{bj_b}$ (step size $=\in$)
end for end for end while

*Picture: Distributed SGD for Matrix Factorization (Baalbaki, 2016)*

Other actions that were taken to achieve the distributed computing fashion are the following.

- The dataset was stored in Hadoop Distributed File System (HDFS), to efficiently store the large dataset file across the 2 worker nodes and the 1 master nodes in a cluster (Shvachko et. al., 2010).
- The dataset was transformed into Resilient Distributed Datasets (RDDs), which facilitate the efficient performance of in-memory computations, based on Based on coarse-grained transformations like MapReduce and with efficient fault-tolerance (Karau, 2015).
- The RDDs were cached into memory, for efficient processing.
- The RDDs were distributed into more than 1 partition.

The performance of the algorithm was tested against different distributed computing parameters and different hyper-parameters.

In particular, it was tested against:

- different number of latent factors
- different number of partitions
- different number of iterations
- different size of the dataset

For the evaluation of the algorithm, the computation time was calculated and the l2 loss after each iteration. We consider a good performance when the l2 loss is low and the computation time low.

For future improvements, the factorization of two matrices can be improved, in terms of efficient computing.

## Numerical evaluation

### L2 loss vs Number of latent factors vs number of iterations for different data size

Comparing the L2 loss with the number of iterations and the number of latent factors used, we see a different behaviour when the matrix is of 17,559 order and when it is double the size. For the 17K dataset we see that the L2 loss increases significantly on higher number of iterations, so it suggests that no more than 1 iteration should be performed. For the 35K dataset, the best L2 loss values are seen at the second iteration. Regarding the number of factors, when the dataset is larger, smaller number of latent factors return a smaller L2 loss value, but for the 17K dataset we observe a higher variability.

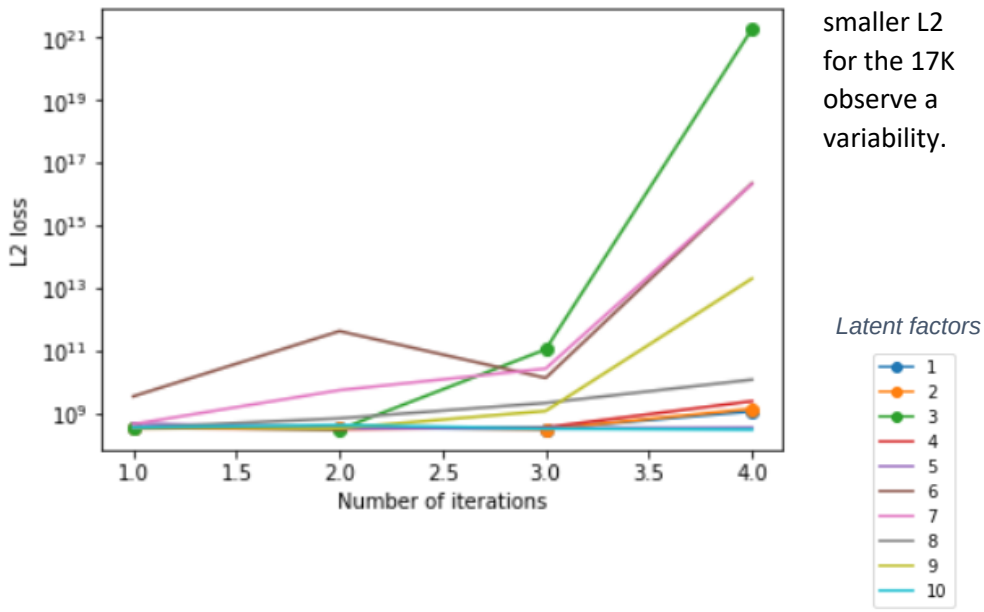



*Figure 2.1 : L2 loss vs Number of latent factors vs number of iterations for a 17,559 x 17,559 matrix.*

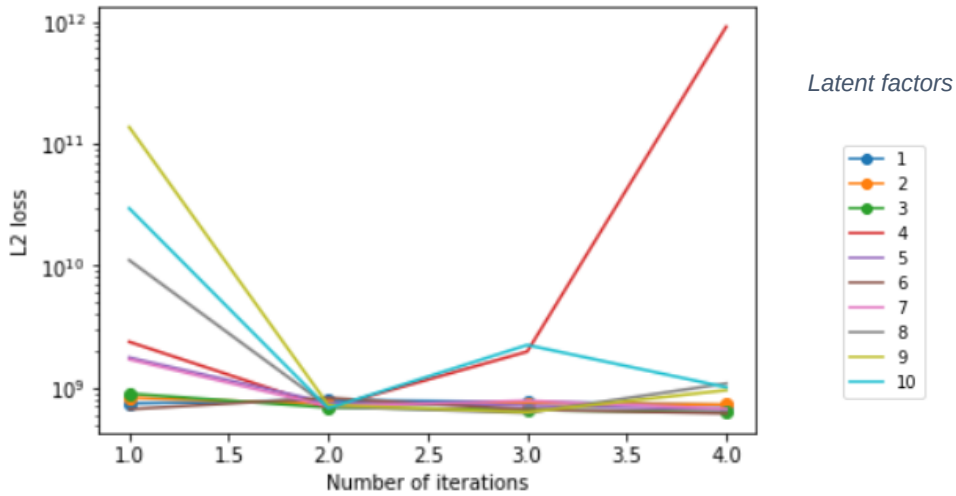



*Figure 2.2: L2 loss vs Number of latent factors vs number of iterations for a 35,119 x 35,119 matrix.*

## Computation time vs number of latent factors

The relationship observed between the computation time and the number of latent factors used is fairly random.

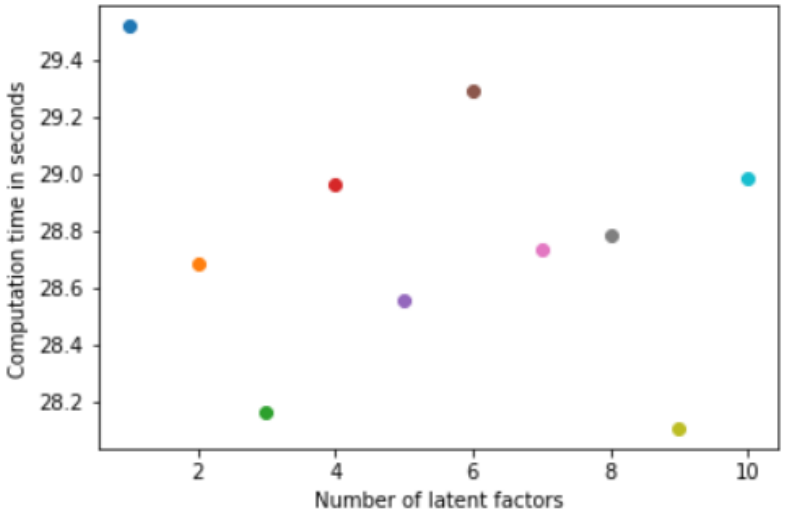



*Figure 2.3: Computation time vs number of latent factors for a 17,559 x 17,559 matrix.*

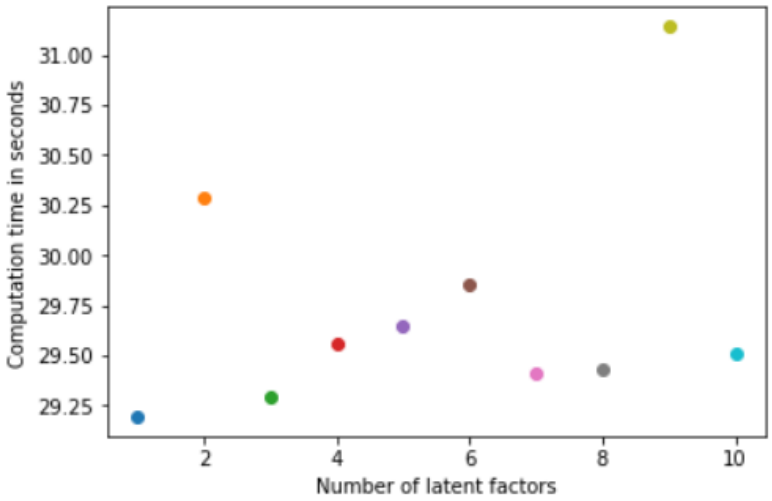



*Figure 2.4: Computation time vs number of latent factors for a 35,119 x 35,119 matrix.*

## L2 loss vs Number of latent factors

We see smaller l2 loss values for fewer number of latent factors. Also, it is worth noting that for a larger sample size, the l2 loss values are smaller, which makes sense since the information given to the algorithm is more.

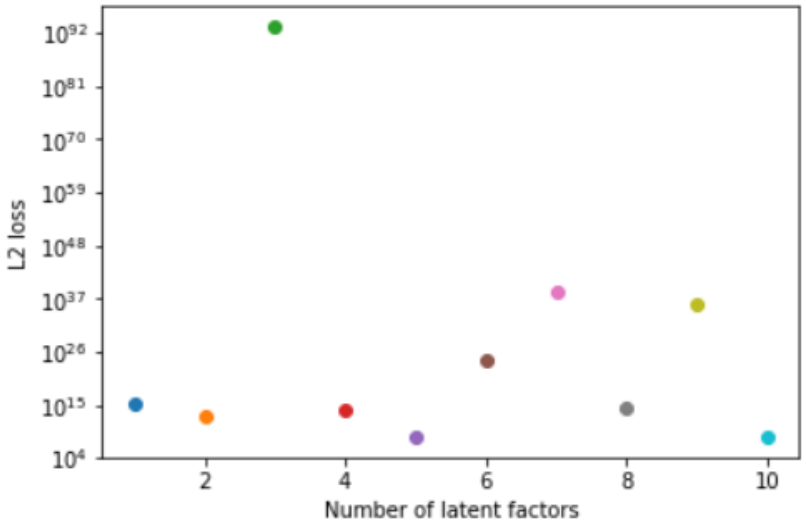



*Figure 2.5: L2 loss vs number of latent factors for a 17,559 x 17,559 matrix.*

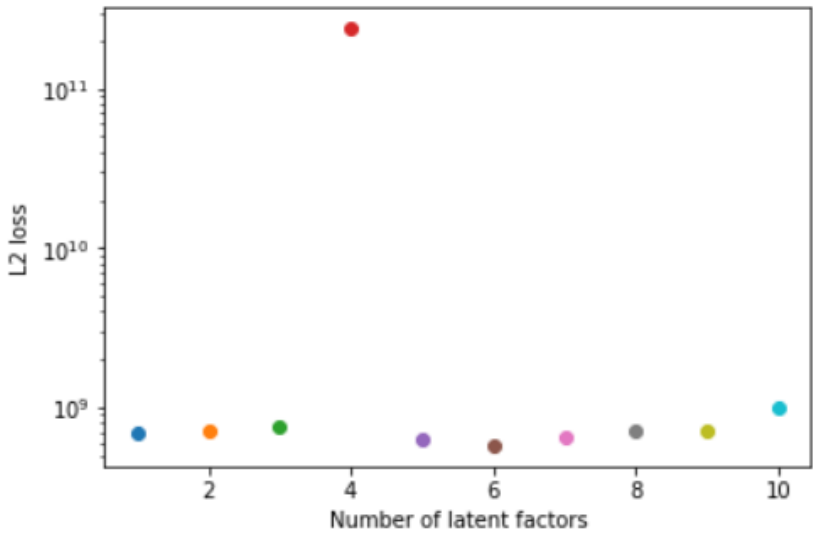



*Figure 2.6: L2 loss vs number of latent factors for a 35,119 x 35,119 matrix.*

## Computation time vs number of partitions

It is observed that when the data is partitioned into at least 2 partitions, the computation time of the algorithm decreases significantly, and we can see the benefits of distributed computing. For both sample sizes, it is seen that 2 partitions perform slightly better than 3 or 4 partitions. The reason behind that can be that since we have 2 workers, increasing the partitions to 3 or 4 doesn't mean that the 3 or 4 partitions can be processed simultaneously. It means though that there is some time spent to partition the data and process the different partitions. Therefore, the optimal number of partitions is 2 for this situation.

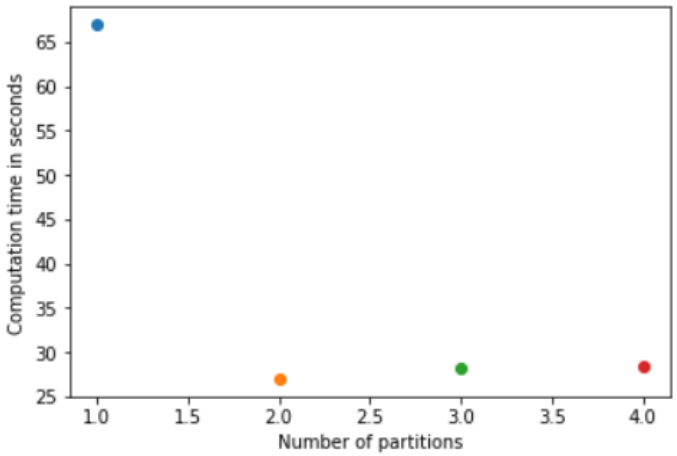



*Figure 2.7 : Computation time vs number of partitions for a 17,559 x 17,559 matrix.*

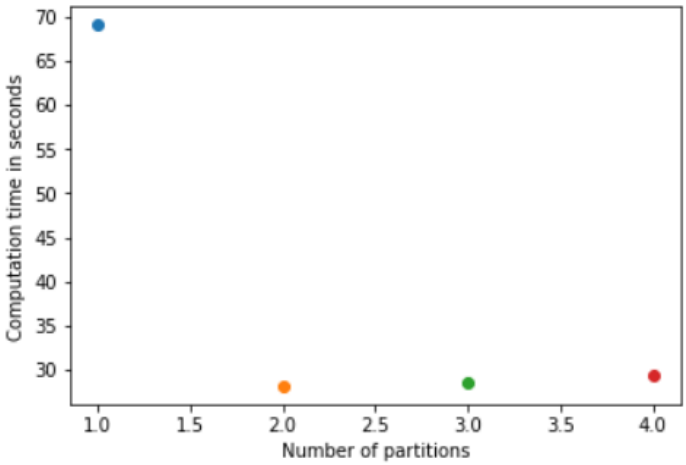



*Figure 2.8: Computation time vs number of partitions for a 35,119 x 35,119 matrix.*

# 3. Alternating Least Squares (ALS) for Implicit Feedback

## Methodology

The second approach we developed on this dataset is a recommender system, that treats the dataset as an implicit feedback dataset. This model was built and trained using only the data stored in "usersha1-artmbid-artname-plays.tsv". This dataset does not contain any explicit feedback on user preferences on artists: it only represents the number of times each user played an artist. Therefore, the recommendation system must be able to infer preferences by observing user behaviour (Koren et al., 2009).

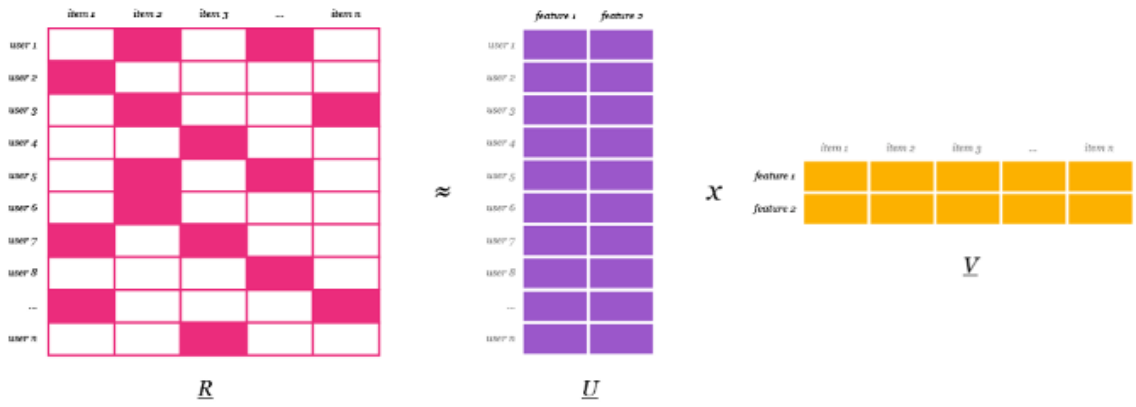
Implicit Feedback data is usually more available than Explicit Feedback data: users do not often submit their preferences on items they have consumed, while users' behaviour is relatively easy to track. However, some features of this kind of data prevent the use of recommendation techniques that are designed for explicit feedback (Ryza et al., 20187):

1. The dataset only contains information on user activity, from which we can infer which artist a user might enjoy. However, the lack of negative feedback complicates the prediction of which artist the user did not like.
2. Implicit feedback is noisy: we cannot always associate the action of a user listening to an artist with positive feedback. Users may listen to artists they do not like.
3. The numerical values of implicit feedback represent confidence, rather than preference

(Ryza et al., 2017)

Collaborative Filtering is the leading approach in recommendations generation (Baalbaki, 2016). To identify new user-item associations, this method investigates the relationships between users and interdependencies between items (Hu et al., 2008). Among the several Collaborative Filtering solutions, we focus our work on the Alternating Least Square model for implicit feedback proposed in the paper (Hu et al., 2008).

The model proposed uses Matrix Factorization, which is considered one of the best techniques to compute Collaborative Filtering in terms of recommendation quality and scalability (Baalbaki, 2016). Let $R$ be the recommendation matrix, in which every entry $(u, i)$ is some type of feedback of user $u$ to artist $i$. Let $m$ be the number of users and $n$ be the number of artists. The idea is to calculate the matrix $U$ of size $m\ x\ f$ and the matrix $V$ of size $n\ x\ f$ for some (small) value $f$ such that $R \approx U\ x\ V^T$.




(picture taken from https://medium.com/radon-dev/als-implicit-collaborative-filtering-5ed653ba39fe)

U and V are randomly populated, and then iteratively optimized using ALS by alternating the optimization of U with V fixed and vice versa.

In the next section, we highlight how the model is adapted to implicit feedback and scaled in the paper *"Collaborative Filtering for Implicit Feedback Datasets"* (Hu et al., 2008).

## Model (Hu et al., 2008)

The solution suggested in the paper is based on two concepts: confidence and preference.

Let $plays_{ui}$ be the number of times user u played artist i. The *preference* $p_{ui}$ of user u on artist i is:

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

This value represents whether a user has ever played a song for the artist. However, as we stated previously, we cannot always infer a positive feedback just by the fact that u played i. For this reason, the authors of the paper define the variable $c_{ui}$ which represents the *confidence* in observing $p_{ui}$.

$$c_{ui} = 1 + \alpha\, plays_{ui}$$

$\alpha$ controls for the rate of increase.

The objective of the model is to find a vector $x_u \in R^f$ and a vector $y_i \in R^f$ for each user u and item I, such that $p_{ui} = x_u^T y_i$. However, model is different from a regular Matrix Factorization for explicit feedback in two ways. First, here confidence levels are taken into account. Second, optimization should work for all possible users and actors. Given these constrains, the following cost function was developed:

$$\min_{x_\star, y_\star} \sum_{u,i} c_{ui}(p_{ui} - x_u^T y_i)^2 + \lambda \left( \sum_u \|x_u\|^2 + \sum_i \|y_i\|^2 \right)$$

$\lambda$ is a regularization parameter that helps avoiding overfitting.

Here, the Alternating Least Square optimization process allows to work with quadratic cost functions, where global minimum can be easily computed. Moreover, the structure of the variables allows the process to be scaled linearly with the size of the data.

The recommendation is then computed by estimating $\hat{p}_{ui} = x_u T\, y_i$ and then extract the largest K values.

## Metrics

We want to evaluate the outputs of the recommendation system, which consists of an ordered list for each user of 5 artists, sorted in ascending order from the one that is predicted to be most liked.

The two metrics we developed are alterations of the evaluation methodology in (Hu et al., 2008). The first one is the Mean Percentile Ranking for correctly recommended artists. Let $rank_{ui}$ be the percentile-ranking of recommended artist i in list of recommendations for user u.

$$MPR = \frac{\sum_{u,i \in C} rank_{ui}}{|C|}$$

$$where\ C = \{(u,i) \mid i\ is\ recommended\ to\ u\ and\ (u,i)\ is\ in\ the\ validation\ set\}$$

This metric allows us to understand where correct recommendations are placed within the list. Values close to 0% indicate that most of the correct recommendations are near the first position, while values close to 100% indicate they are likely to be at the end of the list.

Since we cannot evaluate the user reactions to the recommendations, precision-based metrics are not ideal. However, as we can compare the recommendation with artists that were listened by users, we can apply recall-oriented measures (Hu et al., 2008). The second metric is the Mean Recall: we first compute the ratio between the correctly recommended artists and the number of artists in the validation set for each user, and then take the average of these values.

$$Mean\ Recall = \frac{\sum_{u \in Val} recall(recommend(u),\ A_u)}{|Val|}$$

$$where\ Val = \{u \mid u\ is\ in\ the\ validation\ set\}$$

$$and\ A_u = \{i \mid (u,i)\ is\ in\ the\ validation\ set\}$$

## Implementation

The ALS model was built using pyspark.ml.recommendation.ALS(*implicitPrefs*=True). The algorithm used by this class is based on (Hu et al., 2008), and has the following hyperparameters:

1. maxIter: number of iterations that the factorization runs. More iterations take more time but may produce a better factorization.

2. rank: The number of latent factors in the model

3. alpha: controls the relative weight of observed versus unobserved user-product interactions in the factorization.

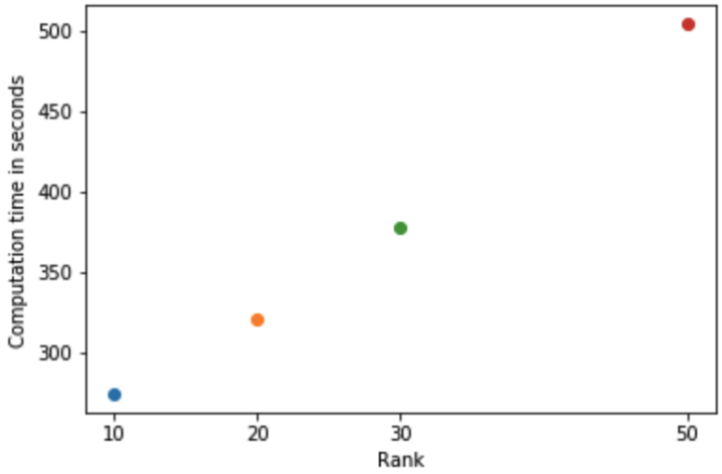
4. regParam: the regularization parameter $\lambda$.

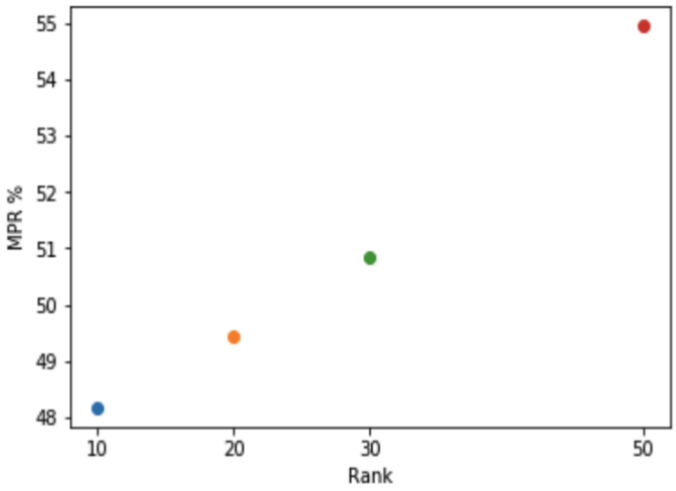
## Numerical evaluation

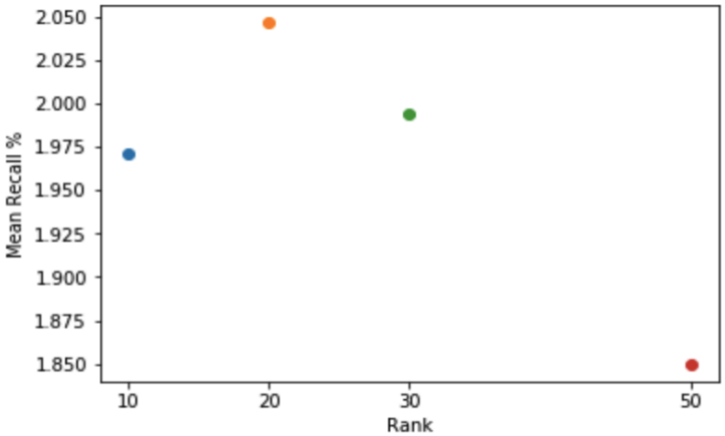
We first evaluated how the model performs on 10% of the data. The training set consisted of 80% of this subset, while the validation set consisted of 20% of the subset.

## Rank VS MPR, Computation Time , Mean Recall, Correct Recommendations - subset

The first evaluation aims to find the best number of latent factors for the model: we trained the same model with 10, 20, 30 and 50 latent factors - ranks.




| Rank | Correctly Predicted Artists |
|---|---|
| 10 | 6481 |
| 20 | 6741 |

| | |
|---|---|
| 30 | 6589 |
| 50 | 6078 |

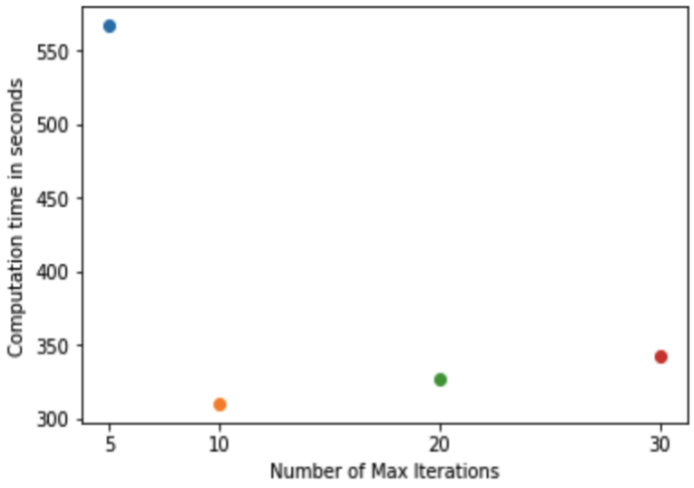
These results suggest that the larger the rank, the higher is the execution time, and the more likely it is for the recommended items to be recommended in the middle of the list of recommendations. Having a MPR close to 0.5 indicates that the model performs like an algorithm that places the recommended artists randomly in the list of top 5 recommended artists. Therefore, from an MPR point of view, the model with rank 10 performs the best.

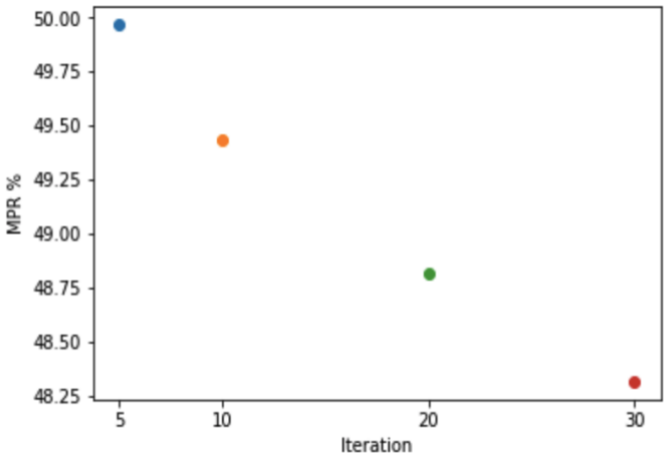
On the other hand, the mean recall does not follow a monotone trend. The best result is reached at rank = 20. Moreover, the largest number of correct recommendations is reached when the rank = 20.

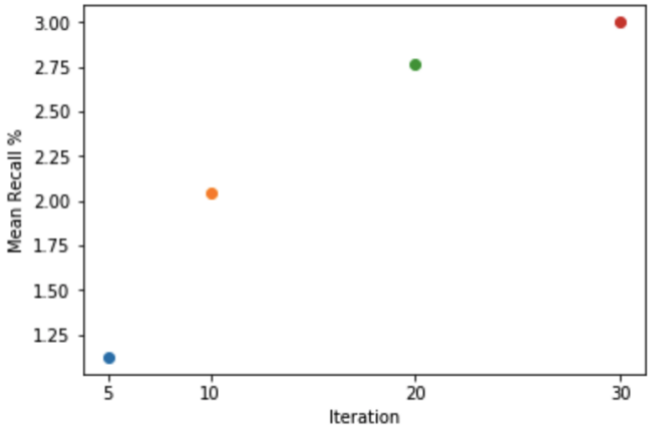
Since we value correct recommendations more important than the percentile ranking of the corrected items, we choose rank = 20 as the best hyperparameter for this dataset.

## Iterations VS MPR, Computation Time, Mean Recall, Correct Recommendations

The second evaluation is conducted on the number of iterations. The model is trained with rank = 20.






| Iterations | Correctly Predicted Artists |
|---|---|
| 5 | 3704 |
| 10 | 6741 |
| 20 | 9084 |
| 30 | 9843 |

The last two graphs show the same result: the more iterations we train the model, the better are the results. MPR reduces from 49.96% to 48.31%, and the Mean Recall reaches a maximum of 3%. The number of correctly recommended artists is 9843 when the number of iterations is 30, which is significantly larger than 3704, reached when the model runs for 5 iterations.

## Iterations VS Rank VS MPR, Computation Time, Mean Recall, Correct Recommendations – full dataset

We then evaluated the performance on the full dataset. The dataset was again split into training and validation set, of size 0.8 and 0.2.

| MPR | Rank = 10 | Rank = 30 |
|---|---|---|
| **Iterations = 10** | 47.067% | 46.57% |
| **Iterations = 20** | 47.08% | 46.49% |

| Mean Recall | Rank = 10 | Rank = 30 |
|---|---|---|
| **Iterations = 10** | 7.53% | 8.97% |
| **Iterations = 20** | 7.68% | 9.03% |

| Execution Time | Rank = 10 | Rank = 30 |
|---|---|---|
| **Iterations = 10** | 1748.37 s | 2167.12 s |
| **Iterations = 20** | 966.82 s | 1394.82 s |

| Correct Recommendations | Rank = 10 | Rank = 30 |
|---|---|---|
| **Iterations = 10** | 133,542 | 158,983 |
| **Iterations = 20** | 136,154 | 160,110 |

These results as very insightful. First, we can notice that big changes happen when we keep the iterations fixed and increase the rank, but not vice versa. In fact, the results do not significantly change as when change the number of iterations.

We achieve the best results for every metric when we train the model with 20 maximum iterations and 30 latent factors. For these values, we achieve a MPR of 46.49% and a Mean Recall of 9.03%, suggesting that this model can not only do better recommendations compared to the others, but can also recommend the true artists at the top of the recommendations list.

# 4. Spark Random Forest Regression

## Methodology

Random Forests Regression is a supervised learning algorithm. It is an ensemble of decision trees that can be used to predict labels. The algorithm is used to predict the total number of plays, given the features: gender, country, and age. For the algorithm to work, the features need to be of a float or integer type, so the gender and country columns are converted to dummy variables. This means that there are 310 features and 267,382 records. Since the dataset is relatively small, Google Cloud Storage is used instead of Hadoop. Furthermore, the cluster used has one master and two worker nodes.

### Partitioning

Spark MLlib is employed, and this supports the resilient distributed dataset (RDD) structure. Using the *.reparition()* function, it is possible to change the number of partitions. Choosing the right number of partitions can optimise the performance of the algorithm. So first, the performance of the RF algorithm on different numbers of partitions is explored. Generally, it is advised to use the same number of partitions as the number of cores in the cluster. (ProjectPro, 2022) I was not able to find the number of cores in the cluster, so instead, I tested the model on datasets that had 1 to 15 partitions. The execution time for each partition was recorded and plotted. From here, it was clear to see which partition gave the fastest runtime and therefore optimal performance.

### Hyperparameter Tuning

Following the approach in the paper, Tuning Random Forest Hyperparameters across Big Data Systems, by Ishna Kaul (Kaul, 2019). each parameter is varied whilst keeping the other values constant (set at the recommended values by Spark MLlib (Apache Spark, n.d.). The run time and the mean squared error for each are plotted against the varying parameters. The model that had the best mean squared error is used. For the number of trees parameter, values 2, 50, 100, 200 and 500 were used. Maximum depth has a range of values of 4, 8, 12,1 6 and 20. Finally, maximum bins has values, 50, 75, 100, 125 and 150.

### Adding more features

Initially, the artist columns is not considered because there are over 1000 distinct values. Because it is a string, it would need to be converted to a dummy variables, leading to a large amount of features. However, after running the initial dataset, we were interested to see how more features would affect the algorithm. As previously done, the runtime and mean squared error between the datasets with and without the artists are compared.

### Spark ML vs Spark MLlib

Spark ML and Spark MLlib are two of Sparks machine learning libraries. They both have similar algorithms but different APIs. Whilst Spark MLib supports RDDs, Spark ML supports dataframes

which is more versatile and flexible. In this project, the performance of both libraries will be explored using Random Forest Regression (RFR).

Due to time constraints a 1% sample of the dataset is taken. The time it took to run each algorithm and mean squared error was measured. Furthermore, the RFR on Spark ML library could not run the dataset with over many columns, so the country and artist columns were not used. The remaining features are therefore age and gender.

## Numerical Evaluation

### Partitioning

Overall having more partitions led to a faster execution. This makes sense, since the datasets are spread over more partitions, and therefore can be run more concurrent tasks. The figure below summarises the results obtained. Here you can see that the run time starts off very slow with one partition. The run time quickly reduces between two and four partitions. After then, it is relatively stable. Overall, partition 14 had the quickest runtime with an execution time of 13.9 seconds. Therefore, 14 partitions were used in the dataset for future calculations.

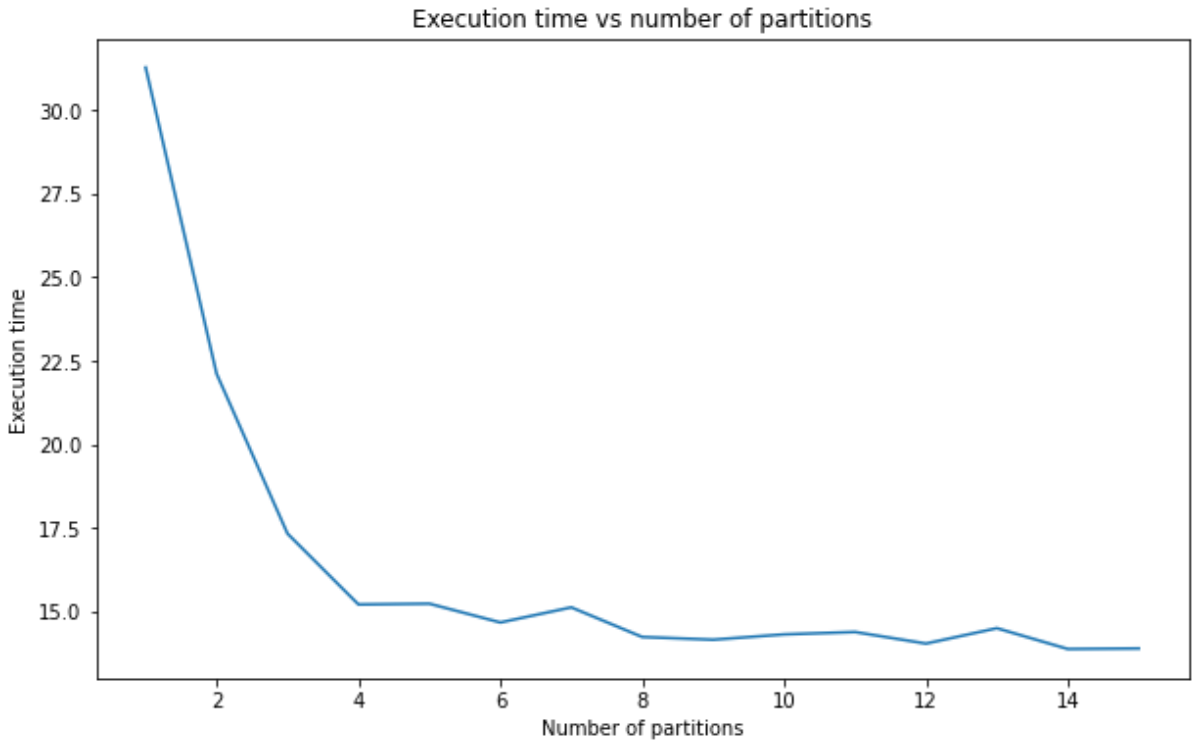



*Figure 4.1 Plot of time it takes to run Random Forest Model for partitions ranging between 2 and 14*

### Hyperparameter Tuning and Adding more features

As the number of trees and maximum depth increased, the execution time increased approximately linearly as expected. Moreover, there was no pattern observed in the changing of the number of bins, however the value of 125 gave the best MSE of 367613. The maximum depth of 20 gave the

best MSE of 381094 and the best number of trees was shown to be 2, with a value of 380426. The plots below shows the results for each parameter change.

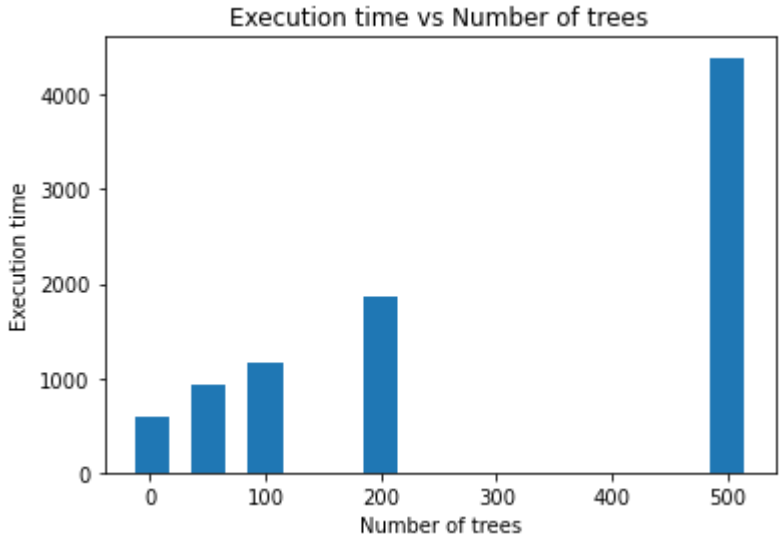



*Figure 4.2a(i) Plot of computation time at different numbers of trees*

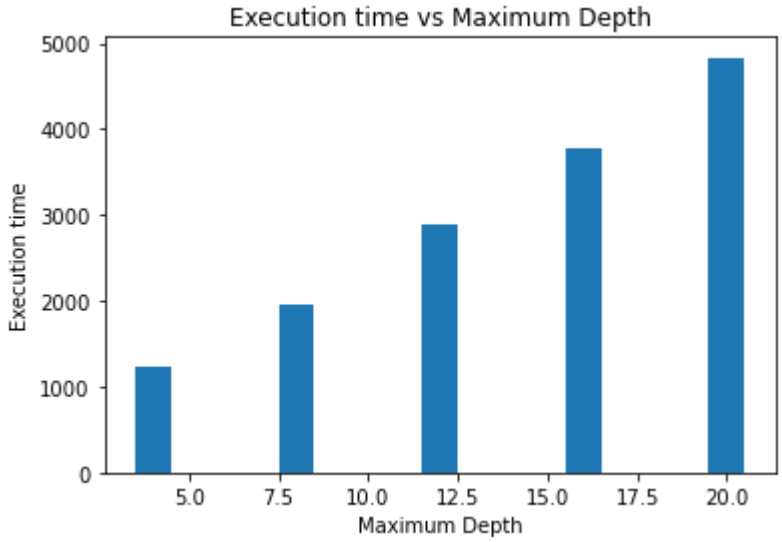



*Figure 4.2a(ii) Plot of computation time at different numbers of maximum tree depth*

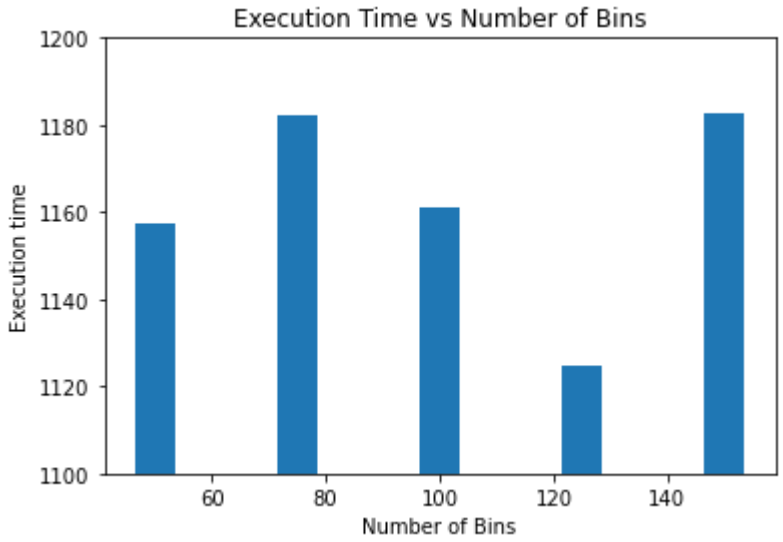



*Figure 4.2a(iii) Plot of computation time at different numbers of bins*

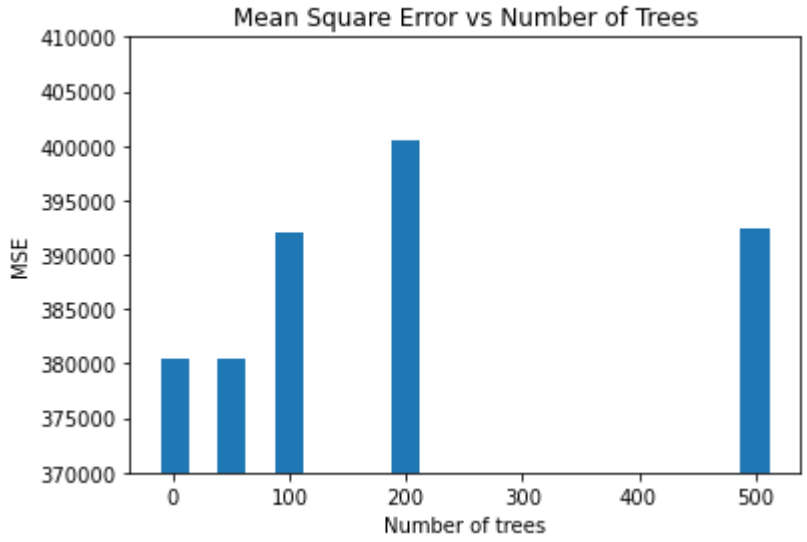



*Figure 4.2b(i) Plot of Mean Squared Error at different numbers of trees*

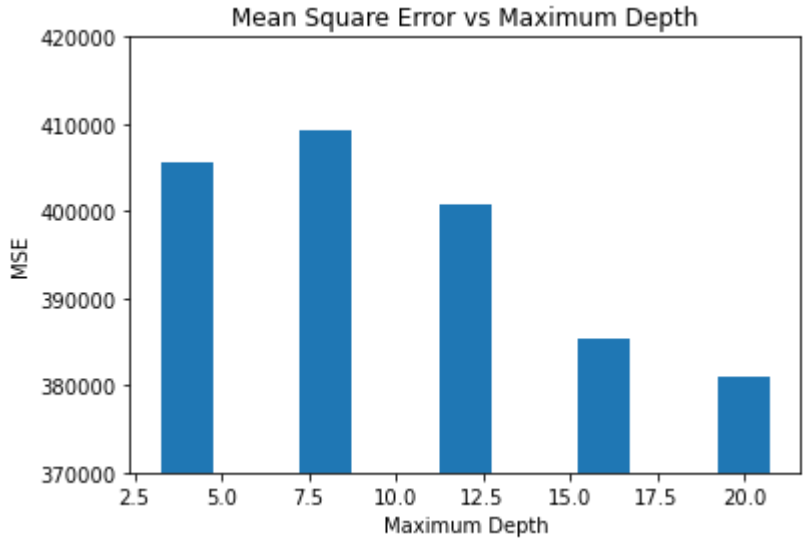



*Figure 4.2b(ii) Plot of Mean Squared Error at different maximum tree depths*

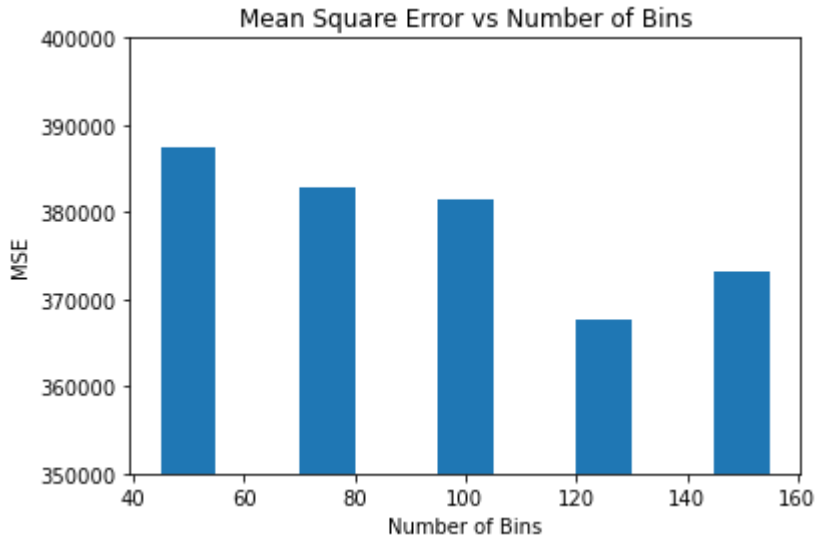



*Figure 4.2b(iii) Plot of Mean Squared Error at different numbers of bins*

Overall, the hyperparameter tuning approach used did not yield the most efficient results, with the runtime being much longer at 44 seconds. The standard settings gave a runtime of less than half, with a value of 16 seconds. Furthermore, using the standard settings gave a better MSE – see the results in the table below. This shows that the combination of the hyperparameters is crucial. Had there been more time, the grid search approach would have been used. This approach looks at different combination of hyperparameters and chooses the best one.

*Table 4.1 Comparing The hyperparameter tuning method and the standard / suggested parameter values suggested by Spark*

| | **Standard / suggested values** | **Hyperparameter tuning approach** |
|---|---|---|
| Mean Squared Error | 216814655 | 233690367 |
| Execution Time (seconds) | 16 | 44 |

Consequently, the standard default settings were used for the comparison between the current dataset (with the age, gender and country columns) and additional features. From the table below, it can be seen that whilst the dataset without the additional columns performs much faster, the mean squared error is higher. This is expected since adding the artist column would lead to the generation of over 100,000 additional features. Given this fact, the algorithm did not take as long as expected. Furthermore, the MSE having higher results, shows that the artist column may be a significant feature. Had there been more time, the feature importance tool in Spark ML would have been

explored to see which features contribute to the model most (Brownlee, 2020). The table below summarises the results.

Next, we wanted to see whether better results could be obtained using a different library – namely Spark ML.

*Table 4.2 Comparing Mean Squared Error and Runtime of datasets with and without artists added*

| | **Without artist name column** | **With artist name column** |
|---|---|---|
| Mean Squared Error | 431000 | 370000 |
| Execution Time (minutes) | 39 | 61 |

### Spark ML vs Spark MLlib

Since Spark ML is newer and is known to work at higher speeds (Data Flair, n.d.), we expected to see a much faster runtime. However, the runtime of Spark ML is more than triple than that of Spark MLlib having values of 22.4 seconds and 6.8 (suggested settings) seconds respectively. When the number of partitions is increased to 14 (the amount that was seen to give the best performance previously), the runtime is further reduced to 4 seconds.

*Table 4.3 Comparing Spark ML and Spark MLlib performance on Random Forest Regression*

| | **Spark ML** | **Spark MLlib (default - 7 partitions)** | **Spark MLlib (14 partitions)** |
|---|---|---|---|
| Mean Squared Error | 994691 | 481210 | 589375 |
| Execution Time (seconds) | 22.4 | 6.8 | 4.15 |

## 5. Implementation

| **Section** | **PySpark notebook** |
|---|---|

| | |
|---|---|
| User Profile Dataset | SparkRandomForestRegressor.ipynb |
| Graph Analysis | **Graph_analysis.ipynb** |
| Stochastic Gradient Descent methodology and evaluation | **SGD_0_001prc.ipynb** (for the 17K dataset)<br>**SGD_0_002prc.ipynb** (for the 35K dataset)<br><br>(Both notebooks have the same code. They differ on the results, since they were executed in different datasets) |
| Spark Random Forest Regression | SparkRandomForestRegressor.ipynb |
| ALS for Implicit Feedback | ALS for Implicit Feedback.ipynb |
| | |

## 6. Conclusion

Regarding the Stochastic Gradient Descent algorithm executed in a distributed fashion, we can conclude that the number of partitions do not need to be more than the number of workers, that the number of latent factors should be fairly small and the number of iterations not to exceed 3.

As for the ALS model, we see that, on the full dataset, the number of iterations does not have a strong positive effect on any of the metrics. On the other hand, the increase in number of latent factors produces significantly better results. The same does not apply on the smaller dataset, in which, by increasing the numbers of iterations, both the ability of the model to recommend the correct artists and to place the right artists at the top of the recommendations list. It would be interesting to verify whether the results keep improving as we keep increasing the number of the latent factors, as proved in (Hu et al., 2008).

With respect to Spark Random Forest Regression, 14 partitions were seen to give the best performance of the algorithm. Furthermore, Spark MLlib gave better results than Spark ML in terms of runtime and MSE. In order to explore all the columns more efficiently, a dimension reduction technique such as single value decompositions or principal component analysis could have been used to improve the mean squared error. Other model evaluation methods such as precision recall could have been considered also. In addition, for the hyperparameter tuning, grid search is a much more effective way to optimise the model.

## 7. References

Last.fm. (2010, March). *Music Recommendation Datasets for Research* . Retrieved from ocelma: http://ocelma.net/MusicRecommendationDataset/lastfm-360K.html

Baalbaki, W., 2016. Professor Reza Zadeh. https://stanford.edu/~rezab/classes/cme323/S16/projects_reports/baalbaki.pdf

Bosagh Zadeh, R., Meng, X., Ulanov, A., Yavuz, B., Pu, L., Venkataraman, S., Sparks, E., Staple, A. and Zaharia, M., 2016, August. Matrix computations and optimization in apache spark. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (pp. 31-38).

Gonzalez, J.E., Xin, R.S., Dave, A., Crankshaw, D., Franklin, M.J. and Stoica, I., 2014. {GraphX}: Graph Processing in a Distributed Dataflow Framework. In *11th USENIX symposium on operating systems design and implementation (OSDI 14)* (pp. 599-613).

Karau, H., Konwinski, A., Wendell, P. and Zaharia, M., 2015. *Learning spark: lightning-fast big data analysis*. " O'Reilly Media, Inc.".

Nitinsaroha, Stochastic-Gradient-Descent-for-Matrix-Factorization-on-spark, github https://github.com/nitinsaroha/Stochastic-Gradient-Descent-for-Matrix-Factorization-on-spark/blob/master/matrix_factorization_dsgd.py

Shvachko, K., Kuang, H., Radia, S. and Chansler, R., 2010, May. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)* (pp. 1-10). Ieee.

Apache Spark. (n.d.). *Random Forest*. Retrieved from Spark.Apache: https://spark.apache.org/docs/1.4.0/api/java/org/apache/spark/mllib/tree/RandomForest.html

Brownlee, J. (2020, August 30). *Machine Learning Mastery*. Retrieved from How to Calculate Feature Importance With Python: https://machinelearningmastery.com/calculate-feature-importance-with-python/

Data Flair. (n.d.). *Apache Spark RDD vs DataFrame vs DataSet*. Retrieved from data-flair.training: https://data-flair.training/blogs/apache-spark-rdd-vs-dataframe-vs-dataset/

Kaul, I. (2019, December 04). *Projects.* Retrieved from Washington: https://courses.cs.washington.edu/courses/csed516/20au/projects/p10.pdf

ProjectPro. (2022, April 26). *How Data Partitioning in Spark helps achieve more parallelism?* Retrieved from ProjectPro: https://www.projectpro.io/article/how-data-partitioning-in-spark-helps-achieve-more-parallelism/297#:~:text=The%20best%20way%20to%20decide,utilized%20in%20an%20optimal%20way

Koren, Y., Bell, R. and Volinsky, C., 2009. Matrix factorization techniques for recommender systems. *Computer*, *42*(8), pp.30-37.

Ryza, S., Laserson, U., Owen, S. and Wills, J., 2017. Advanced analytics with spark: patterns for learning from data at scale. " O'Reilly Media, Inc.".

Hu, Y., Koren, Y. and Volinsky, C., 2008, December. Collaborative filtering for implicit feedback datasets. In 2008 Eighth IEEE international conference on data mining (pp. 263-272). Ieee.

## 8. Contributions

| Github Account | LSE Student ID | Candidate Number | Contributions |
|---|---|---|---|
| Alexia-202065543 | 202065543 | 30499 | - Abstract<br>- Introduction - User-Artist plays Dataset<br>- Graph Analysis<br>- Stochastic Gradient Descent (Methodology, Numeric Evaluation, Implementation)<br>- Conclusion<br>- References |
| SA202055737 | 202055737 | 32456 | - Abstract<br>- Introduction - User profile dataset<br>- Spark Random Forest Regression (Methodology, Numeric Evaluation, Implementation)<br>- Conclusion<br>- References |
| 202146058 | 202146058 | 35387 | - Dataset choice and project proposal<br>- Abstract<br>- Introduction - User-Artist plays Dataset<br>- Alternating Least Squares (Methodology, Numeric Evaluation, Implementation)<br>- Conclusion<br>- References |