POLITECNICO DI BARI

Dipartimento di Ingegneria Elettrica e dell'Informazione – DEI

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

Big Data: Boardgame Recommender System

Professor:
Prof. Simona Colucci

Students:
Sergio STEFANIZZI
Pietro SPALLUTO

Anno Accademico 2021-2022

Contents

1 Introduction

The aim of this project is to build a board games recommender system through the use of two Spark's modules:

- **MLlib**: it provides an easy way to process data and implement various machine learning algorithms;
- **SparkSQL**: it allows powerful integration between python code and SQL queries.

This recommender system is based on implicit and explicit interactions between users and board games. Hence, a large dataset, containing this kind of information, has been used in order to perform precise predictions. Other datasets have been imported containing lots of game features, which are useful to obtain finer results. These datasets are taken from `boardgamegeek.com`, which is the most complete website in the field.

Real world data contains a lot of noise, so an appropriate data preprocessing step is needed to avoid wrong predictions. Several operations have been performed to solve this problem, such as:

- **Data Exploration**: data analysis;
- **Data Cleaning**: missing values replacement, outliers removal;
- **Dimensionality Reduction**: Principal Component Analysis (PCA).

Once problems about data have been settled, actual machine learning algorithms can be executed. In this case supervised learning is the right choice and the following categories of algorithms have been used:

- **Collaborative filtering**: Alternating Least Square (ALS);
- **Factorization Machines (FM)**: FM's Regression and classification;
- **Classification**: Logistic regression, decision tree and random forest.

Finally, results are compared by using some metrics, like Root Mean Square Error (RMSE) or accuracy, and with a special consideration on execution times.

In the following pages every aspect of the project is explained in detail.

2 Data Model

2.1 Datasets

The main dataset is `user_ratings.csv`. In Table 1 there is the representation of its schema. This dataset contains the explicit interactions between users and board games expressed by the `Rating`. There are almost 19 millions rows, which is a quite important information because it has the biggest impact on system performance.

Table 1: `user_ratings.csv`

Field name	Description
BGGId	BoardGameGeek game ID
Rating	Rating value
Username	User giving rating

Another important dataset is `games.csv` that holds information about each board game. This database requires a big data cleaning effort since there are many misleading information. With so many information a finer prediction can be done, according not only to the user-game interactions, but also to games similarities.

Table 2: `games.csv`

Field name	Description
BGGId	BoardGameGeek game ID
Name	Name of game
Description	Description, stripped of punctuation and lemmatized
YearPublished	First year game published
GameWeight	Game difficulty/complexity
AvgRating	Average user rating for gamed
BayesAvgRating	Bayes weighted average for game
StdDev	Standard deviation of Bayes Avg
MinPlayers	Minimum number of players
MaxPlayers	Maximun number of players
Columns[11:48]	Other board game's features

Other datasets are provided containing more features that are binary

encoded. For each game there is a column representing a category of a feature (features represented in this way are about artists, designers, mechanics, publishers, subcategories and themes).

Table 3: Other datasets

Dataset name	Description
`mechanics.csv`	Table of mechanics with binary flags per BGGId. Headers are various mechanics with binary flag
`themes.csv`	Headers are various themes with binary flag
`subcategories.csv`	Headers are various subcategories with binary flag
`artists_reduced.csv`	Artists for each BGGId. Artists are only listed if they have > 3 entries. Any artist(s) with ≤ 3 entries are tagged as binary under the catch-all "Low-Exp Artist"
`designers_reduced.csv`	Designers for each BGGId. Designers are only listed if they have > 3 entries. Any designer(s) with ≤ 3 entries are tagged as binary under the catch-all "Low-Exp Designer"
`publishers_reduced.csv`	Publishers for each BGGId. Publishers are only listed if they have > 3 entries. Any publishers(s) with ≤ 3 entries are tagged as binary under the catch-all "Low-Exp Publisher"
`ratings_distribution.csv`	Headers are ratings; data is number of ratings per BGGId per rating

2.2 Data preprocessing

To avoid giving to the recommender system wrong, incomplete, noisy or non numerical data it is very important to analyze and preprocess it.

2.2.1 Data Exploration

In a real world setting, data collected from explicit feedbacks like board games ratings can be very sparse and data is mostly collected from very popular items and highly engaged users. Large amount of less known items don't have ratings at all.

Below are shown the distributions of the board game ratings frequency and the one of the most active users. Both images presented a *long tail* distribution. Only a small fraction of the board games are rated frequently (Figure 1) as well as only few users give real contribution to the recommendation system (Figure 2).

For this and for storage and computational reasons, only meaningful board games and users have been used in the system (Listing 2).

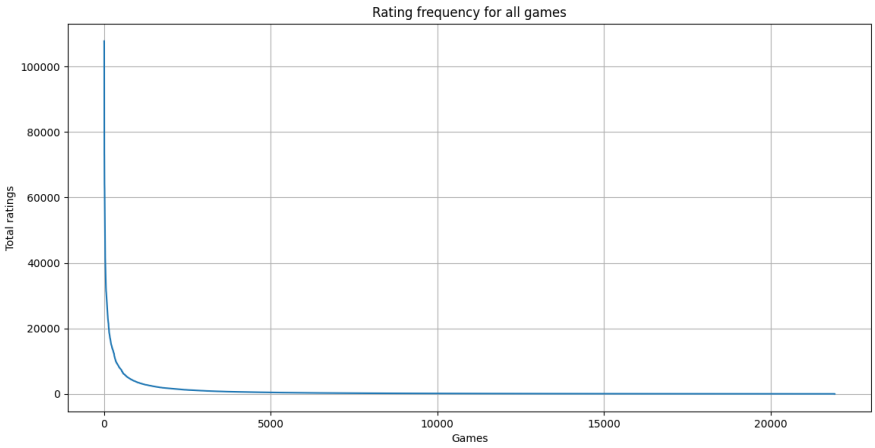


Figure 1: Ratings Frequency for all board games.

Rating values given by users are not uniformly distributed, as shown in Figure 3, because users tend not to give an high range of grades. When used for classification these ratings are subdivided into two bins. The first bin contains ratings below 7 which are not enough to recommend an item, the rest of ratings is put into the other one (Listing 1).

```
1 def discretize_ratings(df):
2     return df.withColumn('buckets', when((df['Rating'] < 7),
    lit(0.0)).otherwise(lit(1.0)))
```

Listing 1: Rating discretization.

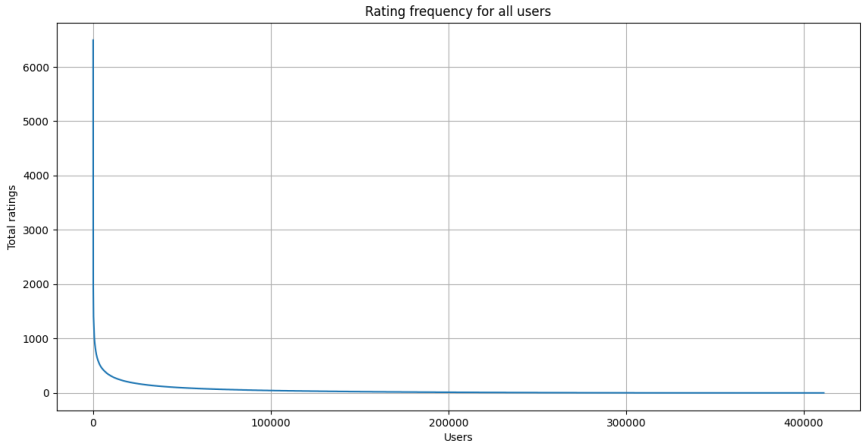


Figure 2: Ratings Frequency for all users.

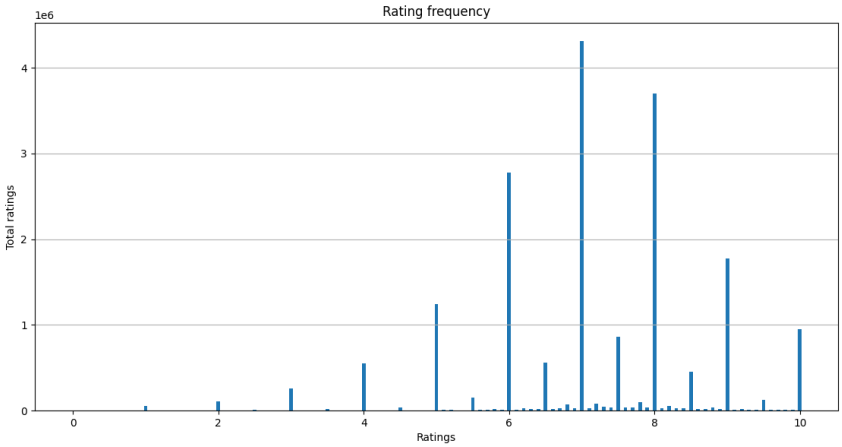
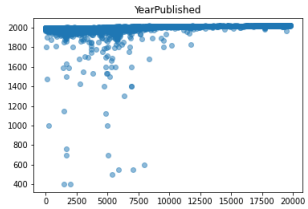
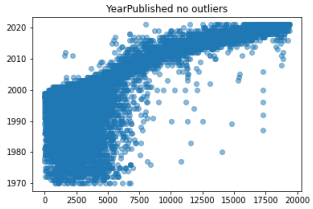


Figure 3: Ratings distribution for all games.

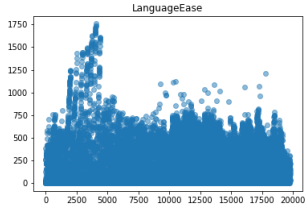
In the last phase is useful to identify possible outliers in game features by using scatter plots. Some of the plots are shown in Figure 4. Most features contain outliers, so, if they are not removed, predictions are highly influenced by noise, thus giving a bad result after executing any machine learning algorithm.



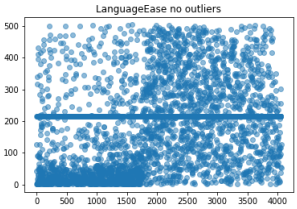
(a) Year of publication for each game.



(b) Year of publication for each game after outliers removal.



(c) Language requirement for each game.



(d) Language requirement for each game after outliers removal.

Figure 4: Scatter plots representing some of the features before (4a and 4c) and after (4b and 4d) outliers removal. It can be noted that data is very sparse initially.

2.2.2 Data Cleaning

After data exploration data structure is well understood and ready to be cleaned. First of all the focus is on `user_ratings.csv` dataset. Out of almost 19 million records, only a small fraction of them, just 63 records, contains missing values. So these samples are dropped.

`boardgamegeek.com` allows ratings between 0 and 10 with a step of 0.1. In the dataset some records have multiple decimal digit, so ratings are rounded to follow the rule.

Finally, board games that received a small number of reviews and users that reviewed just a few games are removed because they are not as significant as the other records to provide useful information to the system (Listing 2).

```
def low_importance_elements(df, numrec, numut):
    df1 = df.groupBy("BGGId").count()
    df1 = df1.filter(df1["count"] > numrec)
    df1 = df.join(df1, 'BGGId', "leftsemi")

```

```
    df2 = df.groupBy("Username").count()
    df2 = df2.filter(df2["count"] > numut)
    df2 = df.join(df2, 'Username', "leftsemi")

    return df1.join(df2, 'Username', "leftsemi")
```

Listing 2: Function to remove unimportant users or games.

Once the user ratings database is cleaned, another data cleaning step is done. There is more than one dataset containing games features. Most of them have hundreds or thousands columns due to binary representation. Hence, databases are transformed into a single column containing categorical features and afterword columns are joined to `games.csv`, the main games features database (Listing 3). This operation also reduces data dimensionality improving performance and predictions.

```
def make_categorical_db(themes_df, games_df, subcategories_df
    , mechanics_df, artists_df, designers_df, publishers_df):
    themes = []
    mechanics = []
    categories = []
    subcategories = []
    artists = []
    publishers = []
    designers = []

    categorical_features = pd.DataFrame()
    categorical_features['Themes'] = binary_to_categorical(
    themes, themes_df)
    categorical_features['Categories'] =
    binary_to_categorical(categories, games_df[games_df.
    columns[40:48]])
    categorical_features['Subcategories'] =
    binary_to_categorical(subcategories, subcategories_df)
    categorical_features['Mechanics'] = binary_to_categorical
    (mechanics, mechanics_df)
    categorical_features['Artists'] = binary_to_categorical(
    artists, artists_df)
    categorical_features['Designers'] = binary_to_categorical
    (designers, designers_df)
    categorical_features['Publishers'] =
    binary_to_categorical(publishers, publishers_df)

    return categorical_features
def binary_to_categorical(cat_features, cat_df):
    for game in range(0,len(cat_df)):
```

```
22          cat_list = [cat for cat in cat_df.columns if cat_df.
    iloc[game][cat] != 0 and cat != 'BGGId']
23          cat_features.append(', '.join(cat_list))
24      return cat_features
```

Listing 3: Binary representation of the features is transformed into a categorical representation.

This last database is joined to the one containing user-games interactions to obtain a set of data useful for most of the machine learning algorithms that will be applied.

The complete database contains a lot of irrelevant and incorrect data. First of all columns `Name`, `ImagePath`, `Description` are dropped because they do not add meaningful information. Then `BestPlayers` and `GoodPlayers` are removed because they contains a lot of missing values, this is due to the fact that is up to the user to insert this kind of information. Lastly `NumComments` is removed because it has only zeros.

Columns `YearPublished`, `MinPlayers`, `MaxPlayers`, `MfgPlaytime`, `ComMinPlaytime`, `ComMaxPlaytime`, `MfgAgeRec` have records with unwanted negative values, so these records are removed.

Missing values are shown in Figure 5 and Table 4 explains how missing values are removed or replaced.

Table 4: Chosen approach for removing missing values

Feature	Remove/replace approach
`Publishers`	Most frequent value (just 1 value)
`Designers`	Most frequent value
`Artists`	Dropped, too many missing values
`Subcategories`	Dropped, too many missing values
`Categories`	Dropped, too many missing values
`Machanics`	Most frequent value
`Themes`	Most frequent value
`Family`*	Arbitrary value (`No Family`)
`LanguageEase`	Average value
`ComAgeRec`	Average value

*Most games do not belong to a family, so missing values have a meaning.

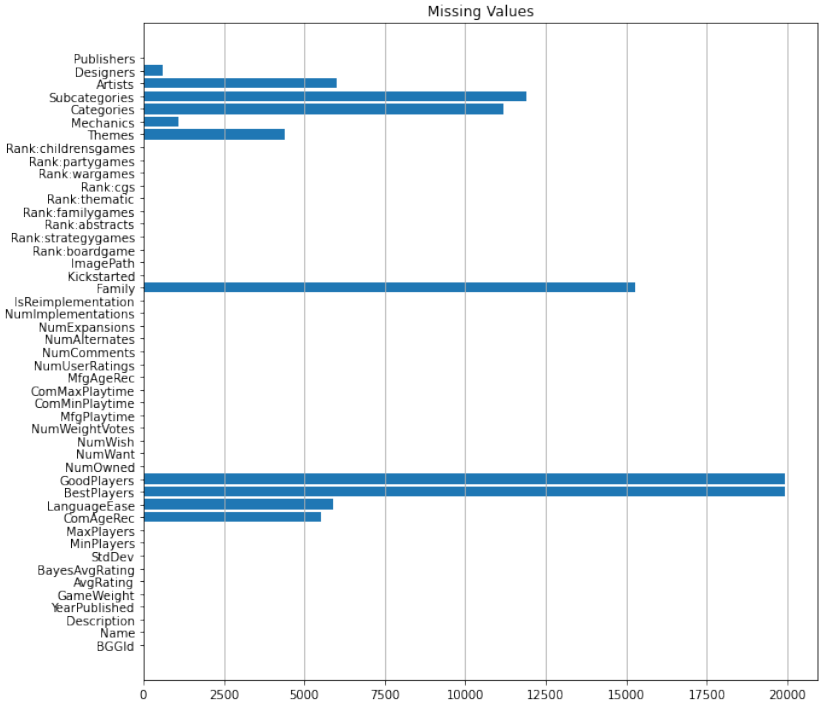


Figure 5: Missing values for every feature.

The last step is outliers removal. Interquartile ranges (IQRs) are used to determine if a value is an outlier. IQR is computed by using the formula: $IQR = Q_3 - Q_1$.

All values that are less than $Q_1 - IQR$ or greater than $Q_3 + IQR$ are considered outliers, thus the corresponding record is removed. This phase produces some zero variance features that can be removed. Code for data cleaning of the complete database is shown in Listing 4.

```
def clean_complete_database(complete_df):
    complete_df.drop('Name', axis=1, inplace=True)
    complete_df.drop('ImagePath', axis=1, inplace=True)
    complete_df.drop('Description', axis=1, inplace=True)
    complete_df.drop('GoodPlayers', axis=1, inplace=True)
    complete_df.drop('BestPlayers', axis=1, inplace=True)
    complete_df.drop('NumComments', axis=1, inplace=True)
    complete_df.drop('Rank:strategygames', axis=1, inplace=
   True)
    complete_df.drop('Rank:abstracts', axis=1, inplace=True)
    complete_df.drop('Rank:familygames', axis=1, inplace=True
```

```
    )
    complete_df.drop('Rank:thematic', axis=1, inplace=True)
    complete_df.drop('Rank:cgs', axis=1, inplace=True)
    complete_df.drop('Rank:wargames', axis=1, inplace=True)
    complete_df.drop('Rank:partygames', axis=1, inplace=True)
    complete_df.drop('Rank:childrensgames', axis=1, inplace=
    True)
    complete_df = complete_df[complete_df['YearPublished']>0]
    complete_df = complete_df[complete_df['MinPlayers']>0]
    complete_df = complete_df[complete_df['MaxPlayers']>0]
    complete_df = complete_df[complete_df['MfgPlaytime']>0]
    complete_df = complete_df[complete_df['ComMinPlaytime'
    ]>0]
    complete_df = complete_df[complete_df['ComMaxPlaytime'
    ]>0]
    complete_df = complete_df[complete_df['MfgAgeRec']>0]
    complete_df['Family'].fillna('No family', inplace=True)
    complete_df['ComAgeRec'].fillna(complete_df['ComAgeRec'].
    mean(), inplace=True)
    complete_df['LanguageEase'].fillna(complete_df['
    LanguageEase'].mean(), inplace=True)
    most_frequent_theme = find_most_frequent_value(
    complete_df, 'Themes')
    complete_df['Themes'].fillna(most_frequent_theme, inplace
    =True)
    most_frequent_mechanic = find_most_frequent_value(
    complete_df, 'Mechanics')
    complete_df['Mechanics'].fillna(most_frequent_mechanic,
    inplace=True)
    most_frequent_publisher = find_most_frequent_value(
    complete_df, 'Publishers')
    complete_df['Publishers'].fillna(most_frequent_publisher,
     inplace=True)
    most_frequent_designer = find_most_frequent_value(
    complete_df, 'Designers')
    complete_df['Designers'].fillna(most_frequent_designer,
    inplace=True)
    complete_df.drop('Artists', axis=1, inplace=True)
    complete_df.drop('Categories', axis=1, inplace=True)
    complete_df.drop('Subcategories', axis=1, inplace=True)
    complete_df = remove_outliers(complete_df)
    mask = [complete_df.std()<=1e-10]
    drop_cols = complete_df.select_dtypes('number').columns[
    mask]
    for col in drop_cols:
```

```
41          complete_df.drop(col, axis=1, inplace=True)
42      return complete_df
43  def find_most_frequent_value(dataframe, feature):
44      features = dataframe.groupby(by='{}'.format(feature))
45      features = {key: len(value) for key, value in features.
    groups.items()}
46      features = pd.DataFrame(data=features.values(), index=
    features.keys())
47      features.sort_values(by=0, ascending=False, inplace=True)
48      return features.iloc[0].name
49  def remove_outliers(df):
50      for column in df.columns:
51          if not column in ['BGGId', 'Description', 'Name', '
    GoodPlayers', 'Family', 'ImagePath', 'Themes',
52                            'Mechanics', 'Categories', '
    Subcategories', 'Artists', 'Designers', 'Publishers',
53                            'NumComments', 'BestPlayers', '
    Kickstarted', 'IsReimplementation']:
54              print(column)
55      for column in df.columns:
56          if not column in ['BGGId', 'Description', 'Name', '
    GoodPlayers', 'Family', 'ImagePath', 'Themes',
57                            'Mechanics', 'Categories', '
    Subcategories', 'Artists', 'Designers', 'Publishers',
58                            'NumComments', 'BestPlayers', '
    Kickstarted', 'IsReimplementation']:
59              if column != 'YearPublished':
60                  quant_df = df.quantile([0.25, 0.75])
61                  IQR = stats.iqr(df[column])
62                  if IQR != 0:
63                      df = df[(df[column] > quant_df.loc[0.25][
    column] - IQR) & (df[column] < quant_df.loc[0.75][column]
    + IQR)]
64                  else:
65                      df = df[(df[column] >= quant_df.loc
    [0.25][column]) & (df[column] <= quant_df.loc[0.75][column
    ])]
66              else:
67                  df = df[df[column] > 1970]
68              print('{} {}'.format(column, df.shape[0]))
69      return df
```

Listing 4: Data cleaning of `games.csv`

Normalization is a fundamental operation to have the same scale for every feature in order to give them the same weight during the training process.

In this case Min-Max Scaling is used, which is based of the formula:

$$X' = a + \frac{(X - X_{min})(b - a)}{X_{max} - X_{min}}$$

Where X' is the normalized value, X_{min} and X_{max} are respectively the minimum and the maximum values of X, a and b are the two boundaries in which values are going to fit in. In this case a=0 and b=1.

Finally the correlation between the features can be represent through a correlation matrix (Figure 6). This heatmap shows that many features are highly correlated, so a dimensionality reduction algorithm can be used to solve this issue. From this matrix the Pearson's coefficients in Table 5 can be obtained.

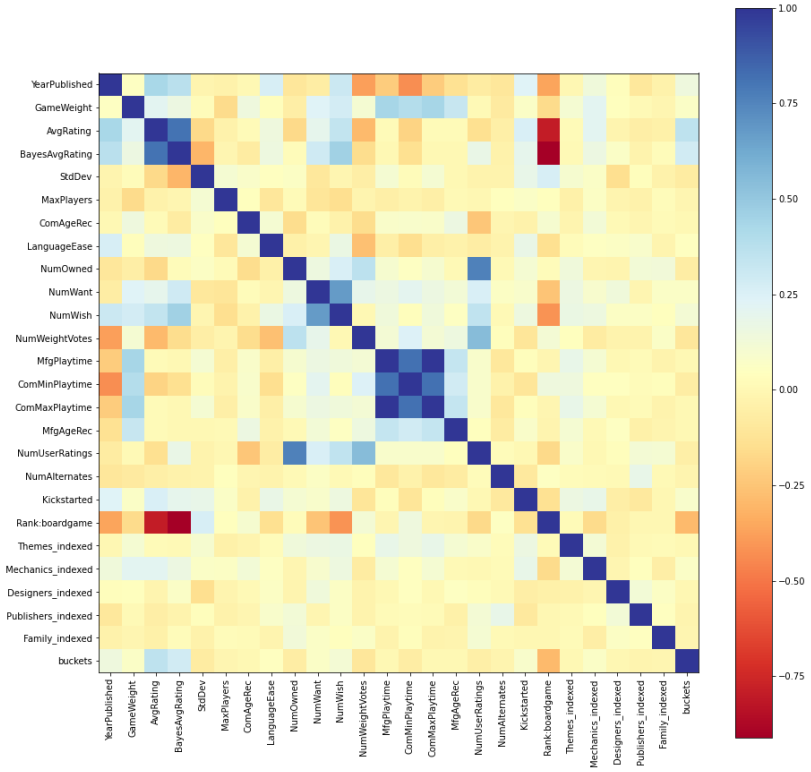


Figure 6: Correlation Matrix

Table 5: Pearson's coefficients for each feature.

Feature	Pearson's coefficient
AvgRating	0.350752
BayesAvgRating	0.288893
YearPublished	0.147991
NumWish	0.116927
Kickstarted	0.087283
Mechanics_indexed	0.072319
GameWeight	0.070770
NumWant	0.069797
LanguageEase	0.051300
MfgAgeRec	0.007185
MfgPlaytime	0.005123
ComMaxPlaytime	0.005123
ComAgeRec	0.003858
Themes_indexed	0.003792
Designers_indexed	-0.003740
MaxPlayers	-0.012867
Family_indexed	-0.013139
NumAlternates	-0.018976
Publishers_indexed	-0.021669
NumUserRatings	-0.047952
NumOwned	-0.060988
ComMinPlaytime	-0.066761
StdDev	-0.072845
NumWeightVotes	-0.097473
Rank:boardgame	-0.284706

2.2.3 Dimensionality Reduction

To avoid the *Curse of Dimensionality* it is needed a dimensionality reduction algorithm. Curse of Dimensionality decreases the performance because, when you increase the number of features, training samples become more sparse and to avoid that data should grow exponentially, which, in most of the cases, is impossible.

For this project Principal Component Analysis (PCA) has been used.

This method maximizes the variance expressed by data and reduces dimensionality by projecting data points in a lower dimensional space. Principal Components obtained do not have a name, but the heatmap in Figure 7 explains the meaning of every component. By projecting data points in a 15-dimensional space more than 90% of the variance is retained as shown in Figure 8.

Table 6 shows the new Pearson's Coefficients.

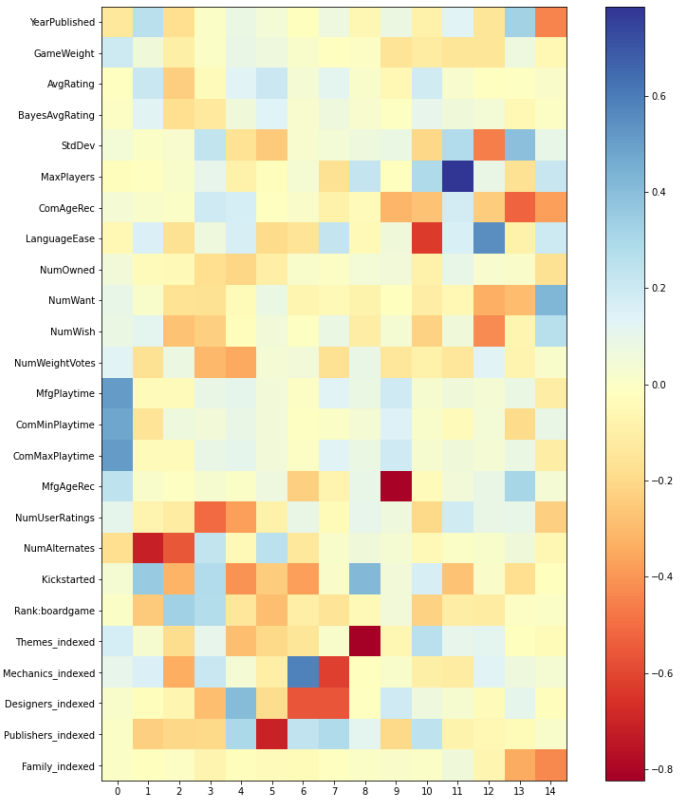


Figure 7: Correlation between original features and Principal Components

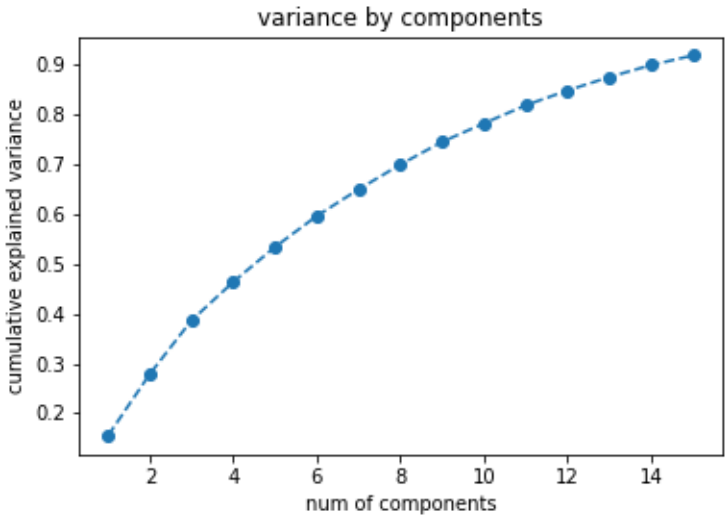


Figure 8: Comulative sum of the percentage of the variance explained by each principal component.

Table 6: New Pearson's coefficient for each principal component

Principal Component	Pearson's coefficient
1	0.178170
5	0.125516
10	0.088621
4	0.087333
7	0.059174
6	0.022698
8	0.010365
11	0.005687
12	0.003214
14	0.002767
13	-0.011019
0	-0.017349
9	-0.026524
3	-0.028799
2	-0.184503

3 Implementation

After data preprocessing, it is possible to implement the recommender system.

The system is structured in several python files, in order to simplify the readability of the code.

- `bgrfunctions.py`: contains functions used in the other scripts
- `datacleaning.py`: every step of the data cleaning is performed
- Recommender algorithms:
  - `als.py`
  - `fmclass.py`
  - `fmreg.py`
  - `logreg.py`
  - `dectree.py`
  - `randforest.py`

Most of the machine learning algorithm are designed to work only with numbers. For this reason every non numerical categorical feature is converted to an integer label (Listing 5). The class `StringIndexer` has been used to perform the conversion. The indices are in [0, numLabels). By default, this is ordered by label frequencies so the most frequent label gets index 0.

```
1 def encode_categorical_features(complete):
2     complete = complete.withColumn('NumOwned', complete['
    NumOwned'].cast(IntegerType()))
3     complete = complete.withColumn('Kickstarted', complete['
    Kickstarted'].cast(BooleanType()))
4     complete = complete.withColumn('Rank:boardgame', complete
    ['Rank:boardgame'].cast(IntegerType()))
5     complete = complete.withColumn('YearPublished', complete[
    'YearPublished'].cast(IntegerType()))
6
7     stringIndexer = StringIndexer(inputCol="Themes",
    outputCol="Themes_indexed")
8     complete_indexed = stringIndexer.fit(complete).transform(
    complete)
9     stringIndexer = StringIndexer(inputCol="Mechanics",
    outputCol="Mechanics_indexed")
```

```
    complete_indexed = stringIndexer.fit(complete_indexed).
    transform(complete_indexed)
    stringIndexer = StringIndexer(inputCol="Designers",
    outputCol="Designers_indexed")
    complete_indexed = stringIndexer.fit(complete_indexed).
    transform(complete_indexed)
    stringIndexer = StringIndexer(inputCol="Publishers",
    outputCol="Publishers_indexed")
    complete_indexed = stringIndexer.fit(complete_indexed).
    transform(complete_indexed)
    stringIndexer = StringIndexer(inputCol="Family",
    outputCol="Family_indexed")
    complete_indexed = stringIndexer.fit(complete_indexed).
    transform(complete_indexed)

    complete_indexed = complete_indexed.drop('Themes')
    complete_indexed = complete_indexed.drop('Mechanics')
    complete_indexed = complete_indexed.drop('Designers')
    complete_indexed = complete_indexed.drop('Publishers')
    complete_indexed = complete_indexed.drop('Family')

    return complete_indexed
```

Listing 5: Categorical features encoding.

Boardgame ID and user ID are one-hot encoded through `OneHotEncoder`. This encoding allows algorithms which expect continuous features to use categorical features. Every learning method is applied on the dataset using `TrainValidationSplit` which used 80% of data as training set. Through this function it is possible to give the machine learning algorithm a set of hyper-parameters that would be validated with the remaining 20% of the original data as shown in Listing 6.

Regarding the hyper-parameters, it is possible to retrieve the best combination of them after `TrainValidationSplit` is done, also shown in Listing 6.

```
def als_prediction(df):
    (training, test) = df.randomSplit([0.8, 0.2])
    als = ALS(userCol='UserId', itemCol='BGGId', ratingCol='
    Rating', coldStartStrategy='drop', maxIter=20, seed=1)
    evaluator = RegressionEvaluator(metricName='rmse',
    labelCol='Rating', predictionCol='prediction')
    evaluator2 = RegressionEvaluator(metricName='r2',labelCol
    ='Rating', predictionCol='prediction')
    paramGrid = ParamGridBuilder()\
```

```
        .addGrid(als.rank, [20, 30]) \
        .addGrid(als.regParam, [0.1, 0.01]) \
        .build()
    tvs = TrainValidationSplit(estimator=als,
    estimatorParamMaps=paramGrid,evaluator=evaluator,
    trainRatio=0.8)
    model = tvs.fit(training)
    prediction = model.transform(test)
    best=model.bestModel
    rmse = evaluator.evaluate(prediction)
    r2 = evaluator2.evaluate(prediction)
    return prediction, rmse, r2, best
```

Listing 6: Example of a machine learning algorithm implemented using a function.

Before any machine learning algorithm, feature columns are merged into a single vector column by using `VectorAssembler`.

3.1 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.ml` 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.ml` uses the Alternating Least Squares algorithm to learn these latent factors through matrix factorization. The idea is basically to take a large matrix and factor it into two or more lower dimensional matrices close to the original one through an iterative optimization process.

The original matrix is $\mathbf{R}$ of size $u \times i$ and contains users, board games and ratings. In $\mathbf{U}$ and $\mathbf{V}$ there are weights for how each user/board game relates to each latent factor. Then $\mathbf{U}$ and $\mathbf{V}$ are computed so that their product approximates $\mathbf{R}$ as closely as possible: $\mathbf{R} \approx \mathbf{U} \times \mathbf{V}$ (Figure 9).

The implementation of ALS is shown below in Listing 7.

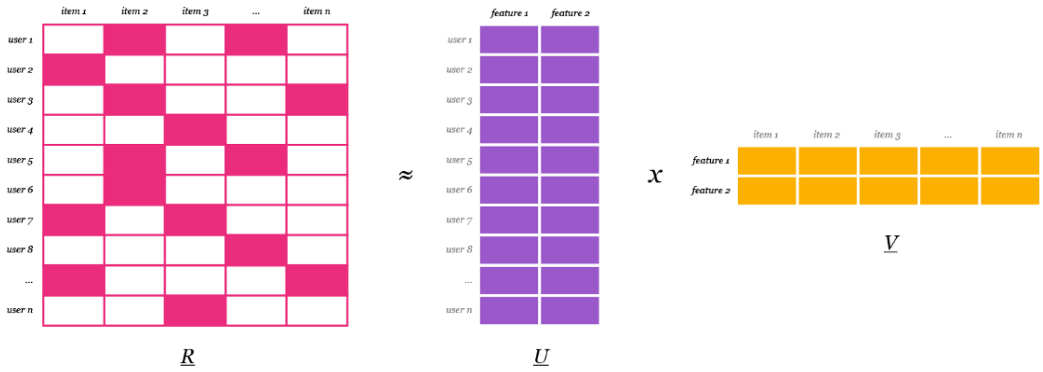


Figure 9: ALS representation.

```
start_time=time.time()
numrec=1000
numut=10
pathIN='/home/sergio/spark-3.2.1-bin-hadoop3.2/esame/data/
    user_ratings.csv'
pathINgames='/home/sergio/spark-3.2.1-bin-hadoop3.2/esame/
    data/games.csv'

spark=SparkSession.builder.appName('boardgamesrecsys').
    getOrCreate()
df=spark.read.options(delimiter=',',header=True,inferSchema=
    True).csv(pathIN)
dfgames=spark.read.options(delimiter=',',header=True,
    inferSchema=True).csv(pathINgames)

indexer=StringIndexer(inputCol='Username',outputCol='UserId')
df=indexer.fit(df).transform(df)

df1 = bgrf.low_importance_elements(df, numrec, numut);
df1=df1.withColumn('Rating', round(df['Rating'], 1))

startpred=time.time()
prediction, rmse, r2, best = bgrf.als_prediction(df1)
endpred=time.time()

print('rmse metric ' + str(rmse))
print('r2 metric = ' + str(r2))
print('Best rank '+str(best.rank))
print('Best regParam '+str(best._java_obj.parent().
    getRegParam()))
print('Best maxIter '+str(best._java_obj.parent().getMaxIter
```

```
    ()))
26
27 userRecs = best.recommendForAllUsers(10)
28 userRecs = userRecs\
29     .withColumn('rec', explode('recommendations'))\
30     .select('UserId', col('rec.BGGId'), col('rec.Rating'))
31 dfgames=dfgames.select('BGGId', 'Name')
32 userRecs=userRecs.join(dfgames,['BGGId'])
33 userRecs=userRecs.withColumn('Rating', round(userRecs['Rating
    '], 1))
34 print('Recommendation of a random user')
35 userRecs.filter(userRecs['UserId']==random.randint(1, df.
    select('UserId').distinct().count())).show()
36 print('ALS Execution time: %s min' %((endpred-startpred)/60))
37 print('Execution time: %s min' %((time.time()-start_time)/60)
    )
38
39 spark.stop()
```

Listing 7: ALS implementation.

ALS model in `spark.ml` is a powerful instrument. Indeed, it provides useful recommendation tool such as `recommendForAllUsers(numItems)` that returns the top `numItems` items recommended for each user, for all users. This method is implemented in the code above (in this case just for a random user) and it is called on the best model among every model generated by the `TrainValidationSplit` on a set of hyper-parameters.

3.2 Factorization Machine

3.2.1 Regression

Factorization Machines (FM) are generic supervised learning models that map arbitrary real-valued features into a low-dimensional latent factor space and can be applied naturally to a wide variety of prediction tasks including regression, classification, and ranking. FMs can estimate model parameters accurately under very sparse data and train with linear complexity, allowing them to scale to very large data sets. Therefore these characteristics make FMs an ideal fit for real-world recommendation problems. Unlike the classic Matrix Factorization model which inputs a user-item interaction matrix, FM models represent user-item interactions as tuples of real-valued feature vectors and numeric target variables.

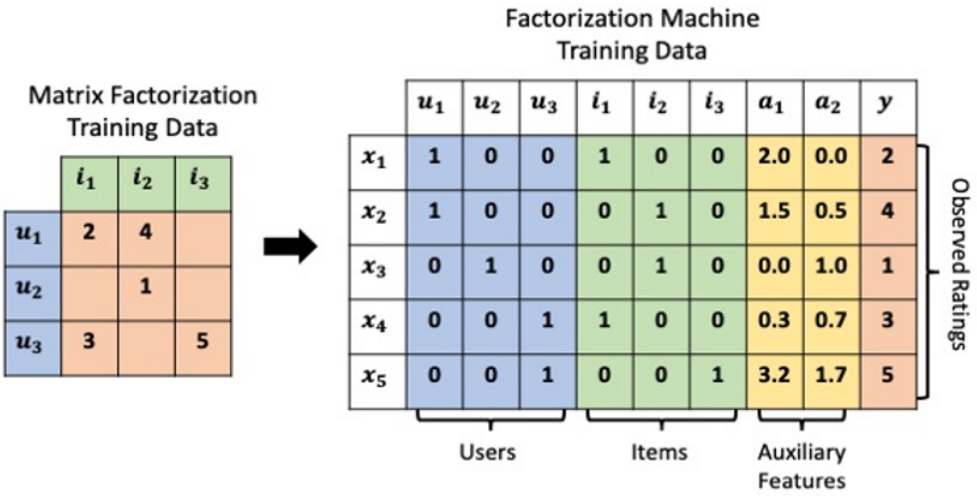


Figure 10: Factorization Machine.

Differently from linear regression models, FM introduces weights that combine users and items interactions. FM's model function is a order 2 polynomial function that can learn a weight w_{ij} for each feature combination. The function is:

$$\widehat{y}(\mathbf{x}) = w_0 + \sum_{i=1}^{n} w_i x_i + \sum_{i=1}^{n} \sum_{j=i+1}^{n} x_i x_j w_{ij}$$

Where w_0 is the global bias, w_i are the weights for the features x_i $\forall i$ and w_{ij} are the weights for the feature combination $x_i x_j$ $\forall i$ $\forall j$. A polynomial model can capture features interaction.

```
spark = SparkSession.builder.appName("boardgamesrecsys").
    getOrCreate()
main_path = '/home/sergio/spark-3.2.1-bin-hadoop3.2/esame'
n_pc = 15
complete = spark.read.csv('{}/data/clean/complete_indexed.csv
    '.format(main_path), inferSchema=True, header=True, sep=',
    ')
df = spark.read.csv('{}/data/clean/user_ratings_disc.csv'.
    format(main_path), inferSchema=True, header=True, sep=',')

indexer = StringIndexer(inputCol="Username", outputCol="
    UserId")
ohe = OneHotEncoder(inputCols=['UserId', 'BGGId'], outputCols
    =['UserId_onehot', 'BGGId_onehot'])
```

```
9
10 pipeline = Pipeline(stages=[indexer, ohe])
11 df = pipeline.fit(df).transform(df)
12
13 games_ratings = complete.join(df, 'BGGId', 'inner')
14
15 assembler = VectorAssembler(inputCols=games_ratings.drop('
   Username', 'Rating', 'BGGId', 'UserId', 'UserId_onehot', '
   BGGId_onehot').columns,
16                             outputCol='features')
17
18
19 scaler = MinMaxScaler(inputCol='features', outputCol='
   scaled_features')
20 pipeline = Pipeline(stages=[assembler, scaler])
21 games_ratings = pipeline.fit(games_ratings).transform(
   games_ratings)
22
23 pca_features = bgrf.principal_components_analysis(
   games_ratings, n_pc)
24
25 startpred=time.time()
26 prediction, rmse, r2, best = bgrf.fmreg_prediction(
   pca_features)
27 endpred=time.time()
28 prediction.show()
29 print("FM Regressor Execution time: %s min" %((endpred-
   startpred)/60))
30 print('Best StepSize '+str(best._java_obj.parent().
   getStepSize()))
31 print('Best FactorSize '+str(best._java_obj.parent().
   getFactorSize()))
32 print("rmse metric = " + str(rmse))
33 print("r2 metric = " + str(r2))
34
35 spark.stop()
```

Listing 8: FM Regressor implementation.

3.2.2 Classification

As said before FM is also applicable for classification tasks. SparkML provides `FMClassifier`. For the board games recommender system a classification is more appropriate because ratings are discrete values. So the classifier has been implemented in order to compare it to the regressor. Unfortunately `FMClassifier` does not support multilabel classification but only the binary one. For this reason ratings have been divided into two groups. The first one contains ratings below 7 and it is identified with `0` and the second one contains ratings from 7 to 10 and is identified with `1`. The logic behind this is due to the fact that an item should be recommended to a user only if it has a rating equal or greater than 7. By splitting ratings into two sets their distribution is more uniform as shown in Figure 11.

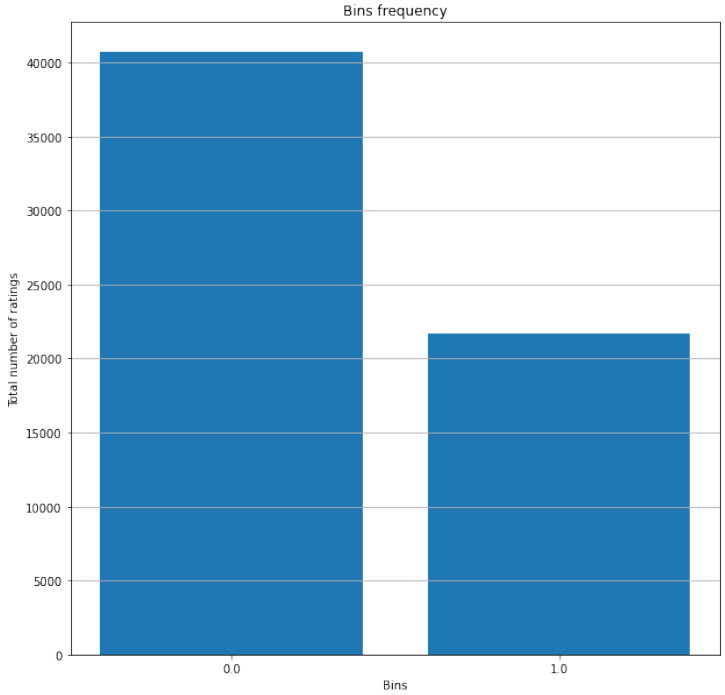


Figure 11: New rating distribution after the splitting.

```
spark = SparkSession.builder.appName("boardgamesrecsys").
    getOrCreate()
main_path = '/home/sergio/spark-3.2.1-bin-hadoop3.2/esame'
n_pc = 15

complete = spark.read.csv('{}/data/clean/complete_indexed.csv
    '.format(main_path), inferSchema=True, header=True, sep=',
    ')
df = spark.read.csv('{}/data/clean/user_ratings_disc.csv'.
    format(main_path), inferSchema=True, header=True, sep=',')

indexer = StringIndexer(inputCol="Username", outputCol="
    UserId")
ohe = OneHotEncoder(inputCols=['UserId', 'BGGId'], outputCols
    =['UserId_onehot', 'BGGId_onehot'])

pipeline = Pipeline(stages=[indexer, ohe])
df = pipeline.fit(df).transform(df)

games_ratings = complete.join(df, 'BGGId', 'inner')

assembler = VectorAssembler(inputCols=games_ratings.drop('
    Username', 'Rating', 'BGGId', 'UserId', 'UserId_onehot', '
    BGGId_onehot','buckets').columns,
                                outputCol='features')

scaler = MinMaxScaler(inputCol='features', outputCol='
    scaled_features')
pipeline = Pipeline(stages=[assembler, scaler])
games_ratings = pipeline.fit(games_ratings).transform(
    games_ratings)

pca_features = bgrf.principal_components_analysis(
    games_ratings, n_pc)

startpred=time.time()
prediction, acc, roc, step_size ,factor_size = bgrf.
    fmclas_prediction(pca_features)
endpred=time.time()
prediction.show()
print("FMClassifier Execution time: %s min" %((endpred-
    startpred)/60))
print('Best StepSize '+str(best._java_obj.parent().
    getStepSize()))
```

```
print('Best FactorSize '+str(best._java_obj.parent().
    getFactorSize()))
print("Test set accuracy = " + str(acc))
print("Test set roc = " + str(roc))

spark.stop()
```

Listing 9: FM Classifier implementation.

3.3 Logistic Regression

Logistic regression is another powerful supervised ML algorithm used for binary classification problems when target is categorical. The best way to think about logistic regression is that it is a linear regression but for classification problems. Logistic regression essentially uses a logistic function defined below to model a binary output variable. The primary difference between linear regression and logistic regression is that logistic regression's range is bounded between 0 and 1. In addition, as opposed to linear regression, logistic regression does not require a linear relationship between inputs and output variables.

```
spark = SparkSession.builder.appName("boardgamesrecsys").
    getOrCreate()

main_path = '/home/sergio/spark-3.2.1-bin-hadoop3.2/esame'
n_pc = 15

complete = spark.read.csv('{}/data/clean/complete_indexed.csv
    '.format(main_path), inferSchema=True, header=True, sep=',
    ')
df = spark.read.csv('{}/data/clean/user_ratings_disc.csv'.
    format(main_path), inferSchema=True, header=True, sep=',')

indexer = StringIndexer(inputCol="Username", outputCol="
    UserId")
ohe = OneHotEncoder(inputCols=['UserId', 'BGGId'], outputCols
    =['UserId_onehot', 'BGGId_onehot'])

pipeline = Pipeline(stages=[indexer, ohe])
df = pipeline.fit(df).transform(df)

games_ratings = complete.join(df, 'BGGId', 'inner')

```

```
assembler = VectorAssembler(inputCols=games_ratings.drop('Username', 'Rating', 'BGGId', 'UserId', 'UserId_onehot', 'BGGId_onehot','buckets').columns,
                                outputCol='features')

scaler = MinMaxScaler(inputCol='features', outputCol='scaled_features')
pipeline = Pipeline(stages=[assembler, scaler])
games_ratings = pipeline.fit(games_ratings).transform(games_ratings)

pca_features = bgrf.principal_components_analysis(games_ratings, n_pc)

startpred=time.time()
prediction, acc, roc, best = bgrf.logistic_regression(pca_features)
endpred=time.time()

prediction.show()
print("LogisticRegression Execution time: %s min\n" %((endpred-startpred)/60))
print('Best regParam '+str(best._java_obj.parent().getRegParam()))
print('Best maxIter '+str(best._java_obj.parent().getMaxIter()))
print("Test set accuracy = " + str(acc))
print("Test set roc = " + str(roc))
spark.stop()
```

Listing 10: Logistic Regression implementation.

3.4 Decision Tree

Decision Trees (DTs) are a non-parametric supervised learning method used for classification and regression. The goal is to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features. A tree can be seen as a piecewise constant approximation. A decision tree is drawn upside down with its root at the top. The nodes represent the condition that splits in branches or edges. The end of the branch that doesn't split anymore is the leaf or rather the decision.

Some advantages of decision tree are:

- Simple to understand, interpret, visualize;

- Can handle both numerical and categorical data. Can also handle multi-output problems;
- Nonlinear relationships between parameters do not affect tree performance.

On the other hand some disadvantages are:

- Decision-tree learners can create over-complex trees that do not generalize the data well. This is called *overfitting*;
- Decision tree learners create biased trees if some classes dominate

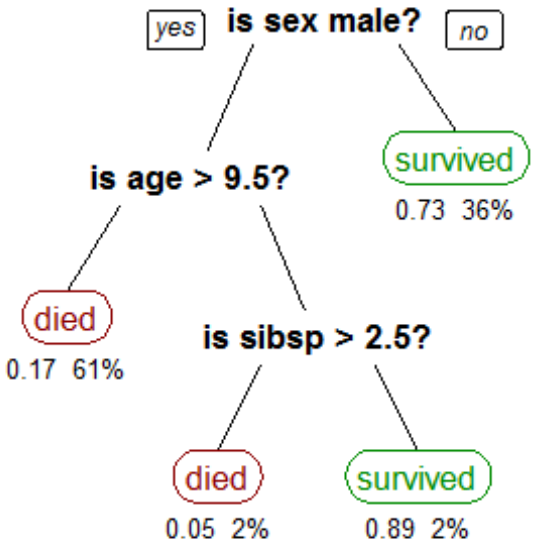


Figure 12: Decision tree example.

```
spark = SparkSession.builder.appName("boardgamesrecsys").
    getOrCreate()

main_path = '/home/sergio/spark-3.2.1-bin-hadoop3.2/esame'
n_pc = 15

complete = spark.read.csv('{}/data/clean/complete_indexed.csv
    '.format(main_path), inferSchema=True, header=True, sep=',
    ')
df = spark.read.csv('{}/data/clean/user_ratings_disc.csv'.
    format(main_path), inferSchema=True, header=True, sep=',')

indexer = StringIndexer(inputCol="Username", outputCol="
    UserId")
ohe = OneHotEncoder(inputCols=['UserId', 'BGGId'], outputCols
    =['UserId_onehot', 'BGGId_onehot'])

pipeline = Pipeline(stages=[indexer, ohe])
df = pipeline.fit(df).transform(df)

games_ratings = complete.join(df, 'BGGId', 'inner')

assembler = VectorAssembler(inputCols=games_ratings.drop('
    Username', 'Rating', 'BGGId', 'UserId', 'UserId_onehot', '
    BGGId_onehot','buckets').columns,  outputCol='features')

scaler = MinMaxScaler(inputCol='features', outputCol='
    scaled_features')
pipeline = Pipeline(stages=[assembler, scaler])
games_ratings = pipeline.fit(games_ratings).transform(
    games_ratings)

pca_features = bgrf.principal_components_analysis(
    games_ratings, n_pc)

startpred=time.time()
prediction, acc, roc, best = bgrf.decision_tree_class(
    pca_features)
endpred=time.time()

prediction.show()
print("Decision Tree Execution time: %s min" %((endpred-
    startpred)/60))
print('Best MaxDepth '+str(best._java_obj.parent().
    getMaxDepth()))
```

```
32 print('Best MinInfoGain '+str(best._java_obj.parent().
       getMinInfoGain()))
33 print("Test set accuracy = " + str(acc))
34 print("Test set roc = " + str(roc))
35
36 spark.stop()
```

Listing 11: Decision Tree implementation.

3.5 Random Forest

Random Forest is an ensemble learning method in which a collection of trees is used to select the prediction by voting or averaging the results of every tree. Each tree is trained on a bootstrap dataset, that is a dataset with the same number of samples as the original one but they are randomly selected with repetitions. The purpose of the bootstrap is to decrease the variance of the forest estimator. Indeed, individual decision trees typically exhibit high variance and tend to overfit. The injected randomness in forests yield decision trees with somewhat decoupled prediction errors. By taking an average of those predictions, some errors can cancel out. Random forests achieve a reduced variance by combining diverse trees, sometimes at the cost of a slight increase in bias. In practice the variance reduction is often significant hence yielding an overall better model.

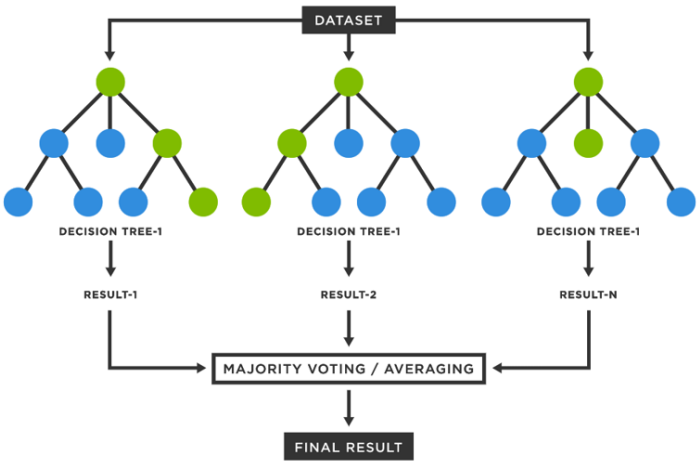


Figure 13: Random Forest example

```
spark = SparkSession.builder.appName("boardgamesrecsys").
    getOrCreate()

main_path = '/home/sergio/spark-3.2.1-bin-hadoop3.2/esame'
n_pc = 15

complete = spark.read.csv('{}/data/clean/complete_indexed.csv
    '.format(main_path), inferSchema=True, header=True, sep=',
    ')
df = spark.read.csv('{}/data/clean/user_ratings_disc.csv'.
    format(main_path), inferSchema=True, header=True, sep=',')

indexer = StringIndexer(inputCol="Username", outputCol="
    UserId")
ohe = OneHotEncoder(inputCols=['UserId', 'BGGId'], outputCols
    =['UserId_onehot', 'BGGId_onehot'])

pipeline = Pipeline(stages=[indexer, ohe])
df = pipeline.fit(df).transform(df)

games_ratings = complete.join(df, 'BGGId', 'inner')

assembler = VectorAssembler(inputCols=games_ratings.drop('
    Username', 'Rating', 'BGGId', 'UserId', 'UserId_onehot', '
    BGGId_onehot','buckets').columns, outputCol='features')

scaler = MinMaxScaler(inputCol='features', outputCol='
    scaled_features')
pipeline = Pipeline(stages=[assembler, scaler])
games_ratings = pipeline.fit(games_ratings).transform(
    games_ratings)

pca_features = bgrf.principal_components_analysis(
    games_ratings, n_pc)

startpred=time.time()
prediction, acc, roc, best = bgrf.random_forest_classifier(
    pca_features)
endpred=time.time()
prediction.show()
print("Random Forest Execution time: %s min" %((endpred-
    startpred)/60))
print('Best MaxDepth '+str(best._java_obj.parent().
    getMaxDepth()))
```

```
print('Best MinInfoGain '+str(best._java_obj.parent().
    getMinInfoGain()))
print("NumTrees = " + str(best._java_obj.parent().getNumTrees
    ()))
print("Test set accuracy = " + str(acc))
print("Test set roc = " + str(roc))

spark.stop()
```

Listing 12: Random Forest implementation.

4 Results and Conclusion

In Table 7 are shown the metrics used to evaluate each machine learning method.

For `ALS` and `FMRegressor` that predict continuous values, Root-mean-square error (RMSE) is used as evaluator. Also the coefficient of determination (R2) is considered for a better understanding of the results.

In the other hand, for `FMClassifier`, `Logistic Regression`, `Decision Tree` and `Random Forest`, that predict discrete values, accuracy and area under ROC are used.

Table 7: Results and execution times of each machine learning algorithm.

	RMSE	R2	Accuracy	ROC	Execution time (mins)
ALS	1.10	0.42	-	-	49.8
FM Regressor	1.38	0.28	-	-	121.98
FM Classifier	-	-	0.91	0.53	283.68
Logistic Regression	-	-	0.91	0.54	47.96
Decision Tree	-	-	0.91	0.5	74.91
Random Forest	-	-	0.92	0.5	61.52

Table 8: Best parameters selected for each algorithm.

	maxIter	rank	regParam	stepSize	factorSize	maxDepth	minInfoGain	numTrees
ALS	20	20	0.1	-	-	-	-	-
FM Regressor	100	-	0	0.1	1	-	-	-
FM Classifier	100	-	0	0.1	1	-	-	-
Logistic Regression	50	-	0.01	-	-	-	-	-
Decision Tree	-	-	-	-	-	5	20	-
Random Forest	-	-	-	-	-	5	20	30

In Table 8 are shown, for each machine learning algorithm, every parameter of the best model returned from the `TrainValidationSplit`. As said in section 3.1, ALS gives the opportunity to do a top-10 recommendation list for each user. After ALS is implemented, the system picks a random user and shows the ten best recommended board games.

```
+------+------+------+--------------------+
| BGGId|UserId|Rating|                Name|
+------+------+------+--------------------+
|182028| 33848|   9.3|Through the Ages:...|
|280794| 33848|   9.1|         Etherfields|
| 60153| 33848|   9.1|War of the Ring C...|
|324856| 33848|   9.0|The Crew: Mission...|
| 25613| 33848|   9.0|Through the Ages:...|
| 18098| 33848|   9.0|  Napoleon's Triumph|
|314040| 33848|   8.9|Pandemic Legacy: ...|
|221107| 33848|   8.9|Pandemic Legacy: ...|
|248562| 33848|   8.8|Mage Knight: Ulti...|
|169427| 33848|   8.8|Middara: Unintent...|
+------+------+------+--------------------+
```

Figure 14: Top-10 recommendation for a random user

In conclusion, `ALS` seems to be the most reliable and versatile method for the implementation of a Recommender System. This is due to the value of its metrics and the reduced execution time, unlike the `FMClassifier` that takes nearly 5 hours.