Dataset:

In this project, we are going to work with two datasets to implement Collaborative Filtering and Content-Based song recommendation system. Datasets:

1- **MillionSongDataset (MSD):** We are working with a subset of this dataset(10k out of 350k) (http://millionsongdataset.com/pages/getting-dataset/) It is possible to access the whole dataset by AWS, however it is not free. Although we are using the subset of this dataset, the code is **scalable**.

2- Implicit rating of 1million User(number of time a song played): http://millionsongdataset.com/tasteprofile/

Packages:

In [99]:
```
import numpy as np
import pandas as pd
import tables   # Note: before installing pytables package, install HDF5, Numexpr, Cython, c-blosc packages
import h5py
import os
import fnmatch
import sys

import matplotlib.pyplot as plt
import seaborn as sns

from pyspark.sql import SparkSession, Row
from pyspark import SparkContext
from pyspark.sql.types import *

from pyspark.ml.feature import MinMaxScaler
from pyspark.ml.feature import VectorAssembler
from pyspark.ml import Pipeline
from pyspark.sql.functions import udf
from pyspark.sql.types import DoubleType
from pyspark_dist_explore import hist # install by "pip install pyspark_dist_explore"

from pyspark.sql.functions import col,isnan,when,count

from pyspark.ml.evaluation import RegressionEvaluator
from pyspark.ml.recommendation import ALS
from pyspark.sql import Row
from pyspark.ml.feature import StringIndexer

import pyspark.sql.functions as F
from pyspark.sql.types import ArrayType, DoubleType
from pyspark.ml.feature import RobustScaler, StandardScaler

import sklearn
from sklearn.metrics.pairwise import cosine_similarity

from pyspark.sql.functions import row_number, monotonically_increasing_id
from pyspark.sql import Window
from pyspark.sql.window import Window
from pyspark.sql.functions import rank, col
```

Create spark cluster:

In [100…
```
spark = SparkSession.builder.appName("Milion Songs Dataset").getOrCreate()
sc = SparkContext.getOrCreate()
```

path to datasets:

In [101…
```
data_MSD_path = 'MillionSongSubset'
data_imp_rating_path = 'train_triplets/train_triplets.txt'
```

1-First dataset:

1.1 Read the MSD dataset from HDF5 directories:

- In order to read HDF5 file, HDFStore function of pandas library is used to read each .h5 file and keep in pyspark dataframe
- Each file(.h5) contains three keys **'/analysis/songs/', '/metadata/songs/', '/musicbrainz/songs/'**
- **Each key allow us to access data and metadata stored in the dataset**

Load the data in parallel with the help of spark RDD

In [102…
```
# load a sample of data to see the columns
hdf = pd.HDFStore(data_MSD_path+'/A/A/A/TRAAAAW128F429D538.h5',mode ='r', header = False)
df1 = hdf.get('/analysis/songs/')
df2 = hdf.get('/metadata/songs/')
df3 = hdf.get('/musicbrainz/songs/')
hdf.close()
```

```
sample_data_MSD = pd.concat([df1,df2,df3], axis = 1)
print(sample_data_MSD.T)
```

```
                                                                  0
analysis_sample_rate                                          22050
audio_md5                          a222795e07cd65b7a530f1346f520649
danceability                                                    0.0
duration                                                  218.93179
end_of_fade_in                                                0.247
energy                                                          0.0
idx_bars_confidence                                               0
idx_bars_start                                                    0
idx_beats_confidence                                              0
idx_beats_start                                                   0
idx_sections_confidence                                           0
idx_sections_start                                                0
idx_segments_confidence                                           0
idx_segments_loudness_max                                         0
idx_segments_loudness_max_time                                    0
idx_segments_loudness_start                                       0
idx_segments_pitches                                              0
idx_segments_start                                                0
idx_segments_timbre                                               0
idx_tatums_confidence                                             0
idx_tatums_start                                                  0
key                                                               1
key_confidence                                                0.736
loudness                                                    -11.197
mode                                                              0
mode_confidence                                               0.636
start_of_fade_out                                           218.932
tempo                                                        92.198
time_signature                                                    4
time_signature_confidence                                     0.778
track_id                                         TRAAAAW128F429D538
analyzer_version
artist_7digitalid                                            165270
artist_familiarity                                         0.581794
artist_hotttnesss                                          0.401998
artist_id                                        ARD7TVE1187B99BFB1
artist_latitude                                                 NaN
artist_location                                     California - LA
artist_longitude                                                NaN
artist_mbid                    e77e51a5-4761-45b3-9847-2051f811e366
artist_name                                                  Casual
artist_playmeid                                                4479
genre
idx_artist_terms                                                  0
idx_similar_artists                                               0
release                                                 Fear Itself
release_7digitalid                                           300848
song_hotttnesss                                             0.60212
song_id                                          SOMZWCG12A8C13C480
title                                                I Didn't Mean To
track_7digitalid                                            3401791
idx_artist_mbtags                                                 0
year                                                              0
```

1.2 Extracting desirable features:

Following columns are chosen to be used in this project.

In [103…

```
attribs=['song_id',
        'title',
        'artist_id',
        'duration',
        'key',
        'loudness',
        'mode',
        'tempo',
        'time_signature',
        'song_hotttnesss',
        'artist_hotttnesss',
        'artist_familiarity',
        'year'
       ]
```

In [104…

```
# This function is used in RDD to read each .h5 file and return a list of string(data of columns)
# f: directory path from spark.wholeTextFiles() function
# d: dataset directory path
def read_h5(f,d):
    # prune the file path is essential here, because the wholeTextFile function returns the absolute path
    hdf = pd.HDFStore(f[f.index(d):],mode ='r', header = False)  # Openning HDFStore to read .h5 file

    # The dataset is HDFS file with 3 main key {analysis, metadata, musicbrains} which 'songs' key allow us
    # to access the data of each song(It should be mentioned that there are)
    df1 = hdf.get('/analysis/songs/')
    df2 = hdf.get('/metadata/songs/')
    df3 = hdf.get('/musicbrainz/songs/')

    hdf.close()                                               # Closing HDFStore

    # concatenate all columns together in a dataframe and pick our desired features
    df_concat = pd.concat([df1,df2,df3], axis = 1)[attribs]
```

```
    # return the result as a list of string to be able to store in rdd
    return df_concat.values.tolist()[0]
```

In [105…
```
# find all path of all files without loading the files
rdd = sc.wholeTextFiles(data_MSD_path+'/*/*/*/*.h5').map(lambda x: read_h5(x[0],data_MSD_path))
```

In [106…
```
rdd.first()
```

Out[106]:
```
['SOMZWCG12A8C13C480',
 "I Didn't Mean To",
 'ARD7TVE1187B99BFB1',
 218.93179,
 1,
 -11.197,
 0,
 92.198,
 4,
 0.6021199899057548,
 0.4019975433642836,
 0.5817937658450281,
 0]
```

- **As you see the first imported data to rdd is the same as the above sample (irrelevant columns are eliminated here)**

Createing Spark dataframe from the RDD:

In [107…
```
# creating the schema for spark dataframe
schema = StructType([
    StructField('song_id', StringType(), True),
    StructField('title', StringType(), True),
    StructField('artist_id', StringType(), True),
    StructField('duration', FloatType(), True),
    StructField('key', IntegerType(), True),
    StructField('loudness', FloatType(), True),
    StructField('mode', IntegerType(), True),
    StructField('tempo', FloatType(), True),
    StructField('time_signature', IntegerType(), True),
    StructField('song_hotttnesss', FloatType(), True),
    StructField('artist_hotttnesss', FloatType(), True),
    StructField('artist_familiarity', FloatType(), True),
    StructField('year', IntegerType(), True)
])
```

In [108…
```
data_MSD = spark.createDataFrame(rdd, schema)
```

In [15]:
```
# here the data is loaded to the memory
data_MSD.toPandas().describe().T
```

Out[15]:

	count	mean	std	min	25%	50%	75%	max
duration	10000.0	238.507278	114.137314	1.044440	176.032196	223.059143	276.375061	1819.767700
key	10000.0	5.276100	3.554087	0.000000	2.000000	5.000000	8.000000	11.000000
loudness	10000.0	-10.485654	5.399786	-51.643002	-13.163250	-9.380000	-6.532500	0.566000
mode	10000.0	0.691100	0.462063	0.000000	0.000000	1.000000	1.000000	1.000000
tempo	10000.0	122.915512	35.184418	0.000000	96.965752	120.161003	144.013245	262.828003
time_signature	10000.0	3.564800	1.266239	0.000000	3.000000	4.000000	4.000000	7.000000
song_hotttnesss	5648.0	0.342822	0.247218	0.000000	0.000000	0.360371	0.537504	1.000000
artist_hotttnesss	10000.0	0.385552	0.143647	0.000000	0.325266	0.380742	0.453858	1.082503
artist_familiarity	9996.0	0.565457	0.160161	0.000000	0.467611	0.563666	0.668020	1.000000
year	10000.0	934.704600	996.650657	0.000000	0.000000	0.000000	2000.000000	2010.000000

1.3 Data Preprocessing:

1.3.1 Dealing with null values

- **Checking Null, empty, None, Nan values for each column:**

In [109…
```
# Loop through all columns for each row and count: empty, None, Null, Nan
ps_df = data_MSD.select([count(when(col(c).contains('None') | col(c).contains('NULL') | (col(c) == '' ) | col(c).isNull
                         c )).alias(c) for c in data_MSD.columns])
```

In [17]:
```
ps_df.toPandas()
```

Out[17]:

	song_id	title	artist_id	duration	key	loudness	mode	tempo	time_signature	song_hotttnesss	artist_hotttnesss	artist_familiarity	year
0	0	2	0	0	0	0	0	0	0	4352	0	4	0

Based on the result, song_hotttnesss feature has huge amount of null values. Thus we decide to not use this feature and remove the column. Title feature has only 2 missing values, since we are not going to work with this feature it is left untouch.

For the artist_familiarity feature we just remove 4 missing values. Year feature: By looking at the describtion of the dataset, min value is 0, which is not valid. Thus, we need to investigate more and count the zero values.

In [18]:
```
data_MSD.filter(data_MSD['year'] == '0').count()
```

Out[18]:
```
5320
```

- **Since the number of 0 values in year are too much and it is better to remove this feature:**

In [110…
```
data_MSD = data_MSD.drop("year")
```

- **The same for song_hotttnesss**

In [111…
```
data_MSD = data_MSD.drop("song_hotttnesss")
```

- **Remove null values of artist_familiarity:**

In [112…
```
data_MSD = data_MSD.dropna(subset=['artist_familiarity'],how='all')
```

Making sure the changes being applied:

In [113…
```
ps_df = data_MSD.select([count(when(col(c).contains('None') | col(c).contains('NULL') | (col(c) == '' ) | col(c).isNull
                                     c )).alias(c) for c in data_MSD.columns])
```

In [23]:
```
ps_df.toPandas()
```

Out[23]:

	song_id	title	artist_id	duration	key	loudness	mode	tempo	time_signature	artist_hotttnesss	artist_familiarity
0	0	2	0	0	0	0	0	0	0	0	0

1.3.2 Observing all features:

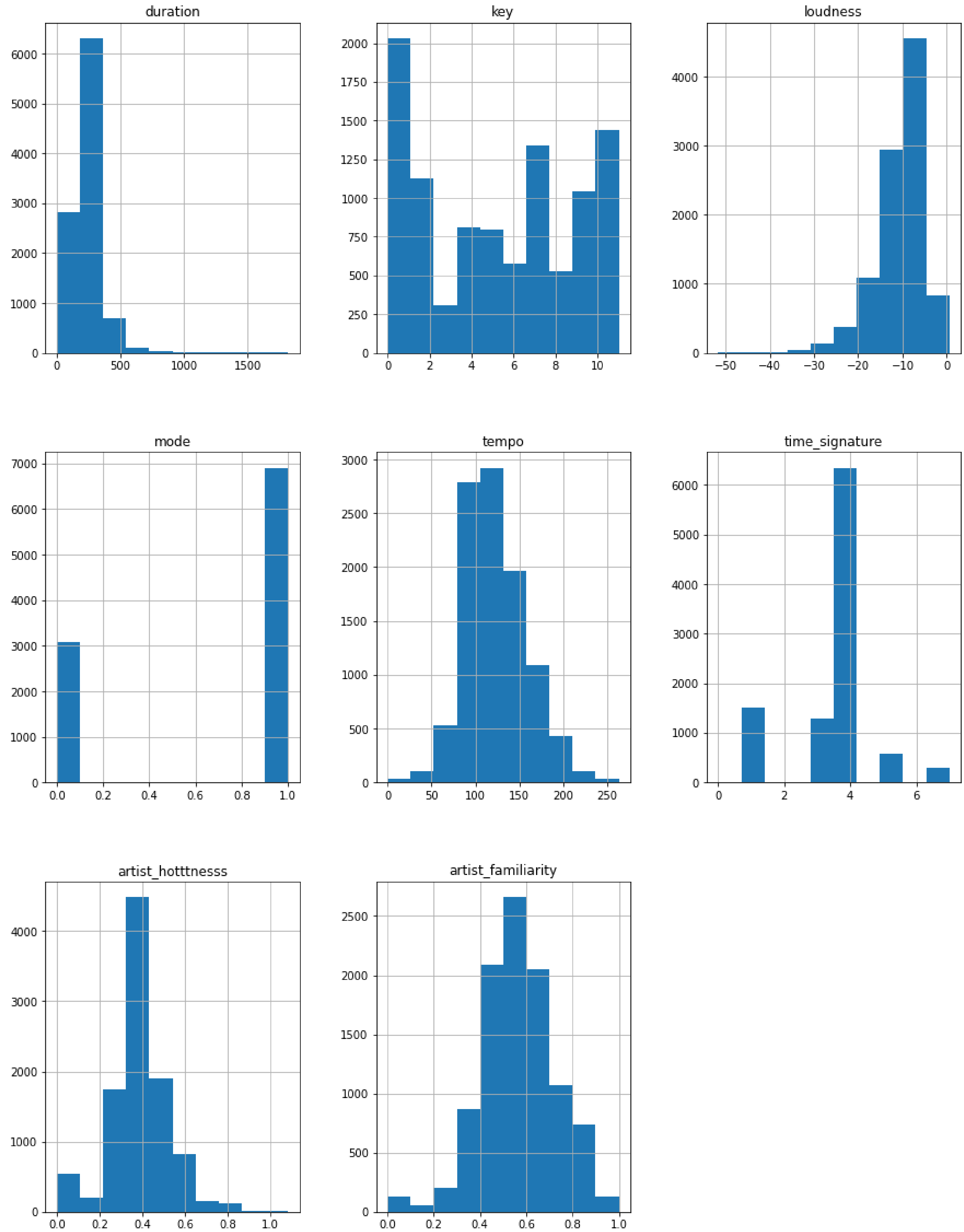
In [114…
```
data_MSD.persist()
```

Out[114]:
```
DataFrame[song_id: string, title: string, artist_id: string, duration: float, key: int, loudness: float, mode: int, tem
po: float, time_signature: int, artist_hotttnesss: float, artist_familiarity: float]
```

In [25]:
```
fig = plt.figure(figsize = (15,20))
ax = fig.gca()
data_MSD.toPandas().hist(ax = ax)
```

```
C:\Users\shbpa\AppData\Local\Temp/ipykernel_2712/2174606417.py:3: UserWarning: To output multiple subplots, the figure
containing the passed axes is being cleared
  data_MSD.toPandas().hist(ax = ax)
```

Out[25]:
```
array([[<AxesSubplot:title={'center':'duration'}>,
        <AxesSubplot:title={'center':'key'}>,
        <AxesSubplot:title={'center':'loudness'}>],
       [<AxesSubplot:title={'center':'mode'}>,
        <AxesSubplot:title={'center':'tempo'}>,
        <AxesSubplot:title={'center':'time_signature'}>],
       [<AxesSubplot:title={'center':'artist_hotttnesss'}>,
        <AxesSubplot:title={'center':'artist_familiarity'}>,
        <AxesSubplot:>]], dtype=object)
```



Now we can decide whether to apply normalization, standardization or both to the data.

standardization: Dealing with outliers by using IQR method

normalization: Scale data between 0-1

- **duration: Obviously we need both here because data is more concentrated between 0 and 500 and we have some outliers above 500 which can be addressed by standardization. To scale the data between 0-1, normalization can be applied.**
- **key: it is evenly distributed from 0 to 11 and we only apply normalization here to scale it between 0-1**
- **loudness: Same as the duration feature, we have some outliers from -50 to -30 which can be solved by standardization. Normalization will be applied too.**
- **mode: Since this value is either 0 or 1, neither normalization is needed nor standardization**
- **tempo: Since data is symmetrically distributed here we can overlook standardization. However, we apply it on the data to get rid of outliers on its head and tail.**
- **time_signature: Just normalization**
- **artist_hottness: Because data distribution is not sparse, no standardization,but normalization to make sure data is between 0-1**
- **artist_familiarity: the same as artist_hottness**

1.3.3 Standardization & Normalization :

- Standardardizer method:

In [115…
```
# standardize a column with IQR method
def standardize(df, column : str, lower, upper):

    split_udf = udf(lambda x: float(list(x)[0].item()), DoubleType())

    # create a vector assembler
    assembler = VectorAssembler(inputCols=[column], outputCol='temp')

    # assembel the vector to dataframe
    df = assembler.transform(df) # add temp column

    scaler = RobustScaler(inputCol = 'temp',outputCol='stndr',withScaling= True, withCentering=False,lower=lower, upper

    # Compute summary statistics by fitting the RobustScaler
    scalerModel = scaler.fit(df)

    # Transform each column to have unit quantile range.
    df = scalerModel.transform(df)

    # drop the created columns and substitute with the old column
    df = df.drop(column,'temp')
    df = df.withColumn('stndr',split_udf(col('stndr')))
    df = df.withColumnRenamed('stndr', column)
    return df , scalerModel
```

- Normalizer method:

In [116…
```
def normalizer(df , column):
    # UDF for converting column type from vector to double type
    split_udf = udf(lambda x: float(list(x)[0].item()), DoubleType())

    # VectorAssembler Transformation - Converting column to vector type
    assembler = VectorAssembler(inputCols=[column],outputCol='temp')
    df = assembler.transform(df)

    # MaxMinScaler to scale between 0-1
    scaler = MinMaxScaler(inputCol='temp', outputCol='normalized')

    scalerModel = scaler.fit(df)
    df = scalerModel.transform(df)

    # drop the created columns and substitute with the old column
    df = df.drop(column,'temp')
    df = df.withColumn('normalized',split_udf(col('normalized')))
    df = df.withColumnRenamed('normalized', column)

    return df , scalerModel
```

Standardize following features:

'duration', 'key', 'loudness', 'mode', 'tempo', 'time_signature', 'artist_hotttnesss', 'artist_familiarity

In [117…
```
scalers_stdr = {'duration': None,
          'loudness': None,
          'tempo': None}
```

In [119…
```
data_MSD_scaled = data_MSD
data_MSD_scaled , scalers_stdr['duration'] = standardize(data_MSD_scaled, 'duration', 0.05, 0.75)
print('Standardizing feature: '+ 'duration' )

data_MSD_scaled , scalers_stdr['loudness'] = standardize(data_MSD_scaled, 'loudness', 0.2, 0.90)
print('Standardizing feature: '+ 'loudness' )

data_MSD_scaled , scalers_stdr['tempo'] = standardize(data_MSD_scaled, 'tempo', 0.20, 0.85)
print('Standardizing feature: '+ 'tempo' )
```

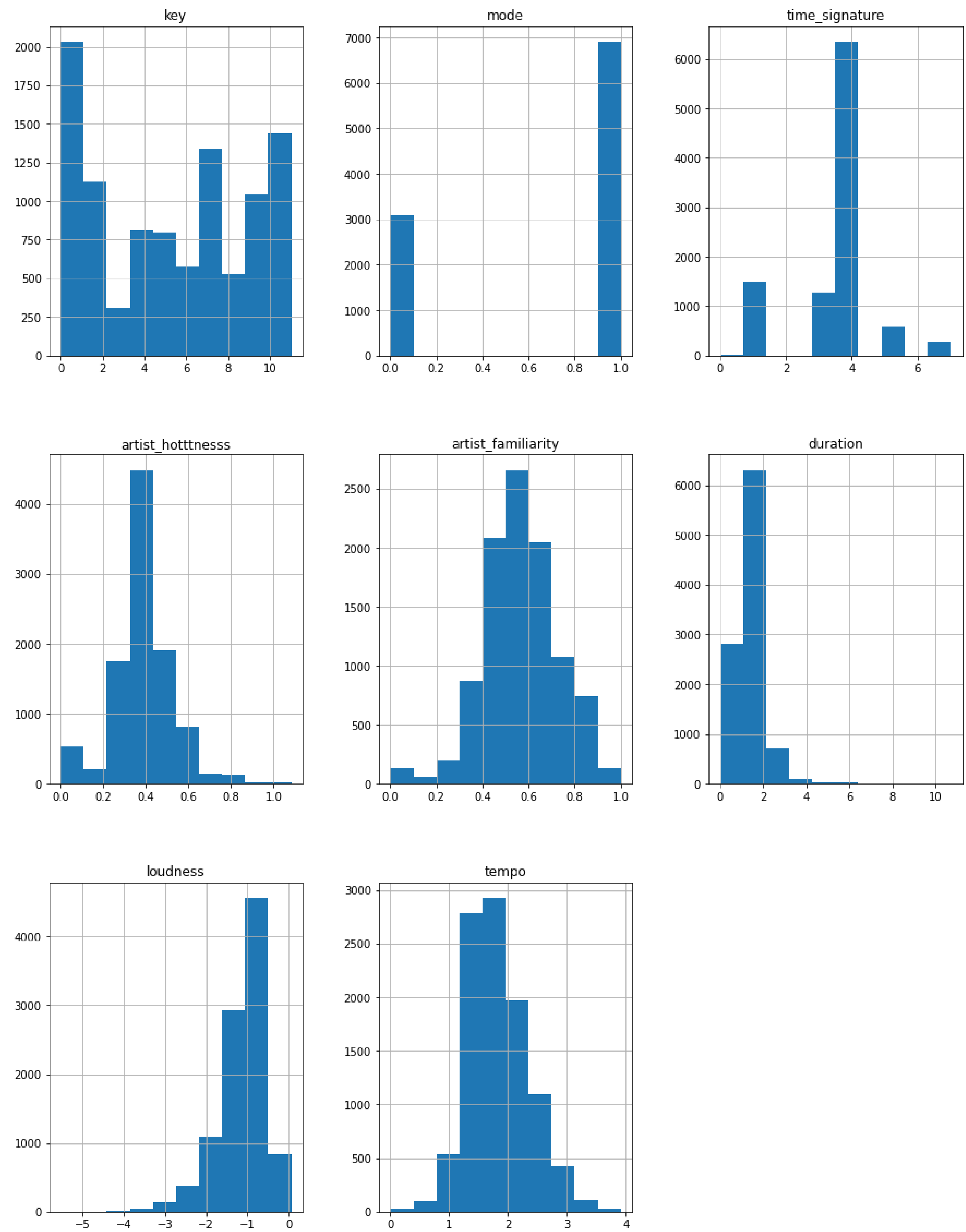
```
Standardizing feature: duration
Standardizing feature: loudness
Standardizing feature: tempo
```

Changes after Standardization:

In [30]:
```
fig = plt.figure(figsize = (15,20))
ax = fig.gca()
data_MSD_scaled.toPandas().hist(ax = ax)
```

```
C:\Users\shbpa\AppData\Local\Temp/ipykernel_2712/3078121536.py:3: UserWarning: To output multiple subplots, the figure
containing the passed axes is being cleared
  data_MSD_scaled.toPandas().hist(ax = ax)
```

Out[30]:
```
array([[<AxesSubplot:title={'center':'key'}>,
        <AxesSubplot:title={'center':'mode'}>,
        <AxesSubplot:title={'center':'time_signature'}>],
       [<AxesSubplot:title={'center':'artist_hotttnesss'}>,
        <AxesSubplot:title={'center':'artist_familiarity'}>,
        <AxesSubplot:title={'center':'duration'}>],
       [<AxesSubplot:title={'center':'loudness'}>,
        <AxesSubplot:title={'center':'tempo'}>, <AxesSubplot:>]],
      dtype=object)
```



In [31]:
```
data_MSD_scaled.toPandas().describe(percentiles=[0.05,.1, .2, .5, .7, .8, .9,.95]).T
```

Out[31]:

	count	mean	std	min	5%	10%	20%	50%	70%	80%	90%
key	9996.0	5.276611	3.554199	0.000000	0.000000	0.000000	1.000000	5.000000	8.000000	9.000000	10.000000
mode	9996.0	0.690976	0.462114	0.000000	0.000000	0.000000	0.000000	1.000000	1.000000	1.000000	1.000000
time_signature	9996.0	3.564626	1.266462	0.000000	1.000000	1.000000	3.000000	4.000000	4.000000	4.000000	4.000000
artist_hotttnesss	9996.0	0.385707	0.143469	0.000000	0.049034	0.257063	0.311438	0.380756	0.434411	0.476761	0.547755
artist_familiarity	9996.0	0.565457	0.160161	0.000000	0.321843	0.379428	0.441050	0.563666	0.637508	0.697113	0.784970
duration	9996.0	1.400214	0.670158	0.006131	0.620109	0.794124	0.971321	1.309459	1.543472	1.711698	2.056737
loudness	9996.0	-1.123764	0.578521	-5.533377	-2.242768	-1.866977	-1.527162	-1.005250	-0.756134	-0.645023	-0.524697
tempo	9996.0	1.830333	0.523933	0.000000	1.103977	1.263097	1.383190	1.789388	2.055493	2.252695	2.529902

In [32]:
```
data_MSD.toPandas().describe(percentiles=[0.05,.1, .2, .5, .7, .8, .9,.95]).T
```

Out[32]:

	count	mean	std	min	5%	10%	20%	50%	70%	80%
duration	9996.0	238.518509	114.157639	1.044440	105.632202	135.274643	165.459137	223.059143	262.921997	291.578308
key	9996.0	5.276611	3.554199	0.000000	0.000000	0.000000	1.000000	5.000000	8.000000	9.000000
loudness	9996.0	-10.488078	5.399334	-51.643002	-20.931750	-17.424500	-14.253000	-9.382000	-7.057000	-6.020000
mode	9996.0	0.690976	0.462114	0.000000	0.000000	0.000000	0.000000	1.000000	1.000000	1.000000
tempo	9996.0	122.910576	35.183144	0.000000	74.134251	84.819500	92.884003	120.161003	138.030502	151.272995
time_signature	9996.0	3.564626	1.266462	0.000000	1.000000	1.000000	3.000000	4.000000	4.000000	4.000000
artist_hotttnesss	9996.0	0.385707	0.143469	0.000000	0.049034	0.257063	0.311438	0.380756	0.434411	0.476761
artist_familiarity	9996.0	0.565457	0.160161	0.000000	0.321843	0.379428	0.441050	0.563666	0.637508	0.697113

Now its time for Normalization:

In [120…

```
scalers_norm = {'duration': None,
          'key': None,
          'loudness': None,
          'mode': None,
          'tempo': None,
          'time_signature': None,
          'artist_hotttnesss': None,
          'artist_familiarity': None}
```

In [121…

```
for i in scalers_norm.keys():
    data_MSD_scaled , scalers_norm[i] = normalizer(data_MSD_scaled, i)
    print('Normalizing feature: '+ i )
```

```
Normalizing feature: duration
Normalizing feature: key
Normalizing feature: loudness
Normalizing feature: mode
Normalizing feature: tempo
Normalizing feature: time_signature
Normalizing feature: artist_hotttnesss
Normalizing feature: artist_familiarity
```

1.3.4 Narmalized and Standardized data:

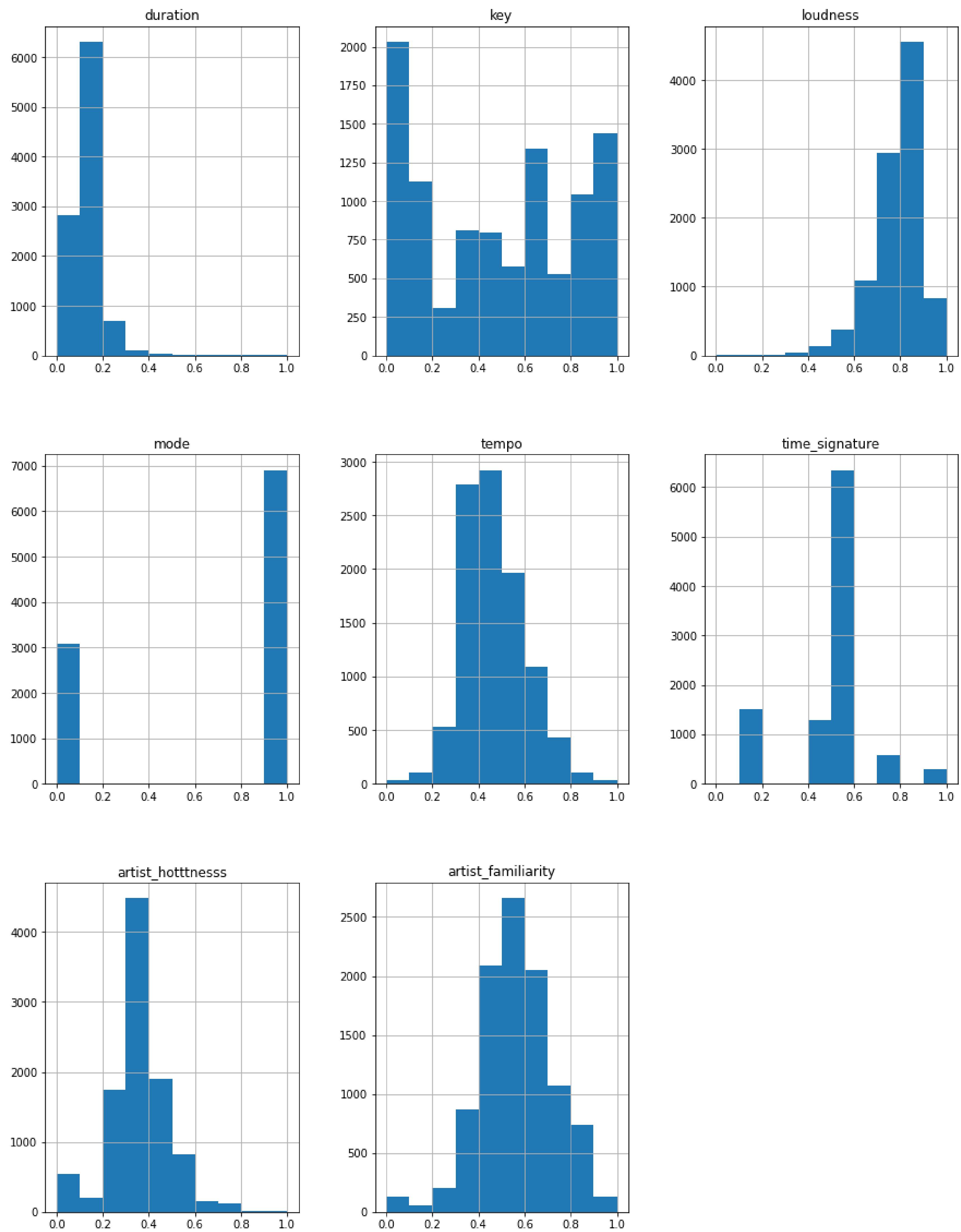
In [35]:

```
fig = plt.figure(figsize = (15,20))
ax = fig.gca()
data_MSD_scaled.toPandas().hist(ax = ax)
```

```
C:\Users\shbpa\AppData\Local\Temp/ipykernel_2712/3078121536.py:3: UserWarning: To output multiple subplots, the figure
containing the passed axes is being cleared
  data_MSD_scaled.toPandas().hist(ax = ax)
```

Out[35]:

```
array([[<AxesSubplot:title={'center':'duration'}>,
        <AxesSubplot:title={'center':'key'}>,
        <AxesSubplot:title={'center':'loudness'}>],
       [<AxesSubplot:title={'center':'mode'}>,
        <AxesSubplot:title={'center':'tempo'}>,
        <AxesSubplot:title={'center':'time_signature'}>],
       [<AxesSubplot:title={'center':'artist_hotttnesss'}>,
        <AxesSubplot:title={'center':'artist_familiarity'}>,
        <AxesSubplot:>]], dtype=object)
```



In [36]:
```
data_MSD_scaled.toPandas().describe(percentiles=[0.05,.1, .2, .5, .7, .8, .9,.95]).T
```

Out[36]:

	count	mean	std	min	5%	10%	20%	50%	70%	80%	90%	95%	max
duration	9996.0	0.130572	0.062768	0.0	0.057506	0.073805	0.090401	0.122072	0.143990	0.159746	0.192063	0.231292	1.0
key	9996.0	0.479692	0.323109	0.0	0.000000	0.000000	0.090909	0.454545	0.727273	0.818182	0.909091	1.000000	1.0
loudness	9996.0	0.788272	0.103418	0.0	0.588237	0.655414	0.716160	0.809458	0.853991	0.873853	0.895363	0.912041	1.0
mode	9996.0	0.690976	0.462114	0.0	0.000000	0.000000	0.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.0
tempo	9996.0	0.467646	0.133864	0.0	0.282064	0.322719	0.353402	0.457185	0.525174	0.575559	0.646385	0.709576	1.0
time_signature	9996.0	0.509232	0.180923	0.0	0.142857	0.142857	0.428571	0.571429	0.571429	0.571429	0.571429	0.714286	1.0
artist_hotttnesss	9996.0	0.356310	0.132534	0.0	0.045297	0.237471	0.287702	0.351737	0.401303	0.440425	0.506008	0.555367	1.0
artist_familiarity	9996.0	0.565456	0.160161	0.0	0.321843	0.379428	0.441050	0.563666	0.637508	0.697113	0.784970	0.839275	1.0

In [122…:
```
data_MSD_scaled.persist()
```

Out[122]:
```
DataFrame[song_id: string, title: string, artist_id: string, duration: double, key: double, loudness: double, mode: dou
ble, tempo: double, time_signature: double, artist_hotttnesss: double, artist_familiarity: double]
```

1.3.5 Store the scaled and pruned MSD dataset to a csv file

In [124…
```
#data_MSD_scaled.write.mode("overwrite").csv('data_MSD_scaled.csv') error
data_MSD_scaled.toPandas().to_csv("data_MSD_scaled1.csv", header=True)
```

In [ ]:
```
df.write.mode(SaveMode.Overwrite).csv("file.csv")
```

In [ ]:
```
df.write.mode("overwrite").csv("file.csv")
```

2. Second dataset (User-Song implicit rating)

2.1 Load User-Items(Music) dataset:

- **This dataset consists of three columns: userId, songId, and number of play for each song(play_count).**
- **Consists of 1 Million user**
- **Since the Million Song Dataset is a subset of all dataset(10000 songs) we should match songs of this dataset with the songs of above dataset**

In [125…
```
# define schema, nullable is set true
schema2 = StructType([
    StructField('userId', StringType(), True),
    StructField('songId', StringType(), True),
    StructField('play_count', IntegerType(), True)])
```

In [126…
```
# Read the csv file, each coloumn is divided by a tab (from csv into a spark dataframe)
data_imp_rating_full = spark.read.option("delimiter", "\t").schema(schema2).csv(data_imp_rating_path)
```

In [127…
```
data_imp_rating_full.columns
```

Out[127]:
```
['userId', 'songId', 'play_count']
```

In [57]:
```
data_imp_rating_full.describe().show()
```

```
+-------+--------------------+------------------+-----------------+
|summary|              userId|            songId|       play_count|
+-------+--------------------+------------------+-----------------+
|  count|            48373586|          48373586|         48373586|
|   mean|                null|              null|2.866858847305635|
| stddev|                null|              null|6.437724686877057|
|    min|00000b72200188206...|SOAAADD12AB018A9DD|                1|
|    max|fffff9534445f481b...|SOZZZWN12AF72A1E29|             9667|
+-------+--------------------+------------------+-----------------+
```

In [58]:
```
data_imp_rating_full.show(2)
```

```
+--------------------+------------------+----------+
|              userId|            songId|play_count|
+--------------------+------------------+----------+
|b80344d063b5ccb32...|SOAKIMP12A8C130995|         1|
|b80344d063b5ccb32...|SOAPDEY12A81C210A9|         1|
+--------------------+------------------+----------+
only showing top 2 rows
```

2.2 Preprocessing:

- **Since we are using a subset of songs dataset(MSD subset), it is possible that some songs in User-Songs dataset not to be in the MSD dataset. To make sure we are going to keep only songs in the MSD dataset with leftsemi join method, as follows:**

In [128…
```
# whole number of user-song data
data_imp_rating_full.count()
```

Out[128]:
```
48373586
```

In [129…
```
# keep records with songs in the MSD data
data_imp_rating = data_imp_rating_full.join(data_MSD, data_imp_rating_full.songId ==  data_MSD.song_id, "leftsemi")
```

In [130…
```
data_imp_rating.persist()
```

Out[130]:
```
DataFrame[userId: string, songId: string, play_count: int]
```

In [131…
```
# count the number of remaining records
data_imp_rating.count()
```

Out[131]:
```
772661
```

In [65]:
```
data_imp_rating.describe().show()
```

```
+-------+--------------------+------------------+-----------------+
|summary|              userId|            songId|       play_count|
+-------+--------------------+------------------+-----------------+
|  count|              772661|            772661|           772661|
|   mean|                null|              null|2.684340221649598|
| stddev|                null|              null|5.454645798218129|
|    min|00001638d61892368...|SOAAAQN12AB01856D3|                1|
|    max|fffff67d54a40927c...|SOZZVMW12AB0183B52|              771|
+-------+--------------------+------------------+-----------------+
```

2.3 Splitting Data:

Since we are going to recommend song to users based on implicit ratings, we only need to split this dataset. Here we are going to use two different methods of splitting data- Stratified sampling and Random split.

Before anything, it should be mentioned that we are going to use ALS algorithm for extrapolating missing values in user-item matrix in Latent_Factor approach. This algorithm is predefined in pySpark. To use it, we should pass userId and songId as integer, while it is string in our dataset. Now, before splitting the data, two columns "songIndex" and "userIndex" should be added to the dataset. These colomns are generated simply by giving index to a column of unique songs and a column of unique users.

2.3.1 Convert userIds and songIds from string to integer:

In [132…

```
# get the userId column, drop the duplicates values and put them in a window to have unique and consequtive indices
userIds= data_imp_rating.select('userId').dropDuplicates().withColumn("index",row_number().over(Window.orderBy(monotoni
```

In [133…

```
userIds.head(5)
```

Out[133]:

```
[Row(userId='2c218a60b3d777e9e12d56c2e065a9644b5e5f41', index=1),
 Row(userId='cc9fc2eccf0d6fe78d1fb2b0c3ff924f54482169', index=2),
 Row(userId='ae0565253d822cdc47c645a1b29cb6a5e2e2ab16', index=3),
 Row(userId='74d0c24a0bb5bde014ffbf57fc5c51b9b5b799a0', index=4),
 Row(userId='58f2d6ed090ba4626486e6ad205eb09365adfbf3', index=5)]
```

In [134…

```
userIds.count()
```

Out[134]:

```
418252
```

In [135…

```
# same process as above
songIds = data_imp_rating.select('songId').dropDuplicates().withColumn("index",row_number().over(Window.orderBy(monoton
```

In [136…

```
songIds.count()
```

Out[136]:

```
3675
```

In [137…

```
songIds.head(5)
```

Out[137]:

```
[Row(songId='SOTSIIH12A8C13A516', index=1),
 Row(songId='SOVPUVS12A6D4F7988', index=2),
 Row(songId='SOCKUUJ12A6D4FA41C', index=3),
 Row(songId='SOJUGKQ12A8C13A83A', index=4),
 Row(songId='SOSIVPO12AB017D5E9', index=5)]
```

Rename the columns and add to the data_imp_rating dataset

In [138…

```
userIds = userIds.withColumnRenamed('index', 'userIndex')
```

In [139…

```
songIds = songIds.withColumnRenamed('index', 'songIndex')
```

In [140…

```
data_imp_rating = data_imp_rating.join(userIds, ['userId'])
```

In [141…

```
data_imp_rating = data_imp_rating.join(songIds, ['songId'])
```

In [142…

```
# Finall data Model of implicit rating dataset
data_imp_rating.columns
```

Out[142]:

```
['songId', 'userId', 'play_count', 'userIndex', 'songIndex']
```

In [ ]:

2.3.2 Stratified sampling:

In order to split data based on userId, users with more than 5 songs are collected to make sure we have enough data in train and test.

In [143…

```
ss = data_imp_rating.groupby('userId').agg({'userId': 'count'}).filter(col("count(userId)")>4)
ss = data_imp_rating.join(ss, ['userId'])
```

In [144…

```
ss.columns
```

Out[144]:

```
['userId', 'songId', 'play_count', 'userIndex', 'songIndex', 'count(userId)']
```

In [145…

```
fractions = ss.select("userId").distinct().withColumn('fraction', F.lit(0.2)).rdd.collectAsMap()
```

In [146…
```
test_imp_data = ss.stat.sampleBy('userId', fractions, seed = 42).drop("count(userId)")
```

In [147…
```
cond = [test_imp_data.userId == ss.userId, test_imp_data.songId == ss.songId]
```

In [148…
```
train_imp_data = ss.join(test_imp_data, cond , "leftanti" )#drop("count(userId)")
# train_imp_data = train_imp_data.join(ss)
```

In [149…
```
train_imp_data = train_imp_data.drop('count(userId)')
```

In [150…
```
test_imp_data.columns
```

Out[150]:
```
['userId', 'songId', 'play_count', 'userIndex', 'songIndex']
```

In [151…
```
train_imp_data.columns
```

Out[151]:
```
['userId', 'songId', 'play_count', 'userIndex', 'songIndex']
```

In [152…
```
test_imp_data.head(5)
```

Out[152]:
```
[Row(userId='0359ab58a65430dd7f652138e86663709a887829', songId='SOEKSGJ12A67AE227E', play_count=2, userIndex=1414, song
Index=3276),
 Row(userId='0486147ef9c026213cf6f77d62577a9cd71f9bd3', songId='SOTXXBT12A6D4F6B25', play_count=1, userIndex=210, songI
ndex=2653),
 Row(userId='0486147ef9c026213cf6f77d62577a9cd71f9bd3', songId='SOCXWEG12A6D4FBEA3', play_count=1, userIndex=210, songI
ndex=3640),
 Row(userId='07e62756f710c6a69bdfd5a7cb7a14bfbeb773cf', songId='SODTTUB12AB0184F48', play_count=1, userIndex=904, songI
ndex=2314),
 Row(userId='0af944c051730d2c2ab2ddc39d3f8f4d41fc58a1', songId='SOIGZOE12AB017F37D', play_count=7, userIndex=1595, song
Index=2555)]
```

In [153…
```
test_imp_data.persist()
```

Out[153]:
```
DataFrame[userId: string, songId: string, play_count: int, userIndex: int, songIndex: int]
```

In [154…
```
train_imp_data.persist()
```

Out[154]:
```
DataFrame[userId: string, songId: string, play_count: int, userIndex: int, songIndex: int]
```

2.3.3 Random sampling:

Splitting the implicit rating dataset randomly. But it causes an error in content-based method because we need for each user at least one song to be able to create the user profile. Therefore spliting randomly rase an error in creating user profile.

In [92]:
```
(train_random_split, test_random_split) = data_imp_rating.randomSplit([0.8, 0.2], seed = 42)
```

In [155…
```
train_random_split.persist()
```

Out[155]:
```
DataFrame[songId: string, userId: string, play_count: int, userIndex: int, songIndex: int]
```

In [156…
```
test_random_split.persist()
```

Out[156]:
```
DataFrame[songId: string, userId: string, play_count: int, userIndex: int, songIndex: int]
```

2.3.4 Store train and test data as csv

In [160…
```
test_random_split.toPandas().to_csv("test_random.csv", header=True)
```

In [162…
```
train_random_split.toPandas().to_csv("train_random.csv", header=True)
```

In [97]:
```
train_imp_data.toPandas().to_csv("train_stratified.csv", header=True)
```

In [98]:
```
test_imp_data.toPandas().to_csv("test_stratified.csv", header=True)
```

In [ ]:
```

```

3. Recommendation Systems:

3.1Content-Based Recommendation System:

For content-based model we can only work with train and test data generated by stratified sampling method, because we have to make sure that is possible to create users profile. If there is no song record for a user, there is no other way to create profile for the user.

3.1.2 Creating user_profile data model:

In [163…
```
train_CB = train_imp_data.select('userId', 'songId', 'play_count')
```

In [164…
```
train_CB.columns
```

Out[164]:
```
['userId', 'songId', 'play_count']
```

In [165…

```
data_MSD_features = ['duration',
                     'key',
                     'loudness',
                     'mode',
                     'tempo',
                     'time_signature',
                     'artist_hotttnesss',
                     'artist_familiarity']
```

In [166…

```
# add features of each song based on song_id from data_MSD dataset to the implicit rating test dataset(user-song-plays)
user_profile = train_CB
user_profile = user_profile.join(data_MSD_scaled, data_MSD_scaled.song_id == train_CB.songId ).drop()
```

In [167…

```
user_profile.columns
```

Out[167]:

```
['userId',
 'songId',
 'play_count',
 'song_id',
 'title',
 'artist_id',
 'duration',
 'key',
 'loudness',
 'mode',
 'tempo',
 'time_signature',
 'artist_hotttnesss',
 'artist_familiarity']
```

In [168…

```
# calculate the product of each feature with play_count column
for x in data_MSD_features:
    user_profile = user_profile.withColumn(x, col(x)*col('play_count'))

user_profile = user_profile.groupby('userId').sum('duration',
                                                  'key',
                                                  'loudness',
                                                  'mode',
                                                  'tempo',
                                                  'time_signature',
                                                  'artist_hotttnesss',
                                                  'artist_familiarity')
```

In [169…

```
user_profile.columns
```

Out[169]:

```
['userId',
 'sum(duration)',
 'sum(key)',
 'sum(loudness)',
 'sum(mode)',
 'sum(tempo)',
 'sum(time_signature)',
 'sum(artist_hotttnesss)',
 'sum(artist_familiarity)']
```

In [170…

```
temp_name = ['sum(duration)',
             'sum(key)',
             'sum(loudness)',
             'sum(mode)',
             'sum(tempo)',
             'sum(time_signature)',
             'sum(artist_hotttnesss)',
             'sum(artist_familiarity)']
```

In [171…

```
#changing the columns name same as songs dataset columns
for x,y in zip(data_MSD_features,temp_name):
    user_profile = user_profile.withColumnRenamed(y,x)
```

In [172…

```
user_profile.columns
```

Out[172]:

```
['userId',
 'duration',
 'key',
 'loudness',
 'mode',
 'tempo',
 'time_signature',
 'artist_hotttnesss',
 'artist_familiarity']
```

In [173…

```
# add a column to store the sum of feature for each user
user_profile= user_profile.withColumn('sum', sum(user_profile[col] for col in data_MSD_features ))

# calculate the user interest probability to each feature (feature_value / sum of the value of all features)
for x in data_MSD_features:
    user_profile= user_profile.withColumn(x, col(x)/col('sum'))
```

In [174…

```
user_profile.columns
```

Out[174]:
```
['userId',
 'duration',
 'key',
 'loudness',
 'mode',
 'tempo',
 'time_signature',
 'artist_hotttnesss',
 'artist_familiarity',
 'sum']
```

In [175…
```
user_profile = user_profile.drop('sum')
```

In [176…
```
user_profile.columns
```

Out[176]:
```
['userId',
 'duration',
 'key',
 'loudness',
 'mode',
 'tempo',
 'time_signature',
 'artist_hotttnesss',
 'artist_familiarity']
```

In [177…
```
user_profile.persist()
```

Out[177]:
```
DataFrame[userId: string, duration: double, key: double, loudness: double, mode: double, tempo: double, time_signature:
double, artist_hotttnesss: double, artist_familiarity: double]
```

Store user profile data model as CSV:

In [178…
```
user_profile.toPandas().to_csv("user_profile.csv", header=True)
```

In [ ]:
```

```

The user profile is created based on songs he/she played!!

Now everything is ready to evaluate most similar songs for each user by Cosine distances(Cosine similarity)

3.1.3 Cosine Similarity:

In [179…
```
# put all features into a dense vector for both songs and user profiles

assembler1 = VectorAssembler(inputCols=['duration', 'key', 'loudness',
                                        'mode', 'tempo', 'time_signature',
                                        'artist_hotttnesss', 'artist_familiarity'],outputCol='Ufeatures')

assembler2 = VectorAssembler(inputCols=['duration', 'key', 'loudness',
                                        'mode', 'tempo', 'time_signature',
                                        'artist_hotttnesss', 'artist_familiarity'],outputCol='Sfeatures')
```

In [180…
```
df1 = assembler1.transform(user_profile).select('userId', 'Ufeatures')
```

In [181…
```
df2 = assembler2.transform(data_MSD_scaled).select('song_id', 'Sfeatures')
```

In [182…
```
# Join df1 and df2 dataframe in order to compare each user profile with each song
# So for each userId we have all songs
df = df1.crossJoin(df2)
```

In [183…
```
df.columns
```

Out[183]:
```
['userId', 'Ufeatures', 'song_id', 'Sfeatures']
```

In [185…
```
# create a new train df and change the name of userId column to 'ui' to be able to
train = train_imp_data.select('userId','songId').withColumnRenamed('userId','ui')
```

In [186…
```
#defining a multiple condition for join
cond=[df.userId == train.ui, df.song_id == train.songId]
```

In [188…
```
result = df.join(train,cond ,'leftanti')
```

In [189…
```
# Get cosine similarity
result = result.rdd.map(lambda x: (x['userId'], x['song_id'],
                              float(cosine_similarity([x['Ufeatures']],
                                                      [x['Sfeatures']])[0,0]))).toDF(schema=['userId', 'song_id', 'cos
```

In [190…
```
#sorting the result by cosine_similarity per each user
window = Window.partitionBy(result['userId']).orderBy(result['cosine_similarity'].desc())
```

In [191…
```
#selecting n most relevent songs for each user
n = 10
predict_top_nSong = result.select('*', rank().over(window).alias('rank')).filter(col('rank') <= n)
```

In [192…
```
predict_top_nSong.persist()
```

Out[192]:
```
DataFrame[userId: string, song_id: string, cosine_similarity: double, rank: int]
```

In [193…
```
predict_top_nSong.show()
```

```
+--------------------+------------------+------------------+----+
|              userId|           song_id| cosine_similarity|rank|
+--------------------+------------------+------------------+----+
|0359ab58a65430dd7...|SOIEAJT12A8AE458EC|0.9782267126569764|   1|
|0359ab58a65430dd7...|SODHTCY12A58A7F125|0.9775481960502393|   2|
|0359ab58a65430dd7...|SOAOPVN12AAF3B1856|0.9757378255362639|   3|
|0359ab58a65430dd7...|SOHKNRJ12A6701D1F8|0.9754747604252393|   4|
|0359ab58a65430dd7...|SOPPCXM12A6D4F66BC|0.9753965499397119|   5|
|0359ab58a65430dd7...|SOJBYGW12A8C13A497|0.9753614136521936|   6|
|0359ab58a65430dd7...|SOYEDIE12A8C142C36|0.9750551760414174|   7|
|0359ab58a65430dd7...|SOULIKU12A58A78CE2|  0.97487902678679|   8|
|0359ab58a65430dd7...|SOKZCJC12AF72A8C79|0.9747497714968003|   9|
|0359ab58a65430dd7...|SOHKXAC12A58A7F6E5|0.9738602569837279|  10|
|0486147ef9c026213...|SODMJKG12A670202EB|   0.9721507471084|   1|
|0486147ef9c026213...|SOJMQQX12AB0185046|0.9715028110137298|   2|
|0486147ef9c026213...|SOIRCEO12A8C134D85|0.9713101301056071|   3|
|0486147ef9c026213...|SOLLXZJ12A6D4F96B0|0.9710945807537237|   4|
|0486147ef9c026213...|SOWJCAE12AC46887E7|0.9710365190370711|   5|
|0486147ef9c026213...|SODPNRD12AB017FB2F|0.9708743522168042|   6|
|0486147ef9c026213...|SOIARWN12AF72A5A63|0.9707851495299543|   7|
|0486147ef9c026213...|SOOHUOU12A8C1399A5|0.9706505479420702|   8|
|0486147ef9c026213...|SOQMDJS12A8C138341|0.9705823830685647|   9|
|0486147ef9c026213...|SOKHRQI12A8C13F53E| 0.970529742935179|  10|
+--------------------+------------------+------------------+----+
only showing top 20 rows
```

In [212…
```
predict_top_nSong.describe().show()
```

```
+-------+--------------------+------------------+--------------------+-----------------+
|summary|              userId|           song_id|   cosine_similarity|             rank|
+-------+--------------------+------------------+--------------------+-----------------+
|  count|              228361|            228361|              228361|           228361|
|   mean|                null|              null|  0.9879646869997364|5.499967157264156|
| stddev|                null|              null|0.010100157605710699|2.872313529858836|
|    min|0000f88f8d76a238c...|SOAAAQN12AB01856D3|   0.954820505706391|                1|
|    max|ffff6f29052de81f5...|SOZZWWW12A58A8146A|  0.9999345849584005|               10|
+-------+--------------------+------------------+--------------------+-----------------+
```

Store predict_top_nSong data model as CSV

In [217…
```
predict_top_nSong.toPandas().to_csv("CB_10Songs.csv", header=True)
```

In [213…
```
user_profile.count()
```

Out[213]:
```
22836
```

In [211…
```
test_imp_data.count()
```

Out[211]:
```
30607
```

In [209…
```
predict_top_nSong.count()
```

Out[209]:
```
228361
```

Calculating F1:

For each user we have n songs recommended. For F1 score we have:

- **Precision = items in common with test and @ n songs recommended(TP) / number of songs in recommendder**
- **Recall = items in common with test and @ n songs recommended(TP) / number of songs in test**
- **F1 = 2*(Precision*Recall)/(Precision+Recall)**

In [219…
```
conditions = [ test_imp_data.userId == predict_top_nSong.userId , test_imp_data.songId == predict_top_nSong.song_id ]
TP = test_imp_data
TP = TP.join(predict_top_nSong , conditions, "leftsemi").count()
```

In [220…
```
nom_of_song_test = test_imp_data.count()
nom_of_song_predicted = predict_top_nSong.count()
precision = (TP / nom_of_song_predicted)
recall = (TP / nom_of_song_test)
F1 = 2 * (precision * recall)/(precision + recall)
```

In [222…
```
print('Precision: ', precision)
print('recall: ', recall)
print('F1: ', F1)
```

```
Precision:  0.001703443232425852
recall:  0.012709510896200215
F1:  0.003004232183126873
```

4. Collaborative Filtering:

4.1 Latent Factor model:

4.1.1 Train and test split with stratified sampling method:

In [223…

```
# ALS Algorithm to fill null values
# Build the recommendation model using ALS on the training data
# Rating is implicit in our model, it is biased too, so the implicitPrefs assigned true to add biased terms to the algo
# In this step we are not dealing with cold-start issues so the coldStartStrategy is droped

als = ALS(maxIter=10,regParam=0.01, userCol="userIndex", itemCol="songIndex", ratingCol="play_count",implicitPrefs = Tr
```

In [224…

```
model_CF = als.fit(train_imp_data)
```

In [225…

```
predictions = model_CF.transform(test_imp_data)
```

In [226…

```
# Evaluate the model by computing RMSE
evaluator = RegressionEvaluator(metricName="rmse",
                                labelCol="play_count",
                                predictionCol="prediction")
rmse = evaluator.evaluate(predictions)
print("RMSE error = " + str(rmse))
```

```
RMSE error = 4.279752515546069
```

In [228…

```
# Generate top 10 songs recommendations
userRecs = model_CF.recommendForAllUsers(10)
```

In [229…

```
userRecs.show()
```

```
+---------+--------------------+
|userIndex|     recommendations|
+---------+--------------------+
|     7240|[[513, 0.876827],...|
|     7880|[[2070, 0.2594938...|
|    57370|[[1594, 1.0294975...|
|    65220|[[2130, 0.7208005...|
|    94950|[[2070, 0.5884137...|
|   111300|[[863, 0.5618871]...|
|   189310|[[1594, 0.496572]...|
|   239370|[[2070, 0.0675105...|
|   243440|[[3276, 0.8137020...|
|   279120|[[2070, 0.3801165...|
|   280340|[[649, 0.8879849]...|
|   308930|[[513, 0.68086624...|
|   310950|[[2070, 0.1632575...|
|   336620|[[2463, 0.5566273...|
|   343180|[[2068, 0.5510056...|
|   369410|[[3293, 0.4774469...|
|   376270|[[2259, 0.0376724...|
|    16861|[[1594, 0.5194273...|
|    38311|[[513, 0.24565198...|
|    49331|[[191, 0.15010667...|
+---------+--------------------+
only showing top 20 rows
```

In [230…

```
userRecs.first()
```

Out[230]:

```
Row(userIndex=7240, recommendations=[Row(songIndex=513, rating=0.8768270015716553), Row(songIndex=272, rating=0.5853037
238121033), Row(songIndex=3056, rating=0.49465104937553406), Row(songIndex=1633, rating=0.4029790163040161), Row(songIn
dex=2092, rating=0.3559951186180115), Row(songIndex=2463, rating=0.2684287428855896), Row(songIndex=3169, rating=0.2235
6276214122772), Row(songIndex=863, rating=0.20198214054107666), Row(songIndex=2259, rating=0.2015409916639328), Row(son
gIndex=436, rating=0.18975648283958435)])
```

4.1.2 Train and test split with random sampling method:

In [26]:

```
als_random = ALS(maxIter=10,regParam=0.01,
          userCol="userIndex",
          itemCol="songIndex",
          ratingCol="play_count",
          implicitPrefs = True,coldStartStrategy="drop")
```

In [27]:

```
model_CF_randomSplit = als_random.fit(train_random_split)
```

In [28]:

```
predictions_random = model_CF_randomSplit.transform(test_random_split)
```

In [30]:

```
# Evaluate the model by computing RMSE
evaluator = RegressionEvaluator(metricName="rmse",
                                labelCol="play_count",
                                predictionCol="prediction")
rmse_random = evaluator.evaluate(predictions_random)
print("RMSE_random error = " + str(rmse_random))
```

```
RMSE_random error = 5.325317016701323
```

In [ ]:

In [ ]: