Find Next Hit

Your Playlist from million songs

ece472 team01

August 2, 2024

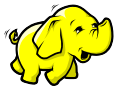
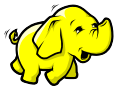


Table of contents

MileStone 0: Data Preparation

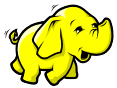
A glance into MillionSongs



Introduction

We, a four people team, are assigned with an ambitious task: build a music platform (by boss of FOCS)! When we began to analyze, FOCS boss raised a series of challenges:

- Data Storage Optimization.
  - Small files consuming more storage and memory resources.
- Data Processing Efficiency.
  - Recommendation system should respond to queries interactively.
- Recommendation Accuracy.
  - Recommendations should be highly accurate and relevant to provided song.
- Scalability.
  - Ensure the method can be applied to more and more users and cases.

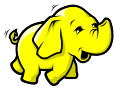


Mounting

At the very beginning, we need to mount Million Song Dataset to local. Here is a `mount.sh` script to run before processing:

```
sudo -S sshfs /home/hadoopuser/ece472 -o allow_other -o Port=2223
↪  ece472@focs.ji.sjtu.edu.cn: -o IdentityFile=~/.ssh/id_rsa
sudo -S mount /home/hadoopuser/ece472/millionsong.iso
↪  /home/hadoopuser/ece472/
```

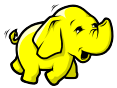
This script enables us to retrieve data from FOCS server, without downloading nearly 300 GiB data to local.



Decoding H5 Files

```
Track.h5
├─ analysis
│  └─ audio features
│     └─ lots data in array
├─ metadata
│  └─ artist characteristics
│     └─ lots tags in array
└─ musicbrainz
   └─ artist tags
```

- HDF5 files
  - hierarchical structure
  - various data types (nested arrays)
- However, Avro files
  - interoperability with Hadoop ecosystem
  - fast serialization and deserialization
- Hence, we need **python h5py** module to extract information from *.h5* files to a big *.avro* file.



Merging avro Files in Parallel

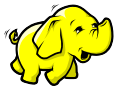
To merge Avro files, we wrote a program utilizing `Spark`:

```
# main function
result_rdd = sc.parallelize(alphabet, 26).map(collect_avro_with_letter)
result_rdd.collect()

merged_results, merged_schema = merge_avro_files()

with open("songs_advanced.avro", 'wb') as f:
    fastavro.writer(f, merged_schema, merged_results)
```

And to write Avro file, we designed an Avro schema that only contains necesary fields for prediction. Hence, the final file size is reduced to `219.1 MiB`.

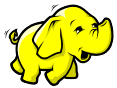


Run Spark-Submit

To run 12 tasks in parallel in local PC (rather than master mode):

```
spark-submit \
--master local[12] \
--conf spark.pyspark.driver.python=python3 \
--conf spark.pyspark.python=python3 \
--driver-cores 2 \
--driver-memory 8g \
--executor-cores 4 \
--num-executors 10 \
--executor-memory 8g \
pyspark.py
```

Thank for `Spark`, with 12 tasks processing in parallel, it took 4 hours. But for 2 tasks in default, it took 8+ hours.



Explanatory Data Analysis

We will check null values before processing:

```
df.isnull().sum().sort_values(ascending=False)
df_zero = (df == 0)
zero_counts = df_zero.sum().sort_values(ascending=False)
```

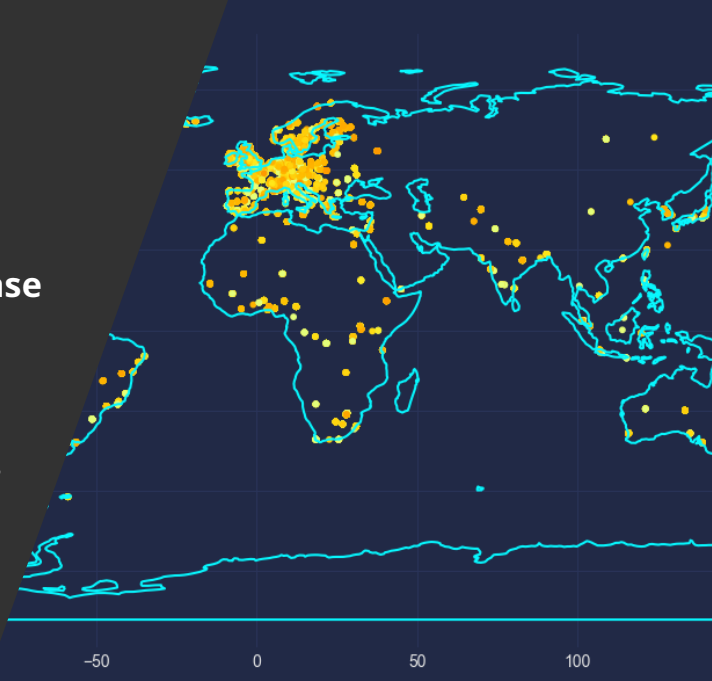
Surprisingly, the output is:

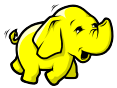
	artist_longitude	song_hotness	year	...
ignoring null or missing	357492	581965	515576	1000000

Table: Statistic of Raw Data

MileStone 1: Drill Database Query

Run SQL commands to verify the database

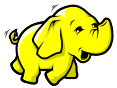




Motivation

In this part, we used drill to perform simple database queries, including:

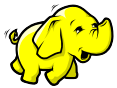
- Find the range of dates covered by the songs in the dataset, i.e. the age of the *oldest* and of the *youngest* songs
- Find the *hottest* song that is the *shortest* and has the *highest energy* with the *lowest tempo*.
- Find the name of the album with the *most tracks*.
- Find the name of the band(artists) who recorded the *longest song*.
- (Optimal) Find *Top 5 popular* songs by given artist, for similar experience of music platforms.



Preparation

Given the avro file, we first created a table in drill:

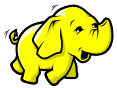
```
CREATE TABLE dfs.tmp.`songs` AS SELECT *
FROM dfs.`/home/hadoopuser/ve472/songs_advanced.avro`;
USE dfs.tmp;
```



Detailed Solutions

1. the oldest and youngest songs

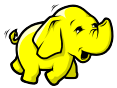
```
SELECT max(2024 - year) AS age
  FROM songs
 WHERE year > 0;
+-------+
|  age  |
+-------+
| 102.0 |
+-------+
1 row selected (0.128 seconds)
```



Detailed Results

```
SELECT min(2024 - year) AS age
  FROM songs
 WHERE year > 0;
+------+
| age  |
+------+
| 13.0 |
+------+
1 row selected (0.134 seconds)
```

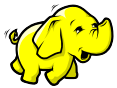
Therefore, the dataset covered songs from 1922 to 2011, namely the age of the songs vary from 13 years to 102 years.



Detailed Solutions

2. the hottest, shortest, highest energy, lowest tempo

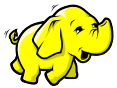
```
SELECT song_id,
       title
  FROM songs
 WHERE song_hotness <> 'NaN'
 ORDER BY song_hotness DESC,
          duration ASC,
          energy DESC,
          tempo ASC
 LIMIT 5;
```



Detailed Results

```
+--------------------+------------------------------------+
|      song_id       |               title                |
+--------------------+------------------------------------+
| SONASKH12A58A77831 | Jingle Bell Rock                   |
| SOAVJBU12AAF3B370C | Rockin' Around The Christmas Tree  |
| SOEWAKD12AB01860D5 | Holiday                            |
| SOAAXAK12A8C13C030 | Immigrant Song (Album Version)     |
| SOAXLDX12AC468DE36 | La Tablada                         |
+--------------------+------------------------------------+
5 rows selected (0.311 seconds)beg
```

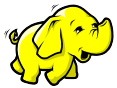
Therefore, `Jingle Bell Rock` is the hottest song that is the shortest and shows highest energy with lowest tempo.



Detailed Solutions

3. the album with the most songs

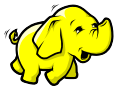
```
SELECT album_id,
       album_name,
       count(album_id) AS numSongs
  FROM songs
 GROUP BY album_id,
          album_name
 ORDER BY numSongs DESC
 LIMIT 1;
```



Detailed Results

```
+----------+-------------------------------------------------+----------+
| album_id |                   album_name                    | numSongs |
+----------+-------------------------------------------------+----------+
| 60509    | First Time In A Long Time: The Reprise Recordings | 85       |
+----------+-------------------------------------------------+----------+
1 row selected (0.651 seconds)
```

First Time In A Long Time: The Reprise Recordings is the album with most tracks.

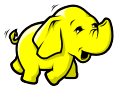


Detailed Solutions & Results

4. the band with longest song

```
SELECT artist_name, title, duration FROM songs
 ORDER BY duration DESC LIMIT 1;
+--------------------------------+------------+-----------+
|          artist_name           |   title    | duration  |
+--------------------------------+------------+-----------+
| Mystic Revelation of Rastafari | Grounation | 3034.9058 |
+--------------------------------+------------+-----------+
1 row selected (0.317 seconds)
```

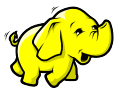
Therefore, the band `Mystic Revelation of Rastafari` has recorded `Grounation` which has highest duration.



Detailed Solutions

5. Top 5 hot songs of given band

```sql
SELECT
    title,
    artist_name,
    song_hotness
FROM
    songs
WHERE
    artist_name = 'The Beatles'
ORDER BY
    song_hotness DESC
LIMIT 10;
```



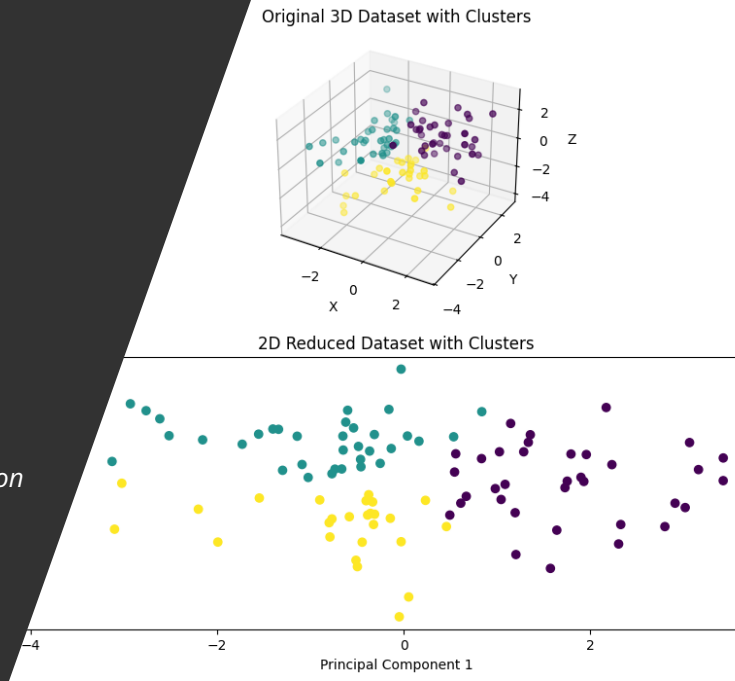
Detailed Results

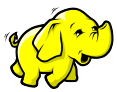
```
+-----------------------------------------------------------+
|                          title                            |
+-----------------------------------------------------------+
| In Sydney With Bob Rogers                                 |
| Larry Kane Interviews Derek Taylor                        |
| John Lennon on Exams                                      |
| George_ John_ Ringo and Fans in Auckland To Bob Rogers    |
| Derek Taylor - Introduction                               |
+-----------------------------------------------------------+
5 rows selected (0.416 seconds)
```

Therefore, SQL is good for queries with specified conditions. However, for song recommendation, we want to be more "clever" to dig into relationship between songs...

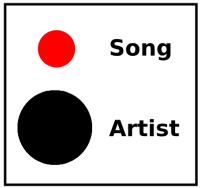
MileStone 2: Big Data Recommendation

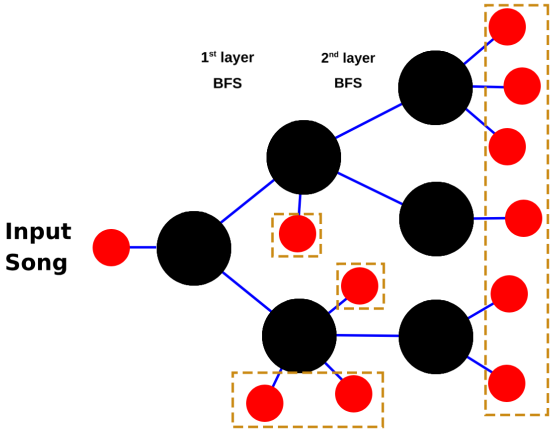
Suggest similar songs based on song input

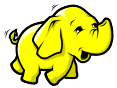




Similar Artists based BFS





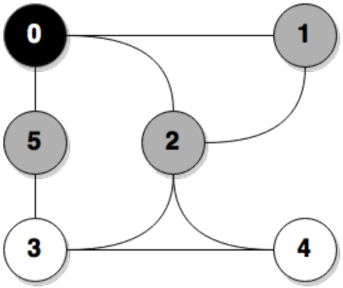


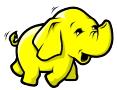
Parallel BFS: Graph representation

All nodes represented in either:

- **Black:** Processed root node.
- **Gray:** Discovered neighbour node.
- **White:** Not discovered node.

MapReduce job is run multiple times. In every job, all nodes are processed. Every job run processes one level of the graph.





Parallel BFS: Visualization

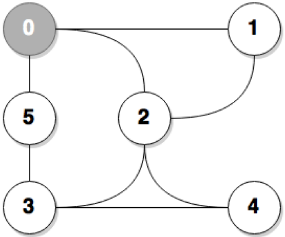


Figure: Iteration 0.

The starting node is colored gray when graph is constructed

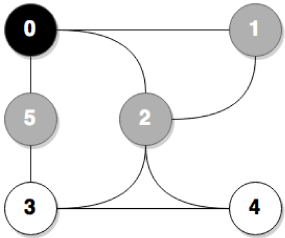
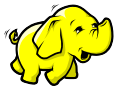


Figure: Iteration 1.

In each job iteration, nodes are recolored.
Neighbour nodes' distances incremented.
Root and neighbour nodes are emitted.



Parallel BFS: Visualization

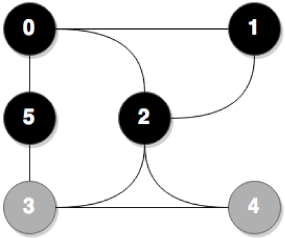


Figure: Iteration 2.

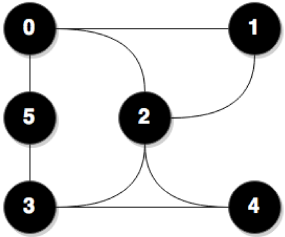
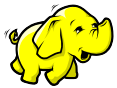


Figure: Iteration 3.

MapReduce Implementation

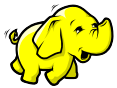
```
def steps(self):
    return [MRStep(mapper=self.map_step, reducer=self.reduce_step) for _ in
    ↪  range(BFS_DEPTH)]
def map_step(self, _, line):
    node = GraphNode()
    node.from_string(line)
    if node.state == 'WHITE':
        self.increment_counter('', "Remaining Nodes", 1)
    if node.state == 'GRAY':
        for neighbor in node.edges:
            neighbor_node = GraphNode()
            neighbor_node.nodeID = neighbor
            neighbor_node.distance = node.distance + 1
            neighbor_node.state = 'GRAY'
            yield neighbor, neighbor_node.to_string()
        node.state = 'BLACK'
    yield node.nodeID, node.to_string()
```



MapReduce Implementation

```
def reduce_step(self, key, values):
    all_edges, min_distance, final_state = [], float('inf'), 'WHITE'
    for value in values:
        node = GraphNode()
        node.from_string(value)
        if node.edges:
            all_edges.extend(node.edges)
        min_distance = min(min_distance, node.distance)
        if node.state == 'BLACK':
            final_state = 'BLACK'
        elif node.state == 'GRAY' and final_state == 'WHITE':
            final_state = 'GRAY'

    result_node = GraphNode(nodeID=key, distance=min_distance, state=final_state,
    ↪  edges=all_edges)
    yield key, result_node.to_string()
```



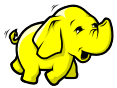
PySpark Implementation

```
def spark(self, input_path, output_path):
    sc = SparkContext(appName='mrjob Spark BFS script')

    lines = sc.textFile(input_path)
    nodes = lines.map(self.from_line)

    for iteration in range(BFS_DEPTH):
        mapped = nodes.flatMap(self.map_step)
        nodes = mapped.reduceByKey(self.reduce_step)

    nodes.saveAsTextFile(output_path)
    sc.stop()
```



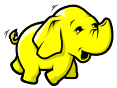
PySpark Implementation

```
def map_step(self, node):
    nodeID, (edges, distance, state) = node
    results = []

    if state == 'WHITE':
        self.increment_counter('', "Remaining Nodes", 1)

    if state == 'GRAY':
        for neighbor in edges:
            results.append((neighbor, ([], distance + 1, 'GRAY')))
        state = 'BLACK'

    results.append((nodeID, (edges, distance, state)))
    return results
```

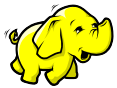


PySpark Implementation

```
def reduce_step(self, data1, data2):
    edges1, distance1, state1 = data1
    edges2, distance2, state2 = data2

    all_edges = list(set(edges1 + edges2))
    min_distance = min(distance1, distance2)
    final_state = 'BLACK' if state1 == 'BLACK' or state2 == 'BLACK' else 'GRAY' if
    ↪  state1 == 'GRAY' or state2 == 'GRAY' else 'WHITE'

    return (all_edges, min_distance, final_state)
```

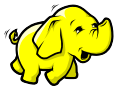


Parallel BFS: In Action

Once the suggested artists are found, all of their songs are queried. Among the queried songs, a song with the highest score as the addition of song hotness and artist hotness is suggested. An example run with BFS depth 2:

Input Song: Billie Jean
Artist: Michael Jackson

Suggested Song: There Goes My Baby
Artist: Usher



MapReduce vs PySpark

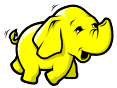
MapReduce runtime:

```
real    1m17.905s
user    1m8.467s
sys     0m9.373s
```

PySpark runtime:

```
real    0m28.160s
user    0m0.806s
sys     0m1.620s
```

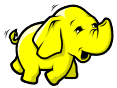
For any given depth of BFS, PySpark is running more than twice as fast as MapReduce!



Alternative Rec: *Preparation*

First question: What methods to apply?

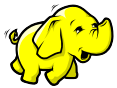
Clustering algorithm is an unsupervised machine learning method. By clustering user behaviors or items, the accuracy of the recommendation system can be improved. It seems good to be employed in our music app!



Alternative Rec: *Preparation*

Second question: Which song is the input song?

Song id: SOCIWDW12A8C13D406
Title: Soul Deep
Album id: 300822
*Album name:*Dimensions

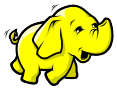


Alternative Rec: *Preparation*

Third question: According to which features, we recommend similar songs?

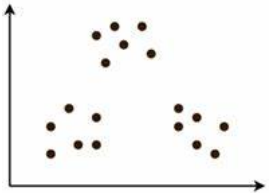
```
['bar_num','beat_num','segments_loudness_max_time_mean','artist_familiarity',
    'key','duration', 'end_of_fade_in', 'loudness', 'tempo',
    'time_signature']
```

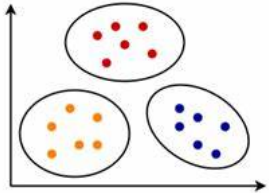
Finally, we can get down to business!

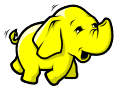


Alternative Rec1: *K-means*

What is K-means?







Alternative Rec1: *K-means*

How we realize our k-means?

1. Set 200 cluster centers.
2. Get 10 most related centers.
3. Calculate the distance to find the most similar songs in each cluster.
4. Evaluate the similarity by cosine correlation factors.

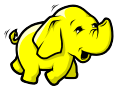
Goal: Recommend similar songs with some difference to refresh their ears!

Alternative Rec1: *K-means*

Result:

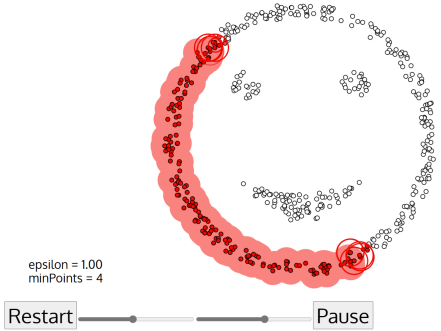
song_id	album_id	title	album_name	year	Cluster	Cos_correlation_factor
b'SOOWOFJ12A8C138EA4'	262484	b"I Was Tellin' Him About You"	b'A Lady In The Street'	0.0	152	0.999664
b'SOYONRY12A6D4FA815'	57486	b'King Of The Town'	b'The Place Where You Will Find Us'	2002.0	80	0.998218
b'SOOQVIL12AAF3B31DC'	382394	b'Everything Is Green '	b'Everything Is Green'	2004.0	28	0.981397
b'SOPKKJH12AB018E91A'	719655	b'Leelou (LMC Remix)'	b'Best of HSOLA 2009'	0.0	125	0.976981
b'SOUAWHK12A6D4FB9DD'	175510	b'Beautiful Soop (1966)'	b'Alien Bog / Beautiful Soop'	1997.0	47	0.950632
b'SOETEPE12AB018BB16'	727283	b'Anne'	b'...tot licht !'	2003.0	165	0.940049
b'SOYALRH12AC9071923'	602304	b'Babylon(e)'	b'Dogma'	0.0	136	0.896025
b'SOJDBKS12A6D4F9872'	27302	b'Two Voices'	b'Drawn From Life'	2001.0	6	0.866784
b'SOUXYES12AC468AB2D'	597790	b'Tranceplant'	b'Tranceplant'	0.0	167	0.842464
b'SOGNACS12A8C140DBA'	220126	b'Variation #9: Pantelleria'	b'Variations For Piano & Tape'	2006.0	103	0.825287

As you can see, similar but not the same!

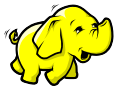


Alternative Rec2: *DBSCAN*

Its full name is: Density-Based Spatial Clustering of Applications with Noise



It performs quite well when there exists some noise.

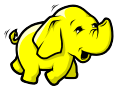


Alternative Rec2: *DBSCAN*

How we realize our DBSCAN?

1. dbscan = DBSCAN(eps=10, min samples=5)
2. Find out which cluster is the input song in.
3. Calculate distance to find 10 similar songs in the cluster.
4. Evaluate the similarity by cosine correlation factors.

Goal: Recommend really similar songs to the input song!

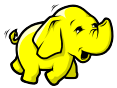


Alternative Rec2: *DBSCAN*

Result:

song_id	title	album_id	album_name	year	Cos_correlation_factor
b'SOCIWDW12A8C13D406'	b'Soul Deep'	300822	b'Dimensions'	1969.0	1.000000
b'SOTNVTQ12A6D4FC422'	b'True Lovers'	123952	b'Feeling Good'	2000.0	0.999996
b'SOAMGGA12AB0184F36'	b'Contigo'	461520	b'Sadel Canta a Los Panchos'	0.0	0.999992
b'SOOGBVU12AAA15E674'	b"Somebody's Stolen My Honey"	528419	b"If You Want Some Lovin'"	1991.0	0.999992
b'SOPQIWC12A8C13D035'	b'Mi Sangre Prisionera'	216258	b'Mi Sangre Prisionera'	0.0	0.999992
b'SOCAPRT12A6D4F6FB1'	b'Nineteen Fifty-Six (LP Version)'	34572	b'Collections'	0.0	0.999988
b'SOHJFQF12A58A77855'	b'The Way You Do The Things You Do'	212249	b'Moon River / Delicious Together'	0.0	0.999986
b'SOICLFE12A8C14102F'	b'My Low'	122491	b'Howie Beck'	0.0	0.999986
b'SOPWCCZ12A67020A26'	b'My 2 Arms - You = Tears'	12147	b'Cellar Full Of Motown Volume 2'	0.0	0.999984
b'SOMWDAK12A8C140AC2'	b'Natural Man'	566005	b'If You Love Me'	2003.0	0.999984

As you can see, they are quite similar. And now, let me show you some musics!



Alternative Rec2: *DBSCAN*

Music!

Alternative Rec: Some try...

Visualization by PCA:

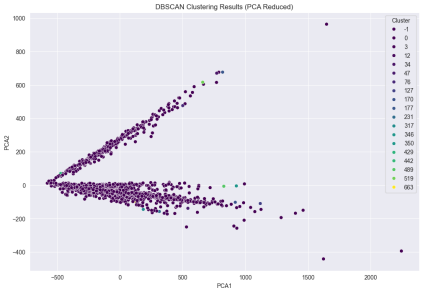


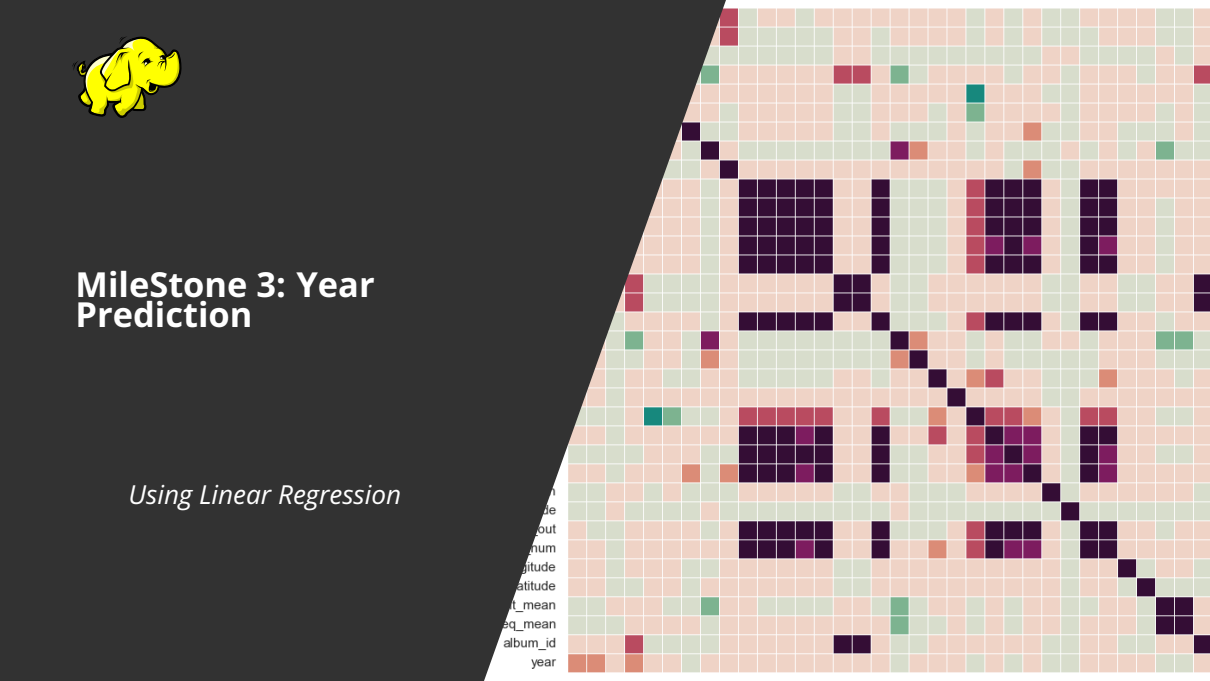
Figure: dbscan

Figure: k-means

Alternative Rec: Some try...

Fill the NAN:

```
avro_file = 'songs_advanced.avro'
with open(avro_file, 'rb') as f:
    avro_data = list(fastavro.reader(f))
df = pd.DataFrame.from_records(avro_data)
columns_to_fill =
    ['bar_num','beat_num','segments_loudness_max_time_mean','artist_familiarity',
    'key','duration', 'end_of_fade_in', 'loudness', 'tempo',
    'time_signature']
df[columns_to_fill]=df[columns_to_fill].fillna(df[columns_to_fill].mean())
df_selected = df.loc[:,columns_to_fill]
```



MileStone 3: Year Prediction

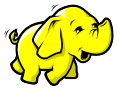
Using Linear Regression

Step 1: Choosing Columns

Intuition: reduce the amount of data

For numerical data, perform correlation analysis to select relevant ones;

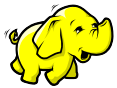
For non-numerical data, choose columns based on our evaluation.



Correlation Analysis

Feature	Value
loudness	0.272819
segments_loudness_max_mean	0.235492
artist_7digitalid	0.205400
release_7digitalid	0.111313
track_7digitalid	0.109126
song_hotness	0.099948
time_signature	0.085411
artist_longitude	0.078351
artist_latitude	0.022122

Table: Data table with some of correlation coefficients

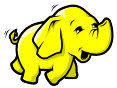


Step 2: Feature Engineering

Intuition: Get more useful information among columns

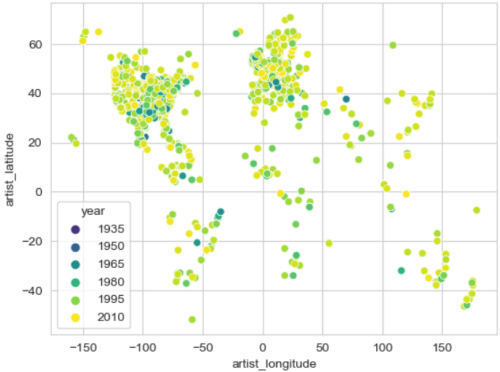
Album Name	Year
Bang!... The Greatest Hits of Frankie Goes to Hollywood	1985
More - EP	2007

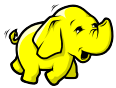
The release year might affect the length of the album name...
Correlation coefficient: -0.113036



Step 2: Feature Engineering

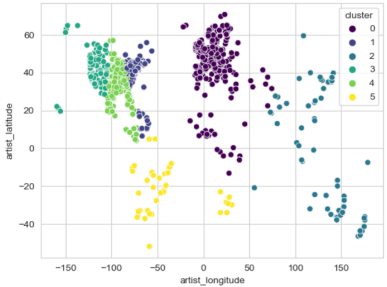
Combine artist's longtitude and latitude





Use K-means algorithm and divide into 6 groups, then one-hot encode them

Correlation coefficient (cluster_3) : -0.132685



Step 3: Data Preprocessing

Deal with left-skewed data

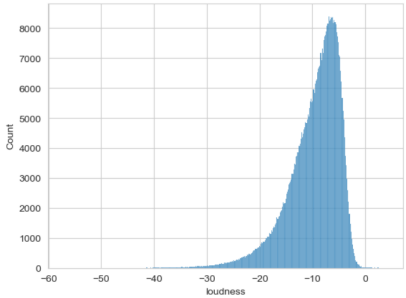


Figure: original data of loudness

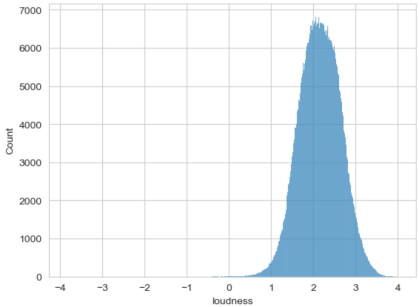
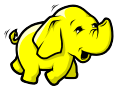
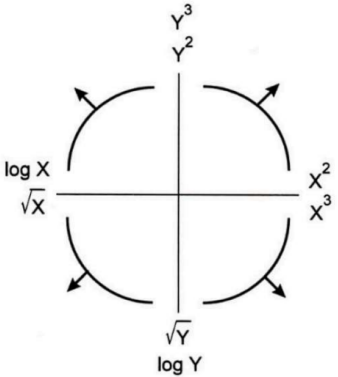


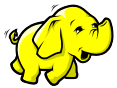
Figure: processed data after log transformation



Step 3: Data Preprocessing

Tukey's Bulging Rule:
Transform the values on the axes, hoping to get a linear relation





Step 3: Data Preprocessing

Tukey's Bulging Rule:
Transform the values on the axes, hoping to get a linear relation

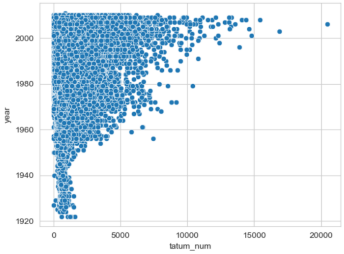


Figure: original data of tatum num

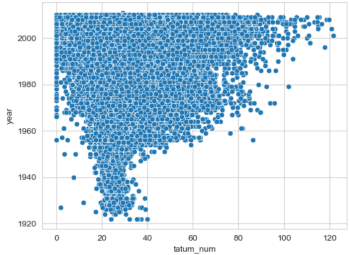
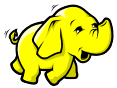


Figure: processed data after square root transformation

Step 3: Data Preprocessing

Other techniques:

1. remove outliers
2. replace NaN values with the means
3. data normalization
4. remove redundant columns
5. one-hot encode the categorical data



Step 4: Apply PCA algorithm to reduce dimension

Run PCA using on Spark

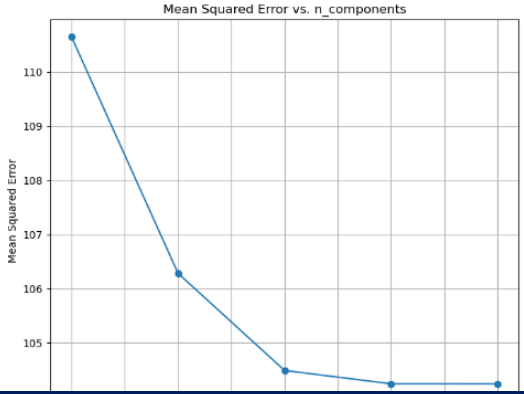
```
pca = PCAml(k=4, inputCol="features", outputCol="pca_features")
pca_model = pca.fit(train_data)
train_data_pca = pca_model.transform(train_data)
test_data_pca = pca_model.transform(test_data)
```

PCA time: 2.110 seconds

Step 4: Apply PCA algorithm to reduce dimension

Choose the number of components

n=4



Step 5: Linear Regression on Spark

Perform linear regression on Spark

```
lr = LinearRegression(featuresCol='pcaFeatures', labelCol='year',
↪  maxIter=1000, regParam=0.3, elasticNetParam=0.8)
lr_model = lr.fit(train_data)
# evaluate the model
predictions = lr_model.transform(test_data)
evaluator = RegressionEvaluator(labelCol="year",
↪  predictionCol="prediction", metricName="mse")
mse = evaluator.evaluate(predictions)
```

Training time: 2.384 seconds

Step 6: Model evaluation

MSE: 97.3526
The release year is predicted within the range of 10 years

Sample tests:

Song	Predicted	Actual
Walk the Walk	2000	2000
Granted Wish	1995	1989
Dusty Roads	2003	2007
Crazy Mixed Up World	1993	1961

Table: Data table with some of predictions

Thank you!