Million Songs

Team *Win or Lose* - Kezhi, Yinchen, Shaoze, Jiache

Table of contents

1. Data Preparation

Mounting

The following codes are added to `~/.bashrc` to automatically mount millionsongs upon booting.

```
echo "password" | sudo -S sshfs /home/hadoopuser/ece472 -o allow_other -o Port=2223 ece472@focs.ji.sjtu.edu.cn: -o IdentityFile=~/.ssh/id_ed25519 1>/dev/null 2>/dev/null
echo "password" | sudo -S mount /home/hadoopuser/ece472/millionsong.iso /home/hadoopuser/ece472/ 1>/dev/null 2>/dev/null
```

H5 Files

```
Track.h5
├─ analysis
│  ├─ bars confidence
│  └─ ...
├─ metadata
│  ├─ artist terms
│  └─ ...
└─ musicbrainz
   ├─ artist mbtags
   └─ ...
```

- We used **python h5py** module to extract information in *.h5* files.
- We collect useful fields in all *.h5* files and summarize them in one single *.avro* file

avro File

- Divide the dataset into 26 parts
- Create one *.avro* file for all files in each part
- Use **pyspark** to parallelize the process
- Merge 26 small *.avro* files into one single big file

With 8 cores processing in parallel, the total time cost to form one *.avro* file is reduced from 6 hours to 3 hours.

The *.avro* file generated which is consisted of all desired features of one million songs is approximately 150Mb.

Table: Statistic of Raw Data

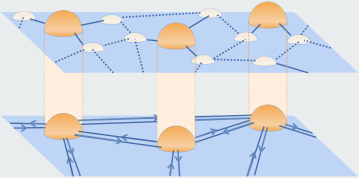
	tempo	hotness	year	time_signature	...
count	1000000	581965	1000000	1000000	
mean	123.889	0.356	1030	3.59	
std	35.056	0.234	999	1.22	
min	0.000	0.000	0	0	
25%	97.995	0.215	0	3	
50%	122.086	0.378	1969	4	
75%	144.089	0.532	2002	4	
max	302.300	1.000	2011	7	

What features are needed to describe a Song?

1. Duration
   - The duration cannot directly determine the style of songs
   - Adjusting granularity & OneHotEncoder
2. Segments_loudness_max / Segments_loudness_max_time
   - The most load segment can be regarded as the **"musical climax"**
   - Classify this feature by the occurrence of musical climax.
3. Segments_pitches
   - Analyze **emotion** of song by computing the mean of pitches.
   - Associate with the Segments_loudness_max

Prepare the data for Model:

- Missing & Abnormal Value
- Distribution (Left/Right Skew)
- Granularity (Year)
- Encoding (One-hot, Hash...)
- Combination
- Normalize & Standardize



Example: Loudness

Issues:

- Mostly < 0 with 359/1000000 abnormal values
- Left skewed

Methods:

- Replace exceptions with median
- $loudness\_log = ln(-loudness)$

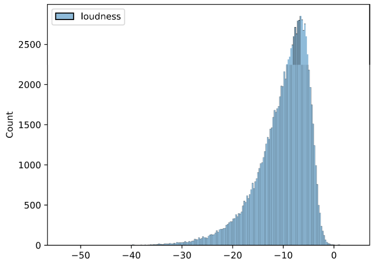


Figure: Distribution of Raw Data

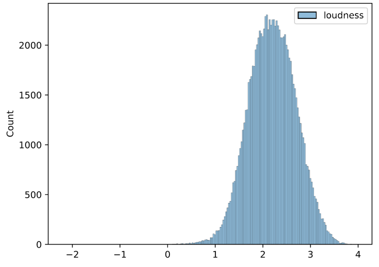


Figure: Distribution of Adjusted Data

Pipeline

```
processed_data = process_data_gm(data, (
    # Customized Column Transformer
    (log_transform_negative_column, ['loudness'], None),
    (classify_year_column, ['year'], None),
    (proportion_fade_out, None, None),
    (fade_out_time, None, None),

    # Exception Handling
    (drop_zeros, [List_of_Features], None),
    (fill_hotness_na_with_0, None,  None),

    # Normalize Selected Features
    (normalized_selected_columns, [List_of_Features], None),

    # Select Features
    (select_columns, ['Log_loudness', 'bar_num', 'beat_num',
                      # Add features here
                      ], None),
))
```

2. Drill Database Query

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*.

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

```
-- read avro file from local file system (~100M)
create table dfs.tmp.`songs` as
select * from dfs.`F:/avro/songs.avro`;
-- change path
use dfs.tmp;
```

1. the oldest and youngest songs

```
select max(year_end) from songs where year_end > 0;
+--------+
| EXPR 0 |
+--------+
| 2011   |
+--------+
1 row selected (0.321 seconds)

select min(year_end) from songs where year_end > 0;
+--------+
| EXPR 0 |
+--------+
| 1922   |
+--------+
1 row selected (0.323 seconds)
```

Therefore, the dataset covered songs from 1922 to 2011, namely the age of the songs vary from 12 years to 101 years.

Detailed Solusions

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

```
select id, title from songs
where hotness <> 'NaN'
order by hotness desc, duration asc, energy desc, tempo asc
limit 5;

+--------------------+------------------------------------+
|         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 (0.497 seconds)
```

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

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;

+----------+------------------------------------+----------+
| album_id |             album_name             | numSongs |
+----------+------------------------------------+----------+
| 60509    | First Time In A Long Time:         | 85       |
|          | The Reprise Recordings             |          |
+----------+------------------------------------+----------+
1 row selected (1.076 seconds)
```

First Time In A Long Time: The Reprise Recordings is the album with most tracks. Indeed, it has 4CDs and 80+ tracks.

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 (0.468 seconds)
```

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

3. Big Data Recommendation

Similarity Metrics

Similarity between $Song_A = [a_1, a_2, ..., a_n]$ and $Song_B = [b_1, b_2, ..., b_n]$

1. L_1 Norm

$$d_{L_1} = \sum_{i=1}^{n} |a_i - b_i|$$

2. Cosine Similarity

$$cos\theta = \frac{\sum_{i=1}^{n}(a_i \times b_i)}{\sqrt{\sum_{i=1}^{n} a_i^2} \times \sqrt{\sum_{i=1}^{n} b_i^2}}$$

3. Combination

$$Similarity(Song_A, Song_B) = \lambda cos\theta - d_{L2}$$

Similar Artists based BFS

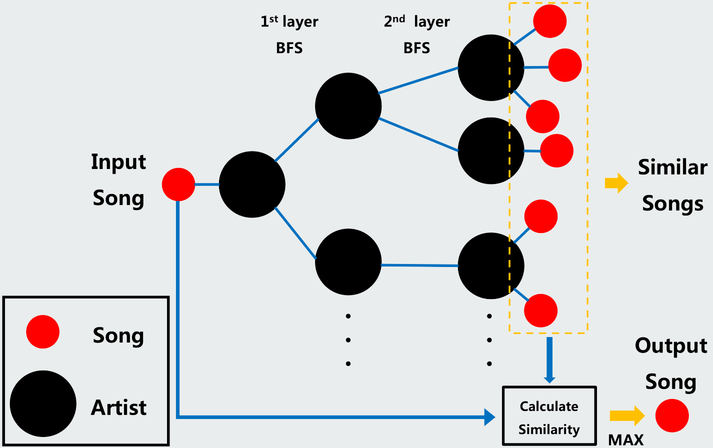


Figure: Two layers BFS

Example

Input song: (Old Man Mose, The Bristols)
Recommended Song:

1. L_1 Norm: (The Story of Two, Micragirls)
2. Cosine Similarity: (Kentish (demo), Modwheelmood)
3. Combination:
   When $\lambda \leq 307$, (Kentish (demo), Modwheelmood).
   When $\lambda \geq 308$, (The Story of Two, Micragirls).

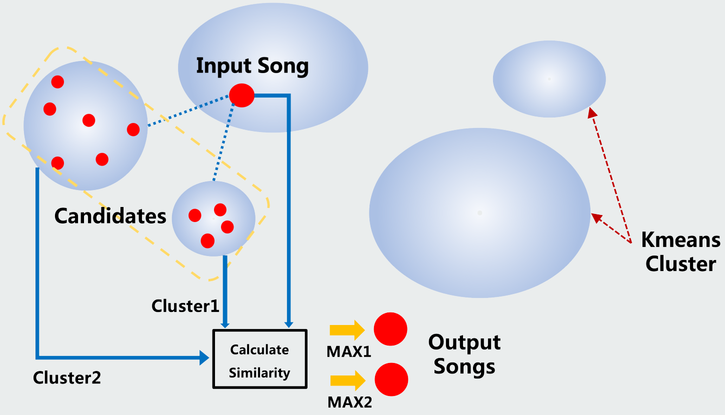
Diverse Recommendation Implementation

Choose different λ to reach diverse recommendations!

```python
import numpy as np
from numpy import ndarray
def calcDistance(song1: tuple, song2: ndarray) -> float:
    # song1: (feature: ndarray, track_Id: str)
    weight = 325
    return weight * (np.dot(song1[0], song2) / \
           (np.linalg.norm(song1[0]) * \
           np.linalg.norm(song2)) \
           - np.sum(np.abs(song1[0]-song2)), song1[1])

```

KMeans Recommendation



KMeans Recommendation

```
def getKMostRelatedCenter(track_id, data, cluster_centers,
↪  k=3)-> list:

song = data[data["track_id"] == track_id]
song = song.loc[:, features_list].to_numpy()
dis = []
for center in cluster_centers:
    dis.append(np.dot(song, center) / (np.linalg.norm(song) *
    ↪  np.linalg.norm(center)))

sorted_np = np.argsort(np.array(dis), axis=0)[:,0] < k
index = range(0, np.shape(cluster_centers)[0])

cluster_centers_withindex = np.hstack((cluster_centers,
↪  np.array(index).reshape(-1, 1)))
selected_centers = cluster_centers_withindex[sorted_np][:, -1]
return list(selected_centers)
```

MapReduce Implementation

Function of `mappers` and `reducer`:

- `mapper_artist.py`: given an artist id as input, find all the similar artists (according to the database)
- `mapper_song.py`: given the similar artists, find all of their songs
- `mapper_distance.py`: given the list of songs, calculate their cosine similarity from the given song
- `reducer.py`: find the song with largest similarity

MapReduce Performance

We implement a `driver.sh` to run the three map reduce job:

```
# How to run
cd MapReduce
time bash ./driver.sh
# Result
+ hdfs dfs -cat /project_distance/part-00000
('Story Of Two', 'TRMHEEB12903C9F3C9',
↪ 0.9140447260983122)

real    4m18.674s
user    1m27.983s
sys     0m5.830s
```

Pyspark Implementation

Use the same strategy, we implement the map and reduce in spark:

```
sc = SparkContext()
# find the list of similar artists
for i in range(depth):
    artists += sc.parallelize(artists, 4)\
                .map(artistNeighbor).reduce(merge_lists)
# find all the songs of similar artists
songs: list = sc.parallelize(artists, 12)\
              .map(getArtistSongs).reduce(merge_lists)
# find the feature of the input song
features = sc.parallelize(features, 100)\
             .map(lambda x: (np.concatenate((x[1:2], x[3:-7]))\
             .astype(np.float64), (x[-6],x[-5],x[-1]))).collect()
# reduce to get the song with largest similarity
result = sc.parallelize(features, 100)\
           .map(lambda x: calcDistance(x, songFeature))\
           .reduce(lambda x, y: max(x, y))
```

For pyspark, there is an interface for you to play with :).

Pyspark Performance

```
Please enter the name of a song:  Old Man Mose
Too many songs have the same name! Please choose a specific author
↪  from the list:
['Jesse Fuller', 'George Lewis And His New Orleans Stompers', 'Louis
↪  Armstrong', 'Manhattan Transfer', 'Kenny Ball And His Jazzmen',
↪  'The Bristols']
The author of your song: The Bristols
The song you choose:
Name: Old Man Mose, Author: The Bristols, Id: TRYESJS12903CDF730
Please enter the depth of the BFS: 2
Num of similar artists in the 1th layer: 48.
Num of similar artists in the 2th layer: 1134.
Num of similar songs: 29818.
(0.914044726098312 3, ("Story Of Two", "TRMHEEB12903C9F3C9", "The
↪  Micragirls"))

real    0m32.016s
user    0m2.668s
sys     0m2.587s
```

That is **8** times speed up than Mapreduce!

4. Year Prediction

Data Selecting

We want to train a model to predict the year of a song with its features.

```
SELECT COUNT(*) AS year_num FROM songs WHERE year<>0;
+----------+
| year_num |
+----------+
| 515576   |
+----------+
1 row selected (0.402 seconds)
```

Approximately half of the songs have years labeled. We use 80% of them as training set and 20% as validation set.

Data prepossessing

- Years are grouped by every 5 neighboring years so that total number of classes is reduced from 88 to 18.
- Each feature is normalized to the range of $[0, 1]$
- Very few data are missing (less than 1%). We just fill them as 0 and it would be not a big deal to the model.

We fitted a PCA model on the training set, which keeps 6 principal components from 15 original features.

```
pca = PCA(k=6, inputCol='features', outputCol='pca_features')
pca_model = pca.fit(feature_df)
return pca_model
pca_df = pca_model.transform(feature_df)
res_df =pca_df.select('coarse_classified_class_year',
'pca_features').rdd.map(lambda x: Row(tag=x[0],
PC1=float(x[1][0]), PC2=float(x[1][1]),
PC3=float(x[1][2]), PC4=float(x[1][3]),
PC5=float(x[1][4]), PC6=float(x[1][5])))
.toDF()
```

Model training

We trained a logistic regression model to do the classification task.

```
lrm = LogisticRegressionWithLBFGS.train(
sc.parallelize(data_train, 200), iterations=200,
numClasses=18)
print("Weights:", lrm.weights)
lrm.save(sc,'file:///home/hadoopuser/project
             /predict/model')
```

Prediction results

```
(dfy['coarse_classified_class_year']==
 dfp['year_predict']).value_counts()
```

No PCA

```
False    79282
True     23275
Name: count,
dtype: int64
```

Applying PCA

```
False    71696
True     30861
Name: count,
dtype: int64
```

Applying PCA raises prediction accuracy from 22.69% to 30.09%. For a 18 classification task, this result is satisfying enough.

Thank you!