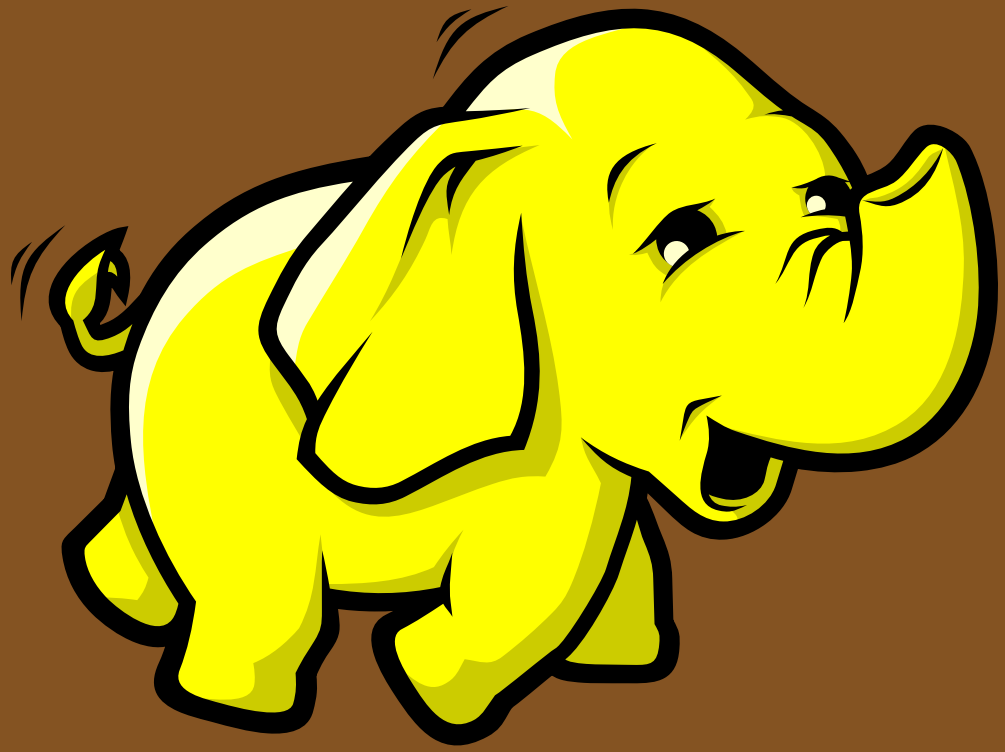


RecomMuse

Your soundtrack to the past, presents the future

ECE472 – Summer 2025

Kantaphat Leelakunwet, Choo Lee Wen, Arsen Aghayan, Ibrahim Daurenov

Table of contents

1. Data Processing

Motivation

Challenge

- One million tiny HDF5 files
- RAM overload on namenode

Solution

- Use JHDF to extract relevant features
- Compact small files into a larger one

JHDF (Java HDF5 Library)

- Pure Java. Easy integration with JVM based project.
- No use of JNI. Avoid issues with calling native code from JVM.
- It provides fast file access and reading is parallelized.

Avro Compaction

- Tailored Avro schema - keeping only essential features such as timbre, tempo, etc for year prediction and drill queries.
- Snappy compression
- Processed files alphabetically - A through Z
- Each letter batch took about 25-30 minutes - producing tidy 66 MB Avro files
- Finally merged everything into one single 1.7 GB file.
- **99.4%** Storage Reduction (Original is 286.6 GB)
- You could analyze it on a single commodity hardware easily.

2. Drill Queries

Drill

Our program allows users to interact with the compacted data through Drill SQL queries. Current supported queries are:

1. The age of the youngest and oldest song
2. The hottest song, which is the shortest with the highest energy and lowest tempo
3. The album with the maximum number of songs in it
4. The artist name of the longest song

Queries

The age of the youngest and oldest song

```
SELECT
    2025 - MAX(year) AS youngest_song_age,
    2025 - MIN(year) AS oldest_song_age
FROM dfs.root.`/home/hadoopuser/compacted/aggregate.avro`
WHERE year > 0;
+--------------------+-----------------+
| youngest_song_age | oldest_song_age |
+--------------------+-----------------+
| 14                 | 103             |
+--------------------+-----------------+
```

Queries

The hottest song, which is the shortest with the highest energy and lowest tempo

```
SELECT
    song_id,
    title
FROM dfs.root.`/home/hadoopuser/compacted/aggregate.avro`
WHERE song_hotttnesss <> 'NaN'
ORDER BY
    song_hotttnesss DESC,
    duration ASC,
    energy DESC,
    tempo ASC
LIMIT 1;
+--------------------+-----------------------------------+
|      song_id       |                title              |
+--------------------+-----------------------------------+
| SONASKH12A58A77831 | Jingle Bell Rock                  |
+--------------------+-----------------------------------+
```

Queries

The album with the maximum number of songs in it

```
SELECT
    release,
    COUNT(release) AS ntrack
FROM dfs.root.`/home/hadoopuser/compacted/aggregate.avro`
GROUP BY release
ORDER BY ntrack DESC
LIMIT 1;
+-----------------------------------------------------+--------+
|                       release                       | ntrack |
+-----------------------------------------------------+--------+
| First Time In A Long Time: The Reprise Recordings | 85     |
+-----------------------------------------------------+--------+
```

Queries

The artist name of the longest song

```
SELECT
    artist_name,
    duration
FROM dfs.root.`/home/hadoopuser/compacted/aggregate.avro`
ORDER BY duration DESC
LIMIT 1;
+-------------------------------+------------+
|          artist_name          |  duration  |
+-------------------------------+------------+
| Mystic Revelation of Rastafari | 3034.90567 |
+-------------------------------+------------+
```

3. Artist Distance

Problem Statement & BFS Overview

- **Problem:** Find the shortest path between two artists in a large-scale graph.
- **Graph:** A network of artists, with edges representing similarity or collaboration.
- **Algorithm:** Breadth-First Search (BFS)
  - Starts at a source node
  - Explores all neighbors at the present depth level before moving on to the nodes at the next depth level.
  - Guaranteed to find the shortest path in an unweighted graph.
- **Our Data:** A dataset of artists and their similarities.

Hadoop MapReduce Implementation

Data Representation

- *Key:* Artist ID
- *Value:* similar artists | distance | status | backpointer

Iterative Process

- *Job 1 (Initialization):* Marks Artist A as READY with distance 0. All others as NOT READY.
- *Jobs 2-11 (BFS Iterations):* A series of jobs run in a loop.
- *Classic MapReduce implementation:* Emits neighbors for READY nodes and the updated current node (Mapper) & merges node data to find the minimum distance and most advanced status (Reducer).

Output

- The final job output contains the data needed to trace the path back from Artist B to Artist A.

Apache Spark Implementation

Data Representation:

- Uses a Resilient Distributed Dataset (RDD)
- Each element is a tuple: (Artist ID, (similar artists, distance, status, backpointer))

Iterative Process

- RDDs and cache(): The graph data is loaded and cached in memory.
- flatMap (Mapper) & reduceByKey (Reducer): Same logic as MapReduce
- Accumulator: A global counter tracks when Artist B is found, acting as a termination signal.

Path Tracing

- The final RDD is collected to the driver, and the path is traced in-memory.

Comparison: Architecture & Performance

In-Memory vs. Disk-Based

- *MapReduce:* Writes intermediate BFS level results to disk (HDFS) after each iteration, causing high I/O overhead.
- *Spark:* Uses cache() to keep graph data in memory across iterations, dramatically reducing disk I/O.

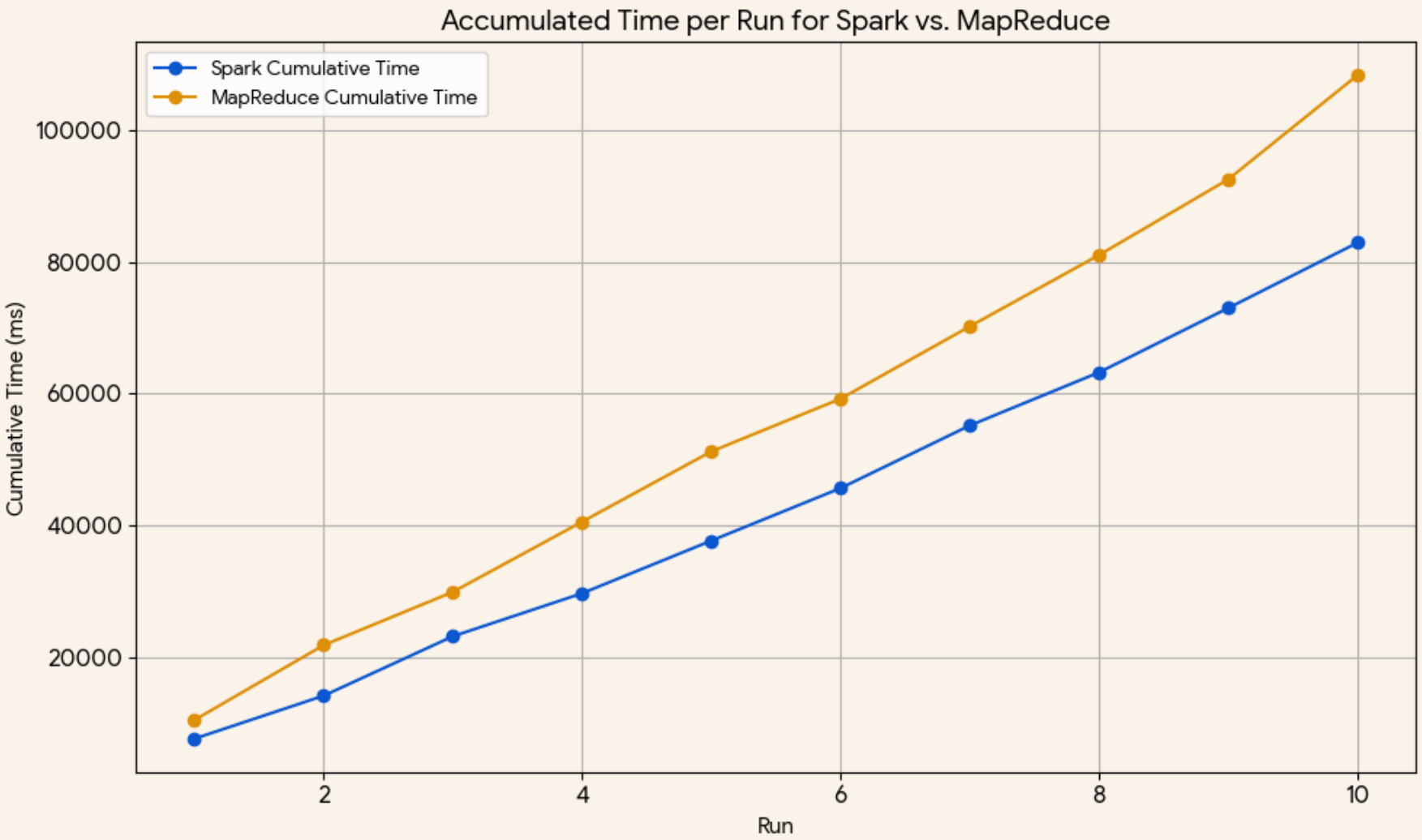
Iterative Processing Efficiency

- *MapReduce:* Each BFS iteration is a separate job with startup/teardown overhead.
- *Spark:* Designed for iteration; maintains a persistent execution context and reuses data, leading to faster multi-pass processing.

Result

- Both implementations yield identical and correct results, but with a significant difference in execution time.

Performance Visualization



Performance Visualization

```
# ========== MAPREDUCE ========== #
Target B (ARFK...) was found 4 levels from Target A (ARO6...)
It was connected to 3 artists at that level.
Tracing path from B to A...
Path found:
ARO6... -> ARF6... -> ARIW... -> ARSO... -> ARFK...

# ========== SPARK ========== #
Target B (ARFK...) was found 4 levels from Target A (ARO6...) and was
↪  connected to 3 artists at that level.
Path found:
ARO6... -> ARF6... -> ARH6... -> ARBQ... -> ARFK...
```

Those are the results for both Spark and MapReduce, proving our point for algorithm consistency and data accuracy.

Summary

- MapReduce: A powerful and robust tool for large-scale batch processing, especially for single-pass jobs.
- Spark: An ideal framework for iterative algorithms and machine learning, where in-memory caching provides a significant performance advantage.
- BFS Implementation: While both frameworks can solve the problem, Spark's architecture is fundamentally better suited for the iterative nature of BFS.

4. Year Prediction

Why Dimensionality Reduction? (PCA Motivation)

Correlation Analysis

- 64 numeric features per song
- **1569 out of 2016** pairs have correlation < 0.2
- **78%** of pairs are weakly correlated

Why it matters

- High dimensionality $\rightarrow$ noise, redundancy, slower models
- PCA finds uncorrelated directions with most variance

Result

- Reduced 64 features -> **32 components**
- Explained variance retained: **90%+**

Musical Features: Pitch and Timbre

Pitch

- What **notes** are used — C, D, E, ..., B
- Melody and harmony
- 12D vector (1 value per pitch class)

Timbre

- **Texture** of sound — smooth, sharp, warm, bright
- Depends on instruments and production
- 12D vector from audio spectrum (e.g., MFCC)

Why important?

- Pitch = **what is played**; Timbre = **how it sounds**
- Both change with time -> useful for predicting song year

PCA Component Comparison (Linear Regression)

- Tested PCA with $\mathbf{k = 10, 32, 52}$ components
- Fewer components $\rightarrow$ worse performance:
  - $\mathbf{R^2}$ dropped: $0.26 \rightarrow 0.21 \rightarrow 0.14$
    (how much variance is explained — higher is better)
  - **RMSE** increased: $9.37 \rightarrow 9.73 \rightarrow 10.14$
    (average prediction error in years)
- $\mathbf{k = 32}$ offers a good trade-off between performance and simplicity
- Problem of having too many features: hard to fit a single linear model

Trying SGD for Linear Regression

Goal:

- Improve performance by switching to **SGD-based linear regression**
- Hope: increase R^2 beyond previous baseline

What we did:

- Used PCA ($k = 32$) and normalized target label (year)
- Tuned key hyperparameters:
  - Learning rate, iterations, regularization

Result:

- $\mathbf{R^2} = 0.21$ (on normalized labels)

Despite tuning, performance remained close to baseline.
We reached the limitation of linear regression for this task.

Logistic Regression for Year Classification

Goal:

- Frame year prediction as a **multi-class classification** task
- Train logistic regression using **PCA-32 features**

Setup:

- Used pipeline: Assemble $\rightarrow$ Scale $\rightarrow$ PCA (k = 32)
- Tried different values for hyperparameters (e.g., `regParam`, `maxIter`)

Results:

- **Accuracy:** 9.7%

Mini-batch Gradient Descent in sklearn

First Attempt

- Use MLPRegressor from sklearn to examine how well timbre feature performs on year prediction

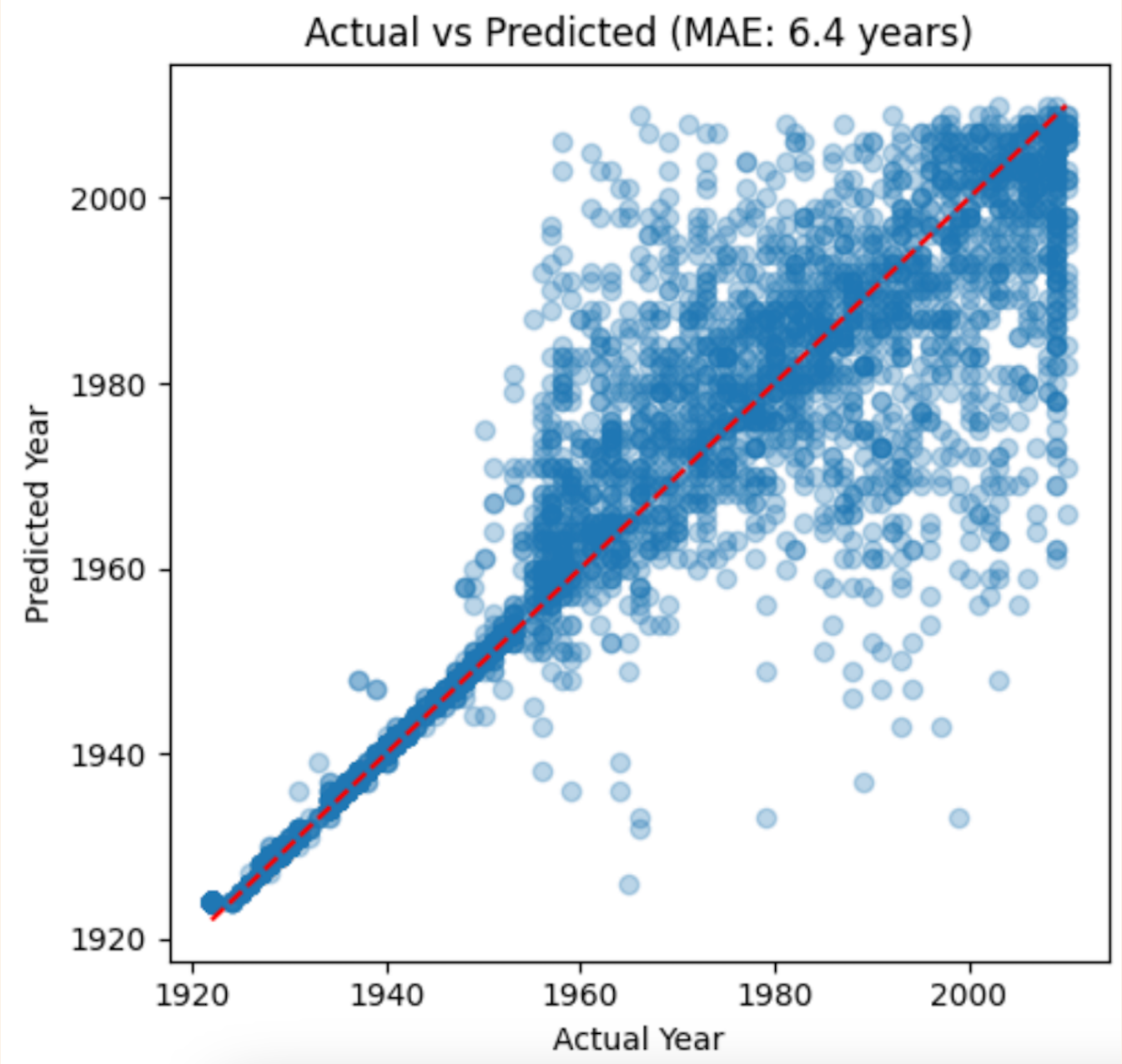
```
mlp = MLPRegressor(
    hidden_layer_sizes=(1024, 512, 512, 256, 256, 128, 64, 32),
    activation='relu',
    solver='adam',
    alpha=0.00001,
    batch_size=128,
    learning_rate='adaptive',
    early_stopping=True,
    validation_fraction=0.15,
    max_iter=1000,
    n_iter_no_change=20
)
```

Mini-batch Gradient Descent in sklearn

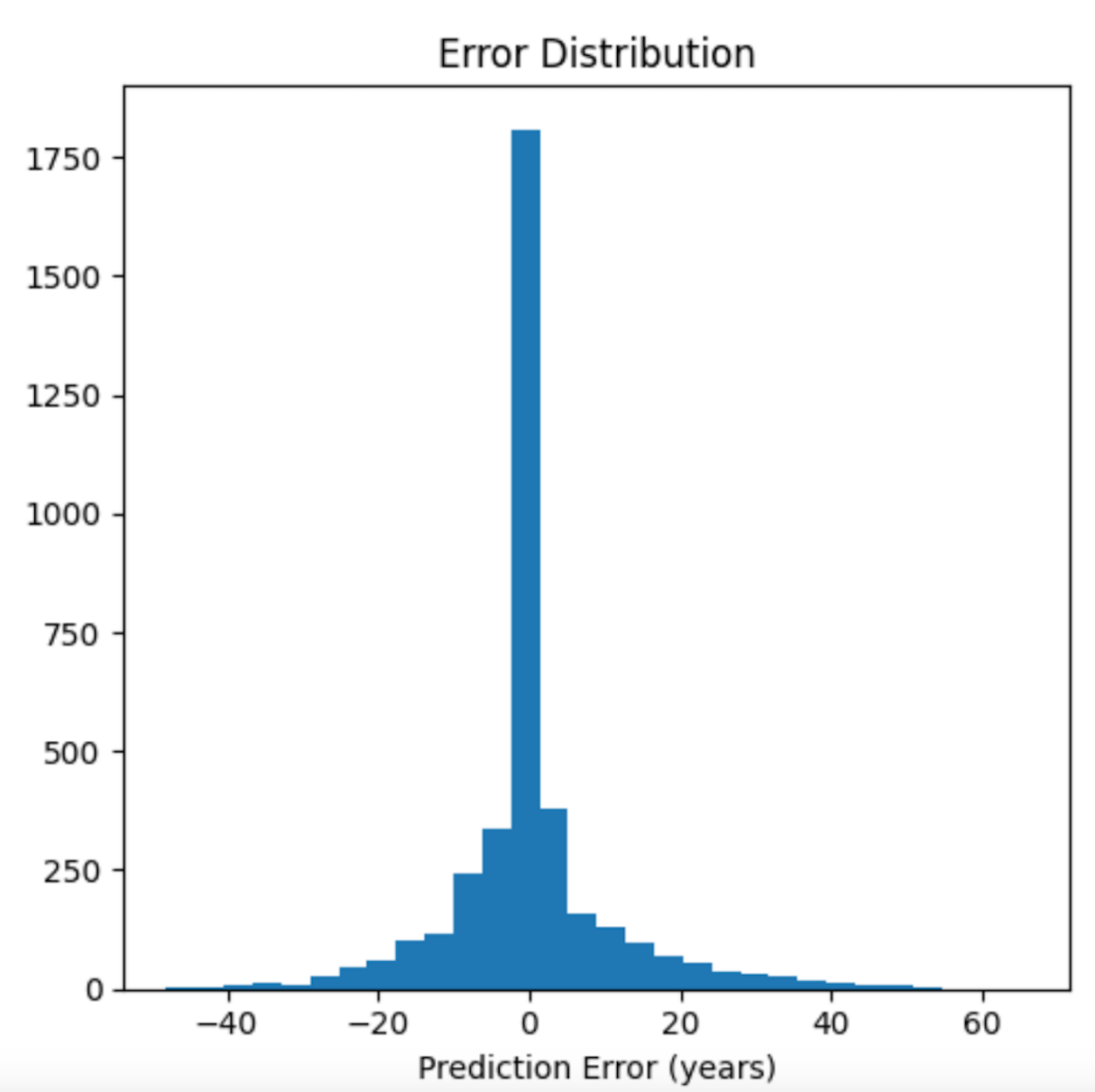
Outcome of MLPRegressor

- Mean Squared Error: 117.11
- Mean Absolute Error: 6.39 years

MLPRegressor



MLPRegressor

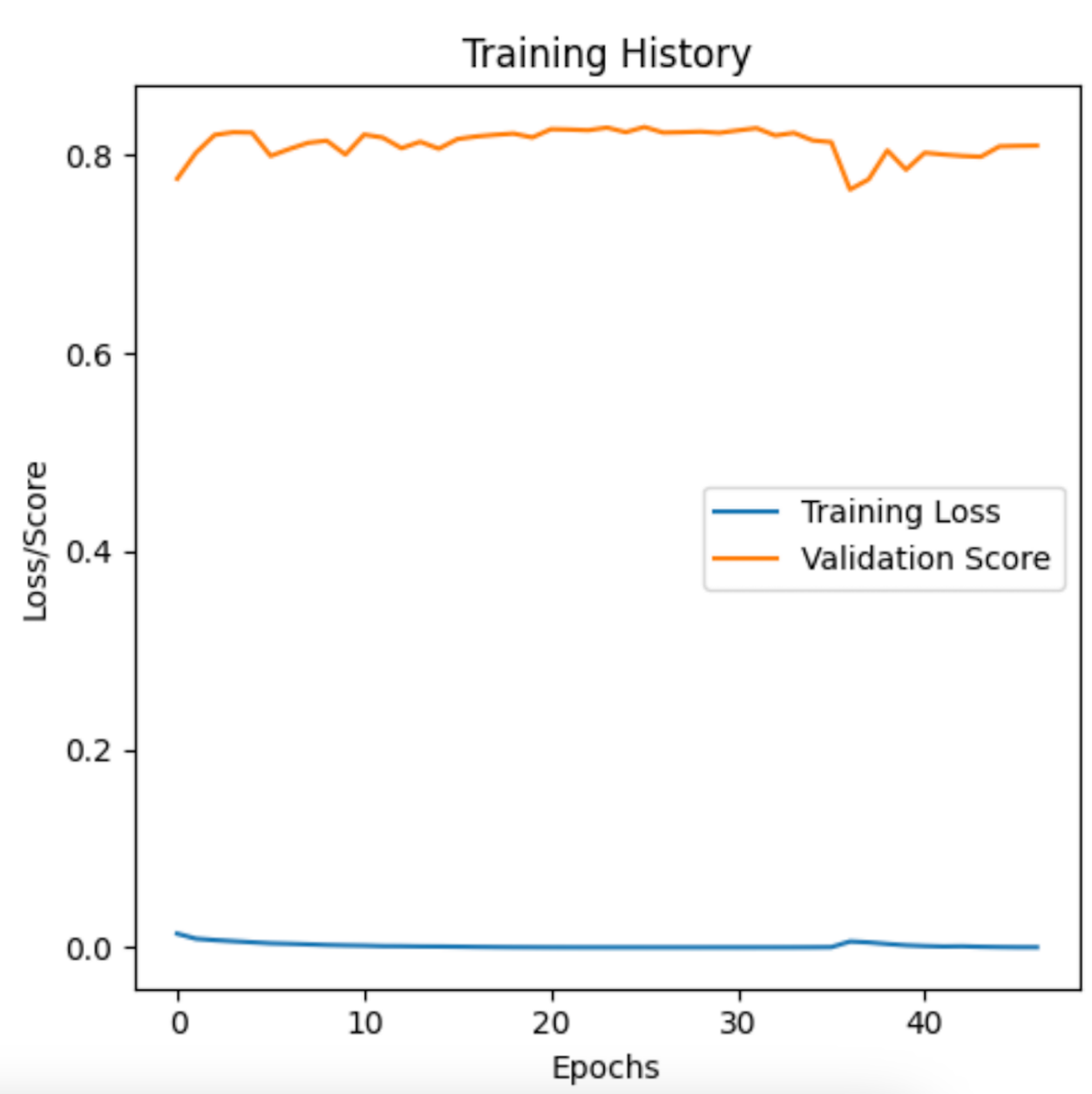


MLPRegressor

It makes sense if we counts the samples in decades.
Data Imbalance!

Decade	Song Count
1920s	17
1930s	20
1940s	56
1950s	508
1960s	3,164
1970s	7,021
1980s	10,994
1990s	32,133
2000s	70,718
2010s	2,043

MLPRegressor



Mini-batch Gradient Descent in Spark

FMRegressor

- Extends linear regression by modeling pairwise feature interactions efficiently using factorized parameters.
- Automatically detects relationships between features (e.g., how "timbre_1" and "pitch_3" jointly affect the year).

```
regressor = FMRegressor(
    labelCol="year_scaled",
    featuresCol="pcaFeatures",
    factorSize=4,
    solver="adamW",
    miniBatchFraction=0.2,
    maxIter=400,
    stepSize=0.05,
    regParam=0.01,
    seed=42
)
```

Mini-batch Gradient Descent in Spark

FMRegressor

- Extends linear regression by modeling pairwise feature interactions efficiently using factorized parameters.
- Automatically detects relationships between features (e.g., how "timbre_1" and "pitch_3" jointly affect the year).

```
regressor = FMRegressor(
    labelCol="year_scaled",
    featuresCol="pcaFeatures",
    factorSize=4,
    solver="adamW",
    miniBatchFraction=0.2,
    maxIter=400,
    stepSize=0.05,
    regParam=0.01,
    seed=42
)
```

Mini-batch Gradient Descent in Spark

Outcome of FMRegressor

1. timbre
   - RMSE: 10.82
   - MAE: 8.09 years
2. timbre + pitch
   - 95% variance PCA, k = 59
   - RMSE: 10.32
   - MAE: 7.58 years

However, the current R^2 is too bad! ($R^2 < 0.1$!). More architecture exploration will be needed.

Linear Regressor in Spark

LinearRegressor

- Latest Spark recommend using LinearRegressor instead of LinearRegressorWithSGD.
- **Model**: Linear Regression with Polynomial Features (degree=2)
- **Scaler**: RobustScaler (handles outliers better)
- **Optimizer**: L-BFGS (better convergence for small/medium datasets)
- **Balancing**: Stratified sampling (5-year bins, 1k samples per bin)

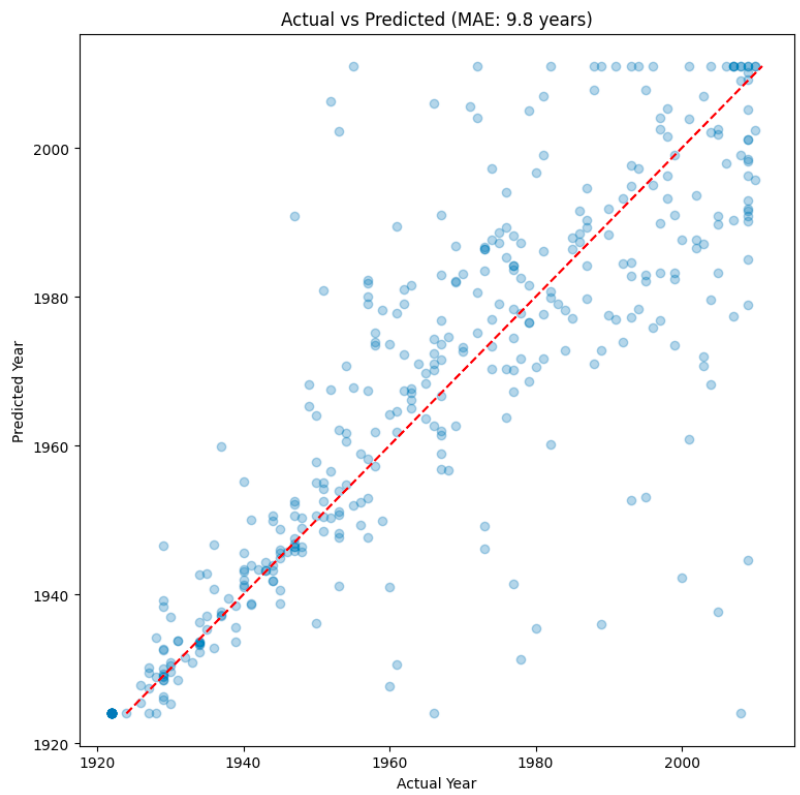
```
lr = LinearRegression(
    featuresCol="polyFeatures",
    labelCol="year",
    solver="l-bfgs",
    maxIter=400,
    tol=1e-6
)
```

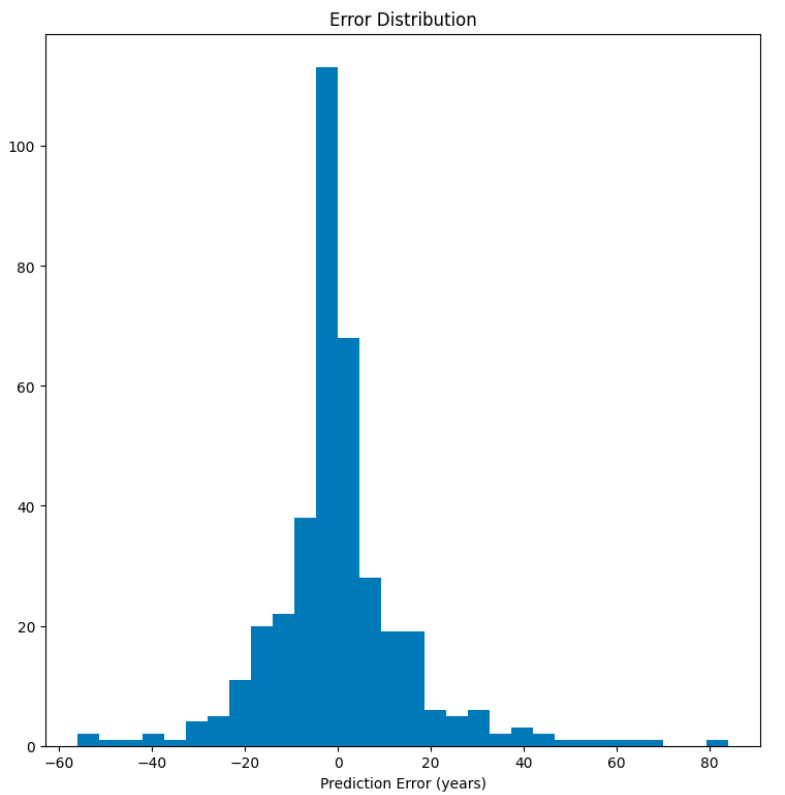
Linear Regressor in Spark

Outcome of LinearRegressor

- RMSE: 15.38
- MAE: 9.82 years
- R^2: 0.68

Linear Regressor in Spark

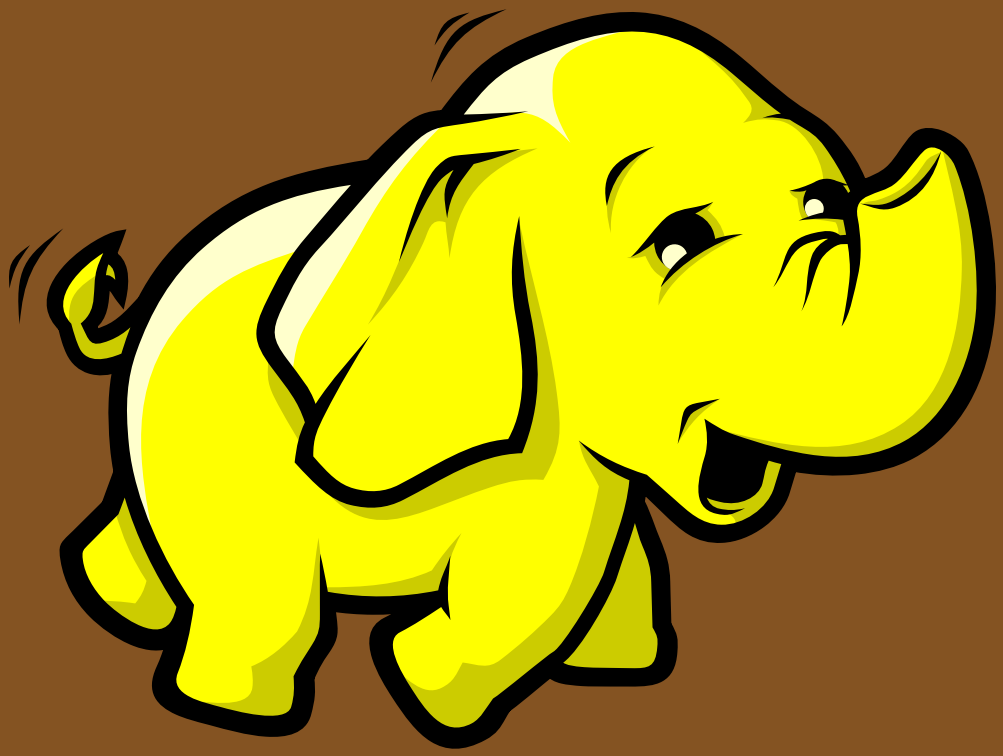




GBTRegressor in Spark

Outcome of GBTRegressor

- timbre
- RMSE: 9.60
- MAE: 6.89 years
- R^2: 0.20



Thank you!