COMP30770 Programming for Big Data

Extracting Trends in Movie Metadata

Project Report

Group Members	
Daniel Fallon	22393263
Colin Wang	22343536
Harry McCormack	22513539

Code Link: https://github.com/Daniel7Fallon/BigDataProject

Section 1

The Movies Dataset

This dataset integrates 45,000 movies from TMDB (metadata, cast/crew, keywords) with 26 million user ratings from GroupLens. It includes structured CSVs (e.g., movies_metadata.csv, ratings.csv) and semi-structured JSON-embedded files (e.g., credits.csv, keywords.csv), enabling cross-domain analysis like recommendation systems, box office prediction, and genre/director impact studies.
Source: https://www.kaggle.com/datasets/rounakbanik/the-movies-dataset?resource=download

Volume:

Size: 943.76 MB
Columns: 43
System Specs:
Some execution times from section 3.

Variety:

Dataset contains 7 files, ordered by descending size:

ratings.csv	692,921 KB
credits.csv	185,467 KB
movies_metadata.csv	33,638 KB
keywords.csv	6,086 KB
ratings_small.csv	2,382 KB
links.csv	966 KB
links_small.csv	180 KB

The dataset's variety stems from structural and functional differences across files. Structured data like *ratings.csv* (flat numerical columns) contrasts sharply with semi-structured files like *credits.csv*, where nested JSON fields describe actors' roles and crew departments.

These files can be combined to extract correlations between themes, cast, revenue and public reception. This is invaluable information to the film industry.

Section 2

Objective:

This project aims to analyse the resource requirements to extract trends and key insights into the film industry and film marketplace.

Value:

Companies within the film industry can use these insights to consult and guide them in their decisions of what projects to pursue.
Specifically, the application of Big Data processing is essential to this application as:

- Markets can be volatile, i.e. subject to sudden change.
- Movies are released to the global market at a staggering rate, multiple per day.
- Reviews of movies are written in their hundreds everyday. Websites like “Letterboxd”, a rapidly growing social media for reviewing films, is an example of how this practice is only becoming more common.

These three factors combined means that for the insights garnered to be relevant, this massive and growing dataset must be continually reprocessed. The quicker the turn over of the processing, the more relevant the insights.

Section 3. Traditional Solution (2.5 pages)

Language Used: Python 3.13.2
Steps:
1. Read in movies_metadata.csv file to pandas dataframe for convenient manipulation and perform necessary cleaning:

```
metadataDF = pd.read_csv('TheMoviesDataset/movies_metadata.csv', usecols=["id", "title", "budget", "genres", "popularity", "release_date", "revenue", "runtime", "vote_average", "vote_count"])
metadataDF['id'] = metadataDF['id'].astype('int64')
```

0.7s

2. Convert JSON entries to lists of elements and skip empty entries:

```
df_copy['genres'] = df_copy['genres'].apply(lambda x: ast.literal_eval(x) if pd.notna(x) else [])
```

0.6s

3. Count genre occurrences:

```
genre_counts = Counter(all_genres)
genre_counts_df = pd.DataFrame(genre_counts.items(), columns=['Genre', 'Count'])
```

0.0s

	Genre	Count
6	Drama	20265
1	Comedy	13182
9	Thriller	7624
5	Romance	6735
7	Action	6596
10	Horror	4673

4. Show mean vote_average and mean popularity per genre:

```
genre_stats = df_exploded_genres.groupby('genre_name').agg({
    'vote_average': 'mean', 'popularity': 'mean'}).reset_index()
```

0.6s

	genre_name	vote_average
2	Animation	6.275556
10	History	6.154220
18	War	6.041119
6	Drama	5.905226
12	Music	5.879599
	genre_name	popularity
1	Adventure	5.998335
8	Fantasy	5.363656
15	Science Fiction	4.997888
0	Action	4.770506
7	Family	4.729328

5. Read in credits.csv file to pandas dataframe for cast information:

```
creditsDF = pd.read_csv('TheMoviesDataset/credits.csv')
```

1.0s

6. Join credits and metadata information:

```
metadata_join_credits_DF = metadataDF.merge(creditsDF, on='id', how='inner')
```

0.0s

7. Show the correlation between an actor and how well their movies are received:

```
cast_stats = df_exploded_cast.groupby('cast_name').agg({
    'vote_average': 'mean',
    'popularity': 'mean',
    'id': 'count' #Count of films they have been in
}).reset_index()
```

9.3s

	cast_name	num_films	vote_average
76731	Idris Ali	1	10.0
52524	Elif Baysal	1	10.0
2899	Al Mackenzie	1	10.0
34789	Chuck Blackwell	1	10.0
147557	Pamela Craig	1	10.0
...	...	...	...
113019	Leo Bruckmann	1	0.0
12586	Anni Timm	1	0.0
181395	Sven Jerring	1	0.0
146781	Oscar Tengström	1	0.0
24	Rosenda Schaschmidt	1	0.0

	cast_name	num_films	popularity
4726	Alex Dowding	1	547.488298
173537	Shaun Newnham	1	294.337037
60652	Frank Allen Forbes	1	294.337037
57188	Eva Dabrowski	1	294.337037
179026	Steve Doyle	1	294.337037

*The extremes are populated with actors who have only been in one film, that being incredibly successful or entirely unseen.

8. Show correlation between how many movies an actor has been in and the reception of those movies:

```
numFilmsCorrelation = cast_stats.groupby('num_films').agg({
    'vote_average': 'mean',
    'popularity': 'mean'
}).reset_index()
```

0.0s

	num_films	vote_average	popularity
0	1	5.880034	5.169291
1	2	5.873058	5.313597
2	3	5.885797	5.195781
3	4	5.859514	4.925731
4	5	5.843277	4.742350
...	...	...	...
103	110	6.102273	5.975370
104	123	6.211382	11.706544
105	125	5.666400	3.092939
106	148	5.870270	4.749606
107	241	5.539004	2.028024

Section 4. MapReduce Optimisation (2 pages)

Please identify 1 or 2 most time-consuming steps in your Section 3 that can be optimised by big data programming paradigms: MapReduce. You are free to use either Hadoop MapReduce or Spark MapReduce (Spark Core API, NOT Spark SQL or Dataframe etc.)

- Explain why they can be optimised using MapReduce and present your expectations (e.g., reduce execution time by 2). (3')
- Present MapReduce solution (3') - Present MapReduce results. (3') - Explain why the results match or deviate from your expectations. (3')

Language Used: Python 3.10.0

1. Initialise pyspark:

```
import findspark
findspark.init()

from pyspark.sql import SparkSession
```

2. Read csv into pyspark dataframe:

```
from pyspark.sql.types import StructType, StructField, StringType, IntegerType, DoubleType, BooleanType

schema = StructType([
    StructField("adult", BooleanType(), True),
    StructField("belongs_to_collection", StringType(), True),
    StructField("budget", IntegerType(), True),
    StructField("genres", StringType(), True),
    StructField("homepage", StringType(), True),
    StructField("id", IntegerType(), True),
    StructField("imdb_id", StringType(), True),
    StructField("original_language", StringType(), True),
    StructField("original_title", StringType(), True),
    StructField("overview", StringType(), True),
    StructField("popularity", DoubleType(), True),
    StructField("poster_path", StringType(), True),
    StructField("production_companies", StringType(), True),
    StructField("production_countries", StringType(), True),
    StructField("release_date", StringType(), True),
    StructField("revenue", IntegerType(), True),
    StructField("runtime", DoubleType(), True),
    StructField("spoken_languages", StringType(), True),
    StructField("status", StringType(), True),
    StructField("tagline", StringType(), True),
    StructField("title", StringType(), True),
    StructField("video", BooleanType(), True),
    StructField("vote_average", DoubleType(), True),
    StructField("vote_count", IntegerType(), True)
])
```

```
file_path = "TheMoviesDataset\\movies_metadata.csv"
movies_df = spark.read.csv(file_path, header=True, schema=schema)

movies_df.show(5)
```

3. Map reduce to count the genres:

```
movies_rdd = movies_df.rdd

genre_counts_rdd = movies_rdd.flatMap(
    lambda row: [
        (genre["name"], 1)
        for genre in row["genres_parsed"] or []
        if genre and genre["name"]
    ]
)

genre_counts = genre_counts_rdd.reduceByKey(lambda a, b: a + b)

results = genre_counts.collect()

for genre, count in sorted(results, key=lambda x: -x[1]):
    print(f"Genre: {genre}, Movie Count: {count}")
```

9.1s to run.

4. Map reduce to get average rating by genre:

```
mapped_rdd = filtered_rdd.flatMap(
    lambda row: [
        (genre["name"], (row["vote_average"], 1))
        for genre in row["genres_parsed"] or []
        if genre and genre["name"]
    ]
)

reduced_rdd = mapped_rdd.reduceByKey(
    lambda a, b: (a[0] + b[0], a[1] + b[1])
)

average_ratings_rdd = reduced_rdd.mapValues(
    lambda x: x[0] / x[1]
)

results = average_ratings_rdd.collect()

for genre, avg_rating in sorted(results, key=lambda x: -x[1]):
    print(f"Genre: {genre}, Average Rating: {avg_rating:.2f}")
```

9.1s to run.

cpu: ryzen 7 5700x3d