Big Data Capstone Report

Bess Yang (qy561@nyu.edu), **Chloe Kwon** (ekk294@nyu.edu), **Iris Lu** (hl5679@nyu.edu)

- GitHub repository: https://github.com/nyu-big-data/capstone-project-cap-19
- Contribution:
  - Q1-2: Iris Lu
  - Q3: Bess Yang
  - Q4-5: Bess Yang, Chloe Kwon

Q1. List of top 100 most similar pairs (include a suitable estimate of their similarity for each pair), sorted by similarity.

What we did:

- Data Preprocessing:
  - We started by loading the ratings.csv into a Spark DataFrame. In order to apply CountVectorizer later, I explicitly typed to make movieId as a string.
  - This df is then grouped by userId and aggregates all movieIds into a list for each user, caching the results to optimize performance.
  - The grouped DataFrame is filtered to keep only those users who have rated at least 5 movies.
  - A CountVectorizer is applied to the grouped dataframe to transform the list of movieIds into a feature vector for each user, fitting the model on the filtered data and transforming the filtered data.
- Applying MinHash for Similarity Estimation:
  - We set up a MinHash LSH model, specifying the input and output columns along with the number of hash tables (5) to use.
  - The MinHash model is fitted on the vectorized DataFrame, and then it transforms the same DataFrame to append hash values.
  - We then perform an approximate similarity join on the transformed DataFrame with itself to find pairs of users with a Jaccard distance of 0.6 or less.
  - We filter out duplicate pairs (ensuring unique pairs by comparing userId) and sort the results by Jaccard distance in ascending order, limiting the output to the top 100 most similar pairs.
- Output:
  - We select and rename the relevant columns (user IDs and Jaccard distance) from the similarity pairs for clearer output.
  - Finally, the processed data is written out to a Parquet file and later on transformed it to a CSV file as requested.

Result:

After inspecting the results multiple times, we found out that there are >100 pairs of users having Jaccard Similarity = 1 (users have been filtered out to only include those who've rated at least 5 movies). We output the results to 2 files - one sorted by userId while the other one doesn't:

- [sorted] q1_all_atleast5movies_allJD0/q1_allJD0_ordered100.csv [link]

- [unsorted] q1_all_atleast5movies_top100/part-00000-e47cd1b2-31e4-4255-aba6-9b158e0bbd4e-c000.csvv [link]

We will continuously use these 2 files to answer question 2.

Conclusion:
The first 100 pairs of users, who rated at least 5 movies, all have Jaccard Distance = 0, meaning they've all rated the exact same movies.
We tested 3 different filters on how many movies they've rated (no filter on number of movies rated; rated at least 3 movies; rated at least 5 movies) and all yielded the same result. The reason behind this testing is that we think it wouldn't be representative to pick the pairwise users who only rated 1 movie as the most similar pairs. Therefore, we filtered users who rated at least 3 movies and at least 5 movies and rerun the code. Finally, we decided to go for the pairs that are in "rated at least 5 movies" as the most similar pairs for the calculation in Q2.

Q2. A comparison between the average pairwise correlations between these highly similar pairs and randomly picked pairs.

What we did:
- Preparing userId dataframes
  - Selecting the random pairs: We randomly select 100 userId pairs (without duplicates and without pairing with themselves) and output it into the same format as the top 100 userId dataframe.
  - Loading Jaccard Similarity = 0, ordered by userId dataset
  - Loading Jaccard Similarity = 0, not ordered by userId dataset
- Preparing Rating Data for Common Ratings DataFrame:
  - *we apply the below method for all the 3 DataFrame
  - From the pairwise userIds DataFrame, we select the unique userIds and join it with the ratings data to filter out ratings that only belong to the users for faster computation. The result is cached for efficient access during subsequent operations.
  - We then perform a self-join on the filtered ratings DataFrame to find pairs of ratings where the same movie is rated by different users. Then we apply VectorAssembler to transform the ratings of each user pair into a feature vector, which is necessary for correlation calculation.
- Computing Correlations:
  - We iterate over the DataFrame to calculate correlation for the pairwise users and store the results into a correlation dictionary
  - We then compute the average correlation from the correlation dictionary

Result:
- Top 100 pairs (not sorted by userId) average numerical ratings correlation is 0.007;
- Top 100 pairs (sorted by userId) average numerical ratings correlation is 0.127;
- Random 100 pairs average numerical ratings correlation is 0.169.

Conclusion:

When we inspected the top 100 pairs (not sorted by userId) correlation data, we found that there are a lot of negative correlations (-0.8, -0.7, etc) in the correlation dictionary, which means the users who rated the same movies have opposite opinions hence resulting in the average correlation being low, thus makes the average correlation low (0.007). Since there are >100 user pairs that have Jaccard Similarity = 1, we also calculate the average correlation from the top 100 pairs (sorted by userId) data. This time, we got an average correlation of 0.127.
We concluded that there's no definite average correlation relationship between the top 100 pairs and the random 100 pairs since we have a quite large base of user pairs that Jaccard similarity = 1. It might be the case that people who rated the same movies have opposite tastes on the same movie or vice versa. Hence, the relationship between top 100 pairs and random 100 pairs could be either higher or lower.

Q3. Documentation of how your train/validation splits were generated and any additional pre-processing of the data that you decide to implement.

See `3_partition_all.py`

I partitioned the ratings dataset by user (i.e. each user's ratings are randomly partitioned) into training, validation, and test splits with a ratio of 7:1.5:1.5. This avoids the possibly of all the ratings of one user going into the same split, skewing the results. Since for the large dataset, the lowest possible number of ratings for a user is 1, which is not enough to partition and for the model and cross-validation to be meaningful, I filtered out the users with less than 20 ratings. All the splits were then converted from csv to parquet format for subsequent use (we kept both formats).

Additional pre-processing: all other data files were converted to parquet as well.

Before getting into q4-5, I have to note that most results and interpretations come from running the full dataset, except for one that was not able to run on large. For a log of all the results from running q4 and 5, please see: https://docs.google.com/document/d/1azDjprD8OVLHUiiIsneinl9QlxV-3mKu4QQO-6s_FvY/edit?usp=sharing

Q4. Documentation and evaluation of popularity baseline.

In recommendation systems, a baseline model can provide a reference point for evaluating the performance of more complex models. A popularity baseline ranks items (movies) based on their "popularity" which is measured by metrics like the average ratings or the number of interactions (ratings) they receive. The baseline model will recommend to a user the "most popular item" that the user has not previously interacted with using "wisdom of the crowds". This offers not only a benchmark for our recommendation model using collaborative filtering, but also is useful in addressing the cold start problem for recommending new users or new items where we have few interaction data.

There are many approaches to how we can approximate the "popularity" which we base on the ratings each movie has received in our database. In the following approaches we mostly

focused on comparing the results for ranking top 100 movies, but generalized our codebase to configure building baseline and evaluating on top N movies.

We tried five different approaches to operationalize popularity, which is documented below:

1. **'True' Popularity Baseline:** `4_baseline_all.py`

This approach operationalizes popularity as mainly the mean rating a movie receives, ranking them in descending order. If there happen to be movies with the same average ratings, they will be ranked based on the number of movies they have in descending order. The top N(100 is what we used) movies were returned and compared against the user's true preference via mean average precision (MAP) scores.

The simple approach offers a quick and straightforward solution but does not take into account many factors such as rating count imbalance, recency, etc. It established a baseline popularity model we could build upon and produced performance metrics that we could compare against when we attempted optimization later.

Results:

Small (need to use this later): Train MAP: 0.0019016393442622952, Validation MAP: 0.00042622950819672133, Test MAP: 0.0005081967213114754

Full: Train MAP: 8.119622584289998e-06, Validation MAP: 2.103275729665481e-06, Test MAP: 2.347842674975421e-06

(Technically only Test MAP is what we look at and what needs to be reported in the end, but I wanted to provide all the MAP scores for a clear comparison.)

Test MAP is extremely low, which confirms the idea that the baseline model fails to capture the multitude and complexity of user preferences.

2. **Weighted Composite:** `4_baseline_all.py`

Using the movie's average ratings is not enough, and the number of ratings a movie receives should also be considered. Otherwise, movies with a high average rating and very few ratings might rank higher than movies with average ratings a bit lower but substantially more ratings.

The script is a variation of the baseline model, where the code was slightly tweaked. We consulted Even's blog post about working on the exact same dataset we are using and transformed raw average ratings into weighted ratings according to the True Bayesian Estimate formula, which factors into both the average rating of the movie and the number of ratings for the movie. The formula is as follows:

$$(WR)=(v\div(v+m))\times R+(m\div(v+m))\times C$$

R = average rating for the movie
v = the number of votes cast for the movie
m = the minimum vote threshold required
C = the mean rating of **all** movies in the dataset

Results: Train MAP: 0.0965340461644507, Validation MAP: 0.020754880333393774, Test MAP: 0.021458646175217533

Test MAP is still relatively small but improved a lot compared to the baseline model, suggesting that both the movie average rating and the number of ratings play a role in representing user preference.

3. **Time Decay:** `4_time_decay_all.py`

Also a variation of the baseline model, where more recent ratings have a larger weight. The rationale is that user preferences change over time, and ratings from a long time ago are less indicative of the user's current preferences. Movies generally gain popularity when they first come out, but will soon fade into obscurity. This method prioritizes recent interactions and gives more weight to what is popular in the present.

Results: Decay Rate: 0.05 - Train MAP: 0.0683103848016269, Validation MAP: 0.014676609128216812, Test MAP: 0.01502956814368808

I experimented with a couple of decay rates, including 0.001, 0.005, 0.01, 0.02, and 0.05. I randomly picked one because they all returned very similar results. Test MAP has significantly increased from baseline but is slightly lower than the weighted composite model.

4. **Mean-centering/normalization:** `4_normalization_all.py`

Some users tend to rate on the higher end, and some might rate consistently on the lower end. To reduce user bias, we tried mean centering/normalization (i.e.subtracting the average rating from each user's rating). It helps to standardize the ratings across users and captures the users' relative preferences as opposed to absolute ratings. This is another variation of the baseline model. The only difference is that instead of ranking the movies by their average rating, movies here are ranked by the average normalized rating.

Results: Train MAP: 5.1359058515087335e-06, Validation MAP: 1.6630552281075897e-06, Test MAP: 1.6141418390456023e-06

Did not improve the baseline model. In fact, the test MAP was even lower. There could be a number of reasons why this has happened. If the dataset is very sparse, mean-centering might amplify this issue. This might also be a result of the cold start problem. For new users or items with very few ratings, mean-centering could do the opposite of improving performance. If users have diverse preferences, this method would not work very well. Moreover, if the rating distribution itself is not symmetrical in the first place, mean-centering is not a good choice either.

5. **Genre:** `4_genre_small.py; 4_genre_all.py`

I was experimenting with ways to incorporate the `movies.csv` dataset. The idea of this approach is to replace the average rating of movies with genre as a popularity metric. We calculated the average rating and number of ratings for each genre, recommended the top 100, and compared against users' explicit interactions with the genres (i.e. the genres they had rated on) to calculate the MAP.

Unfortunately, the code ran on the small dataset but not the full dataset. I used the exact same code but encountered a syntax error I was not able to resolve for the full data. Thus, I am reporting the results from running the small dataset.

Results: Train MAP: 0.135035950766085, Validation MAP: 0.38579761671814805, Test MAP: 0.37429792647594706

Test MAP is much higher than the baseline. In fact, this actually performed better than all the previous methods for the small dataset (you can find the results for all the other models on the small dataset in the log document I linked to earlier). It is a shame I could not get it to run on the full dataset, and it makes sense that users tend to stick to their favorite movie genres.

Q5. Documentation of latent factor model's hyper-parameters and validation and Evaluation of latent factor model.

1. **ALS (with fixed hyperparameters):** `5_als_all.py`

We first constructed a latent factor/collaborative filtering model with Spark ALS with fixed hyperparameters (no hyperparameter tuning) to test it out. The ALS setup looks like this:

```
def train_als_model(ratings):
    # Define ALS model
    als = ALS(
        maxIter=10,
        regParam=0.1,
        userCol="userId",
        itemCol="movieId",
        ratingCol="rating",
        coldStartStrategy="drop",
        nonnegative=True
    )
    # Train the model
    als_model = als.fit(ratings)
    return als_model
```

Results: Train MAP: 3.20871832246641e-05, Validation MAP: 6.945701246802289e-06, Test MAP: 7.483748526484156e-06

Unfortunately, the ALS collaborative filtering performed even worse than the baseline model. This could be attributed to a number of possible factors, including the cold start problem (new users or new items), data sparsity, etc, which all leave the model not enough data to work on. We speculated the most relevant reason might be the fact that we had not done hyperparameter tuning yet.

2. **ALS (with hyperparameter tuning)**

We then conducted ALS with hyperparameter tuning using the recommendation module of pyspark. We tried this two ways:

(At this point, we had given up on using dataproc because our job got killed every time we tried to run the scripts due to limited resources/memory available. We turned to a workaround where we ran our scripts with Jupyter Notebook on the Spark Standalone Cluster using Greene. It took over 4000 stages but ultimately ran successfully. That is why we have both a .py and a .ipynb file. For this step, we were mainly using parquet to speed up the processing as well.)

a. **Hyperparameter tuning with MAP:** `5_ht_map_all.py; 5_ht_map_all.ipynb`

We initially used MAP to tune the hyperparameters. Since there is no existing MAP evaluator available in pyspark.ml.evaluation, we had to create the MAP evaluator ourselves. The ALS setup looks like (please see the code file for how MAP was calculated, too lengthy to include here):

```
def train_als_model_with_tuning(ratings):
    # Define ALS model
    als = ALS(
        userCol="userId",
        itemCol="movieId",
        ratingCol="rating",
        coldStartStrategy="drop",
        nonnegative=True
    )

    # Define parameter grid
    param_grid = ParamGridBuilder() \
        .addGrid(als.rank, [10, 20, 30]) \
        .addGrid(als.regParam, [0.01, 0.1, 0.2]) \
        .build()

    # Define evaluator
    map_evaluator = MAPEvaluator(predictionCol="prediction", labelCol="movieId",
userCol="userId", k=10)

    # Define cross-validator
    cross_validator = CrossValidator(
        estimator=als,
        estimatorParamMaps=param_grid,
        evaluator=map_evaluator,
        numFolds=5,
```

```
        parallelism=2
    )

    # Train the model with cross-validation
    cv_model = cross_validator.fit(ratings)

    return cv_model.bestModel
```

Results:Train MAP: 1.2864221323302808e-05, Validation MAP: 2.9348033437192777e-06, Test MAP: 3.2771970671531937e-06

Test MAP is larger than the popularity baseline by a negligible margin and is even smaller than the ALS results without hyperparameter tuning. One possibility is that my parameter grid search setup is too limited/far from idea. Maybe I should have included more hyperparameters and a larger range of numbers to try. I would have tested out different grid search setups (in terms of range and step). However, running it turned out to be too costly, and there was not enough time to experiment with different hyperparameters.

b. **Hyperparameter tuning with RMSE:** `5_ht_rmse_all.py; 5_ht_rmse_all.ipynb`

Everything else remained the same, we only changed the ALS parameter tuning evaluator from MAP to RMSE. The ALS setup looks like:

```
def train_als_model_with_tuning(ratings):
    # Define ALS model
    als = ALS(
        userCol="userId",
        itemCol="movieId",
        ratingCol="rating",
        coldStartStrategy="drop",
        nonnegative=True
    )

    # Define parameter grid
    param_grid = ParamGridBuilder() \
        .addGrid(als.rank, [10, 20, 30]) \
        .addGrid(als.regParam, [0.01, 0.1, 0.2]) \
        .build()

    # Define evaluator
    evaluator = RegressionEvaluator(
        metricName="rmse",
        labelCol="rating",
        predictionCol="prediction"
```

```
)

# Define cross-validator
cross_validator = CrossValidator(
    estimator=als,
    estimatorParamMaps=param_grid,
    evaluator=evaluator,
    numFolds=5,
    parallelism=2

)
```

Results: Train MAP: 0.00345059503137794, Validation MAP: 0.0007368802062188479, Test MAP: 0.0007785544136996614

Tuning the hyperparameters with RMSE actually yielded better test MAP scores than using MAP for tuning. It performed better than the popularity baseline and ALS without hyperparameter tuning as well. It remains obscure to us why RMSE turned out to be a superior tuning method than MAP. Maybe it is because it is good for explicit feedback and has a differentiable loss function.

c. Incorporating Genome relevance: `5_genome_all.py; 5_genome_all.ipynb`

Circling back to the first ALS model with fixed parameters and no hyperparameter tuning, I attempted to incorporate the genome relevance data from the two genome files for the large dataset. Building upon the code from 5.1 on the rudimentary ALS model, `genome_scores.csv` and `genome_tags.csv` were aggregated to create a feature set for each movie, representing the relevance of each tag to every movie. The feature set is then joined with the ratings dataset, providing additional data beyond user ratings to build the ALS model and output more informed recommendations.

Unfortunately, the code did not run with Jupyter even on the spark standalone cluster due to the amount of memory it would require. I got an error saying `` `Py4JJavaError: An error occurred while calling o90.fit.: org.apache.spark.SparkException: Job aborted due to stage failure: Task 7 in stage 15.0 failed 1 times, most recent failure: Lost task 7.0 in stage 15.0 (TID 425) (cm041 executor driver): java.lang.OutOfMemoryError: Java heap space` ``. I did try changing the spark configuration to more nodes/cores/memory, nothing worked.

`