User-centric Systems for Data Science **Prof. John Liagouris**
Fall Semester 2020 liagos@bu.edu

Assignment 1

Submission deadline: October 2 2020, 11:59 pm

This assignment is inspired by the paper "Explaining Collaborative Filtering Recommendations" by J. L. Herlocker, J. A. Konstan, and J. Riedl (2000), which we discussed in Lecture 4. If you were not able to attend the lecture, make sure you watch the lecture recording before proceeding.

For this and future assignments, you will need to have Python 3.7+, Pytest, and Ray installed in your machine (cf. Section 10 "Resources" for more information).

You must follow the code skeleton provided in the Gitlab repository. Inline comments will help you identify the parts of the code you need to fill in. Keep in mind that the assignment does not require writing much code: the logic of each data operator can be implemented in less than 20 LOC. Always keep your code simple and well documented.

We will be using a mix of real and synthetic data. Real data include movie ratings from a large Netflix dataset whereas friendship relationships between users are synthetic. The input data are available in the Gitlab repository. Make sure you understand the data format first (cf. Section 1 "Data schema"). You might also want to create a toy dataset of the same format to test your code easily.

The queries of the first three tasks are given in SQL syntax only for reference and there is no need to do SQL parsing. The goal of the assignment is to implement the queries manually (as if you had them parsed), following a plan of your choice. We recommend that you employ some of the standard optimizations techniques we discussed in Lecture 1 (part I, part II).

1. Data schema

The data we will use for this assignment consist of two CSV files: `Friends` and `Ratings`. The former contains friendship relationships as tuples of the form `UserID1 UserID2`, denoting two users who are also friends. A user can have one or more friends and the friendship relationship is *symmetric*: if A is a friend of B, then B is also a friend of A and both tuples (`A B` and `B A`) are present in the file. `Ratings` contains user ratings as tuples of the form `UserID MovieID Rating`. For example, tuple `12 3 4` means that "the user with ID `12` gave `4` stars to the movie with ID `3`".

Hint #1: You can use Python's CSV reader to parse input files.

Hint #2: Consider encapsulating your Python tuples in `ATuple` objects (see code skeleton).

2. TASK I: Implement 'likeness' prediction query (credits: 50/100)

The first task is to implement a query that predicts how much a user will like a particular movie. The prediction in this case is based on friends' ratings: the 'likeness' of a movie M for a user A is equal to the average rating for M as given by A's friends. In SQL, this query can be expressed as follows:

```
SELECT AVG(R.Rating)
FROM Friends as F, Ratings as R
WHERE F.UID2 = R.UID
      AND F.UID1 = 'A' AND R.MID = 'M'
```

`Friends` and `Ratings` are both relations (tables): `Friends` stores friendship relationships whereas `Ratings` stores user ratings (cf. Section 1 "Data Schema").

The above query requires implementing five operators:

1. Scan: An operator that scans an input file (table) and returns its contents (tuples).
2. Project: An operator that implements a SQL projection.
3. Filter: An operator that filters individual tuples based on a user-defined predicate. This is equivalent to a selection predicate in SQL.
4. Join: An binary operator that applies a relational equality join.
5. AVG: An aggregation function that returns the average of the input values (e.g. ratings).

Hint #1: You might find useful drawing your query plan in the form of a tree, as discussed in Lecture 1. A node in the plan is an operator that corresponds to a Python class in the code skeleton we provide.

Hint #2: We highly recommend that you employ some of the standard optimization techniques we discussed in Lecture 1, e.g., pushing the selection (filter) down to the leaves of the query plan. Your solution will be graded solely based on correctness, however, we should be able to run tests with your code in a reasonable amount of time.

3. Task II: Implement recommendation query (credits: 20/100)

The second task is to implement a query that recommends a movie to a user: the movie with the higher 'likeness' value as computed by the query of TASK I. In SQL, this query can be expressed as follows:

```
SELECT R.MID
FROM ( SELECT R.MID, AVG(R.Rating) as score
       FROM Friends as F, Ratings as R
       WHERE F.UID2 = R.UID
             AND F.UID1 = 'A'
       GROUPBY R.MID
       ORDERBY score DESC
       LIMIT 1 )
```

The recommendation query requires implementing three additional operators:

1. GroupBy: An operator that groups tuples based on a 'key' attribute (e.g. movie id).
2. OrderBy: An operator that sorts tuples based on a value (e.g. sort candidate movies in descending order of their average rating as given by A's friends).
3. Limit: An operator that returns the first *k* tuples in its input (*k*=1 in the query above). An OrderBy followed by a limit is also known as a *top-k* operator.

Hint: You will need a custom comparator to sort tuples. Have a look at `cmp_to_key()` in `functools`.

4. TASK III: Implement explanation query (credits: 10/100)

The third task is to implement an ad-hoc query that 'explains' the recommendation provided by the query of TASK II. The explanation in this case amounts to a histogram of the ratings for movie M as given by A's friends. In SQL, this query can be expressed as follows:

```
SELECT HIST(R.Rating) as explanation
FROM Friends as F, Ratings as R
WHERE F.UID2 = R.UID
      AND F.UID1 = 'A' AND R.MID = 'M'
```

`HIST` is a user-defined function that generates the required histogram, i.e., the number of A's friends per rating score. The explanation query requires implementing one additional operator:

1. Hist: An operator that generates a histogram of the form *{rating: number of friends}*, where *rating* is a number in [0, 1, …, 5] and *number of friends* is the number of A's friends who gave the respective rating for movie M (A and M are given as inputs to the query).

5. TASK IV: Turn your data operators into Ray actors (credits: 20/100)

The fourth and final task is to turn the operators you have implemented so far into Ray actors, as discussed in Lecture 3. The resulting Ray program must return the exact same result for each task as before.

The Ray documentation contains all information you need to complete this task. Spend some time to become familiar with the concepts of tasks and actors.

Hint #1: Make sure you push your changes for this task in a separate branch 'ray' (cf. Section 8 "Git" for more information on git).

Hint #2: This task requires minimal changes to your code.

Hint #3: You can easily test your Ray program on a single machine (e.g. your laptop). In this case, Ray will run each actor in a separate local process.

6. Testing

You must have a simple test for each operator you implement and we strongly recommend using Pytest for that purpose. In case you are not familiar with Pytest, have a look at the code snippets provided in the documentation.

All test functions must be added to the separate `tests.py` file provided with the code skeleton. Before submitting your solution, make sure the command `pytest tests.py` runs all your test functions successfully.

7. Logging

Logging can save you hours when debugging your program, and you will find it extremely helpful as your codebase grows in size and complexity. Always use Python's logger to generate messages and avoid using `print()` for that purpose. Keep in mind that logging has some performance overhead even if set to INFO level.

8. Git

You will use git to maintain your codebase and submit your solutions. If you are not familiar with git, you might want to have a look here. Note that well-documented code is always easier to understand and grade, so please make sure your code is clean, readable, and has comments.

Make sure your git commits have meaningful messages. Messages like "*Commit*" or "*Fix*", etc. are pretty vague and must be avoided. Always try to briefly describe what each commit is about, e.g. "*Add Join operator*", "*Fix provenance tracking in AVG operator*", etc., so that you can easily search your git log if necessary.

Each time you finish a task (including the related tests), we strongly recommend that you use a commit message "Complete *Task X*". You will find these commits very helpful in case you want to rollback and create new branches in your repository.

9. Deliverables

Each submission must be marked with a commit message "*Submit Assignment X*" in git. If there are multiple submissions with the same message, we will only consider the last one before the deadline. In case all your submission commits are after the deadline, we will only consider the last commit, however, **late submissions will be eligible for up to 50% of the original grade**.

Your submission must contain:

1. The code you wrote to solve the assignment tasks (in `assignment.py`)
2. The code you wrote for testing (in `tests.py`)

Before submitting your solution, always make sure your code passes all tests successfully.

10. Resources

- Explaining Collaborative Filtering Recommendations: https://tinyurl.com/y8otpqnp
- Python tutorial: https://docs.python.org/3/tutorial/
- Python logger: https://docs.python.org/3/library/logging.html
- Pytest: https://docs.pytest.org/en/stable/
- Ray documentation: https://docs.ray.io/en/latest/
- Git Handbook: https://guides.github.com/introduction/git-handbook/