Software Engineering 265 Software Development Methods Fall 2019

Assignment 3 Due: Tuesday, November 19, 9:00 pm by submission via git (no late submissions accepted)

Objectives of this assignment

- Use the Python programming language to write your implementation of basic OLAP (Online Analytical Processing) queries from first principles, in a file named “OLAP.py”
- Test your code against the provided test data, and other test data you create, to ensure it’s working as expected. The instructor will also be testing your program against test data that is not provided to you. Think creatively about how to “break” and strengthen your code!

This assignment: “OLAP.py”

In this assignment, we will implement OLAP queries from first principles. There are numerous libraries that perform these functions, however, learning how to do this from first principles will help to illustrate a few key benefits:

- We can better understand the library implementations and their usage
- We can demystify some of what goes into library development
- We can get hands-on with some of Python’s strong abilities in processing big data

To this extent, we will write code that will calculate the **`min`**, **`max`**, **`mean`**, **`sum`** and **`count`** of numeric columns and the **`group-by`** along with **`top-k`** for categorical columns.

We’ll take as our input a set of time-stamped events (these will be supplied in a CSV file) - and we’ll generate some summary calculations for the whole file. At present, we seem to find ourselves in a “golden era” of data science, so perhaps some of these techniques may prove useful to you.

Background Information

Data Science

A data scientist is someone who applies a combination of insights from computer science and statistics to solve problems relating to information (a.k.a data) -- in particular, the extraction of insights from complex data. Data scientists use numerous libraries, frameworks, modules and toolkits to efficiently implement the most common data science algorithms and techniques.

When you're approaching data science with the Python programming language, some of the useful tools you're likely to encounter include **`NumPy`** and **`pandas`**[1] - they're well-acknowledged for the aid they provide in data science tasks, and abstract away much of the complexity in their underlying implementations. However, by abstracting away said complexity, on the surface that definitely supports bringing more people into the world of data science, but it also shields us from acquiring deeper understanding.

If you are trying to implement your own versions of some well-known functions, you may not end up with the "optimal" implementation (e.g. your implementation may not be able to handle an "astounding level of data"), but you'll definitely advance your understanding as a practitioner... and who knows, having the knowledge on hand could potentially help during a job interview.

In this assignment, we'll *implement some well-known functions from first principles*. The functions we will use in this assignment will be defined in the following sections.

Additional Notes

- It's possible to produce all of the output required for this assignment in *a single traversal of the input file*, i.e. reading each input record only once. This may not sound so important when you're dealing with a few thousand records, but if the big data is really big, or even infinite (in the case of streaming data) it becomes really important to be efficient.
- A ***numeric field*** is one that contains values intended to be interpreted as numbers. They may be integers, floating-point numbers, missing values, or a combination
- A ***categorical field*** is one that contains values *not* intended to be interpreted as numbers, even though they sometimes consist of nothing but digits (e.g. UNIX timestamps; auto-incrementing primary keys in a database table)
- *NaN* means "Not a Number" -- a valid floating point value that is not zero and not infinite, but also... not a number
- A *CSV file* is a plain-text file that has "comma-separated values" - while a CSV file can be read by any application, it consists of contents that are well-supported through spreadsheet applications like Microsoft Excel
- Example CSV file:
    - foo.csv
        - `timestamp,colour,sound`
        - `2019-11-03T03:45:56.000Z,red,moo`
        - `2019-11-04T01:15:78.033Z,green,oink`
- Some CSV dialects use `null` or "" to represent missing values; others just repeat the delimiter, e.g. in the following example, Bob does not have a pet:

[1] these libraries, along with others that address this problem space, are also **strictly prohibited** for use on this assignment

```
name,pet_type,age
Alice,Angora Rabbit,11
Bob,,13
Clara,Domestic Shorthair Cat,15
```

- The first line of the CSV file will be a standard *CSV header line*, in which the **`field names`** are listed. Each subsequent line will contain values in those fields. Lines are separated by line feeds (ASCII code #10, aka Unix "new line" characters, aka \n -- CSV files from Windows systems will often use two characters between lines, CRLF (Carriage Return, Line Feed) -- you can use dos2unix to strip out the carriage returns if you encounter one. (Reminder: your program MUST run in the Lab environment)
- **Aggregate functions** are mathematical functions that take a (potentially very large) number of input values, and produce a single output value. (As an aside to the current assignment: In SQL[2], aggregate functions can appear as "projections", e.g. SELECT min(age), max(age) FROM KidsWithPets)
- Commonly used aggregate functions have different memory requirements:
    - constant space (e.g. sum, min, max, mean)
    - O(n) in the number of input values (e.g. accurate median)
    - O(n) in the number of distinct values observed (e.g. top-k, count distinct)

Command line specification (your program must run in the following form)

python OLAP.py --input <input-file-name> [aggregate args] [--group-by <fieldname>]

Arguments (which may occur in any order)

--input <input-file-name> (e.g. --input input.csv)
name of the input file to be processed. The input file must contain a header line.

aggregate args (see [Aggregate Functions] below for a list of supported functions):
specifies the aggregate functions to calculate on the file. Any number of aggregates (zero or more) can be specified. If no aggregates are specified, default to --count. If one of the aggregate function arguments references a nonexistent field, program exits with error code 8, prints on `stderr`, `Error: <input_file>:no field with name '<missing_field>' found`

--group-by <categorical-field-name>
program computes the requested aggregates independently for each distinct value in <categorical-field-name>. If there are more than 1000 distinct values in categorical-field-name, the program will use the first 1000 values found, and message on stderr to indicate partial output, i.e. `Error: <inputfile>: <categorical_field> has been capped at 1000 values`

[2] SQL a.k.a. the "structured query language", highly popular in the management of relational database management systems (RDBMS)

Aggregate Functions

argument	description	output format
--top <k> <categorical-field-name>	compute the `top k` most common values of categorical-field-name	a string listing the top values with their counts, in descending order, e.g. “red: 456, green: 345, blue: 234” If the values contain double quote or newline characters, they should be escaped using the usual printf syntax, e.g. “\n” for newlines, \” for double quotes
--min <numeric-field-name>	compute the `minimum` value of numeric-field-name	a floating point number, or “NaN” if there were no numeric values in the column
--max <numeric-field-name>	compute the `maximum` value of numeric-field-name	a floating point number, or “NaN” if there were no numeric values in the column
--mean <numeric-field-name>	compute the `mean` (average) of numeric-field-name	a floating point number, or “NaN” if there were no numeric values in the column
--sum <numeric-field-name>	compute the `sum` of numeric-field-name	a floating point number, or “NaN” if there were no numeric values in the column
--count	`count` the number of records	an integer, or zero if there were no records

Group By

Add optional named argument for a `group-by`

```
--group-by categorical-field-name
```

If a `group-by` field is specified:

- If it’s a valid categorical field, produce an output CSV file with one row of output per distinct value in that field, with:
  - the value of the group-by field in the first column
  - the values of any computed aggregates in subsequent columns, in the order they were specified at the command line (or just a count column, if no aggregates were requested at the command line)
- If input CSV file does not contain a field with a name matching `--group-by` argument, exit with error code 9, and print on `stderr`, `Error: <input_file>:no group-by argument with name ‘<group-by_argument>’ found`
- If the field is not a categorical field, or has very high cardinality[3], or is sparse, use your judgment on what to return.

[3] Cardinality is a common idea in big data - high cardinality refers to situations in which data has numerous unique values (low cardinality —> data has few unique values); when we have millions of distinct values, dynamic memory becomes crucial

If no `--group-by` argument is specified, then produce the requested aggregates for the whole file in a single-line (plus header) CSV output file with no extra column for a group-by field.

Group-By Example Output

The query is "`--group-by colour --count --min foo`"

The results indicate:

- There are four distinct values in the `colour` field
- The number of records with `colour == red` is 123, etc.
- The minimum value of `foo` for all records with `colour == red` is 3, etc.

colour	**count**	**min_foo**
red	123	3
green	234	4
blue	345	5
violet	456	6

Requirements

Normal Operation

1. Check that the CSV file provided in the first command-line argument exists and is readable

2. Validate the command line arguments

3. Check that the requested computations are valid, given the fields declared in the input file's header

4. Read the file, during which you should:
    a. keep track of which line you're reading, so you can provide better error messages.
    b. compute the aggregates requested (or just --count if none were requested)
    c. split the output by the group-by field if requested, or produce one row if no `group-by` was requested.

5. Output should be in CSV format on stdout

a. you can use `python OLAP.py --count > somefile.csv`, for example, to capture your program's output to a file so it may be *compared*

6. `count, min, max` and `mean` aggregate results should be printed as numbers
7. `top-k` aggregate results should be printed in one column, as strings with the following format, in descending order of count:

   `(value`$_1$`: count`$_1$`)[, (value`$_n$`: count`$_n$`)...]`

   e.g.
   `blue: 456, green: 345, red: 234`

8. Output fields should be named according to the aggregate function and original field name, separated by an underscore, e.g. "`min_foo`", "`avg_bar`", "`top_colour`"

9. The first row in the output CSV file must be a header row, with each column in the header row listing named accordingly (see examples, which include header rows, to better understand the naming in header)

Error Conditions — all errors must be displayed on `stderr`, each on its own line (this permits us to direct the successful output of a program to a file, but still prints error messages to the screen - this distinction is vitally important and will be graded heavily, up to 30%)

1. If an aggregate is requested on a field that contains non-numeric values, skip that value, and print an error message to stderr (not stdout!) with the following format[4]:

   `Error: <inputfile>:<lineNumber>: can't compute <aggregate> on non-numeric value '<value>'`

   e.g.
   `Error: input.csv:123: can't compute mean on non-numeric value 'asdf'`

2. If more than 100 non-numeric values are found in any one aggregate column, exit with error code 7, and print (on stderr) a message, `Error: <input_file>:more than 100 non-numeric values found in aggregate column '<aggregate_field>'`

3. If a `top-k` aggregate is requested on a field with more than 20 distinct values:
   - print a message on `stderr` (not `stdout`!) that the field has been capped at 20 distinct values, i.e. `Error: <inputfile>: <aggregate_field> has been capped at 20 distinct values`
   - add "_capped" to the field name, e.g. "top_colour_capped"

[4] format provided so you may see how to substitute into any tagged value, e.g. `<inputfile>` in the example given

4. If a `group-by` field has more than 20 distinct values:
    - print a message on `stderr` (not `stdout`!) that the field has been capped at 20 distinct values, i.e. `Error: <inputfile>: <group_by_field> has been capped at 20 distinct values`
    - assign the overflow records to an overflow row named "_OTHER" so the aggregates from the overflow records still get counted. The _OTHER row should be printed last.

5. Should you detect any other error that prevents the program from producing meaningful output, exit with error code 6 and a custom error message of the form: `Error: <input_file>:<meaningful_error_message>`, where the tagged message is replaced with your custom message indicating what went wrong in an appropriate level of detail.

Example to guide your effort

"Almost the Kitchen sink" example (limited to only four companies to save space in this document, actual output would list all)

```
python OLAP.py --input input.csv --group-by ticker --count --min open --max open --mean open --min high --max high --mean high --min low --max low --mean close --min close --max close --mean close
```

ticker	count	min_open	max_open	mean_open	min_high	max_high	mean_high	min_low	max_low	mean_low	min_close	max_close	mean_close
aapl	8364	0.23305	175.11	22.2843502	0.23564	175.61	22.4958666	0.23051	174.27	22.281018	0.23051	175.61	22.281018
adbe	7876	0.2	182.88	26.5608488	0.2	184.44	26.9107583	0.2	181.06	26.5774755	0.2	184.06	26.5774755
amzn	5153	1.41	1126.1	181.747357	1.45	1135.54	183.880652	1.31	1124.06	181.769343	1.4	1132.88	181.769343
axp	11556	0.49838	96.42	22.7530326	0.5074	96.73	23.0014514	0.48981	96.29	22.7552431	0.48981	96.43	22.7552431

...

Provided Materials

We have created a git repository for you - it contains a starter file, OLAP.py, your data set, a limited test suite, and these assignment instructions. There are also is also a hints section in this document.

OLAP.py

The starter file for this assignment, and where you are instructed to place your implementation.

Data Set

We have supplied a single test data file in your root directory. The dataset for this assignment (`input.csv`) contains ticker info for twenty large companies (across three sectors): Apple (`aapl`), Adobe (`adbe`), Amazon (`amzn`), American Express (`axp`), Ali Baba (`baba`), Salesforce (`crm`), Cisco (`csco`), Disney (`dis`), Facebook (`fb`), Google (`googl`), IBM (`ibm`), Intel (`intc`), Mastercard (`ma`), Microsoft (`msft`), Netflix (`nflx`), Nike (`nke`), Nvidia (`nvda`), Oracle (`orcl`), Starbucks (`sbux`) and Visa (`v`).

The data consists of nine columns, `Sector`, `Ticker`, `Date, Open, High, Low, Close, Volume, and OpenInt`. The full dataset may be found here[5]. You can create your own .csv files, with your own data; once you have your implementation completing with the provided data, please feel free to test it with larger and different data sets as you see fit.

Libraries (permitted usage list, if library is not on this list then it may not be used)

+ `csv` (for parsing the input file, and for creating the output file as well)
+ `argparse` (for parsing parameters)
+ `os`
+ `sys`

Test Suite

We will supply a limited suite of test cases that you can use to evaluate your own program, and get a feel for the types of inputs you can expect. The test suite has the same structure as previous assignments - a set of directories, with one test case provided per directory. The tests shall run similarly to previous assignments. We also provide a file, command.log, to illustrate test case creation.

Hints (we may add more to a course announcement page at a later time)

- while calculating the aggregate of a set of data, watch for situations in which there is no data to calculate

Process

To get started, the assignment repository has already been set up in your person GitLab space, listed as /assignment3.git. We set this up in our local machine space with **`git clone`** (all one line, with <NETLINK_ID> replaced with your personal Netlink identifier):

---

[5] Huge Stock Market Dataset, kaggle.com, accessed November 4, 2019

`git clone https://gitlab.csc.uvic.ca/courses/2019091/SENG265/assignments/<NETLINK_ID>/assignment3.git`

To ensure you and the teaching team can see your code, `git push` to the same personal repository you cloned from GitLab!

Use your favourite text editor to write your program. Once you're able to compile your program, test it by running it with the **`input.csv`** files in the provided test suite, and any other files you can think of. Be creative; try to predict what kinds of inputs will break your code! Try random files lying around on your computer or the internet to see what happens!

When you've made changes to the file that you want to keep, **`git commit`** your changes to your local copy of the repository. Git will save the history of past commits. When you want to upload your changes to your repository, use **`git push`**.

Deliverables

- Write a Python program in a file named OLAP.py (case sensitive), placed in the root of your assignment3 repository, that implements the above requirements
- The program must run successfully using **python** (within the lab environment, which runs Python 3.7)
- We will run the script (possibly with aggregate args (and other parameters) in many different orders) with:
  - **`$ python OLAP.py --input input.csv [aggregate args] [--group-by <fieldname>]`**
  - where input_file.csv are the supplied ticker data
- You may only use use python3 libraries/modules specified in this assignment, if not specified, you may not use it library
- You may assume that all test files will be in the same CSV format provided with the assignment
- The assignment will be marked based on the last version of your program pushed before the deadline

Evaluation

Compilation

(10% of the mark, if your program compiles at all)

Correctness

(70% of the mark, based on the proportion of provided and secret test cases that your program successfully passes)
The teaching team will run your code using the provided test suite to verify whether your program meets the requirements:

- When run with correct parameters and given input in the expected format, it produces the expected output
- When run with incorrect parameters or badly formatted input data, it produces the expected error messages and the expected exit codes

We will also evaluate your program using ***an additional test suite that is not provided to you***, to ensure that your solution isn't over-fit to the provided test suite.

Style (20% of the mark, based on the subjective assessment of the teaching team - clean, original, elegant, fit to purpose)

A few examples of bad programming style:

- **inconsistent formatting**, indentation, etc. -- pick a style and use it consistently
- **misleading or non-descriptive names** for variables and functions
- **insufficient whitespace** impairing readability
- long sections of **repetitive code** (sometimes it's better to repeat short sections than introduce an abstraction!)
- **"dead" code**, i.e. present in the file but never called
- **complex code without explanatory comments** (don't go overboard; simple code often doesn't need comments - however, your functions do!)

Programming environment

For this assignment, please ensure your work compiles and executes correctly on the Linux machines in ELW B238. You are welcome to use your own laptops and desktops for much of your programming; if you do this, **give yourself a few days before the due date to iron out any bugs in the Python program you have uploaded to the BSEng machines**. (Bugs in this kind of programming tend to be platform specific, and something that works perfectly at home may end up crashing on a different hardware configuration.)

Individual work

This assignment is to be completed by each individual student (i.e., no group work). Naturally you will want to discuss aspects of the problem with fellow students, and such discussion is encouraged. However, sharing of code fragments is strictly forbidden without the express written permission of the course instructor (Murray). If you are still unsure regarding what is permitted or have other questions about what constitutes appropriate collaboration, please contact me as soon as possible. (Code-similarity analysis tools will be used to examine submitted programs.)