README.md

wikitop25

Application that computes the top 25 pages for each sub-domain in Wikipedia for a given date and hour.

Instructions on how to use it: see section 2.1.

Author: João P C Bertoldo

github: github.com/joaopcbertoldo

linkedin: linkedin.com/in/joaopcbertoldo

1) Subject

Build a simple application that computes the top 25 pages on Wikipedia for each of the Wikipedia sub-domains.

1.1) Requirements

1. Accept input parameters for the date and hour of data to analyze (default to the current date/hour if not passed).
2. Download the page view counts from wikipedia for the given date/hour [...] from https://dumps.wikimedia.org/other/pageviews/.
3. Eliminate any pages found in this blacklist: https://s3.amazonaws.com/dd-interview-data/data_engineer/wikipedia/blacklist_domains_and_pages
4. Compute the top 25 articles for the given day and hour by total pageviews for each unique domain in the remaining data.
5. Save the results to a file, either locally or on S3, sorted by domain and number of pageviews for easy perusal.
6. Only run these steps if necessary; that is, not rerun if the work has already been done for the given day and hour.
7. Be capable of being run for a range of dates and hours.

1.2) Questions

1. What additional things would you want to operate this application in a production setting?
2. What might change about your solution if this application needed to run automatically for each hour of the day?
3. How would you test this application?
4. How you'd improve on this application design?
5. You can write your own workflow or use existing libraries (Luigi, Drake, etc).

2) Application

2.1) Instructions to use it

1. Download the black list here and place the file in `src/black_list/` .
2. Install all the requirements (cf. `requirements.txt`).
3. (1) Start luigi daemon (luigi's remote scheduler):

```
$  luigid
```

3. (2) Otherwise go to `src/configs.py` , go to the class `Options` and change `use_local_scheduler` 's value to `False` .
4. Call the cript run.py with with `-h` to see usage and input format.

```
$ python run.py -h
```

5. Run it with the wished command/inputs.
6. Find results in `ranks/` .

2.2) Design

The app was built using the luigi framework.

The script run.py validates the inputs and provides command line interface

There are two commands:

- `single` runs it for only one date-hour - default is the hour before the current hour (as the current might not yet be available)
- `range` : run it for each date-hour between between the beginnig and end (included) - no default

Results are saved in the `wikitop25/ranks` folder as a .json (text file encoded with utf-8).

2.2.1) Considerations and suppositions

Things marked with * can be altered in `src/configs`

1. the **Main_Page** is excluded from the ranking *
2. a missing page name in the black list file is considered as if the whole domain is black listed
3. results and intermidiate files are stored locally in the project's folder *
4. ranks are 0-indexed (from 0 to 24)
5. luigi's scheduler is not local by default *
6. tasks are paremetered by datetime objects (date-hour for luigi)
7. pages with equal number of pageviews are kept in the rank at the same position
8. ordering is done by pushing items into the rank and puting it in the correct position

2.2.2) Workflow design

1. Tasks dependency: `DonwloadTask --> ComputeRankTask --> SaveRankTask --> CleanUpTask`
2. Targets:
   - `DonwloadTask --> LocalTarget (.txt)`
   - `ComputeRankTask --> LocalTarget (binary from pickle)`
   - `SaveRankTask --> LocalTarget (.json)`
   - `CleanUpTask --> nothing`

2.2.3) General procedure

1. Inputs are interpreted and validated (by the end, the important arguments are `command` and the range of datetime objects that specify the date-hours to process).

   when executed for ther 1st time: -> create necessary folders (temp, ranks, ...)

2. For each input (datetime), check if its rank has already been computed.

If yes, skip it. If not, create a Task to do it.

Send the tasks to luigi's scheduler.

3. Download the .gz file from wikimedia, decompress it, decode it and write everything in a .txt file.
4. Build a Rank object for each domain then, reading line by line, get the domain, page name and pageviews and push page/pageviews in the correct Rank (see the section '**Rank logic**').
5. For each Rank, filter the pages by eliminating those in the black list.

   ****when loading the black list for the first time*** - read the `.txt` file - process it (creating a dict) - save the dict in a pickle file (to be used later)

   IMPORTANT -- The filtering is done after the rank is built, so it might end up shorter than it should be because of eliminated items. Therefore the ranks are created with a bigger size to compensate it (hopping it will be big enough) and later shortened. It is way faster to do this, because the ranks have much less pages than the original list of pages.

6. Shorten the ranks to the correct size (25)
7. Save it in a json file (see the section '**JSON structure**').

2.2.4) Rank logic

2.2.5) JSON structure

Key-Value:

`ranked_pos` is the position in the rank (from 0 to 24)

`score` is the number of page views

`values` is a list of strings with the names of the pages in that rank position

```
{
    domain1: [
        {
            ranked_pos: int,
            score: int,
            values: [page1, page2, ..., pageN]
        },
        {...},
        ...
        {...}
    ],
    domain2: [...],
    ...
    domainN: [...]
}
```

2.3) Answers

1. **Functionalities to add**:

    - add control actions to perform on tasks (as cancel, pause, resume, ...)
    - better control/use of parallelization
    - progressbar in tasks
    - log of activities
    - measure time of the tasks (like a decorator on the run functions)
    - an option to run it automaticaly every hour
    - alerts for completions/errors (possibly to trigger other stuff)
    - option to move/export the results to somewhere else
    - store the results in a database
    - edit the black list
    - option to take a customized black list (from folder or url)
    - enable several outputs formats
    - divide the results in two parts (and tasks as well) because there are several domains a lot less voluminous and less important than others, so I'd separate those that are rather more important and give them priority, which would improve performance

2. **Changes to run automatically**:

It should be relatively easy, I would create an infinite routine that would schedule new tasks every hour by calling the `main` .

It would be interesting to create control commands to operate such routine.

For safety, I would ensure that there is a proper way to pause/resume/stop the app, which could be necessary for versioning it.

The logging/alert functionalities would become particularly important to keep basic track of what happens.

3. **Testing the app** I would create unity tests for the different encapsulated parts of the app.

Overall tests that I would implement*:

- DownloadTask
    - tests related to internet connection/http errors (as 404, for example)
    - check proper behavior in case of corrupted files
- ComputeRankTask
    - test behavior cases of missing values

    - (ideally) run several examples with the black list filtering before computing the rank (which garantees a correct rank) and compare it to the used strategy --> this should estimate how often this strategy could be inconsistent
- SaveTask
    - test that the result json is correctly loaded
- BlackList
    - test the loading of the txt and pickle file
    - test behavior in case of missing values
    - check the `has()` and `doesnt_have()` functions with known examples
- `rank.py` module
    - `RankItem` : test logic with scores and manament of inclusion/deletion of a content
    - `Rank` : nsure that the logic of the algorithm works well (mainly in cases that items are eliminated during the filtering)

* some of these are already done in the modules them selves when run as _*main*_

4. **Design changes**

- remove the input validations from `run()` and place it in a new module that would be called by `main()`

  Reason: this would make the validations reusable in case that another interface or script would trigger the app

- decompose the `CleanUpTask` into a clean up for each task. This could be done with a task that takes another task as parameter and creates the dependency by itself or with an abstract task.

  Reason: This would reduce disk usage by removing temporary stuff as soon as possible.

- (according to performance) change the black list implementation to

    - read from a stream (thus not needing to load it completely each time)
    - store it in a database
    - use regex to check if a page is in it

- (if the black list filtering improves enough in speed) do the filtering before creating the rank

  Reason: this would rather ensure the consistency of the ranks

- in rank, rather use an ordered list

  Reason: which could simplify the implementation to manage the ordering

- build the rank directly with the correct size then, when doing the filtering, only in case that the rank gets shorter, go back to the input and get a new item

  Reason: this could improve performance by consulting the black list fewer times and would ensure consistency.

- (if more options/commands are added) manage the parser creation in a separate package with individual/grouped parsers per module

  Reason: better organized and better reuse of code/text

- encapsulate the filtering function in some sort of callable that could be an option when building the task/workflow

  Reason: more flexible app

- for the output formatation, encapsulate the behavior and make it changeable (as with a callable)

  Reason: easier to personalize the output as needed

5. **Framework** luigi was used (see **Workflow design** in section **2.2.2**).