Inverted Index (II)

Build Inverted Index upon wet file

Table of contents

- Features
- How to run
  - Requirements
  - Installation
    - Recommand for running
      - Virtual env
- For the first time:
  - Usage
    - Example usage
      - Download wet files
      - Lexicon extraction
      - Sort Merging
    - Actual usage
    - More options
  - Notes on Running on Servers
    - HPC
      - Load Python 3 module
- Benchmark
  - Speed
    - Full mode
    - No Chinese mode
    - Dumb mode
    - Merging and Sorting
  - Size
  - Memory

- How it works
  - Questions
- Future Work
  - Distributed
  - Speed up
  - Query optimization
- Development

Features

- Language detection
- Chinese support along with latin-character based languages
- Binary I/O and Stroage
- Progress & Speed display
- Text Frequncy (TF) in documents

How to run

Requirements

- Linux/ macOS/ OS with GNU tools
- Python 3.4+

Installation

Please consider the Recommand for running section before Installation.

If you insist to install directly, It will be okey.

```
$ pip install -r requirements.txt
```

Recommand for running

It is recommanded to use virtual environment for python packages to avoid package conflicts.

Virtual env

For the first time of for this project, start a new venv from as:

```
$ pyvenv .env
```

And then or for later use, activate it:

```
$ source .env/bin/activate
# For the first time:
$ pip install -r requirements.txt
```

Usage

The running of the whole inverted index building has been devided into 3 parts:

- Download wet files
- Lexicon extraction
- Sort Merging

For the first **Lexicon extraction** stage, use python script `extract_lex`, and for **Sort Merging** stage use `merge.py`

Example usage

Download wet files

```
$ ./scripts/dl.sh 100
```

This will download `100` wet files to `data/wet`. (change `100` to get more or less)

Lexicon extraction

```
$ python extract_lex.py --urlTable "data/url-table.tsv"
data/wet/*.warc.wet.gz | sort > "data/all.lex"
```

This will extract all lexicons (that in language English, French, Germany, Italian, Latin, Spanish and Chinese) from the `wet` files in `data/wet/`, and write the sorted lexicons to `data/all.lex`.

Sort Merging

```
$ cat  "data/all.lex" | python merge.py > "data/inverted-index.ii"
```

This will read all **sorted** lexicons, merge them into inverted lists and write to `data/inverted-index.ii`.

Actual usage

Example usage is not practical when you wants to:

- Run on many wet files
- Use binary for performance boost

So there are smarter version provided for these needs:

```
 # extract all wet files in `data/wet`
$ ./scripts/extract-all.sh
* Dealing: data/wet/CC-MAIN-20170919112242-20170919132242-
00000.warc.wet.gz
Building prefix dict from the default dictionary ...
Loading model from cache
/var/folders/dy/dh2zyqj93fg72s9z4w2tnwy00000gn/T/jieba.cache
Loading model cost 0.828 seconds.
Prefix dict has been built succesfully.
40919records [05:00, 136.23records/s]
...
$ ./scripts/merge.sh
```

`extract-all.sh` will individually extract and sort lexicons into fex files to `data/lex`.

`merge.sh` will take all **sorted** lex files and merge them into the final II file `data/inverted-index.ii`.

All oprations are done in binary.

More options

More options over `extract_lex.py` can be fetched help:

```
$ python extract_lex.py -h
usage: extract_lex.py [-h] [-b] [-s <number>] [--skipChinese] [-T
<filepath>]
                      [--bufferSize <number>] [-u] [-c]
                      <filepath> [<filepath> ...]

Extract Lexicons from WETs

positional arguments:
  <filepath>            path to file

optional arguments:
  -h, --help            show this help message and exit
  -b, --binary          output docID as binary form
  -s <number>, --startID <number>
                        docID Assigment starting after ID
  --skipChinese         if set, will not parse chinese words
  -T <filepath>, --urlTable <filepath>
                        if set, will append urlTable to file
  --bufferSize <number>
                        Buffer Size for URL Table Writing
  -u, --uuid            use UUID/ if not specified, use assign new ID
mode
  -c, --compressuuid    compress UUID in a compact form, only valid in
UUID
                        mode
```

Note that `uuid` isn't tested for use. It was built for compatibity of ditributed system.

Notes on Running on Servers

Scripts are created for copying necessary exectables to server. Use of example:

```
$ ./scripts/deploy.sh user@server:path
```

HPC

Distributed version of this II building program is not completed, you will not be able to use it on Hadoop or Spark or Hive. However you could use HPC as ordinary server to run the program.

There were works done for prepration of this program to be distributable. Please read Future Work > Distributed section.

Load Python 3 module

```
$ module load python
```

Benchmark

The following tests are done using Macbook Pro 2016 Laptop

Speed

Sorting and merging speed are significently low compared to `lexicon extraction`.

So the testing are mostly about `lexicon extraction`.

Full mode

(Language detect on, Chinese on, binary)

```
$ python extract_lex.py --binary data/wet/* > "data/delete-this.log"
```

~ 136 records/s

~ 5 mins/wet

No Chinese mode

(Language detect on, Chinese off, binary)

```
$ python extract_lex.py --binary --skipChinese data/wet/* >
"data/delete-this.log"
```

~ 166 records/s

~ 4 mins/wet

Dumb mode

(Language detect off, Chinese off, binary)

```
$ python extract_lex.no_language.py --binary data/wet/* > "data/delete-this.log"
```

~ 513 records/s

~ 1.3 min/wet

Speed is significantly faster however in this mode search result is going to be farily bad, beacuse all languages are jammed together. And for non-latin language it's even un-searchable.

Merging and Sorting

```
$ ./scripts/merge.sh
```

~ $530k$ lines/s (for input)

~ $52k$ inverted lists/s (for output)

~ 13 s/wet (including Chinese words)

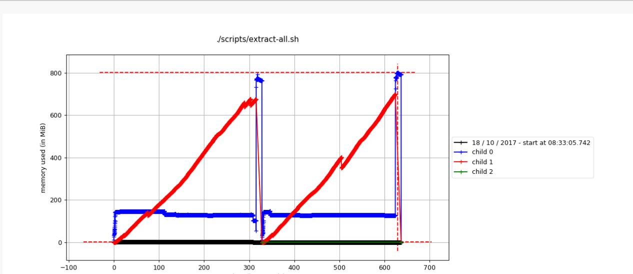
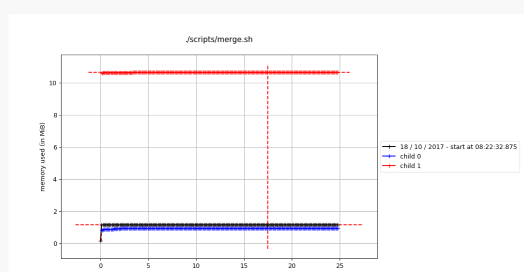
Size

~ $15k$ inverted lists/MB

```
 #count line using:
$ cat "data/inverted-index.ii" | wc -l
```

Memory

Depends on buffer size, for default

Lexicon Extraction (and sort)	Merge
<1GiB	< 15MiB
	

The memory usage are mostly used by GNU Unix Sort, by default GNU Unix Sort would take 80% of system, after that sort would use temprary file to store them.

Luckily mordern computers has a memory typically much greater than 1GiB. So as long as the wet file size maintain as the current scale, this would't be a problem.

(In other words, on very low memory computers, it might slow down.)

How it works

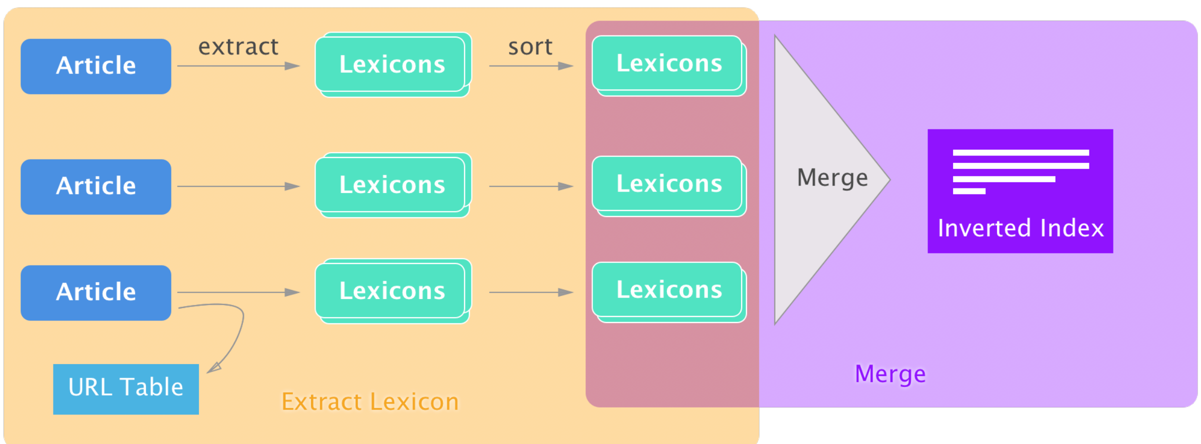


Figure of what `Extract Lexicon` and `Merge` do.

File Structure

```
  ├── README.md  # This file source code
  ├── README.pdf # This file
  ├── data/
  |    ├── inverted-index.ii # (generated) final Inverted List
  |    ├── lex/              # (generated) intermediate lexicons Files
  |    |    ├── CC-MAIN-20170919112242-20170919132242-00000.lex
  |    |    └── ...
  |    ├── url-table-table.tsv # (generated) Index of URL Table
  |    ├── url-table.tsv       # (generated) URL Table
  |    ├── wet/                # (downloaded) WET Files
  |    |    ├── CC-MAIN-20170919112242-20170919132242-00000.warc.wet.gz
  |    |    └── ...
  |    └── wet.paths           # WET Files URLs
  ├── decode-test.py           # Test code for lexicon file verification
  ├── extract_lex.no_language.py # Dumb mode version of `extract_lex.py`
  ├── extract_lex.py           # main file for `Lexicon Extraction`
  ├── merge.py                 # main file for `Merge`
  ├── miscellaneous/           # miscellaneous files
  |    ├── ...
  |    ├── deprecated/
  |    |    ├── extract.sh*
  |    |    ├── hadoop-test.sh*
  |    |    ├── map-reduce-test.sh*
  |    |    └── setup-env.sh*
  |    ├── dumbo-sample.sh*
  |    └── testbeds/
  |         └── ...
  ├── modules/                 # Modules
  |    └── NumberGenerator.py  # binary compatible docID generator
  ├── requirements.txt         # requirement for python dependencies
  └── scripts/
       ├── deploy.sh*          # helper script for deploy
       ├── dl.sh*              # helper script for download
       ├── extract-all.sh*     # main script for `Lexicon Extraction`
       ├── generate_toc.rb*    # helper script for markdown ToC
generation
       └── merge.sh*           # main script for `Merge`
```

Questions

> Why so slow?

Because Language Detect and Chinese Word Seperate uses HMM model (pre-trained) to compute. They are computational intensive.

> How much docIDs are supported? Why?

In short: ~2 billion.

The binary encoding entroy used for docID generation was $(256 - 3) = 253$ out of 256 per byte.

The number of encoding bytes are chosen for 4 as default so there would be $(256 - 3)^3 \times (128 - 3) - 2 = 2,024,284,623 \approx 2$ **billion** documents supported.

> Why -3 ?

The -3 was for `\t` `space` and `\n` 3 different kinds of seperation characters. Those characters are used to seperate words document IDs frequency and future added features.

Compare with plain text number entroy 10 out of $2^8 = 256$, $2^8 - 3$ is a much better figure.

> Why unix sort?

Unix sort is incredibaly fast and supports streaming.

The encoding had been paid much care to support using unix sort.

> Is Unix Sort ok?

They are, as long as you treat the stram as binary, I set flag `LC_ALL=C` to do that.

> Why encode docID in binary form not Text Frequncies (IF)?

Because **most** TF are below 10. They take up 1 byte to store. Thus using binary form for it **won't benifit** much.

Hence designing and **computing** each time when accessing an encoding that convert to and from number would both take more time.

> Why Merge Stage take so little memory ?

Beacuse the design of `merge.py` has take as much advantage of streaming as possible.

It doesn't wait till an Inverted List is completed to unload memory, it streams out all `doc Item` as long as they get them.

> Why not C++

Beacuse most C++ statements are in not-human-like language, I want to keep myself human :)

Actually there are language that are close to C++ level of proformance e.g. Swift, Go

The real reason are:

- Packages on high level langauges are richer than C++
- I'm personally not confident with C++ knowledge (learn to use STL someday?)
- C++ design patterns are not singlar, roughly going into it would cause mix use of different "flavor of codes", which I'm not fond of.

Future Work

There are several works can be done easily but requires more careful thoughts

Distributed

The whole program is written in a `Map and Reduce` concept. They can be easily ported to Hadoop MapReduce. Here is a list of what has been done:

- Python package distribution with virtual env support using Hadoop
- UUID and UUID compression support (higher entroy encoding for UUID)

And what to be done is:

- Map Reduce compatiple URLTable Generation
- Glue code to pipe them all

Considering Hadoop Stream isn't actually efficient, Spark would be a good replacement for that, though how to port `Language Detection` and `Chinese Support` to Scala and Spark.

Speed up

Change a language like `Go` might incredibaly speed up exection. (But packages?)

How about keep using python? Regex used in the current implentation can be further optimized to speed up.

Query optimization

Query process requires to read out the wet file efficiently, building index on wet file is considered nessary.

This project had been use a modified python package `warc` . It has been modified to support reading `wet` files to adapt to this project (check warc3-wet). Futher modification is required for support fast lookup in wet files.

Also `ii` file could be further compressed using text compression among with incremental docIDs (which will not work on UUID aka. distibuted system) block by block.

`IDF` calculations may also be pre-calculated.

Development

Add new requirements if new python packages are used

```
$ pip freeze > requirements.txt
```

If to Change of README.md file. There is a ruby script to build Markdown Table of Content:

```
$ ruby scripts/generate_toc.rb
```