

Documents & Data on the Web

Coursework

Name: weilue lu

ID: 10404762

Date: 16 Nov 2020

Documents & Data on the Web Coursework

Preface

Functionality

Features, Improvements & Problems

- Stream

- Raw Tokenize

- Tokenize From Line

- Expand Tokens

- Stem & Stopword Removal

- Grouping & Reducing

- Sorting

Other Limitations

Performance

Design Patterns & Effects

- Positional TF Term

- Stopword Removal

- In-mapper Aggregation

Other Limitations

- Map Reduce

- Stream

- Expand Tokens

Appendix

Word Count

50 lines output (1700 words)

Preface

I am using IntelliJ with Java 8. Note that there are more technical explanations in the code. Note that I put many utilities functions / classes into the `BasicInvertedIndex` file, this is bad practice, but according to instruction I can only submit this class' file.

Functionality

Features, Improvements & Problems

Stream

Highly readable, efficient, pipelined code using the `Stream` API from Java 8.

Raw Tokenize

It splits text to lines by:

- Any tab, carriage-return, newline, form-feed, double-quote, and single-quote.
- Any dot, question mark, exclamation mark that follows by space/tab/carriage-return/form-feed/newline.

I included quotes because they are common for quoting.

Dot/Question mark/Exclamation mark must follow by delimiter for more precise match of end-of-sentence, e.g. it will not split statements like: $10.0 \times 10! = \lambda$.

While it generate more suitable index term, it assumes usage of quotes which may not be true. Moreover, the tokenization is design for English, separators of other languages are not considered.

Tokenize From Line

It cleans the word and returns true if only the first letter is uppercase, making it more precise.

Expand Tokens

A consideration on multi-term/single term:

- **Number:** keep original token because user may search using it.
- **Date:** Keep both original token and numeric form, because user may just enter number without separator to search for dates.
- **URL:** Break it into pieces by punctuations, because the user may not search the full URL.
- **Connected Word:** Keep each pieces as well, e.g. ease-of-access → [ease, of, access, easeofaccess, EOA].
- **Other:** Keep letters only (which is not good for other languages).

All tokens share the same position to ensure it reflects the actual position. This process may also generate blank token, but it will be removed in latter stage. Normally, letters-only tokens are added for more robust indexing terms.

Stem & Stopword Removal

Lemmatization and stopwords removal are achieve with stemming and `StopAnalyser`, it collapse like-terms into one which make it more suitable for indexing, however, it may introduce over-stemming problem since it is a crude process.

Grouping & Reducing

- Used `Set` for positions, collapsing same expanded terms.
- Note the size of positions is term frequency.

Sorting

Each inverted-index is sorted using simple heuristic.

$$\text{ranking} = \text{tfidf} + \frac{\sqrt{\sigma + \lambda}}{\sigma + \lambda}$$

where σ is the variance of the positions and λ is a constant to reduce score for low-occurrence-words.

- Not using standard-deviation because variance gives smoother step.

- I used $\lambda = 20$ to dampen effect of single occurrence term and avoid division by zero.



Example

• TFIDF

```
// TFIDF
I DF: 6 [(File=Bart_the_Fink.txt.gz TF=6 TFIDF=0.11904151769084564 Ranking=0.11904151769084564 pos=[1333, 1541, 1726, 1983, 2008, 2034]
, (File=Bart_the_Murderer.txt.gz TF=5 TFIDF=0.11374058746900548 Ranking=0.11374058746900548 pos=[1106, 1277, 1304, 1575, 2035]
, (File=Bart_the_Lover.txt.gz TF=3 TFIDF=0.09888852589862468 Ranking=0.09888852589862468 pos=[1978, 2021, 2083]
, (File=Bart_the_Genius.txt.gz TF=3 TFIDF=0.09888852589862468 Ranking=0.09888852589862468 pos=[1332, 1769, 2159]
, (File=Bart_the_Mother.txt.gz TF=3 TFIDF=0.09888852589862468 Ranking=0.09888852589862468 pos=[167, 1773, 2433]
, (File=Bart_the_General.txt.gz TF=2 TFIDF=0.08709978142283419 Ranking=0.08709978142283419 pos=[905, 1385]
```

Although Bart_the_lover has less occurrence of the token "I", it is better than Bart_the_Murderer because those three words have very low variance, therefore the variance is getting too little weight here.

• My heuristic (Standard Deviation)

```
// My heuristic (Standard Deviation)
I DF: 6 [(File=Bart_the_Lover.txt.gz TF=3 TFIDF=0.09888852589862468 Ranking=0.25033525607177665 pos=[1978, 2021, 2083]
, (File=Bart_the_Fink.txt.gz TF=6 TFIDF=0.11904151769084564 Ranking=0.18057041559737616 pos=[1333, 1541, 1726, 1983, 2008, 2034]
, (File=Bart_the_Murderer.txt.gz TF=5 TFIDF=0.11374058746900548 Ranking=0.16919778218056128 pos=[1106, 1277, 1304, 1575, 2035]
, (File=Bart_the_Genius.txt.gz TF=3 TFIDF=0.09888852589862468 Ranking=0.15325698875616384 pos=[1332, 1769, 2159]
, (File=Bart_the_General.txt.gz TF=2 TFIDF=0.08709978142283419 Ranking=0.1515823694447503 pos=[905, 1385]
, (File=Bart_the_Mother.txt.gz TF=3 TFIDF=0.09888852589862468 Ranking=0.13129729918354424 pos=[167, 1773, 2433]
```

By using standard deviation heuristic, Bart_the_lover raised to the first place, but clearly over-done, because although Bart_the_Fink has higher variance, it has low variance with positions: 1983, 2008 and 2034, therefore the variance is getting too much weighting here.

• My heuristic (Variance)

```
// My heuristic (Variance)
I DF: 6 [(File=Bart_the_Fink.txt.gz TF=6 TFIDF=0.11904151769084564 Ranking=0.12283448907771372 pos=[1333, 1541, 1726, 1983, 2008, 2034]
, (File=Bart_the_Lover.txt.gz TF=3 TFIDF=0.09888852589862468 Ranking=0.12208760001627603 pos=[1978, 2021, 2083]
, (File=Bart_the_Murderer.txt.gz TF=5 TFIDF=0.11374058746900548 Ranking=0.11682081724334833 pos=[1106, 1277, 1304, 1575, 2035]
, (File=Bart_the_Genius.txt.gz TF=3 TFIDF=0.09888852589862468 Ranking=0.10184882439333462 pos=[1332, 1769, 2159]
, (File=Bart_the_Mother.txt.gz TF=3 TFIDF=0.09888852589862468 Ranking=0.09993940607922257 pos=[167, 1773, 2433]
, (File=Bart_the_General.txt.gz TF=2 TFIDF=0.08709978142283419 Ranking=0.09126643000512785 pos=[905, 1385]
```

By using variance heuristic, Bart_the_Lover got the optimal position, thus this heuristic is most suitable among three variances.

Note that this heuristic does not work well for some cases, e.g.:

- Single occurrence term, because it has 0 variance which made it score very high.
 - To dampen this effect you can assign a higher λ (e.g. 20).
 - Another options would be make use of the number of positions where less positions gets lower weighting.
- Short documents, $\frac{\sqrt{\sigma}}{\sigma}$ will be very steep which made it sensitive to variance, resulting in overly-high difference.

Other Limitations

- Lack of language support.
- Lack of document type support, it assumes the input are plain text, cannot handle images, videos, etc.
- The handling of special tokens are not robust enough.
 - Some unhandled token includes: measures, citations, hashtags.
- No check for spelling, assumed document text is correct, but this shouldn't have much impact.
- The inverted index is sorted alphabetically which may not be helpful.

Performance

Design Patterns & Effects

The overall design pattern is MapReduce, which is optimized for distributed computing. Obviously, this slows down performance since we only have one computer with little data, but for the purpose of coursework it is fine.

I made heavy use of features such as `Stream`, `lambda expression`, `method references` to chain operations to achieve high efficiency.

Positional TF Term

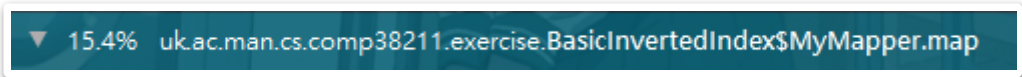
It is used to group all relevant data for term, and provide useful utilities such as merging. It follows the immutability and factory design pattern which made it safer and easier to use.

Stopword Removal

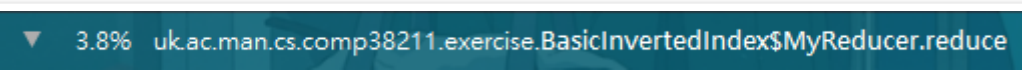
I used `Set` instead of `List` in `StopAnalyser` for faster check.

In-mapper Aggregation

An profile analysis using Java Flight Recorder revealed that `map` and `reduce` functions took 15.4% and 3.8% respectively and overall runtime of 4.52s.

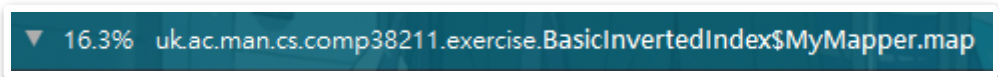


▼ 15.4% uk.ac.man.cs.comp38211.exercise.BasicInvertedIndex\$MyMapper.map




▼ 3.8% uk.ac.man.cs.comp38211.exercise.BasicInvertedIndex\$MyReducer.reduce

Without in-mapper aggregation, the `map` and `reduce` functions took 16.3% and 11.6% respectively and overall runtime of 10.011s.



▼ 16.3% uk.ac.man.cs.comp38211.exercise.BasicInvertedIndex\$MyMapper.map



▼ 11.6% uk.ac.man.cs.comp38211.exercise.BasicInvertedIndex\$MyReducer.reduce

From above we can see my in-mapper aggregation has significant impact on the performance.

Other Limitations

Map Reduce

- Reduce/Sorting is take very long time if map stage generate too many keys.
- It is built for batch processing so not useful for interactive job.
- It is not suitable for tasks that have dependency on each other as they cannot be parallelize.

Stream

It can create unnecessary operations and less flexible than loops.

Expand Tokens

If we were to run this on huge datasets, the bottleneck will be expand token process as it takes comparatively long time to check for special tokens (which is not in most cases).

Appendix

Word Count

Excluding appendix, word count is 978:

Page 1 of 6 978 words

50 lines output (1700 words)