Text Technologies: Assessment 2
A Fast Boolean Search-Engine for Twitter

Aaron Cronin · s0925570 · The University of Edinburgh

1 Introduction

1.1 Aims

This project aims to produce a system written in `Python` for retrieving *tweets* that contain all terms in a given query, as fast as possible. Well-studied scoring algorithms such as *brute-force*, *term-at-a-time*, and *document-at-a-time* have been implemented and compared.

1.2 Dataset

Algorithms were tasked with finding the 5 most recent documents, out of a collection of $22,310$, matching all terms for each of 4747 queries. The best algorithms then faced the same task against $84,983$ queries and $6,979,346$ documents – exploring the limits of their scalability.

2 Existing Algorithms

2.1 Brute Force

An algorithm with the pseudocode detailed in Figure 1 has a worst-case complexity of $O(\|Q\| \cdot \|D\| \cdot \textbf{avg}\ len(d \in D))$ over all Q.

Implementation Although the worst asymptotically of all approaches, `brute.py` performed well for the small collection of documents in the basic assignment. Since only the 5 most-recent matching documents were required for each query, lazy iteration over descending document `id`s gave huge performance increases for common queries matching large numbers of documents. Documents and queries were stored as `Set`s of word units, providing a $O(n)$ subset test a with very low constant compared to most `Python` operations. The basic challenge took around 12.5 seconds for *brute-force*.

2.2 Term-at-a-time

A pseudocode algorithm that calculates document scores while traversing inverted indices of terms (T) is provided in Figure 2. Over all Q this has complexity $O(\|Q\| \cdot \|N\|)$ where N is the total magnitude of all inverted lists, or *postings*.

Implementation Since the search-engine is using *Boolean* matching, the scores for each document were simply incremented each time the document appeared in a posting. Documents with scores equal to the number of unique terms in the query matched and should be emitted. The final stage when returning documents after *term-at-a-time* scoring involves a walk along the set of documents D, filtering those that have sufficient score to be returned. As with `best.py`, significant average-case speedup was achieved by lazily iterating over descending documents until either 5 matches were returned or all document scores had been inspected. The basic challenge took around 12 seconds for *term-at-a-time*.

2.3 Document-at-a-time

An algorithm that uses linear-merge to travel down inverted indices, emitting documents, is shown by pseudocode in Figure 3. This has complexity $O(\|Q\| \cdot \|N\| \cdot \textbf{avg}\ len(q \in Q))$ over Q. However since *document-at-a-time* only requires one member of the inverted index for each term at any one time for merging, it has far lower memory requirements than *term-at-a-time* – $O(\textbf{max}\ len(q \in Q))$.

Implementation Detailed in the pseudocode, returning matching documents for a query Q is simply a linear-merge over the postings of term T. Linear-merge was performed as in Figure 4. Again, the merging can be lazily iterated to support early termination. Even with early termination the naive *doc-at-a-time* algorithm performs poorly on the basic document set, taking around 56 seconds. Uninterestingly, this is mainly due to `Python`'s poor suitability for the large number of additions and comparisons required when moving pointers and the inefficiency of indexed array access compared to a lower-level language such as `C`.

3 Improvements

While all the above implementations benefitted from early termination and lazy iteration of the matching functions, *document-at-a-time* was the most suited for further optimisation. This is novel since it was by far the slowest algorithm when implemented without any performance

enhancements.

3.1 Document-at-a-time with improved linear-merge

Linear-merge in `Python` was the significant bottleneck in the naive *document-a-time* algorithm, so it was replaced with the improved procedure detailed in Figure 5. Finding insertion points using bisection takes $O(log(n))$ time, this is a significant improvement over the linear, repeated-decrement of pointers for merging sparse lists. *Zipf's law* implies that the terms in a query do not co-exist in most documents, despite one or two occurring in almost all, and therefore optimising for merges where most time is spent travelling benefits the search algorithm greatly. This improved algorithm, implemented in `doc2.py`, was included in run-time graphs to show the difference made by speeding up the merge.

3.2 Set-based merging

The fastest solution to the basic challenge is that explained in Figure 6, it shows that further improvement to the performance of the retrieval system can be achieved by using `Set` intersection over a sorted list of inverted indices. Intersection over n inverted indices takes time linear with the length of the smallest `Set`, as long as the intersections are calculated in ascending length order. In addition, since intersection over n inverted indices can terminate as soon as the 'current' intersection is the empty set, by pushing the postings onto a heap we can avoid sorting the entire list of sets by length in the case of early termination. The average complexity of $O(\|Q\| \cdot \mathbf{avg}\ tf \cdot \mathbf{avg}\ len(q \in Q))$ is significantly faster than competing search algorithms tested.

3.3 Heuristical Algorithmic Selection

Empirical analysis, shown in Figure 7, hints that the complexity of converting the inverted indices into sets and maintaining a heap each query is not worth it when evaluating queries over fewer than 900 documents in the basic challenge. With this in mind, the `best.py` solution chooses *improved document-at-a-time merging* when the document set is smaller than 900 and *set-based merging* otherwise.

4 Results

Run-times were gathered by running each retrieval algorithm 10 times for a given document set and averaging the time taken. Figure 9 shows the run-time results for running increasingly-large portions of the document set through the differing retrieval algorithms, plotted on a log-log graph. In order of decreasing performance over $20,000$ documents, the algorithms are: *set-intersection merging*, *improved doc-at-a-time*, *term-at-a-time*, *brute force*, and *simple doc-at-a-time*.

Figure 8 shows the run-time results for matching even larger portions of the extended document-and-query set using the fastest two algorithms, *improved doc-at-a-time* and *set-intersection merging*. Slower algorithms were discarded as they would not have iterated over $84,983$ queries and $5,000,000$ documents in anywhere-near reasonable time.

5 Conclusion

Brute force, although initially attractive in practice, quickly degrades into a completely unusable state so should not be considered as a viable retrieval method. *Improved document-at-a-time* performs well in this assignment, partially because of it's suitability to optimisation in `Python` – bisection is performed by a highly efficient `C` runtime in the library. In theory, *term-at-a-time* should eventually perform better than *document-at-a-time* for a large enough dataset, derived from examination of growth rates. However, it requires all data in memory for iteration so in practice would be greatly restricted in scalability by this hard constraint. Unfortunately *term-at-a-time* performs far too slowly, with the constants involved, to test on a very large dataset and compare its scalability with the *improved doc-at-a-time*. Over the largest dataset it appears that *improved document-at-a-time*, tuned for sparse merges, eventually grows at a slower rate than *set-based-merging*. Though, at this point, it is still taking 10 times as long to complete. This difference in growth may stem from the improved *doc-at-a-time* taking advantage of the properties of natural language. Without further experimentation it is impossible to reason about this reliably.

6 Further Work

The effects of the `Python` runtime heavily skew the results gathered. It would be wise to repeat experimentation on the base algorithms, implemented in a language without *Virtual Machine* complications – such as `C`. With more reliable implementations it would be interesting to increase the number of documents processed to an even higher limit, investigating whether *improved doc-at-a-time* performs better in extreme cases.

7 Acknowledgements

Progress on this project was compared with flatmate Jacob Essex, alongside analysis of algorithms covered in lectures. No code was shared and all evaluation of results is individual.

8 Appendix

8.1 Pseudocode

```
To find D* matching a query Q:
  Iterate through all D:
    Emit d if d is subset of Q.
```

Figure 1: Brute-force pseudocode

```
With scores for D all set to 0:
To find D* matching a query Q:
  Iterate through T in Q:
    For all D in T's inverted index:
      Increase score.
Return recent D with sufficient score.
```

Figure 2: Term-at-a-time pseudocode

```
To find D* matching a query Q:
  Linear-merge postings for T in Q:
    Emit document if contains all terms
```

Figure 3: Doc-at-a-time pseudocode

```
Postings are already sorted,
     by ascending doc_id.
Create pointers P to ends of postings.
Until you have 5 documents:
Pluck doc_ids at pointers as Frontier.
Count how many match max doc_id.
    If all matched, emit doc_id.
    Otherwise, decrement matching P.
    Finish if any P were at start.
```

Figure 4: Simple linear merge pseudocode

```
Find maxmin of all inverted indices P,
(max of the list of final postings)
Using binary search for each P:
Store left insertion point of maxmin, E
Create pointers P to ends of postings
Until you have 5 documents:
Find the minmax of the current merge,
(minimum of the frontier)
Move Ps to right insertion point of minmax
Count how many match max doc_id.
    If all matched, emit doc_id.
    Otherwise, decrement matching P.
    Finish if any P were at end E.
```

Figure 5: Improved linear merge pseudocode

```
To find D* matching a query Q:
  Push postings P for T in Q onto heap:
       Intersect sets for Ps
       (in order of increasing len)
Return <= 5 docs from intersection.
```

Figure 6: Set-based document-at-a-time pseudocode

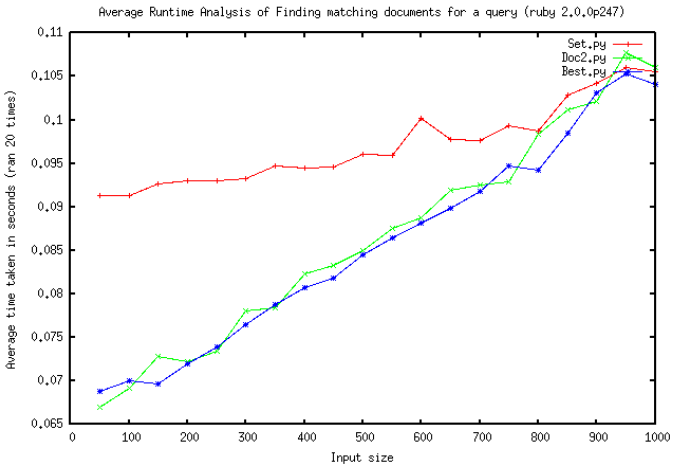

Figure 7: Performance of select retrieval algorithms on smaller document counts

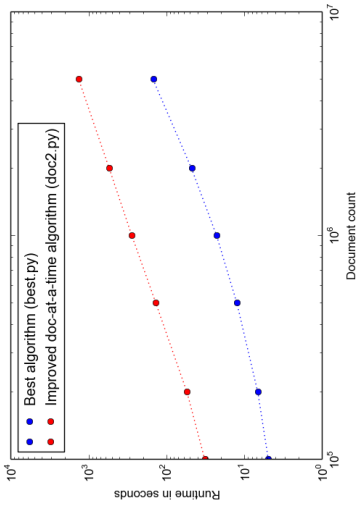

Figure 8: Performance of two best algorithms over the largest document set

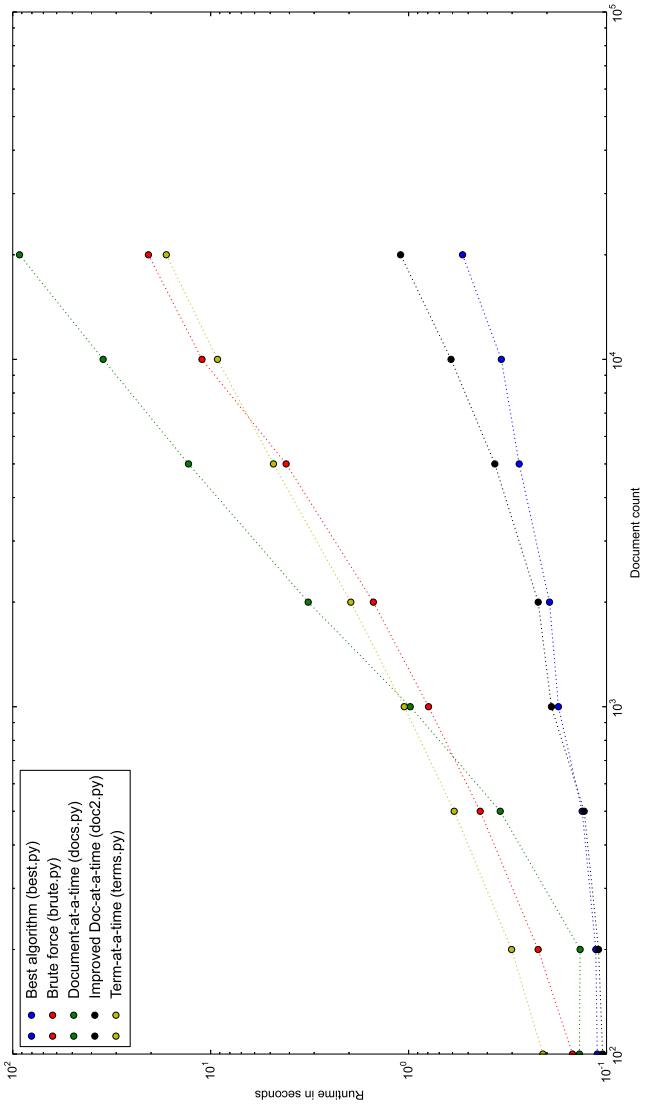

Figure 9: Performance of all retrieval algorithms over the simple document set