Large-scale Generative Query Autocompletion

David Maxwell*
School of Computing Science
University of Glasgow
Glasgow, Scotland
d.maxwell.1@research.gla.ac.uk

Peter Bailey
Microsoft
Canberra, Australia
pbailey@microsoft.com

David Hawking
Microsoft
Canberra, Australia
dahawkin@microsoft.com

ABSTRACT

Query Autocompletion (QAC) systems are interactive tools that assist a searcher in entering a query, given a partial *query prefix*. Existing QAC research – with a number of notable exceptions – relies upon large existing query logs from which to extract historical queries. These queries are then ordered by some ranking algorithm as *candidate completions*, given the query prefix. Given the numerous search environments (e.g. enterprises, personal or secured data repositories) in which large query logs are unavailable, the need for synthetic – or *generative* – QAC systems will become increasingly important. Generative QAC systems may be used to augment traditional *query-based* approaches, and/or entirely replace them in certain privacy sensitive applications. Even in commercial Web search engines, a significant proportion (up to 15%) of queries issued daily have never been seen previously, meaning there will always be opportunity to assist users in formulating queries which have not occurred historically. In this paper, we describe a system that can construct generative QAC suggestions within a user-acceptable timeframe (~58ms), and report on a series of experiments over three publicly available, large-scale question sets that investigate different aspects of the system's performance.

ACM Reference format:
David Maxwell, Peter Bailey, and David Hawking. 2017. Large-scale Generative Query Autocompletion. In *Proceedings of The 22nd Australasian Document Computing Symposium, Brisbane, QLD, Australia, December 7–8, 2017 (ADCS 2017)*, 8 pages.
DOI: 10.1145/3166072.3166083

1 INTRODUCTION

The majority of existing research into *Query Autocompletion (QAC)* has assumed access to substantial volumes of query logs, which are mined in advance for a series of appropriate queries to suggest. These historical queries can be ranked relative to a user's given *query prefix*, with the suggestions displayed in an interactive user interface. These conventional QAC systems that rely upon historical query logs for their *suggestion candidates* are referred to in this paper as *query-based* QAC systems.

Despite the advances in query log-based QAC systems, there are many information search environments (e.g. enterprises, personal or secured data repositories) in which query logs are far less extensive (than those of a commercial web search engine, for example), or even not collected at all. Furthermore, the query and document vocabulary associated with such restricted information domains may diverge from public web information sources, as investigated by Smyth et al. [31]. Thus it is unlikely that a commercial web QAC system could successfully transfer to these other environments without major modifications. Even in major commercial web search engines, ~15% of queries have never been seen previously.[1] As people continue to engage more via natural language in their information seeking interactions, search queries will more often be expressed in natural language. Thus, unseen queries are likely to increase due to the *vocabulary problem* [15] in human-computer information system interactions. Indeed, the natural language capabilities exhibited by modern commercial web search engines will likely increase the expectations of other search systems. QAC approaches based on query logs simply cannot make suggestions for query completions which are not in their data.

To this end, in this paper we detail a *generative* QAC approach, to which a small number of prior works have previously attempted. One of the first to identify an approach to the problem, Bhatia et al. [4] provided a compelling rationale for generative QAC, developed an algorithm using n-gram statistics from target document collections, and carried out effectiveness analysis of the algorithm over two document collections. They discuss many aspects of their approach, and also discuss future work relating to efficiency improvements after acknowledging that *"the target systems for our approach typically have smaller scale datasets and so [...] the efficiency of our algorithm is not critical"*.

We concur with Bhatia et al.'s premise of the importance of generative QAC, but we differ on the importance of efficiency – we believe it is essential that QAC systems should be engineered to deliver results at the speed of individual human keystrokes. As such, we have sought to develop an efficient generative QAC approach that can target web-scale information sources. In our experiments, we used three sets of queries (UQV100 [1], SQuAD [28], and MS MARCO [26]), ranging from ~5,765 to ~203,000 queries per collection, and the titles of the documents from the ClueWeb12 corpus (CW12) [27] as the generative document corpus for creating n-grams. We used the titles-only from CW12, rather than the full document texts, to keep the scale tractable, but this remains a substantial text base (~730 million titles). For every query, we wish to examine all the possible word-boundary substrings as query prefixes. For example, if the original (or target) query is

*Author was an intern at Microsoft when this work was undertaken.

ADCS 2017, Brisbane, QLD, Australia

978-1-4503-6391-4/17/12…$15.00
DOI: 10.1145/3166072.3166083

[1] https://www.cnet.com/news/google-search-scratches-its-brain-500-million-times-a-day/ – accessed October 30th, 2017.

'best computer to buy', then the word-boundary substrings are 'best', 'best computer', and 'best computer to'. Evaluation involves assessing whether we can predict the target query given a word-boundary query prefix, and measuring precision via MRR. For UQV100, this expands from 5,765 queries to a set of 24,989 query prefixes to consider. We then average performance over the query dataset, for all the possible prefixes. Note that this is a slight simplification of many QAC evaluations, which consider every character position within a prefix, which we do not do because of the large scale query sets. More details of our experimental method and evaluation are detailed in Section 3.

In our system, suggestion candidates returned to users are produced using an *n*-gram language model created in real time from documents that match the prefix terms selected from the larger document corpus. This approach dramatically cuts down the scale of data to be considered, but does rely on a very fast query processor. The described system is able to generate these suggestions within a user-acceptable timeframe. Since both the prefix and the *n*-grams used to construct completions are present in the matched documents, our approach honours the implicit user contract of QAC systems, in that any returned suggestion should yield a query that returns results (of at least one document) from the larger corpus. Further details of the various stages of the proposed system are provided in Section 3.1.

We report on a series of experiments that investigate different aspects of the system's performance over three publicly available, large-scale question sets, allowing reproducibility of our findings. Unable to compare our approach to the Bhatia et al. [4] algorithm for reasons discussed in section 3.3, we revert to using a simpler "NextWord" *n*-gram prediction algorithm, using a web-scale *n*-gram corpus, as the baseline. These experiments allow us to address the following generative QAC focused research questions.

RQ1 How does retaining or dropping stopwords from the query prefix when finding matching documents impact suggestion performance?

RQ2 Does increasing the set of matched passages to generate the language model increase the probability of creating a matching QAC suggestion?

RQ3 Can performance of the generative QAC suggestions be improved by re-ranking?

2 RELATED WORK

With early human-computer interfaces heavily *recall-based* — where a user would have to remember a range of commands to enter precisely — early advancements in reducing the user's cognitive load led to the concept of *tab completion*, which we see today in shells such as *Bash*. Term completion (or QAC) carried over to search with *Google Suggest*[2], with all major commercial web search engines today offering some form of QAC functionality. QAC systems aim to reduce the time taken by a searcher to enter a query, reduce the likelihood of spelling mistakes (10-15% of queries entered were misspelt before the introduction of QAC [10]), discover relevant search items, and even assist with potential query reformulation(s) [6, 20].

Literature in the area of QAC is diverse, with QAC often viewed as a ranking problem [2, 6]. Works have focused upon improving QAC suggestion candidates with a variety of techniques (both heuristic-based and learning-based [6]), as well as how such approaches are evaluated. Early QAC approaches ranked suggestion candidates using the *Most Popular Completion (MPC)* [2] approach — that is based upon the popularity of previously issued queries in a query log matching the query prefix entered by the user. Di Santo et al. [11] collate a series of other QAC models that are defined in the literature. A variety of other approaches have been since considered, including *time-sensitive QAC* [5, 7, 30, 32] and *contextual* and *demographic-based QAC* [2, 19, 22, 23, 25, 29]. For evaluation, *Mean Reciprocal Rank (MRR)* [8] has become the *de facto* measure [6]. As cited by Cai and de Rijke [6] and Hofmann et al. [17], other approaches — such as a keystroke-based measure [12], for example — have also been considered. Use of anchor text instead of historical queries was explored for the purpose of query refinement by Kraft and Zien [21]. For a more detailed examination of query log-based QAC literature refer to Cai and de Rijke [6] for a recent and comprehensive survey of the area.

As a majority of prior QAC research is query log-based, we address in this paper the issue of *when query logs are not available.* How can QAC suggestions be generated — and ranked — if no such prior examples exist, particularly as the size of the corpus becomes increasingly large?

One approach is to use entries within structured data sources as the basis for candidates. Hawking and Griffiths [16] mined *n*-grams from structured enterprise data sources such as staff directories, research grant databases, commercial sales offers, etc., and matched them using both prefix and infix methods. Their aim was to construct an extended QAC system which could also navigate directly to targeted information resources, or execute callback operations via JavaScript, in addition to suggesting search queries. They carried out a user study to examine the keystroke savings and user preferences when performing staff contact lookup tasks. The use of staff directories for an enterprise QAC system was also investigated previously by Ji et al. [18]. They used fuzzy matching techniques to address the problem of mis-spelling or name abbreviations in query prefixes.

Another approach is to generate completions from overlaps between the query prefix and a language model, even when the final suggestion may not appear fully within the language model. Work in this generative space is limited; we are aware of only a handful of examples [3, 4, 13, 24]. Recognising and responding to the challenge of providing QAC without query logs, Bhatia et al. [4] for example provided a compelling rationale for generative QAC. They developed an algorithm using *n*-gram statistics from target document collections, and carried out an effectiveness analysis of the algorithm over two document collections. More recently, Mitra and Craswell [24] developed neural embedding models of common query completions (albeit mined from query logs), and demonstrated how a generative QAC approach from these embedding models can be tuned to assist users formulate rare queries in open web search environments.

[2] https://googleblog.blogspot.com/2004/12/ive-got-suggestion.html – accessed October 30th, 2017.

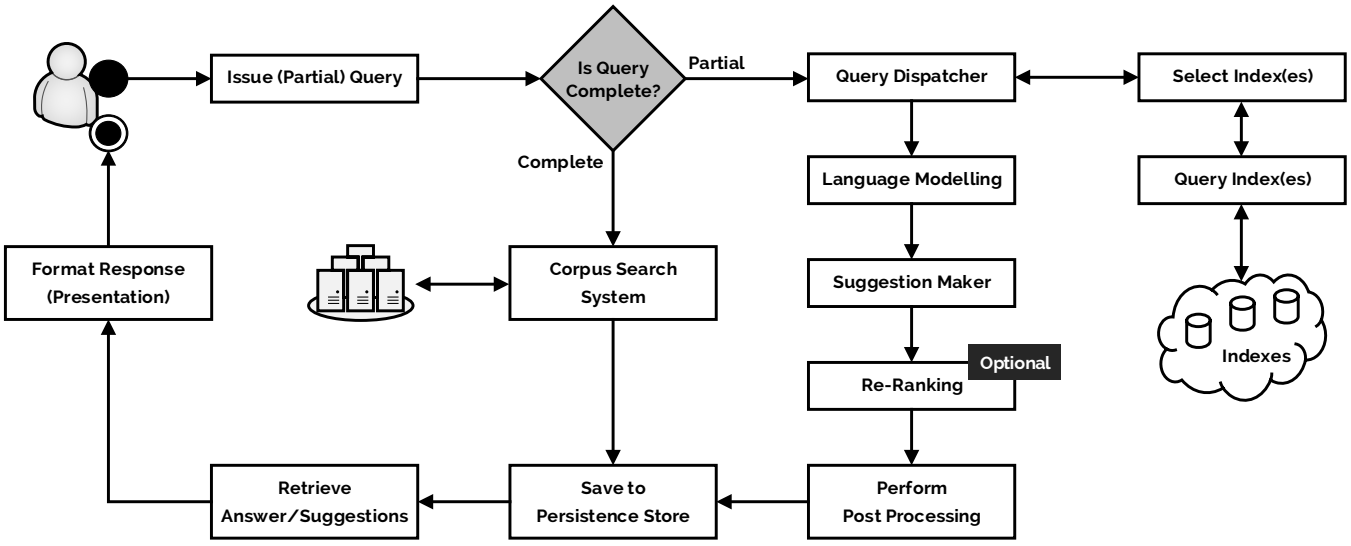

Figure 1: Flow diagram illustrating the overall system architecture of the proposed combined query suggestion and query processing systems. Users begin interaction by posing a (partial) query, after which the system delivers an answer or suggestion to the user. This, while adhering to the principle of delivering a response within an acceptable timeframe. Refer to section 3.1 for more information.

In both these works, the authors identify that one of the problems of generative QAC is to avoid completions that do not make sense in the context of the query prefix. The Bhatia et al. [4] algorithm seeks to minimise such completions by including a phrase-query correlation component of their scoring method ([4], Equation 12) to re-weight candidate completions which are unlikely to occur in the corpus, given the prefix.

In contrast to the Bhatia et al. [4] algorithm, rather than pre-computing a language model from a document collection, we only start to construct a language model from those documents which match the current query prefix as the user types it in real-time. This language model is then used as the source of candidate n-grams for use in generating plausible completions.

3 EXPERIMENTAL METHOD

In this section, we present the experimental system that we used in order to address the three research questions as outlined in Section 1.

3.1 Experimental System

Our system is built using a pipeline architecture, as depicted in Figure 1. The system consists of eight main subsystems – each of which is detailed here. First, a standard **interactive user control** is present for intercepting keystrokes issued by the user. For experimental purposes however, this is replaced with a test harness that takes query prefix terms from the question sets detailed in Section 3.2. A **query dispatcher** then preprocesses the query prefix, and sends the post-processed version to a high performance, **short text query processor**, operating over the CW12 titles index. In other circumstances, multiple indexes might be used here, where the index selected might be conditioned on the query prefix length or other factors. Any standard search engine could be used here, such as *Indri*. We however use a proprietary search engine that is optimised for large-scale short text data, but in two simple matching modes, where documents are returned if:

(i) a match exists for all terms; or

(ii) all terms match such that the last term of the query is handled as a wildcard prefix match (e.g. if the term is `'pi'`, terms are matched with `'pi*'`).

The maximum number of matching documents to be returned will affect the richness of the language model, but there is a tradeoff with increased processing time. A **language model generator** then produces a language model from the top N matching short texts returned by the query processor. Any number of different algorithms for computing the language model can be used to implement this component. For the purposes of this study however, we use a relatively simplistic approach – our starting algorithm obtains all n-grams up to quad-grams from each returned document (considering the titles only to reduce computation time), and the score for the n-gram in the model is simply the count of its occurrences over all documents. A complete **suggestion maker** then combines the query prefix and *candidate completions*, including infix insertions of the candidate completion within the query prefix if appropriate. The algorithm for determining insertions of an n-gram into the current prefix is mildly complex, and given in pseudo-code in Algorithm 1. An optional **re-ranker** can then reorder suggestions based upon various rules, such as source diversification, or additional ranking information. A **truncation and formatting** component then takes the final suggestions, and prepares them for sending to the interactive user control. As the user control is absent from our experimental setup, we instead replace it with a **suggestion scorer** component, which is discussed below.

Algorithm 1 Pseudo-code for *n*-gram insertion algorithm - part of the **Suggestion Maker** component (refer to Section 3.1).

Concepts:

- *Gram*: the current *n*-gram from the (local) language model
- *Prefix*: the query/question characters that have been entered to this point
- *Candidate*: the suggestion which will be constructed (combining the prefix and the gram)
- *Term*: a complete word within a prefix or gram

Pseudo-code:

```
1:  if gram contains spaceCharacter then                          ▷ that is, gram is 2 or more terms in length
2:      firstMatchingTerm ← find first occurrence of any of the gram's ordered terms within the prefix
    ▷ Variation : remove stop words from the gram before doing this match
    ▷ Also, there may be no matching terms between the prefix and the gram
3:
    ▷ Find the position within the prefix to insert the gram
4:      if no matching terms then                                 ▷ firstMatchingTerm is blank
5:          insertionPoint ← end of prefix
6:      else
7:          insertionPoint ← position of firstMatchingTerm in prefix
    ▷ Place gram into prefix
8:      candidatePrefix ← substring of prefix from starting character to insertionPoint
9:      candidate ← candidatePrefix + spaceCharacter + gram + spaceCharacter
10:
    ▷ If gram was not placed at end of prefix, then splice any terms that were omitted onto the end
11:     droppedTerms ← terms in prefix that are not in candidate
12:     if count of droppedTerms < 0 then
13:         candidate ← candidate + droppedTerms
14:     else
15:         do nothing                                            ▷ we already have a complete candidate
16: else                                                          ▷ gram is a unigram
17:     if prefix contains spaceCharacter then                    ▷ that is, prefix is at least 1 full term
18:         if last term in prefix is incomplete AND is a substring of the gram then
19:             candidate ← (prefix − last term) + gram
20:         else
21:             candidate ← prefix + gram
22:     else                                                      ▷ we ignore cases where the prefix is only a single partial term, since these
23:                                                               ▷ are better solved by simple single word probability data structures
24:         do nothing
25:
    ▷ Once this algorithm is complete, candidate is now our suggested query autocompletion.
```

3.2 Corpus and Question Sets

To promote reproducibility of our experimental findings, we used a series of publicly available datasets for this study. The first is the *ClueWeb12* (CW12) [27] document corpus, from which we separately extracted the titles of individual documents. CW12 consists of a total of 733 million documents; we extracted 728, 893, 907 document titles only (with some documents having invalid titles) to be the corpus for our proprietary indexing and search technology. This decision was taken to keep the scale of our experiments tractable, yet still possessing a substantial text base. This document titles corpus has a vocabulary size of ~6.12 million words, with ~233 million bi-grams, and ~556 million tri-grams. This data is substantially greater than the *TREC* corpus as used by Bhatia et al. [4]: 32 times as many unigrams; 21 times as many tri-grams.

Overall, our indexed version of the CW12 index measured approximately 59 GB in size. Using our proprietary indexing software, this index fit comfortably within the 512GB of system RAM available in our high performance server. Ensuring the index fit within available system RAM was crucial to reduce I/O latency with backing storage, allowing us to be sure that responses to (partial) queries could be responded to within an acceptable timeframe.

Three question sets were used, again all publicly available. These were: the *UQV100* query variation test collection (crowdsourced web queries) [1]; the *MS MARCO* machine reading comprehension set (a series of Bing search engine web queries) [26]; and the *SQuAD* natural language question set (crowdsourced passage comprehension questions) [28]. In each case, all queries/questions were normalised by converting to lowercase, dropping punctuation, and removing duplicates. The total number of queries for each question

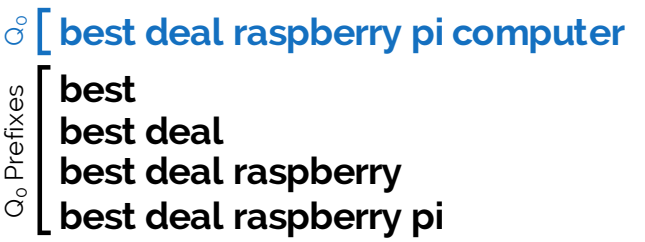

Figure 2: An example query Q_0, along with its four *"unrolled"* query prefixes, Q_0 ***Prefixes***.

set after this pre-processing were: 5764 for UQV100; 202, 768 for MS MARCO; and 98, 169 for SQuAD.

For each query in the three question sets, we "unrolled" all the partial query prefixes of the full query on word boundaries, as illustrated in Figure 2. The goal in all cases was to predict the full query, given just the partial query prefix. Each partial query prefix was processed in two ways and then combined: first, as if the user had stopped typing at the last character of the prefix (in which case the QAC system has to assume the word is incomplete, and the completions may commence with it); and second, as if the user has typed a space character (in which case the QAC system can assume the word is complete). From the last prefix of Figure 2, `'pi'` may match both `'pi'` and `'pie'`, `'pillow'`, `'pit'`, `'pitlochry'` and so on; whereas in the second scenario, it would only match to `'pi'`. For some of our experiments where stopwords were removed, we used Fox's stopwords list [14]. The total number of unrolled query prefixes for each question set after this preprocessing were: 24, 989 for UQV100; 1, 128, 913 for MS MARCO; and 890, 293 for SQuAD.

3.3 Baselines

We considered two key baselines for this study. Both are detailed below, where we explain the strengths and weaknesses for each of the different approaches.

Bhatia et al. [4] We attempted to benchmark the effectiveness of our approach against the algorithm described by Bhatia et al. [4]. In this, however, sadly we failed — not because the algorithm is not implementable, or even that web-scale representations of the required data structures are too large (although we did use a server with half a terabyte of RAM), but because the algorithm's time complexity is prohibitively slow.

One example query (from the UQV100 test collection) suffices to illustrate the problem: "*best deal raspberry pi computer*". The substring prefixes for this query include: `'best'`; `'best deal'`; `'best deal raspberry'`; and `'best deal raspberry pi'`. Considering just this last query prefix, the last term of that prefix is `'pi'`. Equation (9) of [4] requires that we find all possible word completions of this term, and then find all n-gram phrases which include any of these word completions. There are 82, 774 words in the CW12Titles vocabulary that begin with the characters `'pi'` (for example, `'pillows'`), and approximately 5.54 million n-gram phrases containing at least one of these words (for example, `'contour pillows on alibaba'`). Each of these, when combined with the query prefix, becomes a candidate query completion (for example, `'best deal raspberry contour pillows on alibaba'`). Finding all of these candidates and various ordered list and hashtable and hashset data structures for the n-grams and vocabulary data took approximately 2 minutes running on a single core of our high performance server.

Then in order to compute equation (12) of the Bhatia et al. [4] algorithm, which ensures that completion phrases make sense in the context of the query prefix, we need to find the document counts for both these 5.54 million phrases and the combined query prefix and candidate phrase terms (which effectively doubles the number to 11.08 million phrases). As the authors remark, this can be approximated by simply finding the count of documents which contain all the terms of the phrase (or the combined prefix plus phrase), which can be done efficiently treating the phrases as queries and using the postings lists of a search engine index. This operation, running on 10 cores of the same server, such that the CW12 titles index was entirely memory resident, and using a commercial state-of-the-art search processing engine with a custom document-count-only operation scanning the postings lists, took on average 17 milliseconds per phrase. For the 11.08 million phrases, this equates to over 5 hours of processing time for this step alone, dwarfing the time for the first phase of the algorithm.

The final phase of the algorithm is to combine the various score components to produce a final score for the candidate (noting that any combined query prefix plus completion which has no occurrences in the CW12 titles index scores zero immediately), and rank the set of non-zero candidates by this score. This step can be done relatively efficiently. Nevertheless, the total time for making suggestions for one query prefix only exceeds 5 hours.

At over 5 hours per query – and even considering just a *single* query prefix per query in the UQV100 collection of 5, 675 queries – this would require over 1200 days of processing time. For the full set of 24, 989 unrolled query prefixes for UQV100, this expands to more than 14 years of processing time. If we added the query prefixes for SQuAD and MARCO, our total evaluation time would be more than 1000 years. Clearly, the Bhatia et al. [4] algorithm is not practical either as an evaluation baseline to compare against our approach or for making suggestions to users at they type since no one would wait 5 hours for a suggestion - a typical QAC system in production is expected to show suggestions in a time of $100ms - 150ms$. We leave a big-O analysis of the data structure and algorithmic complexity of the Bhatia et al. algorithm to motivated readers.

NextWord A simpler baseline involves predicting just a single next word given the last part of the current prefix. We used an existing large n-gram corpus and probability statistics derived from the titles of web documents crawled in 2013 by the Bing search engine. (Similar probability statistics could be obtained from the CW12 corpus — we used the one available from Bing solely as a matter of convenience.) The last two words of the current prefix (or one word if the prefix was only a single term) was provided as the search key over this corpus and the most probable single term completions then calculated from the n-gram probabilities. The top ten completions from these were amalgamated as suffixes to the current prefix as the suggestions. We refer to this algorithm as *NextWord*; it is akin to a simple smartphone keyboard suggestion service.

Since we only provide two words as the n-gram search key, for prefixes that are three or more terms long, there is no guarantee that the suggestions will make sense in the full context of the prefix. Equally, there is no guarantee that the suggestions would actually lead to documents being retrieved for the suggestion. Both of these are strong requirements in a production QAC setting; we did not impose these on the NextWord baseline on the grounds of simplicity.

3.4 Suggestion Evaluation Process

As per other QAC works, we adopt the MRR evaluation process. This is applied for each unrolled query up to 10 suggestions per query, approximating current commercial QAC suggestion list lengths. Then, if the complete query Q_0 is suggested, our approach is awarded $1/rank$, where $rank$ is the position of the suggestion in the list of suggestions; and 0 if the target query Q_0 is not present as a suggestion. These are known as *complete suggestions*.

Since many of the questions in the evaluation sets are much longer than conventional web queries, we provided a second evaluation process where a system is also rewarded for a partial advance from the current query prefix – a *partial suggestion*. For example, if the query prefix is `'best deal'`, and suggestions provided include `'best deal raspberry pi'`, then it will be awarded for a partial match if there is no complete query suggestion for `'best deal raspberry pi computer'`. We report on both the combined total of partial and complete suggestions, and the complete suggestions separately. For short prefixes and long target queries, it is often not possible to generate a complete suggestion since we restrict the language model to a maximum of quad-grams.

4 EXPERIMENTS AND RESULTS

Our results are presented to address each research question outlined in Section 1. Unless explicitly stated, our results are averaged over all questions in a question set, at all partial prefixes from the unrolled question. On our high performance server, over the CW12 titles index and the UQV100 question set with up to 24 documents, we achieve average elapsed times of 58 milliseconds per prefix to complete all parts of the generation and ranking activities for a set of suggestions. While not optimised, this still leaves time available for roundtrip user interface-to-server network latencies, suggesting that a production system could be built successfully. We provide only NextWord as a baseline for comparison due to the complexities of the other algorithm proposed by Bhatia et al. [4]. We acknowledge that other state-of-the-art *query log-based* QAC approaches do exist; however, we argue that these are not applicable for generating suggestions for target queries where such queries are unavailable (as is assumed here).

4.1 The Impact of Stopwords

RQ1: How does retaining or dropping stopwords from the query prefix when finding matching documents impact suggestion performance? While the majority of queries have relatively few stopwords, both longer queries and natural language questions have more stopwords present. Our first investigation examined the effect of removing stopwords from the query prefix, before sending to the query processor for matching. This experiment involved the modification of our system's query dispatcher pipeline component (refer to Section 3.1) to allow for the selective dropping of stopwords. The results comparing stopword-retained and stopword-dropped approaches are shown in the first two rows of Table 1, across the three question sets used. The maximum number of matching documents was set to 24.

Table 1: Average MRR for combined partial and complete suggestion performance over the three different question sets, varying by stopword retention, dropping, and combined. (Parenthesised) values are the average MRR for complete suggestions only.

Stopwords	UQV100	MARCO	SQuAD
Retained	0.0145 (0.0026)	0.0093 (0.0005)	0.0010 (0.0001)
Dropped	0.0138 (0.0035)	0.0055‡ (0.0013)	0.0015‡ (0.0001)
Combined	0.0174‡ (0.0038)	0.0122‡ (0.0014)	0.0022‡ (0.0001)

Our second stopword investigation concerns the idea of combining the dropped stopwords versus retained stopwords approaches, to examine if the combination yielded a performance improvement. Effectively, this requires two queries to be sent to the index for each prefix (unless the stopword-dropped prefix is an empty string), and so doubles the processing effort by the query processor. For each of the two unique queries issued (stopword-dropped and stopword-retained), the query processor was again set to return a maximum of 24 documents. This resulted in two lists of passages, one for each query. The lists were merged, removing duplicate entries appearing in the second of the lists. Note that this approach might lead to merged lists that were longer than the 24 document maximum for an individual query. The results from this approach are shown in the third row of Table 1. From this, we can see that by average MRR, performance is best when stopwords are retained for the Web query sets (UQV100 and MARCO), but that it is better to drop stopwords for the SQuAD question set. We do not have a convincing explanation for these differences, but do observe that the SQuAD question set consists of well-formed natural language questions (with many more function words on average), while UQV100 and MARCO are less well formed in a linguistic sense (aimed at search engine use). Despite this however, if we combine the two approaches together, the system performs better than either individual approach. Statistical significance between the partial plus complete scores across all prefixes is compared using two-tailed independent samples Student's t-tests, and shown with † for $p < 0.5$, and ‡ for $p < 0.01$ relative to the row above.

If we look instead at the percentage of unrolled query prefixes for which we have at least one correct partial or complete suggestion at any position, we see something different. Figure 3 plots the results of our three different experimental setups over the UQV100 question set, with prefix term counts on the x-axis, and the percentage of unrolled query prefixes of that count for which there was at least one correct partial or complete suggestion. The plot shows a notable improvement for the combined approach in prefix term counts from 2 until 7, and that at prefix term counts less than

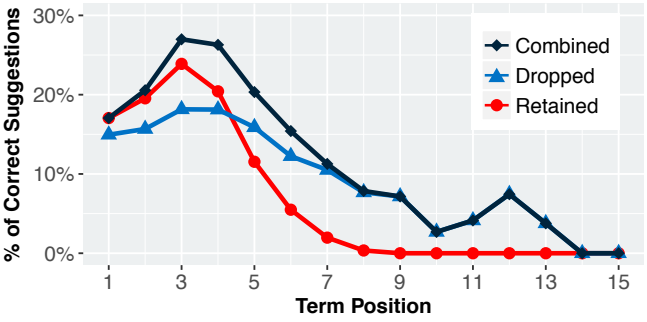

Figure 3: Plot of percentages of at least one correct partial or complete suggestion for prefixes of a given length (x-axis) by different stopword approaches over the UQV100 question set.

5, the *Retained* approach outperforms the *Dropped* approach. Dropping stopwords starts to have better performance than *Retained* for prefix term counts of 5 or greater, and from prefix term counts of 9 or more, the *Retained* approach finds no correct suggestions, so the *Combined* and *Dropped* approaches are identical from thereon.

4.2 Changing the Number of Matched Passages

RQ2: Does increasing the set of matched passages to generate the language model increase the probability of creating a matching QAC suggestion? To address this research question, we ran a series of experiments where we varied the maximum number of matching results returned by the query processor, given a query. We used values of: 8, 16, and 24. Table 2 details the maximum performance attained over each configuration, for each question set. For all questions sets, as we increase the number of documents to form the language model, the overall performance continues to improve. For UQV100, we also considered 100 and 1000 documents, and performance continues to improve still further, but the rate of improvement slows down. As before, maximum performance is achieved where the prefix is in the range 2 – 4 terms, peaking at 3, regardless of the number of documents being used.

4.3 Re-Ranking Candidates

RQ3: Can performance of the generative QAC suggestions be improved by re-ranking? To investigate the final research question, we obtained the neural embedding scoring model from the authors of [24]. These authors had access to large scale web search engine query log data and we used their model without modifications. Refer to Mitra and Craswell [24] for a detailed explanation of how the model was produced. We should also note that in a organizations where query logs were unavailable, using this approach would be impossible. Our goal in using the model was to establish if any re-ranking would offer potential gains to effectiveness.

Rather than considering all possible suffixes of a prefix as explored by Mitra and Craswell [24] – which like the approach by Bhatia et al. [4] would be prohibitively expensive over large-scale corpora – we instead supplied only the candidate completions from our generative model. We then used their neural model to score the pair of query prefix and candidate completion, and used that to re-rank our candidates. The average MRR numbers for the combined suggestion performance for 24 matching documents over the three question sets are shown in the *ReRank24* row of Table 2. These results, almost three times better than our existing base model, demonstrate a significant possibility for improving the average MRR score of our candidate completions using a re-ranking model. In an enterprise corpus absent of large query logs, a neural model of the form described in [24] might be trained instead on anchor text or document titles, to establish the common completions. The mean execution time of ReRank24 using our pipeline was 120ms, which would still be acceptable for modern QAC systems.

Table 2: Average MRR for combined partial and complete suggestion performance over the three different question sets, varying the maximum number of matched documents. (Parenthesised) values denote the average MRR for complete suggestions only.

Matches	UQV100	MARCO	SQuAD
8	0.0089 (0.0029)	0.0052 (0.0010)	0.0014 (0.0001)
16	0.0144‡ (0.0036)	0.0092‡ (0.0013)	0.0019‡ (0.0001)
24	0.0174‡ (0.0038)	0.0122‡ (0.0014)	0.0022‡ (0.0001)
100	0.0243‡ (0.0048)	—	—
1000	0.0261† (0.0050)	—	—
ReRank24	0.0509‡ (0.0101)	0.0313‡ (0.0030)	0.0047‡ (0.0003)
NextWord	0.0879‡ (0.0299)	0.0586‡ (0.0149)	0.0274‡ (0.0054)

As a strong baseline, the final row (*NextWord*) in Table 2 reports performance of a simple next word algorithm (like a predictive keyboard on a mobile phone) from a Web-derived n-gram model. Effectively, it predicts the next word that might be typed, given the previous two words as context. Unlike our approach, there is no guarantee that a suggestion will lead to a query that can surface matching documents from the corpus. It is also limited to predicting just a single word; our approach produces successful partial or complete suggestions that are two or more words longer than the current prefix in 10% of successful cases. However, the performance of the *NextWord* algorithm is impressively strong. In an enterprise environment, the Web n-grams may perform less well; however, it indicates that in the absence of suggestions from our prefix-specific language model approach, the global Web background model could successfully be used as backfill.

5 DISCUSSION AND CONCLUSIONS

In this paper, we have described and explained an approach to QAC that is generative; that is, it does not rely on existing query log data. Unlike the algorithm described by Bhatia et al, our method is able to respond in a timeframe that would be acceptable in interactive user interfaces as required by QAC systems, while operating over large scale data. The key to achieving this performance was using a search index to drastically pre-prune the candidate texts used in populating the n-gram language model. The short text document search index was `AND`-matched against all complete and partial terms of the query prefix, rather than the Bhatia et al approach of

considering all possible completions of the partial term and post-pruning candidates that do not match the other prefix terms.

We found (**RQ1**) that dropping stopwords from query prefixes can assist in situations where natural language queries are being used; and that combining both dropped and retained approaches improves performance regardless of query type. This outcome may be due to an interplay between natural language queries and that dropping stopwords in (partial) queries may lead to a language model that is less *'noisy'* than its stopword-retained counterpart, but we do not have perfect understanding of these factors.

As we increase the number of matching documents used to construct the language model from which to generate suggestions, performance also increases (**RQ2**); this is an intuitive result. A greater number of documents will undoubtedly present the system with more terms from which to generate potential suggestion completions, yielding greater effectiveness.

While efficient, our naïve algorithm for ranking candidates from this single-phase method does not lead to great MRR results. Addressing **RQ3**, a two-phase method – where the candidates are first generated, and then re-ranked using a more expensive model – improves effectiveness over a single-phase approach, costs more in time, but remains within acceptable performance limits. Even so, the improved effectiveness of the two-phase method still fails to beat the much simpler *NextWord* baseline as measured by MRR over the three query sets. Other methods to improve calculating the quality of the generated completions, such as those described by Chen and Goodman [9] and/or using the web *n*-gram corpus probabilities could also be investigated as future work.

The dramatic difference (often a factor of 5 to 10 times) in partial and complete suggestion performance indicates that the use of predicting complete suggestions as the sole success measure for long or natural language queries may be inappropriate. MRR also fails to measure factors like making nonsensical recommendations (e.g. `'best deal raspberry pillows'`).

Efficient, generative QAC could be highly beneficial in search environments where query logs are unavailable or limited. At present, our recommendation would be to use a simple *NextWord*-style algorithm using a large scale *n*-gram corpus, possibly supplemented by personal *n*-grams at the client-side or enterprise-specific *n*-grams if deployed within an enterprise.

Acknowledgments David (lead author) would like to express his gratitude to Peter, Dave, Paul and Nick for his time as an intern at Microsoft Australia in Canberra, ACT. He learnt heaps, and realised just how amazing Australia is. *"You beauty, mate!"* We also gratefully thank Bhaskar Mitra and Nick Craswell for providing us with their query completion neural model for re-ranking experiments, and Ricky Loynd and Yongen Gong for providing us with the web n-gram prediction model for effective simple baselines. Finally, we thank the anonymous reviewers for their helpful feedback.

REFERENCES

[1] P. Bailey, A. Moffat, F. Scholer, and P. Thomas. 2016. UQV100: A test collection with query variability. In *Proc. SIGIR*. 725–728.

[2] Z. Bar-Yossef and N. Kraus. 2011. Context-sensitive Query Auto-completion. In *Proc. WWW*. 107–116.

[3] H. Bast and I. Weber. 2006. Type Less, Find More: Fast Autocompletion Search with a Succinct Index. In *Proc. SIGIR*. 364–371.

[4] S. Bhatia, D. Majumdar, and P. Mitra. 2011. Query Suggestions in the Absence of Query Logs. In *Proc. SIGIR*. 795–804.

[5] F. Cai and M. de Rijke. 2016. Learning from homologous queries and semantically related terms for query auto completion. *IP&M* 52, 4 (2016), 628–643.

[6] F. Cai and M. de Rijke. 2016. A Survey of Query Auto Completion in Information Retrieval. *Foundations and Trends in Information Retrieval* 10, 4 (2016), 273–363.

[7] F. Cai, S. Liang, and M. de Rijke. 2014. Time-sensitive Personalized Query Auto-Completion. In *Proc. CIKM*. 1599–1608.

[8] J. Carbonell and J. Goldstein. 1998. The Use of MMR, Diversity-based Reranking for Reordering Documents and Producing Summaries. In *Proc. SIGIR*. 335–336.

[9] Stanley F. Chen and Joshua Goodman. 1996. An Empirical Study of Smoothing Techniques for Language Modeling. In *Proc. ACL*. 310–318.

[10] S. Cucerzan and E. Brill. 2004. Spelling Correction as an Iterative Process that Exploits the Collective Knowledge of Web Users. In *Proc. EMNLP*. 293–300.

[11] G. Di Santo, R. McCreadie, C. Macdonald, and I. Ounis. 2015. Comparing Approaches for Query Autocompletion. In *Proc. SIGIR*. 775–778.

[12] H. Duan and B-J. Hsu. 2011. Online Spelling Correction for Query Completion. In *Proc. WWW*. 117–126.

[13] A. Feuer, S. Savev, and J.A. Aslam. 2007. Evaluation of Phrasal Query Suggestions. In *Proc. CIKM*. 841–848.

[14] C. Fox. 1989. A Stop List for General Text. *SIGIR Forum* 24, 1-2 (1989), 19–21.

[15] G. W. Furnas, T. K. Landauer, L. M. Gomez, and S. T. Dumais. 1987. The Vocabulary Problem in Human-system Communication. *CACM* 30, 11 (1987), 964–971.

[16] David Hawking and Kathy Griffiths. 2013. An Enterprise Search Paradigm Based on Extended Query Auto-completion: Do We Still Need Search and Navigation?. In *Proc. ADCS*. 18–25.

[17] K. Hofmann, B. Mitra, F. Radlinski, and M. Shokouhi. 2014. An Eye-tracking Study of User Interactions with Query Auto Completion. In *Proc. CIKM*. 549–558.

[18] Shengyue Ji, Guoliang Li, Chen Li, and Jianhua Feng. 2009. Efficient interactive fuzzy keyword search. In *Proc. WWW*. ACM, 371–380.

[19] J-Y. Jiang, Y-Y. Ke, P-Y. Chien, and P-J. Cheng. 2014. Learning User Reformulation Behavior for Query Auto-completion. In *Proc. SIGIR*. 445–454.

[20] M. Kamvar and S. Baluja. 2007. The Role of Context in Query Input: Using Contextual Signals to Complete Queries on Mobile Devices. In *Proc. MobileHCI*. 405–412.

[21] R. Kraft and J. Zien. 2004. Mining Anchor Text for Query Refinement. In *Proc. WWW*. 666–674.

[22] L. Li, H. Deng, A. Dong, Y. Chang, H. Zha, and R. Baeza-Yates. 2015. Analyzing User's Sequential Behavior in Query Auto-Completion via Markov Processes. In *Proc. SIGIR*. 123–132.

[23] B. Mitra. 2015. Exploring Session Context Using Distributed Representations of Queries and Reformulations. In *Proc. SIGIR*. 3–12.

[24] B. Mitra and N. Craswell. 2015. Query Auto-Completion for Rare Prefixes. In *Proc. CIKM*. 1755–1758.

[25] B. Mitra, M. Shokouhi, F. Radlinski, and K. Hofmann. 2014. On User Interactions with Query Auto-completion. In *Proc. SIGIR*. 1055–1058.

[26] T. Nguyen, M. Rosenberg, X. Song, J. Gao, S. Tiwary, R. Majumder, and L. Deng. 2016. MS MARCO: A Human Generated MAchine Reading COmprehension Dataset. In *Proc. Cognitive Computation workshop, NIPS*.

[27] The Lemur Project. 2012. *The ClueWeb12 Dataset*. www.lemurproject.org/clueweb12.php

[28] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang. 2016. SQuAD: 100,000+ questions for machine comprehension of text. In *Proc. EMNLP*. 2383–2392.

[29] M. Shokouhi. 2013. Learning to Personalize Query Auto-completion. In *Proc. SIGIR*. 103–112.

[30] M. Shokouhi and K. Radinsky. 2012. Time-sensitive Query Auto-completion. In *Proc. SIGIR*. 601–610.

[31] B. Smyth, E. Balfe, J. Freyne, P. Briggs, M. Coyle, and O. Boydell. 2004. Exploiting query repetition and regularity in an adaptive community-based web search engine. *User Modeling and User-Adapted Interaction* 14, 5 (2004), 383–423.

[32] S. Whiting and J. Jose. 2014. Recent and Robust Query Auto-completion. In *Proc. WWW*. 971–982.