Generation of Synthetic Query Auto Completion Logs

Unni Krishnan[1][0000−0003−4838−8158], Alistair Moffat[1][0000−0002−6638−0232], Justin Zobel[1][0000−0001−6622−032X], and Bodo Billerbeck[2][0000−0002−9311−8504]

[1] The University of Melbourne, Australia
[2] Microsoft Australia and RMIT University, Australia

Abstract. Privacy concerns can prohibit research access to large-scale commercial query logs. Here we focus on generation of a synthetic log from a publicly available dataset, suitable for evaluation of query auto completion (QAC) systems. The synthetic log contains plausible string sequences reflecting how users enter their queries in a QAC interface. Properties that would influence experimental outcomes are compared between a synthetic log and a real QAC log through a set of side-by-side experiments, and confirm the applicability of the generated log for benchmarking the performance of QAC methods.

1 Introduction

Query auto completion (QAC) systems offer a list of completions while users enter queries in a search interface. Users can either submit one of the completions as their *query*, or *advance* their *partial query* by selecting a completion and then continuing to type [33]. A detailed QAC log capturing the sequence of partial queries, along with the completions presented and the user interactions with them, is required in order to evaluate a QAC system [37, 38]. However, concerns about the privacy of query logs and regulatory requirements such as GDPR mean that there is a need for alternative ways of obtaining logs for academic purposes.

Here we explore a framework for generating synthetic QAC logs, extending the work of Krishnan et al. [33], who suggest converting a QAC log to an abstracted format (an *abstract QAC log*) that records only the length of each partial query and the lengths of words used, minimizing privacy concerns but removing the possibility of performing evaluations on actual strings. Synthetic QAC log generation seeks to produce a list of *plausible* synthetic partial query sequences by mapping the word lengths from the abstract QAC log to strings from a publicly available dataset. An example of the proposed process is shown in Figure 1. On the left are partial queries typed by a user. The abstracting process converts these to the digit sequences shown in the middle column, describing the strings but not their characters; and then the corresponding synthesized strings are shown at the right. Note that it is neither necessary nor sufficient for the synthetic log to contain semantically valid phrases. Comparison between the original and synthetic logs across a range of properties show that the synthetic

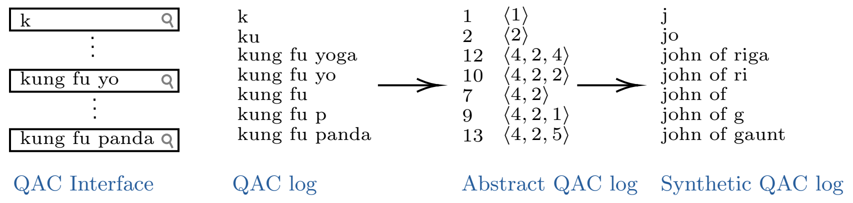

Fig. 1: A sample QAC log entry, the corresponding abstract QAC log entry, and a synthetic QAC log entry generated from the string "`john of gaunt`".

log can be used to evaluate QAC system performance. Moreover, the synthetic log eliminates the privacy concerns associated with the original.

2 Background

A QAC system retrieves a *candidate set* matching the partial query P, drawing from a target string collection, with strings in the target collection having an associated *score*. Query Auto Completion systems typically match P against past queries from a log; or, in the absence of logs, they can also be synthesized [14, 40]. Methods of ranking the candidates include static popularity [9], search context [9, 29], forecast popularity [15], personalized ranking parameters [15, 29, 42], and diversity [16]. It is also possible to choose an initial candidate set based on popularity and then apply a second ranking criteria to obtain the final strings [15, 16]. QAC implementation strategies vary based on how the partial query P is matched against the target strings [32]. A common approach is to use a trie [3, 4, 25, 27] to retrieve candidates that have P as a prefix; or inverted index-based approaches [10, 11, 23] that offer completions independent of the ordering of the words in the partial query. The functionality of a QAC system can be extended beyond character level matches by including contextual cues [11] or synonyms [12, 28]. Error-tolerant QAC approaches [17, 28, 36, 47] allow up to a fixed number of character mismatches to account for possible typing errors.

User interactions are a key factor in implementation and evaluation of QAC systems [26, 33, 38] and have been captured using a wide range of models [31, 32, 33, 37, 38, 43, 44]. In particular, users are not limited to entering single characters, and can alter the partial query by selecting a completion or deleting characters already entered. Until now, the *test collections* used to evaluate traditional search systems have been anonymized commercial logs [1, 19, 34, 48] or synthesized logs [5, 30, 45, 49]. QAC system evaluations have typically been performed over large publicly available string collections [10, 14, 23], with strings taken sequentially from left-to-right to generate partial query sequences [10]. However this approach does not account for the full range of possible interactions [32, 33]. In this work, we explore an approach to generation of synthetic partial query sequences that addresses this gap.

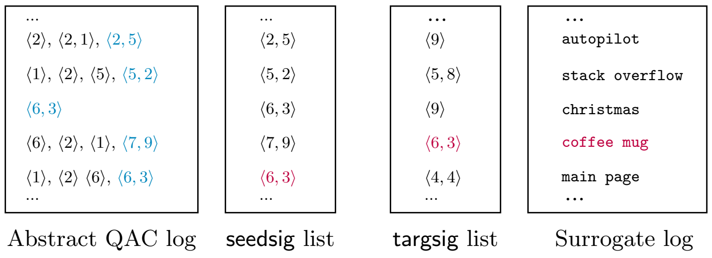

Fig. 2: The synthetic pattern generation process. Signatures of FinalP from the abstract QAC log are used to form the seedsig list, and the signatures from surrogate log form targsig list. A match between the signature $\langle 6, 3\rangle$ in the seedsig list and in the targsig list might correspond to the target string "`coffee mug`".

Terminology. A partial query P is the string currently displayed in the search box. An *interaction* updates P and results in loggable changes. A new *conversation* starts when the user begins a query, and continues until either explicitly terminated by the user or as a result of a session timeout. The last partial query from a conversation is referred to as FinalP. A QAC log records a set of conversations as a sequence of partial queries. A *surrogate log* is a dictionary of strings that can be used as substitutes for final partial queries, with each such string having an associated score reflecting its popularity. For a string T, the ordered tuple $\langle |\mathsf{w}_1|, |\mathsf{w}_2| \ldots |\mathsf{w}_k|\rangle$ representing the lengths of its words $\mathsf{w}_1, \mathsf{w}_2, \ldots \mathsf{w}_k$ in T is referred to as its *signature*. For each partial query P, an abstract QAC log records only its signature and its length $|\mathsf{P}|$, including whitespace. The signature of each FinalP in the abstract QAC log is referred to as seedsig. The signature of a string in the surrogate log is its targsig.

Problem definition. For each conversation in the abstract QAC log, find a *target string* in the surrogate log with the same signature as the final partial query FinalP. Then, starting from the first interaction in the conversation, apply the word lengths from the abstract QAC log to the target string in order to obtain a plausible partial query sequence. For example, consider the last conversation in Figure 2, with its signature sequence $\langle 1\rangle, \langle 2\rangle, \langle 6\rangle, \langle 6, 3\rangle$. The seedsig for this conversation is $\langle 6, 3\rangle$. The string "`coffee mug`" in the surrogate log has the same signature and hence might be selected as a target string. Mapping the signature sequence from the conversation, we get the synthetic partial query sequence "`c`", "`co`", "`cof`", "`coffee`", and then "`coffee mug`".

3 Generation Process

Depending on the distribution of word lengths in the surrogate log, for every final partial query in the abstract QAC log, there may not be a string having the same signature. Moreover, strings are not entered by the users in the word order of the target collection. For instance, a user looking for the Wikipedia main page might enter the queries "`wikipedia`", "`main page wikipedia`", or "`wiki`". We narrow down the possible ways of matching a seedsig with the list of signatures in targsig list to the following hierarchical *modes*:

1. *Exact.* The targsig is equal to the seedsig. For example, the seedsig $\langle 3, 3, 2\rangle$ only matches with the target signature $\langle 3, 3, 2\rangle$. There might be zero, or multiple strings in the surrogate log that match.
2. *Prefix.* The seedsig is a prefix of the targsig. For example, seedsig $\langle 3, 3, 2\rangle$ matches $\langle 3, 3, 2, 4\rangle$ and $\langle 3, 3, 2, 4, 7\rangle$, but not $\langle 9, 3, 3, 2\rangle$.
3. *Match by Drop* (MbD). The seedsig is an ordered subset of the targsig, For example, $\langle 3, 3, 2\rangle$ matches $\langle 3, 4, 3, 2\rangle$ and $\langle 4, 3, 1, 3, 6, 2\rangle$, but not $\langle 3, 4, 2, 3\rangle$.
4. *Bag of Numbers* (BoN). Relaxing the ordering requirement, a BoN match occurs if the sets of word lengths in seedsig are a subset of those in targsig. For example, seedsig $\langle 3, 3, 2\rangle$ matches $\langle 2, 3, 3, 4\rangle$ and $\langle 5, 2, 3, 6, 3\rangle$, but not $\langle 3, 4, 2, 6\rangle$. In a BoN match the target string's words are reordered to match the seed signature ordering.

Locating target strings. The first step locates, for a given seedsig, a set of matching signatures in the targsig list. The set of target candidates is maintained in lexicographically sorted order, so that the exact-match signatures for a given seedsig can be found via two binary searches, establishing a range $[\mathsf{r}_{beg}, \mathsf{r}_{end})$. A similar process can be used to find the prefix-match range, which is a larger contiguous block in the lexicographically sorted array of target candidates.

If a target signature S_t matches with seedsig using MbD, any targsig having S_t as a prefix will also have an MbD match with seedsig. For example, under MbD the seedsig $\langle 3, 5\rangle$ matches $\langle 3, 2, 5\rangle$. Then signatures $\langle 3, 2, 5, 4\rangle$ and $\langle 3, 2, 5, 9, 16\rangle$ will also be a match because they have $\langle 3, 2, 5\rangle$ as a prefix. Using this property, once a matching signature S_t for MbD is found, we can add the prefix match range for S_t to the MbD range for the current seedsig. In contrast to exact and prefix match ranges, MbD ranges may not form a continuous range over SigList.

For BoN matching, the tokens in the signature are sorted to form a canonical representation. For example, $\langle 3, 2, 4\rangle$ becomes $\langle 2, 3, 4\rangle$. This list is sorted into lexicographical order. A BoN match between the canonical representations of a seedsig and a target signature S_{tc} can be verified by a linear scan over the tokens. The matching ranges for BoN in the modified SigList are calculated by finding prefix match ranges for each S_{tc} that matches with seedsig using BoN.

Handling deletions. To include deletions, we assume that something different was initially typed (the *replacement* word) and was converted to the word from the target string after the deletions. In particular, suppose that the current conversation contains the deletion of a word w_k starting from the i th interaction and ending with the j th interaction. Then, for every interaction preceding the j th interaction, w_k is substituted by a *replacement* word constructed via a set of deletion heuristics. If character sequences are deleted and re-entered several times during the conversation, several replacement words will be required. The last replacement word should be close to the original word, and the penultimate

Sequence	$\|\mathsf{P}\|$	Signature	Modified target string	Synthetic P
1	12	$\langle 5,6\rangle$	"`black mirror`"	"`black mirror`"
2	11	$\langle 5,5\rangle$	"`black mirror`"	"`black mirro`"
3	5	$\langle 5\rangle$	"`black mirror`"	"`black`"
4	6	$\langle 5\rangle$	"`black hole`"	"`black `"
5	7	$\langle 5,1\rangle$	"`black hole`"	"`black h`"
6	8	$\langle 5,2\rangle$	"`black hole`"	"`black ho`"
7	7	$\langle 5,1\rangle$	"`black hole`"	"`black h`"
8	8	$\langle 5,2\rangle$	"`black hawk down`"	"`black ha`"
9	15	$\langle 5,4,4\rangle$	"`black hawk down`"	"`black hawk down`"

Table 1: Deriving synthetic partial query sequence for a conversation containing multiple word deletions. The fourth column shows three target strings used.

replacement prior to it should similarly be a modification of the last replacement word. Table 1 gives an example of the partial query sequence generated by rewriting target strings when the conversation contains multiple deletions.

Users delete characters either due to typing errors [6, 20] made within that word, or to switch to an entirely different word. Among various typing errors that can occur in any character level entry systems [22], the following error categories that are predominantly discussed in past research [6, 20, 21, 46] are included in the generation process.

Deletion A deletion error occurs when the user initially missed out one of the characters in the word, for example, in the correction "`acount`" $\rightarrow$ "`account`". Deletion errors are more frequent at character repetitions, and occur more commonly at the beginning of a word [6]. To simulate a deletion error, if there is a deletion sequence from i th interaction to the j th interaction in which more than two characters of $\mathtt{w}_k$ are removed, then $\mathtt{w}_k$ from the $j+1$ th interaction (after the deletions) is examined to see if the last two characters in $\mathtt{w}_k$ are repeating. The error is then simulated by deleting one of the repeating characters of $\mathtt{w}_k$ in the interactions prior to $j+1$.

Insertion These arise when an extra letter is initially typed, for example, the correction "`sherriff`" $\rightarrow$ "`sheriff`". Our experimental results show that only a low fraction of insertion errors are observed in a QAC log and for that reason, they are not included in the synthetic QAC log generation process.

Substitution These arise when the user enters one of the neighboring keys instead of the intended key, for example, the correction "`disturv`" $\rightarrow$ "`disturb`". Single character deletions are assumed to be substitution errors. The substituted character is found via a probability distribution for mistyped keys around the current key.

Transposition This is when the user swaps two characters in a word either not knowing the correct spelling or because of mis-ordered keystrokes, for example, the correction "`wierd`" $\rightarrow$ "`weird`". Deletions of length two are considered to be transposition errors and the last two characters of the word are swapped to get the replacement word.

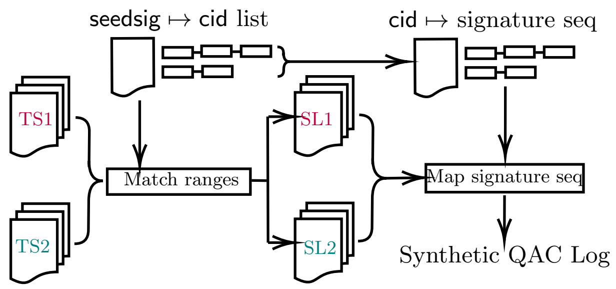

Fig. 3: Framework for generating a synthetic QAC log. See the text for details.

In a QAC interface, typing errors may not be the only reason why users delete characters. Sometimes users replace certain parts of a well-formed partial query, for instance to get a different set of completions. In such cases, referred as *multichar* deletions, the corrected word will differ from the original word in two or more character positions. To allow for such cases, the deleted part of the word is replaced with another string starting with the word's remaining prefix, but differing in the next character. For example, consider the word "`hawk`" in Table 1. A deletion chain removed all of the characters except the "`h`"; and hence (working backwards), "`hawk`" was replaced by an alternate word that starts with "`h`" but not "`ha`", such as "`hole`".

Generation framework. The overall framework for generation of a synthetic QAC log is illustrated in Figure 3. The seed signatures are stored with a mapping from signatures to the list of cids having the same signature. Signatures from the surrogate log are precomputed and stored in lexicographically sorted order (TS1). Additionally, the strings in the surrogate log are re-ordered based on TS1 ordering to obtain the permutation SL1. The permutations TS2 stores a sorted list of within sorted signatures to support BoN matching, and SL2 is the corresponding permutation of the surrogate log. To generate a synthetic QAC log entry, a signature is selected from seedsig $\mapsto$ cid list and a list of target signatures retrieved based on the four modes. Using the mappings SL1 and SL2, strings from the surrogate log corresponding to the target signatures are then obtained.

The strings are converted to target strings by aligning the word lengths with the seedsig. A target string can be generated from multiple originals. If, say, seedsig is $\langle 9, 4\rangle$, then "`wikipedia main page`" gives the target string "`wikipedia page`" using MbD; and another string "`personal web page wikipedia`" gives the same target using BoN mode. If the same target string is produced by more than one string from the surrogate log, then the one with lower score is discarded.

The signature sequences obtained from the file providing the cid $\mapsto$ signature sequence mapping are then applied to the list of target strings, to obtain a synthetic partial query sequence for each conversation. Partial query sequences are generated for each cid from the current seedsig $\mapsto$ cid list until either the cid list or the target strings are exhausted. Finally, after the generation process is completed, the synthetic QAC log is re-ordered so that the conversations in the synthetic QAC log have one-to-one correspondence with the abstract QAC log.

4 Experiments

Datasets used. The Wikipedia clickstream dataset,[3] generated from Wikipedia request logs containing tuples of the form (*referrer*, *resource*, *frequency*), is used to generate the surrogate log. The number of requests for an article (*resource*) is the *frequency*. Data dumps from January to March 2019 were aggregated by updating the *frequency* of each *resource* by the mean *frequency* over the dumps. We refer to the resulting dataset as `Wiki-Clickstream` and the synthetic QAC log generated from `Wiki-Clickstream` as `Wiki-Synth`.

A QAC log was formed by randomly sampling the logs from Bing[4] QAC system over a period of one week from 13 August 2018. This `Bing-QAC-2018` records the partial queries entered by the users and a unique `cid` for each conversation. The abstracted version of `Bing-QAC-2018`, referred to as `Bing-Abs-QAC-2018`, is generated by recording the signature and total length of each partial query along with a unique `cid` for each conversation. Note that `Wiki-Synth` was generated from `Bing-Abs-QAC-2018` only (with no use of `Bing-QAC-2018`), and that the subsequent comparisons between `Wiki-Synth` and `Bing-QAC-2018` were performed on secure Microsoft servers and in accordance with Microsoft privacy requirements.

Preprocessing. Conversations in `Bing-Abs-QAC-2018` where an intermediate signature contained more words than the signature of the final partial query, or contained words that were longer than the corresponding words in the final partial query, were removed. The strings from `Bing-QAC-2018` were *transliterated* from UTF-8 to ASCII encoding using `Unidecode`[5] Python package. Conversations containing non-converting characters in any partial query were discarded. The strings from `Wiki-Clickstream` were transliterated in the same way and non-converting strings were removed. The conversion from UTF-8 to ASCII resulted in the loss of 0.26 million conversations from `Bing-QAC-2018` and 0.02 million strings from `Wiki-Clickstream`. After the preprocessing, there were 1.44 million conversations in `Bing-Abs-QAC-2018` and 5.11 million strings in `Wiki-Clickstream`. In `Bing-Abs-QAC-2018`, 7.53% of signatures were unique, of which 2.26% had no matches in `Wiki-Clickstream`. A total of 0.23 million conversations from `Bing-Abs-QAC-2018` were not included in `Wiki-Synth` because there were not enough matching strings in `Wiki-Clickstream` to map their signature sequence.

Sampling of target strings. The frequency of search queries has been found to exhibit a power law distribution [8] with the probability distribution $p(x) \propto x^{-\alpha}$. Therefore, while generating a synthlog, the target strings need to be sampled

[3] https://meta.wikimedia.org/wiki/Research:Wikipedia_clickstream, accessed 29th October, 2019.

[4] https://www.bing.com, accessed 29th October, 2019.

[5] https://pypi.org/project/Unidecode, accessed 29th October, 2019.

to obtain a power law distribution based on their frequencies. Strings in the `Wiki-Clickstream` log have a score that tends to follow a power law distribution with $\alpha = 3.09$. Using this observation we sample target strings based on a weighted probability distribution over their scores, so that the probability of a string T given a seed signature $\mathsf{seedsig}$ is

$$Prob(\mathsf{T} \mid \mathsf{seedsig}) = \frac{score(\mathsf{T})}{\sum_{i=1}^{n} score(\mathsf{T}_i)} \tag{1}$$

where $\mathsf{T}_1, \mathsf{T}_2, \ldots, \mathsf{T}_n$ are the set of target strings retrieved for $\mathsf{seedsig}$. The frequency distribution of the resulting target strings used as the FinalP in `Wiki-Synth` is analyzed below, where we compare partial query frequencies.

Language model for finding replacement words. For deletion types other than multichar, the replacement word will be a modification of the original word based on the error category. For multichar deletions, the heuristics discussed above are used to find replacement words. The *best* replacement word based on contextual information among the candidate words that passes the heuristics is selected using a 4-gram language model (LM) trained on the title strings from `Wiki-Clickstream`. The language model is generated using KenLM [24],[6] which is based on modified Kneser-Ney smoothing and provides fast model construction and querying. For example, an LM-based replacement for the target string "`live queen`", is "`live together`" while a random replacement gives "`live teufelshorner`"; for the target string "`web server`", a random replacement yields "`web castelvetere`" and an LM-based replacement gives "`web content`". A similar character bigram model trained on the Microsoft spelling-correction dataset[7] is used to find the most likely next mistyped character, given the previous character, to simulate substitution errors.

Comparison of synthetic QAC logs with QAC logs. A synthetic QAC log should have some properties similar to the QAC log, and these can be used to validate the generation process. A comparison of these properties is given in Table 2. Other properties are desirable if a synthetic QAC log is to be a substitute in experiments for a QAC log. While comparing these properties, the partial queries from both the logs are treated as the *test queries* that will be queried against a collection of strings acting as the *test collection*. Extending our assumption that a user's goal was to enter FinalP, we claim that these strings could get eventually indexed by the QAC system and subsequently be queried. Thus, in our experiments, the list of FinalP strings are considered as a representative sample of a larger hypothetical test collection.

Heap's law. Heap's law gives an estimate of the number of unique terms V in a collection as a function of the number of terms N. The relationship is given

[6] https://github.com/kpu/kenlm, accessed 29th October, 2019.

[7] https://www.microsoft.com/en-us/download/details.aspx?id=52418, accessed 29th October, 2019.

Attribute	`Bing-QAC-2018`	`Wiki-Synth`
Number of conversations (millions)	1.74	1.21
Number of interactions (millions)	11.75	7.90
Percentage of unique partial queries	38.99	39.41
Percentage of unique final partial queries	52.30	63.32
Lengths of partial queries – mean (characters)	10.37	9.67
– std.dev. (characters)	9.68	8.53

Table 2: Basic statistics of `Bing-QAC-2018` and `Wiki-Synth`.

Frequency distribution	Exponential		Log-normal	
	R	p	R	p
`Bing-QAC-2018`, P	11 885.4	<0.01	0.04	0.80
`Wiki-Synth`, P	10 895.3	<0.01	0.16	0.15
`Bing-QAC-2018`, FinalP	5903.6	<0.01	−2.42	0.04
`Wiki-Synth`, FinalP	1211.9	0.01	0.12	0.04

Table 3: Goodness of power law fit against other distributions.

by $V = kN^\beta$ and the typical values of the parameters are $30 \leq k \leq 100$, with $\beta \approx 0.50$ for English text [39] and $\beta \approx 0.60$ for web documents [7]. The values for Heap's law parameters estimated from `Bing-QAC-2018` are $k = 14.50$, $\beta = 0.67$ and the parameters from `Wiki-Synth` are $k = 32.61$, $\beta = 0.57$. The difference in growth rate can be explained by the nature of the logs. The growth rate of V is expected to be higher in a collection containing numbers and spelling errors [39]. Each of the FinalP strings from `Wiki-Synth` is a Wikipedia title that comes from a curated collection whereas FinalP strings in `Bing-QAC-2018` are strings entered by users. Considering, all partial queries, the parameters estimated are $k = 5.54$, $\beta = 0.71$ for `Bing-QAC-2018` and $k = 20.94$, $\beta = 0.60$ for `Wiki-Synth`.

Frequency of partial query strings. The likelihood of a query being repeated over time as a result of the power-law distribution affects the performance of various query processing strategies [13, 35]. The frequency distributions of partial queries and final queries from `Wiki-Synth` and `Bing-QAC-2018` are given in Figure 4 (left) with dashed lines showing the power-law fits for the corresponding distributions. The distributions give similar exponents except for the distribution of FinalP from `Wiki-Synth`, which has $\alpha = 3.02$, indicating a steeper decay in the probability of higher frequency partial query strings. The goodness of power-law fit against exponential and log-normal fits is compared using likelihood ratio R [2, 18] along with the significance level p, as reported in Table 3. Except for the frequency distribution of FinalP from `Bing-QAC-2018`, which is better explained with a log-normal distribution ($R = -2.42, p = 0.04$), the other three distributions suit a power-law distribution, perhaps as a result of the sampling process followed. For some datasets, it is not surprising to get a better log-normal fit [18].

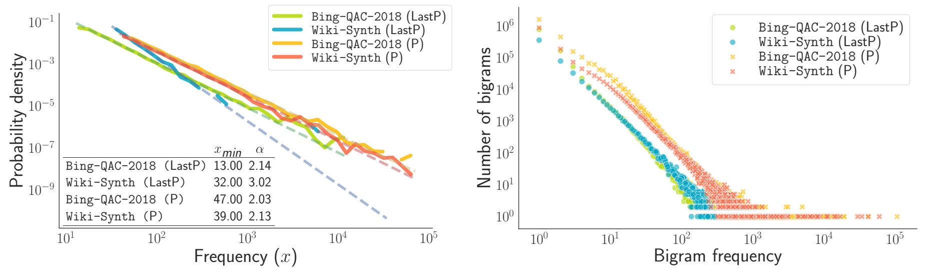

Fig. 4: Power law fit (left) and bigram frequency distribution (right).

N-gram frequency. The distribution of terms in a collection can be modeled using a decay law which estimates the collection frequency of the i th most common term as $\mathsf{F}_i = Ci^k$, where $k = -1$ and C is constant. Similarity in term distribution with a real log is considered to be a desirable property of a synthetic log [45]. Figure 4 (right) shows the bigram frequency distribution from partial queries and final partial queries. We find close correspondence between the bigram frequency distributions (Kolmogorov-Smirnov statistic, $D = 0.05, p = 0.63$ for FinalP and $D = 0.06, p = 0.04$ for all partial queries). Unigram frequencies follow a similar trend. While the bulk of the distribution can be explained with the decay law, the tail of the distributions coming from rare n-grams show deviations.

Empirical entropy. The empirical entropy of the FinalP strings gives a measure of their compressibility. The k th-order empirical entropy of a string $\mathsf{T}[1 \ldots n]$ over an alphabet set Σ is given by

$$\mathsf{H}_k(\mathsf{T}) = \frac{1}{n} \sum_{w \in \Sigma^k} |\mathsf{T}_w| \cdot \mathsf{H}_0(\mathsf{T}_w)\,, \text{ with } \mathsf{H}_0(\mathsf{T}) = \frac{1}{n} \sum_{c \in \Sigma} n_c \cdot \log \frac{n}{n_c}\,, \tag{2}$$

where T_w is formed by collecting the characters that immediately follow *context* w in T, and where n_c is the frequency of character c in T. A lower bound on the number of bits required to encode T is given by $n\mathsf{H}_k(\mathsf{T})$ [41]. For a list of strings $\mathsf{T}_1, \mathsf{T}_2, \ldots, \mathsf{T}_l$, it can be observed that $\sum_i n_i \mathsf{H}_k(\mathsf{T}_i) \leq n_c \mathsf{H}_k(\mathsf{T}_c)$ where T_c is the concatenation of strings T_i. This can be shown by extending the proof given by Navarro [41, Chapter 2], applying Jensen's inequality to $\sum_i n_i \mathsf{H}_k(\mathsf{T}_i)$. Therefore, we consider $n_c \mathsf{H}_k(\mathsf{T}_c)$ as the worst case lower bound for the space required to encode the FinalP strings from `Bing-QAC-2018` and `Wiki-Synth`. The values of $\mathsf{H}_k(\mathsf{T}_c)$ for $k = 0 \ldots 4$ computed from `Bing-QAC-2018` and `Wiki-Synth` are given in Table 4. The values for H_k tend to be similar between the two logs. Slightly higher values of H_k and increased C_t/n for `Bing-QAC-2018` can be explained by the higher degree of randomness expected from web search queries, compared to the results obtained for Heap's law coefficients.

Identification of typing errors. Similarities in typing patterns between the synthetic QAC log and the original QAC log are verified by analyzing the typing

Dataset	$k=0$		$k=1$		$k=2$		$k=3$		$k=4$	
	H_0	C_t/n	H_1	C_t/n	H_2	C_t/n	H_3	C_t/n	H_4	C_t/n
`Bing-QAC-2018`	4.39	0.00	3.64	0.00	2.50	0.17	1.41	2.51	0.64	11.27
`Wiki-Synth`	4.28	0.00	3.43	0.00	2.32	0.08	1.27	0.97	0.54	4.95

Table 4: Values of H_k and number of contexts C_t per length n ($\times 10^{-3}$) of the concatenated string formed by the FinalP strings of `Bing-QAC-2018` and `Wiki-Synth`.

Inter.	Partial query	Word len.	Edit dist.
2	`coffee pa`	2	
3	`coffee pal`	3	
4	`coffee pa`	2	
5	`coffee p`	1	
6	`cofee pl`	2	1
6	`coffee pla`	**3**	**1**
6	`cofee places`	6	4

Table 5: Example for pre-correction and post-correction strings. In the table "`pal`" is taken as the pre-correction string and "`pla`" is taken as the post-correction string. This correction pair will be classified as a transposition error.

errors present in both logs. Typing errors are identified by extracting and comparing a set of *pre-correction* and *post-correction* strings from the conversations, extending the method proposed by Baba and Suzuki [6]. When the k th word w_k in P gets deleted from the i th interaction, w_k from the $i-1$ th interaction is taken as the pre-correction string. To find the post-correction string, the edit distance between the pre-correction string and the words w_k from the interactions after the end of current deletion sequence are calculated until $|\mathsf{w}_k|$ from interaction j is less than that from interaction $j+1$. From this list, the word having minimum edit distance with the pre-correction string is taken as the post correction string. Table 5 gives an example for how pre-correction and post-correction strings are calculated from a deletion sequence. The edit distance between two strings is calculated as a Damerau-Levenshtein distance, so transposition of two characters is given a cost of 1. The pre-correction and post-correction pairs are then compared to classify the possible typing errors as one of the types discussed above. The results obtained by analyzing the typing errors from `Bing-QAC-2018` and the synthetic QAC log are in Figure 5. The fraction of transposition errors was found to be higher in `Wiki-Synth` compared to `Bing-QAC-2018` while the latter contains more deletion errors than `Wiki-Synth`. As a result of the method of simulating substitution errors, they tend to occur more towards the end of words in `Wiki-Synth`. The remainder of the error categories are distributed across similar word positions in both the logs, suggesting similarities in typing patterns.

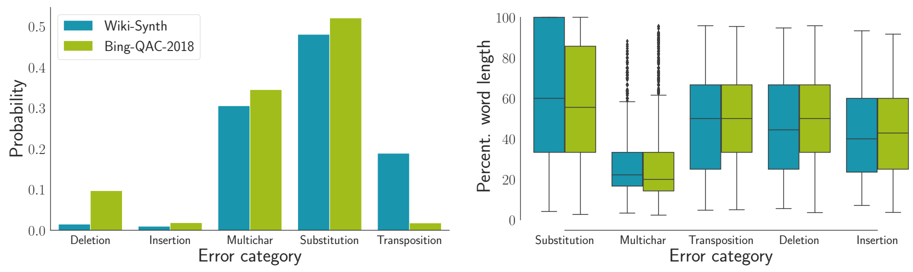

Fig. 5: Comparison between frequency (left) and position (right) of typing errors.

5 Conclusions

We have explored a method for generating a synthetic QAC log from an abstract QAC log, by mapping the word lengths of the abstract QAC log to those of a publicly available string collection, and applying a range of corrective techniques. Synthetic QAC formation can also be posed as a language generation problem relying on various models of QAC systems [31, 33].

We have demonstrated that the synthetic log generated from a pre-existing string collection encompasses many of the properties found in the original QAC log from which the abstract QAC log was derived. In particular, an analysis of typing errors found in actual QAC logs is reported, along with a description of how they were introduced into the synthetic log. As a result, there is a close correspondence between the real QAC log and the synthetic QAC log across a range of properties, each of which might influence the computational cost of providing the completion strings. That is, experiments using the synthetic QAC log can be expected to provide close approximations to behaviors that would be observed on a real QAC log. A particular example is efficiency, which we plan to examine in future work, comparing the performance of a range of QAC implementations using both synthetic and real QAC logs.

Acknowledgments. This work was supported by the Microsoft Research Centre for Social Natural User Interfaces (SocialNUI) at The University of Melbourne. We are grateful to Peter Bailey (Microsoft Australia) for his support of that project, and for additionally facilitating the work that is reported here.

Bibliography

[1] E. Adar. User 4xxxxx9: Anonymizing query logs. In *Proc. WWW Query Log Analysis Wrkshp.*, 2007. URL http://www.cond.org/anonlogs.pdf.

[2] J. Alstott, E. Bullmore, and D. Plenz. powerlaw: A Python package for analysis of heavy-tailed distributions. *PLOS One*, 9(1):1–11, 2014.

[3] N. Askitis and R. Sinha. HAT-trie: A cache-conscious trie-based data structure for strings. In *Proc. Aust. Comp. Sc. Conf.*, pages 97–105, 2007.

[4] N. Askitis and J. Zobel. Redesigning the string hash table, burst trie, and BST to exploit cache. *ACM J. Exp. Alg.*, 15:1–7, 2010.

[5] L. Azzopardi, M. de Rijke, and K. Balog. Building simulated queries for known-item topics: An analysis using six European languages. In *Proc. SIGIR*, pages 455–462, 2007.

[6] Y. Baba and H. Suzuki. How are spelling errors generated and corrected? A study of corrected and uncorrected spelling errors using keystroke logs. In *Proc. ACL*, pages 373–377, 2012.

[7] R. Baeza-Yates and F. Saint-Jean. A three level search engine index based in query log distribution. In *Proc. SPIRE*, pages 56–65, 2003.

[8] R. Baeza-Yates and A. Tiberi. Extracting semantic relations from query logs. In *Proc. KDD*, pages 76–85, 2007.

[9] Z. Bar-Yossef and N. Kraus. Context-sensitive query auto-completion. In *Proc. WWW*, pages 107–116, 2011.

[10] H. Bast and I. Weber. Type less, find more: Fast autocompletion search with a succinct index. In *Proc. SIGIR*, pages 364–371, 2006.

[11] H. Bast and I. Weber. The CompleteSearch engine: Interactive, efficient, and towards IR & DB integration. In *Proc. CIDR*, pages 88–95, 2007.

[12] H. Bast, D. Majumdar, and I. Weber. Efficient interactive query expansion with complete search. In *Proc. CIKM*, pages 857–860, 2007.

[13] S. M. Beitzel, E. C. Jensen, A. Chowdhury, D. Grossman, and O. Frieder. Hourly analysis of a very large topically categorized web query log. In *Proc. SIGIR*, pages 321–328, 2004.

[14] S. Bhatia, D. Majumdar, and P. Mitra. Query suggestions in the absence of query logs. In *Proc. SIGIR*, pages 795–804, 2011.

[15] F. Cai, S. Liang, and M. de Rijke. Time-sensitive personalized query auto-completion. In *Proc. CIKM*, pages 1599–1608, 2014.

[16] F. Cai, R. Reinanda, and M. de Rijke. Diversifying query auto-completion. *ACM Trans. Inf. Sys.*, 34(4):25:1–25:33, 2016.

[17] S. Chaudhuri and R. Kaushik. Extending autocompletion to tolerate errors. In *Proc. SIGMOD*, pages 707–718, 2009.

[18] A. Clauset, C. R. Shalizi, and M. E. J. Newman. Power-law distributions in empirical data. *SIAM Review*, 51(4):661–703, 2009.

[19] A. Cooper. A survey of query log privacy-enhancing techniques from a policy perspective. *ACM Trans. Web*, 2(4):19:1–19:27, 2008.

[20] F. J. Damerau. A technique for computer detection and correction of spelling errors. *Comm. ACM*, 7(3):171–176, 1964.

[21] V. Dhakal, A. M. Feit, P. O. Kristensson, and A. Oulasvirta. Observations on typing from 136 million keystrokes. In *Proc. CHI*, pages 646:1–646:12, 2018.

[22] D. R. Gentner, J. T. Grudin, S. Larochelle, D. A. Norman, and D. E. Rumelhart. *A glossary of terms including a classification of typing errors*, pages 39–43. Springer, 1983.

[23] D. Hawking and B. Billerbeck. Efficient in-memory, list-based text inversion. In *Proc. Aust. Doc. Comp. Symp.*, pages 5.1–5.8, 2017.

[24] K. Heafield. KenLM: Faster and smaller language model queries. In *Proc. Wrkshp. Statistical Machine Translation*, pages 187–197, 2011.

[25] S. Heinz, J. Zobel, and H. Williams. Burst tries: A fast, efficient data structure for string keys. *ACM Trans. Inf. Sys.*, 20(2):192–223, 2002.

[26] K. Hofmann, B. Mitra, F. Radlinski, and M. Shokouhi. An eye-tracking study of user interactions with query auto completion. In *Proc. CIKM*, pages 549–558, 2014.

[27] B.-J. P. Hsu and G. Ottaviano. Space-efficient data structures for top-k completion. In *Proc. WWW*, pages 583–594, 2013.

[28] S. Ji, G. Li, C. Li, and J. Feng. Efficient interactive fuzzy keyword search. In *Proc. WWW*, pages 371–380, 2009.

[29] J. Jiang, Y. Ke, P. Chien, and P. Cheng. Learning user reformulation behavior for query auto-completion. In *Proc. SIGIR*, pages 445–454, 2014.

[30] C. Jordan, C. Watters, and Q. Gao. Using controlled query generation to evaluate blind relevance feedback algorithms. In *Proc. JCDL*, pages 286–295, 2006.

[31] E. Kharitonov, C. Macdonald, P. Serdyukov, and I. Ounis. User model-based metrics for offline query suggestion evaluation. In *Proc. SIGIR*, pages 633–642, 2013.

[32] U. Krishnan, A. Moffat, and J. Zobel. A taxonomy of query auto completion modes. In *Proc. Aust. Doc. Comp. Symp.*, pages 6:1–6:8, 2017.

[33] U. Krishnan, B. Billerbeck, A. Moffat, and J. Zobel. Abstraction of query auto completion logs for anonymity-preserving analysis. *Inf. Retr.*, pages 1–26, 2019.

[34] R. Kumar, J. Novak, B. Pang, and A. Tomkins. On anonymizing query logs via token-based hashing. In *Proc. WWW*, pages 629–638, 2007.

[35] R. Lempel and S. Moran. Predictive caching and prefetching of query results in search engines. In *Proc. WWW*, pages 19–28, 2003.

[36] G. Li, S. Ji, C. Li, and J. Feng. Efficient fuzzy full-text type-ahead search. *Proc. VLDB*, 20(4):617–640, 2011.

[37] L. Li, H. Deng, A. Dong, Y. Chang, H. Zha, and R. Baeza-Yates. Analyzing user's sequential behavior in query auto-completion via Markov processes. In *Proc. SIGIR*, pages 123–132, 2015.

[38] Y. Li, A. Dong, H. Wang, H. Deng, Y. Chang, and C. Zhai. A two-dimensional click model for query auto-completion. In *Proc. SIGIR*, pages 455–464, 2014.

[39] C. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval.* Cambridge University Press, 2008.

[40] D. Maxwell, P. Bailey, and D. Hawking. Large-scale generative query auto-completion. In *Proc. Aust. Doc. Comp. Symp.*, pages 9:1–9:8, 2017.

[41] G. Navarro. *Compact Data Structures: A Practical Approach.* Cambridge University Press, 2016.

[42] M. Shokouhi. Learning to personalize query auto-completion. In *Proc. SIGIR*, pages 103–112, 2013.

[43] C. L. Smith, J. Gwizdka, and H. Feild. Exploring the use of query auto completion: Search behavior and query entry profiles. In *Proc. CHIIR*, pages 101–110, 2016.

[44] C. L. Smith, J. Gwizdka, and H. Feild. The use of query auto-completion over the course of search sessions with multifaceted information needs. *Inf. Proc. & Man.*, 53(5):1139–1155, 2017.

[45] W. Webber and A. Moffat. In search of reliable retrieval experiments. In *Proc. Aust. Doc. Comp. Symp.*, pages 26–33, 2005.

[46] J. O. Wobbrock and B. A. Myers. Analyzing the input stream for character-level errors in unconstrained text entry evaluations. *ACM Trans. Computer-Human Interaction*, 13(4):458–489, 2006.

[47] C. Xiao, J. Qin, W. Wang, Y. Ishikawa, K. Tsuda, and K. Sadakane. Efficient error-tolerant query autocompletion. *Proc. VLDB*, 6(6):373–384, 2013.

[48] L. Xiong and E. Agichtein. Towards privacy-preserving query log publishing. In *Proc. WWW Query Log Analysis Wrkshp.*, 2007.

[49] J. Zobel, A. Moffat, and K. Ramamohanarao. Inverted files versus signature files for text indexing. *ACM Trans. Data. Sys.*, 23(4):453–490, 1998.