Efficient Neural Query Auto Completion

Sida Wang, Weiwei Guo, Huiji Gao, Bo Long
LinkedIn, Mountain View, California
{sidwang,wguo,hgao,blong}@linkedin.com

ABSTRACT

Query Auto Completion (QAC), as the starting point of information retrieval tasks, is critical to user experience. Generally it has two steps: generating completed query candidates according to query prefixes, and ranking them based on extracted features. Three major challenges are observed for a query auto completion system: (1) QAC has a strict online latency requirement. For each keystroke, results must be returned within tens of milliseconds, which poses a significant challenge in designing sophisticated language models for it. (2) For unseen queries, generated candidates are of poor quality as contextual information is not fully utilized. (3) Traditional QAC systems heavily rely on handcrafted features such as the query candidate frequency in search logs, lacking sufficient semantic understanding of the candidate.

In this paper, we propose an efficient neural QAC system with effective context modeling to overcome these challenges. On the candidate generation side, this system uses as much information as possible in unseen prefixes to generate relevant candidates, increasing the recall by a large margin. On the candidate ranking side, an unnormalized language model is proposed, which effectively captures deep semantics of queries. This approach presents better ranking performance over state-of-the-art neural ranking methods and reduces ~95% latency compared to neural language modeling methods. The empirical results on public datasets show that our model achieves a good balance between accuracy and efficiency. This system is served in LinkedIn job search with significant product impact observed.

KEYWORDS

query auto completion; neural language model; deep learning

ACM Reference Format:
Sida Wang, Weiwei Guo, Huiji Gao, Bo Long. 2020. Efficient Neural Query Auto Completion. In *Proceedings of the 29th ACM International Conference on Information and Knowledge Management (CIKM '20), October 19–23, 2020, Virtual Event, Ireland*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3340531.3412701

1 INTRODUCTION

Query auto completion (QAC) [5] is the standard component of search engines in industry. Given a prefix, the system returns a ranked list of completed queries that match users' intents. Query auto completion enhances user experience in two ways: (1) it saves user keystrokes and returns search results in less time; (2) it guides users to type "better" queries that are more likely to lead to desirable search results. For example, given a prefix "soft", "software engineer" is a better query than "software programmer", since the former is a more commonly used job title.

A typical QAC system takes a generate-then-rank two-step framework. The candidate generation component returns the most frequent queries that match the prefix, by memorizing the mapping from prefixes to queries based on search logs [1]. The candidate ranking component extracts features from candidates and uses them to produce the final ranking order. Both components do not involve intense computation so the whole process can be finished within several milliseconds, in order to meet online latency requirements.

However, this traditional approach does not fully exploit the context in the query prefix. For example, in the generation phase, for unseen prefixes, only the last word of prefix is used to generate candidates [27]; in the ranking phase, the most effective feature is the query frequency collected from search log, which lacks deep semantics understanding.

To enable semantic text understanding, neural networks based methods are applied to QAC. Early works [27] focus on the ranking stage: Convolutional Neural Networks (CNN) [34] are adopted to measure the semantic coherence between the query prefix (user input) and suggested suffixes in the ranking phase. Recently, an end-to-end approach for both generation and ranking [31] is proposed: a neural language model is trained to measure the probability of a sequence of tokens. During decoding, candidate generation and ranking are performed for multiple iterations during beam search [24]. While neural language models show better sequence modeling power over CNN, they could take up to 1 second [40], making productionization infeasible.

In this work, our goal is to build an QAC system with more effective query context utilization for real world search systems, while meeting the industry latency standard. We make improvements in the two-stage generate-then-rank framework: (1) In candidate generation, we extend a previous work [27] by incorporating more information from unseen prefixes to generate meaningful candidates. (2) In candidate ranking, we adopt neural language modeling, a more natural approach to model the coherence between a word and its previous sequence. To overcome the latency challenge, we optimize the latency by approximating computation of word probability with a much more efficient structure, reducing 95% latency (~55ms to ~3ms). Offline experiments on public datasets show significant improvement in terms of relevance and latency. We also train our model on the LinkedIn job search dataset and deploy it in production with CPU serving.

In summary, the contribution of this paper is

CIKM '20, October 19–23, 2020, Virtual Event, Ireland

ACM ISBN 978-1-4503-6859-9/20/10…$15.00
https://doi.org/10.1145/3340531.3412701

- We developed an effective candidate generation method that maximizes recall of clicked queries through better context utilization.
- We developed an optimized neural language model for candidate ranking, which has similar sequence modeling power as general neural language models, yet with significantly lower serving latency.
- We deploy this efficient QAC system into commercial search engines, and observe significant product impact.

2 RELATED WORK

In this session, we first introduce traditional methods for QAC, then discuss how neural networks are applied for QAC, followed by details of neural language models.

2.1 Traditional Approaches for QAC

Most of the previous works adopt a two-step framework – candidate generation and candidate ranking [5]. The former aims to increase recall, and the latter focuses on increasing precision. Candidate generation components return a list of completed queries for a given prefix. Most works are based on prefix-to-query statistics calculated from search logs. For example, Most Popular Completion (MPC) [1] directly looks up the top N most frequent historical queries that start with the entire prefix. Mitra et al. [27] extend this method and generate candidates for rare or unseen prefixes by exploiting information from the last word in the prefix. Other works investigate the problem of generating candidates directly from documents when search logs are not available [3, 25]. In the candidate generation phase, we further extend [27] to use more context to generate better candidates.

After candidates are generated, candidate ranking components compute a relevance score for each candidate. Different sources of information have been exploited to improve the ranking, including context or session information [1, 18], time/popularity-sensitivity [6, 36, 41], personalization [6, 35], user behaviors [15, 22, 28, 43], and click through logs [21]. Our paper does not use any of these additional information and focuses on the effectiveness and efficiency of QAC given general query logs. Therefore, our work is orthogonal to methods using additional information and can benefit these methods as well.

2.2 Deep Learning Approach for QAC

In order to tackle insufficient text understanding in traditional approaches, deep learning approaches are adopted in recent years.

One line of work is applying neural networks to extract semantic embeddings from queries and perform ranking. In [27], Convolutional Latent Semantic Model (CLSM) [34] is used for QAC ranking. This model applies Convolutional Neural Networks (CNN) [20] to extract an embedding for query prefix and suffix strings. Then the cosine similarity score between the prefix and suffix embedding determines the ranking.

Another line of work is applying neural language models for both generation and ranking [31, 40]. In such work, a Long Short Term Memory (LSTM) [14] based neural language model [26] is trained on complete queries. After that beam search [24] is used for decoding. In beam search, there are many iterations of generation and ranking, which yield impressive relevance performance with a large computation overhead. There are several advantages of neural language modeling. One advantage of the neural language modeling architecture is that additional features can be seamlessly incorporated. For example, personalization can be modeled by incorporating user ID embeddings [11, 16, 17] in the network. Time aware [11] and spelling errors aware [40] models are also developed under this framework. Another advantage is that language modeling is more effective at sequence coherence modeling, supported by its probabilistic interpretation $P(query) = \prod_i P(w_i|w_0 ... w_{i-1})$.

2.3 Neural Language Modeling

Neural language models measure the probability of a text sequence. Bengio et al. [2] propose a neural language model, where the probability of the next word is computed based on the embeddings of previous several words. Mikolov et al. [26] use Recurrent Neural Networks (RNN) [42] to summarize a sequence of any length into a hidden state of RNN, which generalizes the context better. Following these works are word level [37] and character level [19] neural language models.

However, these neural language modeling approaches are time consuming in both training and decoding stages – in the computation of word probability, these methods compute a costly normalization term which requires iterating over all words in the vocabulary. In order to resolve this issue, unnormalized language models [10, 13] are proposed, targeting latency reduction. The idea is to approximate the normalization over the whole vocabulary in word probability computation. Unnormalized language models have been applied on machine translation [10], speech recognition [8, 33] and word embedding pretraining [29]. We propose an efficient unnormalized language model approach for QAC ranking, and deploy it into LinkedIn's search engines.

3 AN EFFICIENT NEURAL QUERY AUTO COMPLETION SYSTEM

We propose an efficient neural query auto completion system that consists of two phases: candidate generation and candidate ranking. In candidate generation, we aim to increase recall of candidates with more context utilization. This step is finished within 1 millisecond. In candidate ranking, we design an efficient unnormalized neural language model that effectively models the query sequence. Therefore, the following content focuses on (1) how to exploit more contexts for both generation and ranking, as well as (2) how to minimize the computation cost in neural language model based ranking.

3.1 Candidate Generation

The candidate generation component returns a list of queries for a given prefix. The first step is to collect *background* data within a certain time range. The background data contain queries and their corresponding frequency computed from search logs within the time range. The second step is to build a prefix-to-query mapping based on the background data. Given a prefix, candidates are generated by looking up the mapping and choosing the top N most

Table 1: Candidate generation for the query prefix "cheapest flights from seattle to". The grayed-out words are not used in the candidate generation process, and the italicized words are the matched suffixes. The example shows that utilization of more words leads to stronger relatedness between suffixes and prefixes. *E.g.*, hinted by "flights from seattle to", more relevant suggestions like "... to sfo" are suggested. In contrast, given only the last word "to", suggestions tend to have lower quality like "airport".

cheapest flights from seattle to
cheapest *flights from seattle to sfo*
cheapest flights from *seattle to vancouver*
cheapest flights from seattle *to airport*
cheapest flights from seattle *to study*

frequent queries. The whole process can be optimized in microseconds with the Apache Finite State Transducer (FST) library.[1]

One obvious issue is that the prefix may never be seen in the *background* data. Mitra and Craswell [27] overcome this issue by exploiting **suffixes**. First, 100k most frequent suffixes are collected from the background queries. Then, for an unseen user input such as "cheapest flights from seattle to", the algorithm extracts the last word "to",[2] and uses it to match any suffixes that start with "to". Therefore, they are able to find suffixes such as "to dc", "to bermuda", *etc.*, which can be appended to "cheapest flights from seattle". We refer to this method as LastWordGeneration (LWG).

Note that two separate FSTs are built: one is built on background queries, and the other one is built on all suffix n-grams of the queries. We call these two FSTs **QueryFST** and **SuffixFST** respectively.

It's observed that the quality of retrieved suffixes is not high, when only the last word is used for matching. We believe using more contexts to match suffixes should yield more relevant results. Therefore, we extend this approach by greedily matching the longest last few words, instead of only the last one word. For the unseen prefix "cheapest flights from seattle to", we first try to find suffixes that start with "flights from seattle to", then "from seattle to", "seattle to", and finally "to". An example is shown in Table 1. By using more prefix tokens, it works effectively for recall improvement. This approach is referred to as **Maximum Context Generation (MCG)** and described in Algorithm 1.

3.2 Candidate Ranking

In this section, we focus on how to rank the generated candidates with neural networks. As shown in Figure 1, our system consists of two major components: an "unnormalized" language modeling [10, 33] layer and a pairwise learning-to-rank [4] layer. The unnormalized language model layer computes a score for a query candidate efficiently. Then learning-to-rank objective functions are applied on the scores of the clicked and non-clicked query pairs. These two components are trained together in an end-to-end fashion.

[1] https://lucene.apache.org/core/7_3_1/core/org/apache/lucene/util/fst/FST.html

[2] It could also be the last incomplete word. For example, if the prefix is "cheapest flights from seattle t", then the last incomplete word is "t".

Algorithm 1: Maximum Context Generation

Input : QueryFST F_q, SuffixFST F_s, query prefix p
Output: An ordered list of queries $qList$

```
/* 1. Get suggestions from QueryFST                    */
qList = F_q(p)
/* 2. Add suggestions from SuffixFST                   */
rw = ""                          // Keep record of removed words
while p != "" do
    // Remove first word w_0 from current prefix
    w_0 = p.remove(0)
    rw = rw + w_0
    // Prepend removed words to suggestions from SuffixFST
    sList = [rw + s for s in F_s(p)]
    qList.addAll(sList)
end while
return qList
```

To score a query, an intuitive way is neural language modeling with LSTM [31] that computes the sequence probability as the query scores, as previously discussed in Section 2.2. The log probability of the query is computed as:

$$\begin{aligned}\log P(q) &= \sum_{i}^{L} \log P(w_i | w_1, w_2, ..., w_{i-1}) \\ &= \sum_{i}^{L} \log P(w_i | h_{i-1}) \\ &= \sum_{i}^{L} \log \frac{e^{v_i^\top h_{i-1}}}{\sum_j e^{v_j^\top h_{i-1}}} \\ &= \sum_{i}^{L} \left(v_i^\top h_{i-1} - \log \sum_{j}^{N} e^{v_j^\top h_{i-1}} \right) \end{aligned} \tag{1}$$

where h_i is the hidden state of an LSTM [14] cell for word w_i, v_i is the embedding for word w_i, L is the number of words in the query and N is the vocabulary size. In this case, the hidden state h_{i-1} summarizes all information of the sequence $w_1, w_2, ..., w_{i-1}$ before w_i. It is worth noting that two special tokens <sos> and <eos> are used, as illustrated in Figure 1.

However, this natural language modeling approach is inefficient. To compute the probability of a word w_i, we need to compute the normalization term $\log \sum_j^N e^{v_j^\top h_{i-1}}$. This term involves vector multiplication between the hidden state h_{i-1} and each word in the vocabulary. Usually the size of vocabulary can be larger than 30k, which produces a large computation overhead. Therefore, to reduce latency, approximation needs to be applied on the normalization term, similar to "unnormalized" language modeling. Such approximation must satisfy several requirements:

- Computational efficiency. The approximation should not require iterating every word over the vocabulary.
- Ranking effectiveness. Under the approximation, although the absolute value of query probability $P(q)$ will be affected,

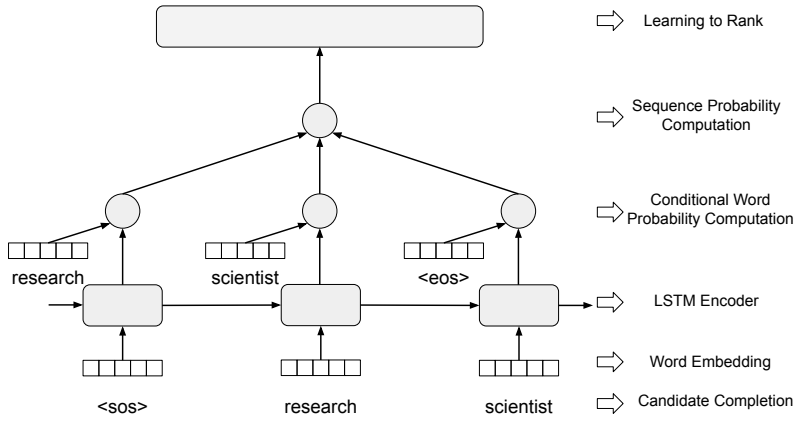

Figure 1: Our neural ranking model architecture. On top of it is a Learning-To-Rank layer that takes in multiple candidate scores. The input query has a special token "<sos> research scientist"; the probability of "research scientist <eos>" is computed based on LSTM hidden states.

it shall preserve its relative ranking position w.r.t. other queries in most cases.

- Length penalty. Long queries should receive more penalty than short ones, as short queries are more frequently typed.[3]

Therefore, we propose our design of unnormalized language modeling, where the normalization term is approximated by:

$$\log \sum_{j}^{N} e^{v_j^\top h_{i-1}} \approx b \tag{2}$$

where b is a scalar parameter to learn. Accordingly, equation 1 becomes:

$$\log P(q) \approx \sum_{i}^{L} \left(v_i^\top h_{i-1} - b\right) \tag{3}$$

This design satisfies the requirements: 1) For computational efficiency, since the approximation term is only a scalar value, latency is significantly reduced. 2) This design also keeps ranking effectiveness. More specifically, Equation 3 measures the semantic coherence of words in the query by $\sum_i v_i^\top h_{i-1}$, hence it assigns a meaningful ranking score for the query. 3) It also imposes length penalty by using a penalty term $b * L$.

With this approximation, the latency can be reduced by more than 95% in our experiments (Table 4). Similar speedup is observed in other unnormalized language model methods [10, 13].

3.3 Comparison to State-of-the-Art Models

3.3.1 Comparison to CLSM Model. CLSM is used for candidate ranking in the previous work [27]. This model applies Convolutional Neural Networks (CNN) [20] to learn an embedding for query prefix and suffix. The coherence score is the cosine similarity score between the prefix and suffix embedding. However, CNN only focuses on extracting n-gram features that is useful for ranking; it ignores the sequence length, and does not explicitly model word coherence $P(w_i|w_0...W_{i-1})$. In our approach, we use neural language models that better model the coherence between words. We train the neural language model with a learning-to-rank layer in an end-to-end approach.

3.3.2 Comparison to an End-to-End Language Modeling Approach. The end-to-end language modeling approaches [31, 40] yield great performance, since generation and ranking are performed at the same time for many iterations (one iteration for one generated token). However, it is also time-consuming. For a character level language model, ranking is required in every character generation, compared to the first-generate-then-rank framework.

3.3.3 Comparison to Existing Unnormalized Language Models. Different designs of unnormalized language models are proposed for machine translation [10, 39] and speech recognition [8, 33] to reduce computation time. In these applications, a valid probability is crucial for candidate generation. The first method is self normalization [10, 33], which adds a regularization term to the task specific loss, to make sure the sum of probability of all words is close to 1. During decoding, the normalization term is dropped to accelerate speed, assuming it is equals to 1. However, the disadvantage is that training is slow, since the normalization term needs to be computed in training to obtain the loss.

Another popular method is Noise-Contrastive Estimation [8, 13, 30, 39]. For each word, k noise words are added as negative examples. This method still requires a certain amount of the training time. Moreover, the normalization term is still computed at inference time and thus it does not meet the industrial latency requirement. In our case, our focus is the relative ranking of sentences (guided by

[3]The same pattern (short queries are more frequent) is observed in both AOL and LinkedIn datasets.

the learning-to-rank loss) and both training and inference efficiency, rather than having an accurate estimation of the word probability. Therefore, we do not compare our model with Noise-Contrastive Estimation in this work.

4 EXPERIMENT SETUP

4.1 Dataset Preparation

All experiments are conducted on the publicly available AOL dataset [32]. Similar results are observed on the LinkedIn job search dataset.

The data preprocessing follows the previous work [27]. We use the AOL data collected over three months (from March to June 2006) for experimentation. For preprocessing, we remove the empty query and keep only one copy of the adjacent identical queries. The dataset is split in the same way as in [27] – data from March 1 to April 30, 2006 are used as background data, with the following two weeks as training data, and each of the following two weeks as validation and test data. This results in 13.88 million background queries, 3.19 million training queries, 1.37 million validation queries, and 1.40 million test queries. For robustness, we exclude queries with frequency < 3 in the background data.

Mapping from prefix to queries is constructed and stored in two FSTs, QueryFST and SuffixFST, as described in Section 3.1: QueryFST is built on the background data; SuffixFST is built on the most frequent 100k suffixes in the background data. No special treatment is applied on out-of-vocabulary words such as mapping them to unknown tokens. If a word cannot be found in FSTs, no suggestion will be made.

4.2 Baseline Models in Candidate Generation

In candidate generation, MostPopularCompletion (MPC) [1] and LastWordGeneration (LWG) [27] are used as baselines. These two methods are state-of-the-art methods that focus on usage of query prefixes and do not take in additional information such as personalization and time awareness.

MostPopularCompletion (MPC): Given a query prefix p, this method searches for the most frequent k (k is the number of candidates to be ranked in candidate ranking) queries starting with p in QueryFST. QueryFST is built in the same way as in Section 4.1.

LastWordGeneration (LWG): Given a query prefix $p = w_1 w_2 ... w_n$, this method first obtains historical queries starting with p from QueryFST like MPC. After that, the last word w_n (it could be an incomplete word) is extracted and the most frequent k suffixes starting with w_n are collected from SuffixFST. Extracted suffixes are prepended with $w_1 w_2 ... w_{n-1}$ (words in prefix except w_n) to make up a suggestion. Finally candidates from QueryFST and SuffixFST are merged together. QueryFST and SuffixFST are built in the same way as in Section 4.1.

4.3 Baseline Models in Candidate Ranking

In candidate ranking, we compare our unnormalized language model to two categories of models, frequency based models and neural network models.

4.3.1 Frequency based models. Frequency based models give higher ranks to more frequent candidates from MPC, LWG and MCG. For MPC, more frequent candidates are ranked higher. For LWG, candidates from the same FST are ranked by frequency. Candidates from QueryFST are ranked higher than those from SuffixFST. For MCG, the ranking is best described by Algorithm 1.

4.3.2 Neural network models. Given results from candidate generation, neural network models generate a score for each candidate and rank the candidates by their scores. We compare our model to the state-of-the-art Convolutional Latent Semantic Model (CLSM) [34] , and implement a simple LSTM model for comparison.

CLSM: Given a sequence and its corresponding embedding matrix, this model first extracts contextual features from n-grams using convolution filters and then extracts salient n-gram features using a max pooling layer. A semantic (dense) layer is then applied on the salient n-gram features to obtain a semantic vector for the sequence. Semantic vectors for both prefix and suggested suffix are extracted. Cosine similarity between these two vectors is computed and treated as the candidate score. In our experiments, all hyperparameters follow the same setting as in [27].

LSTMEmb: The basic architecture of this model is an LSTM network. Given a word sequence, each word is fed into the LSTM cell in order. The final hidden state vector is used as the semantic representation of the sequence. Given this semantic vector, dot product is computed between the semantic representation and a learnable weight vector and used as the ranking score.

For LSTMEmb and our unnormalized language model, we choose an embedding size of 100 as a balance of performance and speed. Xavier initialization [12] is used for embeddings. The hidden state size is set to be the same as the embedding size. The AdamWeightDecay [23] optimizer is used in training with a learning rate of $2 * 10^{-3}$ and a weight decay rate of $1 * 10^{-2}$. The vocabulary size is 30k. Word hashing is not applied in vocabulary generation. Pairwise ranking is used with logistic loss in the learning-to-rank layer. The scoring model and the learning-to-rank model are jointly trained. The maximum size of a candidate list is set to 10 for each user input.

4.4 Evaluation

Model comparison is conducted in terms of relevance and efficiency. In candidate generation, We use recall to measure the performance of candidate generation methods because this metric measures the probability that users' desired queries exist in the candidate list. More specifically, recall among top 10 results is measured.

In candidate ranking, we use mean reciprocal rank (MRR) to measure the relevance performance of each candidate ranking method. MRR is computed for the top 10 candidates. Similar to the previous work [27], MRR is calculated for two groups: 1) seen prefixes, prefixes that have matches in QueryFST and 2) unseen prefixes, prefixes that cannot find matches from QueryFST and therefore completed queries only come from SuffixFST.

Latency is used as the measure of candidate generation efficiency. The time cost of ranking a candidate list with 10 candidates is measured for each model. This is the average time cost over 1000 tests. The average number of words in candidates is 3.20.

Table 2: Performance of different candidate generation methods on AOL. For each method, candidates are generated in the same order as the ranking order described in Section 4.3.1. Recall@10 is computed for all prefixes, seen prefixes and unseen prefixes separately. † indicates statistically significant improvements over LastWordGeneration through a paired t-test with $p < 0.05$.

Candidate Generation Methods	Recall@10		
	All	Seen	Unseen
MostPopularCompletion (MPC)	0.2075	0.5091	0.0000
LastWordGeneration (LWG)	0.3884	0.5207	0.2973
MaximumContextGeneration (MCG)	**0.3992**†	**0.5219**†	**0.3147**†

5 RESULTS ON AOL

5.1 Candidate Generation

Table 2 compares the performance of different candidate generation methods, namely MPC, LWG and MCG. In consideration of efficiency, there's a limit on the number of candidates to be generated and ranked. In our experiment, this limit is set to be 10. For each method, candidates are generated in the same order as the ranking order described in Section 4.3.1. Recall@10 is computed and its value is the same as the maximum MRR@10 that can be obtained in candidate ranking.

Our proposed candidate generation method MCG, has shown lift of Recall@10 over MPC and LWG both in total and in each prefix partition, with the major lift on unseen suffixes. In comparison with MPC, our methods provide a solution for rare and unseen suffixes, effectively leveraging context information of sequences following the first word of the prefix through the use of SuffixFST. In comparison with LWG, although technically MCG and LWG can produce the same candidate set, the candidate generation orders are different. Given the limit on the maximum number of candidates, this order has a large impact on QAC system performance. Our method prioritizes the generation of candidates that share more context with the user input. Therefore, MCG is able to exploit more context in user input to generate candidates with higher quality.

5.2 Candidate Ranking

Table 3 shows the performance of frequency based models, neural models and hybrid models. Frequency based models and neural models are described in Section 4.3. Hybrid models are the combination of neural and frequency based models, denoted by NN+Frequency in Table 3, where NN is a neural model. Among them, neural models only rank candidates from SuffixFST while keeping positions of candidates from QueryFST ranked by frequency.

As shown in the previous section, MCG exhibits the best candidate generation performance. Therefore, all neural ranking methods are performed on the candidates generated by MCG.

Consistent with results from the prior work [31], neural models that rank all candidates cannot outperform frequency-based methods on seen user inputs. This shows neural networks' insufficiency in memorizing strong evidence. However, all neural networks exhibit lift on unseen user inputs, showing their power in evaluating the coherence of the query and the semantic relation between suggested suffixes and words not used by SuffixFST (i.e., words "cheapest flights from" and the suffix "seattle to vancouver" in Table 1).

Based on this observation, we conduct experiments for hybrid models. In these models, we keep the ranking of candidates from QueryFST given by frequency-based methods and apply neural networks only on candidates from SuffixFST. Results show that such combination achieves the best performance, with lift not only on unseen user inputs but also on seen user inputs. Note that the lift on seen user inputs comes from the fact that for seen user inputs, there are also results from SuffixFST when QueryFST provides less than 10 results.

Among results from neural models except normalized neural language models, our neural unnormalized language model performs the best both with and without the combination with frequency-based methods. This gain comes mainly from the design of language modeling, a more effective context modeling architecture.

5.3 Latency

The latency of MCG and neural ranking methods is shown in Table 4. The time cost of MCG is almost negligible because humans generally cannot notice latency in sub-millisecond level in auto completion. This shows the efficiency of MCG in capturing user context. Since the order of generated completions from MCG follows the order of frequency (high to low), the frequency based method with MCG generation has the same latency as MCG. Therefore, the major time cost of our two-step QAC system comes from neural ranking.

Compared with CLSM, our proposed unnormalized language model takes slightly more time, but it is still at the same scale. The extra time cost comes from the sequential dependency of LSTM cell outputs – output of each LSTM cell depends on the state of the previous cell. This is not an issue for CNN as the convolution can be done in parallel. Under the current setting of candidate number and length, our model only takes 3 ms to rank candidates, indicating the unnormalized model is efficient for industrial applications.

To get a sense of the speed of normalized neural language modeling, we also implement a normalized neural language model. This model computes the exact conditional probabilities of word occurrences without any approximation. Results show that a normalized neural language model takes 17 more times than an unnormalized language model. An average latency cost of 53.32ms for model inference is hardly acceptable for an auto completion system. For a fully generation-based neural language model with beam search, the latency will be further increased by a factor of beam search size.

6 RESULTS ON LINKEDIN PRODUCTION SYSTEM

We apply the model on LinkedIn job search dataset, and conduct A/B tests with 50% traffic for two weeks. In the online system, users will receive QAC suggestions while typing. A list of job postings will be retrieved after user queries are submitted. Users can then view the postings and apply for jobs.

Figure 2 investigates the impact of the embedding size/hidden size and the layer number of LSTM on model inference speed and relevance performance. Even with the embedding size and the hidden vector size up to 500, the latency of our model is still lower

Table 3: Performance of different candidate ranking models on AOL. Frequency based models are applied on each generation method. For each neural model, two settings are performed: (1) use the network to rank all candidates; (2) only use the neural network to rank candidates generated by SuffixFST (noted by "model-variant"+Frequency). For methods involving neural networks, percentage lift is computed related to CLSM and CLSM + Frequency respectively. † indicates statistically significant improvements over CLSM and ‡ indicates statistically significant improvements over CLSM + Frequency through a paired t-test with $p < 0.05$.

Generation	Ranking	MRR@10		
		All	Seen	Unseen
MPC	Frequency	0.1805	0.4431	0.0000
LWG	Frequency	0.3147	0.4465	0.2241
MCG	Frequency	0.3283	0.4469	0.2467
	CLSM	0.3270	0.4229	0.2610
	LSTMEmbed	0.3278† (+0.244%)	0.4224	0.2628†
	UnnormalizedLM	0.3328† (+1.769%)	0.4293†	0.2665†
	NormalizedLM	0.3331† (+1.865%)	0.4293†	0.2669†
	CLSM + Frequency	0.3369	0.4472	0.2610
	LSTMEmbed +Frequency	0.3379‡ (+0.297%)	0.4472	0.2628‡
	UnnormalizedLM +Frequency	**0.3402‡ (+0.980%)**	**0.4473**	**0.2665‡**
	NormalizedLM +Frequency	**0.3404‡ (+1.039%)**	**0.4473**	**0.2669‡**

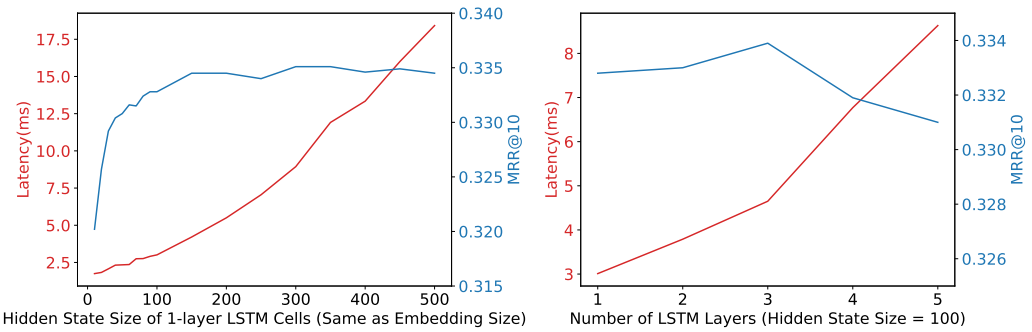

Figure 2: Latency & MRR regarding LSTM hidden size (left) and latency & MRR regarding LSTM layer number (right).

than half of the latency of a normalized language model. Therefore, compared to normalized language models, under the same latency requirement, our unnormalized language modeling design can support more advanced models like Transformers [38] which have a better context understanding capability.

Dataset Preparation: We use one-month click through data from the LinkedIn job search engine to conduct experiments. The data splitting (background, training, validation and test data) and FST construction are done in the same way as that in AOL.

Baseline System: The baseline system follows a two-step ranking framework, with MCG as the candidate generation method and XGBoost [7] as the candidate ranking method. Multiple features are included in the XGBoost ranking model, such as frequency of suggested queries from background query logs, a Kneser–Ney smoothing language model score of the suggested queries, etc.

Metrics: We measure the performance in two aspects: (1) the impact on query auto completion component, by **QAC CTR@5** (the click through rate of the top 5 suggested completions); (2) the impact on the overall job search results, by **Job Views** (number of

Table 4: Latency of different models. The average time cost of ranking a candidate list with 10 candidates is measured for each model. The average number of words in candidates is 3.20. The hidden vector size and embedding size of LM is 100 and the LSTM layer number is 1. CLSM parameters are the same as in [27]. This test is conducted on a Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz machine with 6 cores and 64-GB memory.

Methods	Latency
MaximumContextGeneration	0.18ms
CLSM	2.15ms
Unnormalized LM	3.01ms
Normalized LM	53.32ms

Table 5: Job Search Online Results. All 3 metrics are statistically significant (p < 0.05).

Metric	Lift
QAC CTR@5	+0.68%
Job Views	+0.43%
Job Apply Clicks	+1.45%

jobs that users view through job search) and **Job Apply Clicks** (number of jobs that users apply through job search).

Online Results: We performA/B tests between model *UnnormalizedLM +Frequency* and the baseline system as shown in Table 5. Since the baseline system uses MCG in candidate generation as well, the focus of online experiments is on comparing the performance of a hand-crafted feature based model to an unnormalized neural language model. The QAC CTR@5 metric lift indicates that the quality of query auto completion is improved. The Job Views/Job Apply Clicks metric lifts show that more relevant job postings are retrieved because more meaningful queries are issued. The model is ramped to 100% traffic at LinkedIn's English job search.

7 CONCLUSIONS AND FUTURE WORK

In this paper, we propose an efficient neural QAC system that captures contextual information in both candidate generation and ranking given general logs. Our method is orthogonal to models that use personalized information such as user search history.

On the candidate generation side, more context words in query prefixes are utilized, resulting in more relevant candidates, especially for unseen prefixes. On the candidate ranking side, an unnormalized language model is proposed, which enables real-time deep semantic understanding of queries.

Besides its success in offline experiments, this system has been applied on the LinkedIn platform with great success. This technology not only saves user effort by suggesting queries related to users' intent, but also helps users better reach their goals by providing queries that are more likely to retrieve desirable results.

In the future, we would like to explore acceleration techniques like [40]. This direction is of great importance because it enables more advanced NLP techniques in an industrial QAC system. E.g., normalized neural language generation, an end-to-end QAC system with high recall [31], can be productionized. More advanced semantics encoders such as BERT [9] can be used as well.

REFERENCES

[1] Ziv Bar-Yossef and Naama Kraus. 2011. Context-sensitive query auto-completion. In *WWW*.
[2] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. 2003. A neural probabilistic language model. *JMLR* (2003).
[3] Sumit Bhatia, Debapriyo Majumdar, and Prasenjit Mitra. 2011. Query suggestions in the absence of query logs. In *SIGIR*.
[4] Chris Burges, Tal Shaked, Erin Renshaw, Ari Lazier, Matt Deeds, Nicole Hamilton, and Greg Hullender. 2005. Learning to rank using gradient descent. *ICML*.
[5] Fei Cai and Maarten De Rijke. 2016. A survey of query auto completion in information retrieval. *Foundations and Trends® in Information Retrieval* (2016).
[6] Fei Cai, Shangsong Liang, and Maarten De Rijke. 2014. Time-sensitive personalized query auto-completion. In *CIKM*.
[7] Tianqi Chen and Carlos Guestrin. 2016. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. ACM.
[8] Xie Chen, Xunying Liu, Mark JF Gales, and Philip C Woodland. 2015. Recurrent neural network language model training with noise contrastive estimation for speech recognition. In *ICASSP*. IEEE.
[9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).
[10] Jacob Devlin, Rabih Zbib, Zhongqiang Huang, Thomas Lamar, Richard Schwartz, and John Makhoul. 2014. Fast and Robust Neural Network Joint Models for Statistical Machine Translation. *ACL*.
[11] Nicolas Fiorini and Zhiyong Lu. 2018. Personalized neural language models for real-world query auto completion. In *NAACL*.
[12] Xavier Glorot and Yoshua Bengio. 2010. Understanding the difficulty of training deep feedforward neural networks. In *AISTATS*.
[13] Michael Gutmann and Aapo Hyvärinen. 2010. Noise-contrastive estimation: A new estimation principle for unnormalized statistical models. In *AISTATS*.
[14] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* (1997).
[15] Kajta Hofmann, Bhaskar Mitra, Filip Radlinski, and Milad Shokouhi. 2014. An eye-tracking study of user interactions with query auto completion. In *CIKM*.
[16] Aaron Jaech and Mari Ostendorf. 2018. Personalized Language Model for Query Auto-Completion. In *ACL*.
[17] Danyang Jiang, Wanyu Chen, Fei Cai, and Honghui Chen. 2018. Neural Attentive Personalization Model for Query Auto-Completion. In *IAEAC*.
[18] Jyun-Yu Jiang, Yen-Yu Ke, Pao-Yu Chien, and Pu-Jen Cheng. 2014. Learning user reformulation behavior for query auto-completion. In *SIGIR*.
[19] Yoon Kim, Yacine Jernite, David Sontag, and Alexander M Rush. 2016. Character-aware neural language models. In *AAAI*.
[20] Yann LeCun and Yoshua Bengio. 1995. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks* (1995).
[21] Liangda Li, Hongbo Deng, Anlei Dong, Yi Chang, Ricardo Baeza-Yates, and Hongyuan Zha. 2017. Exploring Query Auto-Completion and Click Logs for Contextual-Aware Web Search and Query Suggestion. In *WWW*.
[22] Yanen Li, Anlei Dong, Hongning Wang, Hongbo Deng, Yi Chang, and ChengXiang Zhai. 2014. A two-dimensional click model for query auto-completion. In *SIGIR*.
[23] Ilya Loshchilov and Frank Hutter. 2019. Decoupled Weight Decay Regularization. In *ICLR*.
[24] Bruce T. Lowerre. 1976. The HARPY speech recognition system.
[25] David Maxwell, Peter Bailey, and David Hawking. 2017. Large-scale generative query autocompletion. In *ADCS*.
[26] Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. 2010. Recurrent neural network based language model. In *InterSpeech*.
[27] Bhaskar Mitra and Nick Craswell. 2015. Query auto-completion for rare prefixes. In *CIKM*.
[28] Bhaskar Mitra, Milad Shokouhi, Filip Radlinski, and Katja Hofmann. 2014. On user interactions with query auto-completion. In *SIGIR*.
[29] Andriy Mnih and Koray Kavukcuoglu. 2013. Learning word embeddings efficiently with noise-contrastive estimation. In *NIPS*.
[30] Andriy Mnih and Yee Whye Teh. 2012. A fast and simple algorithm for training neural probabilistic language models. *arXiv preprint arXiv:1206.6426* (2012).
[31] Dae Hoon Park and Rikio Chiba. 2017. A neural language model for query auto-completion. In *SIGIR*.
[32] Greg Pass, Abdur Chowdhury, and Cayley Torgeson. 2006. A picture of search.. In *InfoScale*, Vol. 152.
[33] Abhinav Sethy, Stanley Chen, Ebru Arisoy, and Bhuvana Ramabhadran. [n. d.]. Unnormalized exponential and neural network language models. In *ICASSP*.
[34] Yelong Shen, Xiaodong He, Li Deng Jianfeng Gao, and Grégoire Mesnil. 2014. Learning semantic representations using convolutional neural networks for web search. In *WWW*.
[35] Milad Shokouhi. 2013. Learning to personalize query auto-completion. In *SIGIR*.
[36] Milad Shokouhi and Kira Radinsky. 2012. Time-sensitive query auto-completion. In *SIGIR*.
[37] Martin Sundermeyer, Ralf Schlüter, and Hermann Ney. 2012. LSTM neural networks for language modeling. In *InterSpeech*.
[38] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *NIPS*.
[39] Ashish Vaswani, Yinggong Zhao, Victoria Fossum, and David Chiang. 2013. Decoding with large-scale neural language models improves translation. In *EMNLP*.
[40] Po-Wei Wang, Huan Zhang, Vijai Mohan, Inderjit S. Dhillon, and J. Zico Kolter. 2018. Realtime query completion via deep language models. In *SIGIR eCom*.
[41] Stewart Whiting and Joemon M Jose. 2014. Recent and robust query auto-completion. In *WWW*.
[42] Ronald J Williams and David Zipser. 1989. A learning algorithm for continually running fully recurrent neural networks. *Neural computation* 1, 2 (1989).
[43] Aston Zhang, Amit Goyal, Weize Kong, Hongbo Deng, Anlei Dong, Yi Chang, Carl A Gunter, and Jiawei Han. 2015. adaqac: Adaptive query auto-completion via implicit negative feedback. In *SIGIR*.