Ranking User Attributes for Fast Candidate Selection in Recommendation Systems

Huichao Xue
LinkedIn Corporation
Sunnyvale, California, USA
huichao@linkedin.com

ABSTRACT

Many recommendation systems use users' attributes to retrieve documents before ranking. Instead of using all attributes, this work explores algorithms that choose a subset, in order to achieve higher precision. We propose a model that forecasts the relevance of documents matched by each individual attribute. By restricting to top-K attributes based on the forecast, we observed 50% reduction in latency at 99th percentile on LinkedIn's job recommendation system, as well as increased users' engagements.

CCS CONCEPTS

• **Information systems** → **Query reformulation**; **Personalization**; *Computational advertising*.

KEYWORDS

recommendation system; deep learning; scalability; information retrieval

ACM Reference Format:
Huichao Xue. 2020. Ranking User Attributes for Fast Candidate Selection in Recommendation Systems. 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/10.1145/3340531.3412742

1 INTRODUCTION

Many real-world recommendation systems tackle the scalability challenge with a two-step paradigm: retrieval and ranking. The retrieval step, which we also refer to as candidate selection [6, 10], first fetches a small subset of documents. Down the pipeline, one or more ranking layers will further scrutinize and recommend the most relevant ones to the users. This work focuses on improving candidate selection, which plays a key role in scaling the systems to millions of queries-per-second (QPS), on millions of documents. An effective candidate selection algorithm should be able to retrieve the most relevant documents (high recall), while leaving out the irrelevant ones (high precision).

One common candidate selection approach is to fetch all documents with matching attribute. Upon each user's request, we may first extract users' attributes (e.g. user's skills in job recommendation, or connections in people recommendation), then fetch documents matching with these attributes. With the help of a prebuilt index (e.g. Lucene, or key-value stores), the lookup can finish quickly. However, this mechanism becomes slower and imprecise as we have ever richer user information. For example, an average job seeker on LinkedIn has dozens of skills, hundreds of connections, as well as browsing activities. Fetching with all attributes can overwhelm the rest of the system with millons of documents, which results in:

(1) Suboptimial recommendations. Downsteam rankers make more mistaks when faced with a large number of irrelevant documents.
(2) Heavy server load. Especially the slowest queries (e.g. the 99th percentile) consume heavy computation resources.

This work studies the issue of choosing a subset of user attributes for candidate selection. The key intuition is that certain attributes tend to match more relevant documents. For example, among the following three skills from a hypothetical machine learning engineer: *Python*, *MS Office* and *tensorflow*, *tensorflow* is likely to bring the best matches. In comparison, *MS Office* and *Python* will dilute the results with jobs such as business administrator or frontend engineer. One challenge in building an algorithm that chooses between user attributes is not having annotated data. To tackle this, we propose a machine learning model to infer from user-document interaction data. We present a family of models that allows us to quickly forecast the relevance of retrieved jobs for each attribute. In production, we first use the model to rank all attributes for a given user, then combine the top-K disjunctively for candidate selection.

This work has the following contributions: (1) To the best of our knowledge, we present the first study on ranking user attributes for candidate selection in recommendation systems. (2) We present a family of models that allows us to quickly forecast the relevance for retrieved documents matching one user's attribute. (3) We present empirical results for proposed models in both offline and online setups. Our offline analysis compares the proposed model with tf-idf on a binary classification task, and shows a 4% AUC increase. Our online experiment on LinkedIn's job recommendation system observes a 50% reduction in latency at 99th percentile, 24.49% reduction in serving cost, along with 7.6% more job applications.

2 RANKING ATTRIBUTES FOR RECOMMENDATION RETRIEVAL

We study algorithms that rank users' attributes. Our candidate selection algorithm will use the ranking result to pick top user attributes, in order to write disjunctive queries.

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

ACM ISBN 978-1-4503-6859-9/20/10…$15.00
https://doi.org/10.1145/3340531.3412742

Formally, suppose a user U has N attributes $A_U = \{a_1, a_2, \ldots a_N\}$ (e.g. from his/her profile or recent activities). Our candidate selection algorithm will pick a subset of attributes $A'_U = \{a_{i_1}, a_{i_2} \ldots a_{i_K}\}$, by first assigning scores to each attribute, as $s(a_1|U)$, $s(a_2|U)$, …, $s(a_N|U)$, then picking ones that have the highest scores. A'_U is then used to generate a disjunctive query: $a_{i_1} \cup a_{i_2} \cup \ldots \cup a_{i_K}$; the query then fetches all documents that match with *any* attribute among them. The core component is the scoring function: s. We aim to find an s that generates queries with both high precision and recall.

To illustrate with an example, suppose a user has the following four attributes:

- *title=machine learning engineer*
- *industry=information technology*
- *skill=tensorflow*
- *skill=Python*

Suppose an algorithm assigns highest scores to *skill=tensorflow* and *title=machine learning engineer*, our system will use it to generate a query that retrieves all jobs that were indexed under them, with *skill=tensorflow* ∪ *title=machine learning engineer*.

Our above definition limits the study to a family of simplistic yet expressive candidate selection algorithms.

First, we only consider fetching documents using subsets of the user attributes. Fetching by user attributes can quickly narrow down the retrieval to a small subset of documents. To guide the recommendation systems, many users also regularly update their profiles. However, the rich user information may overly extend the matching set. For example, on LinkedIn, users often update their job-related information such as their skills, titles, and company. Among them, some attributes (e.g. *skill=C++* for a product manager) may mislead the system in retrieving many irrelevant jobs. By narrowing down the attribute set, we can improve systems' precision.

Second, we only consider disjunctive queries. To speed up document retrieval, inverted indices pre-organize all documents with their attributes, into "buckets" [21]. Disjunctive queries during retrieval time can be understood as fetching documents from a list of buckets. Although the full Disjunctive Normal Form[1] is supported by the indices, we restrict our algorithm to only consider union of atomic matching clauses. This restriction disallows expressing blacklisting attributes (e.g. NOT *seniority=3*), or shrinking the scope (e.g. *title=manager* AND *industry=IT*). However, in practice we can delegate to the ranking layer for handling the more complex logics.

Third, we focus on algorithms that are backed by a scoring function on individual attributes. Considering all attributes in isolation eliminates the need to enumerate their combinations. However, we also realize that one potential downside is being incapable of capturing the correlations between attributes, such as *skill=machine learning* is repetitive to *skill=data mining*.

The scoring function is the core of the family of algorithms under study. We have considered both manual heuristics as well as machine learning algorithms.

Manually exploring for rule-based heuristics can be overwhelmed by the large space of possible solutions. To illustrate with an example, although title matchings often bring precise matches, there are many exceptions: postdocs are often not looking for another post-doc position; broad titles such as manager can match a wide variety of positions such as inventory manager and product manager. Manually fixing the exceptions often quickly leads to a locally optimal solution, where any precision increase will come at the price of recall decrease, and vice versa.

This work presents our machine learning approach. Unlike classical machine learning algorithms, annotated data is unavailable as gold-standard for index queries. Our work was inspired by the k-nearest-neighbor (k-NN) retrieval algorithm described in Youtube's recommendation system [9]. The idea is to first define a probabilistic function over documents, then leverage the index to quickly compute the documents with the highest scores. Particularly, their algorithm learns an embedding for each user/document, so that closer pairs translates to higher relevance. Fetching the most relevant documents then reduces to a k-NN search in the vector space. However, the k-NN search index, which is the backbone of the algorithm, is not directly supported by inverted indices. Inverted indices instead optimizes for faster attribute-level retrieval.

We present a family of document relevance functions customized for inverted index (Section 3). The probability functions can still be trained on the user activity data. We show the functions allow efficient forecasting for the relevance of documents associated with each attribute. The resulting model can be viewed as an enhanced version of tf-idf (term frequency, inverse document frequency [2]), with re-weighed tf's.

3 FORECASTING ATTRIBUTE EFFECTIVENESS

Our central idea for scoring attributes is to forecast the relevance of documents retrieved using one single attribute. We normally observe two common traits for effective attributes: they are both *trending* and *specific*. For example, if we compare *skill=tensorflow* and *skill=Python* for the machine learning engineer user in Section 2:

(1) The former is more trending, as many people who have similar background would lean on *skill=tensorflow* when looking for jobs.
(2) The former is also more specific, as it will match a smaller number of jobs in the database.

By combining effective individual attributes, the overall disjunctive query is also likely to be effective.

Formally, we define an attribute's individual effectiveness as the average of matched documents' relevance.

Document relevance: For any user U, similar with the problem formulation in [9], we assume a probability function $\Pr(d|U)$ that measures each document d's relevance in the database D. $\Pr(d|U)$ is a distribution function over all documents, representing an "extreme multi-class classification" problem.

$$\sum_{d \in D} \Pr(d|U) = 1 \tag{1}$$

This classifier predicts the next document the user will interact with, by assigning each document a probability. We may use user's activities as training data to tune $\Pr(d|U)$.

[1] Disjunctive normal are disjunctions of conjunctions, which can express arbitrary logics https://en.wikipedia.org/wiki/Disjunctive_normal_form.

[2] https://en.wikipedia.org/wiki/Tf-idf

Attribute relevance: Suppose d_a is the subset of documents that match attribute a, we define the relevance of a user's attribute $R(U, a)(a \in A_U)$ to be the average relevance matching documents retrieved by a:

$$R(U, a) = \frac{\sum_{d \in d_a} \Pr(d|U)}{|d_a|} \tag{2}$$

Although theoretically any parameterization of document relevance (Equation 1) leads to a valid attribute effectiveness measure (Equation 2), the summation is expensive. Summation is especially impractical during the candidate selection phase, where the system has not fetched any documents. In the next sections, we constrain $\Pr(d|U)$ to a family of functions to allow easier aggregations over attributes (Equation 2). This enables fast forecasting during online serving.

3.1 Proposed Model Family

Our fundamental idea for eliminating summations is to account for each attribute's contribution separately. Intuitively, each document match can be viewed as a result of multiple attribute matches. For example, suppose the *machine learning engineer* mentioned in Section 2 expresses interest in another machine learning position. This match can be viewed as a result of several user/doc attribute matches: including *skill=Machine Learning* and *skill=Python*. If we can separately account for each attribute's contribution during offline training, it can help to relieve the computation burden during online serving.

We introduce a hidden variable a_H to capture the role each attribute plays in a document match (Figure 1). Our generative model describes a two-step process for finding a relevant document:

(1) Pick an attribute a_H among all the user attributes, with $t(a|U)$. $t(a|U)$ is a distribution function across users' attribute set A_U, capturing users' preference among attributes.
(2) Among all the documents d_{a_H} that match a_H, randomly pick one to interact with, with equal probabilities.

This process defines a probability distribution over all documents.

The hidden variable allowed us to breaks down the document relevance function into a sum of each attribute's contribution. To see that, note that the overall document probability is a sum over all hidden variables' values:

$$\Pr(d|U) = \sum_{a_H \in A_U} \Pr(a_H, d|U) = \sum_{a_H \in A_d \cap A_U} \underbrace{\frac{t(a_H|U)}{|d_{a_H}|}}_{\bar{R}(U, a_H)} \tag{3}$$

We use $\bar{R}(U, a)$ to represent attribute a's contribution to the document's relevance. Because each attribute contributes positively to a document's relevance, each $\bar{R}(U, a)$ can be seen as a lower-bound for documents d_a retrieved by a:

$$\Pr(d|U) = \sum_{a_H \in A_d \cap A_U} \bar{R}(U, a_H) \geq \bar{R}(U, a) \quad (d \in d_a) \tag{4}$$

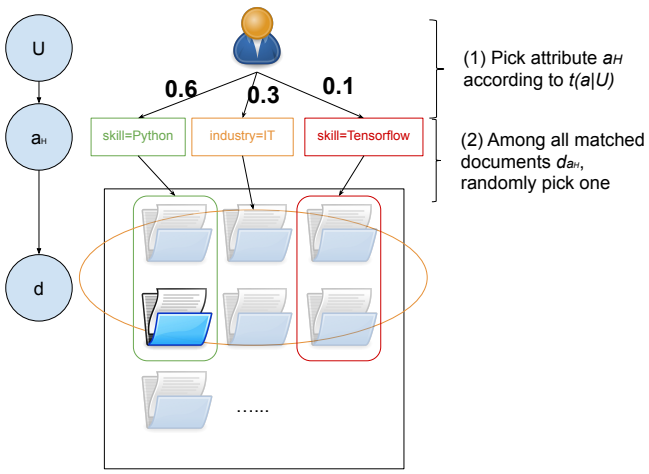

Figure 1: Generative model for choosing a job for a given user. Each user U picks one attribute a_H; then among all the matching docs, the user has equal probability to match each one. Each document's probability therefore is a sum over different picking paths (Equation 3). Take the highlighted document as an example. Its probability is a sum of two components: (1) first picking *skill=Python*, then picking among the two documents matched with *skill=Python*; (2) first picking *Industry=IT*, then picking among the six documents matched with *Industry=IT*. Therefore, the probability of fetching the highlighted document is $0.6 \times \frac{1}{2} + 0.3 \times \frac{1}{6}$.

Therefore $\bar{R}(U, a)$ is also a lower-bound for a's attribute relevance defined in Equation 2:

$$\begin{aligned} R(U, a) &= \frac{\sum_{d \in d_a} \Pr(d|U)}{|d_a|} \\ &\geq \frac{|d_a| \bar{R}(U, a)}{|d_a|} \quad \text{(Equation 4)} \\ &= \bar{R}(U, a) = \frac{t(a|U)}{|d_a|} \end{aligned} \tag{5}$$

We use $\bar{R}(U, a)$ (in Equation 5) as an approximate for the attribute relevance, which we use to rank user attributes during retrieval.

3.2 Parameterizations

We may interpret the two sets of parameters $t(a|U)$ and $|d_a|$ as capturing *trend* and *specificity* for each attribute, respectively.

- t favors attributes that users tend to choose. For example, the model can learn users who have both *skill=tensorflow* and *skill=Python* tend to choose the former, from users' activites.
- $\frac{1}{|d_a|}$ favors more specific attributes. For example, *skill=tensorflow* may have less count than *skill=Python* among job postings, therefore having a higher score.

Between them: $|d_a|$ can be obtained by simply counting frequencies, whereas we may parameterize t differently, which we will show in the following subsections.

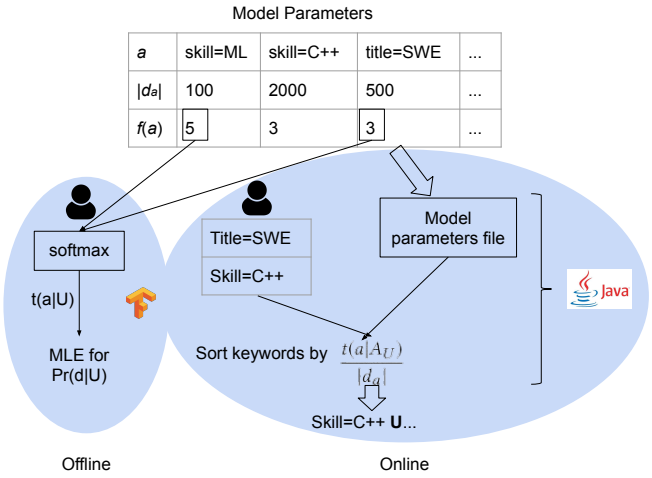

Figure 2: Production setup for attribute scoring.

3.2.1 Equal Preference Scores. In extreme, we may have no parameters in t, by assigning constant values. Formally,

$$t(a|A_U) = \begin{cases} \frac{1}{|A_U|} & (a \in A_U); \\ 0 & \text{otherwise.} \end{cases}$$

Ranking attributes with the equation above is equivalent to ranking with tf-idf, with tf being either 0 or 1 (1 if present among users' attributes).

The fundamental issue with using tf-idf is that it only considers attributes' rareness, without considering the job seeking trend. For example, between two attributes *skill="emacs lisp"* and *skill=tensorflow*, tf-idf will rank the former higher, because fewer jobs will require such a skill.

3.2.2 Global Trend Scores. One simple parameterization is to represent the trend into a static score mapping. For example, if people who have both *tensorflow* and *"emacs lisp"* generally prefers to look for *tensorflow* jobs, the mapping can assign a higher score for *tensorflow*.

Formally, we define f as a static mapping from attributes to numerical values; t as a softmax of f:

$$t(a|U) \propto \exp f(a) \quad (a \in A_U) \tag{6}$$

Compared with tf-idf, f can compensate for more trending attributes (e.g. raising *skill=tensorflow* above *skill=elisp*).

One limitation of the formulation in Equation 6 is not considering the context's influence on the preferences. For example, *Python* may be more preferred to *C++* for data science engineers, but less preferred for infrastructure engineers. By incorporating more features from the users, the scoring has room for improvements. We leave the exploration as future work.

3.3 Shutterspeed – Our Production Workflow Overview

Figure 2 shows an overview of our production implementation on the model described in Section 3.2.2.

During offline training, we use Maximum Likelihood Estimation (MLE) to tune the model over job application records, with back-propagation. Formally, we search for optimal parameters Φ with:

$$\arg\max_{\Phi} \sum_{(U,d)\in T} \log \Pr(d|U;\Phi) \tag{7}$$

Here T is the training data containing users' apply records where each record (U, d) is a user U's application to job d; $\Pr(d|U;\Phi)$ is defined by Equation 3 and 6. We use tensorflow [1] with mini-batches and Adam optimizer [15] to perform this optimization. We chose neural network software packages over traditional expectation-maximization (EM), for the ease of use, and potential future work on more complex $t(a|U)$ formulations.

The training yields two sets of parameters: frequency mapping $(a \to |d_a|)$ and global trends $(a \to f(a))$. The parameters are stored into a dictionary, and uploaded into the online serving component. The online serving components will rank user attributes with Equation 5, and pick top K in generating retrieval queries.

K is an important parameter, which we refer to as *shutterspeed*. The name comes from its analogy from photography, that higher K (similar to longer exposure time) will "expose" more documents.

4 OFFLINE EXPERIMENTS

During offline experiments, we evaluate candidate selection algorithms' accuracies by comparing retrieval queries with real user activities.

4.1 Metrics

Note that query generation is fundamentally different from many ranking/classification tasks, in that it is difficult to establish "gold-standard". Ranking tasks are often formulated as binary classification tasks. However, query generation's output is free-form text, whose grammar is tied to a particular retrieval system (e.g. lucene retrieval language, v.s. numerical vectors on a k-NN search index).

The idea behind our evaluation mechanism is to treat generated queries as "mini-classifiers", and evaluate their accuracies. For example, a lucene query *"title:software engineer"* can be treated as a classifier that returns positive for all software engineer job postings, and false for all other jobs. We can compare the output from "mini-classifiers" with true/false labels in traditional ranking datasets, and compute precision/recall metrics. Specifically, we evaluate on a balanced set where we assign:

- Positive labels to users' positive interactions with the system (e.g. click/job apply);
- Negative labels to users and randomly sampled jobs that they have not interacted with (e.g. jobs other people have applied).

We try to get a holistic view by varying the top K in generating queries, and compute AUC-ROC, f_1 score and precision at 95% recall. For each K, we generate a disjunctive query using the top K keywords extracted from the member profile, using our ranking algorithms. Intuitively, a smaller K will translate to shorter disjunction queries with lower recall, but higher precision; and vice versa. By varying K from 1 to $+\infty$, we are able to obtain an ROC (Receiver

	Cardinality	Average # per User
Skills	35K	92.08
Titles	25K	3.2
Seniorities	10	0.94
Functions	26	0.99
Industries	147	0.97

Table 1: Distribution of user profile attributes. We aggregate the statistics based on one day of user's job application data. Note that users have more than one title attribute because they can set preferred titles on our UI.

operating characteristic) curve, as well as gauging statistical significance. The aforementioned evaluation metrics are computed as:

AUC-ROC The area-under-curve for the ROC curve.

f_1 score The maximum f_1 score by varying K.

precision at 95% recall The precision at the K which obtains at least 95% recall. We use this metric to forecast the potential precision improvements (e.g. reduced latency) when sacrificing minimum recall (e.g. job applications).

4.2 Experimental Setup

We compare query rewriters on LinkedIn's job recommendation dataset. The dataset contains one day of job applications from all LinkedIn products (search, recommendation and notifications). We use 80% of data for model training, 10% for development, and the remaining 10% for evaluation. The data volume is adequate for tuning our model parameters, as well as gauging statistical significance.

We use the following users' profile fields as the initial attribute sets.

Skills: a list of skills that (1) users explicitly entered, or was endorsed (2) implied skills from other skills, e.g. tensorflow implies mastery of Python (3) inferred skills, e.g. from text in profile description, job title description [5].

Titles: user's current primary position's job title, as well as the preference title. Preference titles are entered by users via UI.

Seniorities: We categorize users' seniority into their careers into a categorical value from one to ten: one being unpaid, ten being business owner/partner. The category is inferred with an in-house classifier.

Functions: A coarse grouping of titles, including broad areas such as "engineering", "healthcare" and "sales".

Industries: the industry of the user's current company, e.g. IT industry.

The statistics of the aforementioned attributes are shown in Table 1. Among the attributes, skills and titles are more finer-grained. On average, each user has 100 attributes we use for retrieving the candidate sets. The majority of these attributes are skills.

We compare the following methods for picking a subset of user attributes for job retrieval.

Random: Assign random scores to attributes. Note that even randomly picking attributes will yield better than random accuracies. This is because each profile attribute carries some information on distinguishing matching documents.

Prioritize title: Still randomly assign scores to each attribute, at the same time boosting title's scores by a large constant number. We use this as an example of heuristics-based systems. Titles are preferred attributes, because they often have a stronger association with career transitions.

tf-idf: Computing the inverse document frequency for each attribute.

Shutterspeed: The proposed model in Section 3.

Ranking Model	AUC	f_1 score	Precision at 95% recall
Random	62.89%	67.92%	52.60%
Prioritize title	70.47%	68.67%	52.73%
tf-idf	77.96%	73.50%	54.16%
Shutterspeed	**81.86%**	**76.21%**	**55.03%**

Table 2: Offline evaluation results for various models.

4.3 Results

During our offline experiments, we try to answer the following questions:

(1) Which attribute scoring mechanism can best distinguish matching documents from non-matching ones?
(2) Does the trend parameter in the proposed model help to correct tf-idf's bias?
(3) How does the "shutterspeed" parameter K affect the recommendation results?

The main comparisons are shown in Table 2. We also conducted anecdotal analysis to compare algorithm outputs for a software engineer in the Bay area, shown in Table 3 and Figure 3. We make the following observations.

First, the top-K keywords chosen by the Shutterspeed model can better distinguish relevant jobs from irrelevant ones.

Second, capturing users' preferences can affect the ranking for different keywords (Table 3). The tf-idf measure always prefers rare skills/titles on the job market. In comparison, the Shutterspeed model adjusted the ranking based on market trends. For example, it boosts *Title: Senior Software Engineer* and *Skill: Pattern Recognition*, also suppresses *Skill:Graph Theory, Skill: Scala*. However, we also observed that the top-ranked keywords still contain keywords that may fetch irrelevant jobs. In our case, *Skill: Java Enterprise Edition* and *Skill: Django* may retrieve general backend/system development positions that may misalign with the user's interest. In future work, we hypothesize that a more accurate modeling for $\Pr(a|U)$ may help the model make clearer decisions.

Third, setting the K properly is critical in retrieving what the users need. In Figure 3, we illustrate the recommendations for each choice of K. A smaller K limits the diversity of the retrieved jobs, which in our example, fetches primarily data related engineering positions. By loosening K, a wider range of documents can be fetched. This will increase system recall at first (e.g. at $K = 40$, the system discovered many top-ranked jobs). However, increasing K will finally increase the system's burden, forcing the system to

tf-idf	Shutterspeed
Skill: Graph Theory	Title: Staff Software Engineer
Title: Staff Software Engineer	Skill: Pattern Recognition
Skill: Information Retrieval	Title: Senior Software Engineer
Skill: Algorithm Design	Skill: Computer Vision
Skill: Pattern Recognition	Skill: Algorithm Design
Title: Machine Learning Engineer	Skill: Java Enterprise Edition
Skill: Natural Language Processing (NLP)	Skill: Image Processing
Skill: Scala	Skill: Django
Skill: Scalability	Skill: Natural Language Processing (NLP)
Skill: Chinese	Skill: Information Retrieval

Table 3: Top-10 keywords chosen by tf-idf v.s. shutterspeed model

title	original	k=10	k=40	k=100
Python Platform Engineer	1	1	1	1
Machine Learning Engineer	2	2	2	2
Staff Software Engineer, Data	3	6	6	4
Senior Data Engineer	4			5
Software Engineer - Timelines	5			6
Senior Manager/Director, Software Engineering	6		9	7
Senior Software Engineer	7			8
Sr Software Engineer	8	9	10	9
Research Data Scientist	9			10
Software Engineer- HD Maps	10			
Senior Machine Learning Engineer		3	3	
Machine Learning Engineer		4	4	
Backend Engineer		5		
Principal Engineer, Machine Learning		7	7	
Applied Research Scientist, Natural Language Processing		8	8	
Staff Software Engineer, Backend		10		
Senior Principal Software Engineer			5	3

(a) Altering K for a software engineer user.

title	original	k=10	k=40	k=100
Product Manager, Digital Commerce	1			1
Product Manager - Learning - Relevance	2	3	3	3
Founder, New Product Experimentation	3		6	4
Principal Product Manager - Tech	4			5
Software Engineer - Timelines	5			6
Project Manager - Customer Products	6			7
UX Researcher	7			8
Product Manager - Marketing Technology	8			9
Product Manager, Terminal	9			10
Senior Strategic Growth Manager	10			
Product Manager, Outcomes		1	1	
Senior Product Manager		2	2	2
Sr Manager, Product Management		4	5	
Product Manager		5	7	
Senior Growth Product Manager		6	8	
Product Manager, WhatsApp		7		
Product Manager		8		
Product Manager, Credit		9	9	
Product Manager - Reinventing		10	10	
Founder, New Product Experimentation			4	

(b) Altering K for a product manager user.

Figure 3: Differences in recommendations when we alter K. We illustrate the change of recommendations for two users who are machine learning engineer and product manager, respectively. In the tables, each line represents a job. The last three columns represent the rankings of recommendations for each choice of K. The numbers in each grid is the ranking of the job among all the top-10 recommendations for the algorithm variant. In the above examples, lower K's retrieves more focused recommendations, centered around the member's current title. Larger K's will enrich the recommendations with richer title/skill choices, however, also at the risk of diluting the results.

early-terminate (e.g. in Figure 3a, at $K = 100$, the system will lose the jobs used to rank at 3rd and 4th at $K = 40$).

5 ONLINE EXPERIMENT

We also experimented with the shutterspeed model in a production environment, for LinkedIn's job recommendation system[3]. Our hypothesis is that with a more restrictive candidate selection algorithm, we are able to significantly improve the system's precision, without sacrificing recall.

We compare shutterspeed with our baseline system which employed a first pass ranker (Figure 4). Before shutterspeed, our recommendation system employed a first pass ranker (L1) serving as a rough filtering layer. The idea behind is by using a simplistic (80 features) logistic regression, we can quickly filter a majority of irrelevant documents, therefore reducing server load. By design, the classifier will have limited accuracy due to the tight time budget. In our online experiment, we tried to replace the first pass ranker with a tighter candidate selection layer. Compared to 1st pass ranker, a smarter candidate selection algorithm has great advantages over time budget. This is because query generation is done only once per query, compared to the ranking algorithm which applies to all the matching documents per query.

We use two K's to retrieve jobs in production, K_1 and K_2 (with $1 \leq K_1 < K_2 \leq 100$), to enable a fallback mechanism. When the smaller K_1 is unable to retrieve enough jobs, we fall back to the larger K_2. At LinkedIn, we implement this mechanism via Flex queries on Galene (LinkedIn's in-house adaptation of lucene) [13]. When such functionality is not natively supported by the query language, we may also consider sending two index queries per member request. Between K_1 and K_2:, K_2 has more influence on the overall number of retrieved jobs, whereas K_1 further controls the

[3]http://www.linkedin.com/jobs

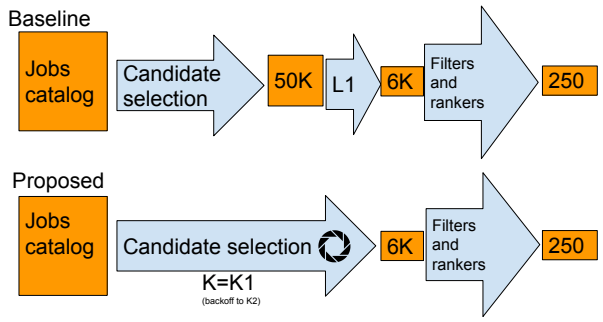

Figure 4: During online experiments, we compared the enhanced candidate selection layer with the legacy setup, which used a light-weight ranker for basic filtering. The legacy setup uses a loose candidate selection algorithm which fetches jobs that match with two out of three matching criteria: function, title or skills. We then employed a light-weight logistic regression layer, which we refer to as L1, to filter jobs that have a lower matching confidence. The new setup enhanced the candidate selection layer with Shutterspeed, and also dropped the L1 layer. The candidate selection layer first tried to fetch documents with top 20 clauses according to shutterspeed; it will back off to fetching with a wider range ($K = 50$) if the number of matched documents is below a threshold.

relevance of the jobs. To ensure fair comparison, we first binary-search for K_2 that retrieves the same average number of jobs with control. With the fixed K_2, we grid-search for K_1 that maximizes the ranking scores for the retrieved jobs.

We compared business metrics corresponding to precision and recall. We conduct the comparison using our in-house A/B testing framework, which randomly partitions the population into "buckets", and compares statistics between them. We particularly consider the following metrics

Job applies. The total number of apply clicks divided by population size for each treatment. We consider this as a recall metric, for measuring how many good jobs the system is capable of discovering. This is also one of the key business metrics for measuring system performance.

Latency. The average response time for each query. We consider this as a precision metric, ranging from 0 (high precision) to ∞ (low precision). The majority of the system computation time is spent on ranking every fetched job. A smaller set of retrieval results often implies reduced computation time, as well as higher precision. Latency is a key measure for the system's scalability. We further divide the metrics into the following categories

p50, p90, p95, p99. We distinguish latency across different quantiles to further understand the impact to system performance. In practice, the slowest queries often have the biggest impact on system performance, (e.g. causing garbage collection to occur, or consuming system bandwidth).

Serving cost. We compute the overall average CPU time for each query as a measure for serving cost. This metric often serves as a reference on how much parallel computation resources are needed.

Onsite Job Applies	+7.66%
Latency	
p50	-1.78%
p90	-34.72%
p95	-39.2%
p99	-38.42%
Serving Cost	-24.49%

Table 4: Online ramping results

The A/B test was conducted between 20% and 20% of users for two weeks[4].

The experimental results are shown in Table 4. We observed a large lift in both recall and precision metrics. The model is able to discover jobs that the baseline system did not have time to rank. On the other hand, the system becomes much faster, because our keyword set reduces the number of retrieved documents significantly. The effect is especially evident for the slowest (e.g. p99) queries. This contributes to a significant reduction in overall serving cost. At the time of this writeup, the proposed model had been serving all job recommendation requests for one year.

6 RELATED WORK

Many studies on recommendation systems focus on ranking, i.e. determining which document is more relevant given a pool of documents. Ranking is one fundamental challenge for recommendation systems. Theoretically, one ranker is able to power an entire recommendation system, by scanning through all documents, and picking the most relevant ones per user request. Many different models were studied, including both linear models [20] and neural network based models [8, 12, 19].

Today, commercial recommendation systems operate at a large scale: serving millions of documents to millions of people. Scalability becomes another challenge. Many recent work studies how to quickly fetch an initial subset of documents before the finer-grained ranking. One common way is to rely on an index structure. A nightly job can pre-organize all the documents into the index, so that the online serving becomes faster. The type of index structure determines what optimizations are possible on top of it.

One recent trend is candidate selection on k-NN indices, which supports vicinity based lookups by treating each document as a vector [17]. The indices structure fits well with recent advancements in deep learning [4, 9]. One common formulation is to represent both users and documents as vectors, then define the relevance functions based on distances between vectors. This formulation reduces computing arg max for the relevance function into a k-NN lookup. A rich set of probability functions can be used to learn the embeddings, with advances in training techniques [11, 14, 16].

Another active research area is on top of traditional inverted indices [6, 10]. Inverted indices have a long history of research,

[4]Ideally all experiments can be conducted with 50%-50% comparisons. However, in practice the bandwidth is often limited because multiple experiments on the same system component are running simultaneously.

also supporting many different lookup operators [7, 13]. Several advantages of inverted indices include:

(1) Easy to incorporate business rules (e.g. limiting retrieval results for a certain geo-location)
(2) Easy to explain the retrieval query, especially when debugging improper recommendations
(3) Able to share the infrastructure with search systems [2].

One strategy to improve the query accuracy is to implement filtering logics during query execution, for example, with a linear model [6] or a tree model [10]. The algorithms in [6, 10] generate complex queries that implement the filtering logic (e.g. a score goes above a threshold); which will apply to documents during scanning.

This work studies filtering documents by cutting off query clauses, i.e. during query generation time, before they execute. We have now faced overly riched user information, opposite to cold-start [18]. By shrinking down the set of user attributes to use, we can improve the system's precision and scalability. Ranking the keyword attributes shares similarities with query expansion [3], where both's goal is to generate accurate queries. However, the fundamental difference is all the query terms in our case are explicitly specified by the user. As a result, certain strong signals in query expansion (e.g. similarity with original query) do not have analogies in our setup.

This work borrows the common idea applied on k-NN indices, and applied it to the inverted indices setup. By defining a probability function over documents, we can leverage the inverted indices to efficiently retrieve the most relevant documents. We show (in Section 3.2.1) that the trivial form of our proposed model reduces to ranking attributes with tf-idf. Further parameterizing the probability function can help personalize the queries, and improve accuracy.

7 CONCLUSIONS

The candidate selection step scales recommendation systems by reducing the number of documents to score. This work studies subsetting the number of user attributes at candidate selection. We propose a model to rank user attributes based on the relevance of the documents they match. By only using the top K attributes during candidate selection, we are able to improve both precision and recall. We demonstrated deploying the component into a real-world setup on job recommendations at LinkedIn, where we observed 24% p99 latency reduction, along with user engagement boost.

ACKNOWLEDGEMENTS

The author would like to thank Gheorghe Muresan, Dawei Wang, Badrul Sarwar and the anonymous reviewers for insightful comments and suggestions. Special thanks to Jeffrey Lee for the guidance on productionizing the online solution. Last but not the least, thank the continuous support from Ram Swaminathan, Josh Hartman, Ada Yu and Suman Sundaresh.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 265–283.

[2] Dhruv Arya and Ganesh Venkataraman. 2017. Search Without a Query: Powering Job Recommendations via Search Index at LinkedIn. In *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval, Shinjuku, Tokyo, Japan, August 7-11, 2017*, Noriko Kando, Tetsuya Sakai, Hideo Joho, Hang Li, Arjen P. de Vries, and Ryen W. White (Eds.). ACM, 1347. https://doi.org/10.1145/3077136.3096470

[3] Hiteshwar Kumar Azad and Akshay Deepak. 2019. Query expansion techniques for information retrieval: A survey. *Information Processing & Management* 56, 5 (2019), 1698–1735.

[4] Oren Barkan and Noam Koenigstein. 2016. Item2vec: neural item embedding for collaborative filtering. In *2016 IEEE 26th International Workshop on Machine Learning for Signal Processing (MLSP)*. IEEE, 1–6.

[5] Mathieu Bastian, Matthew Hayes, William Vaughan, Sam Shah, Peter Skomoroch, Hyungjin Kim, Sal Uryasev, and Christopher Lloyd. 2014. Linkedin skills: large-scale topic extraction and inference. In *Proceedings of the 8th ACM Conference on Recommender systems*. 1–8.

[6] Fedor Borisyuk, Krishnaram Kenthapadi, David Stein, and Bo Zhao. 2016. CaSMoS: A framework for learning candidate selection models over structured queries and documents. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 441–450.

[7] Andrei Z Broder, David Carmel, Michael Herscovici, Aya Soffer, and Jason Zien. 2003. Efficient query evaluation using a two-level retrieval process. In *Proceedings of the twelfth international conference on Information and knowledge management*. 426–434.

[8] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, Rohan Anil, Zakaria Haque, Lichan Hong, Vihan Jain, Xiaobing Liu, and Hemal Shah. 2016. Wide & Deep Learning for Recommender Systems. *CoRR* abs/1606.07792 (2016). arXiv:1606.07792 http://arxiv.org/abs/1606.07792

[9] Paul Covington, Jay Adams, and Emre Sargin. 2016. Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM conference on recommender systems*. 191–198.

[10] Aman Grover, Dhruv Arya, and Ganesh Venkataraman. 2017. Latency Reduction via Decision Tree Based Query Construction. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management* (Singapore, Singapore) *(CIKM âĂŹ17)*. Association for Computing Machinery, New York, NY, USA, 9. https://doi.org/10.1145/3132847.3132865

[11] Michael Gutmann and Aapo Hyvärinen. 2010. Noise-contrastive estimation: A new estimation principle for unnormalized statistical models. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. 297–304.

[12] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. 2017. Neural collaborative filtering. In *Proceedings of the 26th international conference on world wide web*. 173–182.

[13] Dmytro Andriyovich Ivchenko and Niranjan Balasubramanian. 2016. Flexible operators for search queries. US Patent App. 14/584,813.

[14] Sébastien Jean, Kyunghyun Cho, Roland Memisevic, and Yoshua Bengio. 2014. On using very large target vocabulary for neural machine translation. *arXiv preprint arXiv:1412.2007* (2014).

[15] Diederik P. Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. arXiv:1412.6980 [cs.LG]

[16] Omer Levy and Yoav Goldberg. 2014. Neural word embedding as implicit matrix factorization. In *Advances in neural information processing systems*. 2177–2185.

[17] Ting Liu, Andrew W Moore, Ke Yang, and Alexander G Gray. 2005. An investigation of practical approximate nearest neighbor algorithms. In *Advances in neural information processing systems*. 825–832.

[18] Andrew I Schein, Alexandrin Popescul, Lyle H Ungar, and David M Pennock. 2002. Methods and metrics for cold-start recommendations. In *Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval*. 253–260.

[19] Suvash Sedhain, Aditya Krishna Menon, Scott Sanner, and Lexing Xie. 2015. Autorec: Autoencoders meet collaborative filtering. In *Proceedings of the 24th international conference on World Wide Web*. 111–112.

[20] XianXing Zhang, Yitong Zhou, Yiming Ma, Bee-Chung Chen, Liang Zhang, and Deepak Agarwal. 2016. Glmix: Generalized linear mixed models for large-scale response prediction. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 363–372.

[21] Justin Zobel and Alistair Moffat. 2006. Inverted Files for Text Search Engines. *ACM Comput. Surv.* 38, 2 (July 2006), 6âĂŞes. https://doi.org/10.1145/1132956.1132959