Latency Reduction via Decision Tree Based Query Construction

Ben Trovato*
Institute for Clarity in Documentation
P.O. Box 1212
Dublin, Ohio 43017-6221
trovato@corporation.com

G.K.M. Tobin†
Institute for Clarity in Documentation
P.O. Box 1212
Dublin, Ohio 43017-6221
webmaster@marysville-ohio.com

Lars Thørväld‡
The Thørväld Group
1 Thørväld Circle
Hekla, Iceland
larst@affiliation.org

Lawrence P. Leipuner
Brookhaven Laboratories
P.O. Box 5000
lleipuner@researchlabs.org

Sean Fogarty
NASA Ames Research Center
Moffett Field, California 94035
fogartys@amesres.org

Charles Palmer
Palmer Research Laboratories
8600 Datapoint Drive
San Antonio, Texas 78229
cpalmer@prl.com

John Smith
The Thørväld Group
jsmith@affiliation.org

Julius P. Kumquat
The Kumquat Consortium
jpkumquat@consortium.net

ABSTRACT

LinkedIn as a professional network serves the career needs of 450 Million plus members. The task of job recommendation system is to find the suitable job among a corpus of several million jobs and serve this in real time under tight latency constraints. Job search involves finding suitable job listings given a user, query and context. Typical scoring function for both search and recommendations involves evaluating a function that matches various fields in the job description with various fields in the member profile. This in turn translates to evaluating a function with several thousands of features to get the right ranking. In recommendations, evaluating all the jobs in the corpus for all members is not possible given the latency constraints. On the other hand, reducing the candidate set could potentially involve loss of relevant jobs. This work provides a novel way of improving the candidate jobs for the job seeker without hurting the overall relevance of the product. We propose a novel way to model the underlying complex ranking function with a decision tree. The variable within the branching of the decision tree would be candidate query clauses. We developed an offline framework which evaluates the quality of the decision tree with respect to latency and recall. We tested the approach on job search and recommendations on LinkedIn and A/B tests show significant improvements in member engagement and latency. Our techniques helped reduce job search latency by over **67%** and our recommendations latency by over **55%**. **As of writing the approach powers all of job search and recommendations on LinkedIn.**

*Dr. Trovato insisted his name be first.
†The secretary disavows any knowledge of this author's actions.
‡This author is the one who did all the really hard work.

WOODSTOCK'97, El Paso, Texas USA

DOI: 10.475/123_4

CCS CONCEPTS

• **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability;

KEYWORDS

Information Retrieval, Personalized Search, Recommender Systems

ACM Reference format:
Ben Trovato, G.K.M. Tobin, Lars Thørväld, Lawrence P. Leipuner, Sean Fogarty, Charles Palmer, John Smith, and Julius P. Kumquat. 1997. Latency Reduction via Decision Tree Based Query Construction. In *Proceedings of ACM Woodstock conference, El Paso, Texas USA, July 1997 (WOODSTOCK'97),* **??** pages.
DOI: 10.475/123_4

1 INTRODUCTION

The jobs ecosystem in LinkedIn powers the career needs of 450 Million plus professionals around the world. The two main means of finding a job via LinkedIn are through search and recommendations. Traditional information retrieval involves matching and scoring a set of documents given query. Personalized search involves scoring documents given query and **user** [**?**]. Both search and recommendations pose unique set of constraints on quality and latency.

LinkedIn job search and recommendations are used in different parts of the ecosystem with different types of constraints. Figures [**??**] and [**??**] show the use case for jobs recommendations in LinkedIn jobs home and LinkedIn feed (home page) respectively. The LinkedIn feed [**?**] personalizes the experience by merging different sources of information including posts, comments, news articles and jobs. Each of these sources (including job recommendations) must respond with very tight SLA (Service Level Agreements, usually latency constraints imposed by the callers of a service)s in order for the home page or the app to be responsive. In job recommendations, there is no explicit query to filter the results. The underlying recommendations model is deeply personalized with

per member and per job coefficients. Such personalization helps us achieve very high relevancy [**?**]. Scoring all possible jobs for each member in this complex model would be prohibitively expensive and will certainly not meet our SLA requirements. At the same time, scoring very few documents would result in loss of relevance.

Job search has similar issues as well. While search has an explicit query, sometimes the query can be way too broad (example: *marketing*). These broad queries match several millions of jobs and ranking all of them would not meet our latency requirements. There are several such head queries (*marketing, sales*) which exhibit similar characteristics. In this case we could leverage information in the members profile to bias the query towards particular jobs he/she may be interested in.

One common way of handling too many matching documents is to pass through multiple levels of rankers. The ranker closest to the retrieval stage is coarse and biases it's decision towards recall and not precision. But, as we move to higher levels, we move on to more complex rankers which take more time, but also rank fewer documents. In this regard, we generally separate out the search/recommendations flow into two steps pre-retrieval and post retrieval. Pre retrieval involves constructing the query based on both the explicit query from user as well as personalization added on top of it and post retrieval involves going through multiple layers of ranking. The post retrieval ranking for both search and recommendations is well studied [**?**].

The objective of query construction is to formulate a query to our retrieval engine such that we minimize the number of documents ranked (direct impact on latency) while making sure we score the relevance documents. In this work we propose a decision tree based approach towards query construction that meets the aforementioned objective. The key contribution of the work includes:

(1) We convert the query construction problem into a decision tree problem with branches in the tree corresponding to query clauses and show that such a formulation scales for very large number of features. We show that such a formulation can have a huge impact on latency without affecting quality of the results - all at LinkedIn scale.
(2) We propose an offline replay mechanism which does a true evaluation of the underlying ranking metric while evaluating gain in latency. It also increases the experimentation velocity by near accurate reproduction of production behavior and reduces number of A/B tests needed with live traffic.
(3) The framework is generic across search and recommendations so long as the underlying retrieval is based on an inverted index.

The techniques described in this work are all in production. In job search it resulted in p99 (worst 1% of the queries) latency reduction of **67%** and in jobs recommendations, we show a p99 latency reduction of **55.96%**. While p99 queries are more important on an operational perspective, we also show significant reduction in latency for median case as well. While we did not explicitly target quality metrics, we did find a **3.5%** improvement in job application as a result of this approach. We attribute this improvement mainly due to reduction in timeouts from our callers (due to improved latency). All results shown were obtained by online A/B testing. As of writing this paper, the techniques described in this work power all of job search and recommendations at LinkedIn.

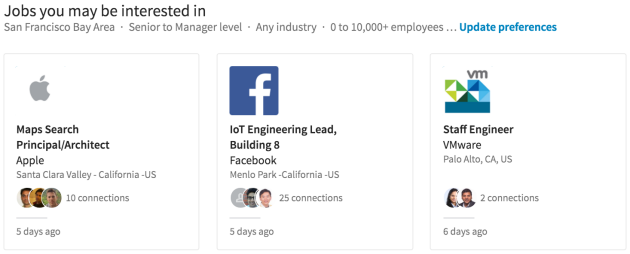

Figure 1: Jobs recommendations in jobs homepage

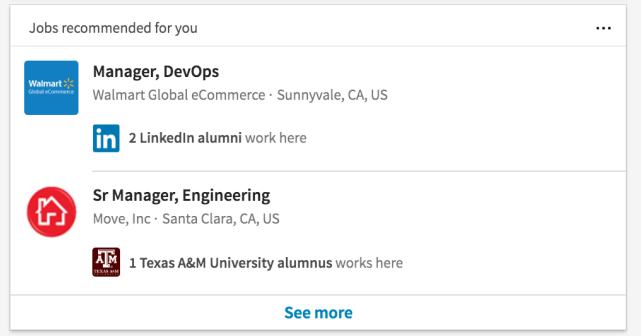

Figure 2: Jobs recommendations in LinkedIn feed

2 RELATED WORK

Query formulation is a well studied area in Information Retrieval (IR). For most part, query optimization deals with reducing latency. It also enables use of more complex/time consuming ranker with fewer number of documents. [**?**] formally introduces the concept of a WAND (weighted AND) operator which has since then been extensively used in large scale search systems including LinkedIn. The WAND operator requires weights and [**?**] proposes tf-idf based heuristics to tune those weights. Search systems organized in terms of inverted index [**?**]. For inverted index based search systems, a common technique used to speed up retrieval is the use of early termination [**? ? ?**]. Use of early termination requires that the posting list is ordered in such a way that the highest quality document is placed first. Note that this quality is dependent only on the document and not the query or the user. This score computed on a per document basis is often referred to as "static rank". Various techniques used for computing suitable static rank include term frequencies [**?**] or the more popular pagerank [**?**]. Commonly used scores include within-document term frequencies [**?**], or some notion of document quality or popularity [**?**].

[**?**] proposes a two pass approach to ranking. The first pass retrieves the top-k documents using approximate ranking. The second pass uses a more accurate model. The document and items are represented in a vector space. The first state ranking leverages this vector space representation to retrieve a limited set of documents.

This gives more time for the second pass to work on a more accurate model. More recently [**?**] propose a neural network based candidate selection for YouTube recommendations. In this work, the authors reduce the number of videos to be ranked to a few hunderds by converting the recommendation problem into an extreme multi-class classification problem. A high dimensional embedding of each document and item (in this case user and videos) is learned using neural networks. A nearest neighbor search is executed in real time in this high dimensional space to retrieve the top N documents. [**?**] propose a way to learn Boolean queries in such a way that they are manageable by humans. Such queries are then used as filters.

The work closest to our existing work [**?**] proposes a framework to prune irrelevant documents and retrieve documents that are likely to be in the top-k results of the query. This is done by training a constrained feature selection algorithm to learn positive weights for feature combinations of the weighted candidate query. Our approach while using the same retrieval engine differs in the following aspects:

(1) We propose a decision tree based framework to come up directly with query clauses.
(2) We propose an offline replay framework which accurately estimates the differences between constrained query formulation vs retrieving all documents. [**?**] uses AUC as evaluation. AUC is a classification metric while the underlying problem we are trying to model is a ranking problem. Pointwise metrics (like AUC) are shown to be less accurate [**? ?**]. Since our replay framework models production queries on a production index, it reduces the number of iterations needed for online A/B testing.
(3) [**?**] does not make use of retrieval operators like early termination and suitable static rank. Getting a good representative static rank can reduce the latency significantly.

3 PROBLEM FORMULATION

Let

- EQ represent the explicit user query. In search EQ represents the search term and in recommendations EQ is empty.
- U represent set of user attributes in the profile. Example - current company, current title etc.
- C represent the context. This may include past job searches, job applies, jobs views etc.
- $D(q)$ represents the set of document retrieved from inverted index for a given query q. $|D(q)|$ represents it's cardinality.
- k represent the top documents required to be shown to the user and M represent the total number of documents that match the query. Typically $k << |M|$.
- $R_f(d, k)$ represent the underlying ranking function which generates top k documents from document set d.

The ranking function takes a (user, query, document) tuple and converts it into a score. Our ranking function is highly personalized using per member and per job coefficients. This work doesn't touch the ranking function but modifies the query. Interested reader may refer to [**?**] for more details. The objective of this work is to come up with a constructed query CQ number of documents retrieved is minimized while the top k documents are not lost due to restricted query construction. More formally:

$$\begin{aligned} \text{Given} \quad & R_f \\ CQ = g(U, EQ, C) & \\ \text{minimize} \quad & |D(CQ)| \\ s.t. \quad R_f(D(CQ), k) &= R_f(D(EQ), k) \end{aligned} \tag{1}$$

Assume that we need the top 25 jobs and explicit query matched 1 Million jobs. This implies that we need to rank 1 Million jobs without query construction. Suppose the constructed query matches 10K jobs, as long as the top 25 jobs are still part of the 10K retrieved jobs, we are have no loss of recall and significant reduction in latency. We come up with a decision tree based technique which will automatically learn the function g to construct the query while trying to optimize the problem stated in equation **??**.

have skipped preferences since it is somewhat confusing, need to revisit it

4 SYSTEM ARCHITECTURE

In this section we desribe the underlying inverted index architecture, query operators, query processing and offline replay framework used to build job search and recommendations at LinkedIn. Both search and recommendations are powered by an inverted index. The underlying engine is custom build called Galene [**?**]. Galene still uses the indexing format of the open source search engine Lucene [**?**]. The Lucene primitives are used to build the index, construct queries and retrieve matching documents. All other components are custom built. The major components/features of our architecture are as follows:

4.1 Indexing

Indexing on Hadoop [**?**] takes the form of multiple map-reduce operations that progressively refine the data into the data models and search index that ultimately serve live queries (Figure **??**). HDFS contains raw data containing all the information we need to build the index. We first run map reduce jobs with standardization algorithms embedded that enrich the raw data resulting in the derived data. The raw data goes through standardization where it gets converted to structured entities. For example, a job posting may have the following free form description *"Looking for Software Engineers with 5+ years experience in Java"*. Standardization would extract the fact that the job requires a skill called **"java"** and store that directly as a field in the index.

4.2 Early Termination

rephrase this, ATM it is a copy paste from the blog Galene provides the ability to assign a static rank to each entity. This is a measure of the importance of that entity independent of any search query, and is determined offline during the index building process. Using the static rank, we order the entities in the index by importance, placing the most important entities for a term first. The retrieval process can then be terminated as soon as we obtain an adequate number of entities that match the query, and not have to consider every entity that matches the query. This strategy is called early termination [**? ? ?**]. Early termination works if the static rank of

an entity is somewhat correlated to its final score for any query. The main benefit of early termination is performance, scoring is usually an expensive operation and the fewer the entities scored the better. Looked at in another way, early termination allows us to use more sophisticated scorers. Consider a broad query term like "marketing". Assume that we are looking for the term in the job description, a traditional query would look like this:

```
+DESCRIPTION: marketing
```

The above query implies the following - *retrieve those documents that MUST have the term "marketing" in the field description.* Note that this would retrieve prohibitingly high number of documents. Let's say we only rank *numToScore*. documents. We could rewrite the query using early termination as:

```
+DESCRIPTION: marketing[numToScore]
```

The above query implies - *retrieve those documents that MUST have the term "marketing" in the field description, but stop after retrieving the first numToScore documents*

For the rest of the paper we shall refer to early termination as *numToScore*.

4.3 WAND and FLEX operators

Our retrieval engine gives us the capability to do two powerful operators - WAND and FLEX.

Consider the simple example where the index has three fields - TITLE, DESCRIPTION and SKILL. Assume that the query is "marketing". One way to rewrite this query is:

```
+(?TITLE:marketing ?SKILL:marketing ?DESCRIPTION:marketing)
```

The above query would imply fetching documents which match the term "marketing" in *any* of the fields and let the ranker identify the top documents. This could be prohibitingly expensive as it may retrieve irrelevant documents. Suppose we have a way to construct the following query *retrieve all documents which match at least one of the two fields in (TITLE, SKILL, DESCRIPTION)*, it would significantly reduce both the number of irrelevant documents as well as latency. It will reduce the work load on the ranker.

WAND operators provide such flexiblity [**?**]. More formally assume that $X = (x_1, x_2 \dots x_n)$ represents n query operators associated with weights $W = (w_1, w_2, \dots w_n)$. WAND($X$, W) is true iff:

$$\sum_{1 \le i \le k} t_i w_i >= \theta \tag{2}$$

t_i's are Boolean variables that are set to 1 if clause x_i is satisfied and 0 otherwise.

For the query "marketing", consider a WAND query of the following form:

X = `(?TITLE:marketing, ?SKILL:marketing, ?DESCRIPTION:marketing)`

W = (1, 1, 1)

θ = 2

In the WAND formulation shown above, **at least two clauses out of the fields must match** for the document to be retrieved. This significantly reduces the number of irrelevant documents retrieved. We will later describe how we train the weights of the WAND clauses using decision trees.

To motivate the need for FLEX operator, consider the example where the query is "oracle". We understand that the term "oracle" represents a company with very high probability. So, the query could be written as:

```
+COMPANY:oracle[1000]
```

The above query retrieves at most 1000 documents where the company field MUST match the term "oracle". Rewriting the query to retrieve only documents which match the company field assumes that we got the intent of the user right 100% of the time. In this case, sometimes the user may have meant "oracle db" or something else. Let's say that the user intent while typing a query like "oracle" is to find jobs at the company Oracle 99% of the time and mean something else 1% of the time. FLEX operator provides the flexiblity to handle such cases.

More formally given k conjuntive query clauses $X = (x_1, x_2 \dots x_k)$ and k weights $W = (W_1, W_2, \dots W_k)$, let d_i represent the number of documents where the clause X_i is satisfied, then retrieval will continue until the following conditions are met or all documents are considered:

$$d_i \ge w_i \quad \forall 1 \le i \le k \tag{3}$$

In the example for "oracle", let us assume that the query represents a company *Oracle* with 99% probability and skill with 1% probability. Then we could set:

X = `+(?COMPANY:oracle, ?SKILL:oracle)`

$W = (990, 10)$

This would ensure at least 10 documents retrieved where the term "oracle" is a match in the skills field.

4.4 Replay Framework

At the core of the query construction process is an offline replay framework shown in Figure [**??**]. This helps us in validating how we are doing compared to the baseline candidate selection model as well as to only take the best and validated queries to production. We document the individual components that are part of the replay cycle and show examples from job recommendations vertical.

4.4.1 Query, Impression and Activity Tracking. For the replay of the queries we require support to track the query that was done online along with the results that were retrieved, ranked and shown to the members and guests. The tracking requires the exact retrieval query to be tracked. The exact query allows us to make modifications to query construction and use the same in the replay framework. This removes the need for regenerating the query with all member and job features and be sure what query was used to generate the tracked results.

The tracked results are annotated with the specific actions that are needed for evaluation in the replay framework. As an example in case of job recommendations we track job views and applications, and dismiss actions for each of the tracked job recommendations.

4.4.2 Offline Replay. The offline replay can be broken down into the following steps:

- Online data aggregation
- Query log preparation
- Query log replay

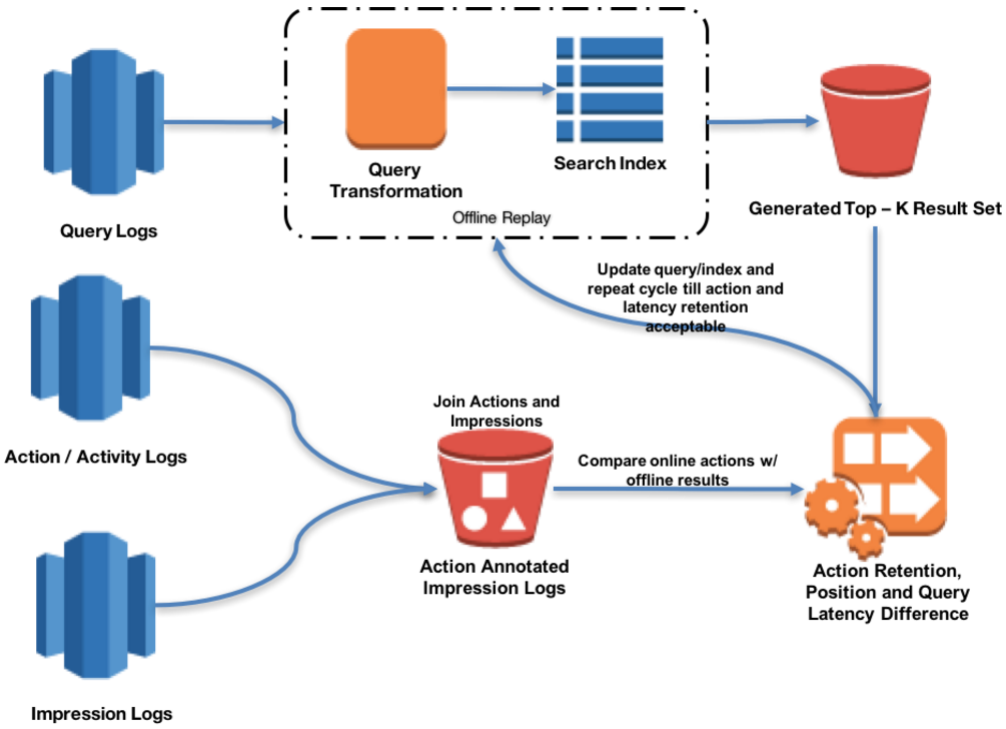

Figure 3: Replay Framework

- Analysis of replay results

It utilizes the online query logs and runs a transformation to allow operations such as:

- Query construction
- Ranking models
- Operational flags (retrieval flags) what is this?

The original queries are parsed through the query transformer and new candidate selection model is added into the query config. As the next step we take both the modified queries as well as the original queries and replay them on hadoop using offline retrieval infrastructure (known as **offline Galene**)

The results collected through the replay are then passed to a result comparator which goes through both the baseline and new ranklist and computes the action retention metrics. As an example in the case of job recommendations we compute.

- Job View/Application/Dismiss Retention
- Number of hits scored

As an additional step we also compute the difference between the baseline and modified queries to see what jobs that had actions were missed. This allows us to revisit the candidate selection model to debug why we missed retrieving the jobs.

The workflow is implemented in a generic manner with the workflow implemented in Hadoop and can be utilized for tasks outside of query construction such as training data generation, model validation and ranklist metrics computation.

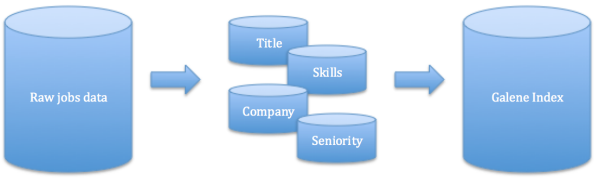

Figure 4: Offline Indexing

5 QUERY CONSTRUCTION ALGORITHM

In this section we describe the core algorithm used to for query construction. The entire flow is depicted in Figure ?? and detailed below. The algorithm consists of two steps:

(1) Construct decision tree using offline replay data
(2) Convert the decision tree into query clauses with weight. These weights will then be used in the WAND operators described in Section [??].

The process of decision trees and extracting clauses can be iterative. The depth of the decision tree is a hyper parameter which can be tuned based on the output of the clauses. We will describe details in this section.

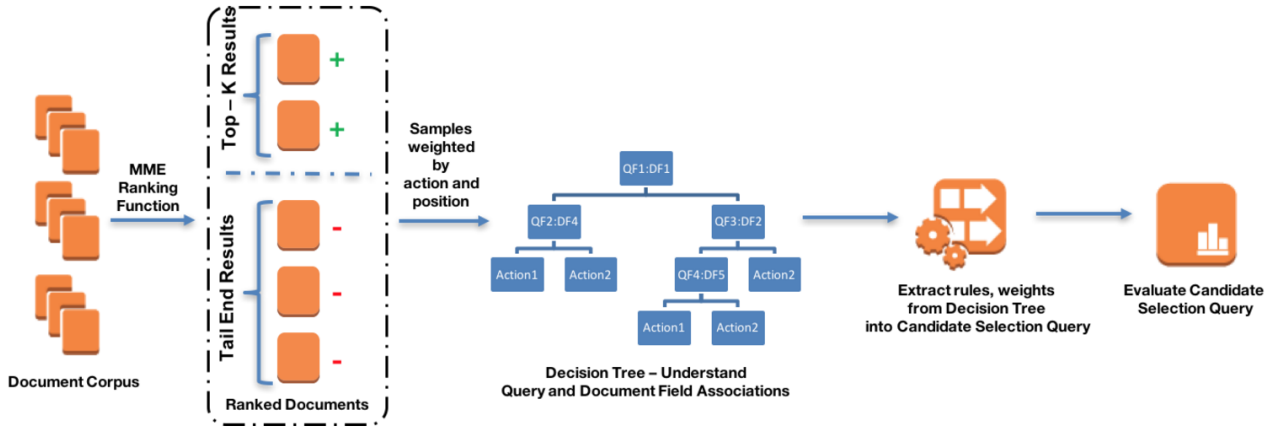

Figure 5: Offline Training Process

5.1 Decision Tree Construction

The decision tree construction which in turns can be used for query construction is described in Algorithm **??**. The branching variables in the decision trees are all features described below. Consider J_f and M_f to be relevant fields in job details and member profiles respectively. All *cross matches* in $J_f x M_f$ are potentially cross features which can be used as query clauses as well as branching points in decision trees. Example, consider field in the job called *Job Title* and a member profile field called **Member Current Title**, a potential feature would be *Job Title matches member current title*. Other similar examples include:

(1) Job location matches member profile location
(2) Job company size matches member preference for company size

Let F represent the cumulative set of such features. We first sample set of queries from production (say Q). We then run each query in this set through our offline replay framework. Offline replay ranks the document set as well as extacts the value of features in the feature set F. Assuming that the original ranking is not bad (a reasonable assumption), we could use the top k_1 results as positive and bottom k_2 as negative. We then have a dataset of features and labels which can be run through pretty much any decision tree based algorithm [**?**] to extract a tree.

5.2 Query Construction

The entire algorithm to construct the queries is shown in Algorithm **??**. The input to the algorithm is query logs and some hyper parameters. These hyper parameters control the depth of the tree as well as the number of positive and negative labels needed to get an appropriate query clause. Like any decision tree, we expect to know the number of positive and negative labels at each leaf node. The algorithm goes through a loop terminating when a suitable solution is found or maximum tree depth is reached. We start with min depth and construct a decision tree. Once the tree is constructed, we enumerate all paths and cut off at a point when the number of positive samples (at leaf nodes of each path) is more than a certain

Algorithm 1 Decision Tree Construction

1: **Input**: User and query logs with queries, member activity, member profile and job details.
2: **Output**: Decision tree represented by Graph $G(V, E)$. Each vertex $v \in V$ represents a feature. Each edge $e \in E$ to be represented by the tuple (u, v, d, w) where (u, v) represent nodes, d represents the decision and w represents the weight.
3: Derive feature set F which consists of cross query level cross features between job details and member profiles
4: Sample queries from production into query set Q
5: **for** query $q \in Q$ **do**
6: Run q through offline replay, generate top K results
7: Extract features for each result
8: Mark top k_1 results as positives
9: Mark bottom k_2 results as negatives
10: Add both positives and negatives to training set T
11: Run any standard decision tree algorithm on training set T to get decision tree graph $G(V, E)$. Use entropy at the decision tree edges to derive weight of each edge.
12:

a fraction of the total positives. We also need to make sure that the number of negative samples is low enough. If the constraint for both positive and negative samples is met, we stop and return the path. If not, we try to build a deeper tree.

example sample tree and how paths are converted to clauses

To give a general idea of how the queries are constructed we can look at a sample decision tree in Figure **??**. Lets assume total positive coverage α is 0.06 and hyper parameter β is 0.04. We describe the query construction in the following steps:

(1) Lets say we have a member M with following fields:
- Current Title = Software Engineer
- Member Seniority = Entry Level
- Member skill = Computer Science

(2) We use standard depth first search on our decision tree to extract all the paths starting from root to leaf. We will refer to this set as S_c.
(3) S_c is then sorted by number of positives labels that each path holds. We define the number of positives in a set by P_+ and number of negatives by P_-
(4) We iterate over the set S_c to find a subset S'_c which satisfy the constraint that P'_+ is greater $\alpha * P_+$ and $\frac{numNegativeLabels(P_-)}{numPositiveLabels(P_+)} \leq \beta$.
(5) In our example the paths indicated in green color satisfies the above constraints, since $P'_+ = 5000$ is greater than $P_+ * \alpha = 4320$ and $P'_-/P'_+ = 0.03 \leq \beta$
(6) All the paths in subset S'_c can then be constructed as a OR lucene query where each path is an AND of features. In our example we have total four paths starting from root to leaf and only two paths marked in green color satisfy the constraints. Constructed query CQ formed using set of all valid paths S'_c and a member M will look like the following:
    - `OR((jobTitle:Software Engineer AND jobSkill: Computer Science) ( jobTitle:Software Engineer AND -jobSkill: Computer Science))`

Algorithm 2 Query Construction

Input: Query logs Q_l with member activities, job details and member profile. α hyper parameters representing percentage of positives we want to cover. β hyper parameter representing the min ratio of positives to negatives. $h_{max}(h_{min})$ represents the maximum (minimum) depth of the decision tree
Output: Query clauses for search or recommendations
function CONSTRUCTCLAUSES(α, β, h_{max})
    $h \leftarrow h_{min}$
    while True **do**
        $G(V, E) \leftarrow$ DECISIONTREELEARNER(Q_l, h)
        $P \leftarrow allPaths(G)$
        Sort P by numPositiveLabels
        $P_+ \subseteq P$ such that numPositiveLabels(P_+) $\geq \alpha * numPositiveLabels(P)$
        if $h \geq h_{max}$ or $\frac{numNegativeLabels(P_-)}{numPositiveLabels(P_+)} \leq \beta$ **then**
            break
        $h \leftarrow h + 1$
    Clauses = extractClauses(P_+)

5.3 Overall Algorithm

6 EXPERIMENTS

this section needs more details, ATM it only has the raw results
In this section, we describe the experimental results using the proposed query construction techniques. We used the techniques to run on two separate products - job search and recommendations. We present the effectiveness in latency reduction as well as quality.

6.1 Experimental Setup

As mentioned in Section [**??**], our underlying retrieval is powered by an inverted index. We have two separate indices for both job search and recommendations. The data required to train our models is stored in HDFS. Our training pipeline is written using a combination of tools including Apache Spark [**?**] and Photon [**?**].

Our entire experimentation process consists of the following steps:

(1) Come up with query construction using techniques described earlier
(2) Measure impact offline using the replay framework
(3) Launch it online for a small percentage of population
(4) Retrain underlying ranking model using the results from new query construction
(5) Launch model online with new query construction and retrained ranking model

We had two types of experiments - offline and online. Offline experimentation uses the framework described in Section **??**. In this framework, we have ways to run a representative sample of production queries through the new and old query construction. All online experiments were conducted using LinkedIn's A/B testing framework [**?**].

Retraining the underlying ranking models is done via our model training flow details about which can be found in [**?**] and [**?**] for jobs recommendations and search respectively. We use user actions to infer labels in our models. These include:

- Job clicks
- Job applies (typically stronger intent)
- Job dismisses. Our UI allows a user to dismiss a job from our jobs home page. This is generally signal for poor match

6.2 Jobs Recommendations

Offline replay framework was used to sample a bunch of queries from the query log and ran it through two different systems - one that uses old query formulation and another one that uses the new proposed technique. We use the framework to train the decision trees as described in Section 5.

For jobs recommendations, we use above labels inferred from member activities to build a deeply personalized model. The objective of the model is to predict if a member m will apply for a job j in context t. Let q_m and s_j denote the feature vector extracted from the member profile m and job j respectively. Let x_{mjt} represent the overall feature vector for the the (m, j, t) tuple. We use logistic regression to predict the likelihood of member applying to job by:

$$g(E[y_{mjt}]) = x^T b + s_j^T \alpha_m + q_m^T \beta_j \quad (4)$$

In the above equation:

- $g(x) = log\frac{x}{1-x}$.
- b represents the global coefficient vector or simple the *global model.*
- α_m and β_j are the coefficient vectors of member m and job j respectively. These coefficients are also called as random effects coefficients and enable deep personalization.

The details of the ranking model can be found in [**?**]. Once we obtain a decision tree, we retrain the model from activity data

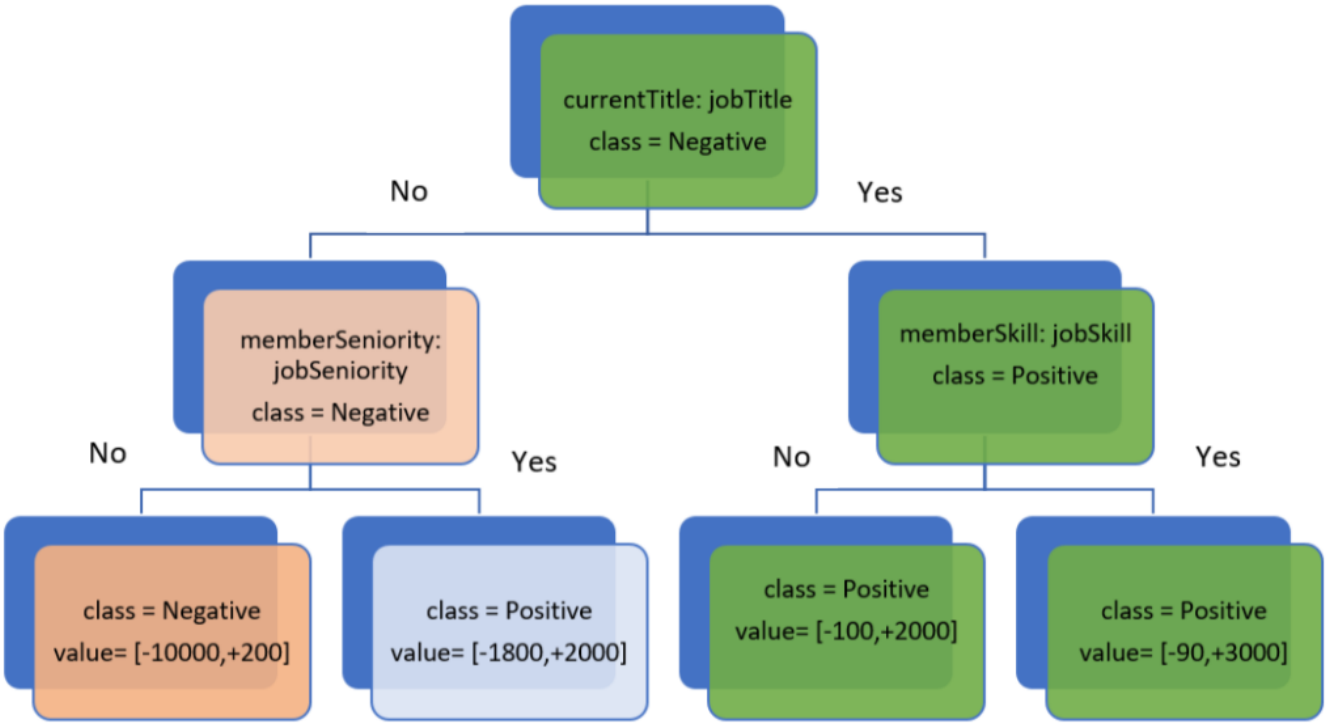

Figure 6: Decision Tree

obtained from the new model. This process is critical for both search and recommendations. We have noticed that changing query formulation often requires us to retrain the underlying ranking model. Note that we haven't changed anything in the ranking model, but only retrained it using new data.

The main goal of the work is to reduce latency. Within latency, we measure average, p95 (which represents the worst 5% of queries) and p99 (worst 1% of queries). Operational requirements are dictated more via the worst queries than average case and hence the need to measure p95/p99

Table [??] gives the p99 latency reduction while running our decision tree based approach. The baseline approach we used was also a machine learned algorithm described in [?]. As seen from the table, we notice significant reduction in latencies across the spectrum. Note that these numbers were obtained by A/B testing online with a sample of users directed to the new query construction techniques. These are true latencies measured in our backend servers. Of particular interest in the performance on the worst 1% of the queries where we reduced latency by over 55%. Average latencies were down by 35.6%.

write stuff about feed

6.3 Job Search Experiments

Table [??] gives the p99 latency reduction while running our decision tree based approach. The baseline approach we used was also a machine learned algorithm described in [?].

7 DEPLOYMENT LESSONS

this section needs more details

When we changed the query construction, our first set of experiments were not successful. While debugging the issue, we found that ranking also changed significantly. This is due to the fact that the underlying model was trained on data generated by the a different query construction. We then retrained our model and found it to be successful. Whenever we change pre retrieval stage significantly, we need to retrain the model. This pattern has been observed repeatedly.

8 CONCLUSIONS

Table 1: Impact of Query Construction on Jobs Recommendations Latency

Metric	Baseline	Decision Trees	Percentage Difference
Average Latency	37.67 ms	24.26 ms	**-35.6%**
p95 latency	163.2 ms	109.9 ms	**-48.49%**
p99 latency	562.4 ms	247.7 ms	**-55.96%**

Table 2: Impact of Query Construction on Job Search Latency

Metric	Baseline	Decision Trees	Percentage Difference
Average Latency	108.0 ms	101.0 ms	**-6.48%**
p95 latency	629 ms	214 ms	**-34.55%**
p99 latency	1921 ms	620 ms	**-67.72%**