Surrogate Ranking for Very Expensive Similarity Queries

Fei Xu [1], Ravi Jampani [2], Mingxi Wu [3], Chris Jermaine [4], Tamer Kahveci [5]

[1,2,4,5]*CISE Department, University of Florida, Gainesville, FL, 32601, USA*
[3]*Oracle Corp. Redwood Shores, CA, 94065, USA*
[4]*Computer Science Department, Rice University, Houston, TX, 77005, USA*
{feixu,rjampani,mwu,cjermain,tamer}@cise.ufl.edu

***Abstract*— [1] We consider the problem of similarity search in applications where the cost of computing the similarity between two records is very expensive, and the similarity measure is not a metric. In such applications, comparing even a tiny fraction of the database records to a single query record can be orders of magnitude slower than reading the entire database from disk, and indexing is often not possible. We develop a general-purpose, statistical framework for answering top-k queries in such databases, when the database administrator is able to supply an inexpensive surrogate ranking function that substitutes for the actual similarity measure. We develop a robust method that learns the relationship between the surrogate function and the similarity measure. Given a query, we use Bayesian statistics to update the model by taking into account the observed partial results. Using the updated model, we construct bounds on the accuracy of the result set obtained via the surrogate ranking. Our experiments show that our models can produce useful bounds for several real-life applications.**

I. Introduction

Traditionally, two assumptions have governed the design of algorithms for "best match" or similarity search in databases. The first is that the cost of determining the similarity of a database record to a query is negligible once a record is read into main memory [20], [2]. The second is that the distance measure between two records is a metric or near-metric (i.e., the triangle inequality can be violated only up to a given amount) [21], [7]. If the second assumption holds, then indexing is usually a viable option [2], [4], [21], [7], [23]. If the first holds, then in the worst case a sequential database scan can always be used [9].

Unfortunately, these assumptions do not hold for many emerging applications. For instance, in the following applications, computing the distance/similarity between even two records is exceedingly expensive.

- Protein structure search. When the 3-dimensional structure of a new protein is determined, the next task is often to check if there are any known proteins in a protein structure database that have a similar 3-dimensional shape. Such similarities can help in predicting functions of the newly found protein [11]. Here, "similarity" is typically defined as the best root-mean-square deviation (RMSD) between the atoms in the two proteins after one of them is rotated and translated to superimpose on the the other one in the best possible way [16] or a statistical measure such as the Z-score of the alignment. Finding a good superimposition between two proteins is a difficult problem. Existing heuristics, such as CE [22], take ten seconds to a minute for a pair of proteins. Furthermore, RMSD or Z-score can violate the triangle inequality.
- Drug molecule search. Once a protein or enzyme is determined to be a drug target, an important problem is to find the drug molecules that have highest affinities to the target among all drug molecules [12]. Existing algorithms, such as DOCK [5] and Glide [10], compute the affinity of a drug molecule to a pocket in a protein or enzyme by fitting the drug molecule's structure to that pocket for many alternative orientations of the drug molecule. Glide takes five to ten minutes to compute the affinity of a single drug molecule to a protein. Again, the affinity measure is not a metric.
- Pathway search. Aligning two pathways to find the similarity between them is related to the subgraph or graph isomorphism problem. Such an alignment helps in functional annotation of paths [13]. Computing an alignment for a pair of pathways takes from a few seconds to a few minutes, depending on their size and complexity [19].

In addition to the extreme cost of comparing two records, the similarity measures in the above examples are not metric. Hence, indexing these databases is difficult. Perhaps the only available method for finding records similar to a query is to compute the similarity of each database record one by one. The problem with this approach is not the cost of reading the database. For example, the 3-dimensional structure of all the proteins in PDB takes up less than 250 MB. The difficulty is the tremendous expense of each application of the similarity computation software. For example, 30 seconds per protein similarity computation translates to weeks of computer time to search the PDB.

Problem statement. In this paper, we study similarity search when there is a computationally expensive, possibly non-Euclidean ranking function R that may not satisfy the triangle inequality and cannot be effectively indexed. Given a query record q and a ranking function R, our goal is to find the k items in a database that are closest to q, where "closest" is defined as the set of database objects D that minimize (or maximize) the value $R(q, d)$ for $d \in D$.

We develop a general-purpose framework for answering such queries efficiently, using virtually any expensive ranking function (i.e., similarity measure), as long as the database administrator can choose a *"surrogate"* ranking function R'. A surrogate ranking function is one that is

(a) computationally efficient, and
(b) tends to produce the same ordering of database objects as the original ranking function R.

Supplying an appropriate surrogate ranking function is often quite easy for a domain expert. For example, in the 3-dimensional protein structure alignment application, a good surrogate ranking function $R'(q, d)$ is the similarity of q and d found by Blast [1]. Blast is a sequence comparison software which measures the

[1]Material in this paper was supported by the National Science Foundation under grants 0803511, CCF-0829867, DBI- 0606607 and IIS- 0845439 and by a Sloan Foundation Fellowship.

(1, 89) (2, 78) (3, 82) (4, 80) (5, 65) (6, 53) (7, 62) (8, 71) (9, 43) (10, 47) (11, 51) (12, 48)

Fig. 1. (Score, rank) pairs for a database of twelve records, sorted according to the surrogate ranking function R'. The first number in each pair is the rank of the record, according to R'. The second number is the similarity of that record to the query q, according to the expensive similarity (or ranking) function R. R' is computed for all the records. However, in this example, $k' = 5$, so R is computed only for the underlined five records. In this example, if $k = 3$, then the surrogate ranking function has correctly returned three out of the top three results (since 89, 82, and 80 are the top three records, according to R).

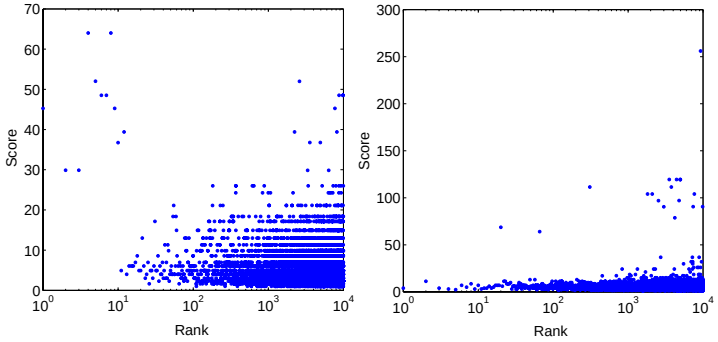


Fig. 2. Examples of two protein-matching queries. The surrogate ranking produced by Blast is plotted against the $e^{Z-score}$ of the CE alignment. The rank (horizontal) axis is in log scale. For the left query, there is a strong relationship between Blast and CE, and the top 10 Blast results are all in the top 20 CE results. The relationship for the right query seems to be mostly random.

extent to which the sequence of residues in the proteins are similar. This makes sense because Blast is computationally very efficient. Blast aligns a query protein with a database of 10,000 proteins in a few seconds, while it takes days to align the structures of the proteins in the same database using CE on a single computer. Also, the sequence similarity of two proteins has close relationship with their 3-dimensional structural similarity. This is because the sequence of amino acids of protein is the main determinant of its 3-dimensional structure.

Given an appropriate surrogate ranking function R', the algorithm that we propose to answer a top-k query is simple:

1) First, find the top k' matches for q using the surrogate ranking function R', for some $k' > k$.
2) Next, compute the top k matches for q using the actual ranking function R, considering only the k' objects returned in step (1).
3) Return the set of k objects obtained from step (2), as well as probabilistic guarantees on result accuracy, to the user.

The general process is illustrated above in Figure 1. The probabilistic guarantee in step (3) above takes the general form:

"With probability p, at least h of the true top k items are present in the result set."

If the user is not happy with this result (because p is too low for a given h), then k' can be made larger by applying R to more objects from the database, and p will grow. This can even be done anytime, as the user waits. Such an online computation is immensely preferable to waiting weeks or months for a single, monolithic computation to complete. Typically, k is much smaller than the database size. Therefore, if R' is chosen appropriately so that it has a close relationship with R, then the user may be able to end the computation in only a fraction of the time required to process the entire database.

Figure 2 illustrates this on a real-life example. In this example, the goal is finding the few proteins closest to a query protein (in terms of RMSD) in a database of 10,000 proteins. The surrogate ranking function is provided by Blast. For the query on the left, Blast does quite well. The top two matches in the entire database according to RMSD are among the first ten proteins returned according to Blast's surrogate ranking. Also the top ten proteins with the best RMSD are all in the top 20 proteins according to Blast's ranking. In this case, our algorithms would ideally inform the user that after a few dozen proteins have been examined, there is a very high probability that the top one or two proteins according to RMSD have been returned.

For the query on the right in Figure 2, Blast does poorly. The top protein according to RMSD is near the end of the Blast ranking, and the set of top proteins according to RMSD seem to be scattered randomly. In this case, our algorithms would ideally inform the user that after many proteins have been examined, there is still a good chance that the top proteins have not yet been returned.

Once a surrogate ranking function is determined, the algorithm itself is almost trivial. However, providing mathematically meaningful, probabilistic guarantees on the accuracy of the result set that has been returned remains a difficult task. *The guarantees should be accurate whether R' is a high-quality surrogate for R or not.* After all, both R and R' are arbitrary, domain-specific, and user-supplied. So the framework that we develop cannot rely on any specific property of these two functions.

Our contributions. We develop a general-purpose statistical framework that learns the relationship between R and R', and uses it to provide the user the required guarantees after a subset of the database is investigated. Our framework is generic as it does not require that these ranking functions follow metric rules. It uses a set of previously-answered training queries to model mathematically how predictive R' is for R. When a user issues a query q, it uses this model to provide probabilistic guarantees to the result that is returned to the user. Over time, as R is applied to more database records ranked by R' for the particular query point q our understanding of the relationship between the ranking of R' and R improves. For example, it may become clear that for q, R and R' are not as closely related as predicted by the model — or, it may become clear that R and R' are exceedingly close for this particular q. We use Bayesian statistical techniques to update the model in the light of the observed partial results, as the query is answered. In this way, the model is tailored to the particular q in a mathematically rigorous fashion, as the query is processed. We show in our experiments that our methods can produce highly accurate bounds, in the sense that the user can rest assured that if the software claims that there is a probability p of having enough true results in the answer set, the actual probability is indeed p.

The following summarizes our technical contributions:

1) We propose a unique statistical model for the relationship

between a user-defined surrogate ranking function R' and a true ranking function R.
2) We derive learning algorithms for the model, in the form of an expectation-maximization- (EM-)based [15] maximum likelihood estimation (MLE) [6].
3) We provide algorithms for updating the model in response to the partial computation associated with an actual query q, and for using the updated model to bound result set accuracy.
4) We experimentally evaluate our algorithm in several real application domains.

The rest of the paper is organized as follows. Section II presents an overview of our method. Section III describes our mathematical model. Section IV discusses how we learn the parameters of our model. Section V discusses how we update the model parameter for a query. Section VI discusses our algorithm for computing the bounds for the results reported. Section VII presents the experimental results. Section VIII concludes the paper.

II. Overview of the Method

At a high level, our goal is to provide a probabilistic guarantee on the quality of the set of k database objects that are returned to the user. We propose a unique statistical approach to this problem. Our solution avoids making assumptions on the intrinsic properties of the ranking function R (such as whether it is a metric measure). We instead rely on its extrinsic, statistical properties — specifically, how it observably relates to its surrogate R'. We are aware of no similar research in the computer science literature.

Our approach first learns a set of query classes from the historical query workload, each of which, in principle, describes a possible relationship between R and R'. Given a new query, our method then determines which class of queries it belongs to, and then uses that class membership to predict the accuracy of the surrogate ranking on the current query. Our approach has four steps:

1) We define a parametric, statistical model for different classes of queries that a user may postulate. Each class of queries includes a model for the relationship between R and R'.
2) We derive a method to learn the entire model from a historical query workload, so that it can be tailored to a specific R' and R for a specific database. In this context, "learning" the model refers to the process of choosing the set of model parameters so that the result accurately describes the historical workload. Note that the models will include *all* types of historical queries. For some of those queries R' may be a poor surrogate, while for others it may be excellent.
3) After processing a small fraction of the database for an actual query, we update the model. This is because the underlying model is generic, and covers all possible types of queries. For an actual query where it is clear that R' is (or is not) a reasonable surrogate for R, a generic probabilistic guarantee may be too loose (or too tight). To deal with this, we update the model as we start evaluating an actual query. We use the portion of the database for which both R and R' are computed to alter the model in Bayesian fashion [18]. For example, if R' has served as a close surrogate for R so far, the updated model will prefer the query class for which R' works well. On the other hand, if R' has done a very poor job so far, the updated model will tend to increase its "belief" in the possibility that R' is poor for this query.
4) Finally, we use the updated model to compute the probability that the k answers returned to the user thus far contain at least h of the actual top-k database objects, and this information is returned to the user.

Underlying Assumptions. We end this overview section with a brief discussion of when our methods will tend to work well, and when they might not.

The utility of our approach is measured by its ability to provide for accurate guarantees. If our algorithms tell a large number of different users that there is a $p \times 100\%$ chance that at least h of the top-k database objects are present in the the query result set, then for $p \times 100\%$ of those users, at least h of the top-k database objects should be in the result set. Thus, a very reasonable question is: What assumptions are required for the mathematics we employ to guarantee this sort of correctness? The answer is simple: if the model learned in step (2) above correctly describes the process by which future queries are generated, then the bounds will be correct.

There is one major way in which the learned model may be flawed: when the past query workload is not a reasonable predictor of the workload in the near future. Although our methods can adapt to shifts in the distribution of queries, pathological cases will cause problems. For example, if for each and every training query, R' and R have a very close relationship, then the learned model will not admit the possibility that R' could do a poor job. But if R' can perform very poorly in some future case, our method will be unable to recognize this, and mislead the user.

That being said, the advantage of our algorithm is that even if the ranking defined by R' and R differ greatly, the bounds it computes will be appropriately wide as long as this distribution is modeled by at least one of the learned query classes.

Why Model Rank vs. Score? In this paper, we attempt to predict the score of an object, given its surrogate rank. A reasonable question is, why not model its score, given its surrogate score?

We could have done this, and indeed, all of the algorithms we suggest in this paper could be trivially adapted to a score vs. score, rank vs. rank or score vs. rank model. In fact, as future work we plan to investigate which method is preferred. However, there is an obvious reason that one would initially prefer rank vs. score: ranks are scale-free. Imagine two queries, one with surrogate scores that range from 10 to 20, and another with surrogate scores that range from 10 to 10000, but where both sets of surrogate scores produce the exact same ranking. Both queries will look exactly the same if one models rank vs. score (and thus they can share the same model), but will look radically different if one models score vs. score. This means that two different models must be used to capture both queries. Effectively, using rank vs. score removes a parameter from the model—the scale of the surrogate score.

Notation. For simplicity we will use the following convention for our mathematical notation in this paper unless otherwise stated. We will use capital letters for sets, random variables

TABLE I

COMMONLY USED SYMBOLS IN THIS PAPER.

q, d	query record, database record
N	number of database records
M	number of observed query results
k, k', h	parameters of top-k query.
$R()$, $R'()$	actual and surrogate ranking functions
r_i, r	ranks obtained by R'
s_i, s	scores obtained by R
c, g	number of query classes, Gaussian components
α_i, β_j	weight of a query class, Gaussian
$F_*()$	probability density function
Θ_q, Ψ_j	parameter set for a query class, Gaussian
$L()$, $Q()$	likelihood and Q-functions

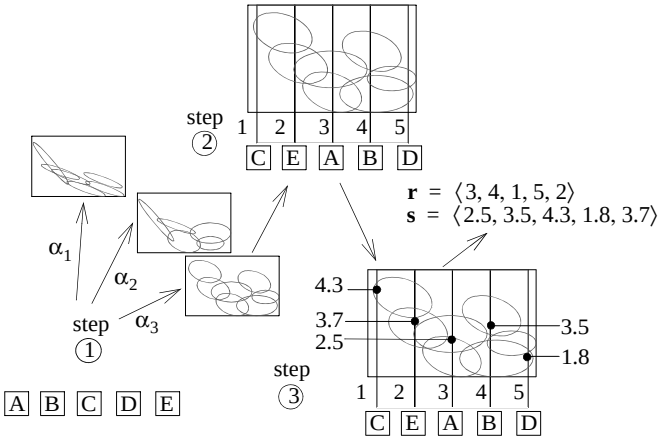


Fig. 3. An illustration of the stochastic process that governs the generation of a set of ranks and scores for a user's query over the database containing objects A, B, C, and D. First, a query class is assigned to the query ; the probability of assigning class i is α_i (step 1). Next, the database objects are ranked using the surrogate ranking function (step 2). Finally, the ranks are used to stochastically assign scores to the database objects, conditioned on the assigned ranks (step 3). This results in two vectors describing the user's query result set: a vector of scores for the objects, and a vector of ranks.

and functions. Lower case letters will be used for variables. We will use bold-face letters to denote vectors and matrices. Table I summarizes the most frequently used symbols in this paper.

III. THE MODEL

In this section, we define the mathematical model that forms the basis for our algorithms. Without compromising the generality of the problem definition, assume that the actual ranking function is a similarity measure; that is, a large value of R indicates high similarity. If R is a distance function so that small values indicate best results, then the negative value of R can be used instead.

A. The Generative Process

The first step is to postulate a generative, statistical model for the process of answering a top-k query using the surrogate ranking function R'. The reason that we need a model is that by tailoring the model to the historical workload, it will subsequently allow us to learn how accurate (or inaccurate) the process of ranking via a surrogate ranking function is. This will allow us to make predictions regarding the possibility that by only considering the highest ranked database objects according to R', we might miss those objects with a high score according to R.

The model will take the form of a probability density function (PDF) which can be manipulated mathematically. The PDF

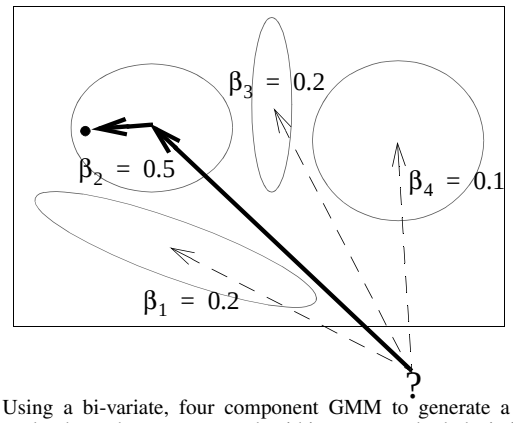


Fig. 4. Using a bi-variate, four component GMM to generate a data point. The four ovals show the area covered within one standard deviation of each normal/Gaussian component. To generate a point, a component is first selected at random (in this example, the most likely one has been selected) and then that component is used to generate the data point.

describes the stochastic process of a user first issuing a query, ranking all of his or her database object using the surrogate ranking function R', and then assigning scores to the database objects using the actual ranking function R. The generative process is illustrated pictorially in Figure 3. In this figure (and in the rest of the paper), **r** denotes the vector of ranks for database objects, and r_i denotes denotes the rank of the i^{th} database record according to the surrogate ranking function R' for a given query. For example, if the ith database record is the third closest database record to the query according to R', then $r_i = 3$. **s** is the vector of scores according to the expensive function R, and we use s_i to denote the actual similarity of the i^{th} database record to the query; the top-k query attempts to find records with the k largest s_i values.

The specific steps in the generative process are:

1) A user first poses a query q. There are c different query classes that this query can belong to. Depending upon which class it belongs to, there may be a different relationship between R and R'. The specific class is determined by rolling a biased die (step 1 in Figure 3). We denote the probability of choosing the i^{th} query class by α_i, $\forall i \in \{1, 2, \cdots, c\}$.
2) We rank the N database objects with respect to the query using the surrogate function $R'(q, d)$. This is a deterministic process that assigns the labels $\langle 1, 2, \cdots, N\rangle$ to the different database objects. The result of this is the vector **r** (see step 2 in Figure 3).
3) Let $F_s(s_i|r_i)$ be a function that describes in a probabilistic sense the relationship between s_i and r_i. We obtain the actual score $R(q, d_i)$ for the i^{th} database object by sampling a single score value from $F_s(s_i|r_i)$, where the likelihood of generating a specific score depends upon the rank. We do this for all database objects. As depicted in Figure 3– step 3, there is a different form of F_s associated with each query class — for some query classes, a rank close to one may always mean a very high score; for others, the rank and the score may be less closely related.

B. Generating a Score, Given a Rank

Given such a setup, the big remaining question is regarding the specific form of F_s. There are many possibilities for this PDF. We use a standard, two-dimensional Gaussian (normal) mixture model (GMM) [14] as the basis for F_s. Figure 4 illustrates the basic process of using a GMM to generate a data point. This process works as follows:

- Assume that there are g Gaussians for a given GMM. First, one of the g GMM components is chosen at random, the probability of choosing the j^{th} component is β_j.
- Next, a two-dimensional point (r, s) is sampled from the normal distribution $F_N(s, r|\Psi_j)$. In this formula, F_N is the two-dimensional normal PDF, and Ψ_j is the set that contains the means, variances, and covariance for the j^{th} normal PDF.

There are many reasons to choose a GMM. GMMs are very flexible, and can model very complex, multi-modal, multi-variate distributions. Even though one of our dimensions is discrete—a GMM is still the most natural choice. If we chose to use a discrete distribution for the object's surrogate rank and a separate, continuous distribution for the object's score, we would lose the ability to easily model the covariance between these two quantities on a per-component basis, which would greatly compromise the flexibility of the model.

In our model, there are c different GMMs, with one associated with each query class. Assume that query q belongs to the i^{th} query class. The GMM associated with q will have the parameter set Θ_i. Θ_i contains the means, variances, and covariances for each individual normal/Gaussian distribution in the GMM, as well as the weights (the β_j's) associated with each of the individual GMMs. Under this setup, the general form of the PDF for a g-component GMM is:

$$F_{GMM}(s, r|\Theta_i) = \sum_{j=1}^{g} \beta_j F_N(s, r|\Psi_j)$$

The only problem with employing a GMM as the basis of F_s is that F_s must describe the PDF for a score *given a rank*, whereas F_{GMM} describes the joint distribution of scores and ranks. However, a simple transformation can be used to obtain a PDF of appropriate form:

$$F_{GMM}(s, r|\Theta_i) = F_{GMM}(s|r, \Theta_i) F_{GMM}(r|\Theta_i)$$

So,

$$F_s(s|r, \Theta_i) = F_{GMM}(s|r, \Theta_i) = \frac{F_{GMM}(s, r|\Theta_i)}{F_{GMM}(r|\Theta_i)}$$

where $F_{GMM}(s, r|\Theta_i)$ is as given above, and $F_{GMM}(r|\Theta_i)$ is nothing but the one-dimensional GMM obtained by ignoring the score. That is,

$$F_{GMM}(r|\Theta_i) = \sum_{j=1}^{g} \beta_j F_N(r|\Psi_j)$$

C. Putting It All Together

Given this setup, we model the process for generating a single query result set $(\mathbf{s}, \mathbf{r})$ via the following PDF:

$$F_q(\mathbf{s}|\mathbf{r}, \Theta) = \sum_{i=1}^{c} \alpha_i \prod_{j=1}^{N} F_s(s_j|r_j, \Theta_i)$$

Let M be the size of the training data (i.e., then number of queries for which we know the value of the actual ranking function as well as the surrogate ranking function). Let us denote the entire historical query workload using the $M \times N$ matrix $\mathbf{S} = \langle \mathbf{s}_1; \cdots; \mathbf{s}_M \rangle$. Here, the vector $\mathbf{s}_i = \langle s_{i,1}, s_{i,2}, \cdots, s_{i,N} \rangle$ is the i^{th} row of the the matrix $\mathbf{S}$. It shows the scores of all the database records for the i^{th} query in the training data according to the actual ranking function. In other words $s_{i,j}$ is the score of the i^{th} query and j^{th} database record. $\mathbf{R}$ denotes the same for the surrogate ranking function. Then the PDF for an entire historical query workload is simply:

$$F(\mathbf{S}|\mathbf{R}, \Theta) = \prod_{m=1}^{M} F_q(\mathbf{s}_m|\mathbf{r}_m, \Theta)$$

In this expression, $\Theta = \{(\alpha_i, \Theta_i)|1 \leq i \leq c\}$ is the set of variables that contains all of the various α_i's, as well as the individual Θ_i's.

IV. Learning the Model

Now that an appropriate parametric model family has been defined, the next task is figuring out how to learn the model. That is, we need to be able to choose Θ so that the resulting model is a good fit for the observed historical query workload defined by $(\mathbf{S}, \mathbf{R})$.

There are two obvious ways to learn such a complicated model. One is to use a traditional maximum likelihood estimate (MLE) [6]; another is to rely on a more modern, Bayesian approach, such as a Markov Chain Monte Carlo (MCMC) method [8]. There are several pros and cons of each approach. We use MLE for one main reason: MLE procedures are generally much faster than MCMC methods and tend to scale to larger data sets. This is significant because the number of individual (score, rank) pairs in $(\mathbf{S}, \mathbf{R})$ can easily be in the millions, which is beyond what an MCMC method might be expected to handle for the sort of reasonably complicated, hierarchical model that we utilize.

A. Expectation Maximization

MLE aims to choose the parameters of a distribution that maximize the probability or likelihood that the observed historical query workload would have been produced. That is, given the log likelihood function:

$$\log L(\Theta, \mathbf{R}|\mathbf{S}) = \log F(\mathbf{S}|\mathbf{R}, \Theta) = \sum_{m=1}^{M} \log F_q(\mathbf{s}_m|\mathbf{r}_m, \Theta)$$

We need to solve the problem:

$$\text{argmax}_{\Theta} \{\log L(\Theta, \mathbf{R}|\mathbf{S})\}.$$

Unfortunately, this is quite difficult as we do not know which query class for each of the M different queries that are in the historical query workload. This is a classic "hidden data" problem. We address this problem using the well-known and widely-used Expectation Maximization (EM) algorithm [15]. The EM algorithm, rather than being a single "algorithm", is actually a generic framework for solving hidden data problems. In theory, EM can be used to derive a solution for any particular hidden-data MLE, though in practice utilizing EM for a specific MLE can be quite challenging.

A complete description of EM could span several papers, and so we refer to interested reader to the relevant literature on

the subject [15]. Rather than attempting to maximize the log likelihood function directly, EM works by repeatedly trying to maximize the *expected value* of the complete-data log likelihood function, where the hidden data are treated as a random variable whose values are conditioned on the current value for Θ. Specifically, let $\mathbf{q} = \langle q_1, q_2, \cdots, q_M \rangle$ be the unknown (hidden) set of classes for each of the M different historical queries. For example, if the first query belongs to the fourth query class then $q_1 = 4$. Let Θ' refer to the current guess for the value of Θ. EM works by repeatedly maximizing the value of the Q function:

$$Q(\Theta, \Theta') = E[\log F(\mathbf{S}, \mathbf{q}|\mathbf{R}, \Theta)|\mathbf{S}, \mathbf{R}, \Theta']$$

The Q function represents the expected value of the complete-data log likelihood function. Assume that the underlying query belongs to the i^{th} query class. Since $\log F(\mathbf{S}, \mathbf{q}|\mathbf{R}, \Theta)$ must take into account the likelihood of observing the vector $\mathbf{q}$, its expression is:

$$\log F(\mathbf{S}, \mathbf{q}|\mathbf{R}, \Theta) = \sum_{m=1}^{M} \log \alpha_i F_q(\mathbf{s}_m|\mathbf{r}_m, \Theta)$$

Given the Q-function, in pseudo-code, EM runs the loop:

```
start with an initial guess for Θ;
Repeat
   Θ' = Θ;
   maximize Q(Θ, Θ') wrt Θ;
Until (change in Θ is small)
```

B. Maximizing the Q-Function

In our case the Q-function can be re-written as:

$$Q(\Theta, \Theta') = \sum_{m=1}^{M} \sum_{i=1}^{c} (\log \alpha_i + \log F(\mathbf{s}_m|\mathbf{r}_m, \Theta_i)) Pr[q_m = i|\mathbf{s}_m, \mathbf{r}_m, \Theta']$$

Maximizing the Q-function with respect to Θ requires maximizing it with respect to each of $\Theta's$ parts: all of the α_i's and all of the Θ_i's. By taking the derivative with respect to each of the α_i's and by setting the result to zero, we can get the following simple maximization rule for each α_i:

$$\alpha_i \leftarrow \sum_{m=1}^{M} \frac{Pr[q_m = i|\mathbf{s}_m, \mathbf{r}_m, \Theta']}{M}. \quad (1)$$

In this expression, $Pr[.]$ is known as the "posterior probability" that the m^{th} query was produced by class i. We compute this as:

$$Pr[q_m = i|\mathbf{s}_m, \mathbf{r}_m, \Theta'] = \frac{\alpha'_i F_{GMM}(\mathbf{s}_m, \mathbf{r}_m|\Theta'_i)}{\sum_{j=1}^{\mathbf{c}} \alpha'_j F_{GMM}(\mathbf{s}_m, \mathbf{r}_m|\Theta'_j)}.$$

C. Handling the Model Parameters

Maximizing $Q(\Theta, \Theta')$ with respect to each Θ_i is a harder problem than maximizing the Q-function with respect to each α_i. To perform this maximization, we first observe that since the various α_i's and Θ_i's are added together in the Q-function, one can maximize all of the parameters separately and still obtain a global maximum. This observation simplifies the problem. Removing everything in Θ from the Q-function except for those terms depending upon Θ_i, we have:

$$Q(\Theta_i, \Theta') = \log \sum_{m=1}^{M} \log F(\mathbf{s}_m|\mathbf{r}_m, \Theta_i) Pr[q_m = i|\mathbf{s}_m, \mathbf{r}_m, \Theta']$$

Maximizing this function directly is still difficult. However, after some algebraic manipulation, we can rewrite it as:

$$Q(\Theta_i, \Theta') = \log \prod_{m=1}^{M} \prod_{j=1}^{N_m} F_s(s_{m,j}|r_{m,j}, \Theta_i)^{Pr[q_m = i|\mathbf{s}_m, \mathbf{r}_m, \Theta']}$$

where N_m is the number of (score, rank) pairs in the m^{th} query from the historical workload.

Given this rewriting, the Q-function is then reduced to the log of a product of a large number of individual likelihoods, each raised to a certain power. At this point, we can view the Q-function itself as a new likelihood function. We thus evaluate it over a new dataset that we "created" by adding $n \times Pr[q_m = i|\mathbf{s}_m, \mathbf{r}_m, \Theta']$ copies of the historical query $(\mathbf{s}_m, \mathbf{r}_m)$ to the new data set, for some very large n. It is worth saying that we do not literally create the simulated dataset. We conceptually assume that it exists for our mathematical derivation. We require a large n here only so that an integral number of copies of each query are added to the new, simulated data set; for very large n, rounding $n \times Pr[q_m = i|\mathbf{s}_m, \mathbf{r}_m, \Theta']$ to the nearest integer will have no effect on the result of the maximization.

Let the new, synthetic database D' be the multiset

$$\bigcup_{m=1}^{M} \bigcup_{j=1}^{N_m} \cdots \bigcup_{i=1}^{n \times Pr[q_m = i|\mathbf{s}_m, \mathbf{r}_m, \Theta']} \{(s_{m,j}, r_{m,j})\}$$

In this expression, $s_{m,j}$ is the j^{th} score value in the m^{th} query; $r_{m,j}$ is defined similarly.

Then for large n, we have:

$$Q(\Theta_i, \Theta') \approx \log \sqrt[n]{\prod_{(s,r) \in D'} F_s(s|r, \Theta_i)}$$

Since the n^{th} root of the multiplicative term is a monotonically increasing function, we can maximize $\log \prod_{(s,r) \in D'} F_s(s|r, \Theta_i)$ separately for each Θ_i and obtain the same result as if we had maximized the original Q-function directly. Let $\mathbf{S}'$ be the set of all score values in D', so that s_m refers to the m^{th} score in this set. Let $\mathbf{R}'$ be defined similarly, so that (s_m, r_m) is the m^{th} pair in D'. The problem of finding the maximum value of this function over all possible Θ_m is then itself equivalent to an MLE:

$$\text{argmax}_{\Theta_i} \{\log L'(\Theta_i, \mathbf{S}'|\mathbf{R}')\}$$

with

$$\log L'(\Theta_i, \mathbf{S}'|\mathbf{R}') = \sum_{m=1}^{|D'|} \log F_s(s_m|r_m, \Theta_i).$$

Again, this is a difficult problem because $F_s(.|\Theta_i)$ encodes the mixture model for the i^{th} class of query, and we do not know which mixture component produced the m^{th} point in D'. Let g_m denote the unknown identity of the Gaussian used to produce the m^{th} point in D'. Again, we can use an EM algorithm to solve this problem. For this second EM, we obtain the following Q-function:

$$Q(\Theta_i, \Theta'_i) = E[\log F_s(\mathbf{S}', \mathbf{g}|\mathbf{R}', \Theta_i)|\mathbf{S}', \mathbf{R}', \Theta'_i]$$

where $\log F_s(.|\Theta_i)$ can be derived (using Bayes' law) as:

$$\log F_s(\mathbf{S}', \mathbf{g}|\mathbf{R}', \Theta_i) = \sum_{m=1}^{|D'|} \log \frac{\beta_{g_m} F_N(s_m, r_m|\Psi_{g_m})}{F_{GMM}(r_m|\Theta_i)}$$

In this expression, Ψ_{g_m} refers to the parameters associated with the $g_m{}^{th}$ Gaussian or normal in the mixture model parameterized by Θ_i, and β_{g_m} gives the probability of selecting the $g_m{}^{th}$ normal in the mixture.

Via algebraic manipulations, $Q(\Theta_i, \Theta_i')$ can be written as:

$$Q(\Theta_i, \Theta_i') = \sum_{m=1}^{|D'|} \sum_{j=1}^{c_i} [\log \beta_j + \log F_N(s_m, r_m|\Psi_j) - \log F_{GMM}(r_m|\Theta_i)] Pr[g_m = j|s_m, r_m, \Theta_i']$$

In this expression, $Pr[g_m = j|s_m, r_m, \Theta_i']$ is the probability that the j^{th} normal was used to produce the m^{th} data point, and is:

$$Pr[g_m = j|s_m, r_m, \Theta_i'] = \frac{\beta_g F_N(s_m, r_m|\Psi_j)}{F_{GMM}(s_m, r_m|\Theta_i')}$$

Unfortunately, maximizing this Q-function is still quite difficult, due to the term $\log F_{GMM}(r_m|\Theta_i)$. Since F_{GMM} is itself a summation, we are attempting to perform a maximization over the logarithm of a summation of terms, which is hard. Thus, as an approximation, we drop this term, the justification being that as a one-dimensional PDF, its logarithm will generally be much less significant than the two-dimensional F_N that also appears. The final update rules for α_j and Ψ_j (the parameter set for j^{th} normal in the i^{th} query class) are:

$$\beta_j \leftarrow \frac{\sum_{m=1}^{M} \sum_{j=1}^{N_m} Pr[q_m=i|.] Pr[g_m=j|.]}{\sum_{m=1}^{M} N_m Pr[q_m=i|.]}$$

$$\mu_{j,r} \leftarrow \frac{\sum_{m=1}^{M} \sum_{j=1}^{N_m} Pr[q_m=i|.] Pr[g_m=j|.] r_{m,j}}{\sum_{m=1}^{M} N_m Pr[q_m=i|.]}$$

$$\sigma_{j,r}^2 \leftarrow \frac{\sum_{m=1}^{M} \sum_{j=1}^{N_m} Pr[q_m=i|.] Pr[g_m=j|.] (r_{m,j}-\mu_{j,r})^2}{\sum_{m=1}^{M} N_m Pr[q_m=i|.]}$$

$$Cov_j \leftarrow \frac{\sum_{m=1}^{M} \sum_{j=1}^{N_m} Pr[q_m=i|.] Pr[g_m=j|.] (r_{m,j}-\mu_{j,r})(s_{m,j}-\mu_{j,s})}{\sum_{m=1}^{M} N_m Pr[q_m=i|.]}$$

$Pr[g_m = j|.]$ denotes $Pr[g_m = j|s_{m,j}, r_{m,j}, \Theta_i']$ and $Pr[q_m = i|.]$ denotes $Pr[q_m = i|\mathbf{s}_m, \mathbf{r}_m, \Theta']$. $\mu_{j,r}$ is the mean rank for the j^{th} normal in the GMM for the i^{th} query class. $\sigma_{j,r}^2$ is the variance of this normal, and Cov_j is the covariance between the rank and the score in this normal. While no formula is given above for updating $\mu_{j,s}$ and $\sigma_{j,s}^2$, these can be obtained by simply replacing s for r in the above rules.

D. The Complete Learner

Given all of this, the complete learning algorithm is as follows:

```
start with an initial guess for Θ;
Repeat
  Θ' = Θ;
  for each query class i in 1 ··· c:
    update α_i; (Equation (1) in Section 4.2)
    Repeat
      Θ_i' = Θ_i
      for each Gaussian j in 1 ··· g:
        update β_j, Ψ_j; (Section 4.3)
      end for
    Until (change in Θ_i is small)
  end for
Until (change in Θ is small)
```

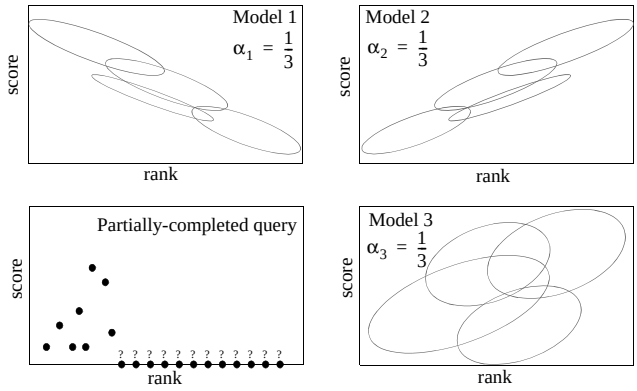


Fig. 5. Updating the component weights in response to a partially-completed query. In this case, the prior weights are all even at $\frac{1}{3}$. However, after observing a few points that were highly ranked according to the surrogate ranking function R', it is possible to mathematically "rule out" the possibility that this particular query was produced by the first GMM; the relationship between score and rank is all wrong. The third model looks most plausible, though it may not be possible to rule out the second with so little data. Therefore, the updated set of weights might be $\langle 0, \frac{1}{10}, \frac{9}{10} \rangle$.

This algorithm contains nested loops. The innermost loop updates the weights of the Gaussians for each of the query classes. The outermost loop updates the weights of query classes once the best weights for the Gaussians are selected.

V. Updating the Model

The previous section discussed how to learn the model parameter set Θ in order to fit the model to a historical query workload. The learned model is generic, in the sense that it covers the entire historical query workload. This generic model can be used to correctly bound the accuracy of the surrogate ranking function for new queries. To do this, one needs to compute the probability that for an arbitrary query generated via the learned model, the surrogate ranking function will return at least h of the true top k items in the database after processing k' items ($k' \geq k$). This probability could correctly be returned regardless of the particular query in question.

Although we can compute a bound for a query, using the existing model blindly is problematic. This is because some of the query classes can describe the relationship between R' and R well for one query and other classes can do it well for another query. We do not know in advance which query model is the best choice. In order to have a much better idea of how accurate the surrogate ranking will be, we tailor the model to the current query, rather than reporting a probability that will hold for an arbitrary query, about which no information is known.

Intuition. It is quite easy to update the query class weights α_1, $\alpha_2, \cdots, \alpha_c$ in response to a particular query. Intuitively, what one has access to at model update time are a number of learned GMMs, one for each query type, as well as a set of score, rank pairs that have been processed thus far for the current query. Those GMMs that were more likely to have produced the current query will be given a higher updated weight, as depicted in

Figure 5. In this Figure, we can see that the third model likely explains this query best and the first model explains the worst. Thus, it makes sense to have a high weight for the third model and low weight for the first model.

Update Equation. Let α_i^{new} denote the updated weight for the i^{th} query class. Assume that the vector $\mathbf{s}$ denotes the partially-observed scores. Let $\mathbf{r}/\mathbf{s}$ denote only those ranks for which a score observation has been made — obviously, for a database of size N, $\mathbf{r}$ will always be of size N once the surrogate ranking is complete; however, for the moment we are only interested in those ranks for which we actually have score values. Finally, let q^* be the (unknown) query class (GMM) that was used to produce this particular query. Then,

$$\alpha_i^{new} = Pr[q^* = i|\mathbf{r}, \mathbf{s}, \Theta] = \frac{\alpha_i f_{GMM}(\mathbf{s}, \mathbf{r}/\mathbf{s}|\Theta_i)}{\sum_{j=1}^{\mathbf{c}} \alpha_j F_{GMM}(\mathbf{s}, \mathbf{r}/\mathbf{s}|\Theta_j)}$$

VI. Obtaining the Bounds

The last significant technical hurdle to surmount is being able to use the model and the updated weights to compute the probability that the number of actual top k objects returned to the user exceeds some user-specified value h.

A. Applying the Updated Weights

Our basic algorithm (described in Section 1) computes $R(q, d)$ exactly for those database objects with the k' best rankings according to R'. So if the database object d is in the top k' according to R', and d is also in the top k according to R, then d will correctly be returned to the user. Let us denote the random variable which controls the number of true results observed (i.e., the objects that are in the top k according to R and are also in the top k' according to R') with H. Our goal is then to compute the quantity $Pr[H \geq h|\mathbf{r}, \mathbf{s}, \Theta]$.

To compute the above probability, we start by doing algebraic manipulations on it. We know that:

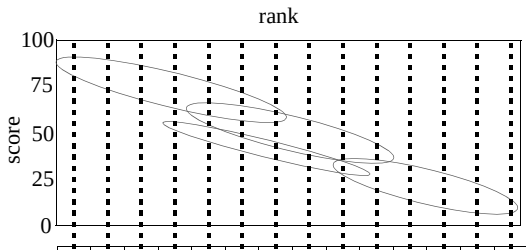
$$\begin{aligned} Pr[H \geq h|\mathbf{r},&\mathbf{s}, \Theta] \\ &= \sum_{i=1}^{\mathbf{c}} Pr[\mathbf{q} = i|\mathbf{r}, \mathbf{s}, \Theta] Pr[H \geq h|\mathbf{q} = i, \mathbf{r}, \mathbf{s}, \Theta] \\ &= \sum_{i=1}^{\mathbf{c}} \alpha_i^{new} Pr[H \geq h|\mathbf{q} = i, \mathbf{r}, \mathbf{s}, \Theta] \end{aligned}$$

where the computation of α_i^{new} is as described previously.

Unfortunately, analytically computing the term $Pr[H \geq h|\mathbf{q} = i, \mathbf{r}, \mathbf{s}, \Theta]$ is difficult. Let $\bar{\mathbf{s}}$ denote the set of scores that are currently unknown (that is, scores that currently have rankings in $\mathbf{r}$, but no associated value in $\mathbf{s}$). We have to compute the probability that the number of values in $\mathbf{s}$ that are in the top k in the combined set $(\mathbf{s} \cup \bar{\mathbf{s}})$ exceeds c. Computing this probability exactly likely requires exponential time complexity in the size of $\bar{\mathbf{s}}$, because it seems as if we would have to explicitly consider the likelihood of all possible combinations of values for the items in $\bar{\mathbf{s}}$. Thus, we resort to Monte Carlo methods.

B. Naive Monte Carlo

We begin by describing a relatively simple Monte Carlo algorithm to obtain an approximate value for $Pr[H \geq h|\mathbf{q} = i, \mathbf{r}, \mathbf{s}, \Theta]$. Note that we repeat this algorithm for each query class i. Our Monte Carlo algorithm works by using the i^{th} GMM to generate a vector of possible scores, with one score associated with each rank in $\mathbf{r}$. This set of possible scores constitutes a hypothetical "database instance". We create many database instances by repeatedly generating vectors of possible scores. This is illustrated above in Figure 6. Notice that the entries for each database instance are sorted according to their surrogate ranks, not the actual similarity scores.



MC instance #1	88	83	79	70	50	55	48	45	33	47	22	18	15	16
MC instance #2	87	82	75	68	55	47	38	40	36	25	25	25	19	9
MC instance #3	92	79	75	72	76	72	45	37	31	35	21	19	13	5
MC instance #4	90	86	40	71	42	70	42	41	32	39	32	17	22	15
MC instance #5	88	80	81	67	66	59	48	43	29	22	19	18	16	13

Fig. 6. Generating five hypothetical database instances. For every rank $r \in \mathbf{r}$, five possible scores are generated by sampling from the GMM corresponding to the current query class, conditioned upon r. In other words, for a given r, the GMM is projected down onto a the score axis, and the resulting one-dimensional distribution is sampled from five times.

														$H \geq 5$?
MC instance #1	***88***	***86***	***83***	41	44	42	***48***	45	33	47	22	18	15	16	No
MC instance #2	***88***	***86***	***83***	41	***44***	42	38	40	36	25	25	25	19	9	Yes
MC instance #3	***88***	***86***	***83***	41	44	42	***45***	37	31	35	21	19	13	5	No
MC instance #4	***88***	***86***	***83***	41	***44***	42	42	41	32	39	32	17	22	15	Yes
MC instance #5	***88***	***86***	***83***	41	44	42	***48***	43	29	22	19	18	16	13	No

$Pr[H \geq 5] \approx 0.4$

Fig. 7. Estimating $Pr[H \geq 4]$. In this example, the set of six (score, rank) pairs for which an actual score has been computed are "superimposed" over each of the five hypothetical database instances. Specifically, $(88, 1)$, $(86, 2)$, $(83, 3)$, $(41, 4)$, $(44, 5)$, $(42, 6)$ are all superimposed over ranks one through six. Then, each database instance is checked to see whether the number of actual or non-simulated scores in the top k (with $k = 4$) database scores is at least four. The score values in the top k are shown in bold-italic. In this case, the number of non-simulated scores in the top k is at least four in two of the five hypothetical database instances. Thus, $\frac{2}{5} = 0.4$ is returned as as estimate for $Pr[H \geq 4]$.

Once a large number of database instances have been created, then the actual score vector is "superimposed" over each database instance, by adding $s_i \in \mathbf{s}$ to the r_i^{th} position in each database instance. We then check, for each database instance, whether H is at least as large as the cutoff value h. If it is, then the database instance "passes the test." By counting the fraction database instances that pass the test, we obtain an approximation for $Pr[H \geq h|\mathbf{q} = i, \mathbf{r}, \mathbf{s}, \Theta]$. It is possible to bound the approximation accuracy using standard techniques. This basic process is illustrated in Figure 7.

One issue that requires a bit of clarification is how to conditionally generate a score, given a specific rank. The process of sampling from $F_s(s|r, \Theta_i)$ can be viewed as generating a very large number of (s, r) pairs from the two-dimensional GMM

parameterized by Θ_i, and then accepting the first (s, r) pair where the rank is precisely the desired rank. Let G be the identity of the component that produces the acceptable pair. Then the probability that $G = j$ is:

$$Pr[G = j|r, \Theta_i] = \frac{Pr[G = j|\Theta_i]F_N(r|\Psi_j)}{F_{GMM}(r|\Theta_i)}$$

This means that we can select an appropriate Gaussian to generate our s value by rolling a biased die, where the probability of obtaining a j is $Pr[G = j|r, \Theta_i]$. If we happen to select the j^{th} Gaussian to generate our score, we then generate our actual score by sampling from the PDF $F_N(s|r, \Psi_j)$. It is a widely-known fact that when a Gaussian or normal distribution is conditioned on one or more of its dimensions, the resulting distribution is still a normal distribution in a lower-dimensional space, with (possibly) updated parameter values. In our case, $F_N(s|r)$ is a one-dimensional normal distribution with mean

$$\mu = \mu_{j,s} + \frac{Cov_j}{\sigma^2_{j,r}}(r - \mu_{j,r})$$

and variance

$$\sigma^2 = \sigma^2_{j,s} - \frac{(Cov_j)^2}{\sigma^2_{j,r}}$$

In these expressions, $\mu_{j,s}$, $\mu_{j,r}$, $\sigma^2_{j,r}$, and Cov_j are all parameters within Ψ_j.

C. A Faster Monte Carlo

Although the naive Monte Carlo method computes the bounds correctly, its running time makes it impractical for online querying. If one thousand database instances for each query class are generated, and each database instance has 10^5 items, and there are ten query classes, then one billion scores must be generated. Generating these scores to compute bound every time we need to compute a bound will take significant amount of time.

The obvious way around this problem is to generate the various database instances, and store them in memory once. Then, we can compute the bounds using this pre-generated data for all the queries. After all, the 8GM of RAM needed to store one billion scores only costs a few dollars in 2008 prices. This strategy avoids generating Monte Carlo instances online. However, counting the number of synthetic scores in each instance that make their way into the top k (as illustrated in Figure 7) must be done carefully.

We develop an efficient algorithm that computes the bound using a small subset of the entries in the pre-computed instances. Figure 8 illustrates our algorithm. We simply keep the entries of each database instance sorted according to the actual score R instead of the surrogate rank R'. For a given query, assume that we have computed the score R for the k' database records with highest surrogate rank. Assume that the the h^{th} best score among these observed results is s. We compare s to the entries of the Monte Carlo instances in descending order of score starting from the first one until one of the two stopping conditions is met.

(1) The top example in Figure 8 describes the first stopping criteria. The score in the Monte Carlo instance becomes less than or equal to s. This means that there are no other entries in the Monte Carlo instance that has better score than the observed h^{th} best score. If the we check the number of entries examined from this instance that are not ranked in top k' according to surrogate ranking. Such entries *"reject"* the h^{th} best observation. If this number is at most $k - h$, the Monte Carlo Instance supports that the observed results contain at least h of the top k results. Otherwise, it does not.

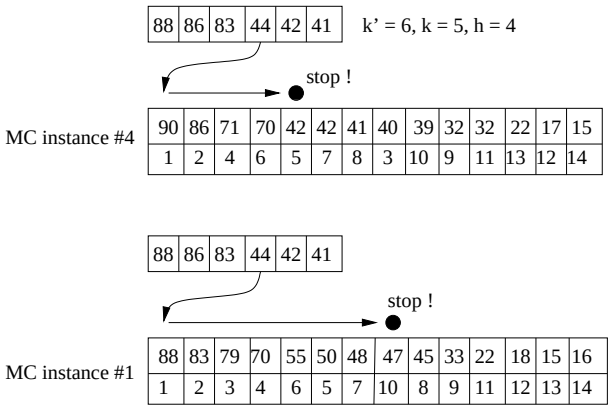


Fig. 8. Fast counting of the number of items on two Monte Carlo database instances. In this example, 44 is the fourth (h^{th}) best actual score of the six (k') obtained thus far. In the instance on the top of the figure, only the top four scores in the Monte Carlo instance are better than 44. None of them have a rank greater than k'. Therefore, this instance results in a "yes". In the instance at the bottom of the figure, we stop after seeing score 47. This is because now we have seen two entries whose score exceed 44 and rank exceed 6. This number greater than the difference $k - h = 5 - 4 = 1$. Therefore, this instance results in a "no".

(2) The bottom example in Figure 8 describes the second stopping criteria. It follows the observation made in the first one. If the total number of examined entries that reject the h^{th} best observation exceeds $k - h$, the Monte Carlo Instance does not support that the observed results contain at least h of the top k results. Thus, there is no need to examine the rest of the entries of the Monte Carlo instance.

VII. Applications and Evaluation

In this section, we apply our model to three application domains: protein structure alignment, image matching, and biochemical pathway alignment. The two questions that we wish to evaluate are: First, do reasonable surrogate ranking functions exist for these example application domains? Second—and much more important for this particular paper—can our model correctly predict when the surrogate ranking will work well, and when it will not?

A. Protein Structure Search

Problem Domain. The first problem we tackle is searching a database of protein structures for the k proteins that most closely align with a query protein, according to the CE structure matching algorithm [22]. We use the Z-score computed by CE as the similarity measure. As described in the introduction, we used Blast's sequence alignment score as the surrogate for the Z-score.

Experimental Setup. To apply our model to this problem domain, we created a database of 10,000 protein structures, sampled at random from PDB (http://www.rcsb.org). We then selected at random 101 proteins as training proteins, and 100 proteins as query proteins from the remaining ones in PDB. Thus, the database, training, test sets do not have any common proteins. We learn a model having ten different query classes, each a ten-component GMM.

We then queried each of the query proteins in the test database. For each query, we compute the exact CE score for the query's

TABLE II

AVERAGE $Pr[H \geq h]$ VALUES FOR VARIOUS k', h COMBINATIONS, WITH $k = 10$, FOR THE PROTEIN DATA SET. SECTION VII-A PROVIDES A DETAILED EXPLANATION OF HOW TO INTERPRET THIS TABLE, UNDER THE HEADING "RESULTS AND DISCUSSION". "—" DENOTES THAT NO QUERIES FELL IN THAT PARTICULAR CELL. THE ROW "OBS." SHOWS THE TOTAL NUMBER OF QUERIES THAT FALL INTO CATEGORY "Y" OR "N" (I.E., SURROGATE RANKING IS SUCCESSFUL OR FAILS).

	$h = 1$		$h = 5$		$h = 8$		$h = 10$	
k'	"Y"	"N"	"Y"	"N"	"Y"	"N"	"Y"	"N"
50	0.70	0.10	0.53	0.00	0.24	0.00	—	0.00
100	0.78	0.12	0.52	0.00	0.44	0.00	—	0.00
500	0.78	0.00	0.44	0.04	0.56	0.01	—	0.01
obs	248	52	45	255	24	276	0	300

top k' matches according to BLAST, and return the top k CE scores from that set. Among those k proteins, we then count the number of proteins that are actually among the top k proteins according to the Z-score of CE in the entire database. We also use our model to predict the probability that this value exceeds a constant h for all $h \in \{1, 2, \cdots, k\}$ using the model of this paper.

We ran two sets of experiments for the protein structure matching application. In the first set, we use $k = 10$ and test all combinations of $k' \in \{50, 100, 500\}$ and $h \in \{1, 5, 8, 10\}$. In the second set, we use $k = 1$, and so $h = 1$. We test $k' \in \{50, 100, 500\}$ in this experiment as well.

Results and Discussion. The overall results are summarized in Tables II, III, and Figure 9.

The Figures require a bit of explanation. In all of these figures, queries are divided into two classes: "Y" or "Yes" queries, and "N" or "No" queries. "Y" queries are those for which the surrogate ranking was successful. That is, for a particular h value, a query is a "Y" query if at least h of the top k queries were in the k' queries returned by the surrogate ranking. Thus, if the surrogate ranking were perfect in all cases, there would be only "Y" queries and no "N" queries. The reason that we partition all queries into "Y" or "N" queries is that our model returns a Bayesian belief or probability that a given query is a "Y" query. Thus, for "Y" queries we hope that the probability returned by our model is close to one. For "N" queries, we hope for a zero.

TABLE III

AVERAGE PROBABILITY VALUES FOR PROTEIN DATA SET, WITH $k = 1$. SEE SECTION VII-A FOR DETAILS. THE ROW "OBS." SHOWS THE TOTAL NUMBER OF QUERIES THAT FALL INTO CATEGORY "Y" OR "N" (I.E., SURROGATE RANKING IS SUCCESSFUL OR FAILS).

	$h = 1$	
k'	'Y"	"N"
50	0.32	0.09
100	0.38	0.25
500	0.47	0.32
obs.	187	133

In each of these Figures, the number that we are most interested in is the probability value assigned by our model to each query. Tables II and III show average probabilities over different groupings of queries. Thus, in Table II, the 0.70 at the top left of the table means that for those "Y" queries with $h = 1$, $k' = 50$, our model (on average) said that there was a 70% chance that the query would fall in the "Y" class. The bottom row in both of these Figures counts the total number of queries that fell in that column. For example, 248 queries were "Y" queries for $h = 1$.

Table III shows the same data in more detail for specific h, k, k' combinations. In this figure, each *individual* probability assigned by our model is plotted. The "Yes" portion of the figure shows each of the different probabilities assigned for all of the queries where the surrogate ranking was effective. If the model were perfect, all of these values would be one. The "No" portion shows the probabilities assigned when the surrogate ranking did not return at least h of the top k. If the model were perfect, all values would be zero.

There are several interesting results. First, BLAST *is* a reasonable surrogate for CE's structure match. Consider Table III. Out of 300 queries total (100 each for k in 50, 100, 500), more than half of the time, BLAST was able to find the single best match out of the 10,000 proteins in the database. That being said, BLAST is not perfect. For example, it was never able to find ten out of the top ten CE answers in any of the experiments where $h = k = 10$.

Significantly, our model was able to do a very good job in bounding the accuracy of BLAST as a surrogate ranking function; in fact, our model even tended to be a bit too pessimistic, which we observed systematically throughout our experiments. If there must be any bias, then a bit of bias towards the pessimistic side is preferable, since it does not give the end-user a false sense of security. Consider the case of $k' = 500$. Except for $h = 5$ where the average probability returned was 0.04—so it always got the "N" cases right. Over all "Y" cases, the model returned an average probability of 0.74.

The plots in Figure 9 are particularly interesting. Here we plot the probabilities for particular sets of "Y" and "N" cases side-by-side. It is clear that our model assigns much higher probabilities of success to queries where the surrogate ranking produces good results, and so in general, our model is able to successfully distinguish among the queries where BLAST will be successful, and when it will not. It is also interesting that more data generally seems to help boost the accuracy, which is definitely a desirable characteristic of any machine learning method. Consider the two plots furthest to the right. They test exactly the same scenario, except that k' is much lower in the right plot. As one might expect, this relative lack of data hurts our model's ability to correctly assign high probabilities to "Y" queries.

B. Image Search

Problem Domain. The second problem that we consider is image similarity search using Earth Mover's Distance (EMD). EMD is in fact a metric and so it is possible to index, but it still serves as a good testbed for our models for a couple of reasons. First, in our experience generic, metric-type indexes do not often work well with EMD. Second, computing the EMD between two images can be arbitrarily expensive, and so sequential scans are not a good option. As a surrogate for EMD, we use Euclidean distance.

Experimental Setup. The database in question consists of 3500 retinal images obtained from UCSB Bioimage database, with 105 training queries and 105 test queries. Just as before, we use $k' \in \{50, 100, 500\}$ and $h \in \{1, 5, 8, 10\}$. For brevity we only report the results for $k = 10$. All of the model learning parameters in this experiment were the same as that in the protein matching experiment.

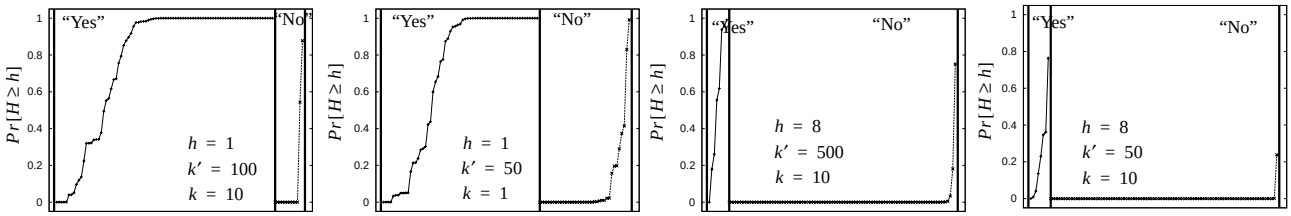


Fig. 9. Detailed results for four particular settings, protein data set. A discussion of how to interpret this plot is provided in Section VII-A.

TABLE IV

AVERAGE $Pr[H \geq h]$ VALUES FOR VARIOUS k', h COMBINATIONS, WITH $k = 10$, FOR THE IMAGE DATA SET USING THE EUCLIDEAN SURROGATE RANKING. "—" DENOTES THAT NO QUERIES FELL IN THAT PARTICULAR CELL.

	$h = 1$		$h = 5$		$h = 8$		$h = 10$	
k'	"Y"	"N"	"Y"	"N"	"Y"	"N"	"Y"	"N"
50	1.00	—	0.98	0.95	0.59	0.43	0.13	0.05
100	1.00	—	0.98	—	0.88	0.90	0.55	0.23
500	1.00	—	1.00	—	0.99	—	0.86	0.59
obs	315	0	314	1	302	13	258	57

Results and Discussion. The detailed results for the original Euclidian ranking are given in Table IV. It is immediately clear is that the surrogate ranking performs exceedingly well in this domain. For example, the Euclidean surrogate returns at least eight out of the top ten in 302 of 315 of the total cases. It is even very accurate in the case of $k' = 50$, when for 95 out of 105 cases, at least eight out of the top ten images were returned.

Significantly, our model was able to capture the near perfect relationship between EMD and Euclidean distance. For both $h = 1$ and $h = 5$, our model correctly predicted that in nearly every case, the Euclidean surrogate would return at least h of the top ten.

The more interesting cases to examine are all of the $h = 10$ cases, and the case of $h = 8$, $k' = 50$, which has a h value close to ten coupled with a small sample size. In such cases, the goal is to predict whether or not there will be a very slight error: one or two missed results. It is in these cases where our model tended to be less precise, and gave much less sure predictions. Contrast the leftmost plot of Figure 10 where our model almost always assigned one probabilities to every "Y" case to the other three plots in the figure, where the model assigned a variety of probabilities to both the "N" and "Y" cases, indicating a level of uncertainty. The reason for this is that while there is a very strong relationship between Euclidean distance and EMD, the relationship is not perfect. It is not unusual to encounter one or two errors out of ten results, which would correspond to a "N" for $h = 8$ or $h = 10$. However, it is exceedingly difficult to predict exactly when these errors will happen, because unlike for the protein structure problem, the surrogate uniformly performs quite well. In the protein domain, it is often clear after just a few proteins have been examined that the query is a bad (or good) one, and then our model returns an appropriately small (or large) probability. But there is not much of a clue that there will or will not be one or two errors out of ten. This leads to probabilities that are scattered throughout the range of zero to one, for both the "N" and "Y" cases.

TABLE V

AVERAGE $Pr[H \geq h]$ VALUES FOR VARIOUS k', h COMBINATIONS, WITH $k = 10$, FOR THE METABOLIC PATHWAY DATA SET.

	$h = 1$		$h = 5$		$h = 8$		$h = 10$	
k'	"Y"	"N"	"Y"	"N"	"Y"	"N"	"Y"	"N"
50	0.96	0.00	0.95	0.00	0.90	0.00	0.62	0.00
100	0.98	0.00	0.97	0.00	0.97	0.00	0.95	0.00
500	1.00	0.08	1.00	0.00	1.00	0.00	0.99	0.00
obs	198	198	198	198	198	198	198	198

C. Pathway Search

Problem Domain and Experimental Setup. In this particular domain, the query is a metabolic pathway for a particular organism, and the goal is to find the few pathways in the database that most closely match the query. The actual ranking is obtained using the sum of the topological and homological similarities of the two pathways (We used the alignment software and the scoring measure developed in [3] for this purpose). We compute a surrogate ranking function as the ratio of the number of common enzymes and compounds in two pathways to the total number of enzymes and compounds in the union of the two pathways. In this experiment, the database contains 6000 metabolic pathways from the KEGG database [17] from randomly selected organisms. In addition to these pathways, we use 128 training queries and 198 test queries.

Unfortunately for testing the utility of our model, this particular surrogate is even better than Euclidean distance for EMD. It is uniformly perfect, which results in trivial plots and tables (results omitted). This is because usually if two organisms share the same pathway, they often have almost the same set of enzymes and compounds. Furthermore, the biological features of these enzymes and compounds dictates the same topology on these pathways. It is worth saying that even for these kinds of databases where the surrogate function and the expensive score produce almost the same ranking, our method still worked correctly, and correctly predicted that the top results will be found very quickly.

To make the modeling problem more challenging, we do the following. We randomly select 10% of the training queries, and reverse all of the ranks for each of those queries, so the top-ranked pathway for a query according to the surrogate function actually becomes the lowest-ranked pathway. These means that for 10% of the training queries, the surrogate will perform very poorly. Then, we do the same swapping for 50% of the test queries, in order to test the case where the fraction of queries that the surrogate performs well on has evolved over time.

Results and Discussion. The overall results are summarized in Table V; some particular results are shown in detail in Figure 11.

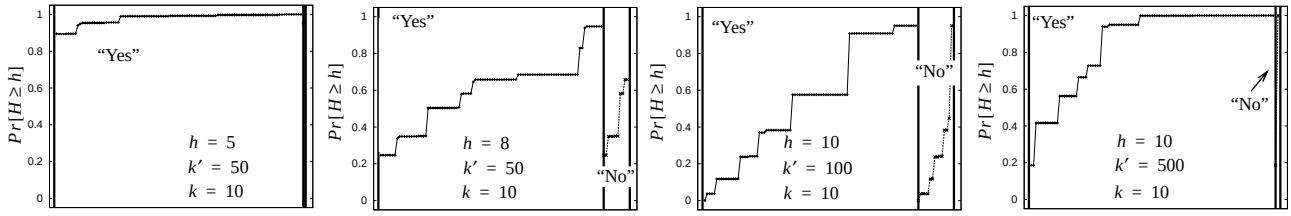


Fig. 10. Detailed, selected results, for the image data set.

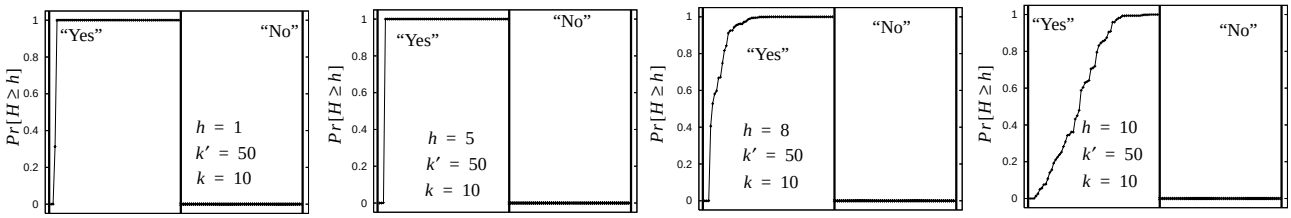


Fig. 11. Detailed, selected results, for the pathway data set.

By design, the surrogate ranking function works well exactly one half of the time, but our model is found to function almost perfectly; for $k' = 100$ and $k' = 500$, nearly all "Y" cases are assigned one probabilities, and nearly all "N" cases are assigned zero. We note that this excellent accuracy was obtained after the prevalence of the "reversed" queries increased five-fold from training to testing.

VIII. Discussion and Results

In this paper, we have proposed a general framework for handling similarity queries with very expensive scoring functions: use an inexpensive, surrogate ranking function, and then apply a statistical model to accurately predict the quality of a surrogate ranking function on a per-query basis. The goal of the paper was not to suggest surrogate ranking functions. Rather, our primary technical goal is to predict the quality of the ranking function. Judged along those lines, our results are very good. Consider the case of protein structure search. This is perhaps the most interesting application, because it is the application for which the surrogate function was most inaccurate. Even in this case, our models were able to accurately distinguish between cases where the surrogate works, and cases where it does not. Examine the case where $k = 10$ in Table 2. In this case, when 5 out of the top 10 matches for the query are actually in the result set, our model gives an average result of around 0.5—meaning that (on average) our model says that there is a 50% chance that 5 of the top 10 matches are present when they actually are there. On the other hand, in the case where less than 5 matches are present, our model gives an average result of 0.01—meaning that with almost total accuracy it successfully recognizes that the surrogate ranking is not working.

References

[1] S. Altschul, W. Gish, W. Miller, E. W. Meyers, and D. J. Lipman. Basic Local Alignment Search Tool. *Journal of Molecular Biology*, 215(3):403–410, 1990.

[2] S. Arya and D. M. Mount. Approximate nearest neighbor queries in fixed dimensions. In *SODA*, pages 271–280, 1993.

[3] F. Ay, T. Kahveci, and V. de Crecy-Lagard. Consistent alignment of metabolic pathways without any abstaction in modeling. In *Computational Systems Bioinformatics Conference (CSB)*, 2008.

[4] S. Berchtold, B. Ertl, D. Keim, H.-P. Kriegel, and T. Seidl. Fast Nearest Neighbor Search in High-dimensional Space. In *ICDE*, pages 209–218, 1998.

[5] R. Brenk, J. J. Irwin, and B. K. Shoichet. Here Be Dragons: Docking and Screening in an Uncharted Region of Chemical Space. *J Biomol Screen*, 10(7):667–674, 2005.

[6] S. R. Eliason. *Maximum Likelihood Estimation: Logic and Practice* . Sage Publications, 1993.

[7] R. F. S. Filho, A. J. M. Traina, J. Caetano Traina, and C. Faloutsos. Similarity Search without Tears: The OMNI Family of All-purpose Access Methods. In *ICDE*, pages 623–630, 2001.

[8] D. Gamerman and H. F. Lopes. *Markov Chain Monte Carlo: Stochastic Simulation for Bayesian Inference*. Chapman & Hall/CRC, 2006.

[9] H. Garcia-Molina, J. D. Ullman, and J. D. Widom. *Database Systems: The Complete Book*. Prentice Hall, 2002.

[10] T. A. Halgren, R. B. Murphy, R. A. Friesner, H. S. Beard, L. L. Frye, W. T. Pollard, and J. L. Banks. Glide: A new approach for rapid, accurate docking and scoring. 2. enrichment factors in database screening. *Journal of Medicinal Chemistry*, 47(7):1750–1759, 2004.

[11] H. Hegyi and M. Gerstein. The Relationship Between Protein Structure and Function: a Comprehensive Survey with Application to the Yeast Genome. *Journal of Molecular Biology*, 288(1):147–164, 1999.

[12] W. L. Jorgensen. The Many Roles of Computation in Drug Discovery. *Science*, 303(5665):1813–1818, 2004.

[13] J. Kim and S. D. Copley. Why metabolic enzymes are essential or nonessential for growth of escherichia coli k12 on glucose. *Biochemistry*, 46(44):12501–12511, 2007.

[14] G. McLachlan and D. Peel. *Finite Mixture Models*. Wiley-Interscience, 2000.

[15] G. J. McLachlan and T. Krishnan. *The EM Algorithm and Extensions*. Wiley-Interscience, 1996.

[16] D. W. Mount. *Bioinformatics: Sequence and Genome Analysis*. Cold Spring Harbor Laboratory, 2004.

[17] H. Ogata, S. Goto, K. Sato, W. Fujibuchi, H. Bono, and M. Kanehisa. KEGG: Kyoto Encyclopedia of Genes and Genomes. *Nucleic Acids Research*, 27(1):29–34, 1999.

[18] A. O'Hagan and J. J. Forster. *Bayesian Inference*. Volume 2B of *Kendall's Advanced Theory of Statistics*. Arnold, second edition, 2004.

[19] R. Y. Pinter, O. Rokhlenko, E. Yeger-Lotem, and M. Ziv-Ukelson. Alignment of metabolic pathways. *Bioinformatics*, 21(16):3401–8, 2005.

[20] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest Neighbor Queries. In *SIGMOD*, San Jose, CA, 1995.

[21] T. Seidl and H. Kriegel. Optimal Multi-Step k-Nearest Neighbor Search. In *SIGMOD*, 1998.

[22] I. Shindyalov and P. Bourne. Protein structure alignment by incremental combinatorial extension (CE) of the optimal path. *Protein Engineering*, 11(9):739–747, 1998.

[23] Q. H. Vu, B. C. Ooi, D. Papadias, and A. K. H. Tung. A graph method for keyword-based selection of the top-k databases. In *SIGMOD*, pages 915–926, 2008.

[24] P. Yianilos. Data Structures and Algorithms for Nearest Neighbor Search in General Metric Spaces. In *SODA*, pages 311–321, 1993.