Learned Cardinality Estimation for Similarity Queries

Ji Sun
Tsinghua University
Beijing, China
sun-j16@mails.tsinghua.edu.cn

Guoliang Li
Tsinghua University
Beijing, China
liguoliang@tsinghua.edu.cn

Nan Tang
QCRI, HBKU
Doha, Qatar
ntang@hbku.edu.qa

ABSTRACT

In this paper, we study the problem of using deep neural networks (DNNs) for estimating the cardinality of similarity queries. Intuitively, DNNs can capture the distribution of data points, and learn to predict the number of data points that are similar to one data point (a *similarity search*) or a set of data points (a *similarity join*). However, DNNs are data hungry; directly training a DNN often results in poor performance. We propose two strategies to improve the accuracy and reduce the size of training data: *query segmentation* and *data segmentation*. Query segmentation divides a query into query segments, trains a neural network for each query segment, and combines their outputs with subsequent DNNs to get the query embedding. Data segmentation groups similar data into data segments, train a *local-model* for each data segment, and learn a *global-model* to decide which local-models should be used for a given query. The estimates from selected local-models will be summed up as the final estimate. We also extend our model to support similarity joins, which trains a DNN to directly estimate the cumulative sum of objects that are similar to a set of queries. Experiments show that our methods can efficiently (i.e., with small training data) learn to estimate the cardinality of similarity searches/joins, and yield effective estimates (i.e., close to real cardinalities).

ACM Reference Format:
Ji Sun, Guoliang Li, and Nan Tang. 2021. Learned Cardinality Estimation for Similarity Queries. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 18–27, 2021, Virtual Event , China.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3448016.3452790

1 INTRODUCTION

Similarity search, which aims at finding objects that are *similar* to given object(s), is a fundamental problem in computer science and is crucial to many applications, such as text search, image search, product recommendation, database optimizations [26, 29, 30, 46], network traffic [12, 47], and so forth.

Similarity-aware Cardinality Estimation. Let D be a collection of data objects, e.g., images, text, and tuples. We study two problems.

Similarity search is to provide an estimation $\widehat{\text{card}(q, \tau, D)}$ for the number of objects in D whose distances to a query q are not greater than a distance threshold τ.

Similarity join takes a set Q of search objects as input, and provides an estimation $\widehat{\text{card}(Q, \tau, D)}$ for the total number of pairs (q, p) whose distance between $q \in Q$ and $p \in D$ is not greater than τ.

Learned Cardinality Estimation. Essentially, cardinality estimation for similarity queries is a regression problem: Given a similarity search (a query object q and a threshold τ over a dataset D), the problem is to estimate the cardinality of this input.

A straightforward solution is to train a DNN to learn a function $F(x_q, x_\tau, x_D)$, as shown in Figure 1(A), where x_q is the feature vector of query q, x_τ is a one-dimensional vector of threshold τ, and x_D is a k-dimensional vector that each dimension is the similarity between q and one sample in D (k samples in total).

Recently, DL-based method has been studied to estimate the cardinality for similarity search [53]. They utilize VAE (Variational Autoencoders) [27] to learn the cardinality of similarity search, and learn embeddings for different thresholds separately for enhancing accuracy and guaranteeing monotonicity. However, they produce relative large errors with few shots of training samples (e.g., hundreds or thousands), with two main reasons: (1) they learn query feature embeddings by fully connected neural networks, which is hard to capture the data distribution of high-dimensional dataset, and (2) they cannot fully utilize the points clustering information (which can be obtained by unsupervised clustering) in the model.

Challenges. DL is building a system by assembling parameterized modules to perform a task by optimizing the parameters using a gradient-based method. As will be shown later by empirical results in Section 6, the simple approach in Figure 1(A) gives poor estimates, mainly because using one *big module* is hard to well capture the distribution of distances between data and an arbitrary query. Naturally, the main challenges for estimating the cardinality for similarity queries are to design *small modules*, where each small module captures a part of knowledge for the given task, and by assembling these small modules it can better serve the task with higher accuracy and less training data.

Our Methodology. We design small modules from two aspects, *smaller* queries and *smaller* data.

[Query segmentation.] This is to divide a query q's feature vector x_q into smaller query segments $\{x_q^{(1)}, \ldots, x_q^{(m)}\}$ (Figure 1(B)), and train a module E_1 to produce an embedding z_q of x_q. Note that, different from treating x_q as one feature vector, the neural network (NN) in the first layer of E_1 treats each query segment $x_q^{(j)}$ as input separately (see Section 3.1 for more details).

[Data segmentation.] This is to group similar data together as

*Guoliang Li is the corresponding author. This paper was supported by NSF of China (61925205, 61632016), BNRist, Huawei, and TAL education.

SIGMOD '21, June 18–27, 2021, Virtual Event , China

ACM ISBN 978-1-4503-8343-1/21/06…$15.00
https://doi.org/10.1145/3448016.3452790

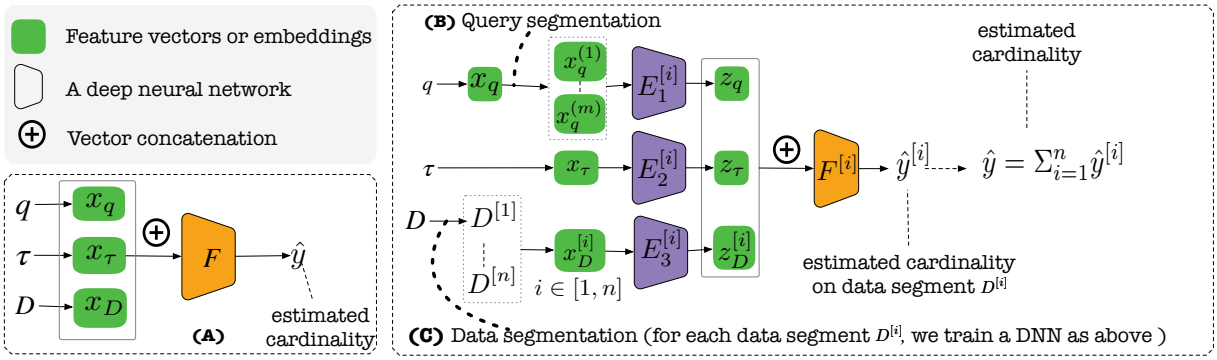


Figure 1: Solution Overview

non-overlapping data segments $\{D^{[1]}, \ldots, D^{[n]}\}$, and train n *local-models*, with the i-th local-model for $D^{[i]}$. Each local-model has three DNNs, $E_1^{[i]}$, $E_2^{[i]}$ and $E_3^{[i]}$ for learning embeddings of x_q, x_τ and $x_D^{[i]}$ respectively, and one DNN ($F^{[i]}$) to estimate the cardinality on $D^{[i]}$ relative to query q and threshold τ (i.e., $\hat{y}^{[i]} = F^{[i]}(z_q \oplus z_\tau \oplus z_D^{[i]})$), as shown in Figure 1(C).

The overall estimated cardinality $\hat{y}$ is the summation of local estimates from all local-models, i.e., $\hat{y} = \Sigma_{i=1}^{n} \hat{y}^{[i]}$.

Contributions. We summarize our contributions below.

(1) Problem Statement. We define the problem of cardinality estimation for similarity queries. (Section 2)

(2) Learned Estimation for Similarity Search. We describe our design of an end-to-end DNN model for estimating the cardinality for similarity search. We further present two enhancements, based on query segmentation and data segmentation. (Section 3)

(3) Learned Estimation for Similarity Join. We extend the above framework to support similarity join (i.e., a set of queries). The main challenge is to effectively decide which queries should be routed to which local-models to be evaluated. (Section 4)

(4) Implementation Details. We provide the details for the used DNNs, and algorithms for selecting hyperparameters. (Section 5)

(5) Experiments. We conduct extensive experiments to show the effectiveness and efficiency of our approach. (Section 6)

2 PRELIMINARIES

Dataset. A dataset D consists of a set of data objects, with each data object p a d-dimensional vector x_p.

Let $|x_p|$ denote the length of x_p, e.g., $|x_p| = d$.

Objects (e.g., images, text, tuples) are often transformed to feature vectors or embeddings [10] for using deep learning (DL) techniques.

Queries. The feature vector of a query q is denoted by x_q. A query set Q is a collection of query objects with the same dimensions.

Note that, the data/query objects for one dataset have the same d-dimension. While, data/query objects for different datasets may have different number of dimensions.

Distance Functions. The key to similarity search is the distance function $\text{dis}(x_p, x_q)$ between two objects p and q. The smaller the distance is, the more similar they are.

Naturally, distance functions are application-dependent. For example, L_m-Norm [15] is typically used for datasets where values in all dimensions are equally important. Angular distance [15] is widely used for datasets with sparse vectors containing lots of zero values, and Hamming distance [15] is normally used for datasets that compare each dimension with binary result.

Error Metrics for Cardinality Estimation. The most commonly used errors for regression problems are Mean Absolute Percentage Error (MAPE) [40] and Q-error [28, 39], which are defined as:

$$\text{MAPE}(\widehat{\text{card}}, \text{card}) = |\frac{\widehat{\text{card}} - \text{card}}{\text{card}}|$$

$$Q\text{-error}(\widehat{\text{card}}, \text{card}) = \frac{\max(\widehat{\text{card}}, \text{card})}{\min(\widehat{\text{card}}, \text{card})}$$

where $\widehat{\text{card}}$ is the estimated cardinality and card is the real cardinality. Note that, in practice, using MAPE as loss function makes the model prone to underestimate the cardinality. Q-error can overcome this drawback, but it may reduce the percentage of error when the error is close to 1. (If $\min(\widehat{\text{card}}, \text{card}) = 0$, we set it with a small value, e.g., 0.1.) In summary, when MAPE exceeds 1, Q-error works better; otherwise, MAPE works better. Therefore, we consider both metrics when training DL models.

Given a dataset D, a distance function dis(), and a distance threshold τ, we study two cardinality estimation problems.

Problem 1: *Cardinality estimation for similarity search (i.e., a query q).* Let $\text{card}(q, \tau, D)$ be the number of data objects in D whose distances to q are no greater than τ, i.e., $\text{card}(q, \tau, D) = \Sigma_{p \in D} \text{dis}(x_q, x_p) \leq \tau$. The problem is to provide an estimate $\widehat{\text{card}(q, \tau, D)}$.

Problem 2: *Cardinality estimation for similarity join (i.e., a query set Q).* Let $\text{card}(Q, \tau, D)$ be the number of all pairs of objects (q, p) where $q \in Q$, $p \in D$ and $\text{dis}(x_q, x_p) \leq \tau$. The problem is to provide an estimate $\widehat{\text{card}(Q, \tau, D)}$.

Because dataset D is always clear from the context, we will write $\widehat{\text{card}(q, \tau)}$ and $\widehat{\text{card}(Q, \tau)}$ when it is clear.

There are three **desired properties** for cardinality estimation: *accuracy*, *efficiency* and *monotonicity*.

(1) *Accuracy* measures the "goodness" of the estimated number, comparing with its real number.

(2) *Efficiency* is naturally desired for cardinality estimators.

(3) *Monotonicity with threshold guarantee* ensures that for any search

Variable	Notation	Description
x_q: a query vector	$x_q^{(i)}$	the i-th query segment
D: a dataset	$D^{[i]}$	the i-th data segment
Q_{batch}: a training batch	$Q_{\text{batch}}^{\{i\}}$	the i-th training sample
Q: a set of queries	$Q^{\langle i \rangle}$	the i-th query object

Table 1: Notations using Superscripts

Algorithm 1: Training the Basic DL Model

Input: Q_{train}: a set of triples (q, τ, card), and dataset D
Output: Return the trained model

1 **for** *number of training iterations* **do**
2 **for** $N <$ *number of batches* **do**
3 $Q_{\text{batch}}, D_{\text{sample}} \leftarrow$ get-minibatch (Q_{train}, N, D);
4 $\widehat{\text{card}} \leftarrow$ model.forward_propagate ($Q_{\text{batch}}, D_{\text{sample}}$);
5 $loss \leftarrow |\frac{e^{\widehat{\text{card}}} - \text{card}}{\text{card}}| + \lambda \cdot \frac{max(e^{\widehat{\text{card}}}, \text{card})}{min(e^{\widehat{\text{card}}}, \text{card})}$;
6 model.backward_propagate ($loss$);
7 **return** model;

object, the estimate for a bigger threshold is no less than the estimate for a smaller threshold.

Moreover, we summarize notations using *superscripts* in Table 1.

3 LEARNED CARDINALITY ESTIMATION FOR SIMILARITY SEARCH

3.1 A Basic DL Model for Similarity Search

Feature Vectors. The basic way of using a DNN is to convert the input (q, τ, D) to their feature vectors (x_q, x_τ, x_D) and concatenate them to one feature vector as the input x (i.e., $x = x_q \oplus x_\tau \oplus x_D$) of a DNN F, as shown in Figure 1(A).

The ***query feature vector*** x_q has d-dimensions, where d is application dependent. For example, if q is a greyscale image (i.e., 1 channel) with $28 * 28$ pixels, then x_q is a $28 * 28 = 784$ dimensional vector; if q is tuple with d numeric attributes, then q itself is a vector x_q; if q is categorical value, then x_q could be its 1-hot encoding where d is the number of distinct categories. Of course, x_q could be distributed representations (for example, q is a word and x_q its word embedding).

The ***distance feature vector*** x_τ is just a one-dimensional vector for the distance threshold τ.

The ***data feature vector*** x_D has k-dimensions, where each dimension is the distance between a data sample to query q, and we use k *data samples* instead of the entire dataset D.

By concatenating them together, the input $x_q \oplus x_\tau \oplus x_D$ in Figure 1(A) is a $(d + 1 + k)$-dimensional vector.

Learning Embeddings for Feature Vectors. Instead of directly concatenating feature vectors $x_q \oplus x_\tau \oplus x_D$ as described above, we propose to use three neural networks (NNs), E_1, E_2 and E_3, to learn the embeddings of x_q, x_τ and x_D, which will result in their corresponding embeddings z_q, z_τ and z_D, i.e., $z_q = E_1(x_q)$, $z_\tau = E_2(x_\tau)$ and $z_D = E_3(x_D)$, as shown in Figure 2. The three embeddings are then concatenated as one vector $z_q \oplus z_\tau \oplus z_D$ to the network F, which will be trained to estimate $\hat{y}$ that is $\widehat{\text{card}(q, \tau, D)}$.

Loss Function. As our model aims at solving a regression problem, the loss function should make the output (i.e., the estimated cardinality) close to the true cardinality. There are ***two challenges***: (1) The cardinalities vary from zero to millions, thus it's hard to fit them all. (2) Either MAPE or Q-error has its limitations. If we use MAPE as loss, the model is prone to underestimate cardinalities; if we use Q-error as loss, it will ignore errors when it is small.

For challenge (1), we regress the logarithm instead of the true cardinality. For challenge (2), we adopt a hybrid loss function which combines MAPE and Q-error. The loss function is formulated as:

$$\mathcal{J}(\theta) = |\frac{e^{\widehat{\text{card}}} - \text{card}}{\text{card}}| + \lambda \cdot \frac{max(e^{\widehat{\text{card}}}, \text{card})}{min(e^{\widehat{\text{card}}}, \text{card})}$$

where $\widehat{\text{card}}$ is the estimated cardinality, card is the true cardinality, and λ is a tunable weight (i.e., a hyperparameter).

Model Training (Algorithm 1). Each training item contains a query q, its corresponding threshold τ, and its true cardinality card. For each epoch (lines 1–6), and each batch (lines 2–6), it first gets a mini-batch and k data samples (line 3), performs forward-propagation to compute the estimate (line 4), calculates the loss (line 5), and then runs backward-propagation using gradient-descent to update the model parameters (line 6).

3.2 Query Segmentation

As will be shown in Section 6, directly applying a DNN on the entire query feature vector x_q often produces poor estimates, especially when the query and data objects have high dimensions. One promising direction is to divide x_q into multiple lower dimensional vectors, and learn the embedding z_q of x_q from these lower dimensional vectors, with the intuition that it is easier to estimate the distances between lower than higher dimensional vectors.

The key question is: *Whether the distance between two high dimensional vectors can be computed from their divided low dimensional vectors?* Next let's illustrate with an example.

Example 1. *Let u be a vector with segments $u^{(1)}$ and $u^{(2)}$, where $u = u^{(1)} \oplus u^{(2)}$, $|u| = d$, and $|u^{(1)}| = |u^{(2)}| = d/2$. Let $V = \{v_1, v_2, v_3, v_4\}$ be a set of four vectors, with each $v_i \in V$ with dimension $|v_i| = |u|$, and is divided into two vectors $|v_i^{(1)}| = |v_i^{(2)}| = d/2$ ($i = [1, 4]$). The L1-norm distances between u and v_i ranges from 0.1 to 0.6, as shown in Figure 4.*

We can first learn a function $f()$ using a DNN to output which segments have distance 0.1. *For example, $f(u^{(1)}, 0.1) = 0101$ indicates that the distance between $u^{(1)}$ and $v_2^{(1)}$ or $v_2^{(4)}$ is* 0.1. *Afterwards, we can merge the distribution (which is a binary indicator in this example) of the two query segments by conducting a learned function $g()$ (which is a* **bitand** *operation here), and we can generate the density at each distance threshold for the final distribution. For example, we have $g(u, 0.3) = f(u^{(1)}, 0.1) \& f(u^{(2)}, 0.2) + f(u^{(1)}, 0.2) \& f(u^{(2)}, 0.1)$ because if L1-norm distance of $u^{(1)}$ and $v_1^{(1)}$ is 0.1 and distance of $u^{(2)}$ and $v_1^{(2)}$ is 0.2, then distance of u and v_1 is $0.1 + 0.2 = 0.3$.*

Example 1 shows that we can estimate on query segments, and then combine them to get the overall estimate, for discrete distances.

Learning Continuous Distribution Functions. Next we discuss how to support a continuous distance threshold τ.

Consider a query vector x_q, which is segmented to $x_q^{(1)}$ and $x_q^{(2)}$. Let $f()$ be the distance-aware data distribution function of

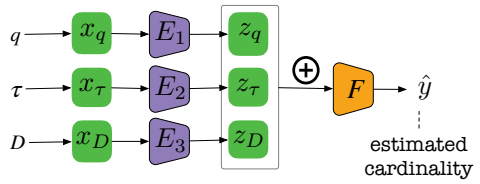


Figure 2: Improving Fig. 1(A) by Learning Embeddings

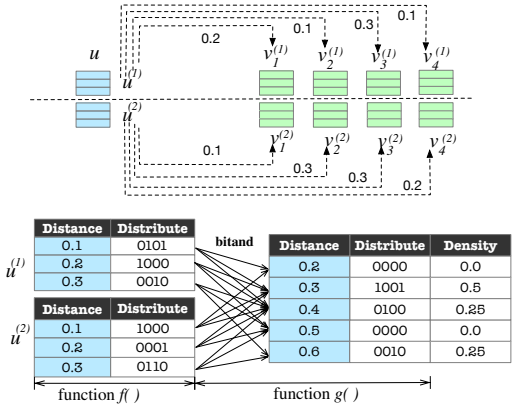


Figure 4: From Query Segment Density Distributions to Query Density Distribution for Discrete Distances

segments and $g()$ be the function for merging the segments distributions. If $x_q^{(1)}$ and $x_q^{(2)}$ are independent, then we have: $g(x_q, \tau) = \int_0^\tau operation(f(x_q^{(1)}, \tau - t), f(x_q^{(2)}, t))dt$.

The reason we learn the query segment distribution by $f()$ and merge $f()$ by another function $g()$ is that segments are not independent, simply multiply the density of each segment would loss a lot of information (e.g., $\int_0^\tau density(x_q^{(1)}, \tau - t) \cdot density(x_q^{(2)}, t)dt$).

Hence, we need to learn two functions, $f()$ for segment-level distribution and $g()$ for combining the distributions from different segments. Both $f()$ and $g()$ are DNNs (or multi-layer NNs) because DNNs can learn functions by fitting training data and offer fast evaluation. The output is a latent vector z_q, the embedding of x_q.

Query Segmentation. The feature vector x_q of a query q can be divided into n equal-length segments as $\{x_q^{(1)}, x_q^{(2)} \ldots, x_q^{(n)}\}$. The size of each segment $x_q^{(i)}$ is $|x_q^{(i)}| = \lceil \frac{d}{n} \rceil$, where d is the length of x.

Given the query segments $\{x_q^{(1)}, x_q^{(2)} \ldots, x_q^{(n)}\}$ of x_q, we unfold the module E_1 in Figure 2, as shown in Figure 3. In Figure 2, E_1 takes x_q as input and produces a latent vector z_q; in Figure 3, it takes query segments $\{x_q^{(1)}, x_q^{(2)} \ldots, x_q^{(n)}\}$ as input and produces a (possibly different) latent vector z_q.

More specifically, it has l layers, e_1–e_l. The first layer e_1 learns the density distribution of each segment $x_q^{(i)}$ (i.e., the function $f()$ in Figure 4) and the layers e_2–e_l learn to merge segments recursively (i.e., the function $g()$ in Figure 4).

Note that, l is a hyperparameter. All e_i's in the same layer are identical, i.e., they share the same weight matrix W, bias matrix B, and activation function (e.g., ReLU).

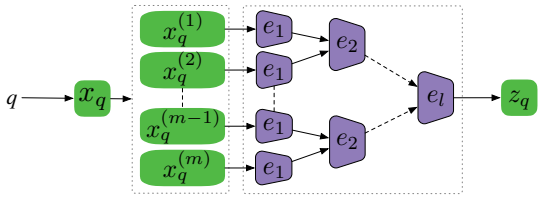


Figure 3: Query Segmentation (Revise E_1 in Figure 2)

Supporting Other Distance Functions. Most distance functions used for similarity search on feature vectors can be computed from their corresponding distances on query segments. Next, we will discuss some basic distance functions, including Euclidean distance ($L2$ distance), Manhattan distance ($L1$ distance), Cosine distance, Angular distance and Hamming distance, between vectors u and v.

L_m Distance. The L_m distance can be written as:

$$\text{dis}_{L_m}(u, v) = \sqrt[m]{\sum_{j=1}^{d}(u[j] - v[j])^m}$$

$$= \sqrt[m]{\sum_{i=1}^{n}\sum_{j=|u^{(i)}|\cdot(i-1)}^{|u^{(i)}|\cdot i}(u[j] - v[j])^m} = \sqrt[m]{\sum_{i=1}^{n}(\text{dis}_{L_m}(u^{(i)}, v^{(i)}))^m}$$

Therefore, the L_m distance can be rewritten as the summation of L_m distances on query segments.

Cosine Distance. The cosine distance is the cosine of the angle between two vectors, and it is closely related to Euclidean distance. Assume the input vectors have been normalized, then the normalized cosine distance can be computed as:

$$\text{dis}_{\cos}(u, v) = 1 - \frac{u \cdot v}{|u| \cdot |v|} = |u| \cdot |v| - u \cdot v$$

$$= \frac{u^2 + v^2 - 2uv}{2} = \frac{\text{dis}_{L_2}(u, v)}{2}$$

Therefore, If $|u| = 1$ and $|v| = 1$, the cosine distance equals to euclidean distance, and can be expressed as the summation of euclidean distances of query segments.

Angular Distance. Angular distance is the angle of the cosine distance, and can be expressed as the summation of cosine distances of segments:

$$\text{dis}_{\text{angular}}(u, v) = \frac{\arccos \text{dis}_{\cos}(u, v)}{\pi}$$

The angular distance is more usable than cosine distance because its value is always between 0 and 1.

Hamming Distance. The hamming distance is calculated by the number of unmatched tokens in corresponding position of two vectors (or strings) which can be formulated as:

$$\text{dis}_{\text{ham}}(u, v) = \sum_{j=1}^{d}\text{equal}(u[j], v[j])$$

$$= \sum_{i=1}^{n}\sum_{j=|u^{(i)}|\cdot(i-1)}^{|u^{(i)}|\cdot i}\text{equal}(u[j], v[j]) = \sum_{i=1}^{n}\text{dis}_{\text{ham}}(u^{(i)}, v^{(i)})$$

Here, $u[j]$ denotes the j-th element of vector u. Hence, Hamming distance can also be expressed as the summation of distance of query segments.

Moreover, Jaccard distance on finite sets can be transformed to an equivalent Hamming distance on binary sets. Consider a

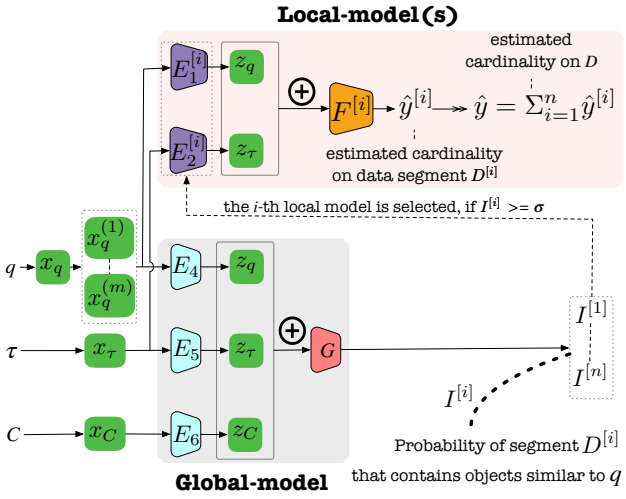


Figure 5: The Global-local Model for Similarity Search

universe set $\{a, b, c, d\}$. Let $u = \{a, b, c\}$ and $v = \{a, b, d\}$. The Jaccard distance is $\frac{2}{4} = 0.5$. u, v can also be represented as $x_u = \{1, 1, 1, 0\}$ and $x_v = \{1, 1, 0, 1\}$, the Hamming distance is also 0.5.

3.3 Data Segmentation

Data Segmentation. We want to divide D into a set of n segments as $\{D^{[1]}, \ldots, D^{[n]}\}$, such that data objects within a segment are similar and across segments are dissimilar. We use a simple and efficient segmentation method which uses Principal Component Analysis (PCA) to reduce the dimensionality first and then divide data by using batch K-means [16]. Note that, we have compared Locality Sensitive Hashing [20, 35, 58], DBSCAN, and K-means; K-means with PCA shows the best on both accuracy and efficiency.

Model Structure. After data segmentation, we will train a local-model for each data segment $D^{[i]}$, as shown in Figure 1(C). More specifically, it contains a module $E_1^{[i]}$ that transforms a query vector to its embedding (i.e., $z_q = E_1^{[i]}(x_q)$), a module $E_2^{[i]}$ that transforms the threshold to its embedding (i.e., $z_\tau = E_2^{[i]}(x_\tau)$), a module $E_3^{[i]}$ that transforms k-dimensional distance vector from k samples in $D^{[i]}$ to its embedding (i.e., $z_D^{[i]} = E_3^{[i]}(X_D^{[i]})$), and a module $F^{[i]}$ that computes the cardinality estimate on data segment $D^{[i]}$ (i.e., $\hat{y}^{[i]} = F^{[i]}(z_q \oplus z_\tau \oplus z_D^{[i]})$).

Note that, the module E_1 could either be the simple model that takes x_q as input (i.e., Figure 2), or the modified module that takes query segments as input (i.e., Figure 3). We will discuss algorithms for selecting hyperparameters in Section 5.2.

A Global-local Framework for Selecting Local-Models. Data segmentation improves the accuracy of the model. However, using all local-models for estimation is costly. Intuitively, only a small number of data segments are needed for a given low selectivity query. Hence, we propose a *global-local* framework that trains a global-model to decide which local-models should be used, in order to improve the efficiency.

The *local-model* for a data segment $D^{[i]}$ is similar to what we have described in Figure 1(C). The minor difference is that we remove the sample distance vector x_D in Figure 1(C) and use x_C in Figure 5 instead. Here, the distance vector x_C represents the distances between the query vector x_q to all centroids data segments, so $|x_C| = n$ where n is the total number of data segments. The reason to remove the sample distance vector x_D is because the distance distribution in each data segment can be easily learned by the other layers faster, under the global-local framework.

The *global-model* G is, given a query x_q and a threshold τ, to decide which data segments may contain data objects that are similar to x_q. In other words, the global-model is to select local-models that may produce non-zero estimates.

Next we provide more details about how the global-local framework works in Figure 5. The modules E_4, E_5 and E_6 will learn the embeddings z_q, z_τ and z_C of query vector x_q, distance threshold vector x_τ and distance vector to the centroids of all data segments x_C, respectively. The global-model G is trained to produce high probabilities for the data segments that may contain data objects similar to x_q. That is, $E_6(z_q, z_\tau, z_C) = \{I^{[1]}, \ldots, I^{[n]}\}$, where each $I^{[i]}$ ($i \in [1, n]$) is the probability in $(0, 1)$, indicating the likelihood $D^{[i]}$ contains objects similar to x_q. For a local-model j, if $I^{[j]} > \sigma$ (e.g., $\sigma = 0.5$), then the local-model for $D^{[j]}$ will be used.

The loss function for global-model should have the features below: (1) It is differentiable. (2) The optimal solution can make the precision and recall of segment selection optimal. (3) It is aware of cardinality of each segment and avoid too many missing segments.

First, we hope that the global-model outputs the probability for each segment being selected, which can be denoted as $I^{\{j\}[i]}$, for the j-th query in a training mini-batch and the i-th data segment. While the probability of data segment $D^{[i]}$ not being selected is $1 - I^{\{j\}[i]}$. If the real label $R^{\{j\}[i]} = 1$ (i.e., it contains similar objects), we should maximize $\log I^{\{j\}[i]}$; otherwise, $R^{\{j\}[i]} = 0$, we should maximize $\log(1 - I^{\{j\}[i]})$.

Second, in order to guarantee not to miss segments with large cardinalities, we need to add an extra penalty. The approach is to give each data segment a normalized weight $\epsilon^{\{j\}[i]}$ based on the cardinality of data segment i with query j. Higher cardinality has higher weight. Hence, the model would prefer not missing large cardinalities for maximizing the likelihood.

The model can be formulated as follows:

$$\epsilon^{\{j\}[i]} = \frac{\text{card}^{\{j\}[i]} - \min_i \text{card}^{\{j\}[i]}}{\max_i \text{card}^{\{j\}[i]} - \min_i \text{card}^{\{j\}[i]}}$$

$$\mathcal{L}(\theta) = \frac{1}{n \times B_S} \sum_{i=1}^{n} \sum_{j=1}^{B_S} R^{\{j\}[i]} \log(I^{\{j\}[i]})(1 + \epsilon^{\{j\}[i]}) + (1 - R^{\{j\}[i]}) \log(1 - I^{\{j\}[i]})$$

$$\mathcal{J}(\theta) = -\frac{1}{n \times B_S} \sum_{i=1}^{n} \sum_{j=1}^{B_S} R^{\{j\}[i]} \log(I^{\{j\}[i]})(1 + \epsilon^{\{j\}[i]}) + (1 - R^{\{j\}[i]}) \log(1 - I^{\{j\}[i]})$$

where n is the number of data segments, B_S is the number of queries in a training batch, $\text{card}^{\{j\}[i]}$ is the true cardinality of query j on data segment i, $R^{\{j\}[i]}$ indicates whether data segment i is selected by query j and it's either 0 or 1, $I^{\{j\}[i]}$ is the estimated probability between 0 and 1. Min-max normalization is applied to the cardinality for adopting different queries, $\min_i \text{card}^{\{j\}[i]}$ is the minimal

Algorithm 2: Global Discriminative Model Training

Input: Q_{train} is the training set, labels R indicates data segments selected by each query.

Output: Return the converged *model*

1 **for** *number of training iterations* **do**
2 **for** $N <$ *number of batches* **do**
3 $Q_{\text{batch}}, D_{\text{sample}} \leftarrow$ get-minibatch (Q_{train}, N, D);
4 $I \leftarrow$ model.forward_propagate ($Q_{\text{batch}}, D_{\text{sample}}$);
5 loss $\leftarrow R\log(I)(1+\epsilon) + (1-R)\log(1-I)$;
6 model.backward_propagate ($loss$);
7 **return** model;

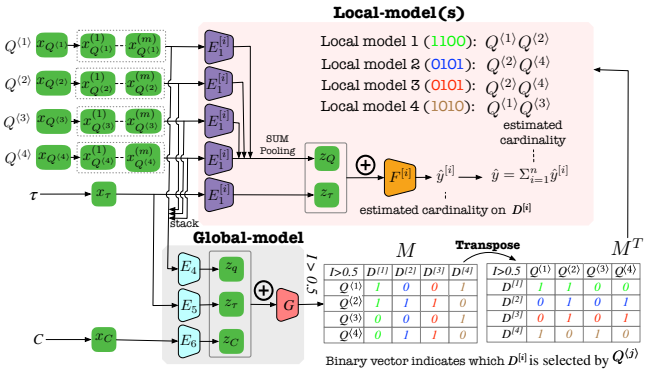


Figure 6: The Global-local Model for Similarity Join

cardinality of query j on all data segments, and $\max_i \text{card}\{j\}[i]$ is the maximal cardinality of query j on all data segments. $\mathcal{L}(\theta)$ is the likelihood given true labels. $\mathcal{J}(\theta)$ is the loss function, and minimizing $\mathcal{J}(\theta)$ is equivalent to maximizing $\mathcal{L}(\theta)$.

Model Training (Algorithm 2). The global-local model training has two phases. Phase 1 trains a local regression model for each data segment, which is similar to Algorithm 1 and is thus omitted here. Phase 2 is to train a global discriminative model on all the data segments, which is given in Algorithm 2. It first loops over the number of epochs (lines 1–6). For the inner loop (lines 2-6), it first gets the mini-batch (line 3), performs forward-propagation to compute the estimate (line 4), computes the loss comparing output probability and the label (line 5), and then runs backward-propagation to update the model parameters (line 6).

4 SUPPORTING SIMILARITY JOINS

Similar to Figure 5, we also employ a *global-local* framework for similarity join, as shown in Figure 6. The ***key difference*** for similarity join is, each local-model computes one embedding z_Q for all the queries that will be evaluated on this local-model, not for a single query z_q. The queries that will be evaluated on which local-models are decided by the global-mode, as discussed below.

Note that we will also use Figure 6 as a running example, for which we assume the query set Q contains four queries $\{Q^{\langle 1\rangle}, Q^{\langle 2\rangle}, Q^{\langle 3\rangle}, Q^{\langle 4\rangle}\}$ and the dataset D is divided into four data segments $\{D^{[1]}, D^{[2]}, D^{[3]}, D^{[4]}\}$.

Global-Model. Given a set Q of query objects with each associated with a threshold τ over a dataset D, the global-model G first predicts a binary *indicating vector* for each query $q \in Q$, indicating those data segments that may contain objects similar to q relative to τ.

That is, the global-model G will output a 2-dimensional *indicating matrix* M. For example, the 1st row (1, 0, 0, 1) of M in Figure 6 means that $D^{[1]}$ and $D^{[4]}$ may contain data objects that are similar to $Q^{\langle 1\rangle}$. The meaning of the other rows in M is similar.

Mask-based Routing. Next, we transpose M as M^T, then each row in M^T acts as a *mask*, indicating which queries will be routed to which data segments. The mask is to remove queries with zero cardinality, so as to improve both efficiency and effectiveness.

Local-Model. For each local model, the masked queries will be removed. For example, the first row (1, 1, 0, 0) in M^T indicates that queries $Q^{\langle 1\rangle}$ and $Q^{\langle 2\rangle}$ will be estimated on local model 1 relative to $D^{[1]}$, and similar for the other rows.

Query Set Embedding. We also need to modify the local-model such that the output module runs only once for a query set. To this end, we add a Sum Pooling layer between the output module and the query embedding module, which will combine multiple query embeddings into one embedding. This method has three advantages: (1) It is fast and small because there is no extra parameters being added. (2) It can easily generalize both the size and distribution of the join query set by sum pooling layer. (3) Experiments show that the modified model can be easily transferred from original model by training on a few samples and by only 2-3 iterations.

The estimated cardinality of the query set Q, relative to threshold τ and dataset D, is the sum all estimation from all local-models, shown at the bottom of Figure 6.

5 IMPLEMENTATION DETAILS

5.1 Details of DNN Models (Figure 7)

Query Embedding Network (E_1, E_4). Given a query feature vector x_q and its segments $x_q^{(1)}, \ldots, x_q^{(m)}$, we adopt convolutional neural network (CNN) to learn distribution function of segments $f()$ and combining function $g()$. Each convolutional layer contains one kernel filter and a pooling layer (omitted on the graph) that act like a distance density function adaptive to all the inputs segments. The first layer is to learn the function $f()$ of the distance aware distributions of query segments. The following layers learn the distribution progressively, with the last layer outputting the distribution for the entire query, i.e., the learned function $g()$.

Threshold Embedding Network (E_2, E_5). The threshold τ needs to be transformed to an embedding vector, for which we use a multilayer perceptron (MLP) with one hidden layer. In particular, for guaranteeing the monotonicity, we need to enforce all the weights in the threshold embedding to be positive to make the latent embedding monotonic with the threshold.

Distance Embedding Network (E_3, E_6). We need to have a feature vector x_D (resp. x_C) for E_3 (resp. E_6), to calculate the distances between k data points and the query. The difference is, for E_3, these k data points are data samples [28, 45]; for E_6, these k data points are the centroids of all data segments.

For E_6, we use centroids because data objects in the same data segment are relatively concentrated. If a query is "similar" to the centroid (i.e., the mean center of the segment) of some data segment, then it is likely that the query will be also similar to other objects

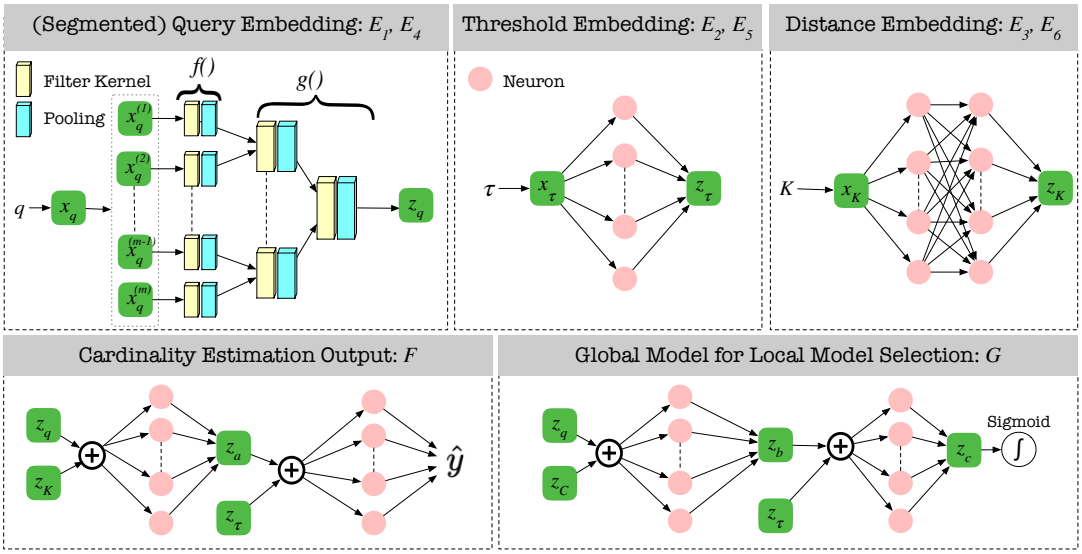


Figure 7: Cardinality Estimation for Similarity Search using CNN and Query Embedding Segmentation

in the same data segment. Otherwise, this query is less likely to be similar to objects in that segment. Using distances to centroids could increase the generality of the model, because we could compute the distance upper bound between a query and a data object in a data segment. This can be done by using triangle inequality on the distance of the query to the centroid, and this segment's radius.

In both cases, we use a MLP with two hidden layers. The reason to use two hidden layers, instead of one hidden layer, is to trade-off between size and efficiency. For each layer, we use ReLU [41] as the activation function.

Cardinality Estimation Output Network (F). So far, query embedding, distances embedding, and threshold embedding have been calculated and we use another dense layer and a linear layer (i.e., no ReLU activation) to transform them into the estimated cardinality.

Global Model for Local Model Selection Network (G). It is to output the probability of each data segment that contains objects similar to a given query. The added learnable threshold before the Sigmoid activator makes the output probability monotonic with the original threshold. The reason why we output the probability before discriminating it as 0 or 1 is that discriminative operation is non-derivable.

Global Discriminative Module. The probability from the *Output module* has considered both the query and the threshold. Hence, this module aims to discretize the probability of each data segment to 0 or 1 simply by a const value (e.g., 0.5), where probability above the value means 1, and otherwise 0. This module is only for measuring the model and select data segments but is not involved in model training, hence is not shown in Figure 7.

5.2 Selecting Hyperparameters for Query Embedding Network of Local Models

Hyperparameters of CNN have an impact on performance of local regression model significantly, but different datasets and hundreds of data segments make hyperparameters selection very hard.

Algorithm 3: Greedy Hyperparameters Tuning

Input: Q_{train} is the training set, card is the true cardinality of each query on current local model.
Output: Return the configured $model$

$S_{\text{train}} \leftarrow$ RandomSample(Q_{train}, card, 1000);
$S_{\text{validate}} \leftarrow$ RandomSample(Q_{train}, card, 200);
Θ_{full}=GetConfigs();
Θ_{init}=RandomSample(Θ_{full}, 3);
model $\leftarrow \emptyset$, error $\leftarrow \infty$;
$\theta, error' \leftarrow$ SelectBestFrom(Θ_{init}, model, S_{train}, S_{validate});
while $\frac{error-error'}{error} \geq 0.02$ **do**
  $error \leftarrow error', error'' \leftarrow error', error' \leftarrow \infty$;
  while $\frac{error'-error''}{error'} \geq 0.02$ **do**
    $error' \leftarrow error''$;
    $\theta, error'' \leftarrow$ Update(θ, model, S_{train}, S_{validate}, $error'$);
  model $\leftarrow$ Append(θ, model);
  $\theta, error' \leftarrow$ SelectBestFrom(Θ_{init}, model, S_{train}, S_{validate});
return model;

In this section, we propose an efficient automatic hyperparameter tuning method for query embedding module in local models, because it is the most complicated part. We first show all the tunable hyperparameters. We then give a definition of the hyperparameter optimization. Afterwards, we will present a greedy optimization algorithm and reduce the search space for improving performance.

Hyperparameters. We select some key hyperparameters for tuning, including number of channels (θ_{ch}), kernel size (θ_{ker}), kernel stride (θ_{stri}), padding size (θ_{pad}), Pooling size (θ_{pker}) and Pooling function (θ_{op}). We denote configuration of each layer in query embedding module as a tuple and each layer contains 6 tunable hyperparameters, it can be formulated as the following:

$$\Theta = \{\theta_{ch}, \theta_{ker}, \theta_{stri}, \theta_{pad}, \theta_{pker}, \theta_{op}\}$$
$$\theta_{ker}, \theta_{stri}, \theta_{pad}, \theta_{pker}, \theta_{ch} \in \mathcal{N}^+$$
$$\theta_{op} \in \{\text{MAX, AVG, SUM}\}$$

where $\mathcal{N}^+$ is the positive integer set.

id	Method	Embed	Auto-tuning	Framework	Opt	Data Segment
1	QES	CNN	No	Local	Select	No
2	Local+	CNN	Yes	Local	Select	Yes
3	GL-MLP	MLP	No	Global-Local	Select	Yes
4	GL-CNN	CNN	No	Global-Local	Select	Yes
5	GL+	CNN	Yes	Global-Local	Select	Yes
6	CardNet	VAE	No	Local	Select	No
7	Sampling	-	No	-	Select	No
8	Kernel-based	-	No	-	Select	No
9	MLP	MLP	No	Local	Select	No
10	SimSelect	-	-	-	Select	-
11	CNNJoin	CNN	No	Local	Join	No
12	GLJoin	MLP	No	Global-Local	Join	Yes
13	GLJoin+	CNN	Yes	Global-Local	Join	Yes

Table 2: Tested Algorithms

Problem Formulation. Given a local data segment D_i, the optimization objective is to minimize the validation error of estimated cardinalities, and can be formulated as: $\hat{\Theta} = arg\min_{\Theta} \mathcal{J}(\Theta)$, where $\mathcal{J}(\Theta)$ is the loss function defined in Section 5.1 on validation queries, however, the hyperparameters (structure of query embedding module) are variables in this Section.

Greedy Solution. In order to avoid too many times model training for trial, we propose a greedy solution for each data segment, as given in Algorithm 3. We first obtain a subset of training and validating data by random sampling (lines 1-2), and all the trials are conducted on this subset. We then randomly select 3 configurations from the range of hyperparameters (lines 3-4) for cold start (the range is shown in next part). We continuously select the optimal hyperparameters for a new layer and put it on the model until the error does not decrease any more (lines 6-13). In each layer, we start from the best one of the 3 configurations, and update (line 11) all 6 hyperparameters mentioned above in turn until convergence. Finally, we return the optimal hyperparameter configuration of query embedding module (line 14).

5.3 Supporting Data Updates

GL+ model supports incremental learning for updates because GL+ is highly modular. More specifically, each data point of the dataset belongs to only one cluster. If several data points is inserted/deleted in an operation, we first distribute these data points to the nearest clusters according to the distances with centroids, and then update query labels in clusters and incrementally train local models and the global model.

6 EXPERIMENTS

Datasets. The statistics of all used datasets are shown in Table 3, similar to a related work [53]. BMS [5] contains product entries. ImageNET [7] contains one-hot vectors of images preprocessed by HashNet [11]. GloVe300 [2] contains 300-dimension distributed representations of words. YouTube [6] contains raw face images from YouTube videos. Aminer [4] and DBLP [1] contain binary vectors transformed from publication titles by using the method proposed in [53]. Metrics in Table 3 show the raw distance metric of datasets, and the Jaccard and Edit distance have been transformed to Hamming distance by the methods proposed in [3, 53].

Dataset	Dimension	#Data	#Training	#Testing	Metric	τ_{max}
BMS	512	515,597	8,000	2,000	Jaccard	0.50
GloVe300	300	1,917,494	8,000	2,000	Angular	0.60
ImageNET	64	1,431,167	8,000	2,000	Hamming	0.90
Aminer	2,943	1,712,433	4,000	1,000	Edit	0.05
YouTube	1,770	346,194	2,400	600	Euclidean	0.15
DBLP	5,373	1,000,000	2,400	600	Edit	0.20

Table 3: Datasets

Algorithms for similarity search (Table 2).

[*Our methods (rows 1-5).*] (1) QES uses CNNs for query segmentation (Section 5.1). (2) Local+ adopts data segmentation but removing the global model (i.e., only local models) in Figure 1, and it employs automatic hyperparameter selection for each local model (Section 5.2). (3) GL-MLP uses data segmentation but without using query segmentation, i.e., using MLP for query embedding. (4) GL-CNN uses both query segmentation and data segmentation. (5) GL+ improves GL-CNN by using the hyperparameter tuning algorithm in Section 5.2. In addition, methods 2–4 use a penalty in global loss to avoid missing data segments with large cardinalities.

[*Competitors (rows 6-10).*] (6) CardNet is the state-of-the-art method proposed in a SIGMOD 2020 paper [53][‡]. (7) Sampling is a sampling-based method. We test on 1% random samples, 10% random samples and samples with the same size with GL+ model. For a query, we calculate the results on the samples and estimate the cardinality by the sample ratios. (8) Kernel-based method models distance density distribution for each sample by Gaussian function, and estimates cardinality of a query as sum of cumulative density of all samples. (9) DL-based MLP uses fully connected NNs for query/distance/threshold embeddings. (10) SimSelect is a state-of-the-art exact threshold-based similarity search methods [44], which can return the exact results with an efficient index.

Algorithms for similarity join. Beside Sampling-based approaches, SimSelect, and CardNet, we also compare different variants of our proposal, as discussed below.

(11) CNNJoin uses a sum pooling for combining all segmented query embeddings, but does not use data segmentation. (12) GLJoin does not perform query segmentation, but adopts data segmentation. (13) GLJoin+ uses both query and data segmentation. It also uses the same tuned hyperparameters as used by GL+. We also use estimation methods of similarity search as baselines for join estimates.

Query Selection. We randomly select a set of points in dataset as similarity search queries Q, and divide it into Q_{train}(80%) and Q_{test}(20%). For each q in Q_{train}, we uniformly generate 10 thresholds from range [0, τ_{max}] by selectivities just as paper [53] does (where τ_{max} is the maximal threshold we support for a realistic similarity search query). In order to show the generalizability of our methods, for each q in Q_{test}, we generate 10 thresholds from range [0, τ_{max}] according to geometrical distribution of selectivities (more queries with lower selectivity). Typically, both training and testing queries have selectivities less than 1% of size of dataset (in line with conventions of many similarity search researches [34, 44, 58]). For each training join set, we first select the size N from range [1, 100), and then select N queries from Q_{train}. We also evenly select 10 thresholds from the threshold range. For testing, we generate three

[‡]Code was obtained from the authors.

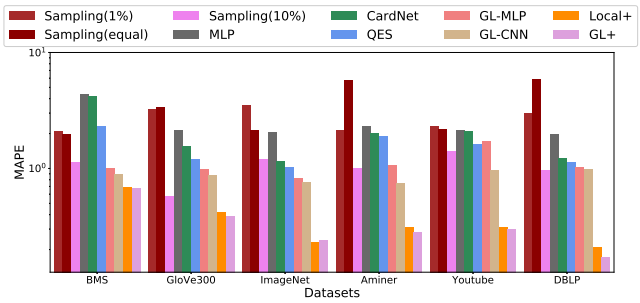


Figure 8: MAPE of Different Methods

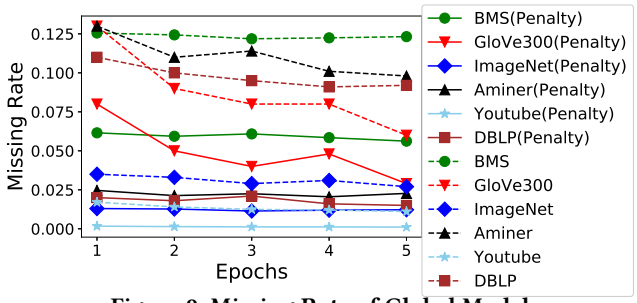
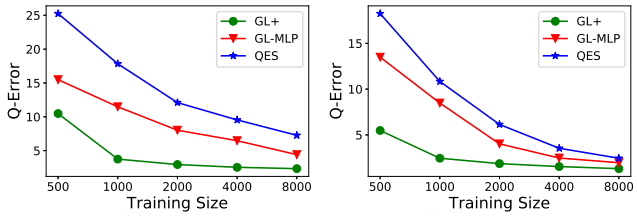


Figure 9: Missing Rate of Global Model



(a) BMS **(b) ImageNet**

Figure 10: Errors with Training Size

types of join sets according to the ranges of set size which are [50, 100), [100, 150), and [150, 200), respectively. For each testing join set, we randomly select 10 thresholds from the threshold range.

Implementation. Our experiments are conducted on 40 cores of Intel(R) Xeon(R) CPU E5-2630v4@2.20GHz, and 128 Gigabytes memory. The DL models are trained in PyTorch 1.0.1, and then we copy parameters of model to a C++ implementation for testing. Baselines (e.g., `Sampling`) are implemented in C++ and use parallel computing to optimize the efficiency.

Default settings. All methods add the penalty to loss functions, and we will compare with not using penalty in Exp-6. The default number of training queries is given in Table 3, and we will vary the training sizes in Exp-7. The default numbers of data segments for these datasets are 100, and we will vary these numbers in Exp-8.

6.1 Evaluation for Similarity Search

Table 4 shows the results of different methods on all testing datasets. The best results are highlighted in bold font. The Mean is the average error, Median is the median error, and 90th/95th/99th is the error larger than 90%/95%/99% of all errors.

Exp-1. [Non-DL (`Sampling` (1%) and `Kernel-based`) vs. DL (`MLP`).] Model in `MLP` method is very small, and thus we compare it with baselines based on small samples. Table 4 tells us that: (i) For mean Q-errors, `MLP` outperforms traditional methods on most datasets, because `Sampling` suffers from 0-tuple problem and `Kernel-based` cannot fit the distance distribution well. (ii) On some datasets (e.g.,Aminer), `Sampling` (1%) and `Kernel-based` outperform `MLP` with median and max Q-error because sampling-based methods can accurately estimate queries with higher cardinalities(e.g., 10,000). (iii) `MLP` often produces largest max Q-errors because the generality of `MLP` for very high-dimensional data is worse (e.g., DBLP is 5373-dimension and YouTube is 1770-dimension).

Exp-2. [`Sampling` (10% and equal) vs. `GL+`.] Because the accuracy of sample-based methods increases with the sample size increases, we extend the sample size to the size of `GL+` model and 10%, and compare the accuracy of `Sampling` and `GL+`. For Q-error, Table 4 shows that `GL+` can outperform `Sampling` (equal) by one order of magnitude, and is comparable with `Sampling` (10%).

Exp-3. [No Query Segmentation (`MLP` and `CardNet`) vs. Query Segmentation (`QES`).] Table 4 shows that `QES` outperforms `MLP` by nearly 40% on mean error of BMS, more than 50% on mean and median error of GloVe300 and ImageNet, nearly 40% on Q-errors of Aminer and 30% on YouTube. `QES` also outperforms `CardNet` on mean error of all datasets. Figure 8 compares them using error metric MAPE, which shows that `QES` also outperforms `MLP` and `CardNet` on all the datasets, because CNN can catch distance density distribution of segments better, and brings better generality.

Exp-4. [No Data Segmentation (`QES` and `CardNet`) vs. Data Segmentation (`GL-MLP` and `GL-CNN`).] Table 4 shows that segment data and train a local model for each segment bring significantly accuracy enhancement on all datasets. For Q-errors, `GL-CNN` outperforms `QES`, `MLP` and `CardNet`, and it produces only 1.83 mean error and 1.27 median error on Aminer which is more than 3 times better than `QES`. In addition, Figure 8 evaluates using MAPE, which shows consistent results with the observation from Table 4.

Exp-5. [Same Configuration (`QES`, `GL-MLP`, `GL-CNN`) vs. Automatic Local Hyperparameter Tuning (`GL+`).] Table 4 is for Q-errors and Figure 8 is for MAPE. They tell us that `GL+` has the best accuracy on all metrics for all datasets. For Q-errors, `GL+` reduces the mean error on GloVe300, ImageNet, Aminer and DBLP below 1.6. On BMS, `GL+` improves mean accuracy by about 30% comparing to `GL-CNN`. For MAPE, `GL+` reduces the error to 0.17 on DBLP, which is very accurate, because local hyperparameter tuning can make the query embedding module adaptive to each local distribution.

Exp-6. [No Penalty vs. With Penalty.] Note that, all methods use penalty in loss function by default. This group of experiment is to study what if we remove penalty for these methods, shown in Figure 9. It tells us that adding a penalty to loss function of global model training can reduce the cardinality missing by global. In particular, the cardinality missing is reduced by around 100% on BMS, 90% on GloVe300, and nearly 4 times on DBLP, because loss function with penalty can avoid missing data segments with large cardinalities. Moreover, the judiciously designed loss function makes the global index model very accurate, and thus the accuracy of `GL+` is similar to that of `Local+` according to Table 4 and Figure 8.

Dataset	Method	Mean	Median	90th	95th	99th	Max
BMS	GL+	**2.34**	1.09	**2.47**	**4.32**	19.7	111
	Local+	2.37	**1.05**	2.51	4.36	18.4	**98.3**
	Sampling (10%)	5.18	1.83	11.2	17.4	55.0	165
	GL-CNN	3.50	2.42	8.21	10.6	**15.7**	291
	GL-MLP	4.41	3.02	9.78	12.8	19.7	439
	QES	7.27	5.05	16.5	21.6	32.2	644
	CardNet	12.4	5.16	31.3	48.8	99.1	335
	MLP	11.2	8.03	36.8	47.7	71.0	700
	Kernel-based	12.8	8.81	29.7	39.2	59.5	135
	Sampling (equal)	12.3	7.0	31.0	41.0	74.0	111
	Sampling (1%)	19.6	13.0	55.0	66.9	74.0	200
GloVe300	GL+	**1.45**	**1.11**	3.39	5.84	19.2	210
	Local+	1.51	1.29	3.44	6.05	18.8	241
	Sampling (10%)	1.67	1.20	**1.86**	**2.36**	20.0	**35.0**
	GL-CNN	2.11	1.46	4.79	6.39	**9.60**	166
	GL-MLP	2.20	1.53	5.04	6.62	10.2	208
	QES	3.57	2.46	8.37	10.7	16.1	341
	CardNet	4.78	2.20	8.71	14.0	40.2	1099
	MLP	7.29	5.07	16.7	21.8	33.2	753
	Kernel-based	15.1	10.6	35.2	45.4	67.8	148
	Sampling (equal)	27.9	4.74	84.0	113	204	274
	Sampling (1%)	25.7	3.88	63.0	113	152	274
ImageNET	GL+	**1.31**	**1.04**	**2.0**	**2.23**	4.36	45.0
	Local+	1.35	1.14	2.11	3.13	**4.12**	52.3
	Sampling (10%)	2.12	1.57	2.73	3.43	15.0	**26.0**
	GL-CNN	1.62	1.12	3.73	4.89	7.55	71.5
	GL-MLP	1.96	1.35	4.55	5.90	8.74	142
	QES	2.45	1.71	5.67	7.41	11.1	222
	CardNet	3.07	2.0	6.02	8.48	16.4	89.4
	MLP	5.43	3.78	12.4	16.0	24.9	442
	Kernel-based	11.6	8.15	26.7	34.7	52.4	155
	Sampling (equal)	8.78	2.23	26.0	35.0	85.0	114
	Sampling (1%)	22.0	6.40	63.0	85.0	152	204

Dataset	Method	Mean	Median	90th	95th	99th	Max
Aminer	GL+	**1.54**	**1.07**	**2.05**	**2.98**	7.79	152
	Local+	1.61	1.12	2.36	3.01	**6.46**	321
	Sampling (10%)	2.41	1.72	3.90	5.26	14.2	**31.0**
	GL-CNN	1.83	1.27	4.21	5.39	8.38	154
	GL-MLP	3.09	2.14	7.10	9.18	14.2	290
	QES	5.22	3.63	11.9	15.4	24.4	541
	CardNet	5.45	2.05	7.59	12.9	43.1	3526
	MLP	8.39	5.80	19.4	25.1	38.6	780
	Kernel-based	9.85	6.91	22.6	28.7	44.6	117
	Sampling (equal)	66.5	42.0	182	245	245	245
	Sampling (1%)	19.5	4.20	56.0	75.0	136	245
YouTube	GL+	**1.69**	**1.04**	**2.29**	**3.93**	13.3	98.7
	Local+	1.70	1.12	2.55	5.78	12.1	58.5
	Sampling (10%)	3.82	1.90	9.0	12.0	21.1	**50.0**
	GL-CNN	2.52	1.74	5.88	7.59	**11.2**	241
	GL-MLP	4.12	2.88	9.57	12.3	18.8	394
	QES	6.65	4.68	15.3	19.9	29.4	801
	CardNet	13.2	5.47	29.4	54.8	126	205
	MLP	9.82	5.13	34.5	45.0	67.3	1191
	Kernel-based	10.8	7.50	25.1	32.3	49.5	102
	Sampling (equal)	14.9	9.0	37.0	50.0	50.0	**50.0**
	Sampling (1%)	15.4	9.0	37.0	50.0	50.0	**50.0**
DBLP	GL+	**1.49**	**1.05**	**2.31**	**2.88**	9.22	102
	Local+	1.52	1.16	2.55	3.62	7.13	156
	Sampling (10%)	2.16	1.86	4.0	4.42	**7.0**	**21.0**
	GL-CNN	2.01	1.38	4.64	6.01	9.32	196
	GL-MLP	3.20	2.23	7.25	9.60	15.2	298
	QES	4.61	3.19	10.6	13.8	21.2	425
	CardNet	4.59	2.01	9.33	20.1	51.3	474
	MLP	4.77	3.12	14.2	26.4	38.9	1047
	Kernel-based	5.63	3.87	12.9	16.8	26.0	54.2
	Sampling (equal)	34.2	10.5	128	128	234	234
	Sampling (1%)	9.15	4.0	21.0	38.0	70.0	70.0

Table 4: Test Errors for Similarity Search

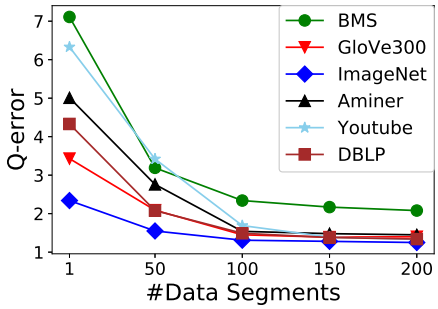


Figure 11: Mean Errors of Varying #-Data Segments

Model	BMS	GloVe300	ImageNET	Aminer	YouTube	DBLP
Sampling (1%)	12.7	27.7	3.66	243	24.5	239
MLP	4.11	3.09	3.21	9.01	8.23	15.3
QES	0.25	0.17	0.18	0.41	0.35	0.58
CardNet	38.8	35.3	16.2	54.5	52.8	55.1
GL-MLP	111	106	101	176	171	203
GL-CNN	29.2	21.3	7.32	35.6	32.1	55.6
GL+	28.3	22.1	7.51	34.2	30.7	50.1
GLJoin+	30.1	21.5	9.04	35.9	31.8	59.1

Table 5: Model Size Comparison (MB)

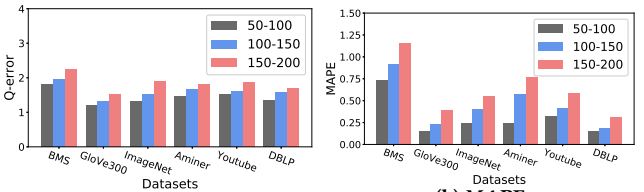
Exp-7. [Varying Training Sizes.] Figure 10 shows how Q-error decreases with training size increasing of methods GL+, GL-MLP and QES on BMS and ImageNet. We don't show the results for other datasets because they have similar results and the space is limited. It tells us that increasing training size can increase the accuracy of all three learning-based methods. The accuracy of QES increases drastically when training size increases from 500 to 4000, the accuracy GL-CNN increases drastically when training size increases from 500 to 3000, and GL+ increases drastically when training size increases from 500 to 1000. Also, the smaller training size is, the more GL+ outperforms other two methods. The reason is that more training queries can reveal more distribution details from more perspectives, and segmentation-based method with CNN query embedding increases the ability of catching proper distance distribution with limited training queries.

Exp-8. [Varying #-Data Segmentations.] Figure 11 depicts how Q-error decreases with data segments increasing of method GL+ on all datasets. It shows that mean Q-errors reduce drastically when data segments number increases from 1 to 100, and errors are reduced by nearly 7 times on YouTube and 4 times on BMS. The main reason is that local models can learn more details on smaller clusters.

Exp-9. [Estimation Time.] Table 6 shows the average time of estimating cardinality for a similarity search query of different methods on all datasets. It shows that traditional methods Sampling and

Model	BMS	GloVe300	ImageNET	Aminer	YouTube	DBLP
SimSelect	3.96	12.1	5.22	5.87	12.5	18.6
Kernel-based	10.3	15.1	6.43	125	21.3	138
Sampling (10%)	30.9	70.1	10.5	587	69.5	598
Sampling (equal)	6.78	6.77	2.31	9.56	3.26	2.55
Sampling (1%)	3.21	7.23	1.12	61.4	7.46	61.5
CardNet	0.36	0.18	0.13	0.68	0.62	0.73
Local+	1.46	1.12	0.79	5.12	2.55	3.24
GL-MLP	0.51	0.65	0.28	3.43	2.35	3.69
GL-CNN	0.35	0.21	0.15	0.81	0.49	0.55
GL+	0.33	0.22	0.13	0.80	0.53	0.57
MLP	0.14	0.11	0.046	0.18	0.15	0.27
QES	0.015	0.012	0.007	0.042	0.021	0.032

Table 6: Avg. Latency for Similarity Search (milliseconds)



(a) Q-errors **(b) MAPE**

Figure 12: Join Errors with Query Set Size

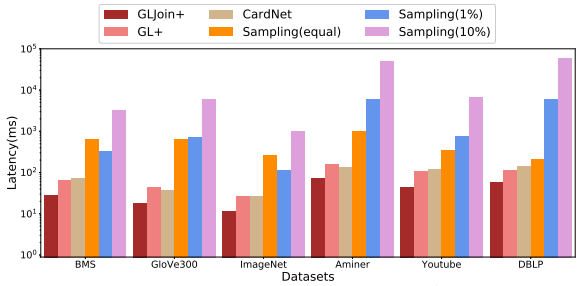


Figure 13: Avg. Latency for Similarity Join (query size = 200)

Kernel-based are much slower than our methods. For example, on dataset GloVe300, our methods outperform traditional approaches by nearly 1 order of magnitude. The reasons are three folds. (i) Our model is very small, we can see from table 5 that GL+ model is even smaller than 1% samples on datasets GloVe300, Aminer and DBLP. (ii) Only a part of parameters participate the estimation for a query because of the global index for GL+ or the dropout for DNN. (iii) Neural network is mainly composed of matrix multiplications, and can utilize hardware efficiently. However, Sampling methods conduct lots of online high-dimension distance computing. For traditional methods, Sampling is faster because Kernel-based needs an extra Gaussian process for each sample when estimating cardinality. It also tells us that GL+ outperforms Local+ by 5 times on all datasets, because for queries with low selectivities, only several local models need to be evaluated with the help of the global selection model. We can also see than GL+ is much faster than SimSelect.

Exp-10. [Training and Query Construction Time.] Figure 14 shows the training time and query construction time of learning-based estimation methods on different datasets. We can see several facts. First, the overhead of training query construction time is non-negligible because the construction computes the distances between all pairs of datasets and queries, and thus it's necessary to constraint the amount of training queries. Second, GL+ takes 2 times more

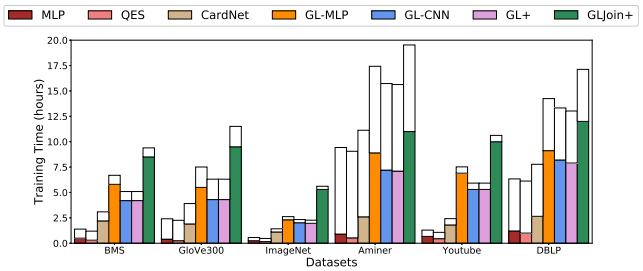


Figure 14: Training and Label Time

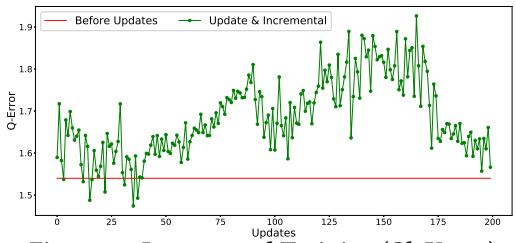


Figure 15: Incremental Training (GloVe300)

overhead for training than CardNet because GL+ has to train 50-100 light-weighted local models separately. This is a trade-off to pay for offline training get better online evaluation accuracy. Offline training is done once and we can support incremental training for data updates. Third, GLJoin+ takes the longest training time because a join query may contain hundreds of vectors. Last, MLP and QES are very fast for training but they suffer from bad accuracy.

We didn't include the data clustering time, because it takes less than 1 minute, which is negligible compared with the training time.

Exp-11. [Data Updates.] We incrementally inserted 2K new records on GloVe300 by 200 update operations each with 10 records. After each update, we update the labels for 8K queries and incrementally train the existing model, which takes 1-3 minutes, while a retraining from scratch takes several hours. We compared the Q-error of incremental training with error before updates. Figure 15 tells us that incremental learning can constantly keep high accuracy of the estimation model with hundreds of update operations.

6.2 Evaluation for Similarity Join

Exp-12. [Accuracy.] The comparison of cardinality estimation methods for similarity join is shown in Table 7. It shows that methods with data segmentation still do a good job in accuracy. Comparing with Q-errors in Table 4, grouping query embeddings in a join set together by sum pooling brings a better accuracy than single estimates. The reason is that sum pooling can keep most of the informations of single queries. It also tells us that model with data and query segmentation is still the best. For example, GLJoin+ outperforms small samples by 1-2 orders of magnitude on BMS, GloVe300, YouTube and DBLP, and GLJoin+ also outperforms CardNet by 2-8 times.

Figure 12 shows the Q-errors and MAPEs of methods GL+ for different sizes of join query sets. We observe that on both Q-error and MAPE, sum-pooling based join query embedding can generalize to different join query sizes, the performance decay for grouping 100-200 queries is still moderate and reasonable. The reason is

Dataset	Method	Mean	Median	90th	95th	99th	Max
BMS	GLJoin+	**1.87**	**1.31**	**4.31**	**5.51**	**8.55**	174
	GL+	2.01	1.36	4.59	6.12	9.34	205
	Sampling (10%)	3.99	2.18	8.46	13.5	23.1	**37.0**
	GLJoin	2.51	1.72	5.78	7.56	11.5	265
	CNNJoin	5.63	3.90	12.9	16.9	26.2	508
	CardNet	8.35	5.88	19.1	25.2	37.2	857
	Sampling (equal)	19.3	2.50	15.2	40.9	302	451
	Sampling (1%)	144	3.86	451	800	1505	2701
GloVe300	GLJoin+	1.22	**1.02**	1.83	3.70	5.62	119
	GL+	1.36	1.03	2.14	4.08	6.23	131
	Sampling (10%)	**1.18**	1.13	**1.38**	**1.46**	**1.69**	**2.06**
	GLJoin	1.86	1.30	4.28	5.48	8.22	170
	CNNJoin	4.34	3.02	9.94	12.6	19.8	457
	CardNet	4.22	2.92	9.81	12.5	19.0	348
	Sampling (equal)	20.6	1.39	96.0	171	231	416
	Sampling (1%)	22.4	1.56	96.0	128	231	311
ImageNET	GLJoin+	**1.31**	1.03	2.91	3.79	6.02	134
	GL+	1.32	**1.02**	3.03	3.98	5.91	117
	Sampling (10%)	1.67	1.64	**2.06**	**2.21**	**2.40**	**3.09**
	GLJoin	2.15	1.47	4.95	6.44	10.0	192
	CNNJoin	7.39	5.15	16.9	21.9	34.9	727
	CardNet	3.09	2.11	7.23	9.28	14.0	274
	Sampling (equal)	5.57	1.73	3.04	5.67	96.2	126
	Sampling (1%)	7.38	1.73	2.90	71.0	96.2	171

Dataset	Method	Mean	Median	90th	95th	99th	Max
Aminer	GLJoin+	**1.42**	**1.08**	3.26	4.16	6.26	121
	GL+	1.70	1.18	3.95	5.10	7.94	171
	Sampling (10%)	2.06	1.90	**2.90**	**3.35**	**4.57**	**5.12**
	GLJoin	2.02	1.40	4.66	5.94	9.25	193
	CNNJoin	6.58	4.67	15.2	19.6	30.5	788
	CardNet	5.16	3.55	11.7	15.2	24.3	766
	Sampling (equal)	124	7.77	371	501	909	1221
	Sampling (1%)	5.96	1.94	3.98	5.21	86.2	151
YouTube	GLJoin+	**1.54**	**1.06**	3.59	4.67	**7.01**	126
	GL+	1.61	1.12	3.73	4.87	7.39	122
	Sampling (10%)	1.82	1.32	**1.95**	**2.46**	16.0	**31.0**
	GLJoin	2.23	1.54	5.19	6.71	10.3	216
	CNNJoin	6.54	4.47	15.3	19.9	29.9	628
	CardNet	9.98	6.91	23.1	29.9	44.7	943
	Sampling (equal)	15.6	2.38	41.0	56.0	101	136
	Sampling (1%)	31.9	16.0	101	136	246	246
DBLP	GLJoin+	**1.31**	**1.06**	**2.95**	**3.96**	**6.13**	123
	GL+	1.43	1.07	3.01	4.25	6.89	111
	Sampling (10%)	2.51	1.37	6.0	6.0	16.0	**16.0**
	GLJoin	1.98	1.35	4.54	5.92	9.14	268
	CNNJoin	4.54	3.12	10.5	13.6	20.7	389
	CardNet	5.14	3.62	11.7	15.2	24.1	502
	Sampling (equal)	221	56.0	636	1166	1166	1166
	Sampling (1%)	12.9	3.11	31.0	56.0	186	186

Table 7: Test Errors for Similarity Join (size $\in [50, 100)$)

that sum pooling can incorporate the number of queries in the aggregated embedding.

Exp-13. [Batch Embedding vs. Single Embedding.] Figure 13 shows the latency of average estimating cardinalities for a join set with 200 queries. We observe that batch evaluation in GLJoin+ is much faster than evaluation for each query in GL+, and less layers in query embedding layer and lower dimensionality of query makes the superiority more obvious. Sampling (10%) is the slowest because the sample size is too large. On DBLP, 10% means 100,000 samples, and we should conduct $100,000 \times 200$ distance computations for each join query on 5,373 dimensions.

7 RELATED WORK

Learning-based Cardinality Estimation for Exact Queries. Malik et al [36] first classify queries according to the query structure (join condition, attributes in predicates etc.), and then train a model on the values of the predicates. [28] trains a multi-set convolutional network on queries. [42] proposes a vision of training representation for the join tree with reinforcement learning. [21, 55] propose deep likelihood models to capture the data distribution of multiple attributes and estimates the cardinality of conjunctive queries. [45] proposes an end-to-end learning-based cardinality and cost estimator. [43] proposes a selectivity estimation method by using uniform mixture model. However, these methods only support exact range queries[13, 14, 24, 31–33, 51, 56, 57] and cannot support distance-aware similarity queries because cardinalities of similarity queries are related to both query vector and distance threshold. Also, similarity queries do not follow the transitivity property [59, 60].

Cardinality Estimation for Similarity Queries. [37] proposes a kernel-based method to estimate the cardinality for a similarity query, and they use Gaussian functions as the kernel function, and take the sum of cumulated probability of all kernel functions on samples as the cardinality. However, the kernel-based method still relies on samples, and suffers from 0-tuple problem with sparse data space. The methods [8, 9] first cluster all the existing queries and find a representative query (a.k.a., a "query prototype" or prototype for short) for adjacent queries. They then build threshold-based linear model on each prototype. For an unknown query, they project it to prototypes, collect cardinalities, and use weighted sum as the estimated cardinality. They show good result on low-dimension (≤ 10) datasets, but on high-dimension datasets, it's hard to find prototypes and learn cardinalities by using a simple linear model.

Data Clustering. Unsupervised Hash-based methods transform high dimensional data or non-metric data into a short hash code [19, 49], where data with the same hash code will be put into the same bucket, and search nearest neighbors from several adjacent buckets. Hash-based methods include Local Sensitive Hashing [17, 20, 25, 34, 35, 48, 50, 58], Learning to Hash [22, 52, 54] and quantization-based methods [18, 23]. Traditional methods like K-means are often used to cluster data in low dimensionality. While for high dimensional data, K-means can be used for subspace of data [38], or dimension reduced data via methods like PCA [16].

8 CONCLUSION

In this paper, we have studied the feasibility of applying deep learning based methods on cardinality estimation for similarity queries. We have proposed two novel methods to improve the accuracy and to reduce the number of training data for similarity search query segmentation and data segmentation, and use a global-local framework to support both similarity search and similarity join. We have conducted extensive experiments to show that our proposed methods can significantly outperform existing solutions.

REFERENCES

[1] *https://dblp2.uni-trier.de/.*
[2] *https://nlp.stanford.edu/projects/glove/.*
[3] *https://scikit-learn.org/stable/modules/generated/sklearn.metrics.hamming_loss.html.*
[4] *https://www.aminer.cn/.*
[5] *https://www.kdd.org/kdd-cup/view/kdd-cup-2000.*
[6] *http://www.cs.tau.ac.il/ wolf/ytfaces/index.html.*
[7] *http://www.image-net.org/.*
[8] C. Anagnostopoulos and P. Triantafillou. Learning set cardinality in distance nearest neighbours. In *2015 IEEE International Conference on Data Mining*, pages 691–696, 2015.
[9] C. Anagnostopoulos and P. Triantafillou. Query-driven learning for predictive analytics of data subspace cardinality. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 11:1 – 46, 2017.
[10] Y. Bengio, A. C. Courville, and P. Vincent. Representation learning: A review and new perspectives. *IEEE Trans. Pattern Anal. Mach. Intell.*, 35(8):1798–1828, 2013.
[11] Z. Cao, M. Long, J. Wang, and P. S. Yu. Hashnet: Deep learning to hash by continuation. In *IEEE International Conference on Computer Vision, ICCV 2017, Venice, Italy, October 22-29, 2017*, pages 5609–5618. IEEE Computer Society, 2017.
[12] A. Chen, L. E. Li, and J. Cao. Tracking cardinality distributions in network traffic. In *INFOCOM*, pages 819–827. IEEE, 2009.
[13] D. Deng, G. Li, H. Wen, and J. Feng. An efficient partition based method for exact set similarity joins. *Proc. VLDB Endow.*, 9(4):360–371, 2015.
[14] D. Deng, Y. Tao, and G. Li. Overlap set similarity joins with theoretical guarantees. In *SIGMOD*, pages 905–920, 2018.
[15] M. M. Deza and E. Deza. *Encyclopedia of Distances.* Springer Berlin Heidelberg, 2009.
[16] C. H. Q. Ding and X. He. K-means clustering via principal component analysis. In *ICML*, volume 69, 2004.
[17] J. Gan, J. Feng, Q. Fang, and W. Ng. Locality-sensitive hashing scheme based on dynamic collision counting. In *SIGMOD*, pages 541–552, 2012.
[18] T. Ge, K. He, Q. Ke, and J. Sun. Optimized product quantization for approximate nearest neighbor search. In *CVPR*, pages 2946–2953, 2013.
[19] Z. Gharibshah, X. Zhu, A. Hainline, and M. Conway. Deep learning for user interest and response prediction in online display advertising. *Data Science and Engineering*, 5(1):12–26, 2020.
[20] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *VLDB*, pages 518–529, 1999.
[21] S. Hasan, S. Thirumuruganathan, J. Augustine, N. Koudas, and G. Das. Deep learning models for selectivity estimation of multi-attribute queries. In D. Maier, R. Pottinger, A. Doan, W. Tan, A. Alawini, and H. Q. Ngo, editors, *SIGMOD*, pages 1035–1050. ACM, 2020.
[22] J. He, W. Liu, and S. Chang. Scalable similarity search with optimized kernel hashing. In *SIGKDD*, pages 1129–1138, 2010.
[23] H. Jégou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *TPAMI*, 33(1):117–128, 2011.
[24] Y. Jiang, G. Li, J. Feng, and W. Li. String similarity joins: An experimental evaluation. *Proc. VLDB Endow.*, 7(8):625–636, 2014.
[25] X. Jin and J. Han. Locality sensitive hashing based clustering. In *Encyclopedia of Machine Learning and Data Mining*, pages 758–759. Springer, 2017.
[26] T. Kim, W. Li, A. Behm, I. Cetindil, R. Vernica, V. R. Borkar, M. J. Carey, and C. Li. Similarity query support in big data management systems. *Inf. Syst.*, 88, 2020.
[27] D. P. Kingma and M. Welling. An introduction to variational autoencoders. *Foundations and Trends in Machine Learning*, 12(4):307–392, 2019.
[28] A. Kipf, T. Kipf, B. Radke, V. Leis, P. A. Boncz, and A. Kemper. Learned cardinalities: Estimating correlated joins with deep learning. In *CIDR*, 2019.
[29] H. Lee, R. T. Ng, and K. Shim. Power-law based estimation of set similarity join size. *Proc. VLDB Endow.*, 2(1):658–669, 2009.
[30] H. Lee, R. T. Ng, and K. Shim. Similarity join size estimation using locality sensitive hashing. *Proc. VLDB Endow.*, 4(6):338–349, 2011.
[31] G. Li, D. Deng, J. Wang, and J. Feng. PASS-JOIN: A partition-based method for similarity joins. *Proc. VLDB Endow.*, 5(3):253–264, 2011.
[32] G. Li, J. He, D. Deng, and J. Li. Efficient similarity join and search on multi-attribute data. In *SIGMOD*, pages 1137–1151, 2015.
[33] G. Li, J. Hu, J. Feng, and K. Tan. Effective location identification from microblogs. In *ICDE*, pages 880–891, 2014.
[34] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li. Multi-probe LSH: efficient indexing for high-dimensional similarity search. In *VLDB*, pages 950–961, 2007.
[35] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li. Intelligent probing for locality sensitive hashing: Multi-probe LSH and beyond. *PVLDB*, 10(12):2021–2024, 2017.
[36] T. Malik, R. C. Burns, and N. V. Chawla. A black-box approach to query cardinality estimation. In *CIDR*, pages 56–67, 2007.
[37] M. Mattig, T. Fober, C. Beilschmidt, and B. Seeger. Kernel-based cardinality estimation on metric data. In *EDBT*, pages 349–360, 2018.
[38] D. Mautz, W. Ye, C. Plant, and C. Böhm. Discovering non-redundant k-means clusterings in optimal subspaces. In *SIGKDD*, pages 1973–1982, 2018.
[39] G. Moerkotte, T. Neumann, and G. Steidl. Preventing bad plans by bounding the impact of cardinality estimation errors. *Proc. VLDB Endow.*, 2(1):982–993, 2009.
[40] A. D. Myttenaere, B. Golden, B. L. Grand, and F. Rossi. Mean absolute percentage error for regression models. *Neurocomputing*, 192:38–48, 2016.
[41] V. Nair and G. E. Hinton. Rectified linear units improve restricted boltzmann machines. In *ICML*, pages 807–814, 2010.
[42] J. Ortiz, M. Balazinska, J. Gehrke, and S. S. Keerthi. Learning state representations for query optimization with deep reinforcement learning. In *DEEM@SIGMOD*, pages 4:1–4:4, 2018.
[43] Y. Park, S. Zhong, and B. Mozafari. Quicksel: Quick selectivity learning with mixture models. In D. Maier, R. Pottinger, A. Doan, W. Tan, A. Alawini, and H. Q. Ngo, editors, *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 1017–1033. ACM, 2020.
[44] J. Qin and C. Xiao. Pigeonring: A principle for faster thresholded similarity search. *PVLDB*, 12(1):28–42, 2018.
[45] J. Sun and G. Li. An end-to-end learning-based cost estimator. *PVLDB*, 13(3):307–319, 2019.
[46] J. Sun, Z. Shang, G. Li, Z. Bao, and D. Deng. Balance-aware distributed string similarity-based query processing system. *PVLDB*, 12(9):961–974, 2019.
[47] J. Sun, Z. Shang, G. Li, D. Deng, and Z. Bao. Dima: A distributed in-memory similarity-based query processing system. *PVLDB*, 10(12):1925–1928, 2017.
[48] Y. Tao, K. Yi, C. Sheng, and P. Kalnis. Quality and efficiency in high dimensional nearest neighbor search. In *SIGMOD*, pages 563–576, 2009.
[49] S. Tian, S. Mo, L. Wang, and Z. Peng. Deep reinforcement learning-based approach to tackle topic-aware influence maximization. *Data Science and Engineering*, 5(1):1–11, 2020.
[50] H. Wang, J. Cao, L. Shu, and D. Rafiei. Locality sensitive hashing revisited: filling the gap between theory and algorithm analysis. In *CIKM*, pages 1969–1978, 2013.
[51] J. Wang, G. Li, and J. Feng. Can we beat the prefix filtering?: an adaptive framework for similarity join and search. In *SIGMOD*, pages 85–96, 2012.
[52] J. Wang, W. Liu, S. Kumar, and S. Chang. Learning to hash for indexing big data - A survey. *Proceedings of the IEEE*, 104(1):34–57, 2016.
[53] Y. Wang, C. Xiao, J. Qin, X. Cao, Y. Sun, W. Wang, and M. Onizuka. Monotonic cardinality estimation of similarity selection: A deep learning approach. 2020.
[54] Y. Weiss, A. Torralba, and R. Fergus. Spectral hashing. In *NIPS*, pages 1753–1760, 2008.
[55] Z. Yang, E. Liang, A. Kamsetty, C. Wu, Y. Duan, X. Chen, P. Abbeel, J. M. Hellerstein, S. Krishnan, and I. Stoica. Deep unsupervised cardinality estimation. volume 13, pages 279–292. VLDB Endowment, 2019.
[56] M. Yu, G. Li, D. Deng, and J. Feng. String similarity search and join: a survey. *Frontiers Comput. Sci.*, 10(3):399–417, 2016.
[57] M. Yu, J. Wang, G. Li, Y. Zhang, D. Deng, and J. Feng. A unified framework for string similarity search with edit-distance constraint. *VLDB J.*, 26(2):249–274, 2017.
[58] Y. Zheng, Q. Guo, A. K. H. Tung, and S. Wu. Lazylsh: Approximate nearest neighbor search for multiple distance functions with a single index. In *SIGMOD*, pages 2023–2037, 2016.
[59] X. Zhou, C. Chai, G. Li, and J. SUN. Database meets artificial intelligence: A survey. *IEEE Transactions on Knowledge and Data Engineering*, pages 1–20, 2020.
[60] X. Zhou, J. Sun, G. Li, and J. Feng. Query performance prediction for concurrent queries using graph embedding. *VLDB*, 13(9):1416–1428, 2020.