Appl Intell (2018) 48:2546–2567
https://doi.org/10.1007/s10489-017-1093-y

Scalable aggregation predictive analytics

A query-driven machine learning approach

Christos Anagnostopoulos[1] · Fotis Savva[1] · Peter Triantafillou[1]

Published online: 12 December 2017

Abstract We introduce a predictive modeling solution that provides high quality predictive analytics over aggregation queries in Big Data environments. Our predictive methodology is generally applicable in environments in which large-scale data owners may or may not restrict access to their data and allow only aggregation operators like COUNT to be executed over their data. In this context, our methodology is based on historical queries and their answers to accurately predict ad-hoc queries' answers. We focus on the widely used set-cardinality, i.e., COUNT, aggregation query, as COUNT is a fundamental operator for both internal data system optimizations and for aggregation-oriented data exploration and predictive analytics. We contribute a novel, query-driven Machine Learning (ML) model whose goals are to: (i) learn the query-answer space from past issued queries, (ii) associate the query space with local linear regression & associative function estimators, (iii) define query similarity, and (iv) predict the cardinality of the answer set of unseen incoming queries, referred to the Set Cardinality Prediction (SCP) problem. Our ML model incorporates incremental ML algorithms for ensuring high quality prediction results. The significance of contribution lies in that it (i) is the only query-driven solution applicable over general Big Data environments, which include restricted-access data, (ii) offers incremental learning adjusted for arriving ad-hoc queries, which is well suited for query-driven data exploration, and (iii) offers a performance (in terms of scalability, SCP accuracy, processing time, and memory requirements) that is superior to data-centric approaches. We provide a comprehensive performance evaluation of our model evaluating its sensitivity, scalability and efficiency for quality predictive analytics. In addition, we report on the development and incorporation of our ML model in Spark showing its superior performance compared to the Spark's COUNT method.

Keywords Query-driven predictive analytics · Predictive modeling · Aggregation operators · Set cardinality prediction · Regression vector quantization · Self-organizing maps

1 Introduction

Recent R&D efforts in the *modern* big data era have been dominated by efforts to accommodate distributed big datasets with frameworks that enable highly **quality** and **scalable** distributed/parallel data analyzes. Platforms such as MapReduce [14], Yarn [29], Spark [32] and Mahout [22] are nowadays commonplace. Predictive modeling [26], [23] and exploratory analysis [2, 3, 6, 20] are commonly based on statistical aggregation operators over the results of *exploration* queries [4, 7]. Such queries involve large datasets (which may themselves be the result of linking of other different datasets) and a number of range predicates over multidimensional data vectorial representation, structured, semi- and unstructured data. High quality query-driven data exploration and quality modeling is becoming increasingly important in the presence of large-scale data since *accurately* predicting aggregations over range predicate queries

✉ Christos Anagnostopoulos
christos.anagnostopoulos@glasgow.ac.uk

Peter Triantafillou
peter.triantafillou@glasgow.ac.uk

[1] School of Computing Science, University of Glasgow, Glasgow G12 8QQ, UK

is a fundamental data exploration task [12] in big data systems. Frequently, data analysts, data scientists, and statisticians are in search of approximate answers to such queries over unknown data subspaces, which supports knowledge discovery and underlying data function estimation. Imagine exploratory and predictive analytics [9] based on a *stream* of such aggregation operators over data subspaces being issued, until the scientists/analysts extract sufficient statistics or fit local function estimators, e.g., coefficient of determination, product-moment correlation coefficient, and multivariate local linear approximation over the subspaces of interest.

In modern big data systems like Spark [32], often data to be analyzed possibly extends over a large number of federated data nodes, perhaps even crossing different administration domains and/or where data owners (nodes) may only permit restricted accesses (e.g., aggregations) over their data. Similarly, in the modern big data era, large datasets are often stored in the Cloud. Hence, even when access is not restricted, accesses to raw data needed to answer aggregate queries are costly money-wise. Quality predictive modeling solutions which are widely applicable, even in such scenarios, are highly desirable.

Consider a d-dimensional data space $\mathbf{x} \in \mathbb{R}^d$.

Definition 1 (Range Query) Let a d-dim. box be defined by two boundary vectors $[a_1, \ldots, a_d]^\top$ and $[b_1, \ldots, b_d]^\top$, $a_i \leq b_i, a_i, b_i \in \mathbb{R}$. A range query is represented by the $2d$-dimensional vector $\mathbf{q} = [a_1, b_1, a_2, b_2, \ldots, a_d, b_d]^\top$ where a_i and b_i is lower and higher value, respectively, for the i-th dimension. Query $\mathbf{q}$ is a hyper-rectangle with faces parallel to the axes.

Definition 2 (Query Distance)[1] The normalized Euclidean distance between queries $\mathbf{q}$ and $\mathbf{q}'$ is $\|\mathbf{q} - \mathbf{q}'\|_2 = \frac{1}{\sqrt{2d}} \sum_{i=1}^{d} \left(a_i - a'_i\right)^2 + \left(b_i - b'_i\right)^2$, where $\frac{1}{\sqrt{2d}}$ is a normalization factor since $0 \leq \|\mathbf{q} - \mathbf{q}'\|_2 \leq \sqrt{2d}$.

Definition 3 (Answer Set Cardinality) Given a range query $\mathbf{q}$ and a dataset $\mathcal{B}$ of data points $\mathbf{x} \in \mathbb{R}^d$, $y \in \mathbb{N}$ is the cardinality of the answer set of those $\mathbf{x} \in \mathcal{B}$ in the interior of the hyper-rectangle defined by query $\mathbf{q}$ satisfying $a_i \leq x_i \leq b_i, \forall i$.

The reader could refer to Appendix for a nomenclature.

The reason we focus on the COUNT aggregation operator is that the answer Set Cardinality Prediction (SCP) of a multidimensional range query is a fundamental task, playing a central role in predictive modeling. With multidimensional range queries, analysts define the subspaces in $\mathbb{R}^d$ of interest within the overall data space. High quality cardinality prediction in such subspaces then becomes important for data mining, data exploration, time series analysis, and big data visualization tasks [9, 12] of data (sub)spaces of interest.

In predictive modeling, data scientists routinely define specific regions of a large dataset that are worth exploring and wish to derive and accurately predict statistics over the populations of these regions. This amounts to the SCP of the corresponding range queries. In addition to being an important aggregation operator, in database systems accurate cardinality prediction (which amounts to the well known selectivity estimation problem) is explicitly used for query processing optimization, empowering query optimizers to choose, for instance, the access plan which produces the smallest intermediate-query results (which have to be retrieved from disks and communicated over the network) saving time, resource waste, and money (e.g., in Clouds). Furthermore, SCP is a core operator in modern big data frameworks. Notably, in Spark [32] one of the five fundamental actions defined is the so-called *count* action, which is executed over the underlying raw data at each data node.

1.1 Motivation & research objectives

Well-established and widely adopted techniques for Approximate aggregation-Query Processing (AQP) based on sampling, histograms, self-tuning histograms, wavelets, and sketches [13] have been proposed. Their fundamental and naturally acceptable assumption is that the underlying data are always accessible and available, thus it is feasible to create and maintain their statistical structures. For instance, histograms [15] require scanning of all data to be constructed and being up-to-date; the self-tuning histograms [1] require additionally the execution of queries to fine tune their statistical structures; the sampling methods [16] execute the queries over the sample to extrapolate the cardinality prediction result.

Consider now a big data environment, where a federation of data nodes store large datasets. There are cases where the data access to these nodes' data may be either restricted, (e.g., government medical and DNA databases and demographic and neighborhood statistic datasets). Furthermore, many real-world large-scale data systems may limit the number of queries that can be issued and/or charge for excessive data accesses. For example, there may exist per-IP limits (for web interface queries) or per developer key limits (for API based queries). Even when the (daily) limit is high enough, repeated executions actually have high monetary cost (e.g., in cloud deployments), waste communication overhead due to remote query execution, and computational resources. The accessed data nodes can either fully execute

[1]When dealing with mixed-type data points, e.g., consisting of categorical & continuous attributes,we can adopt other distance metrics; this does not spoil the generality of our solution.

the queries (to produce exact results) or locally deploy an AQP technique to produce estimates. In the latter case, we must rely upon the SCP accuracy provided by the applied traditional AQP technique. Hence, the cardinality prediction accuracy is upper bounded by the predictability capability of the AQP method.

The above discussion raises the following desiderata: it is important to develop quality AQP techniques that:

– D1: are *applicable to all data-environment* scenarios (restricted-access or not),
– D2: are *inexpensive*, i.e., avoid relying on excessive querying of and communication with the data nodes, while
– D3: offering *high prediction accuracy*, and
– D4: being prudent in terms of compute-network-store resource utilization.

Let us consider an indicative baseline solution for AQP in our environment. One approach is to store, e.g., locally to a central node, all the AQP structures (e.g., histograms, samples, sketches, etc.) from a federation of data nodes. Thus, we can simply locally access this node for SCP. Firstly, this violates our first desideratum, as privacy issues emerge (data access restrictions). Obviously, retaining all AQP structures, provides one with the *whole* valuable information about the underlying data (e.g., in the case of histograms, we obtain the underlying probability data distribution $p(\mathbf{x})$, while in sampling methods we retain actual samples from the remote datasets). Even, in cases where the local accesses to AQP structures were secured (which is again subject to major security concerns), we would have to cope with the problem of AQP structure updates. The maintenance of those structures in the face of updates demands high network bandwidth overhead, cost for data transfer (in a Cloud setting), latency for communicating with the remote nodes during updates of the underlying dataset at these nodes, and scalability and performance bottleneck problems arise at the central node. Therefore, this approach does not scale well and can be expensive, violating our 2nd and 3rd criteria above.

An alternative baseline solution would be to do away with the central node and send the query to the data nodes, which maintain traditional AQP statistical structure(s) and send back their results to the querying node. As before, this violates many of our desiderata. It is not applicable to restricted-access scenarios (violating criterion 1) and involves heavy querying of the data node (violating criteria 2 and 4). Even if this was the case (by violating criteria 1, 2, and 4), the construction and maintenance of an AQP structure would become a prohibited solution; we struggle with huge volumes of data (data universe explosion phenomenon; imagine *only* the creation of a multidimensional histogram over 1 zettabyte). These facts help expose the formidable challenges to the problem at hand, (a significant problem for large-scale predictive analytics) which to the best of our knowledge, has not been studied before. In this work we study a query-driven SCP in a big data system taking into consideration the above-mentioned desiderata. Although significant data-centric AQP approaches for cardinality prediction have been proposed [13] a solution for our intended environments of use is currently not available.

There are three fundamental pressures at play here. The first pertains to the development of a solution for cardinality prediction that is **efficient**, and **scalable**, especially for distributed scale-out environments, wherein extra communication costs, remote invocation techniques, and estimation latency are introduced. The second pertains to the **quality** of cardinality prediction results in terms of accuracy and model fitting, where as we shall see traditional solutions fall short. The third concerns the wide-applicability of a proposed method, taking into account environments where data accesses may be restricted, We propose a solution that addresses all these tensions. Conceptually, its fundamental difference from related works is that it is **query-driven**, as opposed to data-driven, and is thus based on a ML model (trained by a number of queries sent to a data node) and later utilized to predict answers to new incoming queries.

The challenging aim of our approach is to swiftly provide cardinality prediction of ad-hoc, unseen queries while (i) *avoiding executing them over a data node*, saving communication and computational resources and money, and (ii) not relying on any knowledge on the $p(\mathbf{x})$, and any knowledge about nodes' data. Through our query-driven SCP, an inquisitive data scientist, who explores data spaces, issues aggregate queries, and discovers hidden data insights, can extract accurate knowledge, efficiently and inexpensively.

1.2 Related work

Given a d-dim. data space $\mathbf{x} \in \mathbb{R}^d$ the holy grail approaches focus on: (i) inspecting the (possibly huge) underlying dataset and estimate the underlying probability density function (pdf) $p(\mathbf{x})$. Histograms (typically multidimensional) as fundamental data summarization techniques are the cornerstone, whereby the estimation of $p(\mathbf{x})$ is highly exploited for SCP of range queries, e.g., [1, 15]. The traditional methods of building histograms do not scale well with big datasets. Histograms need to be periodically rebuilt in order to update $p(\mathbf{x})$ thus, exacerbating the overhead of this approach. Central to our thinking is the observation that a histogram is constructed solely from data, thus obviously being not applicable to our problem for the above-mentioned reasons. Histograms are also inherently unaware on the cardinality prediction requests, i.e., query patterns. Their construction method rely neither on query distribution $p(\mathbf{q})$ nor on joint $p(\mathbf{q}, y)$ but only on $p(\mathbf{x})$. As a result, such methods do not yield the most appropriate histogram for a given $p(\mathbf{q})$ [11]. The limitations of this method are also well-known [27, 30].

To partially address some of the above limitations, prior work has proposed self-tuning histograms (STHs) e.g., [1, 27]. The STHs learn a centrally stored dataset from scratch (i.e., starting with no buckets) and rely only on the cardinality result provided by the execution of a query, referred to as Query Feedback Records (QFR). STHs exploit the actual cardinality from QFR and use this information to build and refine traditional histograms. Formally, given a query $\mathbf{q}$ over data with cardinality y, the methods of STHs estimate the conditional $p(\mathbf{x}|y, \mathbf{q})$ since the main purpose is to construct and tune a histogram conditioned on query patterns. Fundamentally, the limitations in STHs in our problem stem from the fact that they estimate $p(\mathbf{x}|y, \mathbf{q})$, thus, having to data access (in multidimensional STHs, at least one scan of the set $\mathcal{B}$ is required), deal with the underlying data distribution and make certain assumptions of the statistical dependencies of data.

Other histogram-based cardinality prediction methods utilize wavelets [31] or entropy-based [28]; the list is not exhausted. Briefly, the idea is to apply wavelet decomposition to the dataset to obtain a compact data synopsis based on the wavelet coefficients. By nature, wavelets-based AQP relies on the synopsis construction over data thus could not be applied to our problem. Overall, STHs and the other advanced histogram-based approaches, are associated with data access for estimating $p(\mathbf{x})$ or any other $p(\mathbf{x}|\mathbf{q}, \ldots)$ thus not applicable in our problem. Sampling methods [16] have been also proposed for SCP. They share the common idea to evaluate the query over a small subset of the dataset and extrapolate the observed cardinality. Finally, another approach for AQP answering to SCP is data sketching; we refer the reader to [13] for a useful survey of sketching techniques. Sketching algorithms construct estimators from the raw data and yielding a function of these estimators as the answer to the query. Therefore, as discussed above, we neither have access to data nor to a sample of them, thus yielding the data sketching and sampling methods inapplicable to our problem.

In conclusion, the data-centric approaches in related work are not applicable to our problem since they require explicit access to data to construct their AQP structures and maintain them up-to-date. For this reason, our proposed solution to this novel setting is query-driven.

Our model can be highly useful when it is very costly (in time, money, communication bandwidth) to execute aggregation operators over the results of complex range queries (including joins of datasets and arbitrary selection predicates), when data are stored at the cloud, or at federations of data stores, across different administration domains, etc. And, to our knowledge, it is the only approach that can address this problem setting. It is worth noting that this paper significantly extends our previous work presented in [5]. The interesting reader could refer to [5] to assess the performance of our solution with respect to traditional data-centeric (AQP) systems for cardinality prediction namely with multidimensional histograms, popular self-tuning histograms, and sampling methods. In [5], through comprehensive experiments we showed that the query-driven approach, which extracts knowledge from the issued queries and corresponding answers, provides higher cardinality prediction accuracy and performance, while being more widely applicable. Based on the scalability and efficiency of this approach, we further generalize our model in [5] and implement generalized ML algorithms within the most popular big data system, Spark. Specifically, the major differences of the proposed generic ML model discussed in this paper with that of our paper in [5] are:

- We propose a generalization of the ML model in [5] by introducing (i) associative local linear regression models for cardinality prediction and (ii) the concept of the coefficients lattice in self-organizing maps statistical learning algorithm;
- We provide the theoretical analysis and convergence of the learning algorithms of the generalized ML model (Theorems 2 and 4);
- We implement our ML model within the Spark system;
- We provide comprehensive experiments showing the *quality of prediction* of our ML model through a variety of evaluation metrics.
- We experiment with the *scalability performance* of our ML model compared with the Spark's `COUNT` method for answer-set cardinality estimation.

1.3 Organization

The structure of the paper is as follows: Section 2 reports on the rationale of our approach and the research challenges for the SCP, while summarizes the contribution and our research outcome. In Section 3, we provide preliminaries for unsupervised & heteroassociative competitive statistical learning and the self-organizing maps along with the problem formulation for SCP. Section 4 provides the set cardinality learning methodology, the machine learning algorithms over the novel introduced lattice concepts and the fundamental convergence theorems of our neuron-based model. In Section 5 we provide an implementation of our model in the Spark system, while Section 6 reports on a comprehensive performance and comparative assessment with the build-in Spark `COUNT` over real large-scale datasets introducing different experimental scenarios. Finally, Section 7 concludes the paper with future research directions.

2 Challenges & overview

Our approach is query-driven. The first requirement (and challenge) of our approach is to incrementally learn the query patterns $p(\mathbf{q})$ at any time, thus being able to (i) detect possible changes to user interests on issuing queries and (ii) reason about the similarity between query patterns. The second requirement (and challenge) is to learn the association $\mathbf{q} \rightarrow y$ between a query $\mathbf{q}$ and its cardinality y, i.e., $p(y|\mathbf{q})$, thus being able to predict the cardinality. The third requirement (and challenge) is to learn such association without relying on the underlying $p(\mathbf{x})$ which in our case is totally unknown and inaccessible. The fourth requirement (and challenge) is to update $p(\mathbf{q})$ and $p(\mathbf{q}, y)$ based on changes in query patterns and to data. Query distributions are known to be non-uniform, with specific portion of the data space being more popular. However, query patterns change with time, reflecting changes of users interests to exploring different sections of the datasets of nodes. Hence, we must swiftly adapt and learn on-the-fly the *new* query patterns, updating $p(\mathbf{q}, y)$ and $p(\mathbf{q})$. Furthermore, updates on the underlying datasets of nodes can independently occur, altering $p(\mathbf{x})$. We must also deal with such mutations, implying the need to maintain the current $\mathbf{q} \rightarrow y$ association, subject to updates of the underlying data. We require a model to meet the above-mentioned requirements.

2.1 Overview of `COUNT` predictive learning

Consider a set $\mathcal{Q} = \{(\mathbf{q}_i, y_i)\}_{i=1}^{n}$ of training pairs and a new query $\mathbf{q}$ with actual result y. Our major aim is to predict its result $\hat{y}$ using only $\mathcal{Q}$ without executing $\mathbf{q}$. Let us discuss some baseline solutions:

A first idea is to keep all pairs $(\mathbf{q}_i, y_i)$ and given $\mathbf{q}$ we find the most *similar* query $\mathbf{q}_j$ with respect to Euclidean distance and predict $\hat{y} = y_j$, with $(\mathbf{q}_j, y_j) \in \mathcal{Q}$. We can also involve the k closest queries to $\mathbf{q}$ and average their cardinality values, i.e., k-nearest neighbors regression, as will be further analyzed later. The major problems here are: (i) we must store and search all previous pairs for each new query; $\mathcal{Q}$ can be huge. Deciding which pairs to discard is not a trivial task (a new pair might convey useful information while another new one might be a redundant / repeated query); (ii) when data change (updates on raw data), which impacts the query results, it is not trivial to determine which pairs from $\mathcal{Q}$ and how many to update. Even worse, all pairs may need updating; (iii) when query patterns change (new user interests), then there may be many pairs in $\mathcal{Q}$ that will not contribute to cardinality prediction (the new queries are actually far distant to the previous ones) or even negatively impact the final result.

To avoid such problems we extract knowledge from $\mathcal{Q}$ as to how query and cardinality depend on each other. We could cluster similar queries given the Euclidean distance, thus forming a much smaller set $\mathcal{L}$ of representative (prototype) queries $\mathbf{w}$ with $|\mathcal{L}| \ll |\mathcal{Q}|$. For instance, $\mathbf{w} \in \mathcal{L}$ can be the centroid of those queries from $\mathcal{Q}_{\mathbf{w}} \subset \mathcal{Q}$ with distances from $\mathbf{w}$ be the smallest among all other representatives. However, we are not just interested in clustering $\mathcal{Q}$. We should partition $\mathcal{Q}$ aiming at cardinality prediction. An approach could be to assign to each $\mathbf{w}_i \in \mathcal{L}$ a 'representative' cardinality value, e.g., the average cardinality of those queries that belong to $\mathcal{Q}_{\mathbf{w}_i}$. Once this assignment is achieved, we only keep $\mathcal{L}$ and discard $\mathcal{Q}$.

Nonetheless, our requirements include incremental learning of the query space in light of cardinality prediction. We require an adaptive clustering algorithm that incrementally, i.e., with only one pass of $\mathcal{Q}$, quantizes $\mathcal{Q}$ but also with respect to minimizing the prediction error. Also, the adoption of an on-line quantization algorithm, like on-line k-means is not directly applicable in our case as we don't wish to simply quantize the query space; we explicitly require quantization of the query space *in light of cardinality prediction*. Moreover, on-line regression methods, e.g., incremental regression trees [17], on-line support vector regression [24], could not fulfill all requirements. This is because, we also deal with the fact that queries are continuously observed, conveying the way users are interested in data exploration. The capability of the model to adapt to such changes requires explicit information on accessing the very specific regions of the query patterns space; this is neither easily provided nor supported by incremental regression methods. Moreover, the problem here is not only to adapt to changes on the query patterns but to decide which and how representative(s) or regions of the query patterns space to update upon data and/or query updates.

2.2 Contribution & research outcome

We introduce a novel and scalable Machine Learning (ML) model $\mathcal{M}$ that incrementally extracts information about the $\mathbf{q} \rightarrow y$ association by learning $p(\mathbf{q})$ and, in parallel, $p(y|\mathbf{q})$. Once trained, model $\mathcal{M}$ predicts the cardinality of an *unseen* query without requesting its execution. The major technical contributions are:

- a prediction error-driven, associative local regression model for predicting the aggregate results of range queries.
- theoretical analysis of convergence of our machine learning algorithms over large-scale squared and absolute loss minimization.
- implementation of our algorithms in the Spark system.
- comprehensive experimental results analyzing the performance of our model and showcasing its benefits vis-à-vis the data-centric Spark's `COUNT` method for set-cardinality estimation.

3 Preliminaries & problem formulation

3.1 Preliminaries

We overview the essentials of our ML model, namely Unsupervised Competitive Learning (UCL) [21] and Heteroassociative Competitive Learning (HCL) [19].

3.1.1 Unsupervised competitive learning

UCL partitions a query pattern space $\mathbb{R}^{2d}$ characterized by an unknown $p(\mathbf{q})$, $\mathbf{q} \in \mathbb{R}^{2d}$. A prototype or *neuron* $\mathbf{w}_j$ represents a local region of $\mathbb{R}^{2d}$. UCL distributes M neurons $\mathbf{w}_1, \ldots, \mathbf{w}_M$ in $\mathbb{R}^{2d}$ to approximate $p(\mathbf{q})$. A UCL model learns as $\mathbf{w}_j$ changes in response to random training patterns. Competition selects which $\mathbf{w}_j$ the training pattern $\mathbf{q}$ modifies. Neuron $\mathbf{w}_j$ wins if it is the closest (based on 2-norm distance $\|\mathbf{q} - \mathbf{w}_j\|_2$) of the M neurons to $\mathbf{q}$. During the learning phase of UCL, patterns $\mathbf{q}$ are projected onto their winning neurons, which competitively and adaptively move around the space to form optimal partitions that minimize the quantity $\int \|\mathbf{q} - \mathbf{w}_j\|_2^2 p(\mathbf{q}) d\mathbf{q}$ with winning neuron $\mathbf{w}_j$: $\|\mathbf{w}_j - \mathbf{q}\|_2 = \min_i \|\mathbf{w}_i - \mathbf{q}\|_2$. The neurons upon a t-th training pattern $\mathbf{q}$ are incrementally updated as follows:

$$\Delta \mathbf{w}_j = \beta(t)\left(\mathbf{q} - \mathbf{w}_j\right) \text{ and } \Delta \mathbf{w}_i = \mathbf{0}, \text{ if } i \neq j, \tag{1}$$

where learning rate $\beta(t) \in (0, 1]$ slowly decreases with the update step.

3.1.2 Kohonen's self-organizing maps

Kohonen's self-organizing maps (SOM) [19] is an advanced variant of a UCL, in which $\mathbf{w}_j$ corresponds to the j-th position $\mathbf{r}_j = [r_{j1}, r_{j2}]$ of a 2-dim. square lattice/matrix $\mathcal{L}$ (we notate $\mathbf{w}_j \in \mathcal{L}$). In SOM, neurons that are topologically close in the lattice correspond to patterns that are also closein $\mathbb{R}^{2d}$. This way a *topographic* mapping is learned between query pattern and lattice space. This is achieved by adapting not only the winner neuron $\mathbf{w}_j$ of a pattern $\mathbf{q}$ but also its topographical neighbors $\mathbf{w}_i$ to some degree through a Kernel distance function $h(i, j; t)$ over the positions $\mathbf{r}_i$ and $\mathbf{r}_j$ of neurons $\mathbf{w}_i$ and $\mathbf{w}_j$ in $\mathcal{L}$, respectively. Usually, $h(i, j; t)$ is a Gaussian neighborhood function:

$$h(i, j; t) = \exp\left(-\frac{\| \mathbf{r}_i - \mathbf{r}_j \|_2^2}{2\rho^2(t)}\right). \tag{2}$$

Parameter $\rho(t)$ is the width of the neighborhood with initial value ρ_0 defined as $\rho(t) = \rho_0 \exp(-\frac{t}{T_\rho})$, where T_ρ is a constant. A small width value corresponds to narrow neighborhood. We obtain SOM through an incremental update rule that adapts all neurons that are topographically close to $\mathbf{w}_j$:

$$\Delta \mathbf{w}_i = \beta(t) h(i, j; t) \left(\mathbf{q} - \mathbf{w}_i\right), \forall i. \tag{3}$$

A good choice of $\beta(t)$ improves significantly the convergence of SOM [19]; usually $\beta(t) = \frac{\beta(t-1)}{1+\beta(t-1)}$ with $\beta(0) = 1$. SOM yields a high quality vector quantization from all UCL variants because of producing a structured *ordering* of the pattern vectors, i.e., similar query patterns are projected to similar neurons, making it ideal for our purposes. Figure 1 shows a SOM structure with neuron and position vectors before and after an update. UCL/SOM does not learn any conditional or joint association between different pattern spaces. In our case, we desire also to estimate an association between $\mathbb{R}^{2d}$ and $\mathbb{N}$, i.e., estimate $p(\mathbf{q}, y)$ with $\mathbf{q} \in \mathbb{R}^{2d}$, $y \in \mathbb{N}$, HCL comes into play.

3.1.3 Heteroassociative competitive learning

HCL estimates indirectly an unknown joint $p(\mathbf{q}, y)$, while directly estimates a function $f : \mathbb{R}^{2d} \rightarrow \mathbb{N}$ over random pairs $(\mathbf{q}, y)$. In statistical learning theory [21], HCL refers to a function estimation model $\mathcal{M}(f, \alpha)$ (or simply $\mathcal{M}$) with parameter $\alpha \in \Lambda$ (Λ is a parameter space defined later) for estimating f. The problem of learning $\mathcal{M}$ is that of

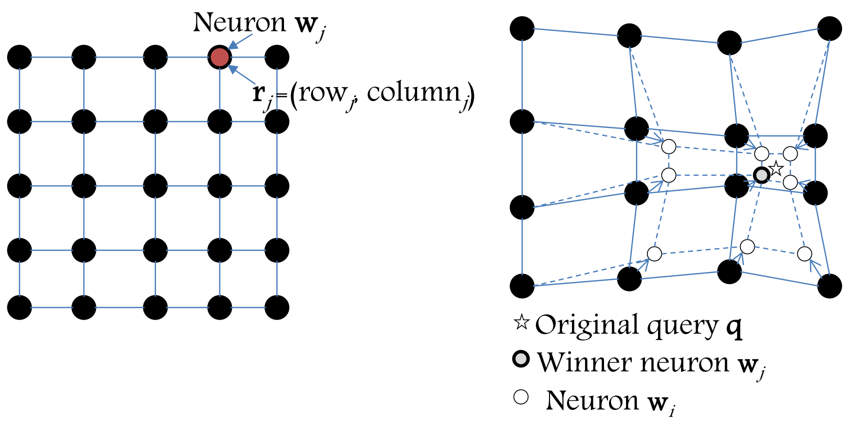

Neuron $\mathbf{w}_j$
$\mathbf{r}_j$=(row$_j$, column$_j$)
☆ Original query $\mathbf{q}$
Winner neuron $\mathbf{w}_j$
○ Neuron $\mathbf{w}_i$

Fig. 1 **a** A Self-organizing Map with neuron vectors $\mathbf{w}_j$ and position coordinates vector $\mathbf{r}_j$; **b** The adaptation of the self-organizing map after the projection of a query vector $\mathbf{q}$ to its closest neuron $\mathbf{w}_j$ on the neurons lattice $\mathcal{L}$

choosing from a set of functions $f(\mathbf{q}, \alpha)$, $\alpha \in \Lambda$, the one which minimizes the risk function:

$$\mathcal{J}(\alpha) = \int L(y, f(\mathbf{q}, \alpha)) \mathrm{d}p(\mathbf{q}, y), \tag{4}$$

given random pairs $(\mathbf{q}, y)$ drawn according to $p(\mathbf{q}, y) = p(\mathbf{q})p(y|\mathbf{q})$ with *loss* or estimation error $L(y, \hat{y})$ between actual y and predicted $\hat{y} = f(\mathbf{q}, \alpha)$, e.g., $L(y, \hat{y}) = |y - \hat{y}|$. The goal for HCL is to learn $\mathcal{M}(f, \alpha_0)$ which minimizes $\mathcal{J}(\alpha)$ subject to unknown $p(\mathbf{q}, y)$, i.e., $\alpha_0 = \arg\min_{\alpha \in \Lambda} \mathcal{J}(\alpha)$.

3.1.4 Stochastic gradient descent

Stochastic gradient descent (SGD) is considered to be one of the best methods for large scale loss minimization and has been experimentally and theoretically analyzed by [10]. Upon the presence of a t-th pattern $(\mathbf{q}, y)$, $\alpha(t)$ is updated by:

$$\Delta\alpha(t) = -\beta(t)\nabla L(y, \hat{y}; \alpha(t)), \tag{5}$$

where ∇L is the gradient of L at t-th pattern w.r.t. $\alpha(t)$.

3.2 Problem formulation

Consider a model $\mathcal{M}$ that estimates the **cardinality prediction function**

$$f : \mathbb{R}^{2d} \rightarrow \mathbb{N}$$

given a finite set $\mathcal{Q}$ of training pairs $(\mathbf{q}, y)$ drawn from the unknown $p(\mathbf{q}, y)$, i.e., $y = f(\mathbf{q})$. The model $\mathcal{M}$ learns the mapping from query pattern space to cardinality domain by minimizing the risk function $\mathcal{J}(\alpha)$ in (4) with respect to a **loss function** (prediction error) $L(y, \hat{y})$. A loss function can be, e.g., λ-insensitive $L(y, \hat{y}) = \max\{|y - \hat{y}| - \lambda, 0\}, \lambda > 0$, 0–1 loss $L(y, \hat{y}) = I(y \neq \hat{y})$ with I be the indicator function, squared loss $(y - \hat{y})^2$,or absolute $|y - \hat{y}|$.

The fundamental problem of the ML model for cardinality prediction is:

Problem 1 Given a dataset $\mathcal{B}$ and training pairs of queries and their answer-set cardinality values $(\mathbf{q}, y) \in \mathcal{Q}$, incrementally train a model $\mathcal{M}$ which minimizes $\mathcal{J}(\alpha)$.

4 Set cardinality predictive learning

4.1 Machine learning methodology

A natural, baseline solution for cardinality prediction is distance nearest-neighbors regression. This prediction scheme is based on utilizing the set cardinality values of *similar* historical queries to predict the set cardinality value for a new, unseen query. The notion of neighborhood is materialized by the *distance* (in some metric space, e.g., Euclidean space) of the unseen query $\mathbf{q}$ to a (stored) query $\mathbf{q}_i \in \mathcal{Q}$, whose cardinality value is y_i. Hence, the regression function for cardinality prediction $y = f(\mathbf{q}; k)$ refers to the average value of the cardinality values of the k-th closest stored queries $\mathbf{q}_i$:

$$y = f(\mathbf{q}; k) = \frac{1}{|\mathcal{N}_k(\mathbf{q})|} \sum_{i=1}^{|\mathcal{N}_k(\mathbf{q})|} y_i : \mathbf{q}_i \in \mathcal{N}_k(\mathbf{q}), \tag{6}$$

where the neighborhood $\mathcal{N}_k(\mathbf{q})$ is the set of the k-th closest queries to unseen query $\mathbf{q}$:

$$\mathcal{N}_k(\mathbf{q}) = \{\mathbf{q}_i \in \mathcal{T}, \mathbf{q}' \in \mathcal{T} \backslash \mathcal{N}_k(\mathbf{q}) : \|\mathbf{q}_i - \mathbf{q}\|_2 \leq \|\mathbf{q}' - \mathbf{q}\|_2\}. \tag{7}$$

In this k-nearest neighbors regression (k-nn), the cardinality of the neighborhood k plays a significant impact on the accuracy of prediction. The choice of k is very critical: (i) a small value of k means that noise will have a higher influence on the prediction result; (ii) a large value of k, evidently, yields a computationally expensive prediction result and defeats the basic philosophy behind, i.e., queries that are near might have similar densities in cardinality values; e.g., by involving in the final prediction result *irrelevant* and non-similar queries. In general notion, k is chosen to be $\sqrt{|\mathcal{Q}|}$, where $|\mathcal{Q}|$ is the number of stored queries in $\mathcal{Q}$, thus, interdependent of the query dimensionality $2d$. Moreover, a straightforward k-nn algorithm for cardinality prediction is $O(|\mathcal{Q}|d \log(k))$, which obviously, is not applicable for large-scale data-sets, especially when $k \sim \sqrt{|\mathcal{Q}|}$. This means that this (non-parametric) solution does not scale with the number of queries and dimensionality, thus, not suitable for scaling out for predictive analytics tasks like our problem.

We propose a solution, which **scales** with the number of queries and deals with the *curse of dimensionality* based on parametric regression, i.e., we attempt to *incrementally* extract knowledge from the $\mathcal{Q}$ set of historical queries and then, abstract a *parametric* model suitable to scale and, simultaneously, be computationally inexpensive for predictions. In this context, our scalable methodology learns from incoming queries and answers and dynamically builds a parametric model, thus (i) avoiding to maintain and process historical queries for making prediction and (ii) being capable to swiftly predict cardinality independent on the numbers of the queries.

Our objective is a scalable, parametric ML model $\mathcal{M}$ to:

1. incrementally quantize (cluster) the query pattern space, thus, abstracting the query space by certain M parameterized prototypes, with a user-specific fixed M;

2. learn the localities of the association $\mathbf{q} \rightarrow y$, thus, dealing with the curse of dimensionality [18] based on localized regression models;
3. predict the set cardinality given an unseen query in $O(d \log(M))$ independent of the number of queries $|\mathcal{Q}|$.

The novelty of our model relies on the introduction of two *simultaneous* incremental learning tasks:

– **Task 1**: incremental query space quantization (UCL/SOM; unsupervised learning);
– **Task 2**: incremental *local* learning of the $\mathbf{q} \rightarrow y$ association within the region of these neurons (HCL; supervised learning).

Both tasks rely on certain 2-dimensional *lattices*, where reside the parameters of the model. In Task 1, we abstract the lattice parameters as the query representatives (neurons). The parameters of the Task 2 refer to local output representatives (prototypes) depending on the representation of the prediction function, residing on a different lattice. In this work, we propose two variants for the cardinality prediction function f.

4.2 The lattice concept in machine learning methodology

4.2.1 Neuron input lattice

In this *input* lattice, hereinafter referred to as the neuron lattice $\mathcal{L}$, we estimate the parameters, i.e., SOM neurons, that represent the input space in our problem, i.e., the query patterns. The $2d$-dimensional neurons $\mathbf{w}_i \in \mathcal{L}$ quantize the query space into a fixed number of M query sub-spaces. As will be elaborated later, this lattice is used for projecting an unseen query $\mathbf{q}$ onto a query sub-space and then leading to its associated output lattice for cardinality prediction.

4.2.2 Cardinality output lattice

In this *output* lattice, we estimate the (local) cardinality prototypes y_j, which are associated with each $\mathbf{w}_j$. The y_j reside on a *cardinality* lattice $\mathcal{C}$ such that the j-th index of $\mathbf{w}_j$ refers to the j-th index of y_j. Hence, a point y_j in the cardinality lattice corresponds to a **local associative constant function**:

$$f_j(\mathbf{q}) = y_j, \mathbf{q} \in \mathbb{R}^{2d} : j = \arg\min_{i \in [M]} \|\mathbf{q} - \mathbf{w}_i\|_2. \tag{8}$$

In the case of input lattice $\mathcal{L}$ and output lattice $\mathcal{C}$, the parameter set for model $\mathcal{M}$ is $\alpha = (\{\mathbf{w}_j\}, \{y_j\}), j = 1, \ldots, M$. Figure 2 shows the idea of the cardinality lattice.

4.2.3 Coefficient output lattice

In this *output* lattice, if the local associative function is varying considerably around a point, a piece-wise constant approximation may require many units. In this case, we refer to the estimation of the local linear regression coefficients $\mathbf{m}_j = [m_{j0}, m_{j1}, \ldots, m_{j2d}] \in \mathbb{R}^{2d+1}$, which are associated with each query prototype $\mathbf{w}_j$. That is the cardinality y is approximated by a linear combination of the query dimensions $\mathbf{q} = [q_1, \ldots, q_{2d}]$, while $\mathbf{m}$ is the $(2d + 1)$-dimensional vector of the linear coefficients, with m_0 being the intercept in the $\mathbb{R}^{2d} \times \mathbb{N}$ space. The $\mathbf{m}_j$ coefficients reside on a *coefficient* lattice $\mathcal{O}$ such that the j-th index of $\mathbf{w}_j$ refers to the j-th regression plane governed by the regression $\mathbf{m}_j\mathbf{q}^\top$. This defines a local regression plane over the query and cardinality space, defined by those queries that are projected on the query prototype $\mathbf{w}_j$. Hence, a point $\mathbf{m}_j$ in the coefficient lattice corresponds to the parameter of the **local linear regression function**:

$$f_j(\mathbf{q}) = \mathbf{m}_j\mathbf{q}^\top, \mathbf{q} \in \mathbb{R}^{2d} : j = \arg\min_{i \in [M]} \|\mathbf{q} - \mathbf{w}_i\|_2. \tag{9}$$

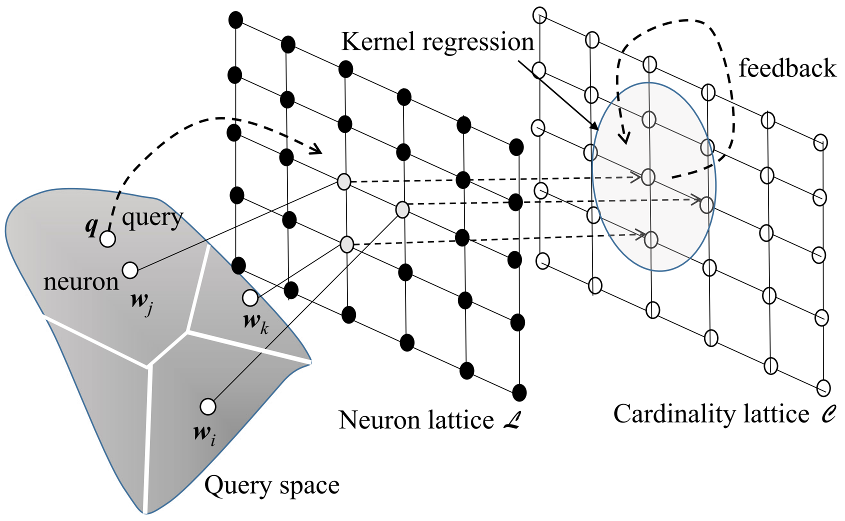


Fig. 2 Cardinality lattice-based prediction: Projection-association-local prediction: Simultaneous UCL and HCL over lattices $\mathcal{L}$ and $\mathcal{C}$

Figure 3 shows the idea of the coefficient lattice. In the case of input lattice $\mathcal{L}$ and output lattice $\mathcal{O}$, the parameter set for model $\mathcal{M}$ is $\alpha = (\{\mathbf{w}_j\}, \{\mathbf{m}_j\})$, $j = 1, \ldots, M$.

4.3 Learning methodology

4.3.1 Overview

Consider the presence of a (random) training pair $(\mathbf{q}, y)$. The following steps demonstrate the methodology of exploiting such training pair for estimating the points on the: neuron, cardinality and regression lattices.

Projection The query $\mathbf{q}$ from the training pair $(\mathbf{q}, y)$ is projected onto its (winner) closest neuron $\mathbf{w}_j \in \mathcal{L}$ from the neuron lattice. Certain neurons, including the winner $\mathbf{w}_j$, are then adapted to this occurrence. In this step, we have to define the update rule $\Delta\mathbf{w}_i$ for the neurons in the neuron lattice.

Association Simultaneously, the actual cardinality y from the training pair $(\mathbf{q}, y)$ is utilized to update certain points from the cardinality and regression lattices. Specifically, the corresponding prototype $y_j \in \mathcal{C}$, i.e., this is associated with the winner query neuron $\mathbf{w}_j$, and the corresponding regression coefficient $\mathbf{m}_j \in \mathcal{O}$ are updated based on y and the query $\mathbf{q}$ (in the latter case) governed by feedback update rules. Such rules derive from the stochastic negative partial derivative (introduced later).

Prediction The model $\mathcal{M}$ after locating the winner neuron $\mathbf{w}_j$ based on the input lattice, predicts the cardinality $\hat{y}$ using Kernel regression over (i) the local associative functions in the $\mathcal{C}$ lattice, and (ii) the local linear regression functions in the $\mathcal{O}$ lattice.

Feedback The prediction result $\hat{y}$ feeds the $\mathcal{C}$ and $\mathcal{O}$ lattices for updating the cardinality prototypes and the regression coefficients, respectively.

4.4 The predictive learning algorithm

We adopt SOM for UCL since based on topology preservation we can claim that: *if queries* $\mathbf{q}$ *and* $\mathbf{q}'$ *are similar due to being projected onto the same neuron* $\mathbf{w}_j$ *of* $\mathcal{L}$*, then their images through the local associative and local regression functions* $f_j(\mathbf{q})$ *and* $f_j(\mathbf{q}')$ *on cardinality lattice* $\mathcal{C}$ *and coefficient lattice* $\mathcal{O}$*, respectively, are likely to be similar, too.* This argument cannot be claimed by any other UCL method (e.g., k-means or fuzzy c-means clustering), which does not guarantee topological ordering of quantization vectors.

At this point, we can define the cardinality prediction function $\hat{y} = f(\mathbf{q}, \alpha)$ based on the local associative and regression functions, and in the sequel, report on the loss function $L(y, \hat{y})$. Consider two range queries $\mathbf{q}, \mathbf{q}'$ normalized firstly in $[0, 1]^{2d}$ (only for simplicity in our analysis). Let the winner neuron $\mathbf{w}_j \in \mathcal{L}$ and its corresponding (i) local associative function $f_j(\mathbf{q})$, i.e., cardinality prototype $y_j \in \mathcal{C}$ and (ii) local linear regression function $f_j(\mathbf{q})$, i.e., regression coefficient $\mathbf{m}_j \in \mathcal{O}$ to a random query $\mathbf{q}$.

The cardinality prediction f is not only based on $f_j(\mathbf{q})$, but also on the contribution of the neighboring $f_i(\mathbf{q})$ defined by the topographical neighborhood of winner $\mathbf{w}_j$. This is achieved by a kernel function $\mathcal{K}_\epsilon(\|\mathbf{r}_i - \mathbf{r}_j\|_2)$ over the normalized location vectors $\mathbf{r}_i$ and $\mathbf{r}_j$ (i.e., $\|\mathbf{r}_i\|, \|\mathbf{r}_j\| \leq 1$) of the associated neurons $\mathbf{w}_i$ and $\mathbf{w}_j$ in the input lattice $\mathcal{L}$, respectively. That is, $\hat{y} = f(\mathbf{q}, \alpha)$ is produced by the (Nadaraya-Watson) Kernel regression model:

$$\hat{y} = f(\mathbf{q}, \alpha) = \frac{\sum_{i=1}^{M} \mathcal{K}_\epsilon(\|\mathbf{r}_i - \mathbf{r}_j\|) f_i(\mathbf{q})}{\sum_{i=1}^{M} \mathcal{K}_\epsilon(\|\mathbf{r}_i - \mathbf{r}_j\|)} \tag{10}$$

with $j = \arg\min_{\mathbf{w}_i \in \mathcal{L}} \|\mathbf{q} - \mathbf{w}_i\|_2$. In this paper, we utilize the kernel $\mathcal{K}_\epsilon(x) = 0.75 \cdot \left(1 - (\frac{x-0.5}{\epsilon})^2\right) \cdot I(|x - \frac{1}{2}| \leq \epsilon)$, which is the Epanechnikov kernel function shifted to 0.5 and scaled by $0 < \epsilon \ll 0.5$. Obviously, any other kernel functions can be also adopted e.g., uniform, triangular, quadratic, with Epanechnikov being most commonly used

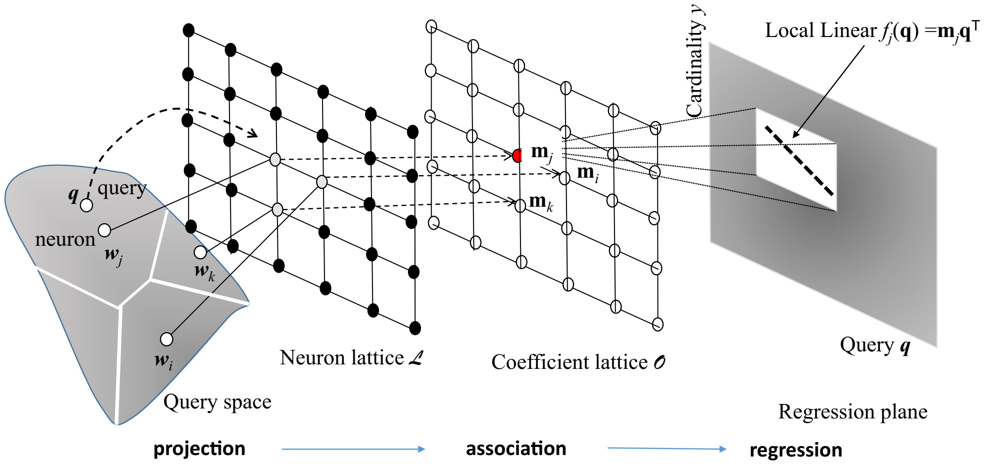


Fig. 3 Coefficient lattice-based prediction: Projection-association-linear regression: Simultaneous UCL and HCL over lattices $\mathcal{L}$ and $\mathcal{O}$

kernel for regression. Topographically close neurons w.r.t. location vectors also imply close neurons w.r.t. Euclidean distance. However, the adoption of a Kernel function over the distance of neurons in $\mathbb{R}^{2d}$ could assume query components to be isotropically Gaussian, which is not a general case when d is relatively large. The predicted cardinality $\hat{y}$ is estimated by a kernel smoothing of those cardinality prototypes and linear regression coefficients, whose associated neurons are topographically close (w.r.t. ϵ) to the winner neuron.

Given actual y and predicted $\hat{y}$ in (10), we then adopt the loss functions:

$$L_1(y, \hat{y}) = |y - \hat{y}| \text{ and } L_2(y, \hat{y}) = (y - \hat{y})^2 \tag{11}$$

since there are widely used for evaluating the prediction error in cardinality prediction as in [11, 15, 27].

We can now provide the (on-line) learning phase of the model $\mathcal{M}$ given a sequence of pattern (training) pairs $(\mathbf{q}(1), y(1)), (\mathbf{q}(2), y(2)), \ldots$ Query patterns $\mathbf{q}(t)$ are used for quantizing the query space (over $\mathcal{L}$) and cardinalities $y(t)$ are used for learning the $\mathbf{q} \rightarrow y$ association (over $\mathcal{C}$ and $\mathcal{O}$). Upon the presence of a pattern pair $(\mathbf{q}(t), y(t))$ the winner $\mathbf{w}_j(t) \in \mathcal{L}$ is determined by

$$j = \arg\min_{\mathbf{w}_i \in \mathcal{L}} \|\mathbf{q}(t) - \mathbf{w}_i(t)\|. \tag{12}$$

After the projection of $\mathbf{q}$ to winner $\mathbf{w}_j$, the model $\mathcal{M}$ updates in an incremental manner the winner and all its neighbors of lattice $\mathcal{L}$ such that they approach the query pattern $\mathbf{q}$ with a magnitude of $\beta(t)h(i, j; t)$. In the same time, the actual cardinality y is used for updating the corresponding: (i) $y_j \in \mathcal{C}$ along with all prototypes $y_i \in \mathcal{C}$ and (ii) $\mathbf{m}_j \in \mathcal{O}$ along with all coefficients $\mathbf{m}_i \in \mathcal{O}$ associated with the neighbors of winner neuron $\mathbf{w}_j$. Notably, the update rules for each y_i and $\mathbf{m}_i$ are governed by the loss function $L(y, \hat{y})$ we aim to minimize, having $\hat{y}$ defined in (10).

In the case of the neuron-cardinality lattices for cardinality prediction, the model $\mathcal{M}$ estimates the parameter $\alpha = (\{\mathbf{w}_i\}, \{y_i\})_{i=1}^{M}$ by minimizing the objective function $\mathcal{J}_1$ in (13)

$$\begin{aligned}\mathcal{J}_1(\{\mathbf{w}_i\}, \{y_i\}) = & \frac{1}{2}\int_{\mathcal{W}} \sum_{\mathbf{w}_i \in \mathcal{L}} h(i, j)\|\mathbf{w}_i - \mathbf{q}\|_2^2 \mathrm{d}p(\mathcal{W}) \\ & + \int_{\mathcal{Y}} \sum_{y_i \in \mathcal{C}} h(i, j)|y - \hat{y}| \mathrm{d}p(\mathcal{Y})\end{aligned} \tag{13}$$

being taken over an infinite sequence of $\mathcal{W} = \{\mathbf{q}(1), \mathbf{q}(2), \ldots\}$ and corresponding $\mathcal{Y} = \{y(1), y(2), \ldots\}$ and $p(\mathcal{W})$, $p(\mathcal{Y})$ is the pdf of $\mathcal{W}$ and $\mathcal{Y}$, respectively, with

$$\hat{y} = \frac{\sum_{i=1}^{M} \mathcal{K}_\epsilon(\|\mathbf{r}_i - \mathbf{r}_j\|) y_i}{\sum_{i=1}^{M} \mathcal{K}_\epsilon(\|\mathbf{r}_i - \mathbf{r}_j\|)} \tag{14}$$

and $j = \arg\min_{\mathbf{w}_i \in \mathcal{L}} \|\mathbf{q} - \mathbf{w}_i\|_2$. The factor $\frac{1}{2}$ is for mathematical convenience. Here, we utilize the L_1 in (11) loss function, since the cardinality prototypes are local scalar constant values within each query sub-space.

In the case of the neuron-regression lattices for cardinality prediction, the model $\mathcal{M}$ estimates the parameter $\alpha = (\{\mathbf{w}_i\}, \{\mathbf{m}_i\})_{i=1}^{M}$ by minimizing the objective function $\mathcal{J}_2$ in (15)

$$\begin{aligned}\mathcal{J}_2(\{\mathbf{w}_i\}, \{\mathbf{m}_i\}) = & \frac{1}{2}\int_{\mathcal{W}} \sum_{\mathbf{w}_i \in \mathcal{L}} h(i, j)\|\mathbf{w}_i - \mathbf{q}\|_2^2 \mathrm{d}p(\mathcal{W}) \\ & + \frac{1}{2}\int_{\mathcal{U}} \sum_{\mathbf{m}_i \in \mathcal{O}} h(i, j)(y - \hat{y})^2 \mathrm{d}p(\mathcal{U})\end{aligned} \tag{15}$$

where $p(\mathcal{U})$ is the pdf of $\mathbf{m}$, with $j = \arg\min_{\mathbf{w}_i \in \mathcal{L}} \|\mathbf{q} - \mathbf{w}_i\|_2$ and

$$\hat{y} = \frac{\sum_{i=1}^{M} \mathcal{K}_\epsilon(\|\mathbf{r}_i - \mathbf{r}_j\|) \mathbf{m}_i^\top \mathbf{q}}{\sum_{i=1}^{M} \mathcal{K}_\epsilon(\|\mathbf{r}_i - \mathbf{r}_j\|)}. \tag{16}$$

Here, we utilize the L_2 in (11) loss function, since we estimate the local linear regression coefficients within each query sub-space based on the ordinary least squares method.

Theorem 1 *Given a training pair $(\mathbf{q}(t), y(t))$, the model $\mathcal{M}$ converges to the optimal parameter α, which minimizes the risk function $\mathcal{J}_1(\alpha)$ in (13) with respect to loss function $L_1(y, \hat{y}) = |y - \hat{y}|$ and $\hat{y}$ is defined in (14), if neuron $\mathbf{w}_i(t) \in \mathcal{L}$ and its associated prototype $y_i(t) \in \mathcal{C}$ are updated as:*

$$\Delta \mathbf{w}_i(t) = \beta(t) h(i, j; t) \left(\mathbf{q}(t) - \mathbf{w}_i(t)\right) \tag{17}$$

$$\begin{aligned}\Delta y_i(t) = & \beta(t) \sum_{k=1}^{M} h(k, j; t) \frac{\mathcal{K}_\epsilon(\|\mathbf{r}_i - \mathbf{r}_j\|)}{\sum_{k=1}^{M} \mathcal{K}_\epsilon(\|\mathbf{r}_k - \mathbf{r}_j\|)} \\ & \times sgn\left(y(t) - \hat{y}(t)\right)\end{aligned} \tag{18}$$

where $sgn(\cdot)$ is the signum function, $\beta(t)$ is the learning rate and $h(i, j; t)$ is the neighborhood function, j is the index of the winner neuron $\mathbf{w}_j(t)$ of pattern query $\mathbf{q}(t)$ and predicted $\hat{y}(t)$ is determined by (14).

The proof of Theorem 1 is provided in [5]; we present it here for self-contained reasons.

Proof We derive the analysis of convergence corresponding to lattices $\mathcal{L}$ and $\mathcal{C}$. We verify whether the quantization error $\|\mathbf{w} - \mathbf{q}\|_2^2$ and loss $L_1(y, \hat{y}) = |y - \hat{y}|$ actually decreases as the learning phase proceeds, converging eventually to a stable state.

The convergence is evaluated through the average expected loss $\mathcal{J}_1$ in (13) being taken over an infinite sequence of $\mathcal{W} = \{\mathbf{q}(1), \mathbf{q}(2), \ldots\}$ and corresponding $\mathcal{Y} = \{y(1), y(2), \ldots\}$ and $p(\mathcal{W})$, $p(\mathcal{Y})$ is the pdf of $\mathcal{W}$ and $\mathcal{Y}$, respectively. Since both pdfs are unknown and sequences

$\mathcal{Y}$ and $\mathcal{W}$ are actually finite we use the Robbins-Monro (RM) stochastic approximation for $\mathcal{J}_1$ minimization to find an optimal value for each $\mathbf{w}_i$, $y_i, i = 1, \ldots, M$. Based on RM the stochastic sample $J_1(t)$ of $\mathcal{J}_1$ is $J_1(t) = \frac{1}{2}\sum_{\mathbf{w}_i \in \mathcal{L}} h(i, j; t)\|\mathbf{w}_i(t) - \mathbf{q}(t)\|_2^2 + \sum_{y_i \in \mathcal{C}} h(i, j; t)|y(t) - \hat{y}(t)|$. The $J_1(t)$ has to decrease at each new pattern at t by descending in the direction of its (partial) negative gradient. Hence, the SGD rule for each $\mathbf{w}_i$ is $\Delta\mathbf{w}_i(t) = -\frac{1}{2}\beta(t)\frac{\partial J_1(t)}{\partial \mathbf{w}_i(t)}$ and for y_i is $\Delta y_i(t) = -\beta(t)\frac{\partial J_1(t)}{\partial y_i(t)}$, where $\beta(t)$ satisfies $\sum_{t=0}^{\infty}\beta(t) = \infty$ and $\sum_{t=0}^{\infty}\beta^2(t) < \infty$ [21]. From the partial derivatives of $J_1(t)$ we obtain the update rules (17) and (18) for parameter set α. □

Remark 1 Note that the update rule (18) for prototypes $y_i(t)$ involves the current prediction $\hat{y}(t)$ of the model during the t-th training pair in the learning phase. Naturally we update each $y_i(t)$ in an on-line supervised regression fashion, in which we take the prediction $\hat{y}(t)$ in (14) as feedback. From (18) we observe that neighbor $y_i(t)$ of $y_j(t)$ is adapted by its relative contribution provided by the kernel function, which is rational since $y_i(t)$ contributes with the same magnitude to the cardinality prediction. If $y(t) > \hat{y}(t)$, then $y_i(t)$ increases linearly with its contribution to prediction approaching the actual $y(t)$. On the other hand, i.e., $y(t) < \hat{y}(t)$, $y_i(t)$ decreases to move away from $\hat{y}(t)$ and approaches $y(t)$. When the current prediction error is zero, i.e., $L(y(t), \hat{y}(t)) = |y(t) - \hat{y}(t)| = 0$, there is no update on the cardinality prototypes. Neuron $\mathbf{w}_i(t)$ moves toward pattern query $\mathbf{q}(t)$ to follow the trend. Obviously, the more similar a pattern query $\mathbf{q}$ and a neuron $\mathbf{w}_i$ are, the less $\mathbf{w}_i$ gets updated.

Theorem 2 refers to the convergence of a neuron $\mathbf{w}_i$ to the local expectation query representative, i.e., *centroid* $\mathbb{E}[\mathbf{q}|\mathbb{Q}_i]$ in the input sub-space $\mathbb{Q}_i$.

Theorem 2 *If $\mathbb{E}[\mathbf{q}|\mathbb{Q}_i]$ is the local expectation query of the subspace $\mathbb{Q}_i$ and prototype $\mathbf{w}_i$ is the subspace representative, $P(\mathbf{w}_i = \mathbb{E}[\mathbf{q}|\mathbb{Q}_i]) = 1$ at equilibrium.*

Proof The update rule for a neuron $\mathbf{w}_i$ based on Theorem 1 is $\Delta\mathbf{w}_i \propto (\mathbf{q} - \mathbf{w}_i)$. Let the i-th neuron $\mathbf{w}_i$ reach equilibrium: $\Delta\mathbf{w}_i = \mathbf{0}$, which holds with probability 1. By taking the expectation of both sides we obtain

$$\mathbf{0} = \mathbb{E}[\Delta\mathbf{w}_i] = \mathbb{E}[(\mathbf{q} - \mathbf{w}_i)] = \int_{\mathbb{Q}_i}(\mathbf{q} - \mathbf{w}_i)p(\mathbf{q})\mathrm{d}\mathbf{q}$$
$$= \int_{\mathbb{Q}_i}\mathbf{q}p(\mathbf{q})\mathrm{d}\mathbf{q} - \mathbf{w}_i\int_{\mathbb{Q}_i}p(\mathbf{q})\mathrm{d}\mathbf{q}.$$

This indicates that $\mathbf{w}_i$ is constant with probability 1, and then by solving $\mathbb{E}[\Delta\mathbf{w}_i] = \mathbf{0}$, the $\mathbf{w}_i$ equals the *centroid* $\mathbb{E}[\mathbf{q}|\mathbb{Q}_i]$. □

If ϵ is selected such that $\mathcal{K}_\epsilon(\|\mathbf{r}_i - \mathbf{r}_j\|) = 0, i \neq j$, then we obtain $\Delta y_j \sim \mathrm{sgn}(y - y_j)$ in which only y_j of the winner $\mathbf{w}_j$ is updated, given that there is no significant impact from other neighboring neurons after convergence, i.e., $\sum_{k=1}^{M} h(k, j) \overset{t\to\infty}{=} h(j, j) \cong 1$. We then provide the following theorem:

Theorem 3 *If $\tilde{y}_j$ is the median of the partition $\mathbb{Y}_j$ corresponding to the image of query sub-space $\mathbb{Q}_j$ of winner $\mathbf{w}_j$ then $P(y_j = \tilde{y}_j) = 1$ at equilibrium.*

The proof of Theorem 3 is provided in [5]; we present it here for self-contained reasons.

Proof Let y_j correspond to $\mathbf{w}_j$ and assume the image of $\mathbb{Q}_j \subset \mathbb{R}^{2d}$ to subspace $\mathbb{Y}_j \subset \mathbb{N}$ via the $y = f(\mathbf{q})$. The median $\tilde{y}_j$ of $\mathbb{Y}_j$ satisfies $P(y \geq \tilde{y}_j) = P(y \leq \tilde{y}_j) = \frac{1}{2}$. Suppose that y_j has reached equilibrium, i.e., $\Delta y_j = 0$, which holds with probability 1. By taking the expectations of both sides and replacing Δy_j with the update rule $\mathrm{sgn}(y - y_j)$:

$$E[\Delta y_j] = \int_{\mathbb{Y}_j}\mathrm{sgn}(y - y_j)p(y)\mathrm{d}y$$
$$= P(y \geq y_j)\int_{\mathbb{Y}_j}p(y)\mathrm{d}y - P(y < y_j)\int_{\mathbb{Y}_j}p(y)\mathrm{d}y$$
$$= 2P(y \geq y_j) - 1.$$

Since $\Delta y_j = 0$ thus y_j is constant, then $P(y \geq y_j) = \frac{1}{2}$, which denotes that y_j converges to the median of $\mathbb{Y}_j$. □

Theorem 4 *Given a training pair $(\mathbf{q}(t), y(t))$, the model $\mathcal{M}$ converges to the optimal parameter α, which minimizes the risk function $\mathcal{J}_2(\alpha)$ in (15) with respect to loss function $L_2(y, \hat{y}) = (y - \hat{y})^2$ and $\hat{y}$ is defined in (16), if neuron $\mathbf{w}_i(t) \in \mathcal{L}$ and its associated linear regression coefficients $\mathbf{m}_i(t) \in \mathcal{O}$ are updated as:*

$$\Delta\mathbf{w}_i(t) = \beta(t)h(i, j; t)\left(\mathbf{q}(t) - \mathbf{w}_i(t)\right) \tag{19}$$

$$\Delta\mathbf{m}_i(t) = \beta(t)\sum_{k=1}^{M}h(k, j; t)\frac{\mathcal{K}_\epsilon(\|\mathbf{r}_i - \mathbf{r}_j\|)}{\sum_{k=1}^{M}\mathcal{K}_\epsilon(\|\mathbf{r}_k - \mathbf{r}_j\|)} \times \left(y(t) - \hat{y}(t)\right)[1; \mathbf{q}(t)]^\top \tag{20}$$

where $\beta(t)$ is the learning rate and $h(i, j; t)$ is the neighborhood function, j is the index of the winner neuron $\mathbf{w}_j(t)$ of pattern query $\mathbf{q}(t)$ and predicted $\hat{y}(t)$ is determined by (16).

Proof As in the proof of the Theorem 1, the convergence is evaluated through the average expected loss $\mathcal{J}_2$ in (13) being taken over an infinite sequence of $\mathcal{W} = \{\mathbf{q}(1), \mathbf{q}(2), \ldots\}$ and corresponding $\mathcal{Y} = \{y(1), y(2), \ldots\}$ and $p(\mathcal{W})$. We rest on RM stochastic approximation for $\mathcal{J}_2$ minimization to find an optimal value for each $\mathbf{w}_i$, $\mathbf{m}_i, i = 1, \ldots, M$. The stochastic sample $J_2(t)$ of $\mathcal{J}_2$ is $J_2(t) = \frac{1}{2}\sum_{\mathbf{w}_i \in \mathcal{L}} h(i,j;t)\|\mathbf{w}_i(t) - \mathbf{q}(t)\|_2^2 + \frac{1}{2}\sum_{\mathbf{m}_i \in \mathcal{O}} h(i,j;t)(y(t) - \hat{y}(t))^2$. Hence, the SGD rule for each $\mathbf{w}_i$ is $\Delta\mathbf{w}_i(t) = -\frac{1}{2}\beta(t)\frac{\partial J_2(t)}{\partial \mathbf{w}_i(t)}$ and for $\mathbf{m}_i$ is $\Delta\mathbf{m}_i(t) = -\frac{1}{2}\beta(t)\frac{\partial J_2(t)}{\partial \mathbf{m}_i(t)}$, where $\beta(t)$ satisfies $\sum_{t=0}^{\infty}\beta(t) = \infty$ and $\sum_{t=0}^{\infty}\beta^2(t) < \infty$ [21]. From the partial derivatives of $J_2(t)$ we obtain the update rules (19) and (20) for parameter set α. □

Remark 2 As seen in (20), when determining the positions of regression coefficients, supervised (prediction) error is not only taken into account, but also the input $\mathbf{q}$ and the impact of *all* neurons (reflected by their neighborhood functions $h(k;j)$ are taken into consideration. Through this coupled training of the regression coefficients and neurons positions, query and regression representatives are placed in the input and output space, respectively, in such a way so as to minimize the loss function L_2.

Remark 3 Let us assume again that an ϵ is selected such that $\mathcal{K}_\epsilon(\|\mathbf{r}_i - \mathbf{r}_j\|) = 0, i \neq j$. Given that both neurons and regression coefficients converge from Theorem 4, then, we obtain the update rule: $\Delta\mathbf{m}_j \sim (y - \mathbf{m}_j^\top\mathbf{q})\mathbf{q}$, given that there is no significant impact from other neighboring neurons after convergence, i.e., $\sum_{k=1}^{M} h(k,j) \overset{t\to\infty}{=} h(j,j) \cong 1$; here, for mathematical convenience, we absorbed the 'intercept' constant of the local regression plane by adding a constant dimension of one to $\mathbf{q}$. Evidently, this corresponds to the stochastic update rule for the multivariate linear regression utilizing the ordinary least squares method.

The learning phase of model $\mathcal{M}$ is described in Algorithm 1. The input is the training set of pairs $\mathcal{Q} = \{(\mathbf{q}, y)\}$, 2-dim. lattices $\mathcal{L}$ and $\mathcal{C}$ (or $\mathcal{O}$) with M entries, and a stopping threshold $\theta > 0$. The algorithm processes successive random pattern pairs until a termination criterion $T_t \leq \theta$. T_t is the 1-norm between successive estimates of neurons and cardinality prototypes:

$$T_t = \sum_{i=1}^{M}\left(\|\mathbf{w}_i(t) - \mathbf{w}_i(t-1)\|_1 + |y_i(t) - y_i(t-1)|\right), \tag{21}$$

or regression prototypes,

$$T_t = \sum_{i=1}^{M}\left(\|\mathbf{w}_i(t) - \mathbf{w}_i(t-1)\|_1 + \|\mathbf{m}_i(t) - \mathbf{m}_i(t-1)\|_1\right), \tag{22}$$

with $\|\mathbf{w}_i\|_1 = \sum_{k=1}^{2d}|w_{ik}|$ and $\|\mathbf{m}_i\|_1 = \sum_{k=1}^{2d+1}|m_{ik}|$. The output is parameter set α.

Algorithm 1 The cardinality learning algorithm

```
Input: training set $\mathcal{Q}$, neuron lattice $\mathcal{L}$, cardinality
       lattice $\mathcal{C}$, and regression lattice $\mathcal{O}$ with $M$
       entries, stopping threshold $\theta$
Output: parameter set $\alpha$
Initialize $(\mathbf{w}_i(0), y_i(0)), i = 1, \ldots, M$;
/*neuron-cardinality lattice variant*/;
Initialize $(\mathbf{w}_i(0), \mathbf{m}_i(0)), i = 1, \ldots, M$;
/*neuron-regression lattice variant/*;
$t \leftarrow 0$;
repeat
    $t \leftarrow t + 1$;
    Get the next training pair $(\mathbf{q}(t), y(t)) \in \mathcal{Q}$;
    $j = \arg\min_{\mathbf{w}_i \in \mathcal{L}}\|\mathbf{q}(t) - \mathbf{w}_i(t)\|_2$ /*project*/;
    Update neurons $\mathbf{w}_i(t), \forall i$ /*quantization
    w.r.t. in (17) and (19)*/;
    Predict $\hat{y}(t)$ /*prediction feedback
    w.r.t. in (14) and (16)*/;
    Update cardinality prototypes $y_i(t), \forall i$
    /*adaptation w.r.t. in (18)*/;
    Update regression coefficients $\mathbf{m}_i(t), \forall i$
    /*adaptation w.r.t. in (20)*/;
until $T_t \leq \theta$;
```

4.5 Set cardinality prediction

Once the parameter set α is trained (for both output lattice variants), and thus no more updates are realized on neurons, cardinality prototypes and local regression coefficients, we predict the cardinality $\hat{y}$ given a random query $\mathbf{q}$ as defined in (14) and (16). That is, we proceed with answer set cardinality estimation without executing the incoming query $\mathbf{q}$.

Firstly, the query $\mathbf{q}$ is projected onto the neuron lattice $\mathcal{L}$ and its winner $\mathbf{w}_j$ is obtained. In the case of the cardinality lattice $\mathcal{C}$, the corresponding cardinality prototype y_j is the associated constant of the query sub-space $\mathbb{Q}_j$. In the case of the regression lattice $\mathcal{O}$, the local regression coefficient $\mathbf{m}_j$ is obtained. The predicted COUNT value is $\hat{y}$ calculated by the Kernel regression over the region around the images $f_i(\mathbf{q}) = y_i$ in lattice $\mathcal{C}$ and $f_i(\mathbf{q}) = \mathbf{m}_i^\top\mathbf{q}$ in lattice $\mathcal{O}$, respectively, such that $\mathcal{K}_\epsilon(\|\mathbf{r}_i - \mathbf{r}_j\|) > 0$, for $i = 1, \ldots, M$.

4.6 Computational complexity

During the learning phase of the model $\mathcal{M}$, we require to (i) find the closest (winner) neuron over the neuron lattice $\mathcal{L}$ and then (ii) update all M prototypes in both input and output lattices based on the neighborhood weight $h(i,j), \forall i$. This requires $O(dM)$ space and $O(dM)$ for the updates. Since prototypes are updated during learning, the learning phase requires $O(d/\theta)$ [10] iterations to get $T_t \leq \theta$. After learning, we obtain cardinality prediction in $O(d \log M)$ by applying an one-nearest neighbor search for the winner using a $2d$-dim. tree structure over the neurons in $\mathcal{L}$. After locating the winner, then we just retrieve those neighboring neurons (constant number) which are determined by the Kernel neighboring function $\mathcal{K}_\epsilon$. In the case of updates, adaptation given a pair requires also $O(d \log M)$ time for searching for the winner. Hence, our proposed parametric model, after training, can provide prediction in $O(d \log M)$, which is independent of the size of the data $|\mathcal{B}|$ and the training set $|\mathcal{Q}|$, thus, being capable for scaling out predictive analytics tasks.

5 Implementation in Spark

We have implemented our model in the Spark system [32]. The reason behind this implementation is to explore how such models can be incorporated into Big Data Engines. In addition, we examine how much faster and how close our cardinality estimations are, compared with the result obtained from the built-in `COUNT` method provided by these engines. This section covers the basic concepts behind Spark [32] (currently a popular Big Data engine) and an overview of how we developed and incorporated our machine learning model into Spark.

5.1 Overview of Spark

The Resilient Distributed Datasets (RDDs) lie in the foundation of Spark. RDDs are fault-tolerant distributed data structures that allow users to save intermediate results in main memory. This means that, RDDs can be easily recovered once something goes wrong and that they can be easily distributed in a cluster environment to improve efficiency. Their recovery is relied on the *lineage graph* produced by Spark. A lineage graph is a Directed-Acyclic Graph that is used to record all of the changes made to a dataset. Hence, once something goes wrong it can be easily re-computed using the steps recorded. In addition, through this functionality, the users can control the number of partitions to optimize the data placement and, also, offer a rich set of operations [32]. The set of operations can be divided into *transformations* and *actions* which are described as follows:

5.1.1 Transformations in Spark

The RDDs are created by loading data files from permanent storage or by using transformations on loaded data. These transformations can change the loaded data through operations such as *filter* and *map*. A comprehensive list of the available transformations can be found at Spark's website.[2] It is worth noting that transformations are not applied immediately. Instead, Spark uses a lineage graph and pipelines successive transformations to the original dataset once an *action* is called [32].

5.1.2 Actions in Spark

Spark contains a type of methods called *actions*. These operations return a value to the application or export data into storage [32]. Example of those types of actions are:

1. `COUNT`, which refers to the *exact* cardinality of a given query and corresponds to our ground truth for assessing the cardinality predictability of our model;
2. `COLLECT`, which returns a list of elements given a query;
3. `SAVE`, which stores the RDD into a permanent storage, e.g., HDFS or a local file system.

5.2 Machine learning model implementation

For UCL (Task 1), we implement the online SOM algorithm with M neurons. We make use of the neurons input lattice concept $\mathcal{L}$ described in Section 4.2.1 and we implement our UCL approach to partition the query space as described in Section 3.1.1. The neuron input lattice $\mathcal{L}$ contains all of our neurons and the winner is determined and updated as in (1) making use of Stochastic-Gradient descent. For HCL (Task 2), we implement the supervised linear regression model making use of the coefficient output lattice $\mathcal{O}$ described in Section 4.2.3, in which the coefficients $\mathbf{m}_j = [m_{j0}, m_{j1}, \ldots, m_{j2d}] \in \mathbb{R}^{2d+1}$ are associated with each query prototype $\mathbf{w}_j \in \mathcal{L}$. We then generate our predictions using Kernel regression in (16).

5.3 Range queries workload

In our implementation and experiments we dealt with multidimensional queries corresponding to a 2-dimensional data space ($d = 2$). The two boundary vectors are $\mathbf{a} = [a_1, a_2]^\top$ and $\mathbf{b} = [b_1, b_2]^\top$, $a_i \leq b_i, a_i, b_i \in \mathbb{R}$. Hence, in the experiments, a range query $\mathbf{q}$ is represented by a 4-dimensional row vector $\mathbf{q} = [a_1, b_1, a_2, b_2]$. We further adjust this representation to ease up the process of generating our query set $\mathcal{Q}$. In this context, our resulting queries are of the form $\mathbf{q} = [c_1, c_2, l]$ with *center* $c_i = \frac{a_i+b_i}{2}$, and *volume* $l = b_i - a_i$, $i = 1, \ldots, d$. Through this representation, each query $\mathbf{q}$ is a hyper-cube.

[2] http://spark.apache.org/

5.4 Cardinality prediction in Spark

To allow the execution of our model by Spark, we had to extend the main abstraction class `RDD`. (Note that the models we used or variations of them are available in MLlib.[3]) However, using these models was not possible when trying to extend the Spark-Core module because of a cyclic-dependency error between MLlib and Spark-Core. Therefore, we introduce some new methods to make use of our model. The main methods are as follows:

1. `Training`: This method trains our model given a list of executed range queries, i.e., the training set $\mathcal{Q}$. This Spark method is written in Scala:

   ```
   def training(Q : String, regression :
   Integer, neurons: Integer)
   ```

   where $\mathcal{Q}$ is the training set of queries, `regression` refers to the number of training queries to be considered for training the local linear regression models, and `neurons` refers to the number of neurons M in lattice $\mathcal{L}$ that partition the query-space.
2. `Estimation`: This method returns a cardinality estimation $\hat{y}$ given a range query $\mathbf{q} = [c_1, c_2, l]$. This Spark method is written in Scala:

   ```
   def estimation(c1: Double, c2:
   Double, l: Double)
   ```

We also developed a number of helper methods to ease the execution of the two main methods.

1. `Euclidean-Distance`, which given two range queries, it calculates the Euclidean distance;
2. `closest`, which given a query $\mathbf{q}$, it finds its closest neuron $\mathbf{w}_j \in \mathcal{L}$;
3. `descent` and `movement`, which update the winner neuron $\mathbf{w}_j$ w.r.t an incoming query $\mathbf{q}$ and learning rate β.

Executing our cardinality predictor in Spark involves two phases: the *training* and the *prediction* phase. The training phase is based on an input file containing the set $\mathcal{Q} = \{(\mathbf{q}_i, y_i)\}_{i=1}^n$ corresponding to previously executed queries. The result of this phase generates an updated list of predicting the cardinality of incoming queries through the execution of the *Estimation* method. Executing these two phases in Spark involves loading up the dataset $\mathcal{Q}$ containing our previously executed queries into an RDD. For the *training* phase, we proceed using a *map transformation* to this set and create a new representation for query set $\mathcal{Q}$. The new representation, includes the corresponding closest neuron for each query $\mathbf{q}_i$, i.e., $\mathbf{w}_j \in \mathcal{L}$, $j = \arg\min_{k\in[M]}\|\mathbf{q}_i - \mathbf{w}_k\|_2, \forall i$. Therefore, each tuple in the query set is now of the form $\mathcal{Q} = \{(\mathbf{q}_i, y_i, \mathbf{w}_j)\}_{i=1}^n$. To train the local linear regression models we iterate through all M neurons in lattice $\mathcal{L}$. For each neuron, we filter the query set $\mathcal{Q}$ such that we obtain a subset $\mathcal{Q}_k \subset \mathcal{Q} : \{(\mathbf{q}_i, y_i, \mathbf{w}_j)\}_{i=1}^n$ and $j = k$. We then based on the subset $\mathcal{Q}_k$, train the associated linear regression models $f_k(\mathbf{q}) = \mathbf{m}_k^\top \mathbf{q}$. For the estimation phase, we do not require any transformations since the only thing we need are the coefficients $\mathbf{m}_j$, $j \in [M]$ for each one of the local linear regression models $f_j(\mathbf{q}) = \mathbf{m}_j^\top \mathbf{q}$; these are simply stored as global variables on runtime or in an external file.

Figure 4 shows the standard approach and our ML approach over Spark through an example where the actual cardinality $y = 3678$ as derived from the Spark's `COUNT`. Specifically, we observe the dataset $\mathcal{B}$ of data points $\mathbf{x} \in \mathbb{R}^d$ being stored along with our training set $\mathcal{Q}$ holding training pairs of $\{(\mathbf{q}_i, y_i)\}_{i=1}^n$ and our stored coefficients $\mathbf{m}$ from the coefficient lattice $\mathcal{O}$. Moreover, Figure 4 demonstrates how the `COUNT` is obtained using a standard approach with transformations and actions. It is worth noting that the standard approach requires expensive operations such as loading the whole dataset $\mathcal{B}$, filtering it in parallel and, then, executing a `COUNT` action. However, in our ML approach, we can skip this procedure and only incur the cost of loading our Coefficient lattice $\mathcal{O}$, or if our ML model was trained at runtime, we could already have this model stored as a global variable. We, then, proceed to *predict* the answer set cardinality $\hat{y}$ using inexpensive operations.

6 Performance evaluation

We now turn to study the performance and accuracy of our implemented machine learning model in Spark. We measure the model's accuracy using various metrics and also demonstrate how our model can be used to speed up joins in databases. In addition, we compare its average execution time and standard deviation with Spark's own `COUNT` method over a standard file size. We also demonstrate the scalability of our solution by varying the file sizes. Finally, we measure the training time required by our machine learning model over different file sizes. Over the next sections, we introduce the evaluation metrics used and also explain how the training dataset was generated. Furthermore, we describe the experiment procedure followed and analyze the findings of our experiments.

6.1 Evaluation metrics

To evaluate our model we chose multiple evaluation metrics. These metrics are divided into two categories, *accuracy* and *performance*. Accuracy metrics help us determine whether the estimations generated by our model are close to the

[3] http://spark.apache.org/mllib/

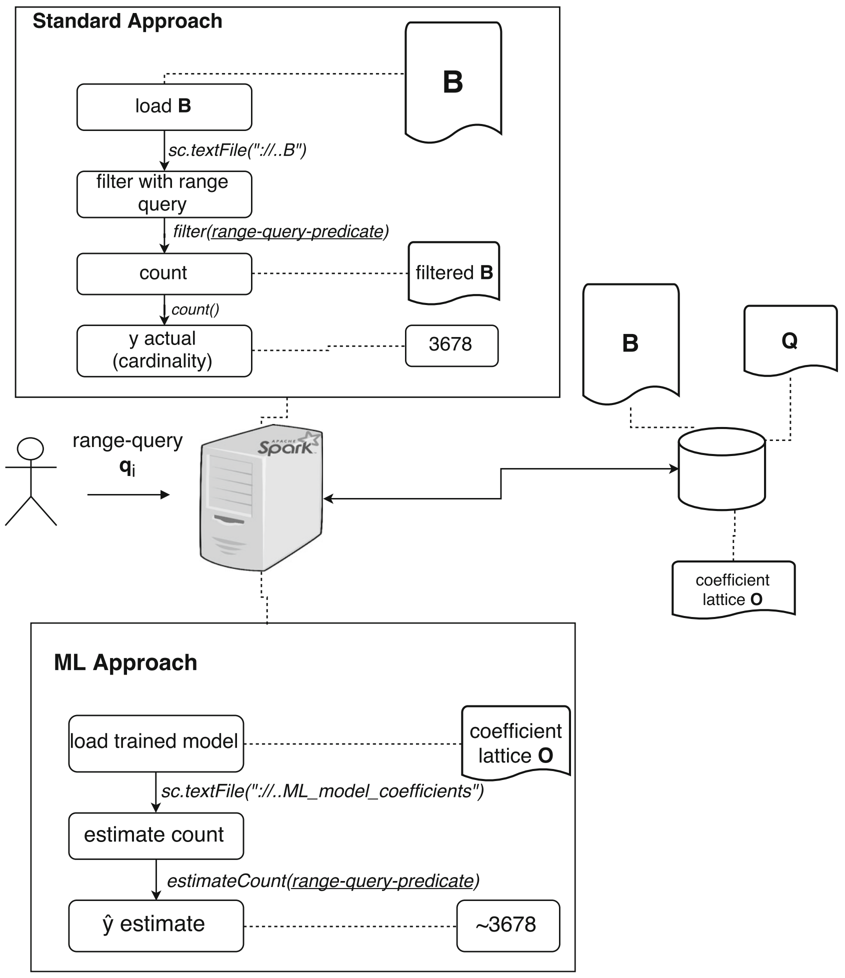


Fig. 4 The standard approach in obtaining the answer set cardinality y (COUNT) and our ML approach as a sequence of *transformations* and *actions* over the dataset $\mathcal{B}$., which predicts the cardinality $\hat{y}$

actual (true) values. These true values are obtained from the exact cardinality by invoking the COUNT Spark method. Performance metrics help us compare our solution with Spark's own implementation of COUNT in terms of query processing time. In addition, they can help us identify bottlenecks in our solution when experimenting with different file sizes or parameters, thus, evaluating the *scalability* of our approach.

6.1.1 Cardinality prediction accuracy

The first accuracy metric for a prediction model is the Root Mean Squared Deviation (RMSD) which aggregates the magnitudes of the errors $(y_i - \hat{y}_i)^2$ corresponding to the actual cardinality y_i and the predicted cardinality y_i of unseen query $\mathbf{q}_i$ for n prediction times:

$$RMSD = \sqrt{\frac{\sum_{i=1}^{n}(\hat{y}_i - y_i)^2}{n}}.$$

By adopting RMSD, we can examine how large our prediction errors are by using our model.

6.1.2 Coefficient of determination

The Coefficient of Determination R^2 is calculated using the sum of squared residuals $\sum_i^n (y_i - \hat{y}_i)^2$ and the total sum of squares $\sum_i^n (y_i - \bar{y})^2$ where $\bar{y}$ is the average cardinality value $\bar{y} = \frac{1}{n}\sum_i^n y_i$, i.e.,

$$R^2 = 1 - \frac{\sum_i^n (y_i - \hat{y}_i)^2}{\sum_i^n (y_i - \bar{y})^2}.$$

The R^2 is a real number within $[0, 1]$ and indicates how closely the obtained approximation of $\hat{y} \approx f(\mathbf{q}, \alpha)$ matches the actual cardinality y. The closer R^2 is to 1, the better our model fits the data and is able to make accurate cardinality predictions.

6.1.3 Normalized RMSD (NRMSD)

A normalized version of RMSD is also used. Knowing the normalized RMSD helps us determine the ratio of the standard deviation to the mean. When calculated, its value is a percentage that indicates us how far off we are from the mean of the true cardinality values $\bar{y}$ we are trying to predict.

$$NRMSD = \frac{RMSD}{\bar{y}}.$$

6.1.4 Forecast interval

To get a better view of the dispersion between the predicted values $\hat{y}$ and the true values y we used the RMSD and NRMSD with the mean value of the true cardinality. By using the equations below we can see how much our estimation and the actual COUNT differ by orders of magnitude, i.e., by defining the intervals:

$$\bar{y} \pm RMSD, \qquad \bar{y} \pm NRMSD.$$

To effectively measure the performance of our solution, we conducted multiple experiments each time recording the time taken by different stages in our solution. The main aspects we wanted to measure was *training time* and *execution time*. Both of these are explained below.

6.1.5 Symmetric mean absolute percentage error (SMAPE)

The mean absolute percentage error (MAPE) for assessing the predictability capability of our model is not symmetric in the sense that interchanging $\hat{y}$ and y does not lead to the same answer. The cause of this asymmetry lies in the denominator of the MAPE formula: $\frac{|y-\hat{y}|}{y}$ dividing by the actual y instead of the predicted value $\hat{y}$ leads to a different result. This issue has been raised in [8] and [25]. Makridakis in [25] proposed a variation of the MAPE to provide symmetry and protected against outliers by dividing the absolute loss $|y - \hat{y}|$ by the arithmetic mean of the actual y and the predicted $\hat{y}$. This is known as the symmetric mean absolute percentage error (SMAPE):

$$SMAPE = \frac{100\%}{n} \sum_{i=1}^{n} \frac{|y_i - \hat{y}_i|}{y_i + \hat{y}_i}$$

SPAME provides a lower and an upper percentage bound, i.e., between [0, 100]%, which is much easier to interpret adopting and due to its advantages of scale-independency.

Remark 4 Note that to evaluate the predictability of our model and fairly compare with the Spark's COUNT built-in aggregation operator we adopt the above-mentioned metrics (RMSD, NRMSD, Coefficient of Determination, Forecast interval, and SMAPE), which align with the objective minimization prediction loss functions in the second part of (13) and (15). Recall, that we learning the query-cardinality association in light of minimizing the prediction loss in (11), thus, the SCP evaluation has to be consistent with these objective functions. Moreover, such prediction accuracy metrics are widely used for evaluating the prediction error in SCP as in [11, 15, 27].

6.1.6 Model training time

The recorded model training time is the time taken for our model to execute the SOM algorithm and then train the individual linear regression models for each neuron.

6.1.7 Query execution time

To be able to compare the performance of the COUNT method of Spark with our solution, the time required for each of the functions to return a result was measured.

6.2 Datasets & workloads

In Spark we used the real dataset RS4[4] containing around 4 million 2-dimensional data ($d = 2$). The dataset contains sensor readings which were normalized and are bound within a range of $[-0.5, 0.5]$. This dataset was used to answer the cardinality range queries. We generate the training set $\mathcal{Q}$ and evaluation set $\mathcal{E}$, such that the evaluation set $\mathcal{E}$ is generated independently of $\mathcal{Q}$, thus, assuring completely unseen queries.

The training set size $|\mathcal{Q}| = s, s \in [10^5, 2 \cdot 10^5, 4 \cdot 10^5]$ and the number of neurons $M = 20$ for lattice $\mathcal{L}$ (and $\mathcal{C}$, $\mathcal{O}$). In addition, the evaluation set size $|\mathcal{E}| = 10^3$. The set $\mathcal{Q}$ contains a number of K query subspaces $\mathbb{Q}_k \subset \mathbb{R}^{2d}$. Each $\mathbb{Q}_k, k = 1, \ldots, K$, is characterized by a query pattern-generator $(\mathbf{c}_k, \mathbf{v}_k, \ell_k)$. The center of each $\mathbf{q}$ of $\mathbb{Q}_k$, for each dimension $i, i = 1, \ldots d$, is sampled from a Gaussian distribution $\mathcal{N}(c_{ki}, v_{ki})$ with mean c_{ki}, variance v_{ki}, and radius ℓ_{ki}; i.e., lower bound $a_{ki} = x_{ki} - \ell_{ki}$ and upper bound $b_{ki} = x_{ki} + \ell_{ki}$, center $x_{ki} \sim \mathcal{N}(c_{ki}, v_{ki})$. The mean $\mathbf{c_k}$ for each dimension i was drawn uniformly at random, with $c_{k1} = [0.05, 0.2]$ and $c_{k2} = [0.1, 0.25]$. The variance was constant for each subspace and at each dimension set to $v_{ki} = 0.01$. The volume of each query $2\ell_{ki}$ is drawn uniformly at random from $[0.01, 0.05]$. A random query $\mathbf{q}$ is then generated as follows: a $\mathbb{Q}_k$ is selected uniformly at random from K query spaces with equal probability $\frac{1}{K}$. Then, from $\mathbb{Q}_k$ we obtain the center (lower and upper values) and volume values for all attributes. Table 1 shows the parameters and the default values.

6.3 Performance assessment

The experiments were ran on a system with an i7 8-core processor and 6 GB RAM, under an Ubuntu 14.04 LTS OS.

[4]http://archive.ics.uci.edu/ml/datasets/Gas+sensor+array+under+dynamic+gas+mixtures

Table 1 Parameters

Parameter	(Default) Value/Range
Data dimension d	2
Dataset size $\lvert\mathcal{B}\rvert$	4,178,504 (RS4)
No. of neurons M	20
Training/Evaluation set $\lvert\mathcal{Q}\rvert$, $\lvert\mathcal{E}\rvert$	$\lvert\mathcal{Q}\rvert = 100,000$, $\lvert\mathcal{E}\rvert = 1000$
Subspaces K	5

Spark (version 1.6) was setup in standalone mode working. We trained our models using the `training` method implemented over Spark. A number of experiments were executed and are outlined below:

1. Execute the evaluation set $\mathcal{E}$ against the dataset RS4 each time recording:
   - the cardinality value y from the Spark's `COUNT`.
   - the predicted cardinality value $\hat{y}$ from our model.
   - the execution time for Spark's `COUNT` method.
   - the execution time for our model.
2. Execute 100 queries over different file sizes: 1GB, 2GB, and 3GB and find the *average execution time* and its standard deviation for both `COUNT` over Spark and our model implementation.
3. Execute the model training method `training` 100 times over different file sizes and obtain the average training time. The training file sizes are: (i) 'Small File' containing $1 \cdot 10^5$ training pairs of queries and their cardinality values, (ii) 'Medium File' containing $2 \cdot 10^5$ training pairs of queries and their cardinality values, and (iii) 'Large File' containing $4 \cdot 10^5$ training pairs of queries and their cardinality values.

Using the results of these experiments, we were able to construct multiple **graphs** and run other **side-experiments**. We analyze our findings in the following section.

6.3.1 Accuracy experiments

Accuracy Experiment 1 Table 2 shows the forecast interval, the RMSD value, the SMAPE value, and the coefficient of determination for our model. We present the actual and predicted cardinality in Fig. 5 (left) and a histogram, in Fig. 5 (right), showing the distribution of the difference $|y - \hat{y}|$ between our predictions $\hat{y}$ and the actual values y.

As shown in Table 2, the true mean value for count is 91876 and our estimation is off at any time by around 10149 or 11% from the actual value for count. In addition, the coefficient of determination $R^2 = 0.99$ indicating that our model fits the cardinality and query space significantly well and can proceed with accurate approximation of the $y = f(\mathbf{q}, \alpha)$ function. Examining the Table 2 and the Fig. 5, we can determine the quality of accuracy of our predictions. Using these metrics, big data analysts can decide whether such estimations are acceptable with respect to the applications needs. Moreover, it is worth mentioning that the SMAPE value is 9% indicating the normalized symmetric absolute deviation from the actual cardinality value.

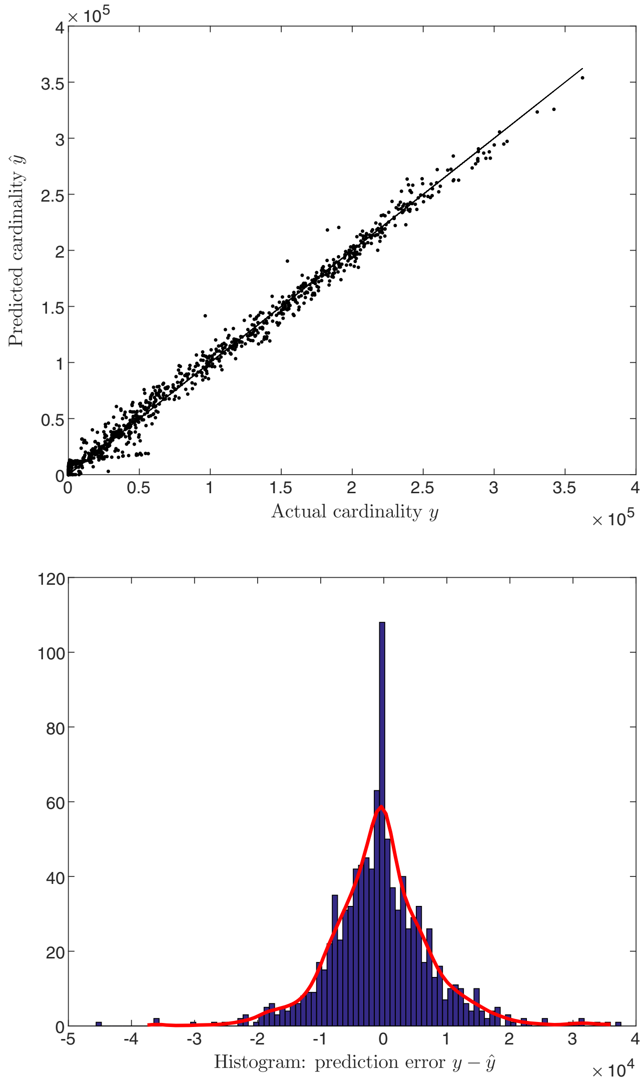


Fig. 5 (Left) The predicted $\hat{y}$ and actual y cardinality values; (right) histogram showing the distribution of the prediction error $y - \hat{y}$

Table 2 Forecast interval for Experiment 1

Forecast Interval	91876 ± 10149
Forecast Interval (%)	$91876 \pm 11\%$
RMSD	10149
R^2	0.99
SMAPE	9%

Accuracy Experiment 2 Analyzing the results of Experiment 1, allowed us to gain some insight as to how accurate our predictions were. However, we wanted an effective way to measure whether such approximations are acceptable in applications we wish to use the proposed cardinality estimator. One of the ways our estimator can be used is for solving the *selectivity estimation* problem. Specifically, we examine our model from a qualitative perspective in the case where such an estimator can be used in query optimization. When joining two database relations, $\mathcal{R} \bowtie \mathcal{S}$, a common practice of optimized `JOIN` algorithms is to find the smaller relation in terms of cardinality and then perform the `JOIN` algorithm. In this case we need the cardinality of the two relations $\mathcal{R}$ and $\mathcal{S}$. In cases where the cardinality is unknown or $\mathcal{R}$ and $\mathcal{S}$ are intermediate relations, we need to perform a `COUNT` operation over both relations to find the smallest one. However, such an operation can be expensive especially in a Big Data environment. Hence, by adopting our SCP estimator, this operation can be performed at a fraction of the time. Our experiment measures the average false rate of our SCP estimator. By false rate, we mean the probability of our estimator being wrong about a comparison between the cardinalities of the two relations under the $\bowtie$ operator. We use a zero–one loss function, which records the false rate for our SCP estimator. To be more precise, the ground truth value of a comparison, termed as Ground Truth Comparison (GTC) between the two relations comes from the actual system and from our SCP estimator. A comparison can have one of the following outcomes: (i) $|\mathcal{R}| > |\mathcal{S}|$; (ii) $|\mathcal{R}| < |\mathcal{S}|$; and (iii) $|\mathcal{R}| = |\mathcal{S}|$. If GTC and PRC are not in *agreement* for the outcome of a comparison then the false rate is incremented, otherwise it remains as is. By the end of our experiment we are left with a percentage denoting the probability of our estimator being incorrect when comparing two relations. For every result we got, we would compare it with the other $N - 1$ results we had. We would first compare the *actual* COUNT y and see which of the two queries had the largest cardinality. We would then compare their corresponding *estimated* cardinality result. The false rate provides insight on the *degree of confidence* on adopting our SCP methodology for the selectivity estimation problem in light of `JOIN` query optimization.

The Algorithm 2 explains this process with comparing the decisions coming from the GTC and our SCP method. The averaged false rate is 0.034342, which basically indicates that out of all the $N - 1 = 999$ query optimization decisions, that we made for each query, only 3% of them were false. That is, if our estimator was used to decide which relation is the smallest out of the ones involved in a `JOIN`, the smallest one would be found 97% of the time. A false rate of 3% indicates the upper qualitative degree of confidence for adopting our model in query optimization regardless of the quantified prediction error. This corresponds to the degree of confidence for using our SCP method for the selectivity estimation in `JOIN` queries without actually executing the query before the relational join $\bowtie$ operator.

Algorithm 2 False rate for selectivity estimation in `JOIN` query optimization

```
1  begin
2      average-false-rate ← 0
3      N ← 1000
4      for $i = 1{:}N$ do
5          false-rate ← 0
6          for $j = 1{:}N$ do
7              if decision(GTC) $\neq$ decision(SCP) then
8                  false-rate ← false-rate + 1
               end
           end
9          average-false-rate ← average-false-rate +
           false-Rate/$(N - 1)$
       end
10     average-false-rate ← average-false-rate/$N$
11     end
   end
```

Accuracy Experiment 3 Figure 6(left) shows the RMSD error for our model against number of neurons $M \in 5, 10, 20, 30, 50$ for a constant number of query subspaces $K = 20$ over RS4 with $d = 2$. Our model achieves very low error as M increases indicating the need for an increased partitioning of our query-space. In addition, Fig. 6(right) demonstrates how an increase in M also affects how well our model $\mathcal{M}$ fits the underlying unknown function $y = f(\mathbf{q}, \alpha)$. We note that for $M > 20$, the coefficient of determination does not change significantly suggesting that there is no need for introducing more neurons.

6.3.2 *Performance experiments*

The next set of experiments were focused around the performance of our solution. This involved experiments to compare the execution time of the `COUNT` method over Spark and our method. In addition, we measured the performance of the training phase, which is essentially, the time required by the clustering process and the training of the linear regression models.

Performance Experiment 1 We measure the execution times obtained for Spark and our implementation by executing the evaluation set $\mathcal{E}$, $|\mathcal{E}| = 1000$ over RS4 holding 4 million rows. We record the time (ms) t_i required to obtain a value for `COUNT` for each query $\mathbf{q}_i \in \mathcal{E}$ and get the Mean Execution Time (MET) for both Spark and our model, i.e.,

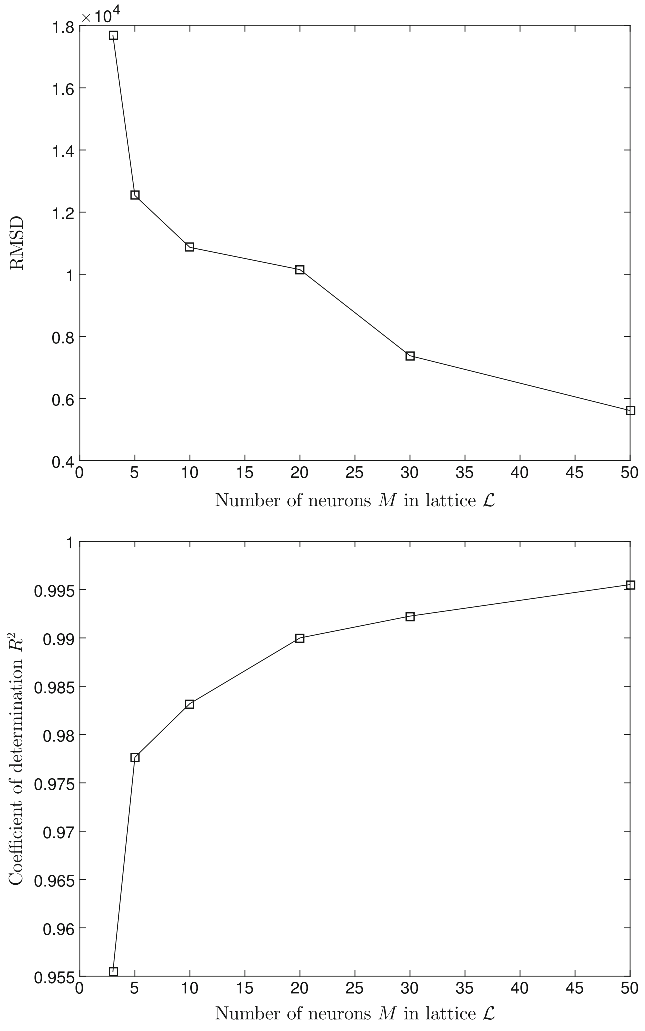


Fig. 6 (Left) The RMSD is improving by increasing the number of neurons M in lattice $\mathcal{L}$; (right) Coefficient Of Determination R^2 improving after varying the number of neurons in lattice $\mathcal{L}$

$MET = \frac{1}{n}\sum_{i=1}^{n} t_i$, with $n = |\mathcal{E}| = 1000$. We compare the two means MET_1 for Spark and MET_2 for our model and observe that we gain superior performance by approximately 4 orders of magnitude as $MET_1 \approx MET_2 \cdot 10^4$. We also measure the Standard Deviation (SD) for the execution times of both Spark (SD_1) and our model (SD_2). The MET, SD, and median of the execution times are shown in Table 3. Again, we observe that the same pattern arising, showing that our model has less fluctuations in performance that Spark as $SD_1 \approx \frac{1}{2} \cdot SD_2 \cdot 10^5$. Therefore, it is safe to conclude that our solution provides a tremendous boost to big data applications and large-scale analytics tasks based on the cardinality of the analytics query answer sets.

Table 3 Performance comparison of Spark and our model

	COUNT over Spark	Our model for COUNT
Mean Execution Time (ms)	83.165	0.0074
Standard Deviation	261.216	0.0055
Median of execution times	69.128	0.0053

Table 4 File sizes

Dataset name	File size (GB)
Small dataset (SD)	1 GB
Medium dataset (MD)	2 GB
Large dataset (LD)	3 GB

Performance Experiment 2 For the second performance experiment, we constructed two larger files from the original dataset $\mathcal{B}$ (RS4). We wanted to vary the size of the dataset to see the performance of our solution compared to Spark when the dataset size is increasing. We kept the original dataset and renamed it as 'Small Dataset' (SD) with $|SD| = |\mathcal{B}| \approx 4 \cdot 10^6$. We then generated two more files, 'Medium Dataset' (MD) and 'Large Dataset' (LD) by replicating the existing rows by a factor of $\gamma \in \{2, 3\}$. In this context, we obtain: $|MD| = \gamma|\mathcal{B}| \approx 8 \cdot 10^6, \gamma = 2$ and $|LD| = \gamma|\mathcal{B}| \approx 1.2 \cdot 10^7, \gamma = 3$. The corresponding file sizes for these datasets are shown in Table 4. We, then, executed the evaluation set $\mathcal{E}$ over each data-set and obtained the MET for Spark's COUNT method and our model as in *Performance Experiment 1*. The results of this experiment are shown in Fig. 7.

We can observe that as the file size gets larger the MET for COUNT over Spark exponentially grows. On the contrary, our model has relatively constant MET over all three files. The constant performance is due to the fact that

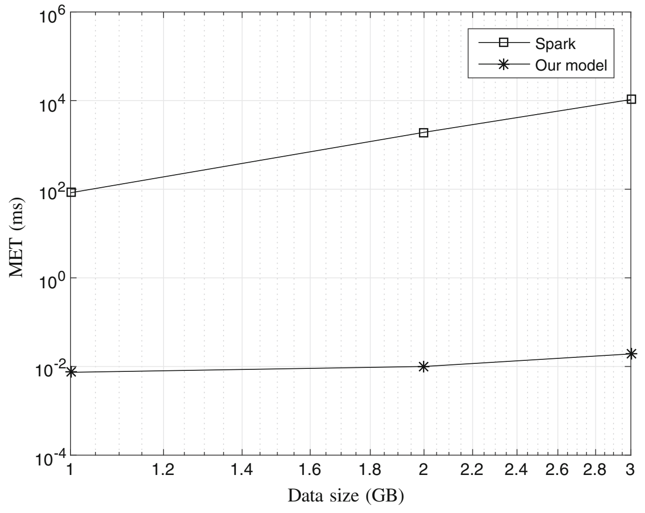


Fig. 7 Scalability: performance comparison of Spark and our method for different data set size (GB)

Table 5 Number of training pairs (query,cardinality)

	For SOM	For linear regression
Small file	40,000	100,000
Medium file	80,000	200,000
Large file	160,000	400,000

our implementation *does not need to execute any transformations or actions over Spark*, thus, no computation or memory usage is required other than evaluating the linear regression model. This trend-line in Fig. 7 demonstrates how data size affects Spark and our solution. Evidently, this indicates the **scalability capability** of our model which is independent on the increase of the underlying data sizes.

Performance Experiment 3 In this performance experiment, we measured the time required for our model to pass through the training phase. Again, the number of neurons was set to $M = 20$. To better examine the training phase, we have created two larger training datasets. We left the original training set $\mathcal{Q}$, $|\mathcal{Q}| = 1 \cdot 10^5$ hereby referred to as 'Small Query-set' (SQ). Using the same process we generated a 'Medium Query-set' (MQ), $|MQ| = \gamma' \cdot |\mathcal{Q}|$ and a 'Large Query-set' (LQ), $|LQ| = \gamma' \cdot |\mathcal{Q}|$ where $\gamma' \in \{2, 4\}$. The number of queries (and their corresponding actual cardinality values) contained in each file, along with the number of queries used for the SOM and the linear regression models training, are shown in Table 5. We run the training process 100 times over each file and obtain the Mean Training Time (MTT).

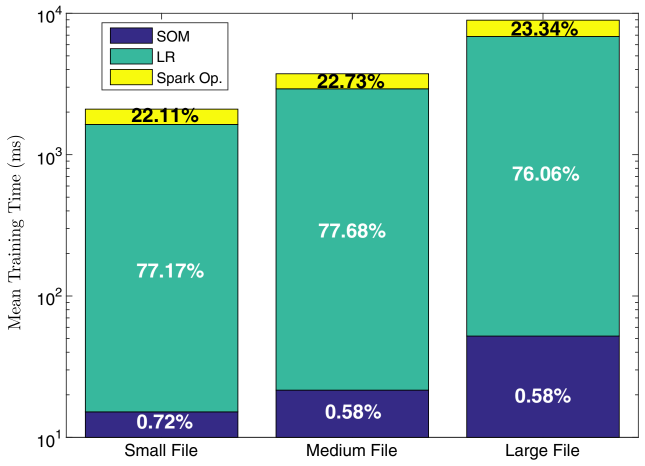


Fig. 8 Break down of the total Mean Training Time (MTT) over different file sizes in logarithmic scale; The mean training time consists of training for SOM, training for Local Linear Regression (LR) and allocated time for Spark Operations (Spark Op.)

The results of this experiment are shown in Fig. 8. We observe that the MTT grows exponentially with the size of the training dataset. Although, this might be a drawback of our implementation, **it is a cost that will incur only once**, as the training process is not required to run again unless the dataset changes. In addition, Fig. 8 shows the time required by the different stages in the training phase. The time allocated to Spark ('Spark Op.' in Fig. 8) refers to time not accounted for by our operations but allocated to Spark's internal operations. The two stages concerned by our solution is SOM training and local linear regression training (LR in Fig. 8). It is clear that training the individual local linear models required longer time. However, the difference in time between the two stages is because of the larger amount of queries involved in the local linear regression training stage.

7 Conclusions & future work

We introduce a novel perspective and solution for the problem of Set Cardinality Prediction. The fundamental unique characteristic of our approach is that it is query–driven. This is especially important for big data settings, as an increase in the underlying dataset size is largely inconsequential for our method's efficiency and accuracy. The contributed neuron-based machine learning model (i) extracts knowledge from previous queries and their results, (ii) projects this knowledge onto two lattices (cardinality and coefficient lattices) of two self-organizing maps, and (iii) predicts the cardinality of the answer set. The model quantizes the query space and forms a cardinality and linear regression plane through learning the query-to-cardinality association. Our comprehensive experiments showcased the model's robustness and that our model achieves very small error rates with small memory footprints outperforming the data-centric state-of-the-art. The proposed model represents the only solution applicable to general modern big data environments, which may include data nodes/owners placing access restrictions (e.g., for sensitive data) and/or where data accesses may be too costly to execute (e.g., in a cloud setting).

Our plans for future work focus on: (i) developing a framework that can dynamically and optimally switch between the training phases and query execution phases as analysts interests shift between data subspaces, and (ii) dealing with data spaces with on-line data mutations (insertions, deletions, and updates).

Acknowledgments We would like to thank the anonymous reviewers for their insightful comments on the paper, which led to an improvement of our work.

Appendix: Nomenclature

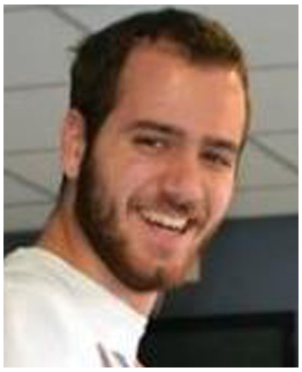
Notation	Explanation
$\mathcal{B}$	Dataset
$\mathcal{Q}$	Set of pairs (query,cardinality)
$\mathcal{E}$	Evaluation set
$\mathcal{L}$	Neuron lattice
$\mathcal{O}$	Coefficient output lattice
$\mathcal{C}$	Cardinality output lattice
M	Number of neurons
$\mathbf{q}$	Range query vector
$\mathbf{w}$	Neuron vector
$\mathbf{m}_j$	Coefficients vector for local linear regression, associated with neuron $\mathbf{w}_j$
$y \in \mathbb{N}$	Cardinality
$\mathcal{M}$	Machine learning model
$\beta \in (0, 1)$	Learning rate
$\mathbf{r}_j = [r_{j,1}, r_{j,2}]$	Location vector of neuron $\mathbf{w}_j$
$h(i, j : t)$	Neighborhood function
ρ	Width of the neighborhood function
α	Parameter set for model $\mathcal{M}$
Λ	Parameter space
$\hat{y}$	Predicted cardinality
$\mathcal{J}(\alpha)$	Risk function
$L(\hat{y}, y)$	Loss function
$\mathcal{N}_k(\mathbf{q})$	Set of the k-th closest queries to query $\mathbf{q}$.
$K_\epsilon(u)$	Kernel function
θ	Stopping training threshold.
T_t	1-norm between successive estimates of neurons and cardinality prototypes

References

1. Aboulnaga A, Chaudhuri S (1999) Self-tuning histograms: building histograms without looking at data. ACM SIGMOD Record 28(2):181–192
2. Anagnostopoulos C (2016) Quality-optimized predictive analytics. Appl Intell 45(4):1034–1046
3. Anagnostopoulos C, Triantafillou P (2014) Scaling out big data missing value imputations: Pythia vs. godzilla. In: Proceedings of the 20th ACM SIGKDD international conference on knowledge discovery and data mining, KDD '14. ACM, New York, pp 651–660
4. Anagnostopoulos C, Triantafillou P (2015) Learning set cardinality in distance nearest neighbours. In: 2015 IEEE international conference on data mining, pp 691–696
5. Anagnostopoulos C, Triantafillou P (2015) Learning to accurately count with query-driven predictive analytics. In: 2015 IEEE international conference on big data (big data), pp 14–23. https://doi.org/10.1109/BigData.2015.7363736
6. Anagnostopoulos C, Triantafillou P (2017) Efficient scalable accurate regression queries in in-dbms analytics. In: 2017 IEEE 33rd international conference on data engineering (ICDE), pp 559–570
7. Anagnostopoulos C, Triantafillou P (2017) Query-driven learning for predictive analytics of data subspace cardinality. ACM Trans Knowl Discov Data 11(4):47:1–47:46. https://doi.org/10.1145/3059177
8. Armstrong J, Collopy F (1992) Error measures for generalizing about forecasting methods: empirical comparisons. Int J Forecast 8(1):69–80
9. Balac N, Sipes T, Wolter N, Nunes K, Sinkovits B, Karimabadi H (2013) Large scale predictive analytics for real-time energy management. In: 2013 IEEE international conference on big data. IEEE, pp 657–664
10. Bousquet O, Bottou L (2008) The tradeoffs of large scale learning. In: Advances in neural information processing systems, pp 161–168
11. Bruno N, Chaudhuri S, Gravano L (2001) Stholes: a multidimensional workload-aware histogram. In: ACM SIGMOD Record, vol 30. ACM, pp 211–222
12. Chaudhuri A, Wei TH, Lee TY, Shen HW, Peterka T (2014) Efficient range distribution query for visualizing scientific data. In: 2014 IEEE pacific visualization symposium. IEEE, pp 201–208
13. Cormode G, Garofalakis M, Haas PJ, Jermaine C (2012) Synopses for massive data: samples, histograms, wavelets, sketches. Foundations and Trends in Databases 4(1–3):1–294
14. Dean J, Ghemawat S (2010) Mapreduce: a flexible data processing tool. Commun ACM 53(1):72–77. https://do.org/10.1145/1629175.1629198
15. Gunopulos D, Kollios G, Tsotras J, Domeniconi C (2005) Selectivity estimators for multidimensional range queries over real attributes. The VLDB J Int J Very Large Data Bases 14(2):137–154
16. Haas PJ, Swami AN (1992) Sequential sampling procedures for query size estimation, vol 21 ACM
17. Ikonomovska E, Gama J, Džeroski S (2011) Learning model trees from evolving data streams. Data Min Knowl Disc 23(1):128–168
18. Indyk P, Motwani R (1998) Approximate nearest neighbors: towards removing the curse of dimensionality. In: Proceedings of the thirtieth annual ACM symposium on theory of computing, STOC '98. ACM, New York, pp 604–613
19. Kohonen T, Somervuo P (1998) Self-organizing maps of symbol strings. Neurocomputing 21(1):19–30
20. Kolomvatsos K, Anagnostopoulos C, Hadjiefthymiades S (2015) An efficient time optimized scheme for progressive analytics in big data. Big Data Res 2(4):155–165
21. Kosko B (1991) Stochastic competitive learning. IEEE Trans Neural Netw 2(5):522–529
22. Landset S, Khoshgoftaar TM, Richter AN, Hasanin T (2015) A survey of open source tools for machine learning with big data in the hadoop ecosystem. J Big Data 2(1):24. https://doi.org/10.1186/s40537-015-0032-1
23. Lin CY, Tsai CH, Lee CP, Lin CJ (2014) Large-scale logistic regression and linear support vector machines using spark. In: 2014 IEEE international conference on big data (big data). IEEE, pp 519–528
24. Ma J, Theiler J, Perkins S (2003) Accurate on-line support vector regression. Neural Comput 15(11):2683–2703

25. Makridakis S (1993) Accuracy measures: theoretical and practical concerns. Int J Forecast 9(4):527–529
26. Ong BT, Sugiura K, Zettsu K (2014) Dynamic pre-training of deep recurrent neural networks for predicting environmental monitoring data. In: 2014 IEEE international conference on big data (big data). IEEE, pp 760–765
27. Srivastava U, Haas PJ, Markl V, Kutsch M, Tran TM (2006) Isomer: consistent histogram construction using query feedback. In: 22nd international conference on data engineering (ICDE'06). IEEE, pp 39–39
28. To H, Chiang K, Shahabi C (2013) Entropy-based histograms for selectivity estimation. In: Proceedings of the 22nd ACM international conference on information & knowledge management. ACM, pp 1939–1948
29. Vavilapalli VK, Murthy AC, Douglas C, Agarwal S, Konar M, Evans R, Graves T, Lowe J, Shah H, Seth S, Saha B, Curino C, O'Malley O, Radia S, Reed B, Baldeschwieler E (2013) Apache hadoop yarn: yet another resource negotiator. In: Proceedings of the 4th annual symposium on cloud computing, SOCC '13. ACM, New York, pp 5:1–5:16. https://doi.org/10.1145/2523616.2523633
30. Viswanathan R, Jain P, Laxman S, Arasu A (2011) A learning framework for self-tuning histograms. arXiv:1111.7295
31. Vitter JS, Wang M, Iyer B (1998) Data cube approximation and histograms via wavelets. In: Proceedings of the seventh international conference on information and knowledge management. ACM, pp 96–104
32. Zaharia M, Chowdhury M, Das T, Dave A, Ma J, McCauley M, Franklin MJ, Shenker S, Stoica I (2012) Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: Proceedings of the 9th USENIX conference on networked systems design and implementation. USENIX Association, pp 2–2

Dr. Christos Anagnostopoulos is an Academic Research Fellow (tenure track) in the School of Computing Science at the University of Glasgow. His expertise is in the areas of network-centric intelligent systems, in-network statistical predictive modelling, and contextual information processing in large-scale sensor/UxV/Edge networks. He has received funding for his research by the EC/H2020, EPSRC, is coordinating (Principal Investigator) the projects: EU H2020/GNFUV and EU H2020 Marie Skłodowska-Curie (MSCA)/ INNOVATE, and is a co-PI of the EU PRIMES. Dr Anagnostopoulos is an author of over 100 publications, with over 1500 citations in refereed scientific journals/conferences. He is a member of the Information, Data, and Analysis (IDA) and associate member of the Glasgow Systems Section (GLASS) at Glasgow. Dr Anagnostopoulos before joining Glasgow was appointed as an Assistant Professor at Ionian University and University of Thessaly in the area of network-centric information systems. He has held postdoctoral research positions at University of Glasgow (UK/EPSRC) and University of Athens (EC-funded projects) in the areas of large-scale statistical learning & predictive analytics in distributed environments. He holds a BSc (Hons. and Valedictorian) in Informatics & Telecommunications, MSc (distinction) in Advanced Information Systems, and PhD in Computing Science, University of Athens. He is an associate fellow of the HEA, member of ACM and IEEE.



Fotis Savva is currently a PhD student and a member of the IDEAS research group within the University of Glasgow. Fotis has obtained his B.Sc. in 2016 from the School Of Computing Science at the University of Glasgow. His research interests are: Machine Learning, Big Data Engines, Expediting Complex Query Processing using ML Models, and Explanations in Aggregate Query Results.

Peter Triantafillou is Professor of Data Systems at the Department of Computer Science at the University of Warwick since September 2017, Associate Director of the Urban Big Data Research Centre– a national infrastructure for urban data services and analytics – and Honourary Senior Research Fellow at School of Computing Science at the University of Glasgow. Prior to that, he held the Data Systems Chair at the School of Computing Science at the University of Glasgow. Peter has also held professorial positions at Simon Fraser University (1991-1995), the Technical University of Crete (1995-2002), the University of Patras (2002-2013), and visiting professorships at the Max-Planck Institute for Informatics (in 2004-2005 and in 2012-2013). Peter received his PhD in computer science from the University of Waterloo in 1991, being the Department of Computer Science and the Faculty of Mathematics nominee for the Gold Medal for outstanding achievements at the Doctoral level.

Peter has published extensively in top journals and conferences in the above areas, has served in the Technical Program Committees of more than 120 international conferences, and has been the PC Chair or Vice-chair in several prestigious conferences. Peter has received the best paper awards at the ACM SIGIR Conference (on Information Retrieval) in July 2016 and at the ACM CIKM Conference (on Information and Knowledge Management) in November 2006 and is a co-designer of several innovative systems (such as the MINERVA decentralized search engine and the eXO decentralized social networking system).