Learning to Sample: Counting with Complex Queries[*]

Brett Walenz, Stavros Sintos, Sudeepa Roy, and Jun Yang
Duke University
Durham, NC, USA
{bwalenz, ssintos, sudeepa, junyang}@cs.duke.edu

ABSTRACT

We study the problem of efficiently estimating counts for queries involving complex filters, such as user-defined functions, or predicates involving self-joins and correlated subqueries. For such queries, traditional sampling techniques may not be applicable due to the complexity of the filter preventing sampling over joins, and sampling after the join may not be feasible due to the cost of computing the full join. The other natural approach of training and using an inexpensive classifier to estimate the count instead of the expensive predicate suffers from the difficulties in training a good classifier and giving meaningful confidence intervals. In this paper we propose a new method of *learning to sample* where we combine the best of both worlds by using sampling in two phases. First, we use samples to learn a probabilistic classifier, and then use the classifier to design a stratified sampling method to obtain the final estimates. We theoretically analyze algorithms for obtaining an optimal stratification, and compare our approach with a suite of natural alternatives like quantification learning, weighted and stratified sampling, and other techniques from the literature. We also provide extensive experiments in diverse use cases using multiple real and synthetic datasets to evaluate the quality, efficiency, and robustness of our approach.

PVLDB Reference Format:
Brett Walenz, Stavros Sintos, Sudeepa Roy, Jun Yang. Learning to Sample: Counting with Complex Queries. *PVLDB*, 13(3): 389-401, 2019.
DOI: https://doi.org/10.14778/3368289.3368302

1. INTRODUCTION

Counting is a fundamental problem in query processing. Counting queries can be expensive to evaluate, especially if it involves testing a complex predicate to decide whether an object should be counted towards the total. Consider the following example.

[*]This work was supported by NSF grants CCF-1513816, CCF-1546392, IIS-1408846, IIS-1552538, IIS-1703431, IIS-1718398, IIS-1814493, NIH grant 1R01EB025021-01, ARO grant W911NF-15-1-0408, and a Google Faculty Award. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the funding agencies.

Proceedings of the VLDB Endowment, Vol. 13, No. 3
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3368289.3368302

EXAMPLE 1 (COUNTING POINTS WITH FEW NEIGHBORS). *Suppose table $D(\underline{id}, x, y)$ stores a set of 2d points, and we would like to count how many points have fewer than k points within distance d from them. We can write the following SQL query:*

```
SELECT COUNT(*) FROM
(SELECT o1.id FROM D o1, D o2
 WHERE SQRT(POWER(o1.x-o2.x,2)+POWER(o1.y-o2.y,2))<=d
 GROUP BY o1.id HAVING COUNT(*) <= k);
```

Here, the objects to be counted are produced by a self-join with a complex condition, followed by GROUP BY and HAVING. This "neighborhood" query has been well studied, with specialized index structures and processing algorithms. Still, there is a good chance that a typical database system will perform poorly, either because it has no specialized support for this query type, or it simply fails to recognize this query type from the way the query is written. Thus, making such queries run faster can require a lot of effort and expertise. There are even more complex cases involving expensive user-defined functions commonly found in machine learning workloads. The problem we tackle in this paper is how to evaluate counting queries efficiently, and in a general way.

Approximate answers are widely accepted for such expensive counting queries. Sampling is a powerful technique for producing approximate answers with statistical guarantees, with a long tradition and active research of its applications in databases. Yet sampling for complex queries remains a difficult problem. In general, not all query operators "commute" with sampling. For instance, in Example 1, if we only take a sample of D and evaluate the query on this sample, it would be difficult to make sense of the result because even the neighbor counts produced by the inner aggregation query would be off to begin with. Worse, if the predicate involves a black-box function with table inputs, we cannot expect sampling input tables to produce usable results.

Still, a viable approach is to conceptually treat the problem as counting the number of objects satisfying a predicate, where the objects can be enumerated or sampled efficiently, but the predicate is complex and expensive (e.g., involving user-defined functions or arbitrarily nested subqueries). We would sample some objects for which we evaluate the predicate "in full," and then use these results to derive an estimate. For instance, in Example 1, given a point o1 from D, the predicate would be a query over (full) D parameterized by the values of o1.x and o2.x. Of course, evaluating the predicate in full for each sampled object can be expensive, but evaluating the original query as a whole can be much worse—there may be no better way for the database systems to process this query than a nested-loop join. While this sampling-based approach is simple and general, a question is whether we can make it more efficient.

Machine learning is another natural approach to this problem. It has the potential of being more "sample-efficient" because of its ability to generalize to unseen objects. One could draw some samples, pay the cost to "label" them (i.e., evaluate the expensive predicate), and use the labeled samples to learn a cheap classifier that approximates the result of the expensive predicate. The learned classifier can then be applied to objects to obtain an estimated count. Beyond this naive approach, we can apply ideas from *quantification learning* [6]. However, some difficulties remain: it is hard to offer meaningful statistical guarantees (such as confidence intervals provided by sampling), and training a good classifier can be difficult and tricky itself (e.g., with challenges such as feature and model selection as well as overfitting).

A natural question is whether we can combine learning and sampling to get the "best of both worlds": we want the ability to generalize by learning, but at the same time we want the statistical guarantees offered by sampling. This paper answers this question in positive. One idea is to use sampling to assess the errors produced by the learned classifier and correct its estimated count. We also provide a novel alternative that "learns to sample." The key idea here is not to rely directly on the learned classifier's predictions, but instead exploit the classifier's knowledge in a more controlled manner by using it to design a sampling scheme. Then, we apply the sampling scheme to derive our estimates, complete with statistical guarantees. A good classifier leads to an efficient sampling scheme that uses few samples to get low-variance estimates; on the other hand, a poor classifier can lead to a less efficient sampling scheme that needs more samples to achieve the same accuracy, but we will always have unbiased estimates with confidence intervals.

Specifically, we make use of the scores produced by classifiers that reflect how confident they are in their predictions. Such scores are readily available for popular classification methods in standard libraries. A straightforward method is *learned weighted sampling*, which assigns higher sampling probabilities to objects that are more confidently predicted to contribute to the result count. This method is still sensitive to the scores produced by the classifiers, and tends to focus more on confidently positive objects instead of uncertain objects—but arguably, uncertain objects intuitively provide more information when labeled.

Hence, we further propose *learned stratified sampling*, which relies even less on the quality of the classifier. Instead of using the values of the scores, we use the scores only to induce an ordering among the objects. Based on this ordering, and with help from some additional samples, we find the optimal stratified sampling design that jointly considers the partitioning of objects into strata and the allocation of additional samples across strata. The score-induced ordering is useful because it brings together objects with similar levels of uncertainty, and in particular encourages putting the certainly positive objects and certainly negative objects into separate strata with low within-stratum variances. The sampling design problem is challenging because of joint consideration of stratification and allocation; we propose algorithms for this optimization problem with trade-offs between speed and optimality.

Our experiments show that our learn-to-sample approach generally outperforms approaches that are based on either sampling or learning alone, or those that apply sampling only to error assessment and correction. We achieve unbiased estimates with lower variances than other approaches, and in practice, the overhead of learning and sampling design is negligible compared with the total cost of evaluating expensive predicates on samples. Moreover, learned stratified sampling delivers robust performance even with poor classifiers. Finally, a key practical advantage of our learn-to-sample approach is that it is easy to implement: its constituent learning and sampling components are available off-the-shelf, so we readily benefit from both the classic sampling literature and a growing toolbox of classification algorithms. For example, for our experiments, we were able to apply standard classification algorithms out-of-box with very little tuning, thanks to the robustness of the learn-to-sample approach.

2. PROBLEM DEFINITION

Consider a set of objects $\mathcal{O}$, and a Boolean predicate $\mathtt{q} : \mathcal{O} \to \{0, 1\}$, where 1 denotes true. We say an object o is *positive* if $\mathtt{q}(o) = 1$, or *negative* if $\mathtt{q}(o) = 0$. Our goal is to estimate $C(\mathcal{O}, \mathtt{q})$, the number of positive objects in $\mathcal{O}$; i.e., $C(\mathcal{O}, \mathtt{q}) = \sum_{o \in \mathcal{O}} \mathtt{q}(o)$.

In general, each object o can have a complex structure (with multiple attributes including set-valued ones), and $\mathtt{q}(o)$ can be arbitrarily complex (e.g., accessing related information beyond the contents of o, comparing o with other objects in $\mathcal{O}$, etc.).

We make two assumptions: 1) evaluation of $\mathtt{q}$ is costly; 2) members of $\mathcal{O}$ can be efficiently enumerated. The terms "costly" and "efficient," of course, are relative. While the techniques in this paper do not depend on these assumptions for correctness, our proposed approach is intended for situations where these assumptions hold. For example, a costly $\mathtt{q}$ would make it attractive to use sampling to avoid evaluating $\mathtt{q}$ for all objects, or to use a learned model that predicts the outcome of $\mathtt{q}$ at a lower cost.

It should be obvious that the problem formulation above handles single-table selection queries whose conditions potentially involve expensive user-defined functions. The problem formulation is also general enough to capture more complex queries. The first example below illustrates the case where $\mathtt{q}$ is a complex SQL condition involving an aggregate subquery; the second illustrates the case where $\mathtt{q}$ involves a black-box function.

EXAMPLE 2 (k-SKYBAND SIZE). *Consider a set of 2d points in table D(__id__, x, y). A point p_1* dominates *another point p_2 if p_1's x and y values are (resp.) no less than those of p_2 (i.e., $p_1.x \geq p_2.x \wedge p_1.y \geq p_2.y$), and at least one of them is strictly greater (i.e., $p_1.x > p_2.x \vee p_1.y > p_2.y$). The so-called* k-skyband *for the point set D is the subset of points that are dominated by fewer than k others. Given $o \in$ D, we define $\mathtt{q}(o)$ to test its membership in the k-skyband using the following SQL condition:*

```
(SELECT COUNT(*) FROM D
 WHERE x >= o.x AND y >= o.y AND (x>o.x OR y>o.y)) < k
```

Note that this predicate involves an aggregate subquery parameterized by o. The number of points in the k-skyband is then the number of points satisfying $\mathtt{q}$. *Here, object enumeration is efficient (just scan D), while predicate evaluation is costly in comparison (without specialized indexes).*

Alternatively, we can write the whole k-skyband size query using a self-join and nested aggregation, without explicitly referring to $\mathtt{q}$:

```
SELECT COUNT(*) FROM
(SELECT o1.id FROM D o1, D o2
 WHERE o2.x >= o1.x AND o2y >= o1.y
   AND (o2.x > o1.x OR o2.y > o1.y)
 GROUP BY o1.id HAVING COUNT(*) < k);
```

EXAMPLE 3 (RELEVANT DOCUMENT COUNT). *Consider a set of documents in table D(__id__, text). Each document, based on the content of its text, can be associated with zero or more labels from a predefined set of labels of interest. For example, during electronic discovery for a legal proceeding, D can be a set of emails and documents, and one such label may indicate whether a document is in support of or against a particular action. Let labels(text) denote a function that examines a document and returns the subset of labels that it is associated with. We mark a document as highly relevant if it is associated with at least k labels. The following query returns the number of highly relevant documents:*

```
SELECT COUNT(*) FROM D o
 WHERE len(labels(o.text)) >= k;
```

Here, q *is the* WHERE *predicate, but it involves a complex black-box function* `labels` *whose evaluation can be very expensive. For example, if labels are highly specialized for a given proceeding, there may not exist good automated labeling procedures and we would have to evaluate* `labels` *manually. In general, the predicate that determines whether a document is relevant can be even more complicated than counting how many labels it is associated with, but our problem formulation and solutions are designed to work with arbitrarily complicated* q.

Handling More General SQL Queries An observant reader will notice the similarity between the last query in Example 2 and the one counting points with few neighbors in Example 1. Despite the latter query's lack of an explicit per-object predicate, it is not hard to see that we can define $\mathsf{q}(o)$ for $o \in \mathtt{D}$ as the following complex SQL condition involving an aggregate subquery (analogous to Example 2 above):

```
(SELECT COUNT(*) FROM D
 WHERE SQRT(POWER(o.x-x,2)+POWER(o.y-y,2)) <= d) <= k
```

More generally, suppose we are interested in counting the number of results for the following SQL aggregate query:

```
SELECT E FROM L,R -- (Q1)
WHERE θL AND θLR
GROUP BY GL HAVING φ;
```

In the above, $\mathbf{G_L}$ is the list of group by columns, $\mathbf{L}$ denotes the list of tables with columns in $\mathbf{G_L}$, and $\mathbf{R}$ denotes the list of other tables in the join with no group-by columns; $\theta_{\mathbf{L}}$ refers to the part of the WHERE condition that be evaluated over $\mathbf{L}$ alone, $\theta_{\mathbf{LR}}$ refers to the remaining part of the WHERE condition, and ϕ refers to the HAVING condition; finally, $\mathbf{E}$ is the list of output expressions for each group. The problem of counting the number of results can be formulated by defining the set $\mathcal{O}$ of objects as:

```
SELECT DISTINCT GL FROM L WHERE θL; -- (Q2)
```

and the predicate $\mathsf{q}(o)$ as:

```
EXISTS(SELECT GL FROM L, R -- (Q3)
       WHERE θLR AND GL=o.*
       GROUP BY GL HAVING φ)
```

Again, the key takeaway is that our problem formulation is general enough to support complex queries involving joins and aggregates (besides the final counting). Our approach works well as long as the set of objects is cheap to enumerate (i.e., the local selection $\theta_{\mathbf{L}}$ in (Q2) is easy to evaluate), while the per-object predicate (Q3) is relatively more expensive (which is usually the case because of join and aggregation).

3. BASELINE METHODS

We present a number baseline methods for estimating $C(\mathcal{O}, \mathsf{q})$. While these methods are not new, we note that some connections to our problem (e.g., quantification learning and sampling-based data cleaning) have never been made explicit or evaluated previously.

3.1 Sampling-Based Methods

Simple Random Sampling (SRS) The problem of estimating $C(\mathcal{O}, \mathsf{q})$ using sampling has been studied extensively in the context of estimating proportions [24]. A straightforward method is *simple random sampling* (*SRS*). Let $\mathcal{S} \subseteq \mathcal{O}$ denote the set of n objects drawn randomly without replacement from the set $\mathcal{O}$ of all N objects. For each $o \in \mathcal{S}$, we evaluate $\mathsf{q}(o)$. Then, an unbiased estimator of $C(\mathcal{O}, \mathsf{q})$ is $\hat{p}N$, where the estimated proportion $\hat{p} = \frac{1}{n}\sum_{o \in \mathcal{S}} \mathsf{q}(o)$. There are a number of ways to derive a confidence interval for this estimation. The most popular one is the *Wald interval*, which approximates the error distribution using a normal distribution: the $(1-\alpha)$ confidence interval for $\hat{p}$ in this case is

$$\hat{p} \pm z_{\alpha/2}\sqrt{\frac{\hat{p}(1-\hat{p})}{n} \cdot \frac{N-n}{N-1}}.$$

The usual caveats apply: if q is highly selective or highly non-selective, the Wald interval is unreliable because normal distribution approximation fails; one can use the more reliable *Wilson interval* instead. See standard sampling literature [24] for details.

Stratified Sampling (SSP and SSN) Stratified sampling is a method that works especially well when the overall population can be divided into subpopulations (strata) where objects are homogeneous within each stratum. For example, if there is a way to divide $\mathcal{O}$ into two strata where one contains mostly positive objects and the other contains mostly negative objects, we can sample the two strata independently and use much fewer samples overall than SRS to achieve the same confidence interval. The problem, of course, is that we do not know the outcome of each $\mathsf{q}(o)$ unless we first evaluate it. A practical solution is to choose some attributes of o whose values are readily available and likely correlated with the outcome of $\mathsf{q}(o)$; we can then stratify $\mathcal{O}$ according to these *surrogates*. In our case, a natural choice for surrogates would be the attributes of o used in computing $\mathsf{q}(o)$; e.g., for Example 1, we would choose x and y and grid the 2d space into the desired number of strata.

Suppose we are given a partitioning of $\mathcal{O}$ into H strata $\mathcal{O}_1, \mathcal{O}_2, \ldots, \mathcal{O}_H$, where $N_h = |\mathcal{O}_h|$ denotes the size of each stratum h, and an allocation of samples $n_1, n_2, \ldots, n_H$, where n_h is the number of samples allotted to stratum h. Stratified sampling randomly draws the allotted number of samples from each stratum; denote these samples by $\mathcal{S} = \cup_{h=1}^{H}\mathcal{S}_h$, where $n_h = |\mathcal{S}_h|$. For each stratum h, using $\mathcal{S}_h$, we can derive an unbiased estimator for the proportion $\hat{p}_h$ of positive objects therein (as described for SRS above). Then, an unbiased estimator of $C(\mathcal{O}, \mathsf{q})$ is $\hat{p}N$, where $\hat{p} = \sum_{h=1}^{H} W_h\hat{p}_h$ is the estimated overall proportion and $W_h = N_h/N$ is the weight of stratum h. The variance in $\hat{p}$ is

$$\mathrm{Var}(\hat{p}) = \sum_{h=1}^{H}\frac{W_h^2 S_h^2}{n_h} - \frac{1}{N}\sum_{h=1}^{H} W_h S_h^2, \qquad (1)$$

where S_h is the standard deviation of stratum h (i.e., of the multiset $\{\mathsf{q}(o) \mid o \in \mathcal{O}_h\}$). The $(1-\alpha)$ confidence interval for $\hat{p}$ is $\hat{p} \pm t_{\alpha/2}\sqrt{\widehat{\mathrm{Var}}(\hat{p})}$,n where $\widehat{\mathrm{Var}}(\hat{p})$ is an unbiased estimate of $\mathrm{Var}(\hat{p})$ computed using (1) with S_h^2 substituted by an unbiased estimate from $\mathcal{S}_h$. See standard sampling literature [24] for details.

A simple strategy is *proportional allocation*, where the number of samples allotted to each stratum is proportional to its size, i.e., $n_h \propto N_h$. We refer to stratified sampling with proportional allocation as SSP. A more sophisticated alternative, *Neyman allocation*, optimally allocates samples according to $n_h \propto N_h S_h$, which minimizes $\mathrm{Var}(\hat{p})$. We refer to this alternative as SSN. In practice, as we do not know S_h in advance, SSN proceeds in two stages:

1. Randomly draw a set $\mathcal{S}^1$ of samples to evaluate q with, and use them to derive an estimate of S_h for each stratum h. Then calculate the Neyman allocation using these estimates.[1]
2. Randomly draw the allotted number of samples from each stratum.

[1]Standard caveats apply: given the desired total number of samples, we ensure that no stratum is allotted more samples than it contains, and that no stratum is allotted fewer than a prescribed minimum number of samples (even if its estimated standard deviation is close to 0); we do so by rebalancing the allocation after meeting these constraints.

3.2 Learning-Based Methods

Since q is expensive to evaluate, it is natural to consider learning a binary classifier $f : \mathcal{O} \to \{0, 1\}$ to approximate the behavior of q. We can draw a random sample $\mathcal{S}$ from $\mathcal{O}$, evaluate q on them to obtain the ground truth, and then use the results to train the classifier. The classic classification problem strives to classify each input object correctly, but for our problem, we are concerned only with the *number* of objects whose ground-truth labels are 1. The resulting problem is an instance of *quantification learning* [6], whose goal is to estimate the class distribution as opposed to individual labels. While specialized algorithms are possible, it is appealing to adapt classic classification algorithms for quantification learning, thereby leveraging a rich palette of mature techniques. In this section, we first explore how, given a classifier f that approximates q, we can use quantification learning to estimate $C(\mathcal{O}, \mathsf{q})$.

We will not delve into specific classification algorithms here, because they are not this paper's focus; our methods can work with any of them. For feature selection, we use a simple heuristic that selects the attributes of o referenced in q, e.g., columns of $\mathbf{L}$ referenced by $\theta_{\mathbf{LR}}$ in (Q1) (Section 2). We also note that training can be improved by *active learning* [6]; for additional discussion, please see the full version of this paper [25].

Classify-and-Count (QLCC) A straightforward and natural approach is *Classify-and-Count* [6], which we refer to as *QLCC*. Suppose we randomly select $\mathcal{S} \subseteq \mathcal{O}$ as training data and let $C_{\mathcal{S}} = C(\mathcal{S}, \mathsf{q})$ denote the count of positive objects therein. After learning f from $\mathcal{S}$, we evaluate $f(o)$ for each "test object" $o \in \mathcal{O} \setminus \mathcal{S}$. Let $C_{\text{obs}} = \sum_{o \in \mathcal{O} \setminus \mathcal{S}} f(o)$ denote the "observed count" of f over the test data. We simply return $C_{\text{obs}} + C_{\mathcal{S}}$ as the estimate for $C(\mathcal{O}, \mathsf{q})$. Should the classifier be accurate over the test data, this estimate will be accurate as well. However, it should be clear that QLCC is susceptible to classification errors and can produce wildly skewed estimates when false positive/negative counts are imbalanced.

Adjusted Count (QLAC) To mitigate this problem, a recommended approach is *Adjusted Count* [6], which we refer to as *QLAC*. The basic idea is to further adjust C_{obs} using the rates of true and false positives estimated empirically from the training data. In more detail, we use k-fold cross validation on the samples $\mathcal{S}$ to compute $\widehat{tpr}$ and $\widehat{fpr}$, the estimated true and false positive rates, respectively. Then, we obtain an "adjusted count" C_{adj} of f over the test data by adjusting the observed count C_{obs} as follows[2]:

$$C_{\text{adj}} = \frac{C_{\text{obs}} - \widehat{fpr} \cdot |\mathcal{O} \setminus \mathcal{S}|}{\widehat{tpr} - \widehat{fpr}}. \tag{2}$$

Finally, we return $C_{\text{adj}} + C_{\mathcal{S}}$ as the estimate for $C(\mathcal{O}, \mathsf{q})$.

3.3 Learning with Sample-based Correction

One idea for combining learning and sampling is to follow QLCC (Classify-and-Count) with another phase, where we randomly sample additional objects, evaluate q on them, assess the errors in the learned classifier f, and correct the result of Classify-and-Count accordingly. We call this method **QLSC**, for "quantification learning with *SampleClean*," as it is inspired by the work of [26] on using sampling for data cleaning.[3] More precisely, recall that QLCC samples $\mathcal{S} \subset \mathcal{O}$, learns f, and estimates the positive count over remaining objects as $C_{\text{obs}} = \sum_{o \in \mathcal{O} \setminus \mathcal{S}} f(o)$. QLSC then proceeds with drawing (uniformly at random) another set $\mathcal{S}'$ of objects from $\mathcal{O} \setminus \mathcal{S}$, and for each $o \in \mathcal{S}'$ computes the error $f(o) - \mathsf{q}(o)$. The average error $\hat{\epsilon}$ over $\mathcal{S}'$ gives an unbiased estimator for the average error over $\mathcal{O} \setminus \mathcal{S}$, so we can correct the count over $\mathcal{O} \setminus \mathcal{S}$ as $C_{\text{obs}} - \hat{\epsilon}|\mathcal{O} \setminus \mathcal{S}|$. Adding $C_{\mathcal{S}}$ (positive count in $\mathcal{S}$) yields the overall estimate. Confidence intervals can be derived as in Section 3.1 because the second phase of QLSC is basically SRS.

QLSC is similar to QLAC (Section 3.2) in that both seek to correct the result of QLCC by assessing its errors on labeled samples. However, QLAC produces only a point estimate while QLSC can provide confidence intervals.

4. LEARNING-TO-SAMPLE METHODS

In the previous section, we have seen how sampling and learning can be applied to problem of estimating $C(\mathcal{O}, \mathsf{q})$. Learning is attractive for its ability to "generalize" knowledge of q to unsampled objects, but it does not offer the guarantees provided by sampling (e.g., confidence intervals), and its accuracy depends heavily on the quality of the classifier it learns. A natural question is whether we can combine learning and sampling to get the "best of both worlds." QLSC (Section 3.3) represents a baseline approach towards this goal: it uses sampling to correct the count predicted by the classifier, but its sampling scheme does not take advantage of the learned model in any way, and a poor classifier would result in a poor starting point.

This section proposes two methods that combine learning and sampling more effectively. Both methods proceed in two phases. ***The first phase is learning***, and is identical for the two methods: we randomly sample objects, evaluate q on them, and train a binary classifier, as we did in Section 3.2. However, we are not going to use this classifier to get a count (as a starting point or otherwise). Instead, we assume that the classifier provides a *scoring function* $g : \mathcal{O} \to [0, 1]$: if $g(o) = 1$ (or 0), the classifier is totally confident in predicting $\mathsf{q}(o)$ to be 1 (or 0, resp.); a value strictly between 0 and 1, on the other hand, indicates uncertainty (e.g., 0.5 means a toss-up). For some classifiers (e.g., random forest), one can intuitively interpret $g(o)$ as the probability that $\mathsf{q}(o) = 1$, but in general, $g(o)$ may not have a probabilistic interpretation. Regardless, the scoring function g gives us a way to gauge the certainty in the predicted labels. We assume that, compared with q, g is cheap to evaluate (in practice it is often a byproduct of classification).

The second phase is sampling, but differs between the two methods. The first method, *Learned Weighted Sampling* (*LWS*), is the more straightforward one of the two. Treating $g(o)$ has a guess of how much each object o contributes to $C(\mathcal{O}, \mathsf{q})$, LWS samples objects with higher $g(o)$ with higher probability. The second method, *Learned Stratified Sampling* (*LSS*), uses g to guide the partitioning of objects into strata, with the goal of reducing the variance of estimates using stratified sampling.

[2]To see why, note that the proportion $\hat{p}$ of "observed positive" objects in the test data can be computed by $\hat{p} = p \cdot tpr + (1-p) \cdot fpr$, where p denotes the actual positive proportion, and tpr and fpr are the true and false positive rates. We can solve for p, and note that multiplying $\hat{p}$ and p by the size of the test data yields the observed and actual counts. Replacing tpr and fpr with their estimates then gives us (2).

[3]While *SampleClean* [26] deals with the different problem of evaluating aggregates over dirty data, its techniques can be adapted to our quantification learning setting by conceptually regarding the labels produced by the learned classifier as dirty data; "cleaning" a dirty label involves sampling the object and paying the cost of evaluating q. Specifically, QLSC corresponds to their *NormalizedSC* technique, which corrects the aggregate result computed over dirty data using the errors observed on data randomly selected for cleaning. Their *RawSC* technique, which randomly cleans data and estimates the result from only the cleaned labels, basically corresponds to the sampling-based baseline methods in our Section 3.1.

The novelty of these two methods lies in their use of learning to inform sampling. Thanks to sampling, we still get accuracy guarantees in the form of confidence intervals. At the same time, we get the benefit of learning without relying on it for correctness. A good classifier leads to more efficient sampling designs; on the other hand, a poor classifier leads to a less efficient sampling design, but we still have unbiased estimates with confidence intervals. As we will see, between the two methods, LSS is even more robust and less dependent on the quality of the learned classifier than LWS.

The remainder of this section describes the second phase for these two methods in detail. Let $\mathcal{S}^{\mathrm{L}}$ denote the samples used in the first phase for learning a classifier with scoring function g. We now focus on estimating $C(\mathcal{O} \setminus \mathcal{S}^{\mathrm{L}}, \mathsf{q})$ in the second phase. In the following, we will abuse notation for simplicity: we shall refer to $\mathcal{O} \setminus \mathcal{S}^{\mathrm{L}}$ simply as $\mathcal{O}$ instead, and let $N = |\mathcal{O}|$.

4.1 Learned Weighted Sampling

The second phase of LWS can be seen as a form of *probability-proportional-to-size* (*PPS*). In general, PPS relies on a "size measure" that is believed to be correlated to the variable of interest. Objects with large size measures are deemed more important in estimation; hence, objects are drawn with probabilities proportional to their size measures. In our case, the variable of interest is the result of $\mathsf{q}(o)$, so the learned $g(o)$ can serve as the size measure. However, to guard against an overconfident (and potentially inaccurate) classifier, we adjust the sampling probabilities so every o has some chance of being sampled (even if $g(o) = 0$). Specifically, we assign each o an initial sampling probability $\pi(o) \propto \max(g(o), \epsilon)$, where $\epsilon > 0$ is a (small) prescribed threshold. We then sample objects from $\mathcal{O}$ according to π without replacement, evaluate q on the sampled objects, and estimate $C(\mathcal{O}, \mathsf{q})$.

There are a number of estimators available from the literature [14], including the popular Horvitz-Thompson estimator. We use the Des Raj estimator, whose calculation is simpler and can provide "ordered" estimates, i.e., running estimates of mean and variance as samples are being drawn. Let $o_1, o_2, o_3 \ldots$ denote the sequence of objects drawn according to π without replacement. We compute the following quantity after drawing each o_i (with the summations below yielding 0 in case of $i = 1$):

$$p_i = \tfrac{1}{N}\left(\sum_{j=1}^{i-1} \mathsf{q}(o_j) + \tfrac{\mathsf{q}(o_i)}{\pi(o_i)}\left(1 - \sum_{j=1}^{i-1} \pi(o_j)\right)\right). \quad (3)$$

The estimate for $C(\mathcal{O}, \mathsf{q})$ after drawing the n-th sampled object would be $\hat{p}^{(n)} N$, where the estimated proportion $\hat{p}^{(n)}$ of positive objects is simply the average of all p_i's so far:

$$\hat{p}^{(n)} = \tfrac{1}{n} \sum_{i=1}^{n} p_i.$$

And the variance in $\hat{p}^{(n)}$ can be estimated as follows:

$$\widehat{\mathrm{Var}}(\hat{p}^{(n)}) = \tfrac{1}{n(n-1)} \sum_{i=1}^{n} (p_i - \hat{p}^{(n)})^2.$$

LWS is very efficient when the learned classifier is accurate and confident. To see why, suppose the true proportion of positive objects in $\mathcal{O}$ is p. For an accurate and confident classifier, assuming an arbitrarily small ϵ, $\pi(o)$ would be arbitrarily close to 0 if $\mathsf{q}(o) = 0$, or $\frac{1}{pN}$ otherwise. Therefore, each sampled object o_i will have $\mathsf{q}(o_i) = 1$ and $\pi(o_i) = \frac{1}{pN}$. Plugging these into (3) and simplifying the equation yields $p_i = p$ for all i, so the estimate $\hat{p}^{(i)}$ at every step will be perfectly accurate.

On the other hand, LWS's efficiency can suffer with a poor classifier. Even though it still produces unbiased estimates (regardless of the choices of $\pi(o)$'s), it may require many more samples to achieve a tight confidence interval if it gets the priorities wrong.

Another indication that LWS may not be best for our setting is its preference for objects with high $g(o)$. Intuitively, focusing instead on objects with $g(o)$ in the toss-up range reveals more information. Note that traditionally, PPS applies to the more general setting where the variable of interest can be of any value; hence, it is natural to focus on objects with potentially higher contribution to the result. In our setting, however, the value of interest, $\mathsf{q}(o)$, is either 0 or 1. This limited range makes our problem easier, as we do not need to worry about cases where inclusion or exclusion of objects with extremely high values can seriously impact the estimates. At the same time, this more constrained setting also enables the possibility for better sampling designs, which we explore next.

4.2 Learned Stratified Sampling

As discussed in Section 4.1, the quality of the learned classifier can adversely impact the efficiency of LWS, because the values of scoring function g directly control the sampling probabilities. We now present LSS, which uses g more conservatively, and in a way that naturally encourages exploration of uncertain outcomes (as opposed to certain positives).

Following the learning phase, LSS conceptually sorts the objects in $\mathcal{O}$ by g (say, in increasing score order). At a high level, LSS applies stratified sampling to $\mathcal{O}$, where stratification is done according to this ordering; i.e., each stratum covers objects whose g scores fall into a consecutive range. More specifically, the second phase of LSS proceeds in two stages:

1. Randomly draw $\mathcal{S}^{\mathrm{I}} \subseteq \mathcal{O}$ to evaluate q, and use the results to design a sampling scheme for the second stage—namely, the partitioning of $\mathcal{O}$ into strata as well as an allocation of second-stage samples among these strata.
2. Randomly draw $\mathcal{S}^{\mathrm{II}} \subseteq \mathcal{O} \setminus \mathcal{S}^{\mathrm{I}}$ to evaluate q, according to the sampling scheme designed by the first stage, and use the results to estimate $C(\mathcal{O}, \mathsf{q})$.

Several points are worth noting:

(*Versus LWS*) While LWS uses the actual g values in its sampling design, LSS uses only the *ordering* of g values among objects. Hence, LSS relies less on the learned classifier. We will validate this observation with experiments in Section 5. On the other hand, the ordering induced by g is useful to LSS because it intuitively brings together objects with similar levels of uncertainty, and in particular encourages putting the confidently positive objects and confidently negative objects into separate strata with low within-stratum variances.

(*Versus Basic Stratified Sampling*) While the second phase of LSS uses stratified sampling, this phase differs from the baseline methods in Section 3.1 in important ways: (i) stratification in LSS is based on the learned g instead of surrogate object attributes; (ii) LSS uses $\mathcal{S}^{\mathrm{I}}$ to jointly design stratification and allocation; in contrast, SSN only uses $\mathcal{S}^{\mathrm{I}}$ to design allocation (given stratification), while SSP does not have a first stage.

(*Samples in Learning and Sampling Phases*) The samples we draw in the sampling phase of LSS ($\mathcal{S}^{\mathrm{I}} \cup \mathcal{S}^{\mathrm{II}}$ above) are separate from those drawn in the learning phase. Since the samples from the learning phase already affect (through the learned g) the ordering of $\mathcal{O}$ for stratification, we choose to use new, independent samples ($\mathcal{S}^{\mathrm{I}}$) for sampling design in order to minimize reliance on the classifier quality.[4]

The remainder of this section discusses how we design the sampling scheme for the second stage in detail. Formally, we define the design problem as follows. Consider an ordered set $\mathcal{O}$ of objects $o_1, o_2, \ldots, o_N$ ordered by g with ties broken arbitrarily,

[4]As future work, it would be interesting to investigate safe reuse of samples from the learning phase in less conservative ways.

which can be efficiently computed as we assume that the classifier is easy to execute. A stratification of $\mathcal{O}$ into H strata, specified by $(N_1, N_2, \ldots, N_H)$ where $\sum_{h=1}^{H} N_h = N$, defines the partitioning of $\mathcal{O}$ into subsets $\mathcal{O}_1, \mathcal{O}_2, \ldots, \mathcal{O}_H$. Here $\mathcal{O}_1$ includes objects with indices $\leq N_1$, and $\mathcal{O}_h, h \geq 2$ denotes the subset of objects whose indices fall within the interval $(\sum_{j=1}^{h-1} N_j, \sum_{j=1}^{h} N_j]$. Recall from Section 3.1 that (1) gives the variance in the estimator of $C(\mathcal{O}, \mathsf{q})/N$ for stratified sampling, given the stratification $(N_1, N_2, \ldots, N_H)$ and a sample allocation $(n_1, n_2, \ldots, n_H)$ where we draw n_h objects from $\mathcal{O}_h$. However, we do not know the S_h terms in (1) in advance, since they denote the standard deviation of the actual $\mathsf{q}(o_i)$ values of the objects $o_i \in \mathcal{O}_h$ that are expensive to compute, so LSS instead seeks to minimize the variance of $C(\mathcal{O}, \mathsf{q})$ given by (1) estimated using the first-stage samples $\mathcal{S}^{\mathrm{I}}$.

More precisely, suppose the first-stage sample $\mathcal{S}^{\mathrm{I}}$ consists of m objects $o_{\imath_1}, o_{\imath_2}, \ldots, o_{\imath_m}$ where $1 \leq \imath_1 < \imath_2 < \cdots < \imath_m \leq N$. We aim to find a stratification $(N_1, N_2, \ldots, N_H)$ to minimize the objective given in (5) below that estimates the variance in the estimator of $C(\mathcal{O}, \mathsf{q})$ using n samples in total in the second stage. Here we assume $\mathcal{S}_h^{\mathrm{I}} = \mathcal{O}_h \cap \mathcal{S}^{\mathrm{I}}$, $m_h = |\mathcal{S}_h^{\mathrm{I}}|$, n_h is number of second-stage samples in $\mathcal{O}_h$, $\sum_{h=1}^{H} n_h = n$, and the variances S_h^2 using the first-stage samples $\mathcal{S}^{\mathrm{I}}$ are estimated as

$$s_h^2 = \tfrac{1}{m_h - 1} \sum_{o \in \mathcal{S}_h^{\mathrm{I}}} (\mathsf{q}(o) - C(\mathcal{S}_h^{\mathrm{I}}, \mathsf{q})/m_h)^2. \quad (4)$$

Then the variance of the estimated $C(\mathcal{O}, \mathsf{q})$ obtained by simplifying (1) is given by:

$$V(N_1, N_2, \ldots, N_H) = \sum_{h=1}^{H} \tfrac{N_h^2 s_h^2}{n_h} - \sum_{h=1}^{H} N_h s_h^2. \quad (5)$$

The remainder of this section describes our algorithms for computing the optimal stratification given $\mathcal{S}^{\mathrm{I}}$. Note that the optimality of stratification depends on the allocation strategy used. We first present the case of using Neyman allocation, which minimizes the variance for a given stratification. In this case, LSS gives the overall optimal sampling design that jointly considers stratification and allocation. Then, we briefly discuss the case of proportional allocation, which is simpler but not optimal for a given stratification. In this case, we would find the stratification that makes proportional allocation most effective; the optimization problem is much easier than the case of Neyman allocation.

Optimizing the Stratification

Recall from Section 3.1 that under Neyman allocation using $\mathcal{S}^{\mathrm{I}}$, $n_h = n(N_h s_h)/(\sum_{h=1}^{H} N_h s_h)$. Hence, we can further simplify (5), the minimization objective, as follows:

$$V(N_1, N_2, \ldots, N_H) = \tfrac{1}{n} \left(\sum_{h=1}^{H} N_h s_h \right)^2 - \sum_{h=1}^{H} N_h s_h^2. \quad (6)$$

A naive algorithm would compute V for all possible stratifications $(N_1, N_2, \ldots, N_H)$ and pick the best, but the number of possibilities is $\Omega(N^H)$, and computing V involves going over $\mathcal{S}^{\mathrm{I}}$, which is expensive even for small number of partitions (e.g., when $H = 3$).

Before presenting our algorithms, we describe some ideas useful to combat these challenges. *First*, note that in the expression for V in (6), from (4), the s_h terms depend only on the subset of objects $\mathcal{S}_h^{\mathrm{I}}$ sampled in $\mathcal{S}^{\mathrm{I}}$ in each stratum h, and the precise locations of stratum boundaries between these sampled points only affect the N_h terms. This observation suggests that we may be able to first consider the partitioning of $\mathcal{S}^{\mathrm{I}}$ among strata, and then decide where precisely the stratum boundaries lie among $\mathcal{O}$. Later in this section, we will start with an algorithm that uses this strategy, where given the partitioning of $\mathcal{S}^{\mathrm{I}}$, the optimal N_h's can be solved directly and (almost) exactly in the case of $H = 3$. Building on the insights revealed in this simple case, we then present two general algorithms for any H providing different trade-offs between speed and accuracy. Both of these algorithms tame complexity by restricting the potential locations of the stratum boundaries.

Second, we can speed up the computation of V significantly using precomputation. By sorting the m objects in $\mathcal{S}^{\mathrm{I}}$ by g, we can compute a prefix-sum index Γ, such that $\Gamma(k) = \sum_{j=1}^{k} \mathsf{q}(o_{\imath_j})$ (for $1 \leq k \leq m$) returns the number of positive objects among the first k objects in $\mathcal{S}^{\mathrm{I}}$. To obtain the indices of sampled objects within the ordered $\mathcal{O}$ (i.e., $\imath_1, \ldots, \imath_m$), there is no need to sort all objects in $\mathcal{O}$ by g. Instead, note that the m objects in $\mathcal{S}^{\mathrm{I}}$ divide the range of g values into $m + 1$ buckets; we can simply make one pass over $\mathcal{O}$ and maintain the count of objects whose g values fall within each bucket. After the pass over $\mathcal{O}$ completes, we scan the bucket counts to determine $\imath_1, \ldots, \imath_m$.

- **DirSol (an almost optimal stratification for $H = 3$)**: Here we try all pairs of $\mathcal{S}^{\mathrm{I}}$ as possible *rough* boundaries. In particular, for each pair of consecutive samples as per g, we assume that the first element is the last sampled object in the first strata, while the second element is the first sampled object in the third strata. In order to find the exact boundaries in $\mathcal{O}$, we formulate and solve an optimization problem.
- **LogBdr (an approximate stratification for any H, generalizing DirSol)**: It considers all possible ways of partitioning the m sampled objects in $\mathcal{S}^{\mathrm{I}}$ among H strata generalizing the ideas in DirSol. Unlike DirSol, however, for each such partitioning, we do not attempt to solve directly for the actual stratum boundaries within $\mathcal{O}$; instead, we consider only a set of candidate boundary indices, chosen judiciously to ensure that we can still find a reasonably good solution. In particular, between two consecutive objects $o_{\imath_k}$ and $o_{\imath_{k+1}}$ in $\mathcal{S}^{\mathrm{I}}$ (with respect to g), we consider the objects in $\mathcal{O}$ that are 2^i apart from $o_{\imath_k}$ as boundary indices.
- **DynPgm (a dynamic-programming-based algorithm for any H, faster than LogBdr but with worse approximation guarantees)**: A straightforward application of dynamic programming does not work since the objective in (6) is *non-separable*. To overcome this difficulty, we isolate the non-separable term in the objective function and solve a suite of dynamic programs where each of them operates under a different upper bound on the non-separable term. In order to improve the running time, we only consider as possible boundaries the set $\mathcal{S}^{\mathrm{I}}$ and the additional boundary indices similar to DirSol. In the end, we return the best result over the dynamic programs (details in [25]).
- **DynPgmP (2-approximation for proportional allocation)**: Recall from Section 3.1 that under proportional allocation, $n_h = nN_h/N$. Hence, we can further simplify (5) to $V(N_1, \ldots, N_H) = \frac{N-n}{n} \sum_{h=1}^{H} N_h s_h^2$. The objective is much simpler than the objective for Neyman allocation and the resulting optimization problem is indeed separable, so it can be solved readily by dynamic programming. To improve the efficiency, we use the same idea as in LogBdr and DynPgm with additional boundary indices (details in [25]).

In addition to optimizing the objective in (6), we impose the following constraints for each stratum h: For two chosen thresholds $N_{\llcorner\lrcorner}$ and $m_{\llcorner\lrcorner}$, (i) $N_h \geq N_{\llcorner\lrcorner}$, i.e., each stratum is large enough, and (ii) $m_h \geq m_{\llcorner\lrcorner}$, i.e., the stratum contains enough first-stage samples such that s_h is a reasonable variance estimate. In practice, we have set $m_{\llcorner\lrcorner}$ to be around 5 and $N_{\llcorner\lrcorner}$ larger.

DirSol: We now give more details on DirSol. For $H = 3$, we need to pick two boundaries separating strata $\mathcal{O}_1$, $\mathcal{O}_2$, $\mathcal{O}_3$. To this end, suppose the last sampled object (with the largest g value) in $\mathcal{O}_1$ is the i-th object in $\mathcal{S}^{\mathrm{I}}$, and the first sampled object (with the smallest g value) in $\mathcal{O}_3$ is the j-th object in $\mathcal{S}^{\mathrm{I}}$. The algorithm considers every possible (i, j) pair where $m_{\sqcup} \leq i < i + m_{\sqcup} < j \leq m - m_{\sqcup} + 1$.

Given $o_{\imath_i}$ as the last sampled object in $\mathcal{O}_1$ and $o_{\imath_j}$ as the first sampled object in $\mathcal{O}_3$, we can readily compute s_1, s_2, s_3 in (6) using the precomputed index Γ: $s_1^2 = \frac{\Gamma(i)}{i-1}\left(1 - \frac{\Gamma(i)}{i}\right)$, $s_2^2 = \frac{\Gamma(j-1)-\Gamma(i)}{j-i-2}\left(1 - \frac{\Gamma(j-1)-\Gamma(i)}{j-i-1}\right)$, and $s_3^2 = \frac{\Gamma(m)-\Gamma(j-1)}{m-j}\left(1 - \frac{\Gamma(m)-\Gamma(j-1)}{m-j+1}\right)$.

Then, noting that $N_2 = N - N_1 - N_3$, we can write $V(N_1, N_2, N_3)$ as bivariate quadratic function $f(N_1, N_3)$ of the form $a_1N_1^2 + a_2N_3^2 + a_3N_1N_3 + a_4N_1 + a_5N_3 + a_6$, where coefficients $a_1, \ldots, a_6$ are computed from $s_1, s_2, s_3, n,$ and N (see [25] for detailed derivation). Our goal is to minimize $f(N_1, N_3)$ subject to the following constraints:

- $\max\{N_{\sqcup}, \imath_i\} \leq N_1 \leq \imath_{i+1} - 1$; i.e., the last sampled object in $\mathcal{O}_1$ is indeed the i-th one in $\mathcal{S}^{\mathrm{I}}$, and $|\mathcal{O}_1| \geq N_{\sqcup}$.
- $\max\{N_{\sqcup}, N - \imath_j + 1\} \leq N_3 \leq N - \imath_{j-1}$; i.e., the first sampled object in $\mathcal{O}_3$ is the j-th in $\mathcal{S}^{\mathrm{I}}$, and $|\mathcal{O}_3| \geq N_{\sqcup}$.
- $N_1 + N_3 \leq N - N_{\sqcup}$; i.e., $|\mathcal{O}_2| \geq N_{\sqcup}$.

These constrains define a 2-dimensional polygon R with at most 5 sides. We optimize the function f over R using a standard algebraic method by considering (i) the critical points of f, and (ii) the boundary of R. We repeat the above procedure for each pair of sampled objects, and in the end return the stratification with the overall minimum variance (see [25] for additional details and the pseudocode). We call this algorithm *DirSol* (for direct solve). The following theorem summarizes its time complexity and accuracy.

THEOREM 1. *Given an ordered set $\mathcal{O}$ of N objects and a sampled subset $\mathcal{S}^{\mathrm{I}}$ of m objects, let v^* denote the minimum value of estimated variance defined in* (6) *achievable using n samples under stratified sampling with $H = 3$ strata where each stratum contains at least $N_{\sqcup}$ objects. Assuming $N_{\sqcup} > n$, DirSol runs in $O(N \log m + m^2)$ time and finds a stratification resulting in estimated variance $v \leq (1 + \frac{2}{N_{\sqcup}} + \frac{2}{N_{\sqcup}-n} + \frac{4}{N_{\sqcup}(N_{\sqcup}-n)})v^*$.*

Note the assumption of $N_{\sqcup} > n$ above; without it, the approximation factor would be arbitrarily bad. In practice, however, this assumption is weak and often holds in practice: e.g., if we take a 5% sample of $\mathcal{O}$ in the second stage, this assumption means that each stratum in $\mathcal{O}$ contains at least 5% of $\mathcal{O}$.

LogBdr: Given a partitioning of the sampled objects, consider two consecutive sampled objects $o_{\imath_k}$ and $o_{\imath_{k+1}}$ that are put into different strata (there are $H - 1$ such pairs of objects). When deciding where exactly to draw the boundary between $o_{\imath_k}$ and $o_{\imath_{k+1}}$, the algorithm only considers the set B_k of candidate boundary indices $\imath_k, \imath_k + 2^0, \imath_k + 2^1, \imath_k + 2^2, \ldots$ up to (but not including) $\imath_{k+1}$; we also add $\imath_{k+1} - 1$ if it is not already in B_k. Choosing a particular index i from B_k means the stratum containing $o_{\imath_k}$ ends with o_i. Then we simply check all candidate stratifications formed by choosing one index from each of the $H - 1$ sets of candidate boundary indices. We call this algorithm *LogBdr* (for logarithmic number of candidate boundary indices). Theorem 2 summarizes its time complexity and accuracy (proof is in [25], the approximation factor can be improved if we increase the running time).

THEOREM 2. *Given an ordered set $\mathcal{O}$ of N objects and a sampled subset $\mathcal{S}^{\mathrm{I}}$ of m objects, let v^* denote the minimum value of estimated variance defined in* (6) *achievable using n samples under stratified sampling with H strata where each stratum contains at least $N_{\sqcup}$ objects. Let N_h^* denote the size of stratum h in this optimum solution. Assuming $N_{\sqcup} > n$, LogBdr runs in $O(N \log m + Hm^{H-1} \log^{H-1} N)$ time and finds a stratification resulting in estimated variance $v \leq \max\{4,\ 2 + 2\max_{1 \leq h \leq H} \frac{N_h^*}{N_h^*-n}\}v^*$.*

5. EXPERIMENTS

Most of our experiments are based on three scenarios, each with its own real-world dataset and counting query template:

(Sports) The data contains yearly performance statistics for players in the Major League Baseball. We focus on pitching statistics, which exclude a portion of the players. We consider the k-skyband size query in Example 2, where each point is a player-year combination (there are about 47,000 of them), and x and y refer to runs and home runs.

(Neighbors) The data comes from KDD Cup 1999, where the goal was to learn a predictive model that could distinguish legitimate and illegitimate (intrusion attacks) connections to a machine. The original dataset contains 4.9 million records with 41 features and a binary label. We removed many sparse rows, resulting in 73,000 points. We consider the query in Example 1 that counts points with few neighbors.

(Text) We consider the relevant document count query in Example 3. Since we do not want to manually evaluate the predicate ourselves in experiments, we use the *LSHTC* dataset [23], which provides ground-truth labels (Wikipedia categorization) for 2.4M documents from Wikipedia. The same dataset was used in [18]. In our experiments, each algorithm is charged a cost for revealing the true label, which in practice would be expensive.

To experiment with different selectivities of the predicate q, we adjust query parameter settings (k for *Sports*; k and d for *Neighbors*; k for *Text*). We also create synthetic datasets based on *Sports* to study how data distributions affect learned models and the performance of various algorithms; for details see Section 5.2.

We compare the following algorithms:

- Sampling-based (Section 3.1): simple random sampling (SRS) and stratified sampling (SSP, with proportional allocation, and SSN, with Neyman allocation in two stages). For stratified sampling (which applies to *Neighbors* and *Sports* but not to *Text*), we use attributes x and y as surrogates; each stratum is a rectangle in the 2d x-y space. Unless otherwise specified, we stratify using a uniform $\sqrt{H} \times \sqrt{H}$ grid over the ranges of x and y values in the dataset. By default $H = 4$.
- Learning-based (Section 3.2): quantification learning (QLCC, without adjustment, and QLAC, with adjustment).
- Learning with sampling-based correction (Section 3.3): QLSC.
- Learning-to-sample (Section 4): learned weighted sampling (LWS) and learned stratified sampling (LSS). Unless otherwise specified, for LSS we implement a simplified version of LogBdr, which considers candidate boundaries that map to equally spaced ticks over $[0, 1]$ (the range of g scores). By default, $H = 4$ and the spacing between candidate boundaries is 0.05; for the distributions of g scores that arise in practice, these boundaries already provide fine enough resolution for $H = 4$, so more sophisticated choices of candidates in LogBdr are not needed.

For learning-based and learn-to-sample algorithms, we use standard implementations of classifiers from `scikit-learn`. For *Neighbors* and *Sports*, we experiment with kNN (k-nearest neighbors, where k is not to be confused with our query parameter), RF (random forests), and NN (a simple two-layer neural network); by default, we use RF with 100 estimators. For *Text*, we use a naive Bayes classifier with standard full-text features. For QLSC, LWS, and LSS, by default we devote 25% of their allotted samples to training (and including design, if applicable).

Since the estimates of result counts are uncertain, for each experimental setting, we run each algorithm 100 times, and record the distribution of estimates it produces. Recall that unlike sampling-based and learn-to-sample algorithms, those based on learning alone provide no accuracy guarantees by themselves. Nonetheless, the distributions of estimates they produce allow us to evaluate their accuracy empirically. When appropriate, we show distributions using violin plots[5]. We would like our estimates to be unbiased, so ideally the violin plots would be centered around the actual result count. Furthermore, we would like the estimates to have low variance, which means narrower interquartile ranges as well as shorter and wider plots. In some figures, we use MAE (mean absolute error) as a single numeric measure to quantify and summarize an error distribution, so we can report more results than violin plots.

For *Neighbors* and *Sports*, while our queries can be executed directly over a database system, they run slowly even if we construct all appropriate standard indices and enable the maximum level of optimization (on PostgreSQL and another commercial system). To enable faster experiments, we implemented the evaluation of q in Python in main memory. Since our experiments specify sampling budgets in terms of numbers (or percentages) of samples, our results are platform-neutral and easy to translate into time savings on different underlying platforms. The overhead of learning, as we will show later with experiments, is small compared to the cost of labeling samples (evaluating q), even for the in-memory Python implementation; the overhead will be even smaller in the SQL setting.

5.1 Overall Comparison with Real Datasets

We begin with experiments that compare various algorithms using the three scenarios with real datasets, *Neighbors*, *Sports*. and *Text*. Both LSS and LWS used a random forest classifier with estimators and a 25%:75% training:sampling split. Figure 1 compares the MAE of various algorithms when we vary the result size (via query parameters) while keeping the sample size fixed. Figure 2 compares the MAE of various algorithms when we vary the sample size while keeping the result size fixed.

As it turns out, the learned classifier performs pretty well for *Neighbors* and *Sports*, but pretty poorly for *Text*, leading to very different results. We shall focus on *Neighbors* and *Sports* first. F1 scores for the learned classifiers average higher than 0.8 in these scenarios (with small result sizes being more difficult). We make several observations. **First**, learning-based methods are very competitive here thanks to high classifier quality. In fact, QLCC sometimes even delivers the smallest errors even without any adjustment or correction. But to keep things in perspective, QLCC and QLAC do not provide any guarantees; once QLSC uses sampling to provide correction and guarantees, MAE actually takes a small hit because of the extra overhead. **Second**, algorithms without any learning component, namely SRS and SSP are clearly not as competitive here, with much higher MAE than others. **Third**, LSS (highlighted) has consistently low MAE; it is nearly always the leader or not far from the leader, and bear in mind that it offers statistical guarantees, which QLCC does not. LSS also consistently leads QLSC by a good margin. **Fourth**, the comparison between LWS and LSS is difficult, as in some cases LWS leads LSS. The quality of the learned classifier for *Neighbors* and *Sports* is the main factor here. To better understand the situation, we take a closer look at some data points with violin plots showing distributions.

In Figure 3, we get a more detailed sense of the variability in estimates. LSS and LWS are consistently no worse and often better than SRS and SSP. Between LSS and LWS, we make two observations. **First**, when selectivity is low, we expect all sampling-based methods to have some trouble as the particular number of positives that come up by chance in each run will have a large impact on relative error. For *Sports*, LWS dodged this issue with a very good classifier that allows it to draw in a very targeted fashion. In contrast, LSS, as it places much less trust in the learned model compared with LWS, misses the opportunity. **Second**, LWS is not without its own problems. In *Neighbors*, where prediction becomes slightly more challenging, we see LWS underestimating with XS result size; as it turns out, the classifier at those points happens to generate more false negatives. In other words, LWS depends far more on model quality than LSS does—it can benefit more, but also can get hurt more. This effect will be magnified for the *Text* scenario, which we focus on next.

The *Text* tells a completely different story. In this case, classification is hard. Therefore, QLCC, QLAC, and QLSC fare very poorly here, because their performance is too dependent on starting point produced by QLCC. Correction is also difficult. From one representative run (with 857k resize size), true TPR and FPR are .53 and .85, while the estimated TPR and FPR are .35 and .95. Even with sampling-based correction, QLSC still underperforms other algorithms. In contrast, SRS, which does not use learning, actually shines here. Finally, LSS tracks SRS closely. It actually underperforms SRS a bit, which is understandable because learning phase is essentially not that useful, wasting 25% of the samples. However, the impact on the sampling design is limited. Closer examination reveals that it basically degenerates to SRS for the remaining 75% of the samples. This experiment highlights the sensitivity of QLCC, QLAC, and QLSC toward poor models, as well as the resiliency of LSS against poor models.

5.2 Comparison with Synthetic Datasets

Results in Section 5.1 show just three data points along the spectrum of classifier quality: *Neighbors* and *Sports* have good classifiers but *Text* has a bad one. What happens in between? To understand how different algorithms are affected by varying degrees of difficulty in using a learned model to approximate a predicate, we design our next set of experiments by injecting additional "noise" into the *Sports* scenario to adjust the difficulty of classification. Recall from Example 2 that for each object o, we compute a count subquery with $o.\mathtt{x}$ and $o.\mathtt{y}$, and compare the resulting count, say c, with k. Now, we create an additional "noise" table keyed on distinct $(\mathtt{x}, \mathtt{y})$ values, where each $(\mathtt{x}, \mathtt{y})$ is associated with a noise count drawn randomly from another distribution. Instead of comparing c with k, we use another subquery to look up the noise count c' for $(o.\mathtt{x}, o.\mathtt{y})$, and have the predicate combine the original and noise counts into $(1 - \alpha)c + \alpha c'$ to compare with k. By adjusting $\alpha \in [0, 1]$, we control how much noise contributes to the outcome of the predicate: $\alpha = 0$ corresponds to the original *Sports* scenario, where we know we can learn a good model; $\alpha = 1$ means the predicate is simply comparing independent random noise, which is mostly challenging to predict.

We experiment with two noise distributions. One is a Gaussian with standard deviation of 1 truncated and discretized. The other is derived from a Zipf distribution with parameter s, where each draw is used to index into a randomly permuted array of possible noise counts derived from the real count values; large s means some (random) noise count will be far more popular than others.

We compare SRS, QLSC, and LSS, representing sampling-based, learning-based (but with sampling-based correction), and learn-to-sample algorithms, respectively. Figure 4 shows how they compare in terms of MAE when we vary α for synthetic datasets generated using Gaussian noise. Note that when α increases, the result size tends to decrease (but it is random depending on the

[5]A violin plot shows the probability density at different values; additionally, a white dot marks the median of all data, a thick black line spans the lower and upper quartiles.

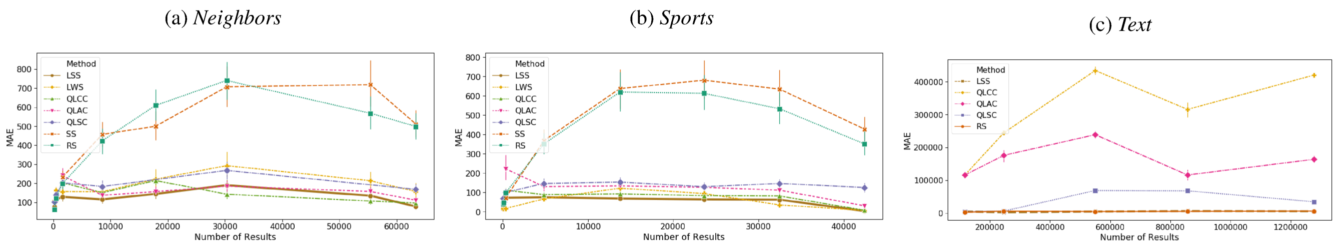

Figure 1: Mean absolute error comparison when varying result size; sample size fixed at 2%.

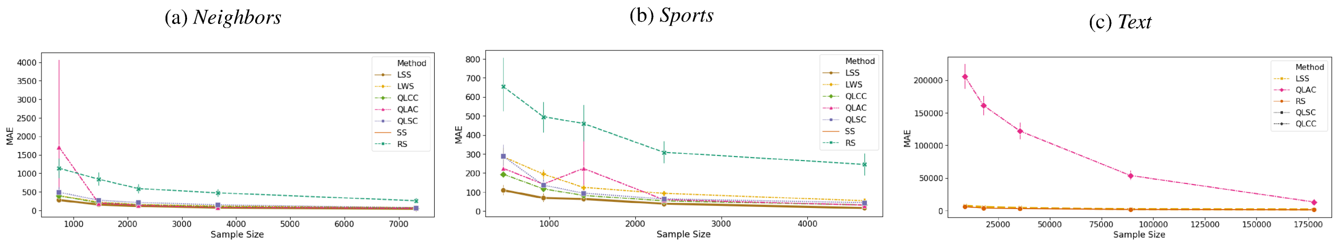

Figure 2: Mean absolute error comparison when varying sample size.

particular dataset being generated), so MAE for SRS tends to decrease accordingly, although its relative error actually increases. The main observation from this figure is that when α is small, good model qualities make LSS and QLSC outperform SRS. However, as α increases, model quality starts to take a toll on LSS and QLSC. Nonetheless, LSS consistently outperforms QLSC, and it is not too far behind SRS even when the predicate outcome is almost completely dictated by noise. Upon closer examination, we see that when $\alpha = 1$, LSS basically degenerates to random sampling in the sampling phase, and it is not surprising that it is slightly worse than SRS because it has wasted 25% of its samples on learning.

Figure 5 shows how the three algorithms compare when we vary the Zipf parameter for synthetic datasets generated using Zipf noise. The results can be difficult to interpret because of the variability in each particular instance of the randomly generated dataset, and the fact that skewness does not necessarily make classification harder. However, once we overlay the quality (F1 score) of learned classifier for QLSC and LSS, a clear pattern emerges: model quality clearly influences the performance of methods that use learning, but LSS is far more resilient than QLSC (consider $s = 7$, for example). Again, LSS is the most consistent performer among all three—it is not far from SRS when the model is very poor, and it is not far from QLSC when the model is very good.

5.3 Running Time and Overhead

Before making a closer examination of LSS, we take a brief look at the running times of our approaches. Both LWS and QL methods (QLAC, QLCC, QLSC) are simpler than LSS, which has more overhead in stratification. Thus, we focus on LSS. In Figure 6, we plot the overhead added by using LSS when compared with SRS. There are three distinct sources of overhead in LSS: *Learning* represents the time to train the classifier; *Design* includes the time to compute the optimal stratified sampling scheme; *Application* accounts for the overhead in applying the chosen scheme, which involves picking objects from their associated strata. (Note that we already charge the samples used by LSS for learning and sampling design towards the total number of samples, which is set to be the same when comparing with other approaches.) In Figure 6, we also list the fraction of overall running time consumed by overhead at the top of each bar. Note these are miniscule (below 0.2%) compared with the overall cost, dominated by the predicate evaluation over samples. Such a low overhead implies that if we give simpler approaches such as SRS additional samples to account for the overhead of LSS, the number of additional samples would be too low to make any difference.

5.4 Closer Looks at LSS

Next, we test a variety of facets involved in LSS: how strata are laid out, the number of strata, allocation of samples for learning/design vs. estimation, and how the underlying classifier affects final estimation quality.

Strata Layout Strategy First, we study the impact of stratification strategy on LSS. Instead of using more sophisticated algorithms to look for optimal bucket boundaries (*optimal-width*), what if we use simpler strategies? In particular, *fixed-width* simply make all strata equal in width; *fixed-height* simply ensures that all strata contain the same number of objects. Figure 7 shows the results, using 4 strata. It is no surprise that *fixed-height* produces poor results for stratified sampling, as each strata may be force to contain a mixture of labels; in particular, for skewed datasets where one label occurs more often (XS and XXL), *fixed-height* has much higher variance in its estimates. *Fixed-width* fares better, but our *optimal-width* (which LSS uses by default) makes further gains—its interquartile range (IQR) is generally lower than the two simpler approaches.

Number of Strata In this experiment, we investigate the effect of the number of strata on estimation quality when using LSS and SSP, both of which use stratified sampling. We vary the number of strata with 4, 9, 25, 49, and 100 strata available. The results are summarized in Figure 8. Overall, as expected, increasing the number of strata tends to improve estimation quality, but not substantially so. Here, with XS result size and a large number of strata, SSP becomes competitive against LSS. The reason is that with superfine uniform gridding of the x-y space, the few positive objects eventually concentrate into a few strata, making SSP effective; in comparison, LSS may occasionally produce an outlier estimate, even though its overall variance is still competitive. Aside from these few extreme settings, however, LSS generally outperforms SSP, and often by significant margins as shown in Figure 8.

Sample Split Next, we test the effect of sample allocation on the quality of estimates produced by LSS. We vary the percentage of samples allocated to classifier training and sampling design from

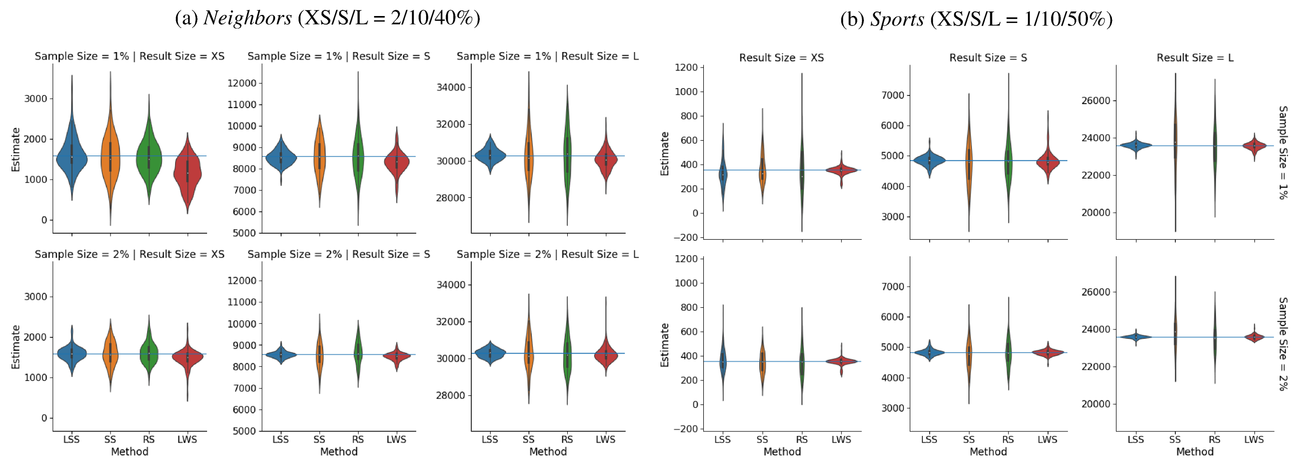

Figure 3: Distributions of estimates. Each row has a different sample size (1%, 2%), and each column has a different result size.

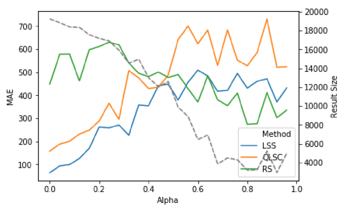

Figure 4: Varying α; synthetic datasets with Gaussian noise; $k = 15000$. Grey dashed line shows the result size (scale on right).

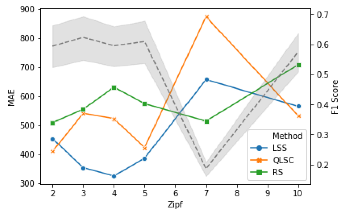

Figure 5: Varying Zipf parameter s; synthetic datasets with Zipf noise; $\alpha = 0.6$; $k = 15000$. Grey dashed line/band show the F1 scores of the classifier (scale on right).

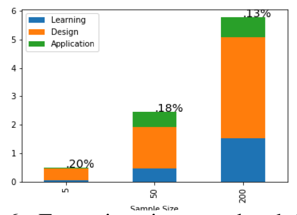

Figure 6: Execution time overhead (in seconds) vs. sample size (in thousands) for LSS, broken down by sources of overhead.

10%, 25%, 50%, to 75%. A 10% split means 10% of the total samples are devoted to learning and design, while the rest (90%) of the samples are used to produce the result estimate. We fix the number of strata at 4. Figure 9 summarizes the results. We see that at 75%, too few samples are devoted to estimation, so the result quality tends to suffer. Conversely, at 10%, too few samples are devoted to learning and design, and the result quality may also suffer. Both middle proportions (25% and 50%) consistently produce the most reliable estimates with lowest IQR's and fewer outliers.

Choice of Classifier As LSS is driven by the scores produced by a classifier, it is naturally dependent on the classifier itself. We tested LSS with four classifiers: k-Nearest Neighbors (KNN, with $k = 3$), simple two-layer neural network (NN, with 5 nodes per layer), random forest (RF, with 100 estimators), and a dummy classifier (Random) that assigns arbitrary random scores to objects. Random can be viewed as a worst case scenario for LSS as the desired effect of stratification (producing homogeneous strata) is completely lost. Across classifiers, we use 25% of the samples for learning and sampling design, and there are 4 strata. As we can see from the results in Figure 10, consistent with intuition, a classifier that performs better than Random produces better estimates. On the other hand, even if a classifier performs poorly (such as Random), LSS still produces reasonable estimates.

6. RELATED WORK

Sampling for Approximate Query Processing (AQP) Sampling is a fundamental problem in databases and has been studied over more than three decades [21, 20, 4]. Random samples are one of the key types of *synopses* [5] frequently used for AQP. Sampling for complex queries has been a long-standing challenge. In particular, sampling over joins is non-trivial, because simply joining independent samples of participating tables is ineffective [20, 4]. This problem has received much attention over the years, with representative works such as *ripple join* [7], *wander-join* [17], and more recently, sampling multi-way acyclic and cyclic joins [27]. The focus of our work is on counting queries with complex predicates. Even though our predicates can include joins as discussed in Section 2, our techniques differ because of different problem assumptions. First, some work on sampling over joins, e.g., [4], aims at producing a random sample of the result tuples, while we aim at estimating the result count. Second, to make our approach general, we adopt a rather simple evaluation model, where sampling a candidate object o to be counted involves evaluating $\mathsf{q}(o)$ exactly, without additional sampling or approximation. In contrast, much of the work on sampling over joins assumes specific forms of join predicates or availability of indexes to avoid enumerating all join results for o. On the other hand, all queries in our experiments are too complex for these approaches to handle, because these queries use constructs such as self-joins, complex non-equality join predicates, subqueries containing `GROUP BY` and `HAVING`, as well as UDFs.

BlinkDB [1] and *VerdictDB* [22] are examples of recent AQP systems aimed at supporting approximate processing of general, ad hoc queries. While these systems deliver very fast response time thanks to optimizations such as precomputation and parallelization, handling the full complexity of SQL remains challenging. For instance, *VerdictDB* does not support self-joins out of the box; our best attempt at adapting the query in Example 2 to run on it resulted in poor estimates compared with other approaches we experimented with in Section 5.

A number of papers are related to ours in the use of sampling. [19] considers stratified sampling design for both streaming and

(a) *Neighbors* (XS/S/L = 2/10/40%) (b) *Sports* (XS/S/L = 1/10/50%)

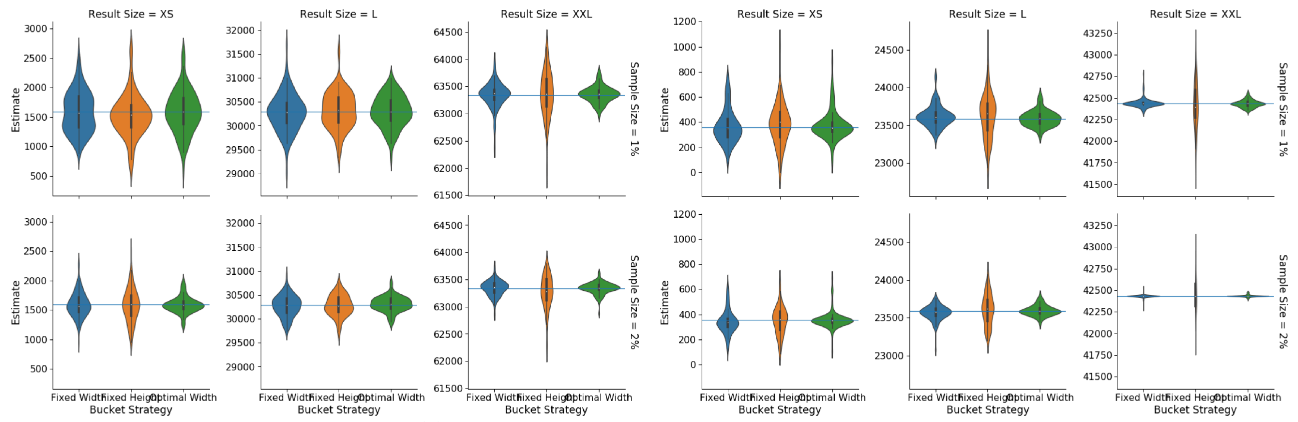

Figure 7: Effect of stratification strategy on LSS estimation quality. Each row represents a different sample size (1%, 2%), and each column represents a change of parameters resulting in a different result set size (XS, S, L).

(a) *Neighbors* (XS/S/L = 2/10/40%) (b) *Sports* (XS/S/L = 1/10/50%)

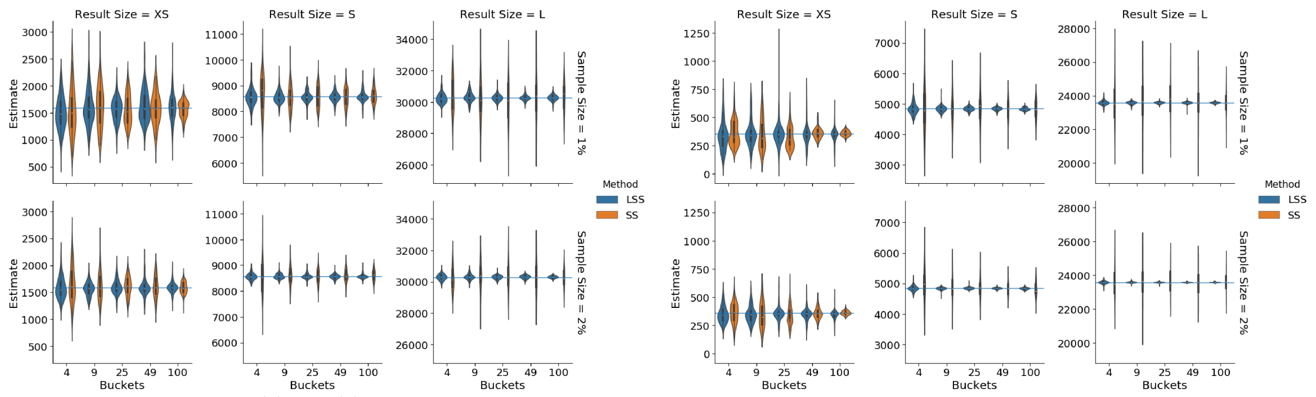

Figure 8: Comparison of LSS and SSP across varying number of strata. Each row represents a different sample size (1%, 2%), and each column represents a change of parameters resulting in a different result set size (XS, S, L).

stored data, and improves upon the Neyman allocation. [8] estimates the size of a query result by partitioning the query result and estimating the sum of the partition sizes. In our setting, each partition would be associated with one object, which contributes either 0 or 1 to the sum. [8] focuses on deriving a sequential sampling procedure, and considers both uniform random sampling and stratified sampling. However, unlike our work, it does not consider how to design stratification in a way to maximize sampling efficiency. Many other sampling papers are concerned with aggregates such as `SUM`, which are more susceptible to sample biases than just counting queries. [12] studies robust stratified sampling for low-selectivity aggregate queries, and uses a pilot sampling phase to estimate variance as we do for SSN and LSS. A combination of outlier-indexing with weighted sampling has been used in [3] to approximate aggregate results, and in [2], where differently biased subsamples can be dynamically selected to answer a query. [13] estimates the results of aggregates over SQL queries with subqueries involving (`NOT`) `IN`/`EXISTS`; notably, it proposes a low-variance estimator by learning a model from data using Bayesian statistical techniques. Compared with [13], our approach is simpler, uses off-the-shelf methods, and relies much less on the quality of the learned models. With the exception of [13], none of the work above applies any machine learning to help with estimation or to inform the sampling design.

Sampling has also been used for answering queries from dirty data with data cleaning [26]. In Section 3.3 we have discussed this connection and introduced the method QLSC inspired by *SampleClean* [26]. Experiments in Section 5 show that our learn-to-sample approach is more effective and less dependent on classifier quality.

Use of Machine Learning There has been a flurry of recent research on the use of machine learning in database systems. One related line of work is the use of machine learning for selectivity estimation, e.g., [15, 9], which can be seen as approximate counting queries. This line of work typically precomputes and maintains data summaries to support query optimization, or more ambitious optimizations across all components of a database system, e.g., *SageDB* [16]. Since their goal is to use estimates for optimization instead of answering counting queries per se, their estimates typically do not come with any guarantees. In contrast, we strive to provide statistical guarantees on our approximate answers.

Finally, two recent papers are very similar to our approach in spirit. To reduce the cost of evaluating expensive UDFs that arise frequently in machine learning pipelines, *Probabilistic Predicates (PP)* [18] use learned classifiers to pre-filter data before processing them further. Given a set of such classifiers and an accuracy requirement (minimum fraction of positives to retain), a query optimizer devises a plan that uses appropriate classifiers with optimal score cutoffs to pre-filter the data: any object scored lower than the cutoff by the classifier is dropped. A key difference between this work and ours is the problem definition: they target *reporting*

(a) *Neighbors* (XS/S/L = 2/10/40%) (b) *Sports* (XS/S/L = 1/10/50%)

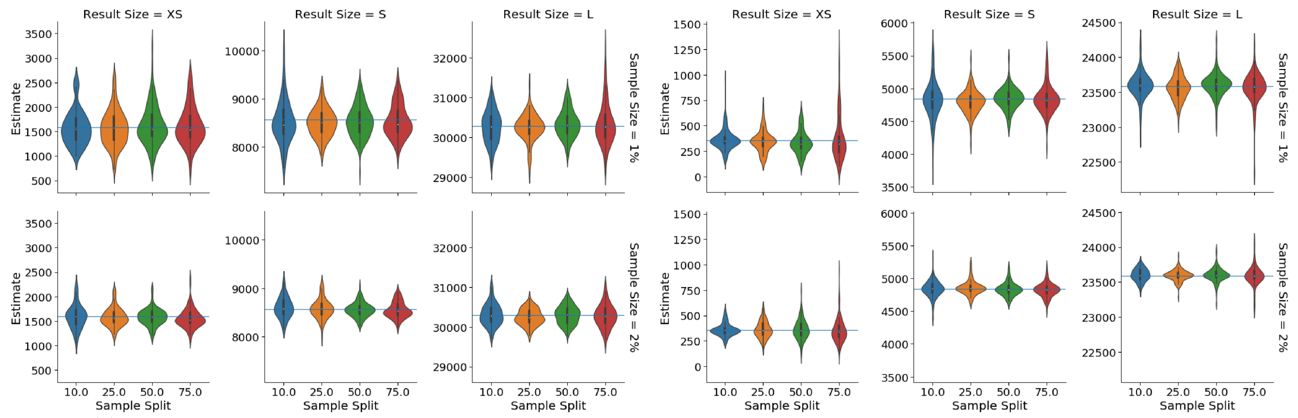

Figure 9: LSS as the percentage of samples used for learning/design (instead of producing the result estimate) varies. Each row represents a different sample size (1%, 2%), and each column represents a change of parameters resulting in a different result set size (XS, S, L).

(a) *Neighbors* (XS/S/L = 2/10/40%) (b) *Sports* (XS/S/L = 1/10/50%)

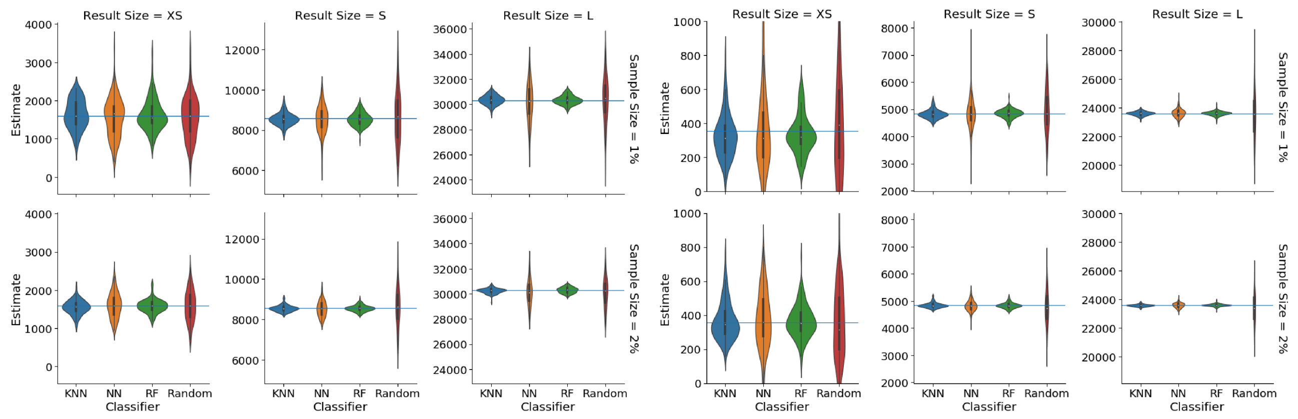

Figure 10: LSS under different classifiers. Each row represents a different sample size (1%, 2%), and each column represents a change of parameters resulting in a different result set size (XS, S, L).

queries while we target *counting* queries. This difference leads to our different use of the classifier scores; applying PP to our setting would result in poor estimates. Furthermore, PP gives no statistical guarantees on the actual recall, and its performance is far more susceptible to bad classifiers because of its heavy reliance on classifier scores. Earlier work by Joglekar et al. [11] similarly tackles queries involving selections with expensive UDFs. By identifying attributes whose values are correlated with UDF results, and grouping objects by the values of such attributes, they judiciously choose the appropriate actions to take for each group of objects (e.g., accept all, return all, or sample some). Like our approach, the use of sampling enables probabilistic guarantees, but the key difference remains that they target reporting instead of counting queries.

7. CONCLUSION AND FUTURE WORK

In this paper, we have developed new techniques to estimate the results of counting queries with complex filters. Our techniques are based on a simple yet powerful idea: replace an expensive filter with a cheap classifier that approximates the filter. This cheap classifier can then be used in a number of different ways with different trade-offs. A key challenge is that too much reliance on the classifier makes result quality highly susceptible to bad classifiers. However, one novel technique we proposed, *learned stratified sampling*, delivers consistently good estimates compared with other alternatives. It is very resilient against bad classifiers, thanks to how it combines machine learning and sampling—the learned classifier is used in a limited but helpful way to design a stratified sampling scheme which in turn produces the estimate. This resiliency makes the technique easy to apply in practice, because we are much less concerned with training a perfect model: a good model will make sampling more efficient, but even if the model is poor and/or the filter is fundamentally hard to approximate, the technique still delivers unbiased estimates with statistical guarantees comparable to random sampling. There is an abundance of future work. In particular, learned stratified sampling is quite conservative by design—to ensure independence, it avoids using the samples it acquired in the learning phase when computing the final estimate. However, there may be ways in which such samples can be safely used. Second, some of the queries we considered in this paper (such as skyband sizes and neighbor counts) have highly specialized solutions. Although our goal is to develop general solutions that can work for far more complex queries, it will still be interesting to carry out a direct comparison with the specialized solutions for these specific queries. Finally, a promising direction is to extend and fully evaluate our approach in an *online aggregation* [10] setting.

8. REFERENCES

[1] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. Blinkdb: Queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 29–42, New York, NY, USA, 2013. ACM.

[2] B. Babcock, S. Chaudhuri, and G. Das. Dynamic sample selection for approximate query processing. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 539–550, New York, NY, USA, 2003. ACM.

[3] S. Chaudhuri, G. Das, M. Datar, R. Motwani, and V. R. Narasayya. Overcoming limitations of sampling for aggregation queries. In *Proceedings of the 17th International Conference on Data Engineering, April 2-6, 2001, Heidelberg, Germany*, pages 534–542, 2001.

[4] S. Chaudhuri, R. Motwani, and V. Narasayya. On Random Sampling over Joins. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, SIGMOD '99, pages 263–274, New York, NY, USA, 1999. ACM.

[5] G. Cormode, M. Garofalakis, P. J. Haas, and C. Jermaine. Synopses for massive data: Samples, histograms, wavelets, sketches. *Foundations and Trends® in Databases*, 4(1–3):1–294, 2011.

[6] P. Gonzlez, A. Castao, N. V. Chawla, and J. J. D. Coz. A Review on Quantification Learning. *ACM Comput. Surv.*, 50(5):74:1–74:40, Sept. 2017.

[7] P. J. Haas and J. M. Hellerstein. Ripple Joins for Online Aggregation. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, SIGMOD '99, pages 287–298, New York, NY, USA, 1999. ACM.

[8] P. J. Haas and A. N. Swami. Sequential sampling procedures for query size estimation. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, SIGMOD '92, pages 341–350, New York, NY, USA, 1992. ACM.

[9] M. Halford, P. Saint-Pierre, and F. Morvan. An approach based on bayesian networks for query selectivity estimation. In *DASFAA (2)*, volume 11447 of *Lecture Notes in Computer Science*, pages 3–19. Springer, 2019.

[10] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, SIGMOD '97, pages 171–182, New York, NY, USA, 1997. ACM.

[11] M. Joglekar, H. Garcia-Molina, A. Parameswaran, and C. Re. Exploiting Correlations for Expensive Predicate Evaluation. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1183–1198, New York, NY, USA, 2015. ACM.

[12] S. Joshi and C. M. Jermaine. Robust stratified sampling plans for low selectivity queries. In *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, Mexico*, pages 199–208, 2008.

[13] S. Joshi and C. M. Jermaine. Sampling-based estimators for subset-based queries. *PVLDB*, 18(1):181–202, 2009.

[14] G. Kalton, K. Graham, et al. *Introduction to survey sampling*, volume 35. Sage, 1983.

[15] A. Kipf, T. Kipf, B. Radke, V. Leis, P. A. Boncz, and A. Kemper. Learned cardinalities: Estimating correlated joins with deep learning. In *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*, 2019.

[16] T. Kraska, M. Alizadeh, A. Beutel, E. H. Chi, A. Kristo, G. Leclerc, S. Madden, H. Mao, and V. Nathan. Sagedb: A learned database system. In *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*, 2019.

[17] F. Li, B. Wu, K. Yi, and Z. Zhao. Wander Join: Online Aggregation via Random Walks. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 615–629, New York, NY, USA, 2016. ACM.

[18] Y. Lu, A. Chowdhery, S. Kandula, and S. Chaudhuri. Accelerating Machine Learning Inference with Probabilistic Predicates. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 1493–1508, New York, NY, USA, 2018. ACM.

[19] T. D. Nguyen, M. Shih, D. Srivastava, S. Tirthapura, and B. Xu. Stratified random sampling over streaming and stored data. In *Advances in Database Technology - 22nd International Conference on Extending Database Technology, EDBT 2019, Lisbon, Portugal, March 26-29, 2019*, pages 25–36, 2019.

[20] F. Olken and F. Olken. Random sampling from databases. *Ph.D. thesis, U.C. Berkeley*, 1993.

[21] F. Olken and D. Rotem. Simple Random Sampling from Relational Databases. In *Proceedings of the 12th International Conference on Very Large Data Bases*, VLDB, pages 160–169, San Francisco, CA, USA, 1986. Morgan Kaufmann Publishers Inc.

[22] Y. Park, B. Mozafari, J. Sorenson, and J. Wang. Verdictdb: Universalizing approximate query processing. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 1461–1476, 2018.

[23] I. Partalas, A. Kosmopoulos, N. Baskiotis, T. Artieres, G. Paliouras, E. Gaussier, I. Androutsopoulos, M.-R. Amini, and P. Galinari. Lshtc: A benchmark for large-scale text classification, 2015.

[24] Y. Tillé. Sampling algorithms. In *International Encyclopedia of Statistical Science*, pages 1273–1274. Springer, 2011.

[25] B. Walenz, S. Sintos, S. Roy, and J. Yang. Learning to sample: Counting with complex queries. `https://arxiv.org/abs/1906.09335`.

[26] J. Wang, S. Krishnan, M. J. Franklin, K. Goldberg, T. Kraska, and T. Milo. A sample-and-clean framework for fast and accurate query processing on dirty data. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 469–480, 2014.

[27] Z. Zhao, R. Christensen, F. Li, X. Hu, and K. Yi. Random sampling over joins revisited. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 1525–1539, New York, NY, USA, 2018. ACM.