Manually Detecting Errors for Data Cleaning Using Adaptive Crowdsourcing Strategies

Haojun Zhang
University of Wisconsin-Madison
hzhang0418@cs.wisc.edu

Chengliang Chai
Tsinghua University
chaicl15@mails.tsinghua.edu.cn

AnHai Doan
University of Wisconsin-Madison
anhai@cs.wisc.edu

Paraschos Koutris
University of Wisconsin-Madison
paris@cs.wisc.edu

Esteban Arcaute
Facebook AI
esteban@fb.com

ABSTRACT

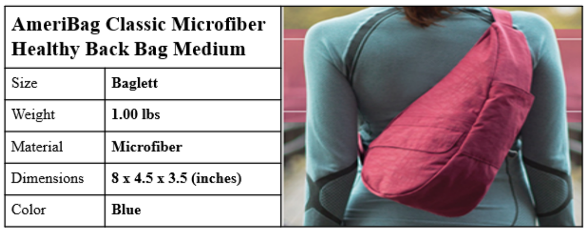
Current work to detect data errors often uses (semi-)automatic solutions. In this paper, however, we argue that there are many real-world scenarios where users have to detect data errors *completely manually*, and that more attention should be devoted to this problem. We then study one instance of this problem in depth. Specifically, we focus on the problem of *manually verifying the values of a target attribute*, and shows that the current best solution in industry, which uses crowdsourcing, has significant limitations. We develop a new solution that addresses the above limitations. Our solution can find a much more accurate ranking of the data values in terms of their difficulties for crowdsourcing, can help domain experts debug this ranking, and can handle ambiguous values for which no golden answers exist. Importantly, our solution provides a unified framework that allows users to easily express and solve a broad range of optimization problems for crowdsourcing, to balance between cost and accuracy. Finally, we describe extensive experiments with three real-world data sets that demonstrate the utility and promise of our solution approach.

1 INTRODUCTION

Data cleaning has received significant recent attention (e.g., [5, 7, 10, 46]), due to the explosion of data science applications, which often need data cleaning before analysis can be carried out. Most recent data cleaning works focus on detecting and repairing data errors [5, 7, 10, 46] (e.g., outliers, incorrect values, duplicate tuples, and constraint violations). In this paper we focus on *detecting* errors.

To detect data errors, current work often employs semi-automatic solutions, which use machine learning or hand-crafted data quality rules (e.g., "age must be between 18 and 80" and "any employee in NYC earns no less than any non-NYC employee at the same level" [5]). In certain cases the user can be involved, e.g., to provide feedback to the solutions or verify that the data instances reported by the solutions are indeed errors.

In practice, however, *there are many scenarios where users still have to detect data errors completely manually.* First, to detect data errors we often need to extract the values of certain attributes. Such extraction can be very difficult for today algorithms, but much easier for human users. This often happens when an attribute value is buried in a picture or text.



AmeriBag Classic Microfiber Healthy Back Bag Medium	
Size	Baglett
Weight	1.00 lbs
Material	Microfiber
Dimensions	8 x 4.5 x 3.5 (inches)
Color	Blue

Figure 1: An example of manual error detection.

Example 1.1. Consider the product in Figure 1. A data quality rule is "the value of attribute color *should be consistent with the color of the product in the picture". There is no algorithm today that can reliably extract the color of the product from the picture. Here the picture shows not just the product, a bag, but also a woman wearing a bag, making the extraction of the bag's color even more difficult. A human user however can quickly detect that the bag's color in the picture is red. This is inconsistent with the value of attribute* color *in the text, which is blue, suggesting a data error.*

Even if an attribute's values are present (so no extraction is required), it can still be difficult for algorithms to judge if those values are correct. For example, there is no good algorithm today to detect if a given URL is indeed the correct URL for a given business (especially where multiple fake URLs exist for a business). So detecting incorrect business URLs (e.g., to clean business listings) is still done largely by human users. Another example is verifying if the category of a product is "athletic (man)", which typically requires a human user to read the product description, examine the picture, etc.

Finally, an algorithmic solution may exist (e.g., extracting a person's gender from a picture can be done reliably today using deep learning [24]), but the business may have no one qualified to develop, debug, and run it. Or there is someone qualified, but developing and debugging the algorithm would take weeks, whereas the cleaning work must be done within days. In such cases, businesses often resort to detecting data errors completely manually, using human users.

In this paper we consider manually detecting data errors. As a first step, we consider the common setting in which *users must manually check the correctness of the values of a target attribute* (e.g., color, category). This problem is often called *manual data validation* [5, 7, 10, 46]. It is pervasive, yet no published work has addressed it in depth, as far as we can tell.

Using in-house experts to do manual data validation is not practical for large amounts of data: it takes too long and is not a good use of their limited time. So companies often use *crowdsourcing*, where the crowd can be for instance contractors or

Mechanical Turk workers. A common solution formulates each error detection as a question, sends it to the crowd, solicits k answers (e.g., $k = 5$), then takes a majority vote. For example, if three out of five workers answer "no" to the question "is the color of this product indeed blue?" for the product in Figure 1, then we can report that product as potentially having a data error.

The above solution is conceptually simple, but inefficient. Intuitively, different data values pose different levels of difficulties to human users. For example, most people know the color "blue", and so can answer questions about this color with high accuracy. But fewer people know the color "chartreuse". So we may want to solicit fewer answers for questions involving "blue" (e.g., 3 answers per question), but more answers for questions involving "chartreuse" (e.g., 7). Such crowdsourcing strategies, which are sensitive to the difficulties of different data regions, can significantly reduce the crowdsourcing cost while achieving the same level of error detection accuracy.

Indeed many companies have now employed such *adaptive crowdsourcing strategies*. A very common solution (e.g., employed at WalmartLabs, Facebook, Johnson Controls, and elsewhere) works as follows:

(1) Compute a ranking K of the data values (in decreasingly order of their difficulties),
(2) Examine K to assign to each data value v a number of answers n_v (such that a data value placed higher in K is assigned a higher number of answers), then
(3) Solicit n_v answers for each question with value v.

Obtaining the ranking K is important for many purposes. For example, after crowdsourcing to obtain answers for all questions, companies often take a sample and manually check the accuracies of the questions in the sample, for quality assurance purposes. The ranking K allows them to bias the sample, e.g., intentionally sample more items with values in the top-10 of K, to check how well crowdsourcing works for these difficult values. Example 4.1 lists other usages of ranking K.

While the above solution has been quite popular in industry, it has several important limitations. In this paper we address those and significantly advance this line of research in several ways.

First, obtaining *a good ranking of the data values* in terms of their difficulties is critical. To do so, the above solution estimates the difficulty score of a data value to be *the average worker accuracy* for a sample set S of questions with that value. To estimate these scores accurately, the size of sample set S must be quite large. But this incurs a lot of domain expert's effort, because he or she must label all data instances in S (as having data errors or not).

Here we show that we do not need a large sample S. Our key idea is that if the average time it takes for a worker to answer questions is high, or if the disagreement among workers is high, then those also indicate that a data value is likely to be difficult. Consequently, *we use all three factors (i.e., worker accuracy, average answer time, and worker disagreement) to directly rank the data values, using a machine learning approach.* We show how to minimize the domain expert's effort, by iteratively expanding sample S and stopping when a convergence condition is met.

Second, once the ranking has been created, domain experts often want to examine, debug, and modify it. To address this problem, *we develop a solution to help a domain expert debug the ranking.* Specifically, he or she can request explanations on why a data value v is considered difficult. Among others, our solution can explain that v is not difficult, but appears so due to spammers, low-quality workers, or careless mistakes from the workers; or that v is indeed difficult, because the value is hard to understand (e.g., "chartreuse"), or the item description is incomplete, or the description has confusing/conflict information, etc.

Third, the existing solution considers only the problem of *minimizing the crowdsourcing cost while achieving the same detection accuracy* (as the baseline solution of soliciting the same number of answers regardless of the data value). We show that in practice, users want to consider a far broader range of problems. Examples include minimizing cost given that the accuracy exceeds a threshold, maximizing accuracy given a budget on the cost, improving the overall accuracy of a set of data items having difficult values, and more. *We develop a unified framework that allows users to easily express and solve a broad range of such optimization problems*, all of which find crowdsourcing strategies that adapt to the data value difficulties.

Finally, the existing solution assumes *golden answers* exist for the questions with each data value (otherwise the worker accuracy for that value cannot be computed). In practice, surprisingly, we found that there are many cases where there are no such golden answers. For example, a product description may show the picture of a bag in sand color, with the value for attribute "color" being "desert sand". So the question for the crowd is "is the color of this product 'desert sand'?". But nobody knows what "desert sand" means. There is no such color. Or more accurately, this is an *ambiguous* color invented by the marketing team. As such, there is no correct, i.e., golden answer to the above question. (In our experiments, 2/3 of workers answer yes, and the rest answer no.) Clearly, this problem of *ambiguous values* must be addressed, before the above adaptive crowdsourcing solution can be applied. In this paper we develop a simple but effective solution to this problem.

Contributions: To summarize, in this paper we make several fundamental contributions to the problem of manually detecting errors for data cleaning:

- We argue that the above problem is pervasive, and needs more attention. As far as we can tell, this is the first work that studies this problem in depth.
- We focus on the problem of manually verifying the values of a target attribute, and shows that the current best solution has significant limitations.
- We develop a new solution that addresses the above limitations and significantly advances the state of the art. Our solution can find a much more accurate ranking of the data values, can help domain experts debug this ranking by providing explanations on why a data value is considered difficult, and can handle ambiguous values for which no golden answers exist. Importantly, our solution provides a unified framework that allows users to easily express and solve a broad range of optimization problems.
- We describe extensive experiments with three real-world data sets that demonstrate the utility and promise of our solution approach.

2 PROBLEM DEFINITION

We now describe the problem of manual detection of data errors considered in this paper.

Data Items, Attributes, and Values: For manually detecting data errors, many problem types exist. As a start, in this paper we will consider the problem of manually verifying the correctness of the categorical values of a target attribute. Specifically, let $D = \{d_1, \ldots, d_n\}$ be a set of data items, such as books, papers, products, etc. We assume that each item is encoded as a tuple of attribute-value pairs, i.e., $d_i = \langle a_1 = v_{i1}, \ldots, a_m = v_{im} \rangle$. For example, a product may be encoded as $\langle category = shirt, gender = male, color = blue \rangle$. We will use $a_j(d_i)$ to refer to the attribute a_j of d_i.

We assume that each attribute $a_j(d_i)$ has a set of correct values $V_j^*(d_i)$. For example, a course about discrete math is suitable for students from both mathematics and computer science departments, therefore its subject contains at least two values: mathematics and computer science. We say that $a_j(d_i)$ is correct if and only if its value v_{ij} is in $V_j^*(d_i)$.

Further, we assume that all attributes of each data item d_i are correct, except one attribute, which is referred to as the *target attribute* and whose values we will need to verify. Without loss of generality, we assume that the target attribute is the last attribute a_m.

Manual Validation of the Target Attribute: Let c_i be the context of d_i, defined as "all other attributes and their values": $c_i = \langle a_1 = v_{i1}, \ldots, a_{m-1} = v_{i(m-1)} \rangle$.

Our problem is to verify $a_m(d_i)$ for all items d_i in D. For each item d_i, verifying whether the value of a_m is correct is equivalent to answering the following question q_i: "is the value of $a_m(d_i)$ indeed v_{im}, given the context c_i and any other background knowledge B that the worker may have?" (we discuss examples of background knowledge B below).

Then the problem of verifying the target attribute a_m for items in D can be translated into answering the set of questions $Q = \{q_1, \ldots, q_n\}$, where the answer for each question q_i is yes or no. If the answer is yes, then our confidence that $a_m(d_i)$ is correct is increased. If the answer is no, then it is likely that there is a data error in d_i (in practice, the error may not be in a_m, but the error must exist because $a_m(d_i)$ is inconsistent with c_i). In this case, d_i is sent for further verification by an expert.

Suppose we have golden answers for all questions in Q, then for any solution to the above validation problem, we can define its overall accuracy to be the fraction of questions whose answers are correct, i.e., n_0/n, where n_0 is the number of questions whose answers match the golden answers, and $n = |Q|$.

Current Manual Solutions: Today, such questions are often answered manually, on a GUI, by an expert or a small set of experts, e.g., data analysts at an e-retailer, data scientists in an R&D group. To give the expert the maximal context information, a question will typically display the entire description of the item, e.g., all attribute-value pairs (see Figure 1).

If the expert still cannot decide after examining these attribute-value pairs, he or she may try to find more information, e.g., examining the same product at a different e-retailer, looking for any new information that can help answer the question. For the question "is the color of this product chartreuse?", the expert may have to first look up the meaning of "chartreuse" on the Web, and so on. We refer to such externally acquired knowledge as *the background knowledge B*.

Clearly, this manual solution is tedious and time consuming. As a result, many real-world applications have turned instead to crowdsourcing to verify attribute values.

Current Crowdsourcing Solutions: The simplest crowdsourcing (CS) solution solicits k answers from crowd workers for each question $q \in Q$, then combines these answers using majority voting to obtain a final answer for q.

Note that we can combine worker answers using more sophisticated strategies, e.g., estimating each worker's accuracy, then taking a weighted sum [19, 22, 27, 39]. Many real-world applications, however, still use majority voting, which is easy to understand, debug, and maintain. This is especially important when there is a high personnel turnover. Further, the application may need to contract with a crowdsourcing company and this may be the only solution being offered by that company. Finally, as far as we know, there is no published conclusive evidence yet that more sophisticated strategies to combine answers work much better in practice. Thus, in this paper we will focus on the above majority-voting solution to verify attribute values, leaving more sophisticated solutions for future research.

The above CS solution, while faster than manual solutions, can incur high monetary costs, especially if the application wants high accuracy for crowdsourcing.

Example 2.1. Suppose an e-retailer A must verify the attributes of 50K newly arrived products. To ensure that product details on its Web pages are error-free as much as possible, A wants crowdsourcing to have an accuracy of at least 95%. To reach this accuracy, soliciting 3 answers per question is often insufficient, A would need to solicit 5, 7, or more answers. Assuming 3 cents per answer, if A solicits 5, 7, or 9 answers per question, crowdsourcing 5 attributes of 50K products costs \$37.5K, \$52.5K, and \$67.5K, respectively.

Thus, it is important that we develop solutions to minimize the crowdsourcing cost, while achieving the same verification accuracy. As discussed in the introduction, intuitively, different data regions often have different degrees of difficulty for human verification. So if we can estimate these difficulty levels, we can adjust the degree of redundancy (i.e., the number of answers solicited for each question in a region). For example, a set of products D can be split into data regions where all products with the same color form a region. Then for each question in the region with "red" color, we only need to solicit 3 answers, say; whereas for each question in the region with the "acid yellow" color, which is more difficult, we would solicit 7 answers.

Indeed many companies have now employed such *adaptive CS strategies.* A very common solution works as follows:

(1) Compute a ranking of the data regions (in decreasingly order of their difficulties),
(2) Examine the ranking to assign to each region a number of answers (such that a region placed higher in the ranking is assigned a higher number of answers), then
(3) Solicit that number of answers for each question in the region.

It is important that this solution outputs both a ranking and a crowdsourcing plan (which specifies how many answers to solicit for each question in a data region). Outputting a ranking serves many important purposes, as discussed in the introduction (see also Example 4.1).

The above solution is appealing, but has significant limitations. (1) The ranking that it computes is often inaccurate, because the solution uses only the average worker accuracy to find the ranking. (2) Domain experts often cannot debug the ranking, e.g., understand why a data region is considered difficult. (3) The task of assigning to each data region a number of answers is often done in an ad-hoc "eyeballing" way, by examining where the data region is in the ranking. (4) It is difficult to express and solve a broad range of optimization problems regarding crowdsourcing costs and accuracy, even though users often have such needs. (5) Finally, this solution cannot handle "ambiguous" data values (e.g., "desert sand"), for which there are no golden answers.

In the rest of this paper we introduce our solution, called VChecker, which addresses the above limitations.

3 RANKING THE DATA REGIONS

In VChecker, we first obtain a ranking of the data regions, in decreasing order of their difficulties. In this section we discuss how we split the data into different regions, then rank them. (The next two sections describe how to debug the ranking, then how to use it to formulate and solve a broad range of optimization problems, to find good crowdsourcing plans.)

3.1 Splitting Data into Regions

We consider scenarios where for each data instance d_i, the difficulty in verifying the target attribute a_m only depends on its value v_{im}. Such scenarios are common in practice. For instance, for products such as the one described in Figure 1, the difficulty of verifying the attribute color only depends on its value (e.g., red, blue, acid yellow, etc.).

In such cases, the expert will split the data such that all questions with the same target attribute value form a region (because all such questions share the same difficulty level). Formally, let $D = \{d_1, \ldots, d_n\}$ be the set of data instances, a_m be the target attribute, $Q = \{q_1, \ldots, q_n\}$ be the set of questions "is the value of attribute a_m of instance d_i indeed v_{im}?", and $V = \{v_1, \ldots, v_r\}$ be the set of all values of a_m for instances in D. Then we can split the set of questions Q into r sets such that all questions (and only these questions) in a set Q_i share the same value for attribute a_m. We refer to each such set as a data region. In general, it is not always possible to so simply split the data into regions. This raises the interesting problem of how to help the expert do so, which we leave for future work.

3.2 Learning to Rank the Regions

To rank the data regions, a common solution in industry is to compute for each region an *average worker accuracy*, then rank the regions in increasing worker accuracy (thus in decreasing difficulties).

Specifically, let Q_i be a data region, i.e., the set of questions (in Q) with the same value v_i for the target attribute a_m. The current solution assumes that all crowd workers have the same probability of answering any question in Q_i correctly (a reasonable assumption in many real-world scenarios). It then takes this probability to be the average worker accuracy for Q_i, denoted as $a(Q_i)$.

To estimate $a(Q_i)$, the solution randomly takes a set x of questions in Q_i, solicits y answers from the crowd for each question, then computes $a(Q_i)$ as the fraction of xy answers that are correct. To determine answers' correctness, the solution uses the golden answers to the x questions, as provided by an expert. Finally, the solution ranks the data regions in increasing order of the computed average worker accuracy $a(Q_i)$.

While conceptually simple, this solution is limited in several important ways. First, it provides no way to determine x and y. If they are set to large values, then we waste a lot of crowdsourcing money and expert time (to provide golden answers). If they are set to small values, then it is difficult to estimate $a(Q_i)$ *accurately*. Second, it fails to exploit extra information that can help better rank the data regions. Finally, the solution does not provide any way to solicit and incorporate knowledge from the expert, even though he or she often has such knowledge about the difficulties of the various data regions.

$V = \{$black, red, iris, lavender, chartreuse$\}$
(a)

$F = \{f_1 = \langle 1, 2.3, 0\rangle,$
$f_2 = \langle 1, 2.4, 0.05\rangle,$
$f_3 = \langle 0.7, 5.1, 0.1\rangle,$
$f_4 = \langle 0.6, 8.4, 0.1\rangle,$
$f_5 = \langle 0.7, 11.2, 0.15\rangle\}$
(b)

$G = \{\langle$black, $f_1\rangle$, $\langle$red, $f_2\rangle$, $\langle$chartreuse, $f_5\rangle\}$
(c)

$\{$chartreuse$\} \geq \{$black, red$\}$
(d)

$S = \{(f_5, 1), (f_1, 2), (f_2, 2)\}$
(e)

Figure 2: Creating training data for SVM Rank.

Key Ideas of Our Solution: Our solution exploits three key ideas. First, we observe that if a value is difficult, it often takes a worker longer to provide an answer (e.g., for a question involving the value "acid yellow", he or she may need to consult the Web to understand its meaning before being able to answer the question). It also often causes more disagreement among the workers. As a result, we capture and exploit these two types of information and use them together with the worker accuracy to learn to rank the values in decreasing order of their difficulties.

Second, to learn the ranking, we ask the expert to provide training data in the form of (v_i, v_j) such that v_i is ranked more difficult than v_j. We also allow the expert to debug the ranking and manually edit it if necessary (see Section 4). Thus, our solution provides a natural way for the expert to provide domain knowledge about the difficulties of the data regions.

Finally, to minimize the crowdsourcing and expert cost, we develop a solution in which we iteratively explore larger values for x (the number of questions sampled per value) and y (the number of answers solicited per question), and stop when a condition is met. We now describe the above ideas in detail.

1. Defining the Problem of Ranking the Values: Let $V = \{v_1, \ldots, v_r\}$ be the set of all values of the target attribute for all data instances in D. Our goal is to find a total ranking K of the values in V, such that v_i being ranked higher than v_j means that v_i is judged more difficult than v_j.

2. Learning the Ranking: For each value $v_i \in V$, we start by sampling x questions from the corresponding region Q_i, then solicit y answers for each question from the crowd (we explain later how to select x and y). This produces a total of xy answers.

Next, we create a feature vector $f_i = \langle a_i, t_i, e_i\rangle$, where a_i is the worker accuracy for v_i, computed as the fraction of xy answers that are correct. To determine if an answer is correct, the expert must provide the golden answers for the x questions. t_i is the time it takes for a worker to answer the questions, averaged over the xy answers. Finally, e_i is the disagreement among the workers in answering the questions, measured as $1 - |N_{yes} - N_{no}|/(xy)$, where N_{yes} and N_{no} are the total numbers of yes/no answers from the workers, respectively (and $N_{yes} + N_{no} = xy$).

Example 3.1. Consider the five colors in set V in Figure 2.a. Figure 2.b shows five feature vectors created for these colors, after sampling x questions from each color region and soliciting y answers from the crowd for each question.

At this point, we have obtained a set of feature vectors $F = \{f_1, \ldots, f_r\}$, one for each value. We now learn to rank the values, using these feature vectors. To do so, we use SVM Rank, a well-known ML algorithm that can be used to rank examples [38].

To use SVM Rank, we create training data as follows. First, we randomly sample a set G of feature vectors (FVs) from F. Next, we need to rank the FVs in G (in terms of the difficulty of their corresponding values). Abusing notation, we use "$f_i \geq f_j$" to indicate that FV f_i is ranked the same or higher than FV f_j (i.e., the value corresponding to f_i is the same or more difficult than that of f_j).

Ideally, we want to create a total ranking on G, i.e., for any pair $(f_i, f_j) \in G \times G$, either $f_i \geq f_j$ or $f_j \geq f_i$, then use this total ranking as training data. However, creating a total ranking is very expensive and often quite difficult for the expert, so we ask him or her to create only a partial ranking. Specifically, the expert merely divides G into two groups U and V based on his or her domain knowledge such that for any $f_i \in U$ and $f_j \in V$, $f_i \geq f_j$.

Then for each $f_i \in U$, we create a training example $(f_i, 1)$. Similarly, for each $f_j \in V$, we create a training example $(f_j, 2)$. Here we assume that an example associated with rank 1 is more difficult than any example associated with rank 2. We output the set S of all these examples as the training data for SVM Rank.

Example 3.2. Continuing with Example 3.1, suppose we have selected the three colors black, red, and chartreuse for creating the training data (see Figure 2.c). Suppose the expert specifies that chartreuse is considered more difficult than both black and red (Figure 2.d). Then we can create the training set S in Figure 2.e for SVM Rank.

SVM Rank then uses S to learn a regression model that assigns a score to each example, such that the higher the score, the higher the example is ranked. Finally, we apply SVM Rank to FVs in F to compute for each FV a score and use these scores to rank the FVs. This produces a total ranking K for the values in V, such that a higher ranked value is said to be more difficult than a lower-ranked one.

The expert can then optionally examine, debug, and edit the ranking K, as we discuss later in Section 4.

3. Determining Parameters x and y: Recall that for each value v_i we take x questions from the set of questions with that value Q_i, then solicit y answers per question from the crowd. We now discuss how to set x and y. Our solution is to start with the smallest x and y, iteratively increase them, computing rankings along the way, then stop when these rankings have "converged". This way, we hope to minimize the cost of the expert (who needs to answer $x|V|$ questions and the crowd (who needs to answer $xy|V|$ questions).

Specifically, we start with (2,2), i.e., $x = 2$ and $y = 2$ (the smallest values that allow us to meaningfully compute feature vectors), and compute the ranking of the values $K(2, 2)$, as described earlier. Next, we increase y to consider (2,3), and compute $K(2, 3)$. Then we consider (3,3) and compute $K(3, 3)$, and so on. To compute a new ranking, say $K(3, 3)$, using SVM Rank, the expert needs to label, i.e., provide golden answers to the new questions, and we need to solicit crowd answers for these questions. But we do not have to create any additional training data.

We use the Spearman score [48], which ranges from 1 to -1, to measure the correlation between any two rankings. Consider three consecutive rankings $K(x_{n-2}, y_{n-2})$, $K(x_{n-1}, y_{n-1})$, $K(x_n, y_n)$. If it is the case that the score Spearman$(K(x_{n-2}, y_{n-2}), K(x_{n-1}, y_{n-1}))$ and the score Spearman$(K(x_{n-1}, y_{n-1}), K(x_n, y_n))$ are both exceeding a pre-specified threshold, or if x_n and y_n reach pre-specified maximal values, then we stop, returning $K(x_n, y_n)$ as the desired ranking. Algorithm 1 shows the pseudo code of the entire process to rank the data regions.

4 DEBUGGING THE RANKING

Recall from the previous section that at the start, we enlist the expert and the crowd workers to create a ranking K of the values in V, in decreasing order of difficulties. In practice, it turns out that the ranking K can be used for many important purposes.

Example 4.1. The ranking K can be used to re-calibrate the worker accuracies of the values (see Section 5.2). It can be used in formulating optimization problems, e.g., a user may want to focus on the top-10 most difficult values in K and try to maximize the average accuracy of those values (see Section 5.1). Finally, K can also be used in quality assurance (QA). For example, after we have crowdsourced to obtain answers for all questions, we may want to take a sample and manually check the accuracies of the questions in the sample, for QA purposes. The ranking K allows us to bias the sample, e.g., intentionally sample items with values in the top-10 of K, to check how well the crowdsourcing process works for these difficult values.

As a result, *it is important to make the ranking K as accurate as possible.* Once K has been created (see Section 4), the expert often wants to examine, debug, and modify it. Currently, however, there is no debugging support. To address this problem, as a first step, in this paper we will develop a way to generate explanations, which can help the expert debug K.

Specifically, given a value v placed high in the ranking K, indicating that it is difficult, the expert can ask for an explanation on why v is judged difficult by the system.

Example 4.2. When asked why "acid yellow" is judged difficult, our system may return explanations that state that this value is actually not difficult, but appears to be difficult due to spammers and low-quality (i.e., bad) workers who gave many incorrect answers. Or the system may return explanations that state that the value is indeed difficult because it is unfamiliar to many workers. Other explanations may include "the product description contains incomplete or confusing information" and "the description is hard to understand", among others.

Clearly, such explanations can significantly help the expert understand and debug the ranking K. To generate such explanations, we first develop a model M on how a crowd worker answers a question. Next, we analyze M to create a taxonomy T of possible explanations. Finally, we develop a procedure that, when given a value v, analyzes answers solicited from the crowd to identify the most likely explanations in T for v. We now discuss these steps in more details.

Developing a User Model for Answering Questions: There are many possible ways to model how a worker answers our questions. For this paper we use the following simple yet reasonable model. Suppose a worker U has to answer a question q, which has a value v for the target attribute and a context c (which is the rest of the description of the data item). Then U first tries to understand v. Next, U tries to understand c. Finally, U determines if v and c are consistent, returning "yes" or "no" if

Algorithm 1 Learning to Rank the Data Regions

```
Input: $Q$: set of questions, $V$: set of values, $x_{\max}$: max num of sampled questions,
    $y_{\max}$: max num of answers to be collected per sampled question, $(x_0, y_0)$: initial
    value for $(x, y)$, $\epsilon$: convergence threshold for ranking, $n_0$: number of training
    examples for SVM Rank
Output: A ranking $K$ of values
 1: $V_t \leftarrow$ Randomly sample $n_0$ values from $V$
 2: $O \leftarrow$ CreatePartialOrder($V_t$)
 3: $P \leftarrow$ GenerateConfigs($x_0, y_0, x_{\max}, y_{\max}$)
 4: $Q_s, A_s, T_s, G_s \leftarrow \emptyset$
 5: $(x_c, y_c) \leftarrow (0, 0)$ // current $(x, y)$
 6: $L \leftarrow [\,]$ // list of rankings
 7: for $(x, y) \in P$ do
 8:     Sample $x - x_c$ new questions per value and add them to $Q_s$
 9:     Collects needed answers and time data and add them to $A_s$
        and $T_s$
10:     Find golden answers for newly sampled questions and add
        them to $G_s$
11:     Compute set of features $F$ from $A_s, T_s, G_s$
12:     $K(x, y) \leftarrow$ SVMRank values in $V$ using $F$, $O$
13:     $(x_c, y_c) \leftarrow (x, y)$
14:     Append $K(x, y)$ to $L$
15:     if IsRankingConverged($L, \epsilon$) then
16:         break
17:     end if
18: end for
19: $K \leftarrow K(x_c, y_c)$
20: $W \leftarrow$ Improve estimated worker accuracy using $K$ in Equation 4
21: return $K, W$

22: procedure CreatePartialOrder($V_t$)
23:     The expert partitions $V_t$ into two groups $U, V$ such that each
        value in $U$ is more difficult than each value in $V$
24:     $O \leftarrow \emptyset$
25:     for $v \in U$ do
26:         Add $(v, 1)$ into $O$
27:     end for
28:     for $v \in V$ do
29:         Add $(v, 2)$ into $O$
30:     end for
31:     return $O$
32: end procedure

33: procedure GenerateConfigs($x_0, y_0, x_{\max}, y_{\max}$)
34:     $P \leftarrow [(x_0, y_0)]$
35:     $n = \min(x_{\max} - x_0, y_{\max} - y_0)$
36:     for $i = 1, 2, \ldots, n$ do
37:         Append $(x_0 + i - 1, y_0 + i)$ and $(x_0 + i, y_0 + i)$ to $P$
38:     end for
39:     if $x_0 + n == x_{\max}$ then
40:         $m = y_{\max} - y_0 - n$
41:         for $i = 1, 2, \ldots, m$ do
42:             Append $(x_0 + n, y_0 + n + i)$ to $P$
43:         end for
44:     else if $y_0 + n == y_{\max}$ then
45:         $m = x_{\max} - x_0 - n$
46:         for $i = 1, 2, \ldots, m$ do
47:             Append $(x_0 + n + i, y_0 + n)$ to $P$
48:         end for
49:     end if
50:     return $P$
51: end procedure

52: procedure IsRankingConverged($L, \epsilon$)
53:     if $len(L) < 3$ then
54:         return False
55:     else
56:         Let $K_{n-2}, K_{n-1}, K_n$ be the last three rankings in $L$
57:         $s_1 \leftarrow$ Spearman($K_{n-2}, K_{n-1}$)
58:         $s_2 \leftarrow$ Spearman($K_{n-1}, K_n$)
59:         if $s_1 \geq \epsilon$ and $s_2 \geq \epsilon$ then
60:             return True
61:         else
62:             return False
63:         end if
64:     end if
65: end procedure
```

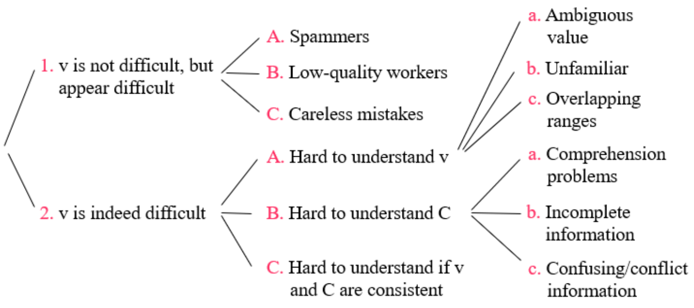


Figure 3: A taxonomy of explanations.

Algorithm 2 Generating Explanations

```
Input: $v$: the value to be explained
Output: $E_v$: the set of possible explanations for value $v$
 1: Collect data $S = \{Q_x, A, W\}$ where $Q_x$ is all $x|V|$ questions sam-
    pled in difficulty estimation stage, $A$ is all answers and their time
    stats for questions in $Q_x$, $W$ is the set of all workers for $A$
 2: Compute accuracy $\alpha$ and average time $t$ for $A$
 3: for each $w \in W$ do
 4:     Apply a procedure on $R_w$ using $\alpha, t$ to classify $w$ as spammers,
        low-quality workers, or regular workers
 5: end for
 6: Let $W_v$ be the set of workers who answer at least one question with
    value $v$
 7: $E_w \leftarrow$ GenWorkerExplanations($W_v$)
 8: Update $S$ to $S^+$ by removing answers and the time stats from spam-
    mers and low-quality workers
 9: Apply a procedure on $R_v$ using $S^+, \alpha, t$ to classify the nature $N_v$
    of value $v$ (i.e., ambiguous, unfamiliar, overlapping)
10: $E_v \leftarrow$ GenValueExplanations($N_v$)
11: Let $Q_v$ be the set of questions in $Q_x$ with value $v$
12: for each $q \in Q_v$ do
13:     Apply a procedure on $R_q$ using $\alpha, t$ to classify the nature of $q$
        (i.e., comprehension, incomplete, confusing/conflict)
14: end for
15: Let $N_q$ be the set of natures for questions in $Q_v$
16: $E_q \leftarrow$ GenQuestionExplanations($N_q$)
17: $E_v \leftarrow E_w \cup E_v \cup E_q$
18: return $E_v$
```

U can make this determination with high confidence. Otherwise U returns the answer ("yes" or "no") judged most likely.

Creating a Taxonomy of Explanations: Analyzing the above user model produces the taxonomy of explanations in Figure 3. Observe that the explanations fall into several clean groups. The first group concerns the *workers:* if they are spammers, bad workers, etc., then the value may not be difficult but will appear difficult. The second group concerns the *nature of the value v itself:* is it ambiguous, unfamiliar, etc? If so, it may explain why v is ranked difficult. The final group concerns the *nature of the question/the description of the data item/the context.* Is the description understandable (e.g., in English)? Is it complete? As we will see below, we can develop solutions to explore each of the above groups of explanations.

Generating Explanations for a Value v: Given a value v, we now seek to generate explanations for why v is difficult. We refer to each node in the above taxonomy (see Figure 3) as an *explanation.* Our solution will return a set of such explanations (in future work we will explore ranking them). To do so, the solution proceeds in the following steps (see Algorithm 2 for the pseudo code).

(1) Collect data S: We first collect data that can be analyzed to generate explanations. This data S consists of Q_x, the set of all questions generated for the difficulty score estimation process, A, the set of all answers solicited for questions in Q_x, and W, the set of all workers who have given at least one answer in A.

(2) Use S to classify the workers: We use a rule-based procedure to classify workers in W into spammers, bad workers, and regular workers. For example, we classify a worker w as a spammer if w's accuracy is significantly lower than the average accuracy, and w's response time is much faster than the average response time (as computed from data S).

(3) Generate explanations regarding the nature of the workers: Next, we identify likely explanations regarding the nature of the workers in the taxonomy T. For example, if a certain percentage of workers that have answered questions involving v are spammers, then we will identify node "1.A (Spammers)" of T as an explanation.

(4) Update data S into S^+: Next, we remove the data involving the spammers and bad workers from S, so that we can work with more accurate statistics in subsequent steps.

(5) Use S^+ to classify the nature of value v: Similar to Step 2, here we use a rule-based procedure to analyze S+, to classify the value v as ambiguous, unfamiliar, etc. For example, if the average worker accuracy for v is high and the average response time is low, then we determine that v is not unfamiliar.

(6) Generate explanations regarding the nature of value v: Again, similar to Step 3, we identity explanations in taxonomy T that involve the nature of value v. This step is straightforward.

(7) Use S^+ to classify the nature of the questions and generate explanations: We proceed similarly to Steps 2-3. For example, if a certain percentage of the questions involving v is confusing, then we will identify node "2.B.c" of T as an explanation. Finally, we return all identified explanations as the set of explanations for value v.

It is important to note that our rule-based procedures for the above steps have been created, only once. They are not dependent on the particular application domain. However, the rules employed do use various parameters (e.g., thresholds). These parameters are set based on analyzing the data S (but can also be tuned by the domain expert).

5 FINDING GOOD PLANS

We now discuss finding good crowdsourcing plans. We begin by considering the *types of problems* that the user wants to solve. As discussed in Section 1, a common baseline crowdsourcing (CS) plan is to solicit t_b answers per question, then take the majority vote to be the final answer. The existing solution has considered a single problem: minimize the total CS cost while keeping the accuracy the same as that of the baseline plan.

In practice, however, we observe that *users often want to express a wide range of other CS problems*. Examples include minimizing cost given that the accuracy exceeds a threshold, maximizing accuracy given a budget on the cost, improving the overall accuracy of a set of data items having difficult values, and more.

As a result, in this section we develop a unified framework in which users can easily express a variety of such CS problems. Some of these problems make use of the ranking K (e.g., maximizing the average accuracy of the values in the top-5 of K). Next, we show how to solve these problems using integer linear programming (ILP). Our solutions often involve the average worker accuracy per data value. Finally, we show how to use the ranking K to improve our estimations of these average worker accuracies.

5.1 Expressing Crowdsourcing Problems

Let $D = \{d_1, \dots, d_n\}$ be a set of data items to be validated. Let $V = \{v_1, \dots, v_r\}$ be the set of values for the target attribute of the items in D. We define a crowdsourcing plan p to be a tuple $\langle\langle v_1, t_1\rangle, \dots, \langle v_r, t_r\rangle\rangle$, where for each question involving the value v_i, plan p will solicit t_i answers from the crowd ($i \in [1, r]$).

Let $S \subseteq V$ be a set of values. We define $acc(S, p)$ to be the accuracy of plan p for the values in S, i.e., the fraction of questions with value $v \in S$ that receive a correct (aggregated) answer when p is executed. We define $cost(S, p)$ to be the total number of answers solicited from the crowd for the questions with value $v \in S$.

We can now define a general CS problem template as follows *"Given a set of plans P and a set of values $\mathcal{S}$, return the plan that maximizes or minimizes an objective O, subject to a constraint C, where O and C involve P and $\mathcal{S}$, and optionally a ranking K of values"*. In this paper we consider the following concrete CS problems that follow the above template.

Finding Plans That Outperform a Baseline Plan: In many scenarios there exists already a baseline plan p_b. The user however wants a plan p that is better than p_b in some aspects. While numerous problem variations exist, in this paper we consider the following variations:

T_1: Minimize cost while achieving the same accuracy

Return the plan p that minimizes $cost(V, p)$, subject to constraints $acc(V, p) \geq acc(V, p_b)$ and $cost(V, p) \leq cost(V, p_b)$. This is the problem considered by PBA [14].

T_2: Maximize accuracy while keeping the same cost

Return the plan p that maximizes $acc(V, p)$, subject to constraints $cost(V, p) \leq cost(V, p_b)$ and $acc(V, p) \geq acc(V, p_b)$.

T_3: Maximize the individual accuracy

In many cases the overall accuracy $acc(V, p)$ can be high, say 95%, yet certain individual accuracies (e.g., $acc(v, p)$ for certain v-s) may be quite low, say 60%. For example, the overall accuracy for color verification can be 95%. Yet the accuracy for "chartreuse" is only 60%.

In such cases, the user often wants to improve the accuracies of the values *across the board* as much as possible, while keeping the overall accuracy at least as high as that of p_b and keeping the overall cost at most as high as that of p_b. To do this, the user can try to solve the following problem: Return the plan p that maximizes $min_{v_i \in V} acc(v_i, p)$, subject to $acc(V, p) \geq acc(V, p_b)$ and $cost(V, p) \leq cost(V, p_b)$. Intuitively, if a plan increases $min_{v_i \in V} acc(v_i, p)$, then it would increase the accuracies of all individual values.

Solving problems $T_1 - T_3$ for only a subset of values

The above problems $T_1 - T_3$ consider all values in V. In certain cases, the user may be interested in optimizing for only a subset of values $S \subseteq V$, such as the top 10 most difficult values, according to the ranking K. In such cases, we can formulate problems similar to $T_1 - T_3$, but replace V with S where appropriate.

Finding Plans That Satisfy General Constraints: In certain cases, the user does not have a baseline plan p_b to compare against. Instead, he or she just wants to find an "optimal" plan that satisfies certain constraints about cost and accuracy. Many variations exist. In this paper we consider the following:

T_4: Minimize cost while keeping accuracy above a threshold

Return the plan p that minimizes $cost(V, p)$, subject to constraint $acc(V, p) \geq \alpha$.

T_5: Maximize accuracy while keeping cost below a threshold

Return the plan p that maximizes $acc(V, p)$, subject to constraint $cost(V, p) \leq \beta$.

Solving problems $T_4 - T_5$ for only a subset of values

Again, in certain cases, the user may be interested in optimizing for only a subset of values $S \subseteq V$. In such cases, we can formulate problems similar to $T_4 - T_5$, but replace V with S where appropriate.

5.2 Solving Crowdsourcing Problems

We have described how users can express a variety of CS problems for detecting data errors. We now discuss how to solve them. The main idea is to formulate them as integer linear programming (ILP) optimization problems, solve these problems to find an optimal CS plan $p* = \langle\langle v_1, t_1\rangle, \ldots, \langle v_r, t_r\rangle\rangle$, then execute $p*$.

In what follows we discuss how to carry out the above idea for problem type T_1, then briefly discuss problem types $T_2 - T_5$. Recall that in problem type T_1, we want to find the plan p that minimizes $cost(V, p)$, subject to constraints $acc(V, p) \geq acc(V, p_b)$ and $cost(V, p) \leq cost(V, p_b)$. We now discuss how to estimate the quantities $cost(V, p)$, $cost(V, p_b)$, $acc(V, p_b)$, and $acc(V, p)$.

Estimating $cost(V, p)$ and $cost(V, p_b)$: It is straightforward to compute $cost(V, p_b)$, the crowdsourcing cost of the baseline solution. Recall that V is the set of values. Suppose each value v_i has n_i questions, then the total number of questions is $\sum_{i=1}^r n_i$. Since t_b answers need to be collected per question, $cost(V, p_b) = t_b \sum_{i=1}^r n_i$.

To compute $cost(V, p)$, recall that in our solution, for each value v_i, we have sampled x questions and collected y answers per sampled question. If plan p states that t_i answers will be collected for each remaining question, then the cost spent on value v_i will be $t_i(n_i - x) + xy$. Then the total cost on V can be computed as $cost(V, p) = \sum_{i=1}^r (t_i(n_i - x) + xy)$.

However, we cannot use t_i's as variables in the resulting ILP optimization problem (because constraints involving them will not be linear). To handle this problem, we use a set of indicator variables to represent t_i. Specifically, suppose $t_{\min}$ and $t_{\max}$ are the min and max number of answers to be collected per question (these two values are pre-specified; $t_{\min}, t_{\max}$ need to be odd positive integers since majority vote is used for aggregation). Let $A = \{t_{\min}, t_{\min} + 2, \ldots, t_{\max}\}$. Clearly, all t_i's are in A. To represent t_i, for each $j \in A$ we create an indicator variable h_{ij}. That is, if $j = t_i$, then $h_{ij} = 1$; otherwise $h_{ij} = 0$ for all $j \neq t_i$. We have $t_i = \sum_{j \in A} j h_{ij}$ and $cost(V, p) = \sum_{i=1}^r ((\sum_{j \in A} j h_{ij})(n_i - x) + xy)$. As we will see shortly, our ILP formulation uses this formula for $cost(V, p)$.

Estimating $acc(V, p_b)$: Let m_i be the number of questions with value v_i whose aggregated answers are correct, then $acc(V, p_b) = (\sum_{i=1}^r m_i)/(\sum_{i=1}^r n_i)$, where n_i is the number of questions for value v_i.

To estimate m_i, for each question q with value v_i we need to compute the probability that q's aggregated answer is correct, which depends on the number of collected answers. Recall that we assume that all questions with value v_i have the same difficulty and workers are i.i.d. (i.e., identically independently distributed) for each value. When we collect the same number of answers per question for a value, the aggregated answers of those questions will have the same probability of being correct.

We define $f_{i,t}$ as the probability that for any question q with value v_i, q's aggregated answer is correct when t answers are collected per question. So if the baseline approach collects t_b answers per question, then $m_i = n_i f_{i,t_b}$, where n_i is the number of questions in region i (for value v_i). We now describe how to compute $f_{i,t}$ for any i and t.

To compute $f_{i,t}$, we use the worker accuracy a_i for value v_i. Since we assume that workers are i.i.d. for value v_i, when t answers are collected for question q in region i, these answers are independent and each answer has the probability a_i of being correct. So the number of correct answers follows the binomial distribution $B(t, a_i)$. Since we use majority voting, q's aggregated answer is correct if and only if more than half of the collected answers of q are correct. So we can compute

$$f_{i,t} = \sum_{j=\lceil t/2 \rceil}^{t} \binom{t}{j} a_i^j (1 - a_i)^{t-j} \tag{1}$$

For each value v_i, we compute f_{i,t_b} using (1), then estimate m_i's and $cost(V, p_b)$ as described above. (At the end of this subsection we will describe how we use the rankings from Section 4 to adjust a_i's for all v_i's.)

Estimating $acc(V, p)$: Recall that $A = \{t_{\min}, t_{\min} + 2, \ldots, t_{\max}\}$ and $f_{i,t}$ is the probability that for any question q with value v_i, q's aggregated answer is correct when t answers are collected per question. Using the indicator variables described earlier, the expected probability that q's aggregated answer is correct can be estimated as $\sum_{j \in A} h_{ij} f_{i,j}$. Then the overall accuracy of our approach is computed as $acc(V, p) = \frac{\sum_{i=1}^r (x + (n_i - x)(\sum_{j \in A} h_{ij} f_{i,j}))}{\sum_{i=1}^r n_i}$.

Formulating T_1 as an ILP Problem: We now can formulate problem T_1 as the following ILP problem:

$$\begin{aligned}
\underset{\substack{h_{ij} \forall j \in A,\\ i=1,2,\ldots r}}{\text{minimize}} \quad & \sum_{i=1}^r (xy + (\sum_{j \in A} j h_{ij})(n_i - x)) \\
\text{subject to} \quad & \frac{\sum_{i=1}^r (x + (n_i - x)(\sum_{j \in A} h_{ij} f_{i,j}))}{\sum_{i=1}^r n_i} \geq \alpha_b \\
& \sum_{i=1}^r (xy + (\sum_{j \in A} j h_{ij})(n_i - x)) \leq t_b \sum_{i=1}^r n_i \\
& \sum_{j \in A} h_{ij} = 1 \quad \forall i = 1, 2, \ldots, r \\
& h_{ij} \in \{0, 1\} \quad \forall j \in A, \forall i = 1, 2, \ldots, r
\end{aligned} \tag{2}$$

The objective function is the total number of answers to be collected, which should be minimized. The first constraint ensures that the overall accuracy is same or better than that of the baseline approach (here α_b is $acc(V, p_b)$). The second constraint ensures that the total cost is no more than that of the baseline. The third constraint ensures that for each value, only one indicator variable is equal to 1. Finally, we solve the above ILP problem using the Gurobi solver [1], and return any solution found to the user, as the crowdsourcing plan p to be executed. We have

Proposition 5.1. *Let t_{min} and t_{max} be the minimal and maximal number of answers that the user wants to solicit for each question. Let t_b be the number of answers that the baseline solution solicit for each question, and x be the number of questions that we sample per value for the difficulty estimation step. If $t_{min} \leq t_b \leq t_{max}$ and $t_b \geq x$, then Equation 2 always has at least one solution.*

Solving Problems $T_2 - T_5$: So far we have discussed solving problem T_1. Problems T_2, T_4, and T_5 can be transformed similarly. For T_2, we only need to change the objective to maximize $acc(V, p)$ (which is the left side part of first constraint in problem T_1). For T_4 (or T_5), we only need to replace the estimated baseline accuracy (or cost) with the given accuracy (or cost) threshold from problem T_1 (or T_2) and remove the unnecessary constraint on cost (or accuracy).

Solving T_3, which maximizes $\min_{v_i \in V} acc(v_i, p)$, is a bit more involved. Let z be the minimum value accuracy among values in V, then the objective function of the transformed optimization is simply to maximize z, and it must add a constraint for each value v_i in V to ensure the accuracy of v_i is at least z (Constraint 3 in Equation 3). Therefore, we formulate problem T_3 as

$$\begin{aligned}
&\underset{\substack{z, h_{ij} \forall j \in A,\\ i=1,2,\ldots r}}{\text{maximize}} \quad z \\
&\text{subject to} \quad \frac{\sum_{i=1}^{r}(x + (n_i - x)(\sum_{j \in A} h_{ij} f_{i,j}))}{\sum_{i=1}^{r} n_i} \geq \alpha_b \\
&\qquad \sum_{i=1}^{r}(xy + (\sum_{j \in A} j h_{ij})(n_i - x)) \leq t_b \sum_{i=1}^{r} n_i \\
&\qquad \frac{x + (n_i - x)(\sum_{j \in A} h_{ij} f_{i,j})}{n_i} \geq z \qquad \forall v_i \in V \\
&\qquad \sum_{j \in A} h_{ij} = 1 \quad \forall i = 1, 2, \ldots, r \\
&\qquad h_{ij} \in \{0, 1\} \quad \forall j \in A, \forall i = 1, 2, \ldots, r
\end{aligned} \tag{3}$$

We have described how to solve problems $T_1 - T_5$ in the cases where they involve the set of all values V. It is easy to see that these problems can be solved in a similar fashion if they involve only a subset of values $S \subseteq V$.

Using the Ranking to Adjust Worker Accuracies: Recall that for each value $v_i \in V$, we have obtained xy answers from the crowd, and have estimated the worker accuracy for v_i as a_i, the fraction of the xy answers that are correct.

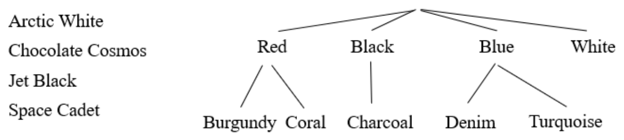
However, a_i is often not a good estimation of the true worker accuracy for v_i, because the set of xy answers is often small (e.g., $x = 4$ and $y = 5$ for 20 answers total). Thus, we seek to improve these estimations, using the ranking K. Our key idea is that if v_i is ranked higher than v_j, thus being perceived as being more difficult, then the worker accuracy for v_i should be no higher than that of v_j. If this is not the case, then we can adjust such worker accuracies so that they become more consistent with the ranking K.

Specifically, suppose K assigns to each value $v_i \in V$ a rank $k_i \in [1, r]$, where a smaller k_i indicates a value closer to the top of the ranked list. Then we model the task of improving the worker accuracies a_i's as the following optimization problem:

$$\begin{aligned}
&\underset{z_1, z_2, \ldots, z_r}{\text{minimize}} \quad \sum_{i=1}^{r}(z_i - a_i)^2 \\
&\text{subject to} \quad 0 \leq z_i \leq z_j \leq 1 \quad \forall i, j \in [1, r] \text{ s.t. } k_i < k_j
\end{aligned} \tag{4}$$

Here z_i is the improved worker accuracy for value v_i, and the cost function is the sum of the squares of $z_i - a_i$ (also known as L_2 cost function). Its constraint ensures that each value has the same or less worker accuracy than any easier value. This model is a simple Isotonic Regression problem, which always has a solution. It can be efficiently solved in $O(r)$ time [23], where r is the number of values. We solve it using Gurobi [1]. We then set $a_i = z_i$ and use a_i's as the worker accuracies in formulating ILP problems, as discussed earlier in this section.

Table 1: An example of handling ambiguous colors.



6 MANAGING AMBIGUOUS VALUES

As discussed in Section 1, in practice, there are many cases when the value for the target attribute is inherently ambiguous, such as "desert sand" and "arctic white". In such cases even the expert has trouble determining what should be the correct answer to the question, let alone asking the crowd workers. Such cases are surprisingly common, and no existing work has addressed them, as far as we can tell.

In this paper we provide a simple yet effective solution to this problem, based on what we have seen working well in industry. Briefly, we ask the expert E to first create a taxonomy Z of only unambiguous values, such as the one in Figure 1. Then the expert E examines each value v in V (the set of all values for the target attribute in the data set D). If E judges v to be inherently ambiguous, E should map v to a value $m(v)$ in Z.

Example 6.1. Table 1 shows a set of values (on the left side of the figure) that are ambiguous. The expert can map "Arctic White" to node "White" in the taxonomy, "Chocolate Cosmos" to "Burgundy", and so on.

A question such as "is the color of this product indeed chocolate cosmos?" is then transformed into "is the color of this product indeed burgundy?", which is unambiguous for crowd workers to answer.

7 EMPIRICAL EVALUATION

We now evaluate our solution. First, we crawled online sources to obtain the three datasets shown under "Datasets A" in Table 2. Their schemas are shown at the top of the table, with the target attribute underlined. Column "# Items" lists the number of data items in each dataset, and column "# Values" lists the number of values for the target attribute.

Since it would be too expensive to crowdsource all items in all datasets, we downsample all three datasets (using stratified sampling in which for each value of the target attribute, we randomly retain only 20% of the data items with that value). The new datasets are listed under "Datasets B" in the same table. Our experiments with real crowd workers are performed on these new datasets. We used Amazon Mechanical Turk for crowdsourcing, and used common turker qualifications, such as allowing only turkers with at least 100 approved HITs and 95% approval rate.

7.1 Learning to Rank

We first examine the performance of learning to rank. Recall that for each dataset, we sample x questions for each value, and then solicit y answers for each question. Thus, the expert must provide golden answers for $x|V|$ questions (where V is the set of

Table 2: Datasets for our experiments.

Products (title, description, picture, price, color)
Courses (title, description, department, #credits, subject)
Apparel (title, description, style, size, picture, category)

	Datasets A		Datasets B	
	# Items	# Values	# Items	# Values
Products	10,869	63	2,131	57
Courses	7,583	148	1,395	133
Apparel	3,480	12	690	11

Table 3: Evaluating the quality of the rankings.

	WAK			VChecker		
	Precision	Recall	F_1	Precision	Recall	F_1
Products	71.97	61.33	66.22	68.64	68.47	68.56
Courses	74.81	51.46	60.98	66.71	64.46	65.57
Apparel	76.67	63.89	69.70	72.22	72.22	72.22

all values for the target attribute), and the crowd must provide $xy|V|$ answers. So it is highly desirable that we minimize these two quantities, to minimize the workload of the expert and the crowd workers.

For our current three datasets, (x, y) are $(4, 5)$, $(5, 5)$, $(5, 5)$ for Products, Courses, and Apparel, respectively. Our iterative expansion process (to find x and y) converged for Products and Courses. These results suggest that indeed VChecker spends relatively little expert and crowd effort to compute the difficulty scores.

Next, we examine the quality of the ranking K of the values that we have obtained. To do so, we need a "golden" ranking $K*$. We obtain $K*$ as follows. First, for each value v, we collect A_v, the set of all answers obtained from the crowd for all questions involving v. Since we have obtained at least 9 answers for each question, this is usually a large number (in the hundreds). Next, we have identified the correct answer for all questions in our datasets, so we can compute the worker accuracy for v as the fraction of answers in A_v that is correct. Since A_v is a large set of answers, we take this worker accuracy to be the golden worker accuracy. Finally, we sort the values in decreasing order of these golden accuracies, to obtain a golden rank $K*$ of the values, in decreasing order of difficulties.

We now compare ranking K with $K*$. Direct comparison turns out to be difficult, because we often have two values v_i and v_j such that v_i is ranked above v_j in K and the reverse applies in $K*$, yet their difficulty scores differ by less than 0.01, say. In such cases, where the difficulty scores differ by less than a small ϵ threshold, we say the two values are not comparable. We translate ranking $K*$ into a set of $S(K*)$ of all (v_i, v_j) pairs that are comparable, and translate ranking K into a similar set $S(K)$.

Table 3 compares these sets. Consider the last three cells of the first row (the cells under "VChecker"). The cell under "Precision" is 68.64%, meaning 68.64% of pairs in $S(K)$ appear in $S(K*)$. The cell under "Recall" means 68.47% of pairs in $S(K*)$ appear in $S(K)$. These two numbers produce a F_1 score of 68.56%. Thus, for Products, the ranking K approximates the golden rank $K*$ quite well, with high precision and recall (though there is still room for improvement). Similar results are shown for Courses and Apparel.

Recall that the current popular solution in industry uses the average worker accuracy to rank the data values. Table 3 shows that the ranking produced by this solution is worse than the VChecker ranking (see the first three columns of the table, under "WAK", shorthand for "worker accuracy-based ranking", which show lower F_1 values). This result suggests that VChecker is indeed able to exploit additional information such as the response time and the worker disagreement to obtain a better ranking of value difficulties than the current existing solution.

Table 4: Evaluating the generated explanations.

Values	Explanations by VChecker	Explanations by Expert	# Compatible Explanations
P.Denim	2Ab,2Ac,2B,2C	2A,2C	3 {2Ab,2Ac,2C}
P.Brown	1,2A,2Ab,2B,2C	1C,2A,2Bc,2C	5 {1,2A,2Ab,2B,2C}
P.Turquoise	2A,2B,2C	2Ab,2Ac,2Bc,2C	3 {2A,2B,2C}
C.German	2A,2B,2C	2A,2Bc,2C	3 {2A,2B,2C}
C.Zoology	2A,2C	2A,2B,2C	2 {2A,2C}
C.Dance	1A,1Ab,2A,2C	1A,2C	3 {1A,1Ab,2C}
A.Tanks	2Aa,2B,2Ba,2Bb,2C	2A,2B,2C	5 {2Aa,2B,2Ba,2Bb,2C}
A.Underwear	2A,2B,2C	2Bc,2C	2 {2B,2C}
A.Socks	2A,2C	2C	1 {2C}

Table 5: VChecker vs. the UCS baseline solution.

Dataset	t_b	Cost			Accuracy	
		UCS	VChecker	Reduction	UCS	VChecker
Products	3	6,393	4,435	30.6	96.10	95.53
	5	10,655	5,961	44.1	96.89	96.03
	7	14,917	7,585	49.2	97.27	96.18
	9	19,179	8,941	53.4	97.49	96.32
Courses	3	4,185	4,063	2.9	95.94	96.08
	5	6,975	5,393	22.7	96.83	97.18
	7	9,765	6,639	32.0	97.17	97.60
	9	12,555	7,715	38.6	97.56	97.69
Apparel	3	2,070	2,016	2.6	97.60	97.62
	5	3,450	2,384	30.9	98.03	97.82
	7	4,830	2,866	40.7	98.36	97.97
	9	6,210	3,202	48.4	98.84	98.02

7.2 Generating Explanations

To evaluate our explanation generator, for each dataset we select 3 values in the top part of the ranking K, then ask for their explanations. For comparison purposes, we also ask a domain expert to manually generate explanations, after carefully examining all the answers solicited from the crowd.

Table 4 lists the explanations for VChecker vs those generated by the experts. "2Ab" for example is the explanation at node "2.A.b" in the taxonomy of explanations in Figure 3 ("v is indeed difficult because it is unfamiliar"). The table shows that the two sets of explanations share large overlaps (see the last column of the table), suggesting that VChecker is effective in generating explanations to explain why a value is considered difficult.

7.3 Finding Good Crowdsourcing Plans

We now show that VChecker can find good crowdsourcing plans for a variety of problem types. Table 5 compares VChecker to the baseline plan of soliciting the same number t_b of answers for each data value. We call this plan UCS, shorthand for "uniform crowdsourcing".

To explain, consider the third row of this table. It shows that for dataset Products, if UCS solicits $t_b = 3$ answers per question, then it incurs a total crowdsourcing cost of 6,393 answers. If

Table 6: VChecker vs UCS in solving problem T_3.

Dataset	Min Value Accuracy		Avg Value Accuracy	
	UCS	VChecker	UCS	VChecker
Products	68.45	83.33	92.34	96.11
Courses	72.45	74.68	96.11	96.81
Apparel	92.23	95.58	97.46	98.30

we solve the CS problem T_1 (as described in Section 5.1; we experiment with other CS problem types below) to find a better CS plan, which would minimize this cost while keeping accuracy at least equal or better than that of UCS, then the cost of this new plan (listed under column "VChecker") is 4,435. This produces a reduction of 30.6% in cost. The last two cells of this row show that the accuracies of UCS and VChecker are comparable (96.1 vs 95.53) [1]. (We obtained these accuracy numbers by executing both plans on Amazon Mechanical Turk.) Subsequent rows are similar, but for different values of t_b.

The table shows that VChecker can significantly reduce the cost of the baseline solution UCS, by 22.7-53.4% in all cases, except two cases where the reduction is a more modest 2.6% and 2.9%. It also shows that the accuracy of VChecker is comparable to that of UCS (with the difference in the range [-1.17%, 0.43%]).

Solving Other Types of CS Problems: Earlier we have shown how VChecker solves CS problems of type T_1. We now show that VChecker is effective in helping users solve other types of CS problems.

In Section 5.1 we discuss problem T_3, where the user wants to improve the accuracies of the values *across the board* as much as possible, while keeping the overall accuracy at least as high as that of the baseline plan p_b and keeping the overall cost at most as high as that of p_b. The goal is to return the plan p that maximizes $min_{v_i \in V} acc(v_i, p)$, subject to $acc(V, p) \geq acc(V, p_b)$ and $cost(V, p) \leq cost(V, p_b)$.

Table 6 shows how well VChecker performs for this problem. The column "UCS" shows the minimal value accuracy (i.e., the lowest accuracy among those of all values) when it solicits 3 answers for each question. The column "VChecker" shows that VChecker is able to improve this minimal accuracy significantly, while keeping the cost no higher than the cost of UCS. The last two columns show that even the average value accuracy (i.e., averaged over all values) of VChecker is higher than that of UCS.

In Section 5.1 we also discuss the problem of maximizing the accuracy of k most difficult values, as taken from the ranking K. Table 7 shows that VChecker is effective for solving this problem, improving the accuracy of the top 5 most difficult values per dataset significantly.

7.4 Additional Experiments

Sensitivity Analysis: In the current VChecker system we set $x_{max} = 5$ and $y_{max} = 5$, meaning that the iterative exploration process (see Section 3.2) never goes beyond these values. Figure 4 shows how iterative exploration is sensitive to varying these values. It shows that this process converges between 3 and 6 for all three datasets, suggesting that setting the values to 5 is a reasonable choice.

[1] When solving the ILP problem, we specified the constraint that VChecker has the same or better accuracy than UCS. When executing the found plan on Mechanical Turk, however, this constraint may not hold, due to spammers, careless workers, etc. Nevertheless, our experiments show that the accuracies of VChecker and UCS differ by a very small range.

Table 7: Maximizing accuracy of 5 most difficult values.

Dataset	Values	Value Accuracy		
		UCS	VChecker	% Improved
Products	Coral	68.45	83.33	14.88
	Denim	81.07	100.00	18.93
	Taupe	76.96	100.00	23.04
	Brown	95.55	97.92	2.37
	Camel	98.52	100.00	1.48
Courses	La Follette School of Public Affairs (PUB AFFR)	78.04	87.50	9.46
	Agronomy (AGRONOMY)	96.13	100.00	3.87
	Geological Engineering (G L E)	93.45	100.00	6.55
	Civil and Environmental Engineering (CIV ENGR)	97.11	100.00	2.89
	German (GERMAN)	87.22	96.90	9.68
Apparel	Underwear	98.43	100.00	1.57
	Tanks	92.23	100.00	7.77
	Pants	94.72	96.85	2.13
	Socks	96.87	96.90	0.03
	Swimwear	99.34	97.35	-1.99

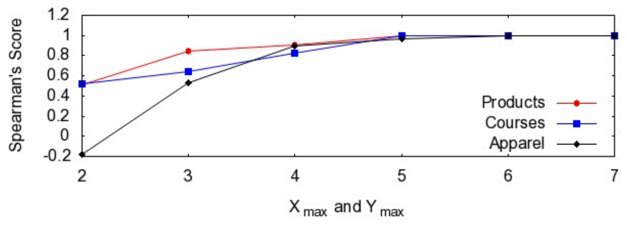


Figure 4: Convergence in iterative exploration.

Managing Ambiguous Values: Finally, we briefly discuss examples of managing ambiguous values. In our experiments it turns out that Products has ambiguous values. Specifically, it has a total of 173 values, 110 of which are considered ambiguous and have to be mapped to 63 values in a taxonomy of unambiguous values. Examples of such mappings include Arctic White mapped to White, Fluorescent Orange mapped to Orange Red, and Saddle Brown mapped to Brown. This clearly suggests that managing such ambiguous values is critical in real-world verification of attribute values.

8 RELATED WORK

Data cleaning has received enormous attention (e.g., [2–4, 6, 9, 11, 12, 16, 32–37, 40–45, 50]). See [5, 7, 10, 46] for recent tutorials, surveys, and books. However, as far as we can tell, no published work has examined the problem of manually detecting data errors in categorical attributes, as we do in this paper.

In recent years, crowdsourcing (CS) has received significant attention and has also been applied to many data cleaning problems (e.g., [8, 14, 15, 20, 22, 25, 28, 30, 31, 47, 49]). Among these, the work [14] also discusses the idea of adapting crowdsourcing strategies to the difficulties of data regions. However, it considers this idea in the setting of crowdsourcing for active learning. Further, it does not consider learning to rank the data regions, nor debugging the ranking, as we do this paper.

A critical challenge in CS is that the quality of workers varies. Researchers have proposed many methods to differentiate workers, such as filtering out spammers [25, 47], measuring the reliability and quality of workers [19, 22, 27, 39], and finding the right group of workers for a given task [17]. These methods usually assume that all the questions are of the same difficulty. In contrast, VChecker utilizes the difficulty heterogeneity among the questions while assuming that all workers have the same quality.

VChecker uses majority voting to aggregate the collected answers of each question. Many other aggregation methods have been proposed [19, 22, 27, 39]. They usually assign higher weights to answers from workers with good quality, then perform weighted aggregation. Many build probabilistic models [22, 39] to iteratively update the estimation of worker quality and weights. However, as far as we can tell, there is no published work yet showing conclusive evidence that these methods can achieve higher accuracy than majority voting, especially when we can only collect a small number of answers per question due to limited budget, as in our setting here.

Researchers also propose other methods to reduce CS cost, e.g., early-stopping strategies [13, 21, 29]. They stop collecting more answers for a question when they realize that collecting more answers will not change the aggregated answer. Such methods can also be used in VChecker to further reduce our cost, when we use the best plan returned by VChecker to crowdsource all the questions.

Finally, most CS works only collect answers from the crowd. [26] also collects the self-reported confidence from workers to improve the accuracy of aggregated answers. However, they also notice that workers have the tendency to overestimate or underestimate their confidence. Recently [18] proposes to collect the time spent by workers to measure CS effort. VChecker also collects the response times, but use these (and other data) to estimate question difficulty.

9 CONCLUSIONS

Detecting data errors completely manually is a ubiquitous problem in data cleaning, yet it has not received much attention. In this paper we have shown that the current common solution of crowdsourcing the above problem using the same number of answers per question can be improved by detecting the difficulties of data regions, then adjusting the number of answers required for each region based on its difficulty. We showed that current work using this idea has several significant limitations. We proposed VChecker, a novel solution to address these limitations, and described extensive experiments with three real-world data sets that demonstrate the promise of our solution. For future work, we plan to improve VChecker in multiple ways, including developing solutions to partition the input data into regions, better solutions to estimate and rank the data regions' difficulties, and better solutions to generate explanations for domain experts.

REFERENCES

[1] [n.d.]. Gurobi. http://www.gurobi.com/.

[2] Z. Abedjan et al. 2016. Detecting Data Errors: Where are we and what needs to be done? *PVLDB* 9, 12 (2016), 993–1004.

[3] A. Arasu et al. 2011. Towards a Domain Independent Platform for Data Cleaning. *IEEE Data Eng. Bull.* 34, 3 (2011), 43–50.

[4] S. Chaudhuri et al. 2006. Data Debugger: An Operator-Centric Approach for Data Quality Solutions. *IEEE Data Eng. Bull.* 29, 2 (2006), 60–66.

[5] Xu Chu et al. 2016. Data Cleaning: Overview and Emerging Challenges. In *SIGMOD.*

[6] X. Chu et al. 2016. Distributed Data Deduplication. In *VLDB.*

[7] X. Chu and I. F. Ilyas. 2016. Qualitative Data Cleaning. *PVLDB* 9, 13 (2016).

[8] S. Das et al. 2017. Falcon: Scaling Up Hands-Off Crowdsourced Entity Matching to Build Cloud Services. In *SIGMOD.*

[9] A. Das Sarma et al. 2012. An automatic blocking mechanism for large-scale de-duplication tasks. In *CIKM.*

[10] T. Dasu and T. Johnson. 2003. *Exploratory Data Mining and Data Cleaning.* John Wiley.

[11] X. Dong et al. 2010. Global Detection of Complex Copying Relationships Between Sources. *PVLDB* 3, 1 (2010), 1358–1369.

[12] V. Efthymiou et al. 2015. Parallel Meta-blocking: Realizing Scalable Entity Resolution over Large, Heterogeneous Data. In *Big Data.*

[13] Aditya G. Parameswaran et al. 2012. CrowdScreen: algorithms for filtering data with humans. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012.* 361–372.

[14] B. Mozafari et al. 2014. Scaling Up Crowd-Sourcing to Very Large Datasets: A Case for Active Learning. *PVLDB* 8, 2 (2014), 125–136.

[15] C. Gokhale et al. 2014. Corleone: hands-off crowdsourcing for entity matching. In *SIGMOD.* 601–612.

[16] D. Haas et al. 2015. Wisteria: Nurturing Scalable Data Cleaning Infrastructure. *PVLDB* 8, 12 (2015), 2004–2007.

[17] H. Li et al. 2014. The wisdom of minority: discovering and targeting the right group of workers for crowdsourcing. In *WWW.* 165–176.

[18] J. Cheng et al. 2015. Measuring Crowdsourcing Effort with Error-Time Curves. In *ACM CHI.* 1365–1374.

[19] J. Whitehill et al. 2009. Whose vote should count more: Optimal integration of labels from labelers of unknown expertise. In *Advances in neural information processing systems.* 2035–2043.

[20] J. Wang et al. 2012. CrowdER: Crowdsourcing Entity Resolution. *PVLDB* 5, 11 (2012), 1483–1494.

[21] L. Mo et al. 2013. *Optimizing task assignment for crowdsourcing environments.* Technical Report. Citeseer.

[22] P. Ipeirotis et al. 2010. Quality management on amazon mechanical turk. In *ACM SIGKDD workshop on human computation.* ACM, 64–67.

[23] P. Mair et al. 2009. Isotone optimization in R: pool-adjacent-violators algorithm (PAVA) and active set methods. *Journal of statistical software* 32, 5 (2009), 1–24.

[24] Q. Deng et al. 2015. Deep learning for gender recognition. In *ICCCS.* 206–209.

[25] S. Jagabathula et al. 2014. Reputation-based Worker Filtering in Crowdsourcing. In *Advances in Neural Information Processing Systems 27.* 2492–2500.

[26] S. Oyama et al. 2013. Accurate Integration of Crowdsourced Labels Using Workers' Self-reported Confidence Scores. In *IJCAI.*

[27] V. Raykar et al. 2009. Supervised learning from multiple experts: whom to trust when everyone lies a bit. In *ICML.* 889–896.

[28] X. Chu et al. 2015. KATARA: A Data Cleaning System Powered by Knowledge Bases and Crowdsourcing. In *SIGMOD.* 1247–1261.

[29] X. Liu et al. 2012. CDAS: A Crowdsourcing Data Analytics System. *PVLDB* 5, 10 (2012), 1040–1051.

[30] Y. Amsterdamer et al. 2013. CrowdMiner: Mining association rules from the crowd. *PVLDB* 6, 12 (2013), 1250–1253.

[31] Y. Tong et al. 2014. CrowdCleaner: Data cleaning for multi-version data on the web via crowdsourcing. In *IEEE.* 1182–1185.

[32] Z. Khayyat et al. 2015. BigDansing: A System for Big Data Cleansing. In *SIGMOD.* 1215–1230.

[33] M. J. Franklin et al. 2011. CrowdDB: answering queries with crowdsourcing. In *SIGMOD.*

[34] J. Freire et al. 2016. Exploring What not to Clean in Urban Data: A Study Using New York City Taxi Trips. *IEEE Data Eng. Bull.* 39, 2 (2016), 63–77.

[35] Helena Galhardas et al. 2001. Declarative Data Cleaning: Language, Model, and Algorithms. In *VLDB.*

[36] D. Haas et al. 2016. CLAMShell: Speeding up Crowds for Low-latency Data Labeling. In *VLDB.*

[37] J. Heer et al. 2015. Predictive Interaction for Data Transformation. In *CIDR.*

[38] T. Joachims. 2002. Optimizing search engines using clickthrough data. In *SIGKDD.* 133–142.

[39] A. Khan and H. Garcia-Molina. 2017. CrowdDQS: Dynamic Question Selection in Crowdsourcing Systems. In *SIGMOD.*

[40] L. Kolb et al. 2011. Parallel Sorted Neighborhood Blocking with MapReduce. In *BTW.*

[41] S. Krishnan et al. 2016. ActiveClean: Interactive Data Cleaning For Statistical Modeling. *PVLDB* 9, 12 (2016).

[42] A. Marcus et al. 2011. Crowdsourced databases: Query processing with people. In *CIDR.*

[43] B. Mozafari et al. 2014. Scaling Up Crowd-Sourcing to Very Large Datasets: A Case for Active Learning. In *VLDB.*

[44] A. G. Parameswaran and N. Polyzotis. 2011. Answering Queries using Humans, Algorithms and Databases. In *CIDR.*

[45] H. Park and J. Widom. 2013. Query Optimization over Crowdsourced Data. In *VLDB.*

[46] E. Rahm and H. H. Do. 2000. Data Cleaning: Problems and Current Approaches. *IEEE Data Eng. Bull.* 23, 4 (2000).

[47] V. Raykar and S. Yu. 2012. Eliminating Spammers and Ranking Annotators for Crowdsourced Labeling Tasks. *Journal of Machine Learning Research* 13 (2012), 491–518.

[48] C. Spearman. 1987. The Proof and Measurement of Association between Two Things. *The American Journal of Psychology* 100, 3/4 (1987), 441–471.

[49] V. Verroios et al. 2017. Waldo: An Adaptive Human Interface for Crowd Entity Resolution. In *SIGMOD.*

[50] J. Wang et al. 2013. Leveraging Transitive Relations for Crowdsourced Joins. In *SIGMOD.*