

**TOWARD MINIMIZING COST, ERROR AND RUNTIME
FOR DATA LABELING**

by

Haojun Zhang

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2020

Date of final oral examination: 04/17/2020

The dissertation is approved by the following members of the Final Oral Committee:

AnHai Doan, Professor, Computer Sciences, UW-Madison

Anthony Gitter, Assistant Professor, Biostatistics and Medical Informatics, UW-Madison

Paraschos Koutris, Assistant Professor, Computer Sciences, UW-Madison

Xiangyao Yu, Assistant Professor, Computer Sciences, UW-Madison

© Copyright by Haojun Zhang 2020

All Rights Reserved

To People Who Fight Against COVID-19

ACKNOWLEDGMENTS

This dissertation would not have been accomplished without the kind supports and help of many individuals. I would like to extend my sincere gratitude to all of them.

Foremost, I would like to express my gratitude to my advisor, Professor AnHai Doan, for his continuous support and advising over my whole PhD journey. I have learned a lot from AnHai. Specifically, I sincerely thank him pointing out my weakness in communication and the importance of communication skills, which will continue benefiting my career in the future.

Next, I would like to thank my dissertation committee members: Professors Anthony Gitter, Paris Koutris, and Xiangyao Yu for their invaluable inputs. I would also like to thank Professors Jeffrey Naughton, Jignesh Patel, and Theodoros Rekatsinas for their valuable comments and suggestions to my research projects.

I am grateful to all other co-authors, collaborators, and individuals that have helped me during my PhD life. An incomplete list includes Paul Suganthan, Fatemah Panahi, Han Li, Yash Govind, Pradap Konda, Adel Ardalan, Jeff Ballard, Kaushik Chandrasekhar, Sanjib Das, and many other students from our wonderful Database group.

I am thankful to all lecturers from courses that I have taken during my PhD journey, who have help broaden my horizon and enrich my life.

Finally, I would like to thank my family for their supports and encouragement over these years. May the force be strong with us!

TABLE OF CONTENTS

	Page
LIST OF TABLES	vi
LIST OF FIGURES	viii
ABSTRACT	ix
1 Introduction	1
1.1 Data Labeling Tasks	2
1.2 Contribution of the Dissertation	4
1.3 Roadmap of the Rest of This Dissertation	5
2 Validating Data Using Crowdsourcing	6
2.1 Introduction	6
2.2 Problem Definition	11
2.3 Ranking the Data Regions	14
2.3.1 Splitting Data into Regions	15
2.3.2 Learning to Rank the Regions	15
2.4 Debugging the Ranking	19
2.5 Finding Good Plans	24
2.5.1 Expressing Crowdsourcing Problems	25
2.5.2 Solving Crowdsourcing Problems	27
2.6 Managing Ambiguous Values	31
2.7 Empirical Evaluation	32
2.7.1 Learning to Rank	33
2.7.2 Generating Explanations	34
2.7.3 Finding Good Crowdsourcing Plans	36
2.7.4 Additional Experiments	37
2.8 Related Work	39
2.9 Conclusions	40
3 Debugging Labeled Data For Entity Matching	41
3.1 Introduction	41
3.2 Problem Definition	43

	Page
3.3 FPN	45
3.3.1 The Overall Solution	46
3.3.2 Reduce Latency between Iterations	47
3.3.3 Utilize Multicores	50
3.4 Mono	52
3.4.1 Sort Probing	55
3.4.2 Reduce Latency between Iterations	56
3.4.3 Revised Version	58
3.4.4 Utilize Multicores	59
3.5 Combining Detectors	59
3.6 Empirical Evaluation	63
3.6.1 Interactive Error Detection	65
3.6.2 Runtime and Scalability	68
3.6.3 Comparison between Detectors	70
3.6.4 Sensitivity Analysis	70
3.7 Related Work	72
3.8 Conclusions	73
4 Large-Scale Active Learning for Entity Matching	74
4.1 Background and Problem Definition	74
4.1.1 Large-Scale Active Learning for Entity Matching	75
4.2 Our First Solution	78
4.2.1 A Cluster Solution Using Spark	78
4.2.2 Empirical Evaluation	80
4.3 Our Second Solution	83
4.3.1 Feature Computation in Entity Matching	83
4.3.2 Speed up Feature Computation with Caching Techniques	84
4.3.3 Number of Features over Iterations	87
4.3.4 Lazy Feature Computation	90
4.3.5 Empirical Evaluation	92
4.4 Our Third Solution	94
4.4.1 Utilize User Labeling Time	94
4.4.2 Empirical Evaluation	95
4.5 Related Work	96
4.6 Conclusions and Future Work	98
5 Conclusions	100
Bibliography	102

	Page
Appendices	111

LIST OF TABLES

Table	Page
2.1 An example of handling ambiguous colors.	32
2.2 Datasets for our experiments.	33
2.3 Evaluating the quality of the rankings.	34
2.4 Evaluating the generated explanations.	35
2.5 VChecker vs. the UCS baseline solution.	35
2.6 VChecker vs UCS in solving problem T_3	37
2.7 Maximizing accuracy of 5 most difficult values.	38
3.1 Datasets for our experiments.	64
3.2 Statistics of detected label errors.	66
3.3 Runtime with our system.	69
3.4 Runtime of the first iteration with multicores.	70
3.5 Performance comparison between detectors.	71
3.6 Sensitivity analysis of batch size.	71
3.7 Sensitivity analysis of error percentage.	72
4.1 Datasets for our preliminary experiments.	80
4.2 Runtime on these datasets.	81
4.3 Feature statistics.	87

Table	Page
4.4 Evaluation of caching and lazy feature computation.	93
4.5 Reducing user waiting time.	95

LIST OF FIGURES

Figure	Page
1.1 An example of validating the color attribute for products.	1
2.1 An example of manual error detection.	7
2.2 Creating training data for SVM Rank.	17
2.3 A taxonomy of explanations.	22
2.4 Convergence in iterative exploration.	38
3.1 An example of matching person entities.	43
3.2 Workflow of our system.	45
3.3 An example with two features.	55
3.4 An example using Sort Probing.	58
3.5 An example to combine two ranked lists.	62
4.1 An example of active learning workflow.	75
4.2 Examples of categorical attributes and their frequency distributions.	86
4.3 Number of features over iterations.	89
4.4 Model accuracy vs. number of labeled pairs over iterations.	97

ABSTRACT

Data labeling is the process of one or more users labeling a set of data instances using a set of given labels. It is a pervasive problem in many data science tasks, such as classification, data validation, tagging, etc. To collect high quality labels, data labeling usually requires a lot of manual effort. Researchers have developed many techniques to help data labeling process, such as crowdsourcing and active learning, but major challenges regarding cost, quality and scalability still remain. In this dissertation I develop solutions to address these challenges for several important data labeling tasks.

First, I have developed *VChecker* for validating data using crowdsourcing. In such tasks, crowdsourcing cost and accuracy of aggregated answers are two important factors that users often need to trade-off between each other. I developed solutions that estimate the difficulties of different values of the attribute to be validated and partition items in the dataset according to the value difficulties, then develop adaptive crowdsourcing strategies to crowdsource items in each partition. These solutions can be used towards different crowdsourcing task scenarios, such as minimizing crowdsourcing cost while maintaining the accuracy of aggregated answers, or maximizing the accuracy of aggregated answers with some budget limit, etc.

Second, in collaboration with Fatemah Panahi, I have developed solutions for detecting label errors for entity matching. Our interactive solutions significantly reduce the user workload. In each iteration, we find the top-k entity pairs whose labels are most suspicious, return them to the users for manual verification, then use their feedback (corrected labels) to find the top-k suspicious entity

pairs for the next iterations. We perform extensive experiments on 17 entity matching datasets, which demonstrate the promise of our solutions.

Finally, I have developed solutions for performing large-scale active learning for entity matching. We start with a simple distributed Spark solution, study its performance on many entity matching datasets, identify opportunities to improve user labeling experience, then we implement several ideas based on our observations and evaluate their effectiveness. Our empirical evaluations show that our solutions effectively improve user labeling experience by significantly reducing both the waiting time before users start labeling and the waiting time between iterations.

Chapter 1

Introduction

Data labeling is the process of one or more users labeling a set of data instances using a set of given labels ¹. It is a pervasive problem in many data science tasks, such as classification, data validation, tagging, etc. In classification tasks, we need labeled examples to train classifiers. For example, in entity matching, to train a matcher (which is a classifier), we usually require a set of labeled entity pairs (whether each pair refers to the same real-world entity, such as Dave Smith vs David D. Smith) before training. In data validation tasks, we often need to label whether a data instance is valid or not. Figure 1.1 shows an example of validating the color attribute of product instances: the given color value of the bag is blue, which is invalid judging from the product picture. In tagging tasks, we are asked to label a given content (e.g., post, picture, video, etc) with one or more tags from some provided taxonomy.

¹Our solutions in Chapter 3 also apply to data labeled using other means.

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 1.1: An example of validating the color attribute for products.

To collect high quality labels, data labeling usually requires a lot of manual effort. Researchers have developed many techniques to help data labeling process, such as crowdsourcing and active learning, but major challenges regarding cost, quality and scalability still remain. First, manually labeling thousands or more data instances usually takes a lot of time and monetary cost. Meanwhile, these labeled data usually contain label errors due to human mistakes, inconsistent labeling criteria, etc. Active learning is widely used to save labeling cost. Active learning is a machine learning method that iteratively selects a batch of most informative unlabeled pairs for users to label. However, when given large sets of data instances, active learning often has scalability issues. For example, in each iteration the labeling workers often need to wait a long time before they get the set of data instances to be labeled. In this dissertation I develop solutions to address these challenges for several important data labeling tasks, which I describe next.

1.1 Data Labeling Tasks

Specifically, my dissertation focused on three important data labeling tasks: (1) validating data using crowdsourcing, (2) detecting label errors in the training data for entity matching; and (3) performing active learning on large-scale entity matching datasets.

Validating Data Using Crowdsourcing: The first task I addressed is to validate data using crowdsourcing. Specifically, given a set of items D and a target attribute A , validate the correctness of attribute A 's values for all items in D using crowdsourcing. To do so, we transform each item into a question of the form "Is v the correct value of attribute A for item d based on the provided context of d ?", then crowd workers will label it with either "yes" or "no" as the answer. In such tasks, crowdsourcing cost and accuracy of aggregated answers are two important factors that users often need to trade-off between each other. I developed solutions that estimate the difficulties of different values of A and partition items in D according to the value difficulties, then develop adaptive crowdsourcing strategies to crowdsource items in each partition. These solutions can be

used towards different crowdsourcing goals, such as minimizing crowdsourcing cost while maintaining the accuracy of aggregated answers, or maximizing the accuracy of aggregated answers with some budget limit, etc.

Detecting Label Errors for Entity Matching: The second task I addressed is to detect label errors for entity matching. In entity matching, given two tables of entities, we usually perform blocking first to remove pairs of entities between the two tables that are obviously NOT matches and get a set of candidate pairs that are likely matches. Then we need a matcher to predict whether each candidate pair is a match or not. To build the matcher, we often need to manually label a subset of candidate pairs, then use them as the training data. However, there are often many label errors in such training data. It would require a lot of effort if users go through each entity pair and verify the correctness of its label. I developed interactive solutions to detect label errors iteratively, which significantly reduce the user workload. In each iteration, we find the top-k entity pairs with the most suspicious labels, return them to the users for manual verification, then use their feedback (corrected labels) to find the top-k suspicious entity pairs for the next iterations.

Large-Scale Active Learning for Entity Matching: The third task I addressed is to perform active learning on large-scale entity matching datasets. As mentioned before, in entity matching, after blocking we often get a set of candidate pairs U and need to label some pairs for building the matcher. Instead of randomly sampling a small set of pairs for labeling, active learning is used to iteratively select a batch of most controversial unlabeled pairs and send them to users for labeling, until some stopping criteria are activated (e.g., it reaches the maximum number of iterations, or the learned model has converged). However, given two large tables of entities, we often have a large set of candidate pairs, which brings scalability issues. For example, it usually takes a lot of time to do feature computation for pairs in U and apply trained models to select the most controversial pairs for labeling, which means that each iteration users need to wait a long time before they get the pairs to be labeled.

The project goal is to reduce the waiting time. Specifically, we aim to reduce (1) the start time (i.e., time from users uploading the dataset to getting the first batch of controversial pairs to be

labeled; and (2) the iteration latency (i.e., time from receiving the current batch of labeled pairs to returning the next batch of controversial pairs for users to label).

To achieve our goals, I first developed a basic solution on Spark, evaluated it on several entity matching datasets and studied its behaviors. Then I identified several opportunities to further reduce the above types of waiting time. I implemented and integrated these ideas into our Spark solution, performed extensive experiments, which demonstrates the promise of our revised solutions.

1.2 Contribution of the Dissertation

My dissertation focuses on developing solutions for several challenges in the above data labeling tasks. My work has made the following contributions:

1. I developed *VChecker*, which uses adaptive crowdsourcing strategies for data validation (Chapter 2). *VChecker* has successfully addressed many limitations in the existing solution and significantly advanced this line of research in many ways. Our empirical evaluation on three real datasets shows that compared with the existing solution, *VChecker* can help reduce crowdsourcing cost by up to 53% while achieving comparable overall accuracy, or reduce the error rate of aggregated answers by up to 35% with the same budget limit.
2. I developed solutions to debug labeled data for entity matching (Chapter 3). We perform extensive experiments on 17 entity matching datasets (which are widely used in industry and research, and are often assumed with no label errors), and our solutions find label errors in 12 of them. The experiment results also demonstrate that our solutions can significantly reduce the user workload and help find such errors effectively.
3. I developed solutions to perform large-scale active learning for entity matching (Chapter 4). Our goals are to help improve user labeling experience during active learning. We evaluate our solutions on several datasets and demonstrate the promise of our solutions: we help significantly reduce both the waiting time (up to 95%) before users start labeling and the waiting time between iterations.

1.3 Roadmap of the Rest of This Dissertation

The rest of this dissertation is organized as follows. First, I describe our VChecker system, which helps reduce the cost when validating data using crowdsourcing in Chapter 2. Next, I describe our interactive solutions to debug label errors for entity matching in Chapter 3. After that, in Chapter 4, I describe how we address the problem of large-scale active learning for entity matching. I conclude this dissertation with Chapter 5.

Chapter 2

Validating Data Using Crowdsourcing

This chapter studies the problem of validating data using crowdsourcing, focusing on detecting attribute value errors. I first discuss the limitations of current common crowdsourcing solutions, then describe how I develop our *VChecker* system to address those limitations. I evaluate our system using three real datasets and demonstrate its promise.

The chapter is organized as follows. I first introduce the background and define the problem in Section 2.2. Then I describe details of our *VChecker* system in Sections 2.3-2.6. I present our experiment results in Section 2.7. After that, I briefly describe the related work, then conclude the chapter with discussions and future work.

2.1 Introduction

Data cleaning has received significant recent attention (e.g., [16, 18, 23, 79]), 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 [16, 18, 23, 79] (e.g., outliers, incorrect values, duplicate tuples, and constraint violations). In this chapter 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” [16]). 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.

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 2.1: An example of manual error detection.

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’s algorithms, but much easier for human users. This often happens when an attribute value is buried in a picture or text.

Example 2.1.1. Consider the product in Figure 2.1. 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, 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 chapter we consider manually detecting data errors ¹. 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* [16, 18, 23, 79]. 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 2.1, 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),

¹This problem is also often referred to as a data validation problem.

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 2.4.1 lists other usages of ranking K .

While the above solution has been quite popular in industry, it has several important limitations. In this chapter 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 chapter we develop a simple but effective solution to this problem.

Contributions: To summarize, in this chapter 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.

2.2 Problem Definition

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

Data Items, Attributes, and Values: For manually detecting data errors, many problem types exist. As a start, in this chapter we will consider the problem of manually verifying the correctness of the categorical values of a target attribute. Specifically, let $D = \{d_1, \dots, d_n\}$ 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., $d_i = \langle a_1 = v_{i1}, \dots, a_m = v_{im} \rangle$. 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”: $c_i = \langle a_1 = v_{i1}, \dots, a_{m-1} = v_{i(m-1)} \rangle$.

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 $Q = \{q_1, \dots, q_n\}$, 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 2.1).

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 [42, 37, 34, 61]. 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 chapter 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 2.2.1. 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 2.4.1).

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 chapter we introduce our solution, called VChecker, which addresses the above limitations.

2.3 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.)

2.3.1 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 2.1, 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 $D = \{d_1, \dots, d_n\}$ be the set of data instances, a_m be the target attribute, $Q = \{q_1, \dots, q_n\}$ be the set of questions “is the value of attribute a_m of instance d_i indeed v_{im} ?”, and $V = \{v_1, \dots, v_r\}$ 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.

2.3.2 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 sampled 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.

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 2.4). 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 $V = \{v_1, \dots, v_r\}$ 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.

$$\begin{array}{ll}
V = \{\text{black, red, iris,} & G = \{\langle \text{black}, f_1 \rangle, \langle \text{red}, f_2 \rangle, \\
\text{lavender, chartreuse}\} & \langle \text{chartreuse}, f_5 \rangle\} \\
\text{(a)} & \text{(c)} \\
F = \{f_1 = \langle 1, 2.3, 0 \rangle, & \{\text{chartreuse}\} \geq \{\text{black, red}\} \\
f_2 = \langle 1, 2.4, 0.05 \rangle, & \text{(d)} \\
f_3 = \langle 0.7, 5.1, 0.1 \rangle, & \\
f_4 = \langle 0.6, 8.4, 0.1 \rangle, & S = \{(f_5, 1), (f_1, 2), (f_2, 2)\} \\
f_5 = \langle 0.7, 11.2, 0.15 \rangle\} & \text{(e)} \\
\text{(b)} &
\end{array}$$

Figure 2.2: Creating training data for SVM Rank.

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 2.3.1. Consider the five colors in set V in Figure 2.2.a. Figure 2.2.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 $F = \{f_1, \dots, f_r\}$, 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 [60].

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 knowledge² 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 2.3.2. Continuing with Example 2.3.1, suppose we have selected the three colors black, red, and chartreuse for creating the training data (see Figure 2.2.c). Suppose the expert specifies that chartreuse is considered more difficult than both black and red (Figure 2.2.d). Then we can create the training set S in Figure 2.2.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 2.4.

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

²The expert can divide G into more groups as long as he/she can order values between different groups.

(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 [86], 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 $\text{Spearman}(K(x_{n-2}, y_{n-2}), K(x_{n-1}, y_{n-1}))$ and the score $\text{Spearman}(K(x_{n-1}, y_{n-1}), K(x_n, y_n))$ 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 2.1 shows the pseudo code of the entire process to rank the data regions.

2.4 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 2.4.1. The ranking K can be used to re-calibrate the worker accuracies of the values (see Section 2.5.2). 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 2.5.1). 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 2.4), 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 chapter we will develop a way to generate explanations, which can help the expert debug K .

Algorithm 2.1 Learning to Rank the Data Regions

Require: 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) , ϵ : convergence threshold for ranking, n_0 : number of training examples for SVM Rank

Ensure: 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  $V$  us-
   ing  $F, O$ 
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 2.4
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  $i = 1, 2, \dots, n$  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  $i = 1, 2, \dots, m$  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  $i = 1, 2, \dots, m$  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  $\text{len}(L) < 3$  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

```

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 2.4.2. 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 chapter 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 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 2.3. 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

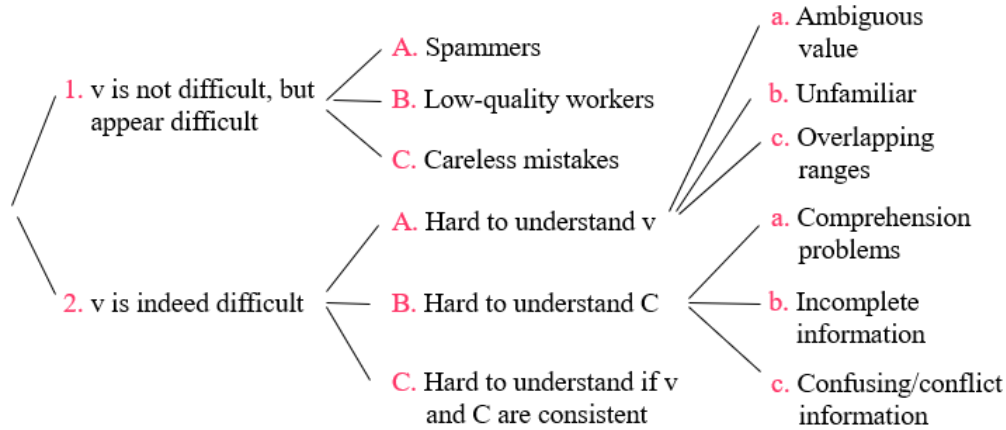


Figure 2.3: A taxonomy of explanations.

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 2.3) 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 2.2 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

Algorithm 2.2 Generating Explanations

Require: v : the value to be explained

Ensure: 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 sampled 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 α and average time t for A
 - 3: **for** each $w \in W$ **do**
 - 4: Apply a procedure on R_w using α, 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: $E_w \leftarrow \text{GENWORKEREXPLANATIONS}(W_v)$
 - 8: Update S to S^+ by removing answers and the time stats from spammers and low-quality workers
 - 9: Apply a procedure on R_v using S^+, α, t to classify the nature N_v of value v (i.e., ambiguous, unfamiliar, overlapping)
 - 10: $E_v \leftarrow \text{GENVALUEEXPLANATIONS}(N_v)$
 - 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 α, 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: $E_q \leftarrow \text{GENQUESTIONEXPLANATIONS}(N_q)$
 - 17: $E_v \leftarrow E_w \cup E_v \cup E_q$
 - 18: **return** E_v
-

“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 identify 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).

2.5 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 2.1, 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.

2.5.1 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 S , return the plan that maximizes or minimizes an objective O , subject to a constraint C , where O and C involve P and S , and optionally a ranking K of values*”. In this chapter 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 chapter 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 [29].

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 chapter 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.

2.5.2 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 $p^* = \langle \langle v_1, t_1 \rangle, \dots, \langle v_r, t_r \rangle \rangle$, 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 $A = \{t_{\min}, t_{\min} + 2, \dots, t_{\max}\}$. 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 $\text{cost}(V, p) = \sum_{i=1}^r ((\sum_{j \in A} j h_{ij})(n_i - x) + xy)$. As we will see shortly, our ILP formulation uses this formula for $\text{cost}(V, p)$.

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

$$f_{i,t} = \sum_{j=\lceil t/2 \rceil}^t \binom{t}{j} a_i^j (1 - a_i)^{t-j} \quad (2.1)$$

For each value v_i , we compute f_{i,t_b} using (2.1), 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 2.4 to adjust a_i 's for all v_i 's.)

Estimating $acc(V, p)$: Recall that $A = \{t_{\min}, t_{\min} + 2, \dots, t_{\max}\}$ 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:

$$\begin{aligned}
& \underset{\substack{h_{ij} \forall j \in A, \\ i=1,2,\dots,r}}{\text{minimize}} && \sum_{i=1}^r (xy + (\sum_{j \in A} j h_{ij})(n_i - x)) \\
& \text{subject to} && \frac{\sum_{i=1}^r (x + (n_i - x)(\sum_{j \in A} h_{ij} f_{i,j}))}{\sum_{i=1}^r n_i} \geq \alpha_b \\
& && \sum_{i=1}^r (xy + (\sum_{j \in A} j h_{ij})(n_i - x)) \leq t_b \sum_{i=1}^r n_i \\
& && \sum_{j \in A} h_{ij} = 1 \quad \forall i = 1, 2, \dots, r \\
& && h_{ij} \in \{0, 1\} \quad \forall j \in A, \forall i = 1, 2, \dots, r
\end{aligned} \tag{2.2}$$

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 [3], and return any solution found to the user, as the crowdsourcing plan p to be executed. We have

Proposition 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 2.2 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 2.3). Therefore, we formulate problem T_3 as

$$\begin{aligned}
& \underset{z, h_{ij} \forall j \in A, i=1, 2, \dots, r}{\text{maximize}} && z \\
& \text{subject to} && \frac{\sum_{i=1}^r (x + (n_i - x)(\sum_{j \in A} h_{ij} f_{i,j}))}{\sum_{i=1}^r n_i} \geq \alpha_b \\
& && \sum_{i=1}^r (xy + (\sum_{j \in A} j h_{ij})(n_i - x)) \leq t_b \sum_{i=1}^r n_i \\
& && \frac{x + (n_i - x)(\sum_{j \in A} h_{ij} f_{i,j})}{n_i} \geq z \quad \forall v_i \in V \\
& && \sum_{j \in A} h_{ij} = 1 \quad \forall i = 1, 2, \dots, r \\
& && h_{ij} \in \{0, 1\} \quad \forall j \in A, \forall i = 1, 2, \dots, r
\end{aligned} \tag{2.3}$$

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:

$$\begin{aligned} & \underset{z_1, z_2, \dots, z_r}{\text{minimize}} && \sum_{i=1}^r (z_i - a_i)^2 \\ & \text{subject to} && 0 \leq z_i \leq z_j \leq 1 \quad \forall i, j \in [1, r] \text{ s.t. } k_i < k_j \end{aligned} \tag{2.4}$$

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 [38], where r is the number of values. We solve it using Gurobi [3]. 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.

2.6 Managing Ambiguous Values

As discussed in Section 2.1, 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 chapter 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 2.1. 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 .

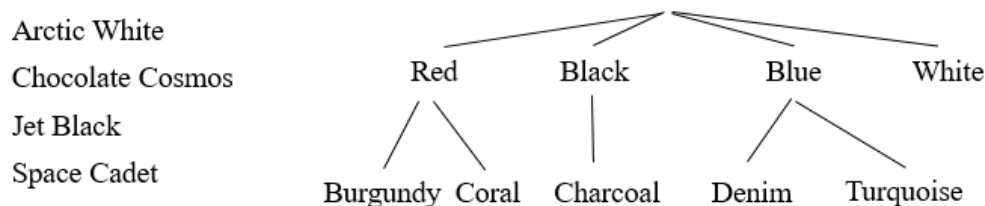


Table 2.1: An example of handling ambiguous colors.

Example 2.6.1. Table 2.1 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.

2.7 Empirical Evaluation

We now evaluate our solution. First, we crawled online sources to obtain the three datasets shown under “Datasets A” in Table 2.2. 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.

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 2.2: Datasets for our experiments.

2.7.1 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 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.

	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

Table 2.3: Evaluating the quality of the rankings.

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 2.3 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 2.3 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.

2.7.2 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

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 2.4: Evaluating the generated explanations.

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

Table 2.5: VChecker vs. the UCS baseline solution.

expert to manually generate explanations, after carefully examining all the answers solicited from the crowd.

Table 2.4 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 2.3 (“ 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.

2.7.3 Finding Good Crowdsourcing Plans

We now show that VChecker can find good crowdsourcing plans for a variety of problem types. Table 2.5 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 we solve the CS problem T_1 (as described in Section 2.5.1; 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)³. (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 2.5.1 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 $cost(V, p) \leq cost(V, p_b)$.

³When 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.

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

Table 2.6: VChecker vs UCS in solving problem T_3 .

Table 2.6 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 2.5.1 we also discuss the problem of maximizing the accuracy of k most difficult values, as taken from the ranking K . Table 2.7 shows that VChecker is effective for solving this problem, improving the accuracy of the top 5 most difficult values per dataset significantly.

2.7.4 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 2.3.2) never goes beyond these values. Figure 2.4 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.

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

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 (GER-MAN)	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

Table 2.7: Maximizing accuracy of 5 most difficult values.

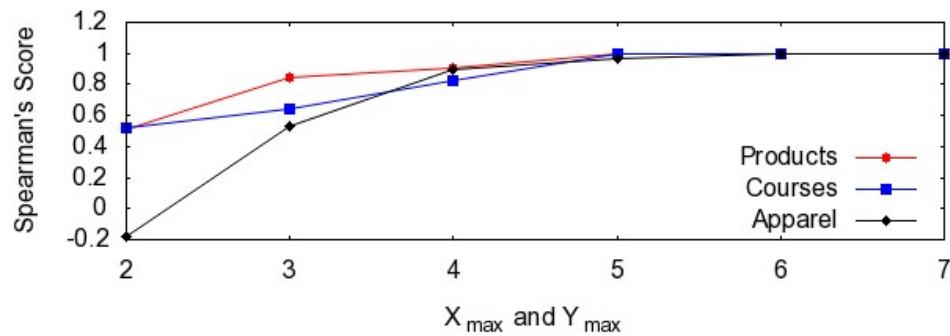


Figure 2.4: Convergence in iterative exploration.

clearly suggests that managing such ambiguous values is critical in real-world verification of attribute values.

2.8 Related Work

Data cleaning has received enormous attention (e.g., [52, 47, 31, 26, 21, 17, 63, 71, 75, 70, 76, 55, 49, 90, 66, 57, 7, 25, 50, 8, 15]). See [16, 18, 23, 79] 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 chapter.

In recent years, crowdsourcing (CS) has received significant attention and has also been applied to many data cleaning problems (e.g., [37, 40, 80, 35, 30, 45, 43, 20, 89, 46, 29]). Among these, the work [29] 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 chapter.

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 [80, 40], measuring the reliability and quality of workers [42, 37, 34, 61], and finding the right group of workers for a given task [32]. 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 [42, 37, 34, 61]. They usually assign higher weights to answers from workers with good quality, then perform weighted aggregation. Many build probabilistic models [37, 61] 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 [44, 36, 28]. 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. [41] 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 [33] 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.

2.9 Conclusions

Detecting data errors completely manually is a ubiquitous problem in data cleaning, yet it has not received much attention. In this chapter 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.

Chapter 3

Debugging Labeled Data For Entity Matching

This chapter studies the problem of debugging labeled data for entity matching. There are many challenges in this problem and I develop an interactive debugging system to help users debug label errors in such datasets. I perform extensive experiments on 17 entity matching datasets, which demonstrate the promise and effectiveness of our solutions.

The chapter is organized as follows. I first introduce the background, then define the problem of debugging labeled data for entity matching in Section 3.2. After that, I describe in details our solutions through Sections 3.3-3.5. I present our experiment results in Section 3.6. After that, I briefly describe the related work, then conclude the chapter with discussions and future work.

3.1 Introduction

Entity matching is the task to classify given pairs of entities into matched and non-matched pairs (a pair of entities is a match if they refer to the same real world entity). When doing entity matching, people often build one or more matchers and use them to predict the label of each pair (whether it is a match or not). Each matcher is usually a classification model such as Random Forests, therefore people need to label a set of entity pairs as the training data.

However, these labeled datasets often have labeled errors. For example, we find label errors in 12 datasets (out of total 17 entity matching datasets used in our experiments). These label errors are caused by many things. First, these datasets are often manually labeled by one or more workers and they may make mistakes. Also, when a worker labels many entity pairs, it is hard for him/her to follow the same matching criteria, which may cause inconsistency in the labels. Moreover,

users sometimes try to use semiautomatic or automatic approaches to label pairs (e.g., write some positive rule and automatically label all pairs satisfying the rule as matches). However, these approaches are often error-prone and likely cause many systematic errors. As a result, people often have to detect and fix these label errors before using them to build matchers with good accuracy.

To detect and fix label errors in such datasets, the current solution is usually to ask some analyst manually going through each labeled pair and verifying its label. Clearly this approach is expensive (in terms of both time and cost), and becomes infeasible when given a large labeled dataset. Therefore we build an interactive debugging system to help the analyst detect and fix label errors in these datasets. Our system interacts with the analyst iteratively. In each iteration our system returns a small set of most suspicious pairs to the analyst for manual verification, then use the feedback from him/her (i.e., fixed errors in those suspicious pairs) to help detect suspicious pairs for the next iterations. The iterative interaction continues until some stopping criteria are triggered (e.g., the maximum number of iterations has reached, or no label errors are found in the last three consecutive iterations, or the analyst decides to stop either because he/she thinks there is no or very few errors in the remaining pairs or the labels of all pairs have been verified).

In this chapter, we present our interactive debugging system that can help the analyst detect and fix label errors for entity matching tasks. Our contributions are:

- We argue that label errors are pervasive in labeled entity matching datasets, and they affect the accuracy of learned models. It is important to detect and fix label errors in these datasets when developing entity matching models.
- We have developed an interactive label debugging system to help reduce the user workload. As far as we can tell, it is the first such system to detect label errors for entity matching datasets. The system can utilize multiple error detectors to improve its precision and recall.
- We have developed two detectors: a learning based detector **FPN**, and a domain-knowledge based detector **MONO**. We evaluate them extensively on entity matching datasets to show the promise of these detectors.

	Name	City	State
a ₁	Dave Smith	Madison	WI
a ₂	Joe Wilson	San Jose	CA
a ₃	Dan Smith	Middleton	WI

	Name	City	State
b ₁	David D. Smith	Madison	WI
b ₂	Daniel W. Smith	Middleton	WI

Figure 3.1: An example of matching person entities.

- We perform incremental updates between iterations and utilize multicores to further improve the scalability of these detectors, which significantly reduce the latency between iterations.

3.2 Problem Definition

We now describe the problem of debugging labeled data for entity matching considered in this chapter.

Entities and Entity Matching: In this chapter, we define an entity to be a distinct real-world object (e.g., person, product, etc.). An entity is usually represented as a list of attribute value pairs. For example, the basic information of the 44th U.S. President can be represented as { name=Barack Obama, gender=Male, birth year=1961, political party=Democratic, ethnic group=African American }. Given a set of entity pairs, entity matching is the task to classify those pairs into matched and non-matched pairs (a pair of entities is considered a match if they refer to the same real world entity).

Example 3.2.1. *Figure 3.1 shows an example of matching person entities from two tables. Each person entity has attributes name, city and state. We wish to know which person from table A and which person from table B refer to the same person in the real world. In this example, Dave Smith from table A and David D. Smith from table B is the same person. Similarly, Dan Smith in table A matches Daniel W. Smith in table B.*

Dataset, Item Pairs, and Labels: Let $D = \{d_1, d_2, \dots, d_n\}$ be a set of entity pairs, each pair having a given label indicating whether the pair is a match or not. That is, each $d_i = \langle a_i, b_i, c_i \rangle$

where (a_i, b_i) is the pair of entities, and c_i is the given label, either 0 or 1. Our problem is to debug the labels in D . That is, for each d_i , determine whether c_i is the correct label for pair (a_i, b_i) . For example, if for some i , $c_i = 1$ but a_i and b_i doesn't refer to the same entity, then c_i is a label error.

Example 3.2.2. *Continue with the example in Figure 3.1. In total there are six entity pairs and suppose their labels are given as following: $(a_1, b_1, 1)$, $(a_1, b_2, 0)$, $(a_2, b_1, 0)$, $(a_2, b_2, 1)$, $(a_3, b_1, 0)$, and $(a_3, b_2, 0)$. Clearly, the labels in $(a_2, b_2, 1)$ and $(a_3, b_2, 0)$ are wrong.*

As mentioned earlier, to detect and fix label errors in such datasets, the current solution usually requires an analyst manually going through each pair and verifying its label. It might be necessary to do so if he/she has to fix all label errors, but we often only need to fix most of the label errors and the label quality is already good enough for training matchers. In that case, going through most pairs and verifying their labels would be too much workload for the analyst. It would be much better if the analyst only needs to verify a much smaller subset of pairs to fix most of the label errors. To achieve this goal, we build an interactive system to iteratively debug label errors.

Example 3.2.3. *Suppose the analyst needs to debug a dataset with 10,000 pairs, and the dataset contains only 500 label errors. The analyst may need to go through more than 9,000 pairs to find about 480 label errors, assuming label errors are uniformly randomly scattered in the dataset. Suppose a new solution returns a subset of 2,000 suspicious pairs for manual verification and this subset of pairs already contains ≥ 480 label errors, then the new solution helps save about 78% user workload.*

Our Interactive Debugging System: Figure 3.2 shows the workflow of our interactive debugging system. Given a dataset D of labeled entity pairs, our system first passes D through one or more detectors. Each detector will find a list of suspicious pairs and rank them in descending order of the error probabilities of their given labels (i.e, most likely wrong labels are ranked on the top). Next, the combiner combines those lists into a single ranked list of suspicious pairs and returns the top- k pairs to the analyst for manual verification. If the analyst finds any label errors and corrects them, those corrected labels will be used as feedback to the detectors to help improve the detection

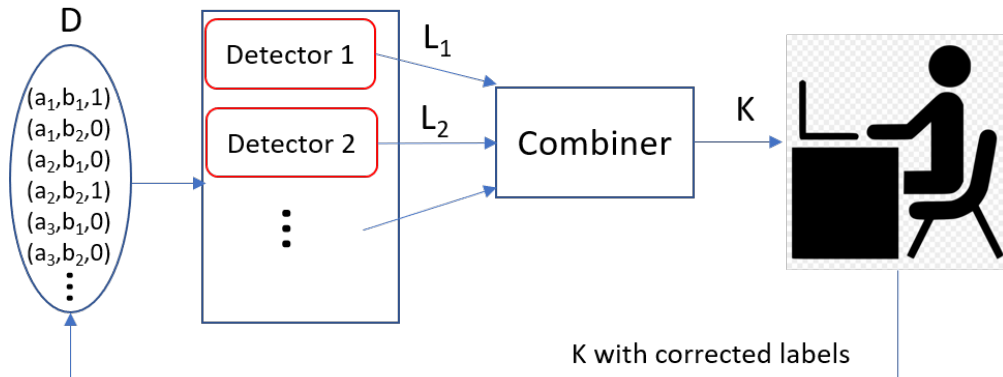


Figure 3.2: Workflow of our system.

of suspicious pairs for the next iterations. The above iterative interaction with the analyst will terminate when some prespecified stopping criteria are met (e.g., at most 30 iterations, or no label errors found in the last three consecutive iterations, or the analyst decides to stop because he/she feels that enough label errors have been fixed, or labels of all pairs have been verified). Finally, our solution outputs the list of entity pairs with corrected labels, which then can be used to train matchers.

In the next two sections we describe our learning based detector FPN and domain-knowledge based detector Mono. After that, we describe how we combine multiple ranked lists into one ranked list.

3.3 FPN

In this section we describe our FPN detector, which is a learning based approach. The key idea is to build classifiers to predict labels of each pair in the dataset, and pairs whose predicted labels differ from their given labels are likely wrong therefore sent to the analyst for manual verification.

Since it is a learning based approach, we need features that can be used to build those classifiers. Recall that each pair in D is in the form of $d_i = \langle a_i, b_i, c_i \rangle$ where (a_i, b_i) is the pair of entities and c_i is its given label (whether a_i and b_i are considered the same real-world entity). For each pair, if the user doesn't provide a feature vector, then we will use Magellan [64] to compute a set of features for each entity pair. Therefore each labeled pair is transformed into an example consisting of a feature

vector and the given label. Let B be the dataset containing the list of the transformed examples. To reduce the chance of overfitting when building those classifiers, we perform feature selection on B and get a new dataset C , which is the input to our detectors FPN and Mono. Suppose m features are selected, then example e_i in C can be represented as $e_i = \langle f_{1i}, f_{2i}, \dots, f_{mi}, c_i \rangle$ where f_{ki} is the value of the k -th selected feature f_k and c_i is the given label.

Feature Selection: There are many feature selection algorithms [62, 93, 87, 24]. To select useful features for the purpose of label debugging, we use one of the algorithms provided in the feature selection package of scikit-learn. Specifically, we use the `SelectFromModel` algorithm. This algorithm uses a given model to evaluate the importance of features, then selects the set of features according to user-specified selection criteria (e.g., top k features, or features whose importance is greater than or equal to given threshold, etc.) In our implementation, we choose Random Forests as the model to evaluate feature importance, and keep features whose importance is greater than or equal to the average feature importance over all features. Using feature selection, we have removed features that are less important and only kept features that are highly correlated with given labels, which in turn helps FPN detect suspicious label by avoiding overfitting the given labels (Feature selection also helps the other detector Mono, which will be explained later).

3.3.1 The Overall Solution

To detect label errors, FPN iteratively interacts with the analyst. In each iteration, it performs the following steps: first, it finds a set of pairs whose labels are likely wrong (i.e., suspicious pairs) and ranks them in descending order of their error likelihood; next, it returns the top suspicious pairs to the analyst for manual verification; when the verification is done, the correct labels of those pairs are sent back to FPN, which are used as the feedback to help improve error detection for the next iterations. FPN terminates the interaction when some prespecified stopping criteria are met (e.g., at most 30 iterations, or no label errors found in the last three consecutive iterations, or the analyst decides to stop because he/she feels that enough label errors have been fixed, or labels of all pairs have been verified). Now we describe how FPN finds the set of suspicious pairs and ranks them. For simplicity, we refer to the labels of suspicious pairs as suspicious labels.

Find and Rank Suspicious Labels: To find suspicious labels in C , FPN performs similar to classification with cross validation. That is, it first randomly partitions C into k -folds (e.g. $k = 5$) P_1, P_2, \dots, P_k ; then for each partition P_i , it trains a classifier M_i (e.g. Random Forests in our implementation) using examples in $C \setminus P_i$ and uses M_i to predict the labels of examples in P_i . Now from each partition we collect those examples whose predicted labels differ from their given labels to form the set of suspicious examples S .

How should we rank suspicious examples in S ? Those classifiers usually provide us their confidence scores of predicted labels, but scores from different classifiers are often not comparable, therefore we cannot use them directly to rank examples in S . To address this issue, FPN trains an additional classifier M on $C \setminus S$, then uses M to predict examples in S . It has two advantages. First, now the confidence scores from M can be used to rank suspicious examples in S . Second, since $C \setminus S$ likely has better label quality, M may have better accuracy than previous M_i 's. If for some example, its predicted label from M now matches its given label, then we decide not to send it for manual verification. Let S_0 be the set of examples in S whose predicted labels from M differ from their given labels, we rank them in descending order of their confidence scores from M . Algorithm 3.1 shows the complete pseudocode of our FPN detector.

3.3.2 Reduce Latency between Iterations

Recall that in each iteration, FPN needs to find the set of suspicious labels and rank them, which requires training of several classifiers and predicting labels with those classifiers. When C is large, training classifiers (line 3 and 11 in Algorithm 3.1) can take minutes or even longer. It would be impractical to let the analyst wait so long between iterations. To reduce the latency, we decide to use classifiers that support incremental updates (i.e., support adding and removing examples from the classifiers, etc.). In our implementation, we use Incremental Random Forest [4] (IRF for short), which is a variant of Random Forests [10]. To support incremental updates, IRF stores extra information in each tree. For example, for each tree it memorizes which examples are used to train the tree. When removing an example, it finds those trees that use the example during training, then traverses those trees to find the affected nodes and update the associated prediction probabilities in

Algorithm 3.1 FPN

Require: C : set of labeled examples, k : number of folds

Ensure: L : ranked list of suspicious examples

```

1: Partition  $C$  into  $k$  folds  $P_1, P_2, \dots, P_k$ 
2: for  $i = 1, 2, \dots, k$  do
3:   Train classifier  $M_i$  using  $C \setminus P_i$ 
4:   Use  $M_i$  to predict labels of examples in  $P_i$ 
5:    $S_i \leftarrow$  examples in  $P_i$  whose predicted labels differ from given labels
6: end for
7:  $S \leftarrow \bigcup_{i=1}^k S_i$ 
8:  $L \leftarrow \text{RANK}(S, C)$ 
9: return  $L$ 

10: procedure RANK( $S, C$ )
11:   Train classifier  $M$  using  $C \setminus S$ 
12:   Use  $M$  to predict labels of examples in  $S$ 
13:    $S_0 \leftarrow$  examples in  $S$  whose predicted labels differ from given labels
14:   Sort  $S_0$  in descending order of confidence scores from  $M$ 
15:   return  $S_0$ 
16: end procedure

```

those nodes. When adding a new example, it decides the trees that should use the new example for training, then inserts the example into those trees. With these operations for incremental updates, when a label error is detected, we can simply remove the associated example from those trees that use the example in training, then add the example with the correct label back to the IRF. Clearly, incremental updates in IRF would be much faster than training from scratch, which in turn reduces the iteration latency significantly. Algorithm 3.2 shows how we perform incremental update to existing classifiers M_1, M_2, \dots, M_k and M given a set of manually corrected examples E .

Limitations of IRF: IRF is fast in doing incremental updates, but there is no guarantee of the prediction accuracy after many incremental updates to the original random forests in IRF. In fact, the prediction accuracy is likely dropping when more and more incremental updates are performed. To solve this problem, we decide to utilize the time when the analyst manually verifies suspicious labels to train new random forests from scratch in the backend, which is described next.

Algorithm 3.2 Incremental Update

Require: C : set of labeled examples, k : number of folds, P_1, P_2, \dots, P_k : folds for cross-validation, M_1, M_2, \dots, M_k : classifiers from cross-validation, M : classifier to rank suspicious examples, S : previous suspicious examples, E : examples whose labels are manually corrected in current iteration

Ensure: L : new ranked list of suspicious examples

```

1: for  $i = 1, 2, \dots, k$  do
2:    $\Delta_i \leftarrow E \cap (C \setminus P_i)$ 
3:   Update classifier  $M_i$  using correct labels of  $\Delta_i$ 
4:   Use  $M_i$  to predict labels of examples in  $P_i$ 
5:    $S_i \leftarrow$  examples in  $P_i$  whose predicted labels differ from given labels
6: end for
7:  $A \leftarrow S \setminus \bigcup_{i=1}^k S_i$ 
8: Add examples in  $A$  to  $M$ 
9:  $B \leftarrow (\bigcup_{i=1}^k S_i) \setminus S$ 
10: Remove examples in  $B$  from  $M$ 
11:  $S \leftarrow \bigcup_{i=1}^k S_i$ 
12: Use  $M$  to predict labels of examples in  $S$ 
13:  $L \leftarrow$  examples in  $S$  whose predicted labels differ from given labels
14: Sort  $L$  in descending order of confidence scores from  $M$ 
15: return  $L$ 

```

Backend Training: Let C_1, C_2, \dots be the set of labeled examples for each iteration. That is, $C_1 = C$, C_2 is after we update C_1 using the feedback from the analyst in the first iteration, and C_{i+1} is after we update C_i using the feedback from the analyst in the i -th iteration. Let $\Delta(C_i, C_j)$ be the changes from C_i to C_j . Suppose \mathcal{F}_1 is the set of IRFs trained in the first iteration (\mathcal{F}_1 contains classifiers M_1, M_2, \dots, M_k and M , where k is the number of folds in cross-validation). In the second iteration, we first update \mathcal{F}_1 using the analyst's feedback, then find suspicious labels, rank them and return top k to the analyst. When the analyst manually verifies those k suspicious pairs, we start to train a new set of IRFs \mathcal{F}_2 on C_2 concurrently. When the analyst finishes verification, if \mathcal{F}_2 is not ready, then we will continue updating \mathcal{F}_1 and use it in the third iteration. If \mathcal{F}_2 is ready, then we replace \mathcal{F}_1 with \mathcal{F}_2 , then incrementally update \mathcal{F}_2 using $\Delta(C_2, C_3)$ before finding suspicious labels. Suppose \mathcal{F}_2 is ready only before iteration i , then after we replace \mathcal{F}_1 with \mathcal{F}_2 , we need to update \mathcal{F}_2 using $\Delta(C_2, C_i)$. Now \mathcal{F}_2 is ready for iteration i . When the analyst manually verifies top k suspicious labels from iteration i , we start to train a new IRF \mathcal{F}_i from scratch on C_i concurrently.

In general, at the beginning of each iteration, if the new set of IRFs is not ready, then we incrementally update the current set of IRFs and use it for the current iteration. If the new set of IRFs is ready, we replace the current set of IRFs with the new set and update the new set of IRFs with all changes in the training examples, then start training another new set of IRFs with latest training examples at the backend. Clearly, each new set of IRFs likely has better prediction accuracy than the current set of IRFs, therefore we can expect that our solution will find more label errors if many iterations are needed before termination. Ideally, if the backend training takes less time than the manual verification, then for iteration i , we are using the set of IRFs trained on C_{i-1} and updated using $\Delta(C_{i-1}, C_i)$, whose accuracy should be close to the set of IRFs trained from scratch (i.e., trained on C_i) since $\Delta(C_{i-1}, C_i)$ contains at most k examples. Algorithm 3.3 shows the pseudocode of backend training.

3.3.3 Utilize Multicores

We have described how we use IRFs to reduce the latency between iterations. However, when C is large, classifiers training takes long time to complete, which causes two problems: first, the analyst needs to wait long time before receiving the first set of suspicious pairs for verification; second, the backend training falls behind incremental updates by many iterations. To address this issue, we use multicores to speed up the training of classifiers M_1, M_2, \dots, M_k and M on C .

Recall that in cross-validation, we need to partition C into k folds and train k classifiers. Clearly, if each classifier can utilize all given cores for training, then we can sequentially train these classifiers effectively. What if each classifier can only use one core during training? Since during training, the classifier only needs to read its training examples as input, which means that training different classifiers are independent from each other by nature, therefore we can assign the training of each classifier to one core, which means we can use at most k cores to train the k classifiers (if there are fewer than k available cores, we can do a round-robin scheduling). Algorithm 3.4 shows how we assign each core a classifier for cross-validation.

Algorithm 3.3 Backend Training

Require: C : set of labeled examples, k : number of folds

```

1:  $t \leftarrow 0$ 
2:  $M_1, \dots, M_k, S, M, L \leftarrow \text{TRAIN}(C)$ 
3:  $x \leftarrow t$ 
4:  $C_x \leftarrow C$ 
5:  $K \leftarrow$  top  $k$  suspicious examples of  $L$ 
6: Send  $K$  for manual verification and wait for their verified labels
7:  $E \leftarrow$  examples in  $K$  whose labels are corrected by the analyst
8: Update  $C$  with  $E$ 
9:  $t \leftarrow t + 1$ 
10:  $y \leftarrow t$ 
11:  $C_y \leftarrow C$ 
12: Create a backend process to execute  $\text{TRAIN}(C_y)$ 
13: while Stopping criteria are NOT met do
14:   if Backend training has complete then
15:     Replace  $M_1, \dots, M_k, S, M, L$  with those from backend
       training
16:      $x = y$ 
17:      $C_x = C_y$ 
18:     Incremental update  $M_1, \dots, M_k, S, M, L$  using differ-
       ence between  $C$  and  $C_x$ 
19:      $y \leftarrow t$ 
20:      $C_y \leftarrow C$ 
21:     Create a backend process to execute  $\text{TRAIN}(C_y)$ 
22:   else
23:     Incremental update  $M_1, \dots, M_k, S, M, L$  using  $E$ 
24:   end if
25:    $K \leftarrow$  top  $k$  suspicious examples of  $L$ 
26:   Send  $K$  for manual verification and wait for their verified labels
27:    $E \leftarrow$  examples in  $K$  whose labels are corrected by the analyst
28:   Update  $C$  with  $E$ 
29:    $t \leftarrow t + 1$ 
30:    $y \leftarrow t$ 
31:    $C_y \leftarrow C$ 
32: end while
33: procedure  $\text{TRAIN}(C_t)$ 
34:   Partition  $C_t$  into  $k$  folds  $P_1, P_2, \dots, P_k$ 
35:   for  $i = 1, 2, \dots, k$  do
36:     Train classifier  $M_{it}$  using  $C_t \setminus P_{it}$ 
37:     Use  $M_{it}$  to predict labels of examples in  $P_{it}$ 
38:      $S_{it} \leftarrow$  examples in  $P_{it}$  whose predicted labels differ from
       given labels
39:   end for
40:    $S_t \leftarrow \bigcup_{i=1}^k S_{it}$ 
41:   Train classifier  $M_t$  using  $C_t \setminus S_t$ 
42:   Use  $M_t$  to predict labels of examples in  $S_t$ 
43:    $L_t \leftarrow$  examples in  $S_t$  whose predicted labels differ from given
       labels
44:   Sort  $L_t$  in descending order of confidence scores from  $M_t$ 
45:   return  $M_{1t}, \dots, M_{kt}, S_t, M_t, L_t$ 
46: end procedure

```

Algorithm 3.4 Multicore FPN

Require: C : set of labeled examples, k : number of folds

Ensure: L : ranked list of suspicious examples

```

1: Partition  $C$  into  $k$  folds  $P_1, P_2, \dots, P_k$ 
2: for  $i = 1, 2, \dots, k$  do
3:    $S_i \leftarrow$  Create a process to execute  $\text{PROCESSPARTITION}(C \setminus P_i, P_i)$ 
4: end for
5:  $S \leftarrow \bigcup_{i=1}^k S_i$ 
6:  $L \leftarrow \text{RANK}(S, C)$ 
7: return  $L$ 

8: procedure  $\text{PROCESSPARTITION}(D_t, D_p)$ 
9:   Train classifier  $M$  using  $D_t$ 
10:  Use  $M$  to predict labels of examples in  $D_p$ 
11:   $S \leftarrow$  examples in  $D_p$  whose predicted labels differ from given labels
12:  return  $S$ 
13: end procedure

14: procedure  $\text{RANK}(S, C)$ 
15:  Train classifier  $M$  using  $C \setminus S$ 
16:  Use  $M$  to predict labels of examples in  $S$ 
17:   $S_0 \leftarrow$  examples in  $S$  whose predicted labels differ from given labels
18:  Sort  $S_0$  in descending order of confidence scores from  $M$ 
19:  return  $S_0$ 
20: end procedure

```

3.4 Mono

In this section we describe our second detector **Mono**, which is a domain-knowledge based approach. In entity matching, similarity measures usually satisfy the monotonicity property, such as cosine and jaccard [56]. That is, the scores of similarity measures on matched pairs are usually greater than those scores on non-matched pairs. This property was first observed in [14]. Based on this observation, we designed our **Mono** detector, which interacts with the analyst in a way similar to **FPN**. What **Mono** differs from **FPN** is the first step of each iteration, that is, how to find suspicious labels and rank them.

Before describing Mono, we first define the monotonicity property between an example with label 1 and an example with label 0, then describe our motivations for using this property to detect label errors. Recall that we have m features for each example $e_i = \langle f_{1i}, f_{2i}, \dots, f_{mi}, c_i \rangle$.

Definition 1. Monotonicity at feature f *Given an example e_i with label 1 and an example e_j with label 0, let f_i and f_j be their values of feature f , then e_i and e_j satisfy monotonicity at feature f iff $f_i > f_j$.*

Example 3.4.1. *Continue with the example in Figure 3.1. If two persons are a match, we expect that they have the same or similar names so the string similarity score between their names are very high. In contrast, if two persons are a non-match, then they usually have different names, which leads to low score between their names. Therefore if we compare the string similarity scores between names for a matched pair of persons and a non-matched pair of persons, we likely notice that they satisfy the monotonicity property at this feature.*

Ideally, we wish that e_i and e_j satisfy monotonicity at all m features. However, in reality entities may have noise in the values of some attribute, or use synonyms, which may cause e_i and e_j to violate the monotonicity property at some features. Therefore we lower the requirement to satisfy monotonicity at at least k features (k -consistency).

Definition 2. k -consistency *Given an example e_i with label 1 and an example e_j with label 0, e_i and e_j is k -consistent iff e_i and e_j satisfy monotonicity at at least k different features.*

Clearly, using a larger k increases our confidence of the correctness of the labels for both e_i and e_j (but we may also find more false positives, i.e., some labels are misreported as suspicious though they are correct). As a start, we study the situation of 1-consistency (e.g., $k = 1$). For simplicity, from now on, e_i and e_j is called consistent if they are 1-consistent, otherwise they are called inconsistent.

Why do we care about the consistency between examples with different labels? If an example is a match but its given label is wrong (i.e., its given label is 0), then when we compare it with other examples with label 1, we are likely to find many inconsistencies. Similarly, we would find

many inconsistencies when we compare a non-matched example whose label is wrong (i.e., its given label is 1) with other examples with label 0. Therefore when an inconsistency is detected between pairs with different labels, the labels of both examples are suspicious and need manual verification. Clearly, if an example causes many inconsistencies, then we are more confident that its given label is likely wrong.

Based on the above motivation, to find suspicious labels, **MONO** identifies all inconsistencies between examples with label 1 and examples with label 0, then collects all the examples causing at least one inconsistency. Then the suspicious examples are ranked in descending order of the number of caused inconsistencies (Algorithm 3.5).

Algorithm 3.5 Mono

Require: C : set of examples

Ensure: L : ranked list of suspicious examples

- 1: Find the list of examples L that cause at least one inconsistency
 - 2: Sort L in descending order of the number of caused inconsistencies
 - 3: **return** L
-

How can we identify all inconsistencies? Naively, for each example e , we can compare it with all examples whose labels differ from e for consistency checking (Algorithm 3.6). Note that inconsistency property is symmetric (that is, if e_i is inconsistent with e_j , then e_j is also inconsistent with e_i), therefore we only need to find the list of inconsistent examples (with label 0) for each example with label 1. If we have n_0 examples with label 0 and n_1 examples with label 1, the naive algorithm has complexity $O(n_1 n_2 m)$, where m is the number of features. Clearly, this algorithm is computationally expensive when n_1, n_2 and m are large, which is common for datasets used in entity matching. To speed up the computation, we implement another algorithm to find all inconsistencies in the dataset: Sort Probing, which is described next.

Example 3.4.2. Consider the example in Figure 3.3. It has 16 labeled pairs, among which half are with label 1 and marked with blue circles, and the others are with label 0 and marked with red stars. With Naive Mono, we need to compare $8 \times 8 = 64$ pairs of feature vectors to find all inconsistencies.

Algorithm 3.6 Naive Mono

Require: C : set of examples

Ensure: X : dictionary that maps each example to its list of inconsistent examples

```

1:  $M \leftarrow$  examples in  $C$  whose given labels are 1
2:  $N \leftarrow$  example in  $C$  whose labels are 0
3:  $X \leftarrow$  empty dictionary
4: for  $e \in C$  do
5:    $X[e] \leftarrow$  empty list
6: end for
7: for  $e_1 \in M$  do
8:   for  $e_0 \in N$  do
9:     if  $e_0$  is inconsistent with  $e_1$  then
10:      Append  $e_0$  to  $X[e_1]$ 
11:      Append  $e_1$  to  $X[e_0]$ 
12:     end if
13:   end for
14: end for
15: Remove all examples  $e$  from  $X$  whose  $X[e]$  is empty
16: return  $X$ 

```

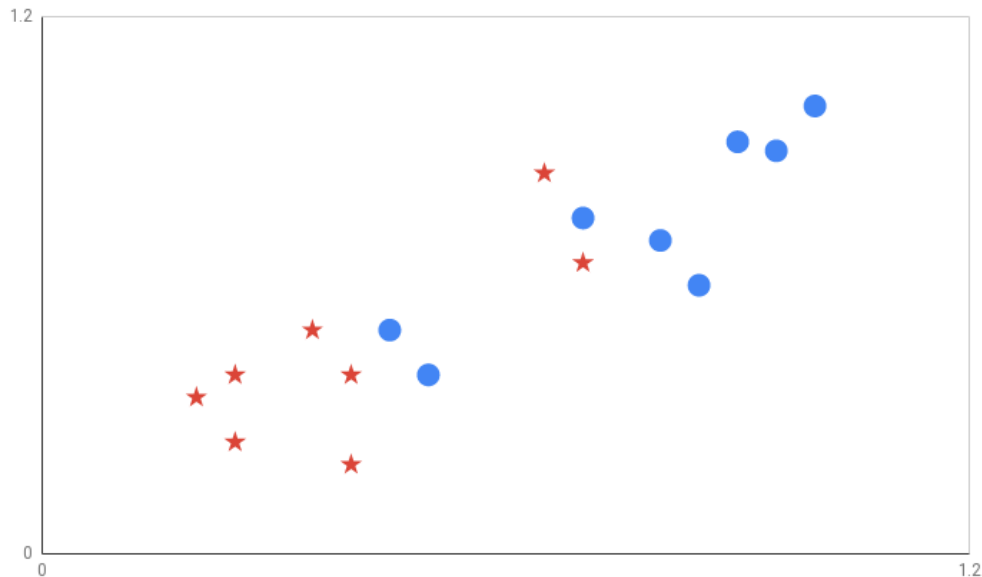


Figure 3.3: An example with two features.

3.4.1 Sort Probing

In the naive algorithm, given an example e , we compare it with all examples whose labels differ from e to find its list of inconsistent examples. Sort Probing uses a different method to find

this list. Recall that given an example e_i with label 1 and an example e_j with label 0, e_i and e_j are inconsistent if e_j has same or higher values than that of e_i at all features. Let L_k be the set of examples with label 0 and has same or higher values than that of e_i at feature k , then $\bigcap_{k=1}^m L_k$ is exactly the list of examples inconsistent with e_i . Motivated by this observation, we developed Sort Probing (Algorithm 3.7).

In Sort Probing, for each feature we need to find all L_k 's for all examples with label 1 (we don't need to do it for examples with label 0 due to the symmetric property of inconsistency). It would still be computationally expensive if we find these L_k 's independent from each other. To amortize the time cost when handling feature f , we first sort all examples with the same label in ascending order of their values at f . Let P be the sorted list of examples with label 1 and Q be the sorted list of examples with label 0. For example $e_i \in P$, let L_i be the sublist of Q with same or higher value than that of e_i at feature f . Since Q is sorted, to find L_i , we only need to find the first example e_j in Q whose value is same or higher than that of e_i , then L_i is the sublist from e_j to the end of Q . Therefore after sorting, we scan P and Q from their last examples. Let i, j be the indices of current examples in P and Q . If $P[i]$ has higher value than that of $Q[j]$, then L_i begins with $Q[j + 1]$, and we reduce i by 1; otherwise we reduce j by 1. In this way, we only need to scan P and Q one time to find L_i 's for all e_i 's in P .

Example 3.4.3. *Continue with the previous example. Consider the two blue pairs marked as A and B in Figure 3.4. Suppose $A = (x_1, y_1)$ and $B = (x_2, y_2)$. Let L_{ax} be the set of red stars with x -values greater x_1 and L_{ay} be the set of red stars with y -values greater than y_1 , then $L_{ax} \cap L_{ay}$ gives the set of red stars inconsistent with A. Let L_{bx} and L_{by} be the corresponding sets for B, then $L_{bx} \subseteq L_{ax}$ since $x_1 < x_2$, and $L_{ay} \subseteq L_{by}$ since $y_1 > y_2$.*

3.4.2 Reduce Latency between Iterations

Sort Probing is much faster than the naive algorithm. However, on a large dataset, it still takes minutes or longer to find all inconsistencies, which is intolerable between iterations. Therefore we decide to incrementally update the inconsistent lists of all examples if the label of some example is changed (because the analyst finds that its given label is wrong).

Algorithm 3.7 Sort Probing

Require: C : set of examples

Ensure: X : dictionary that maps each example to its list of inconsistent examples

```

1:  $M \leftarrow$  example in  $C$  whose given labels are 1
2:  $N \leftarrow$  examples in  $C$  whose given labels are 0
3:  $Y \leftarrow$  empty dictionary
4: for each feature  $f$  do
5:    $Y[f] \leftarrow$  empty dictionary
6:    $M_f \leftarrow$  Sort  $M$  in descending order of values for  $f$ 
7:    $N_f \leftarrow$  Sort  $N$  in ascending order of values for  $f$ 
8:   for  $e \in M_f$  do
9:      $L_f \leftarrow$  examples in  $N_f$  with values same or higher than that of  $e$ 
10:     $Y[f][e] \leftarrow L_f$ 
11:   end for
12: end for
13:  $X \leftarrow$  empty dictionary
14: for  $e \in N$  do
15:    $X[e] \leftarrow$  empty list
16: end for
17: for  $e_1 \in M$  do
18:    $L \leftarrow \bigcap_f Y[f][e_1]$ 
19:   if  $L$  is not empty then
20:      $X[e_1] \leftarrow L$ 
21:     for  $e_0 \in L$  do
22:       Append  $e_1$  to  $X[e_0]$ 
23:     end for
24:   end if
25: end for
26: Remove all examples  $e$  from  $X$  whose  $X[e]$  is empty
27: return  $X$ 

```

Suppose the label of example e is changed and L is its previous list of inconsistent examples. To perform incremental updates, we first remove e from the inconsistent lists of examples in L . Next, we correct the label of e , then compare e with examples whose labels differ from e to find its new list L' of inconsistent examples. Finally, we append e to the inconsistent lists of examples in L' . Now those examples with nonempty inconsistent lists are suspicious and we rank them by their number of inconsistencies.

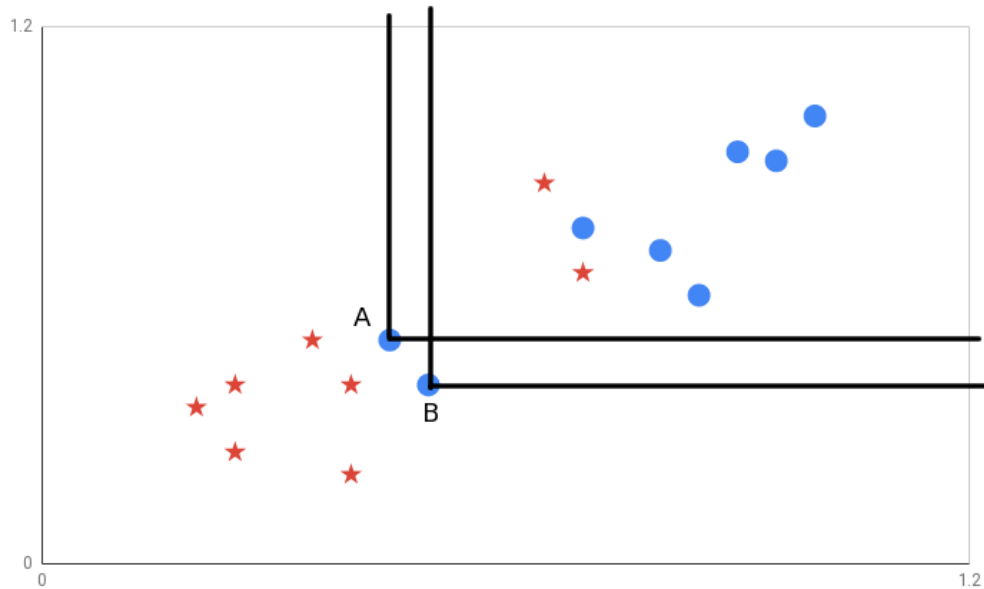


Figure 3.4: An example using Sort Probing.

Since in each iteration only top k suspicious examples are sent for manual verification, the number of examples whose labels are changed is at most k , therefore the above incremental update is much faster than finding all inconsistencies from scratch. However, the above Sort Probing (and also the naive algorithm) keeps all the inconsistent lists in memory, which could be an issue if there are a lot of inconsistencies. For example, given a dataset with 1M pairs (among which 500k pairs are with label 1 and the other 500k pairs are with label 0), suppose on average each pair p is inconsistent with 1% pairs whose labels differ from p (i.e., the average length of inconsistent list is $500,000 \times 1\% = 5,000$), then the total number of inconsistencies is $10^6 \times 5,000 = 5 \times 10^9$, which takes about 40GB memory (assuming it requires 8 bytes per inconsistency in memory). Next we describe how we address the memory issue.

3.4.3 Revised Version

How can we address the above memory issue? In the previous version, we notice that the inconsistent lists are used for two different purposes. First, we use the length of inconsistent lists to rank suspicious examples. Second, if after manual verification, the given label of a suspicious example p is wrong, we need to remove p from the inconsistent lists of examples in the inconsistent

list of p . Clearly, for the first purpose, we only need the length of those inconsistent lists. Therefore instead of storing those inconsistent lists, we only record down the length of the inconsistent list for each suspicious example. But now we need to figure out how to address the second purpose, i.e., how to do incremental update. Note that if the label of example e has changed, we need its previous inconsistent list to perform incremental update. Since each iteration only top- k suspicious examples are sent to the analyst for manual verification, the number of examples whose given labels are wrong cannot exceed k . Since k is usually small, reconstructing the previous inconsistent lists of such examples takes little time. Therefore we first reconstruct the previous inconsistent lists, then perform incremental update. Algorithm 3.8 shows the revised sort probing algorithm.

3.4.4 Utilize Multicores

Now we describe how we use multicores to speed up the computation of all inconsistencies in the first iteration (Algorithm 3.9). Suppose k cores are given, there are two places that we can do parallel processing. First, clearly processing each feature is independent from each other, therefore we can use $\min(k, m)$ cores to parallel process features, where m is the number of features. After that, for each example with label 1 we need to intersect its lists from all features to find its list of inconsistent examples. To use k cores, we partition the set of examples with label 1 into k partitions, then assign the processing of each partition to one core. Once all partitions are processed, we union the inconsistencies found by each core into the final inconsistent map.

3.5 Combining Detectors

If only one detector is used, we can simply return top k from its ranked list of suspicious pairs to the analyst for manual verification. When we use more than one detector, each detector will return a ranked list of suspicious pairs, then the combiner needs to combine them into a single ranked list. Clearly, if a pair is marked suspicious by many detectors, then its label is more likely wrong. Therefore if we only return those marked suspicious by all or most detectors for manual verification, those returned suspicious pairs will have high precision, but we might miss many pairs

Algorithm 3.8 Revised Sort Probing

Require: C : set of examples

Ensure: X : dictionary that maps each example to the length of its list of inconsistent examples

```

1:  $M \leftarrow$  example in  $C$  whose given labels are 1
2:  $N \leftarrow$  examples in  $C$  whose given labels are 0
3:  $Y \leftarrow$  empty dictionary
4: for each feature  $f$  do
5:    $Y[f] \leftarrow$  empty dictionary
6:    $M_f \leftarrow$  Sort  $M$  in descending order of values for  $f$ 
7:    $N_f \leftarrow$  Sort  $N$  in ascending order of values for  $f$ 
8:   for  $e \in M_f$  do
9:      $L_f \leftarrow$  examples in  $N_f$  with values same or higher than that of  $e$ 
10:     $Y[f][e] \leftarrow L_f$ 
11:   end for
12: end for
13:  $X \leftarrow$  empty dictionary
14: for  $e \in N$  do
15:    $X[e] \leftarrow 0$ 
16: end for
17: for  $e_1 \in M$  do
18:    $L \leftarrow \bigcap_f Y[f][e_1]$ 
19:   if  $L$  is not empty then
20:      $X[e_1] \leftarrow$  length of  $L$ 
21:     for  $e_0 \in L$  do
22:       Increase  $X[e_0]$  by 1
23:     end for
24:   end if
25: end for
26: Remove all examples  $e$  from  $X$  whose  $X[e]$  is 0
27: return  $X$ 

```

Algorithm 3.9 Multi-core Sort Probing

Require: C : set of examples, k : number of cores

Ensure: S : set of suspicious examples

```

1:  $M \leftarrow$  examples in  $C$  whose given labels are 1
2:  $N \leftarrow$  examples in  $C$  whose given labels are 0
3:  $Y \leftarrow$  empty dictionary
4: for each feature  $f$  do // parallel process
5:    $Y[f] \leftarrow$  empty dictionary
6:    $M_f \leftarrow$  Sort  $M$  in descending order of values for  $f$ 
7:    $N_f \leftarrow$  Sort  $N$  in ascending order of values for  $f$ 
8:   for  $e \in M_f$  do
9:      $L_f \leftarrow$  examples in  $N_f$  with values same or higher than that of  $e$ 
10:     $Y[f][e] \leftarrow L_f$ 
11:   end for
12: end for
13: Partition  $M$  into  $M_1, M_2, \dots, M_k$ 
14: for  $i = 1, 2, \dots, k$  do // parallel process
15:    $X_i \leftarrow$  PROCESSPARTITION( $M_i, Y$ )
16: end for
17:  $X \leftarrow \bigcup_{i=1}^k X_i$ 
18: Remove all examples  $e$  from  $X$  whose  $X[e]$  is empty
19: return  $X$ 

```

20: **procedure** PROCESSPARTITION(M, Y)

```

21:    $X \leftarrow$  empty dictionary
22:   for  $e_1 \in M$  do
23:      $L \leftarrow \bigcap_f Y[f][e_1]$ 
24:      $X[e_1] \leftarrow L$ 
25:     for  $e_0 \in L$  do
26:       if  $e_0 \in X$  then
27:         Append  $e_1$  to  $X[e_0]$ 
28:       else
29:          $X[e_0] \leftarrow [e_1]$ 
30:       end if
31:     end for
32:   end for
33: end procedure

```

Rank	1	2	3	4	5	6	7	8
L_1	a	b	d	e	f			
L_2	c	a	b	e	g	h		

item	a	b	c	d	e	f	g	h
s1	8	7	3	6	5	4	3	3
s2	7	6	8	2	5	2	4	3
sum	15	13	11	8	10	6	7	6

Rank	1	2	3	4	5	6	7	8
L'_1	a	b	d	e	f		c, g, h	
L'_2	c	a	b	e	g	h		d, f

Final ranking: a, b, c, e, d, g, f, h

Figure 3.5: An example to combine two ranked lists.

whose labels are indeed wrong (i.e., the recall is likely lower). To maximize the recall, we decide to union the suspicious pairs returned from each detector to form the set of suspicious pairs S .

How do we rank pairs in S ? There are many methods to merge ranked lists [48, 83, 78, 74]. Now we describe how we combine the two ranked lists from FPN and Mono, which can be also extended to combining ranked lists from more detectors.

Our method is based on Borda count method [48]. *The key idea is: for each ranked list, assign points to candidates according to their rankings, then for each candidate sum its points from all ranked lists and rank candidates by their total points.* Following Borda count method, for each ranked list, we first assign points to each candidate in reverse proportion to their ranking, so that higher-ranked pairs receive more points. When all ranked lists have been processed, for each suspicious pair we add up its points, then rank all suspicious pairs in descending order of their total points. However, recall that each ranked list is usually only a subset of S , therefore we need to figure out how to assign points to those missing pairs. To solve this problem, we extend each ranked list into a full list by appending each missing pair to the end of each list. Let L_1, L_2 be the two ranked lists from FPN and Mono, then $S = L_1 \cup L_2$. Let $M_1 = S \setminus L_1$ be the set of missing pairs from L_1 . For simplicity we assign the same number of point to all pairs in M_1 . Once we sum up the points for each suspicious pair, there might be pairs with the same number of total points, which will be returned in random order for simplicity.

Example 3.5.1. Figure 3.5 shows how we combine two ranked lists $L_1 = [a, b, d, e, f]$ and $L_2 = [c, a, b, e, g, h]$ (in which each letter represents a pair). Note that c, g, h are not in L_1 . Appending them to the end of L_1 , we get $L'_1 = [a, b, d, e, f, (c, g, h)]$. Similarly, we get $L'_2 =$

Algorithm 3.10 Combine Ranked Lists

Require: L_1, L_2, \dots, L_n : ranked lists to be combined

Ensure: L : combined list

```

1:  $L \leftarrow \bigcup_{i=1}^n L_i$ 
2:  $m \leftarrow$  length of  $L$ 
3:  $R \leftarrow \{v : 0 \forall v \in L\}$ 
4: for  $i = 1, 2, \dots, n$  do
5:    $m_i \leftarrow$  length of  $L_i$ 
6:   for  $j = 0, 1, \dots, m_i - 1$  do
7:      $R[L_i[j]] \leftarrow R[L_i[j]] + (m - j)$ 
8:   end for
9:   for  $v \in L \setminus L_i$  do
10:     $R[v] \leftarrow R[L_i[j]] + (m - m_i)$ 
11:   end for
12: end for
13: Sort  $L$  in descending order of  $R[v]$  for  $v \in L$ 
14: return  $L$ 

```

$[c, a, b, e, g, h, (d, f)]$. Assume that we assign 8 points to the first item in each list, and 1 point less to the next in each list. Now we compute the sum of points for each item, sort them in descending order of summed points, then we get the combined ranked list $L = [a, b, c, e, d, g, f, h]$. Note that points of f and h are the same, and the ordering between them is random in the combined list.

3.6 Empirical Evaluation

Datasets: We use datasets from a diverse range of domains and different sizes to evaluate our debugging system (Table 3.1). The first nine datasets are publicly available and have been widely used for EM ([65],[19],[72]). The two private datasets, Tools and Clothing, come from a major retailer and are used extensively to match their products with another competitor. The last six datasets come from EM projects in a data science course offered in UW-Madison, which are of small size but cover various domains. Each project dataset was labeled by a team of two or three students.

In Table 3.1, column Size shows the number of labeled pairs and the number of matched pairs in each dataset. The next column provides the number of label errors in those datasets for which

Category	Name	Size	# Errors	# Attrs
Public	Citeseer-DBLP	1,117,574 (558,787)	n/a	6
	Songs	292,022 (146,011)	n/a	7
	Cora	71,466 (35,733)	3,827	9
	DBLP-Scholar	10,694 (5,347)	n/a	4
	DBLP-ACM	4,448 (2,224)	n/a	4
	Amazon-Google	2,600 (1,300)	n/a	4
	Products	2,308 (1,154)	n/a	4
	Abt-Buy	2,194 (1,097)	n/a	4
	Fodors-Zagats	224 (112)	0	5
Private	Tools	247,728 (95,640)	n/a	4
	Clothing	247,628 (105,608)	n/a	4
Teamwork	Movies	600 (190)	1	7
	Beer	450 (68)	2	4
	Bike	450 (130)	8	8
	Citations	400 (92)	0	10
	Restaurants	400 (130)	0	6
	Books	374 (232)	4	9

Table 3.1: Datasets for our experiments.

we have golden labels. The last column lists the number of attributes that are common in the two tables for each dataset.

Datasets in private and teamwork category contain both matched and non-matched pairs, while the nine public datasets originally provide only the set of matched pairs, therefore we need to add non-matched pairs into those datasets before using our debugging system.

Adding Non-matched pairs: Now we describe how we add non-matched pairs to a public dataset that has two tables A and B but only provides a set of matched pairs P between A and B . To do so, we first perform some blocking on the given A and B to get a set of candidate pairs C . C should contain most matched pairs. Let $X = A \times B \setminus (P \cup C)$, then X should only contain non-matched

pairs that are very different from those pairs in P . Now we want to randomly sample a set of pairs N from X . However, $A \times B$ can be very huge, which means implicitly generating X would be very costly. To avoid such an issue, each time we randomly sample an entity a from A and an entity b from B , then check whether $(a, b) \in (P \cup C) \cup N$. If not, then add (a, b) to N . We continue doing so until the size of N is roughly the same as that of P (we want N and P to have similar size to reduce its effect on ranking). Finally, $D = P \cup N$ is the labeled dataset that will be passed to our debugging system to detect potential false matched pairs in P .

3.6.1 Interactive Error Detection

To evaluate the usefulness of our debugging system, an analyst tests our system on the above datasets interactively until it converges (i.e., no label errors are found in three consecutive iterations) or 40 iterations are reached. During the tests, both detectors FPN and Mono are enabled. Each detector returns its ranked list of suspicious pairs, then the combiner combines the top 500 pairs from each list into a single ranked list and return the top 20 suspicious pairs from the combined list to the analyst for manual verification. The number of iterations before it stops and the number of found errors are reported in the 4th and 5th columns in Table 3.2. The last column shows the (estimated) lower bound of total number of label errors in each dataset.

3.6.1.1 Lessons Learned from Interactive Debugging

Importance of Label Debugging: The analyst manages to find label errors in 12 of the 17 above datasets, whose labels are often assumed correct in many research work or industry projects. Also, the number of label errors can be quite large. For example, from the detected label errors, the analyst further estimate that the two private datasets contains at least 3,111 and 8,302 label errors.

Effectiveness of Our System: Within 40 iterations, our system helps the analyst find many label errors on many datasets (for example, 614 label errors are found in Cora dataset). Moreover, on many of those datasets our system converges very fast (in fewer than 13 iterations).

Category	Name	Size	# Iter	# Errors	Lower Bound of Total # Errors
Public	Citeseer-DBLP	1,117,574	13	15	15
	Songs	292,022	10	115	115
	Cora	71,466	40	614	3,827
	DBLP-Scholar	10,694	3	0	0
	DBLP-ACM	4,448	3	0	0
	Amazon-Google	2,600	4	2	4
	Products	2,308	7	6	19
	Abt-Buy	2,194	4	2	2
	Fodors-Zagats	224	3	0	0
Private	Tools	247,728	40	109	3,111
	Clothing	247,628	40	314	8,302
Teamwork	Movies	600	5	1	1
	Beer	450	7	2	2
	Bike	450	8	5	8
	Citations	400	3	0	0
	Restaurants	400	3	0	0
	Books	374	4	2	4

Table 3.2: Statistics of detected label errors.

For the three datasets Cora, Tools and Clothing, the analyst interacts with our system until he/she reaches 40 iterations, and in the last few iterations he/she still finds many label errors, which suggests that there might be more label errors in those datasets. The analyst could continue debugging with our system, or choose to analyze those already found label errors for potential systematic error patterns. Also, on several datasets our system doesn't detect any label errors hence terminates after three iterations. Does it mean that those datasets are clean? For Fodors-Zagats, Citations and Restaurants, we have manually created golden labels confirming that their given labels are indeed clean. For DBLP-ACM and DBLP-Scholar, we don't have golden labels, therefore we randomly sample 100 pairs and manually verify that their given labels are correct,

therefore there is high chance that most labels (if not all) of these two datasets are also correct. Hence our system can help increase the confidence of the analyst in the label quality if he/she doesn't find any errors when our system converges in three iterations.

3.6.1.2 Different Debugging Scenarios

First, as shown in our experiments, the analyst can interact with our system to debug label errors. If the label quality is poor, then our system should help him/her detect many such label errors in first few iterations. Without our system, all he/she can do is to randomly sample a subset for manual verification, which clearly contains much fewer label errors if most labels are still correct.

Next, our system can be used to perform a sanity check on the label quality of a given dataset. As we mentioned earlier, if on a given dataset our system terminates after 3 iterations and no errors are found, it usually indicates that the label quality are really good.

Third, our system can help detect systematic errors. Given a dataset, the analyst first interacts with our system iteratively. If he/she notices that many pairs with label errors follow similar pattern, he/she may suspect that there is one or more systematic error patterns in the dataset. Now he/she carefully examines those pairs for potential error patterns, then writes one or more rules to capture such patterns. After that, he/she can apply the rule(s) to the whole dataset and extract pairs that satisfy those rule(s). If the number of extracted pairs are huge, then he/she can randomly sample a small subset of pairs. Next, he/she manually verifies the labels of those (sampled) pairs, and estimates how many such errors may exist in the pairs that satisfy those rules.

For example, on Tools and Clothing datasets, the analyst follows the above procedure and manages to detect a few systematic error patterns, and estimate that these datasets have at least 3,111 and 8,302 label errors. For example, on Clothing dataset, he/she notices that many pairs, whose product types are very different but their product names and descriptions are very similar, were wrongly labeled as non-matches. Therefore he/she writes a rule to extract 7,686 such pairs, samples 100 pairs for manual verification. He/she finds that 82 out of 100 pairs are actually matches. Therefore he/she estimates that there are about 6,302 such errors in the dataset.

Our debugging system can also help detect inconsistent matching definitions. Nowadays the labels are usually created by a small team of experts or a crowd of workers. It's hard for them to follow the same matching definitions (even if there is only one expert for labeling, it's difficult for him to use the same matching definition when there are thousands of pairs to be labeled).

If the analyst wishes to know whether inconsistent matching definitions are used during label creation, then he/she can use our system for help and follow these steps: first, he/she interacts with our system for a few iterations. If he/she detects label errors, he/she can carefully examine them to see whether these errors are caused by inconsistent matching definitions. If true, then he/she can continue with our system to detect more such inconsistencies. However, if all detected pairs with label errors seem to follow the same matching definition, then he/she can randomly sample a small subset from remaining pairs in the dataset, and examine whether they follow the same matching definition as those pairs with label errors. If the answer is no, then inconsistency is detected. Otherwise, he/she should feel confident that a consistent matching definition is used over the dataset.

3.6.2 Runtime and Scalability

Table 3.3 shows the runtime of our debugging system on those datasets. The second column lists the time of the first iteration (i.e., the time between our system receiving preprocessed features and returning the top k suspicious pairs to the analyst). The next three columns reports the average, minimum and maximum time of our system for interactive debugging (i.e., the time to process the feedback from the analyst for current iteration and return the next top k suspicious pairs to the analyst). The last column shows the number of iterations before the analyst stops debugging.

The time for the first iteration depends on several factors: size of labeled pairs, number of features, and noisiness of the dataset. We can see that our system spends no more than 2 seconds on these small datasets. Citeseer-DBLP contains about 1.1 million labeled pairs and it takes less than 200 seconds for our system to finish the first iteration. Both Tools and Clothing have about 250k labeled pairs, but many of their attributes contain noisy textual values, which takes more

Dataset	First Iter	Interactive Iterations (secs)			# iter
		Avg	Min	Max	
Citeseer-DBLP	198.37	12.317	10.080	17.350	13
Songs	78.44	8.096	2.490	13.180	10
Cora	14.48	3.367	1.880	4.240	40
DBLP-Scholar	1.60	0.070	0.070	0.070	3
DBLP-ACM	0.68	0.030	0.030	0.030	3
Amazon-Google	0.46	0.057	0.020	0.120	4
Products	0.44	0.075	0.020	0.130	7
Abt-Buy	0.35	0.050	0.020	0.110	4
Fodors-Zagats	0.10	0.005	0.000	0.010	3
Tools	227.57	7.824	6.410	9.310	40
Clothing	659.85	11.309	9.100	14.000	40
Movies	0.17	0.017	0.000	0.060	5
Beer	0.14	0.022	0.000	0.060	7
Bike	0.26	0.414	0.010	0.100	8
Citations	0.14	0.000	0.000	0.000	3

Table 3.3: Runtime with our system.

processing time for the first iteration. Nevertheless, our system manages to finish the first iteration within 11 minutes.

For the subsequent interactive debugging iterations, thanks to our incremental updating algorithms, our system has fast response time. For most datasets, on average it takes no more than 4 seconds per iteration. Even for datasets with large size or noisy attributes, our system spends no more than 13 seconds per iteration on average. All these statistics suggest that our system is time effective in various debugging scenarios.

Dataset	Number of Cores (time in secs)				
	1	2	4	6	8
Tools	1468.43	697.74	384.78	280.85	231.85
Clothing	5084.11	2444.25	1228.57	854.66	696.08

Table 3.4: Runtime of the first iteration with multicores.

To study the scalability of our system when utilizing multicores, we use Tools and Clothing datasets. Table 3.4 shows the time of the first iteration when we vary the number of cores from 1 to 8. Clearly, multicores help reduce the runtime of the first iteration. Meanwhile, increasing the number of cores has diminishing returns, because of the overhead of multiprocessing (e.g., in Python, there are process communication and memory copy overhead) and the part of our solutions that cannot be parallelized (Amdahl’s law [81]) For other iterations, we observe that it doesn’t benefit to use more than 4 cores due to the multiprocessing overhead.

3.6.3 Comparison between Detectors

In this section, we study and compare the performance of FPN and Mono. To do so, we use 7 datasets with golden labels, randomly insert some percentage of label errors (the percentage of error is set randomly between 5% to 15%), then run the debugger until there is no label errors in three consecutive iterations. Table 3.5 shows the results from these detectors. We can see that they detect most and even all label errors in only a few iterations on these datasets. FPN and Mono have similar performance on most datasets, while on other datasets one detector is better than the other. For example, on Restaurants Mono finds 40 label errors while FPN only finds 24 errors. Meanwhile, on datasets Beer and Bike, they find similar number of label errors, but each of them actually finds some unique errors, then when both detectors are used, more errors are found.

3.6.4 Sensitivity Analysis

In this section, we perform sensitivity analysis of our system. Specifically, we study the effects of the following parameters in our system: batch size (the number of suspicious pairs returned to the analyst for manual verification in each iteration), and the percentage of label errors.

Dataset	% Errors	# Errors	# Iterations			# Detected Errors		
			FPN	Mono	Combined	FPN	Mono	Combined
Movies	7	43	6	7	6	43	43	43
Beer	7	33	7	6	9	28	29	33
Bike	9	48	11	12	16	30	34	43
Citations	9	36	7	9	8	34	35	36
Restaurants	10	40	5	6	6	24	40	40
Books	11	45	6	9	9	40	42	42
Fodors-Zagatz	8	17	4	5	5	16	13	16

Table 3.5: Performance comparison between detectors.

Dataset	% Errors	# Errors	Batch Size			
			5	10	15	20
Movies	7	43	43	43	43	43
Beer	7	33	33	33	33	33
Bike	9	48	39	43	43	43
Citations	9	36	32	32	36	36
Restaurants	10	40	40	40	40	40
Books	11	45	39	42	42	42
Fodors-Zagatz	8	17	16	16	16	16

Table 3.6: Sensitivity analysis of batch size.

Table 3.6 shows the results when batch size varies from 5 to 20. For each dataset, we insert the same percentage of label errors as in Table 3.5. When the batch size is small, we can see that our system finds fewer label errors sometimes. Recall that we will stop debugging if there is no label error in three consecutive iterations. With smaller batch size, the stopping criteria are more likely triggered, which is why we set 20 to be the default batch size.

Table 3.7 shows the results when the percentage of label errors is increased from 5% to 20% (the batch size is default, i.e., 20). Clearly, our system can detect most label errors even when there are 20% label errors in the dataset. The number of undetected label errors might increase when the percentage of errors continue increasing, due to (1) too many label errors might force FPN to learn a different matching definition, and (2) too many label errors will affect the ranking of Mono and more pairs with correct given labels might be ranked higher, then trigger the stopping criteria (i.e., there is no label errors in three consecutive iterations).

Dataset	5% Errors		10% Errors		15% Errors		20% Errors	
	Total	Found	Total	Found	Total	Found	Total	Found
Movies	31	20	61	61	91	90	121	120
Beer	24	24	47	46	69	66	92	88
Bike	30	26	53	49	75	69	98	90
Citations	20	20	40	40	60	60	80	74
Restaurants	20	20	40	40	60	58	80	80
Books	22	22	41	41	60	60	78	77
Fodors-Zagatz	11	11	22	21	33	32	44	41

Table 3.7: Sensitivity analysis of error percentage.

3.7 Related Work

Recently human-in-the-loop systems and solutions have attracted more and more attentions in both research communities [64, 92, 68] and industries [27, 53]. One of their design goals is to help reduce the workloads of users, which is also the main goal of our label debugging system. However, different such systems usually solve different problems, therefore encounter different challenges and developed different strategies. When designing our system, we also try to improve its flexibility and generality. We believe that such system design is a correct direction toward better human-in-the-loop systems.

Label errors and its effects on supervised learning (especially classification) have been widely studied [51]. Recently machine learning communities also contribute enormous efforts to design new learning models that can tolerate errors in training data [91, 69, 77, 88]. However, the tolerance of label errors in these models are usually very limited. Therefore, we believe that designing solutions to detect and fix label errors is still the most promising approach to improve the quality of such data analysis tasks.

As far as we can tell, our solution is the first research work to detect label errors in labeled entity matching datasets. [11] performs similar to our FPN detector. However, instead of detecting and fixing label errors, they eliminate examples whose labels are likely wrong and train model on other examples.

Combining rankings has been widely studies and different methods are proposed, such as Condorcet method [78], Schulze method [83], Voting System [74] and Borda count [48] (which is used

in our system). However, Arrow's impossibility theorem [9] indicates that there is no best combining method in general. We choose Borda count due to its simplicity and $O(n)$ time complexity.

Researchers also work on a different debugging scenario for which labels are assumed clean and they want to do minimal repairs on other attributes (or features) of training data so that the bias in trained model is reduced [12].

3.8 Conclusions

In this chapter, we addressed the problem of debugging label errors in labeled entity matching datasets. As far as we can tell, this is the first research work for this problem. We designed a system that iteratively interact with the analyst to detect and fix label errors. We implemented two detectors to detect label errors: learning based FPN and domain-knowledge based Mono. We conduct extensive experiments with real datasets and demonstrate the effectiveness of our system: it helps significantly reduce the workload of the analyst and can be used towards various debugging scenarios. Our system is extendable can also integrated other types of detectors, such as rule-based detectors. We also implemented various techniques such as incremental updates and utilization of multicores to reduce the latency between iterations. Our experiments show that it can handle datasets of thousands labeled pairs of entities and the latency between iterations is usually within seconds. In the future, we plan to extend our system to support other label debugging tasks, and more generally, data cleaning tasks.

Chapter 4

Large-Scale Active Learning for Entity Matching

This chapter studies the problem of performing large-scale active learning for entity matching. I first develop a distributed solution on a cluster of machines, evaluate it using several datasets and study its limitations, then explore various opportunities to develop better solutions.

The chapter is organized as follows. The next section introduces some background and defines the problem that we plan to address in this project. Section 4.2 describes our first solution in detail and our empirical evaluation results. Section 4.3 describes how I improve our first solution to develop our second solution, and perform experiments to compare these two solutions. Section 4.4 describes how I continue improving our second solution and develop our third solution, followed by experiments to demonstrate its promise. After that, I briefly describe the related work, then conclude the chapter with discussions and future work.

4.1 Background and Problem Definition

For many data management tasks, we need to label data. It will take huge amount of time and money to manually label a large set of data instances. To help reduce labeling cost and time, we usually first uniformly-random sample a small set of data instances from the dataset, label these sampled instances, then use the labeled instances to train a machine learning model, and finally use the model to automatically label the remaining instances in the dataset. However, to train a model with good accuracy, the sample size cannot be too small.

To further reduce the number of instances to be labeled, researchers proposed active learning. The key idea behind active learning is that a machine learning algorithm can perform better with

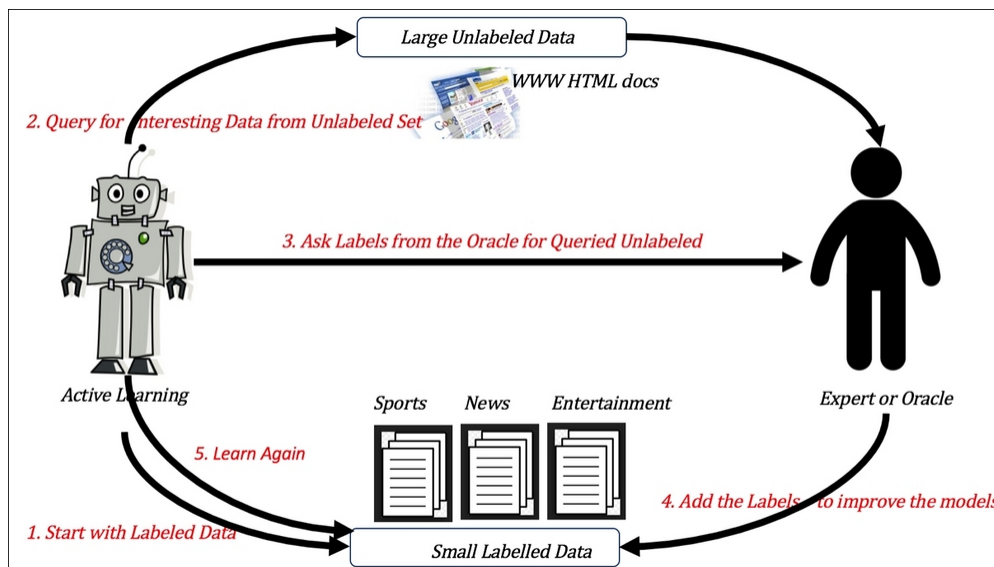


Figure 4.1: An example of active learning workflow.

fewer training examples if it is allowed to choose the data from which it learns [84]. To do so, active learning builds an initial model with a small set of labeled examples. Then it performs iterative interaction with one or more labeling workers (Figure 4.1).

In each iteration, it uses the current model to select the most controversial unlabeled examples, then sends them for manual labeling. Once their labels are returned, active learning adds them to the training data to build a new model. Active learning continues until some pre-specified stopping criteria are met (e.g., the maximum number of iterations has been reached, or the model has converged according to some measurement such as maximum entropy of label prediction confidence among all unlabeled examples).

4.1.1 Large-Scale Active Learning for Entity Matching

In this chapter, we focus on active learning on large-scale entity matching datasets. In entity matching, we are usually given two tables A and B , and a set of unlabeled pairs U (which is often the result of performing some blocking step between A and B), then we wish to label each pair as match or non-match. We assume that there is a target model M to be trained (i.e., some classifier such as Random Forests). Unlike most active learning scenarios (in which feature vectors

of unlabeled examples are given), in entity matching we are often given only a set of similarity functions F to compute features for M .

To perform active learning, we also need some labeled pairs (seeding pairs) S to train the initial model. For our study, we will use Random Forests as the model M and the set of features automatically generated by Magellan [64] between A and B as the set of similarity functions F . Now we first introduce a basic common solution, discuss its limitations and our project goals. After that, we will detail how we develop our solutions step by step.

A Basic Solution: As mentioned earlier, in our scenario, we are given a set of similarity functions F instead of the set of feature vectors for unlabeled pairs. To perform active learning, we first need to compute the feature vectors for pairs in S and train the initial matching model M . To apply the model on unlabeled pairs in U to select the most controversial pairs for labeling, we also need to compute their feature vectors (for simplicity, each iteration active learning uses entropy [13] to select pairs for manual labeling).

Now we can perform the common active learning steps iteratively until some stopping criteria are met: in each iteration, (1) we apply M to feature vectors of unlabeled pairs in U and for each pair in U we compute the entropy score of its label prediction confidence; (2) we sort pairs in U in descending order of their entropy scores, then return the top k pairs (denoted as K) for manual labeling; (3) if the given stopping criteria are met, we stop active learning; otherwise (4) we remove K from U , add K (and their labels) to S and train a new model M , then repeat from step (1).

The above solution is simple and works well on a single machine if the size of U is not large. In fact, when U is small, it only takes seconds or a few minutes to compute all feature vectors, and in each iteration the time to train models and select the most controversial pairs for manual labeling is often less than one second.

However, in each iteration manually labeling pairs may take significantly more time. For example, in our experience, it often takes 5 or more seconds to label one entity pair. Then if in each iteration active learning returns 20 pairs for manual labeling, it will take at least 100 seconds before continuing next iteration. Clearly when U is small, the (time) bottleneck of each iteration is the

time to wait for manual labeling of pairs in K . However, when given a large-scale dataset U , the labeling experience is very different, and the above solution has significant limitations.

Limitations: First, when U has large size, with a single commodity machine, it may take hours to compute feature vectors of pairs in U . It is because entity matching tasks usually use dozens and even hundreds of similarity features to train the matcher (i.e., the model M), and most features are evaluated on pairs of strings, which are often expensive to compute. It means that the user has to wait a very long time before he/she can start labeling the batch of pairs from the first iteration.

Moreover, when given millions of pairs and hundreds of features, the size of all feature vectors of pairs often exceeds the memory size of a single commodity computer. If we save the feature vectors to disk then load and scan through it every iteration, it increases the time between iterations (which also means that users face longer waiting time between iterations). Since active learning often continues tens or even hundreds of iterations before termination, it also increases the total time significantly.

In summary, the above solution has scalability issues when U is large and active learning takes too much time to complete, and users need to wait long time before starting labeling and between iterations.

Project Goals: In this project, we want to address the above limitations. Specifically, we want to build solutions to improve the labeling experience of users when performing large-scale active learning for entity matching. Assuming that users want to label at most n entity pairs (e.g., $n = 600$), our goals are:

1. Reducing the waiting time before users can start labeling pairs selected from the first model;
and
2. Reducing the user waiting time between iterations (which includes time to train a new model and time to select the next batch of pairs for manual labeling, etc.).

In the rest of this chapter, we will detail how we develop our solutions step by step to achieve our goals.

4.2 Our First Solution

As mentioned earlier, when U contains many unlabeled entity pairs, we face the scalability issues: it takes significant amount of time to compute the feature vectors of those pairs, and the feature vectors may not fit into the memory of a single commodity machine. To address the scalability issues, we decide to develop our first solution that can utilize a cluster of commodity machines, since it's easy and cheap to set up a cluster of commodity computers nowadays.

4.2.1 A Cluster Solution Using Spark

We choose Spark to develop our first solution on a cluster of machines. Why? Spark is “an open-source distributed general-purpose cluster-computing framework”, “a unified analytics engine for large-scale data processing” [2]. Due to caching in memory and other optimizations, it can be up to 100x faster than Hadoop. It is easy to use and provides high level APIs in Scala, Java, Python, R and SQL shells. Besides its own cluster resource management mode (standalone), it can also run with many other cluster management systems, such as YARN, Apache Mesos, Kubernetes, which also makes it easy to deploy in the cloud (such as AWS). Therefore we choose Spark (specifically, PySpark) to scale up active learning for large-scale entity matching datasets. For our experiments, our Spark cluster is configured in standalone mode, and use the default task scheduling and failure recovery from Spark.

How should we perform active learning on a cluster of machines? Given one master machine and m worker machines, we can partition U into m partitions U_1, U_2, \dots, U_m , then assign U_i to the i -th worker machine. Each worker machine will need to compute and store feature vectors of pairs in its local partition, predict their labels with M , therefore each worker machine needs a copy of tables A and B , and the set of feature functions F . To perform prediction, in each iteration we also need to broadcast the trained model M to each worker machine.

How should we find the top k controversial pairs in each iteration? We could ask each worker machine to send back the computed entropy scores of pairs in its local partition, then sort all of them and return the top k pairs. However, it could cause a lot of communication overheads. A

Algorithm 4.1 Our First Spark Solution

Require: A, B : the two tables of entities, U : set of unlabeled pairs, S : set of labeled (seeding) pairs, F : set of feature functions, M : the model to be trained, C : stopping criteria, m : the number of worker machines

Ensure: P_U : labels of pairs in U , M : the final trained model

```

1: Broadcast  $A, B, F$  to each machine
2: Partition  $U$  into  $U_1, U_2, \dots, U_m$  and send  $U_i$  to machine  $i$ 
3: Worker machine  $i$  computes and caches feature vectors for pairs in  $U_i$ 
4: Apply  $F$  to  $S$  to compute feature vectors  $X_S$ 
5:  $X_M \leftarrow X_S$ 
6:  $P_U \leftarrow \emptyset$ 
7: while true do
8:   Train  $M$  on  $X_M$ 
9:   Broadcast  $M$  to each worker machine
10:  Each worker machine applies  $M$  and finds its local top- $k$  pairs, then sends them to the master machine
11:  Let  $K_i$  be the local pairs from machine  $i$ 
12:  Find top- $k$  pairs from  $\bigcup_i^m K_i$  and denote them as  $K$ 
13:  if  $C$  is satisfied then
14:    break
15:  else
16:    Broadcast  $K$  to each machine
17:    Send  $K$  for manual labeling and let  $Y$  be the labels of pairs in  $K$ 
18:     $X_K \leftarrow$  compute features of pairs in  $K$ 
19:     $X_M \leftarrow X_M \cup X_K$ 
20:     $P_U \leftarrow P_U \cup Y$ 
21:  end if
22: end while
23: return  $P_U, M$ 

```

better way is to find the top k distributively. The key observation is that if for a worker machine, a pair is NOT among the top k controversial pairs in its assigned partition, then that pair cannot be one of the global top k controversial pairs. That is, the global top k pairs can only exist among the top k pairs of each local partition. Therefore each worker machine finds its local top k pairs and sends them back to the master machine, then the master machine only needs to sort those $m \times k$ pairs and find the global top k pairs.

A new problem in this distributed solution is that now each machine needs to know what pairs have been labeled before that iteration so it can exclude them when selecting the local top k pairs.

Dataset	# Attributes	# Candidate Pairs	# Matched Pairs
Amazon-Google	3	11.5 K	1,300
Walmart-Amazon	5	10.2 K	1,154
DBLP-Google	4	28.7 K	5,347
DBLP-ACM	4	12.4 K	2,224
Clothing	26	247.6 K	105.6 K
Tools	27	249.3 K	96.8 K

Table 4.1: Datasets for our preliminary experiments.

To solve it, in each iteration the master machine will broadcast the global top k pairs to all worker machines. The above distributed active learning procedure is described in Algorithm 4.1.

4.2.2 Empirical Evaluation

Cluster Setup: All our experiments are conducted on a Spark cluster consisting of 10 computers. Each computer has 4 cores and 16GB memory. In our experiments, one computer is used as the master machine, while each of the other 9 computers is configured to run three executors (each executor is a virtual worker machine assigned with one core and 4GB memory), which means in total the cluster contains 27 executors ¹. Therefore each dataset is automatically partitioned into 27 partitions and each executor processes its tasks on its assigned partition.

Datasets: To evaluate our first solution, we use 6 entity matching datasets (Table 4.1). The table shows some important statistics of those datasets, including the number of attributes, the total number of pairs and the number of matched pairs. All pairs in each dataset are labeled as either matched or non-matched pairs, so that we can use them to simulate active learning (for the first 4 datasets, their candidate sets U 's are from DeepMatcher [72]). The first 4 datasets are relatively small: each has only three to five attributes and contains fewer than 30k candidate pairs, and less than 20% of their candidate pairs are matches. The other two datasets, Clothing and Tools, have 26 and 27 attributes and each contains about 250k candidate pairs (of which about 40% pairs are matches).

¹The remaining core and 4GB memory are reserved for other services in the cluster.

Dataset	Time (sec)		Avg Time Per Iteration (sec)			Waiting Time Before First Labeling (sec)	Avg Waiting Time Between Iterations (sec)
	Total	Feature Computation	Sum	Train	Top-k		
Amazon-Google	18.68	7.46	0.38	0.18	0.20	7.56	0.38
Walmart-Amazon	21.70	8.08	0.44	0.25	0.19	8.83	0.44
DBLP-Scholar	24.44	13.62	0.36	0.15	0.21	14.13	0.36
DBLP-ACM	17.08	7.02	0.34	0.14	0.20	7.08	0.34
Clothing	470.87	418.45	1.23	0.95	0.28	435.31	1.23
Tools	407.87	356.73	1.08	0.81	0.27	376.58	1.08
Clothing (x3)	1070.21	1013.25	1.35	0.99	0.36	1030.98	1.35
Tools (x3)	1128.07	1071.32	1.27	0.82	0.45	1091.23	1.27
Clothing (x5)	2141.27	2078.88	1.56	0.97	0.59	2095.90	1.56
Tools (x5)	1807.20	1748.06	1.36	0.81	0.55	1767.65	1.36
Clothing (x10)	4206.38	4135.61	1.86	1.01	0.75	4152.54	1.86
Tools (x10)	3546.41	3476.10	1.75	0.87	0.88	3495.70	1.75

Table 4.2: Runtime on these datasets.

Experiment Settings: For each active learning experiment, we start with 4 seeding pairs (2 matched pairs and 2 non-matched pairs), In each iteration we select the top 20 most controversial pairs from the set of unlabeled pairs, and manual labeling is simulated by automatically retrieving their golden labels. The training model is the Random Forests classifier from scikit-learn (for which the number of trees is set to 10). The set of feature functions are automatically generated from common attributes between table A and table B using Magellan [64]. Active learning continues until in total 600 pairs are selected and manually labeled (in other words, active learning is terminated after 30 iterations in our first Spark solution).

Cluster Runtime: Table 4.2 shows the runtime results on these datasets with our first solution. To better evaluate the scalability of our first solution, we replicate the set of candidate pairs for Clothing and Tools to create 6 additional datasets, which are shown in the last 6 rows. For example, Clothing (x10) means that the set of candidate pairs are replicated 10 times, therefore it contains about 2.5 million candidate pairs.

The second and third column of the table list the total machine runtime and the time for feature computation. For the first 4 datasets, their machine runtime is less than 25 seconds and it also

takes no more than 14 seconds for feature computation, because they only have dozens of features to compute and relatively small number of candidate pairs. However, for Clothing and Tools, the total machine runtime increases to more than 400 seconds, and feature computation takes between 70% and 80% of their total machine time. For those replicated datasets, we can see that both the total machine runtime and time for feature computation increases (approximately) linearly to the number of replications, which is as expected.

The next three columns summarize the average time per iteration (after the pairs from the first iteration are labeled), split into the time to train models and the time to find the top 20 pairs. Clearly, the time to train models is small because each iteration only 20 pairs are labeled and the training set is small (when active learning terminates, there are only 604 labeled pairs). Also, thanks to our distributed top-k algorithm, the time to find the top 20 pairs is small. However, if the size of candidate pairs continue increasing, the average time per iteration will continue increasing, which means that users may need to wait longer between iterations.

The last two columns contain the waiting time before users can start labeling and the average waiting time between iterations. The waiting time before users can start labeling includes both the time to load and distribute datasets to worker machines and the time for feature computation. The average waiting time between iterations is (almost) the same as the average machine runtime between iterations.

We observe that most of the machine runtime is spent for feature computation, which also means that users still have to wait very long time before they can start labeling the batch of pairs from the first iteration. To further speed up feature computation, we may add more computers to the cluster if we have enough budget. Is it possible to speed up feature computation without adding more computers and reduce the user waiting time before they can start labeling? To answer this question, we go deeper and study how features are computed in entity matching and how models change through active learning, then develop our second solution, which will be described in the next section.

4.3 Our Second Solution

As mentioned earlier, to speed up feature computation without adding more computers and reduce the user waiting time before labeling, we first study how features are computed in entity matching and how models change through active learning to search for opportunities that we can explore.

4.3.1 Feature Computation in Entity Matching

Recall that we are given two tables A and B , and a set of features F . To compute the features for a pair of tuples $\langle a, b \rangle$ between A and B , we first retrieve a from table A and b from table B . Now to apply a feature function f , we need to get values of the attribute α required by f (these values are denoted as $\alpha(a)$ and $\alpha(b)$), then compute $f(\alpha(a), \alpha(b))$. Sometimes the feature function also needs to first tokenize the values of the attribute for a and b , then compute the similarity score between the two sets of tokens. Suppose the tokenizer is denoted as t , then what we compute will be $f(t(\alpha(a)), t(\alpha(b)))$. A concrete example is shown here.

Example 4.3.1. *Suppose we want to compare the titles of two books: “Database Systems: Design, Implementation, & Management” and “Database Systems: A Practical Approach to Design, Implementation, and Management”. We might agree that if the two titles have a lot of common words, they are likely the same book. One such similarity measure is Jaccard [5]. After removing punctuation marks, we tokenize both titles into words, therefore we have two sets of words: $S_1 = \{\text{Database, Systems, Design, Implementation, Management}\}$ and $S_2 = \{\text{Database, Systems, A, Practical, Approach, to, Design, Implementation, and, Management}\}$. S_1 contains five words and S_2 contains 10 words, and $S_1 \cap S_2$ contains five words. Therefore the Jaccard score between S_1 and S_2 is $J(S_1, S_2) = \frac{|S_1 \cap S_2|}{|S_1| + |S_2| - |S_1 \cap S_2|} = \frac{5}{5 + 10 - 5} = 0.5$. This score is not high, so we cannot claim that they are the same book (In fact, they are two different books.)*

How should we speed up the computation without adding more computing powers? We need to figure out whether there are any operations that occur more than one time. If so, we may only need to compute that operation once, cache its result in memory, then reuse it when we encounter

the same operation again. In total we develop three caching techniques. Now we first describe what motivates our development, then provide details of how these caching techniques are used.

In entity matching, it is common that several features require the same attribute, especially if the attribute is of type string. For example, Magellan usually provides at least five features on each attribute of type string, such as Jaccard, Cosine, Jaro distance, Levenshtein distance, Jaro–Winkler Similarity, etc. It motivates us to develop our first caching technique - Caching Values for Attributes (which will be described later). Meanwhile, tokenization is an expensive operation, especially on strings containing many characters. But we often use the same tokenizer before applying Jaccard and Cosine similarity functions, therefore we develop our second caching technique - Caching Tokens for Attributes. To introduce our third caching technique, we first describe another common scenario in entity matching next.

Frequency Analysis of Categorical Attributes: In entity matching datasets, categorical attributes are quite common, such as gender, country, state, city, department, etc. Each such attribute only contains limited number of values, which means that the total number of unique pairs of their attribute values are also limited. Given a large dataset of unlabeled pairs, some unique pair of values might occur in thousands of unlabeled pairs. Therefore we perform frequency analysis on a few categorical attributes. We ranked pairs of attribute values in descending order of their frequencies, then plot the corresponding frequency-rank curve shown in Figure 4.2, where the x-axis is the ranking and the y-axis shows the corresponding frequency. In the figures for the datasets Clothing and Tools, we have log-scaled the frequency and cut off the ranking at 200. From those curves, we can see that the frequency curves are similar to Zipfian distribution, i.e., the top frequent pairs of attribute values occur much more than other pairs. Based on this observation, we develop our third caching technique - Caching Values of Features for Categorical Attributes. Next we will provide details of our three cache caching techniques.

4.3.2 Speed up Feature Computation with Caching Techniques

As mentioned earlier, in total we develop three caching techniques to speed up feature computation. Now we describe each in detail.

Caching Values for Attributes: When we compute a feature f on a pair of tuples $\langle a, b \rangle$ and f is operated on attribute α , then we need to first get the values $\alpha(a)$ and $\alpha(b)$, then compute $f(\alpha(a), \alpha(b))$. It turns out that retrieving the value of an attribute from a tuple is expensive. Meantime, in entity matching, it is common that various features are computed on the same attribute. For example, if an attribute is of type string (such as book title in previous example), there are features such as Jaro-Winkler distance, Levenshtein distance, Jaccard and Cosine (these two features need to tokenize the pairs of strings first), etc. It brings us the opportunities to caching attribute values and reusing them. That is, for a pair of tuples, when we first get values from an attribute, we cache them. Later when another feature on the same attribute is encountered, we get the values directly from the cache. When can we free the corresponding cache for an attribute? Clearly, if all features requiring that attribute have been computed, the cache is no longer needed and hence can be freed safely. Alternatively, each machine can cache the values of all attributes for pairs assigned to it in advance before computing any features, then it no longer needs to store tables A and B in the memory therefore discards the two tables. This alternative approach is preferred if tables A and B are huge and pairs in each partition only need small subsets of tuples from tables A and B .

Caching Tokens for Attributes: As mentioned earlier, many features on strings (such as Jaccard and Cosine) require to first tokenize the pair of strings into two sets of tokens, then evaluate between the sets of tokens. Tokenizing strings is very expensive, especially when the string length is large. If more than one feature needs to tokenize the same attribute using the same tokenizing method (e.g., 3-gram), then it is worth caching the sets of tokens and reusing them later. Similarly, we can monitor whether all features on the same attributes and using the same tokenizer have been computed. If so, we can safely free those cached tokens.

One problem of caching tokens is that tokens may take huge amount of memory in Python, since tokens are essentially stored as strings. Also, set operations such as intersection, union are much more expensive on set of strings, compared with set of integers. We also observe that strings often share many common tokens, which means the total number of unique tokens will be much smaller than the sum of size of all sets of tokens. To reduce memory usage, we map each token into an integer, then each set of tokens is converted into set of integers, and now we only need to cache

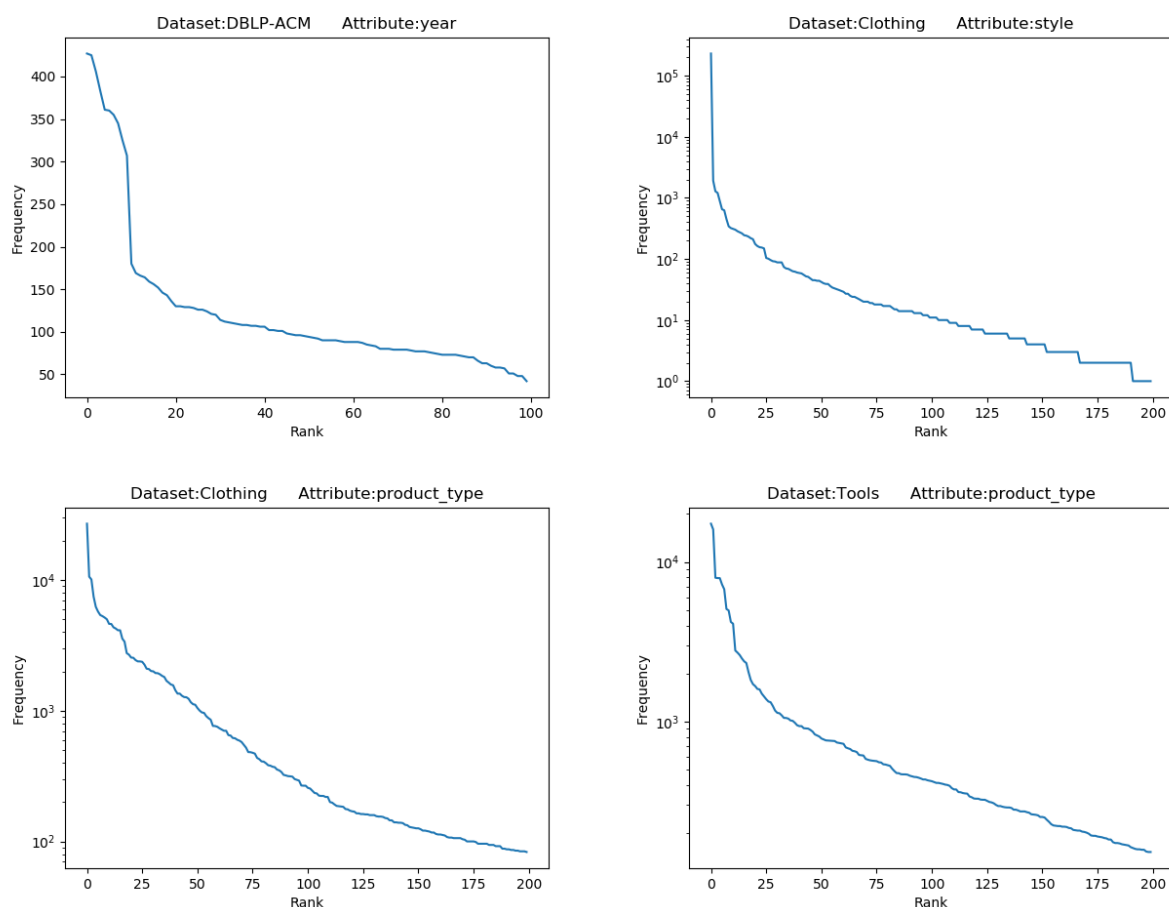


Figure 4.2: Examples of categorical attributes and their frequency distributions.

the set of integers. Though the mapping causes some runtime overhead, it is often compensated by the time savings on those set operations during feature computation.

Caching Values of Features for Categorical Attributes: As mentioned earlier, categorical attributes (i.e., attributes that have only a small set of values, such as cities, gender, states, etc) are common. If the attribute has only k values, then the number of unique pairs of this attribute values is at most k^2 . Given a large set of entity pairs, thousands or more of them will contain the same pair of values for the attribute. As shown in Figure 4.2, they often follows Zipfian distribution. Meanwhile, many distance similarity functions operated on a pair of strings are really expensive to compute. Therefore it is worth caching and reusing such feature scores. When a pair of values

Dataset	# Total Features	# Used Features
Amazon-Google	19	19
Walmart-Amazon	30	30
DBLP-Scholar	17	17
DBLP-ACM	14	14
Clothing	201	79
Tools	170	97

Table 4.3: Feature statistics.

for the attribute is encountered, we first probe the cache. If the feature score between the pair has been computed, we don't need to compute it again. If it is not in the cache, we compute and cache the score. Note that this caching technique only benefits on categorical attributes and for features that are expensive to compute (if a feature is computationally light, such as exact match, caching the scores is not that helpful).

We have described our study of feature computation in entity matching and how it motivates us to develop the above caching techniques to speed up feature computation. Next we will describe our study of those trained models over iterations during active learning, which motivates us to apply another important idea in our second solution.

4.3.3 Number of Features over Iterations

Table 4.3 shows some feature statistics of these datasets, including the total number of computed features, and the number of features used in at least one model during the active learning process. In general, if a dataset has many attributes, then Magellan usually transforms them into more features.

We observe that on the first 4 datasets, all features are used in at least one model during active learning. One plausible explanation is that the creators of those datasets discarded attributes that are not involved in their matching criteria after they created the golden labels (which may also explain why they contain so few attributes).

As mentioned earlier, the last two datasets, Clothing and Tools, have many attributes, which are transformed into 201 and 170 features by Magellan. But we can see that only 79 and 97 features are actually used during active learning. This situation is quite common in entity matching tasks: users try to collect as many attributes as possible since they have little clue which attributes might be useful for their tasks, while many of these attributes turn out not useful when learning the matcher.

We continue to study those trained models over iterations and focus on the number of features used in those models. Recall that we use Random Forests as the active learning models. To get the set of features used in a Random Forest, we first traverse each tree in the model for its list of features in the tree. Then we union the lists of features from all trees in the Random Forest to get the set of features in the Random Forest. Besides the number of features in the trained models over iterations, we also record down the number of new features (that is, features not present among models in previous iterations), and the cumulated number of features (total number of unique features presented in at least one previous and current iterations). These statistics are plotted as three curves shown in Figure 4.3.

Each subfigure has three curves. The blue curve shows the cumulated number of features over iterations, which is clearly monotonically increasing. The orange curve shows the number of features in Random Forests over iterations. As we can see, the first Random Forest often uses only a few features since it is trained with only four initial labeled pairs². When more and more labeled pairs are available, Random Forests in later iterations use more and more features. But the number of features used in Random Forests become relatively stable after enough number of iterations. It is because with enough training data, features that are useful to learn the matcher will be detected and used in Random Forests. The green curve shows the number of new features over iterations. Clearly, most new features start to appear in the first few iterations. For later iterations, there are very few new features and for many iterations there are even no new features used by those Random Forests.

The above is an important observation. Recall that in our first solution we compute all features before active learning starts, which takes too much time and users have to wait before they can start

²In general, increasing the number of trees in the forest likely increases the number of used features.

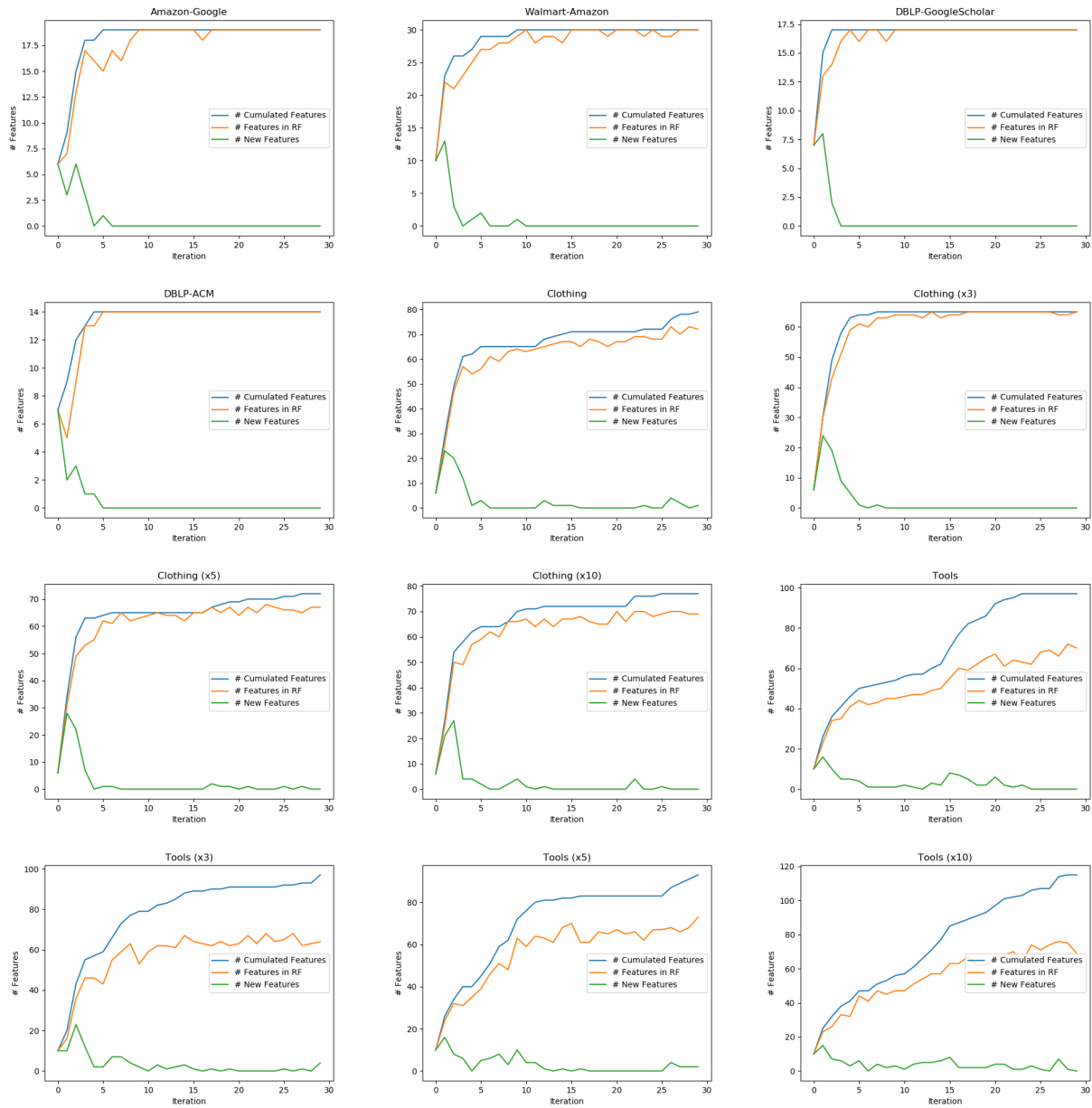


Figure 4.3: Number of features over iterations.

labeling. However, since the first Random Forest often uses only a few features, we don't need to compute all features. Instead, we only need to compute those features required by the first Random Forest, then we are able to apply the first Random Forest to select pairs for manual labeling. It will significantly reduce the waiting time before users can start labeling the batch of pairs from the first iteration. We extend this idea to all iterations and call it lazy feature computation, i.e., a feature is only computed for unlabeled candidate pairs when it is needed the first time by the Random Forest in some iteration. We will provide more details of lazy feature computation in the next subsection.

4.3.4 Lazy Feature Computation

As mentioned above, with our observation from curves in Figure 4.3, we find the opportunity of lazy feature computation. Besides the benefit for users to start labeling much faster, it might also reduce the total machine time. Recall that in Table 4.3, the two datasets, Clothing and Tools, have 201 and 170 features, but only 79 and 97 features are used in trained Random Forests during active learning. That is, the other 122 features for Clothing and 73 features for Tools are NOT useful, but in our first solution we still compute those unused features. With lazy feature computation, those unused features won't be computed at all, therefore lazy feature computation may also reduce the total machine time.

Some may ask why not try to identify and discard features that are not helpful in advance? If we were able to identify them, we would also avoid computing them. However, in general it is really difficult to identify such features. Most feature selection algorithms (if not all of them) require labeled data to perform feature selection (i.e., selecting useful features or identifying useless features). If we have a large enough set of labeled pairs available before hand, we could perform feature selection first before active learning. But as mentioned earlier, in active learning, we often only start with a few labeled pairs (in our settings, only four such labeled pairs). Therefore feature selection won't work.

To perform lazy feature computation, we need the list of features used in the model. Since we use Random Forest as the active learning model, to get the list of used features, we traverse each tree to find the set of features used in the tree, then union the sets of features from all trees in the

Algorithm 4.2 Lazy Feature Computation

Require: A, B : the two tables of entities, U : set of unlabeled pairs, S : set of labeled (seeding) pairs, F : set of feature functions, M : the model to be trained, C : stopping criteria, m : the number of worker machines

Ensure: P_U : labels of pairs in U , M : the final trained model

```

1: Broadcast  $A, B, F$  to each machine
2: Partition  $U$  into  $U_1, U_2, \dots, U_m$  and send  $U_i$  to machine  $i$ 
3:  $X_M \leftarrow X_S$ 
4:  $P_U \leftarrow \emptyset$ 
5: while true do
6:   Train  $M$  on  $X_M$ 
7:   Broadcast  $M$  to each worker machine
8:   Each worker machine computes any missing features required by  $M$ , applies  $M$  and finds its local top- $k$  pairs, then sends them to the master machine
9:   Let  $K_i$  be the local pairs from machine  $i$ 
10:  Find top- $k$  pairs from  $\bigcup_i^m K_i$  and denote them as  $K$ 
11:  if  $C$  is satisfied then
12:    break
13:  else
14:    Broadcast  $K$  to each machine
15:    Send  $K$  for manual labeling and let  $Y$  be the labels of pairs in  $K$ 
16:     $X_K \leftarrow$  compute features of pairs in  $K$ 
17:     $X_M \leftarrow X_M \cup X_K$ 
18:     $P_U \leftarrow P_U \cup Y$ 
19:  end if
20: end while
21: return  $P_U, M$ 

```

trained Random Forest to get the list of all used features. Once we get the list of features required by the current model, we check if any feature is not computed yet, then compute those missing features for all unlabeled pairs. Now we can use the current model to select the most controversial pairs for manually labeling before training the model for next iteration. Algorithm 4.2 shows the revised procedure with lazy feature computation.

One additional challenge due to lazy feature computation is about caching. Recall that one of the cache techniques we develop is to cache values for attributes. That is, when computing features for a pair of tuples, if more than one feature requires the same attribute, we will cache the values for that attribute when first such feature is computed and reuse those values for other features that requires the same attribute. Therefore the cache is usually freed when all features are computed for

the pair of tuples. However, since now we perform lazy feature computation, these features might be computed over several different iterations (worse case is that some feature is still not computed when active learning terminates), which means more memory are used for caching during active learning iterations.

To address this issue, we can cache only a subset of attributes that are likely reused during active learning. How to select such attributes? One method is to select attributes based on feature importance in current model (if an attribute is important in the current model, all other features requiring the same attribute might be used in models from future iterations, hence that attribute is worth caching). This is one of our future work.

4.3.5 Empirical Evaluation

Now we evaluate our second solution using the same datasets and cluster settings. The results are shown in Table 4.4. Recall that for our second solution, we develop and use two important ideas: caching techniques and lazy feature computation. Caching techniques are mainly used to speed up feature computation and Lazy Feature Computation is to reduce the waiting time before users start labeling. Therefore we record down and compare the total machine runtime and time before users start labeling between our first and second solution. Columns with header ‘V1’ are for our first solution, Columns with header ‘LazyFC’, ‘Cache’, and ‘V2’ are when only lazy feature computation is used, only caching techniques are used, and when both ideas are used.

For the total machine runtime, we can see that for the first 4 datasets, all four columns under ‘Total Time’ have similar numbers. Those with lazy feature computation takes slight longer time. Recall that for these 4 datasets, all features are used by the end of active learning, therefore lazy feature computation doesn’t reduce the total machine runtime. Instead, it has slight longer total runtime since the implementation of lazy feature computation creates some small time overhead per iteration.

For the other rows, lazy feature computation indeed helps reduce their total runtime, since those unused features are not computed at all in lazy feature computation. Comparing the columns between V1 and Cache, we can clearly see that caching techniques help significantly reduce the

Dataset	Total Time (sec)				Time Before User Starts Labeling (sec)				V2 Avg Waiting Time Between Iterations (sec)
	V1	LazyFC	Cache	V2	V1	LazyFC	Cache	V2	
Amazon-Google	18.68	20.58	17.19	18.12	7.46	6.04	6.00	5.59	0.43
Walmart-Amazon	21.70	22.58	19.38	19.76	8.08	6.36	5.55	5.50	0.47
DBLP-Scholar	24.44	24.79	17.93	18.08	13.62	7.85	7.00	6.09	0.40
DBLP-ACM	17.08	18.68	16.07	16.67	7.02	6.34	6.08	5.79	0.37
Clothing	470.87	296.26	98.85	96.77	418.45	25.60	46.18	16.83	2.16
Tools	407.87	364.14	92.12	97.82	356.73	39.71	41.28	22.44	1.93
Clothing (x3)	1070.21	601.38	170.65	174.12	1013.25	63.40	112.61	38.03	4.09
Tools (x3)	1128.07	892.44	161.97	170.17	1071.32	95.73	107.29	46.04	3.62
Clothing (x5)	2141.27	1095.41	263.95	255.25	2078.88	97.59	204.43	58.93	6.18
Tools (x5)	1807.20	1380.67	232.40	243.66	1748.06	151.97	172.94	70.23	5.29
Clothing (x10)	4206.38	2244.10	457.64	443.89	4135.61	199.16	386.06	109.30	10.95
Tools (x10)	3546.41	3287.39	394.67	445.41	3476.10	246.83	327.74	129.31	10.22

Table 4.4: Evaluation of caching and lazy feature computation.

total machine runtime. Similarly, V2 (Cache+LazyFC) has much lower numbers than those in V1 and LazyFC, but might be higher than those in Cache due to the overhead of lazy feature computation. However, when the dataset size continues growing, if there are many unused features, the benefits of lazy feature computation to avoid computing such features will eventually beat the overhead caused by the implementation of lazy feature computation, such as Clothing (x5) and Clothing (x10).

When comparing the waiting time before users start labeling, we can clearly see that lazy feature computation helps significantly reduce the waiting time on those large datasets. Even on the first 4 small datasets, lazy feature computation has smaller waiting time.

Clearly both caching techniques and lazy feature computation help achieve our goals. But according to “No free lunch theorem”, we must have sacrificed something for them. For caching techniques, it’s easy to understand that we actually need more memory to cache those attribute values, tokens, and feature scores. For lazy feature computation, we only compute a feature when it is needed the first time in some Random Forest. It means that in some later iteration, if a feature is required but not computed yet, then we still need to compute it. Therefore for that iteration users need to wait longer due to lazy feature computation, which explains why in the last column of Table 4.4, V2 has longer average waiting time between iterations than those numbers in Table 4.2.

Note that even if we don't use lazy feature computation, users cannot continue labeling until they get the batch of pairs from the next iteration. Can we handle this new problem caused by lazy feature computation? More generally, can we reduce or even eliminate the user waiting time between iterations? To address this goal, we develop our third solution, which will be described next.

4.4 Our Third Solution

In our second solution, we use lazy feature computation to significantly reduce the waiting time before users start labeling. It delays the computation of other features to later iterations, which may cause users to wait longer between iterations. To help reduce the waiting time between iterations, we develop our third solution.

The key idea is to utilize user labeling time. Recall that in our experience, labeling one entity pair often takes at least 5 seconds, then labeling a batch of 20 pairs will take 100 seconds or more. In our first two solutions, when users label pairs from the current iteration, the cluster is completely idle. Therefore we decide to utilize user labeling time, which will be described next.

4.4.1 Utilize User Labeling Time

How should we utilize user labeling time to help reduce the user waiting time between iterations? The idea is simple: instead of waiting for the labels of all the 20 pairs then train the next model, we decide to train the next model after getting the labels of 10 pairs. That is, in the first iteration, we select the top 20 pairs and send the first 10 pairs for manual labeling. Once we get their labels, we send the other 10 pairs for manual labeling. Meanwhile we start to train the second model, compute any missing features required by the second model, then apply the second model to select the top 10 pairs from the remaining unlabeled pairs. From now on, we train a new model once we get 10 new labeled pairs, and use it to select the next top 10 pairs for labeling. With this idea, starting from the second iteration, we are able to train the next model concurrently when users label the previous 10 pairs. If the time to train the model, compute features and select

Dataset	Time (sec)				Accuracy (%)		V3 Avg Waiting Time Between Iterations (sec)
	Total Machine Time		Total User Waiting				
	V2	V3	V2	V3	V2	V3	
Amazon-Google	18.12	22.90	12.42	0	73.1	73.2	0
Walmart-Amazon	19.76	25.61	13.50	0	78.8	79.9	0
DBLP-Scholar	18.08	23.28	11.48	0	92.6	92.4	0
DBLP-ACM	16.67	21.68	10.81	0	99.4	99.6	0
Clothing	96.77	107.95	62.50	0	94.6	94.3	0
Tools	97.82	108.27	55.83	0	87.3	88.0	0
Clothing (x3)	174.12	192.60	118.68	0	94.9	94.7	0
Tools (x3)	170.17	188.71	105.11	0	88.3	88.4	0
Clothing (x5)	255.25	291.44	179.08	0	94.2	94.1	0
Tools (x5)	243.66	290.17	153.38	0	88.9	87.6	0
Clothing (x10)	443.89	499.98	317.57	52.79	94.6	94.0	1.82
Tools (x10)	445.41	493.62	296.47	33.87	88.4	87.3	1.17

Table 4.5: Reducing user waiting time.

the next top 10 pairs is no more than the time for users to label 10 pairs, then users can continue labeling without any waiting time.

Note that in our third solution, we need to train more models, therefore the total machine time will increase. However, from the second model, the time for model training, feature computation and selecting the next 10 pairs is concurrent with user labeling time, which means that if we measure the total time from the beginning to the end of active learning, this time is likely reduced.

4.4.2 Empirical Evaluation

We use the same datasets to evaluation our third solution, and compare it with our second solution (which uses both caching techniques and lazy feature computation). The results are shown in Table 4.5, in which columns with header ‘V2’ are for our second solution and columns with header ‘V3’ are for our third solution. The table contains the total machine time, the total user

waiting time after the first iteration, the accuracy of the last model before active learning terminates, and the average waiting time between iterations in V3.

As expected, the total machine time of our third solution is longer than that of our second solution, since our third solution trains more models in order to select and label 600 pairs in total. But now users are able to continue labeling without any waiting time between iterations for most of the datasets. For the last two rows, users still need to wait some time after the first iteration, which is due to that in some iteration, there are many missing features that need computation, therefore the next batch of 10 pairs is not ready when users finish labeling the previous batch of pairs. But our third solution still manages to significantly reduce the total user waiting time, compared with the second solution. We can expect that on larger datasets, our third solution will help reduce more user waiting time.

One concern with our third solution is whether the final learned model can still have similar accuracy. To evaluate the accuracy of the final models, we sample 1,000 pairs (half are matches and half are non-matches), then compute their model accuracy. From the table, we can clearly see that the final models from our second and third solutions have comparable accuracy on most datasets.

Figure 4.4 shows the accuracy of the models vs. the number of labeled pairs for both our second and third solutions. From these curves, we can see that even though the accuracy of models from the first few iterations may have huge difference, models from both solutions will have comparable accuracy with enough labeled pairs. Another interesting finding is that on many datasets, the accuracy of models converge before active learning terminates. However, the final models on dataset Amazon-Google seem not converged yet. It may be caused by the initial seeding pairs.

4.5 Related Work

Active learning is a hot topic among both the academic and industry communities. There are many surveys [84], tutorials [22, 73], and books about active learning. It has been applied to many task scenarios, and most of them can be categorized as pool-based active learning [67]. That is, we have a pool of unlabeled examples for selection during active learning.

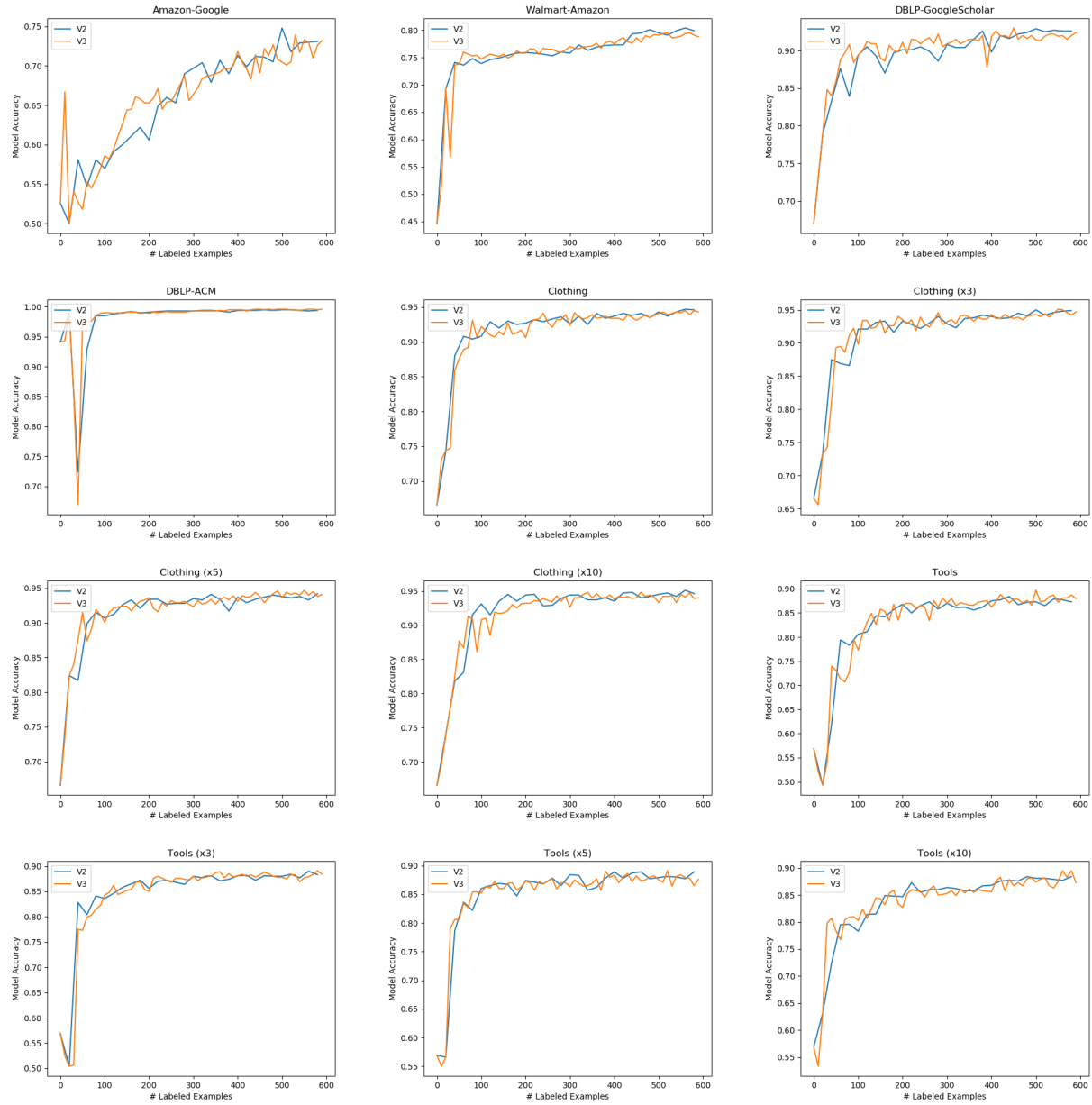


Figure 4.4: Model accuracy vs. number of labeled pairs over iterations.

Many researchers focus on developing query strategies to select examples for labeling, such as Uncertainty Sampling [67], Query-By-Committee [85], Density Weighted Methods [82, 94, 58], etc. Recent research works [29] also study how to make the target model converge faster, which often requires the target model follows some properties, therefore it's hard to generalize their ideas to black-box models.

Since active learning is proven useful in many applications and domains, there are more and more attentions on how to build active learning frameworks [59] and platforms [6, 1] to help users deploy their active learning tasks. Unfortunately these frameworks and platforms either face scalability issues, or can not be used for entity matching tasks.

[54, 19] build and scale up crowdsourced entity matching tasks. They focus on how to minimize user efforts with the help of crowdsourcing. They use active learning to collect labeled pairs for some entity matching steps, but their task scenario is different from ours, therefore their solutions are different from ours.

4.6 Conclusions and Future Work

Large-scale active learning on entity matching is an important data labeling task, yet there is little research on this topic. In this chapter we study this problem and develop solutions to address the scalability challenge. We evaluate our solutions extensively on several entity matching datasets. The results clearly show that our solutions help effectively reduce both the waiting time before users start labeling and the waiting time between iterations, which demonstrate the promise of our solutions.

Future Work: Active learning is a broad research topic and there are many directions we can continue as future work. First, in our third solution, we utilize user labeling time to train models. Another idea to utilize user labeling time and reduce user waiting time is to use the labeling time to compute features that might be needed in later iterations. How to identify such features might be a good challenge. Feature correlation with labels may be used as some criteria to identify them. Or we may train a different model such as SVM using the current set of labeled pairs, which can

provide us the importance of all features, therefore we can use the feature importance to rank features that have been computed and use user labeling time to compute them one feature after another. Next, one important research topic for active learning is detecting whether models have converged. In our experiments, we stop active learning after 600 pairs are labeled. But from Figure 4.4, we can see that on some datasets, models seem to have converged much earlier. We cannot use model accuracy for convergence detection since in real active learning, we don't have such testing set to evaluate model accuracy. For future work, we plan to explore how to effectively detect model convergence when performing active learning on entity matching datasets.

Chapter 5

Conclusions

Data labeling is the process of one or more users labeling a set of data instances using a set of given labels. It is a pervasive problem in many data science tasks, such as classification, data validation, tagging, etc. To collect high quality labels, data labeling usually requires a lot of manual effort. Researchers have developed many techniques to help data labeling process, such as crowdsourcing and active learning, but major challenges regarding cost, quality and scalability still remain. In this dissertation I develop solutions to address these challenges for several important data labeling tasks.

First, I have developed VChecker for validating data using crowdsourcing. In such tasks, crowdsourcing cost and accuracy of aggregated answers are two important factors that users often need to trade-off between each other. I developed solutions that estimate the difficulties of different values of the attribute to be validated and partition items in the dataset according to the value difficulties, then develop adaptive crowdsourcing strategies to crowdsource items in each partition. These solutions can be used towards different crowdsourcing task scenarios, such as minimizing crowdsourcing cost while maintaining the accuracy of aggregated answers, or maximizing the accuracy of aggregated answers with some budget limit, etc.

Second, in collaboration with Fatemah Panahi, I have developed solutions for detecting label errors for entity matching. Our interactive solutions significantly reduce the user workload. In each iteration, we find the top-k entity pairs whose labels are most suspicious, return them to the users for manual verification, then use their feedback (corrected labels) to find the top-k suspicious entity pairs for the next iterations. We perform extensive experiments on 17 entity matching datasets, which demonstrate the promise of our solutions.

Finally, I have developed solutions for performing large-scale active learning for entity matching. We start with a simple distributed Spark solution, study its performance on many entity matching datasets, identify opportunities to improve user labeling experience, then we implement several ideas based on our observations and evaluate their effectiveness. Our empirical evaluations show that our solutions effectively improve user labeling experience by significantly reducing both the waiting time before users start labeling and the waiting time between iterations.

Bibliography

- [1] Amazon sagemaker ground truth. <https://aws.amazon.com/sagemaker/groundtruth/>.
- [2] Apache spark. <https://spark.apache.org/>.
- [3] Gurobi. <http://www.gurobi.com/>.
- [4] Incremental random forests. <https://github.com/pconstr/irf>.
- [5] Jaccard index. https://en.wikipedia.org/wiki/Jaccard_index.
- [6] Prodigy. <https://prodi.gy/>.
- [7] Z. Abedjan et al. Detecting data errors: Where are we and what needs to be done? *PVLDB*, 9(12):993–1004, 2016.
- [8] A. Arasu et al. Towards a domain independent platform for data cleaning. *IEEE Data Eng. Bull.*, 34(3):43–50, 2011.
- [9] K. J. Arrow. A difficulty in the concept of social welfare. *Journal of political economy*, 58(4):328–346, 1950.
- [10] L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.
- [11] C. E. Brodley and M. A. Friedl. Identifying mislabeled training data. *J. Artif. Intell. Res.*, 11:131–167, 1999.
- [12] G. Cadamuro, R. Gilad-Bachrach, and X. Zhu. Debugging machine learning models. In *ICML Workshop on Reliable Machine Learning in the Wild*, 2016.

- [13] T. Carter. An introduction to information theory and entropy. *Complex systems summer school, Santa Fe*, 2007.
- [14] S. Chaudhuri, B. Chen, V. Ganti, and R. Kaushik. Example-driven design of efficient record matching queries. In *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, pages 327–338, 2007.
- [15] S. Chaudhuri et al. Data debugger: An operator-centric approach for data quality solutions. *IEEE Data Eng. Bull.*, 29(2):60–66, 2006.
- [16] X. Chu et al. Data cleaning: Overview and emerging challenges. In *SIGMOD*, 2016.
- [17] X. Chu et al. Distributed data deduplication. In *VLDB*, 2016.
- [18] X. Chu and I. F. Ilyas. Qualitative data cleaning. *PVLDB*, 9(13):1605–1608, 2016.
- [19] S. Das, P. S. G. C., A. Doan, J. F. Naughton, G. Krishnan, R. Deep, E. Arcaute, V. Raghavendra, and Y. Park. Falcon: Scaling up hands-off crowdsourced entity matching to build cloud services. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 1431–1446, 2017.
- [20] S. Das et al. Falcon: Scaling up hands-off crowdsourced entity matching to build cloud services. In *SIGMOD*, 2017.
- [21] A. Das Sarma et al. An automatic blocking mechanism for large-scale de-duplication tasks. In *CIKM*, 2012.
- [22] S. Dasgupta and J. Langford. A tutorial on active learning. In *Proceedings of ICML*, 2009.
- [23] T. Dasu and T. Johnson. *Exploratory Data Mining and Data Cleaning*. John Wiley, 2003.
- [24] H. Deng and G. C. Runger. Feature selection via regularized trees. In *The 2012 International Joint Conference on Neural Networks (IJCNN), Brisbane, Australia, June 10-15, 2012*, pages 1–8. IEEE, 2012.

- [25] X. Dong et al. Global detection of complex copying relationships between sources. *PVLDB*, 3(1):1358–1369, 2010.
- [26] V. Efthymiou et al. Parallel meta-blocking: Realizing scalable entity resolution over large, heterogeneous data. In *Big Data*, 2015.
- [27] C. Emmanouilidis, P. Pistofidis, L. Bertonecelj, V. Katsouros, A. P. Fournaris, C. Koulamas, and C. Ruiz-Carcel. Enabling the human in the loop: Linked data and knowledge in industrial cyber-physical systems. *Annual Reviews in Control*, 47:249–265, 2019.
- [28] A. G. P. et al. 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*, pages 361–372, 2012.
- [29] B. M. et al. Scaling up crowd-sourcing to very large datasets: A case for active learning. *PVLDB*, 8(2):125–136, 2014.
- [30] C. G. et al. Corleone: hands-off crowdsourcing for entity matching. In *SIGMOD*, pages 601–612, 2014.
- [31] D. H. et al. Wisteria: Nurturing scalable data cleaning infrastructure. *PVLDB*, 8(12):2004–2007, 2015.
- [32] H. L. et al. The wisdom of minority: discovering and targeting the right group of workers for crowdsourcing. In *WWW*, pages 165–176, 2014.
- [33] J. C. et al. Measuring crowdsourcing effort with error-time curves. In *ACM CHI*, pages 1365–1374, 2015.
- [34] J. W. et al. Whose vote should count more: Optimal integration of labels from labelers of unknown expertise. In *Advances in neural information processing systems*, pages 2035–2043, 2009.
- [35] J. W. et al. Crowder: Crowdsourcing entity resolution. *PVLDB*, 5(11):1483–1494, 2012.

- [36] L. M. et al. Optimizing task assignment for crowdsourcing environments. Technical report, Citeseer, 2013.
- [37] P. I. et al. Quality management on amazon mechanical turk. In *ACM SIGKDD workshop on human computation*, pages 64–67. ACM, 2010.
- [38] P. M. et al. Isotone optimization in r: pool-adjacent-violators algorithm (pava) and active set methods. *Journal of statistical software*, 32(5):1–24, 2009.
- [39] Q. D. et al. Deep learning for gender recognition. In *ICCCS*, pages 206–209, 2015.
- [40] S. J. et al. Reputation-based worker filtering in crowdsourcing. In *Advances in Neural Information Processing Systems 27*, pages 2492–2500, 2014.
- [41] S. O. et al. Accurate integration of crowdsourced labels using workers’ self-reported confidence scores. In *IJCAI*, 2013.
- [42] V. R. et al. Supervised learning from multiple experts: whom to trust when everyone lies a bit. In *ICML*, pages 889–896, 2009.
- [43] X. C. et al. KATARA: A data cleaning system powered by knowledge bases and crowdsourcing. In *SIGMOD*, pages 1247–1261, 2015.
- [44] X. L. et al. CDAS: A crowdsourcing data analytics system. *PVLDB*, 5(10):1040–1051, 2012.
- [45] Y. A. et al. CrowDMINER: Mining association rules from the crowd. *PVLDB*, 6(12):1250–1253, 2013.
- [46] Y. T. et al. CrowdCleaner: Data cleaning for multi-version data on the web via crowdsourcing. In *IEEE*, pages 1182–1185, 2014.
- [47] Z. K. et al. BigDancing: A system for big data cleansing. In *SIGMOD*, pages 1215–1230, 2015.

- [48] J. Fraenkel and B. Grofman. The borda count and its real-world alternatives: Comparing scoring rules in nauru and slovenia. *Australian Journal of Political Science*, 49(2):186–205, 2014.
- [49] M. J. Franklin et al. Crowddb: answering queries with crowdsourcing. In *SIGMOD*, 2011.
- [50] J. Freire et al. Exploring what not to clean in urban data: A study using new york city taxi trips. *IEEE Data Eng. Bull.*, 39(2):63–77, 2016.
- [51] B. Frénay and A. Kabán. A comprehensive introduction to label noise. In *22th European Symposium on Artificial Neural Networks, ESANN 2014, Bruges, Belgium, April 23-25, 2014*, 2014.
- [52] H. Galhardas et al. Declarative data cleaning: Language, model, and algorithms. In *VLDB*, 2001.
- [53] M. Gil, M. Albert, J. Fons, and V. Pelechano. Designing human-in-the-loop autonomous cyber-physical systems. *Int. J. Hum.-Comput. Stud.*, 130:21–39, 2019.
- [54] C. Gokhale, S. Das, A. Doan, J. F. Naughton, R. Rampalli, J. Shavlik, and X. Zhu. Corleone: Hands-off crowdsourcing for entity matching. In *SIGMOD*, 2014.
- [55] D. Haas et al. CLAMShell: Speeding up crowds for low-latency data labeling. In *VLDB*, 2016.
- [56] L. Hamers et al. Similarity measures in scientometric research: The jaccard index versus salton’s cosine formula. *Information Processing and Management*, 25(3):315–18, 1989.
- [57] J. Heer et al. Predictive interaction for data transformation. In *CIDR*, 2015.
- [58] S. C. H. Hoi, R. Jin, J. Zhu, and M. R. Lyu. Batch mode active learning and its application to medical image classification. In W. W. Cohen and A. W. Moore, editors, *Machine Learning*,

- Proceedings of the Twenty-Third International Conference (ICML 2006), Pittsburgh, Pennsylvania, USA, June 25-29, 2006*, volume 148 of *ACM International Conference Proceeding Series*, pages 417–424. ACM, 2006.
- [59] K. G. Jamieson, L. Jain, C. Fernandez, N. J. Glattard, and R. D. Nowak. NEXT: A system for real-world development, evaluation, and application of active learning. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, 2015.
- [60] T. Joachims. Optimizing search engines using clickthrough data. In *SIGKDD*, pages 133–142, 2002.
- [61] A. Khan and H. Garcia-Molina. Crowddqs: Dynamic question selection in crowdsourcing systems. In *SIGMOD*, 2017.
- [62] R. Kohavi and G. H. John. Wrappers for feature subset selection. *Artif. Intell.*, 97(1-2):273–324, 1997.
- [63] L. Kolb et al. Parallel sorted neighborhood blocking with MapReduce. In *BTW*, 2011.
- [64] P. Konda et al. Magellan: Toward building entity matching management systems. *PVLDB*, 9(12):1197–1208, 2016.
- [65] H. Köpcke, A. Thor, and E. Rahm. Evaluation of entity resolution approaches on real-world match problems. *PVLDB*, 3(1):484–493, 2010.
- [66] S. Krishnan et al. Activeclean: Interactive data cleaning for statistical modeling. *PVLDB*, 9(12):948–959, 2016.
- [67] D. D. Lewis. A sequential algorithm for training text classifiers: Corrigendum and additional data. *SIGIR Forum*, 29(2):13–19, 1995.
- [68] G. Li. Human-in-the-loop data integration. *PVLDB*, 10(12):2006–2017, 2017.

- [69] T. Liu and D. Tao. Classification with noisy labels by importance reweighting. *IEEE Trans. Pattern Anal. Mach. Intell.*, 38(3):447–461, 2016.
- [70] A. Marcus et al. Crowdsourced databases: Query processing with people. In *CIDR*, 2011.
- [71] B. Mozafari et al. Scaling up crowd-sourcing to very large datasets: A case for active learning. In *VLDB*, 2014.
- [72] S. Mudgal, H. Li, T. Rekatsinas, A. Doan, Y. Park, G. Krishnan, R. Deep, E. Arcaute, and V. Raghavendra. Deep learning for entity matching: A design space exploration. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 19–34, 2018.
- [73] R. Nowak and S. Hanneke. Active learning: From theory to practice. In *Proceedings of ICML*, 2019.
- [74] J. O’Connor and E. Robertson. The history of voting. *The MacTutor History of Mathematics Archive. sl, sn*, 2002.
- [75] A. G. Parameswaran and N. Polyzotis. Answering queries using humans, algorithms and databases. In *CIDR*, 2011.
- [76] H. Park and J. Widom. Query optimization over crowdsourced data. In *VLDB*, 2013.
- [77] G. Patrini, A. Rozza, A. K. Menon, R. Nock, and L. Qu. Making deep neural networks robust to label noise: A loss correction approach. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*, pages 2233–2241, 2017.
- [78] M. Pivato. Condorcet meets bentham. *Journal of Mathematical Economics*, 59:58–65, 2015.
- [79] E. Rahm and H. H. Do. Data cleaning: Problems and current approaches. *IEEE Data Eng. Bull.*, 23(4):3–13, 2000.

- [80] V. Raykar and S. Yu. Eliminating spammers and ranking annotators for crowdsourced labeling tasks. *Journal of Machine Learning Research*, 13:491–518, 2012.
- [81] D. P. Rodgers. Improvements in multiprocessor system design. In *Proceedings of the 12th Annual Symposium on Computer Architecture, Boston, MA, USA, June 1985*, pages 225–231, 1985.
- [82] N. Roy and A. McCallum. Toward optimal active learning through sampling estimation of error reduction. In C. E. Brodley and A. P. Danyluk, editors, *Proceedings of the Eighteenth International Conference on Machine Learning (ICML 2001), Williams College, Williamstown, MA, USA, June 28 - July 1, 2001*, pages 441–448. Morgan Kaufmann, 2001.
- [83] M. Schulze. A new monotonic, clone-independent, reversal symmetric, and condorcet-consistent single-winner election method. *Social Choice and Welfare*, 36(2):267–303, 2011.
- [84] B. Settles. Active learning literature survey. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 2009.
- [85] H. S. Seung, M. Opper, and H. Sompolinsky. Query by committee. In D. Haussler, editor, *Proceedings of the Fifth Annual ACM Conference on Computational Learning Theory, COLT 1992, Pittsburgh, PA, USA, July 27-29, 1992*, pages 287–294. ACM, 1992.
- [86] C. Spearman. The proof and measurement of association between two things. *The American Journal of Psychology*, 100(3/4):441–471, 1987.
- [87] Y. Sun, S. Todorovic, and S. Goodison. Local-learning-based feature selection for high-dimensional data analysis. *IEEE Trans. Pattern Anal. Mach. Intell.*, 32(9):1610–1626, 2010.
- [88] A. Vahdat. Toward robustness against label noise in training deep discriminative neural networks. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*, pages 5596–5605, 2017.

- [89] V. Verroios et al. Waldo: An adaptive human interface for crowd entity resolution. In *SIGMOD*, 2017.
- [90] J. Wang et al. Leveraging transitive relations for crowdsourced joins. In *SIGMOD*, 2013.
- [91] T. Xiao, T. Xia, Y. Yang, C. Huang, and X. Wang. Learning from massive noisy labeled data for image classification. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*, pages 2691–2699, 2015.
- [92] D. Xin, L. Ma, J. Liu, S. Macke, S. Song, and A. G. Parameswaran. Helix: Accelerating human-in-the-loop machine learning. *PVLDB*, 11(12):1958–1961, 2018.
- [93] Y. Yang and J. O. Pedersen. A comparative study on feature selection in text categorization. In D. H. Fisher, editor, *Proceedings of the Fourteenth International Conference on Machine Learning (ICML 1997), Nashville, Tennessee, USA, July 8-12, 1997*, pages 412–420. Morgan Kaufmann, 1997.
- [94] X. Zhu, J. Lafferty, and Z. Ghahramani. Combining active learning and semi-supervised learning using gaussian fields and harmonic functions. In *ICML 2003 workshop on the continuum from labeled to unlabeled data in machine learning and data mining*, volume 3, 2003.

A Main Codes for Our First Solution in Chapter 4

Also available at repository <https://github.com/hzhang0418/al4em>

```

1  '''
2  Baseline solution
3  '''
4  import os
5  import copy
6  import heapq
7  import numpy as np
8  import time
9  import pandas as pd
10 from pyspark import SparkContext
11
12 import seeder
13 import labeler
14 import driver
15 import cachemone
16 import helper
17
18 def run(sc, table_A, table_B, candidate_pairs, table_G, feature_table, feature_info,
19        num_executors, seed_size, max_iter, batch_size):
20
21     # prepare driver
22     # driver node
23     driver = driver.Driver()
24     driver.prepare(table_A, table_B, feature_table, helper.tok_name2func,
25                  helper.sim_name2func)
26
27     # seeds
28     seeder = seeder.Seeder(table_G)
29     labeler = labeler.Labeler(table_G)
30
31     # partition pairs
32     pair_rdd = sc.parallelize(candidate_pairs, num_executors)
33     bc_table_A = sc.broadcast(table_A)
34     bc_table_B = sc.broadcast(table_B)
35     bc_feature_info = sc.broadcast(feature_info)
36
37     # compute feature vectors
38     ex_rdd = pair_rdd.mapPartitions(
39         lambda pairs_partition: create_executors(pairs_partition, bc_table_A,
40         bc_table_B, bc_feature_info, num_executor, cache_level),
41         preservesPartitioning=True)
42     ex_rdd.cache()
43
44     # simulate active learning
45     # select seeds
46     pair2label = seeder.select(seed_size)
47     exclude_pairs = set(pair2label.keys())
48
49     num_iter = 0
50     all_features = set()
51
52     while num_iter < max_iter:
53         driver.add_new_training(pair2label)
54         # train model
55         rf = driver.train()
56         # features in RF
57         required_features = nodes.helper.get_features_in_random_forest(rf)
58         all_features.update(required_features)
59
60         # select most informative examples
61         candidates = ex_rdd.mapPartitions(
62             lambda executors: iteration(executors, rf, batch_size, exclude_pairs),
63             preservesPartitioning=True).collect()
64
65         # select top k from candidate
66         top_k = heapq.nlargest(batch_size, candidates, key=lambda p: p[1])
67         top_k_pairs = [ t[0] for t in top_k ]

```

```

68     pair2label = labeler.label(top_k_pairs)
69     exclude_pairs.update(top_k_pairs)
70
71     num_iter += 1
72
73     ex_rdd.unpersist()
74
75
76
77 # map functions that apply to each partition
78 def create_executors(pairs_partition, bc_table_A, bc_table_B, bc_feature_info):
79     pairs = [p for p in pairs_partition ]
80     # executor node
81     executor = cachemone.CacheNone()
82     executor.prepare(bc_table_A.value, bc_table_B.value, bc_feature_info.value, pairs)
83     executor.compute_features(list(range(len(bc_feature_info.value))),
84                               bc_feature_info.value, bc_table_A.value, bc_table_B.value)
85     return [executor]
86
87 def iteration(executors, rf, batch_size, exclude_pairs):
88     combined = []
89     for executor in executors:
90         # apply random forest and select most informative examples
91         top_k = executor.apply(rf, batch_size, exclude_pairs)
92         combined.extend(top_k)
93     return combined
94
95 '''
96 driver.py
97 '''
98 import numpy as np
99 import pandas as pd
100
101 from sklearn.ensemble import RandomForestClassifier
102 from sklearn.tree import DecisionTreeClassifier
103 from sklearn.utils import shuffle
104
105 import py_entitymatching as em
106
107 import helper
108
109 class Driver:
110
111     def __init__(self):
112         # table A, B and candidate pairs
113         self.table_A = None
114         self.table_B = None
115
116         # feature table
117         self.feature_table = None
118         self.tok_name2func = None
119         self.sim_name2func = None
120
121         # features and labels, used for training model
122         self.features = None
123         self.labels = None
124
125         # labeled pairs
126         self.pair2label = {}
127
128     def prepare(self, table_A, table_B, feature_table, tok_name2func, sim_name2func):
129         self.table_A = table_A
130         self.table_B = table_B
131         self.feature_table = feature_table
132         self.tok_name2func = tok_name2func
133         self.sim_name2func = sim_name2func
134

```

```

135 def add_new_training(self, pair2label: dict) -> None:
136     pairs = []
137     labels = []
138     for k,v in pair2label.items():
139         if k not in self.pair2label: # only need to compute features for new pairs
140             pairs.append(k)
141             labels.append(v)
142             self.pair2label[k] = v
143
144     self._compute_features(pairs)
145     if self.labels is None:
146         self.labels = np.array(labels, dtype=int)
147     else:
148         self.labels = np.hstack( (self.labels, np.array(labels, dtype=int)) )
149
150 def _compute_features(self, pairs: list) -> None:
151
152     features_new = np.empty( (len(pairs), len(self.feature_table)), dtype=np.float32)
153     try:
154         f = 0
155         for fs in self.feature_table.itertuples(index=False):
156             lattr = getattr(fs, 'left_attribute')
157             rattr = getattr(fs, 'right_attribute')
158             ltok = getattr(fs, 'left_attr_tokenizer')
159             rtok = getattr(fs, 'right_attr_tokenizer')
160             simfunc = self.sim_name2func[ getattr(fs,'simfunction') ]
161             #func = getattr(fs, 'function')
162
163             if ltok is None:
164                 for k, pair in enumerate(pairs):
165                     ltable_id, rtable_id = pair[0], pair[1]
166                     ltable_value = self.table_A.loc[ltable_id][lattr]
167                     rtable_value = self.table_B.loc[rtable_id][rattr]
168                     features_new[k][f] = simfunc(ltable_value, rtable_value)
169             else:
170                 ltokfunc = self.tok_name2func[ltok]
171                 rtokfunc = self.tok_name2func[rtok]
172                 for k, pair in enumerate(pairs):
173                     ltable_id, rtable_id = pair[0], pair[1]
174                     ltable_value = self.table_A.loc[ltable_id][lattr]
175                     rtable_value = self.table_B.loc[rtable_id][rattr]
176                     features_new[k][f] = simfunc(ltokfunc(ltable_value), rtokfunc(
177                         rtable_value))
178                 f += 1
179     except ValueError:
180         print(pair, ltable_value, rtable_value)
181         raise
182
183     np.nan_to_num(features_new, copy=False)
184
185     if self.features is None:
186         self.features = features_new
187     else:
188         self.features = np.vstack( (self.features, features_new) )
189
190 def train(self) -> RandomForestClassifier:
191     #features, labels = self.features, self.labels
192     #features, labels = shuffle(self.features, self.labels)
193     features, labels = shuffle(self.features, self.labels, random_state=0)
194     rf = RandomForestClassifier(n_estimators=10, max_depth=None,
195                               max_features='auto', random_state=0, n_jobs=1)
196     #rf = DecisionTreeClassifier(random_state=0)
197     rf.fit(features, labels)
198     return rf
199
200

```

```

201
202 '''
203 cacheneone.py
204 '''
205 import heapq
206
207 import numpy as np
208 from scipy.stats import entropy
209 from sklearn.ensemble import RandomForestClassifier
210
211 import helper
212
213 class CacheNone:
214
215     def __init__(self):
216         # pairs assigned to this node
217         self.pairs = None # list of (ltable_id, rtable_id)
218         self.features = None # numpy array of features
219
220     def prepare(self, table_A, table_B, feature_info, pairs):
221         self.pairs = pairs
222         self.features = np.zeros( (len(self.pairs), len(feature_info)), dtype=np.float32
223         )
224
225     def compute_features(self, required_features, feature_info, table_A, table_B):
226         if len(required_features)==0:
227             return None
228
229         # no cache, therefore fetch each pair, then compute required features
230         for k, pair in enumerate(self.pairs):
231             ltuple = table_A.loc[pair[0]]
232             rtuple = table_B.loc[pair[1]]
233
234             for f in required_features:
235                 fs = feature_info.iloc[f]
236                 lattr = getattr(fs, 'left_attribute')
237                 rattr = getattr(fs, 'right_attribute')
238                 ltok = getattr(fs, 'left_attr_tokenizer')
239                 rtok = getattr(fs, 'right_attr_tokenizer')
240                 simfunc = nodes.helper.sim_name2func[ getattr(fs, 'simfunction') ]
241
242                 if ltok==None:
243                     value = simfunc(ltuple[lattr], rtuple[rattr])
244                 else:
245                     ltokfunc = nodes.helper.tok_name2func[ltok]
246                     rtokfunc = nodes.helper.tok_name2func[rtok]
247                     value = simfunc( ltokfunc(ltuple[lattr]), rtokfunc(rtuple[rattr]) )
248
249                 if np.isnan(value):
250                     value = 0
251                 self.features[k,f] = value
252
253     def apply(self, rf: RandomForestClassifier, k: int, exclude_pairs: set) -> list:
254         # prediction
255         proba = rf.predict_proba(self.features)
256         entropies = np.transpose(entropy(np.transpose(proba), base=2))
257
258         # select top k, return list of pairs of (index, entropy)
259         candidates = [ (self.pairs[k],v) for k,v in enumerate(entropies)
260             if self.pairs[k] not in exclude_pairs ]
261         top_k = heapq.nlargest(k, candidates, key=lambda p: p[1])
262
263         return top_k
264
265
266

```

```
267 '''
268 helper.py
269 '''
270 import random
271 from sklearn.ensemble import RandomForestClassifier
272 from sklearn.tree import DecisionTreeClassifier
273
274 import py_entitymatching as em
275
276 tok_name2func = em.get_tokenizers_for_matching(q = [2,3,4,5])
277 sim_name2func = em.get_sim_funs_for_matching()
278
279 def get_features_in_random_forest(rf: RandomForestClassifier) -> list:
280     rf_features = set()
281     for tree in rf.estimators_:
282         for f in tree.tree_.feature:
283             if f != -2:
284                 rf_features.add(f)
285     return list(rf_features)
286
287 def get_features_in_decision_tree(tree: DecisionTreeClassifier) -> list:
288     tree_features = set()
289     for f in tree.tree_.feature:
290         if f != -2:
291             tree_features.add(f)
292     return list(tree_features)
293
```