CORLEONE: HANDS-OFF CROWDSOURCING FOR ENTITY MATCHING

by

Chaitanya Gokhale

A dissertation submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2015

Date of final oral examination: 07/30/2015

The dissertation is approved by the followin members of the Final Oral Committee:
AnHai Doan, Professor, Computer Sciences
Jeffrey F. Naughton, Professor, Computer Sciences
Xiaojin Zhu, Associate Professor, Computer Sciences
Jude Shavlik, Professor, Computer Sciences
Rafael Lazimy, Associate Professor, Operations & Information Management

© Copyright by Chaitanya Gokhale 2015

All Rights Reserved

To my parents.

ACKNOWLEDGMENTS

I would like to begin by thanking my advisor, Professor AnHai Doan, for patiently guiding me throughout this long endeavor. Without his constant encouragement and supervision, I might not have made it this far. I am especially grateful to him for his wholehearted support when I decided to switch my dissertation topic in the middle of my graduate study program. That decision opened the door to the exciting world of crowdsourcing.

I must thank the professors in the database group that I had the fortune to interact with, David DeWitt, Jeffrey F. Naughton, Jignesh Patel, and Chris Ré. I got to learn much more than I could ask for from each one of them. I will especially miss the weekly database seminar, which was a great place not just to learn about research (and eat free food), but also to learn about communication skills.

During my studies, I had the opportunity to interact with a great variety of students from the Computer Sciences department who provided critical feedback and friendly support. In particular, I would like to thank Adel Ardalan, Akanksha Baid, Arun Kumar, Ba-Quy Vuong, Chen Zeng, Fei Chen, Ian Rae, Jessie Li, Khai Tran, Spyros Blanas, Sanjib Das, Ting Chen, Warren Shen, and Xiaoyong Chai.

During my time in Madison, I made a great network of friends who helped me stay sane in the most critical moments and helped me create some wonderful memories. This list would be too long if I were to include all of them, but I am particularly grateful to Akshar Punuganti, Aditya Gore, Hidayath Ansari, Mayank Maheshwari, Ming Li, Sarah Olson, Santhosh Ramani, Sarang Brahma, Sasank Yarlagadda, Saurabh Maiti, Subhash Thapa, Shilpa Nagarajan, and Varun Rao.

Thanks to my family for their love and encouragement. Thanks to WalmartLabs for supporting my research. Special thanks to Professors Jude Shavlik and Xiaojin Zhu for their critical input.

Last, but not the least, thanks to all the turkers who participated in my experiments, provided useful feedback, and made this dissertation possible.

TABLE OF CONTENTS

Page

LIST OF TABLES vii

LIST OF FIGURES viii

ABSTRACT ix

1 Introduction 1

1.1 Entity Matching 2
1.2 Crowdsourced Entity Matching and Current Limitations 4
1.3 Our Solution: Hands-Off Crowdsourcing for Entity Matching 7
1.3.1 Hands-Off Crowdsourcing (HOC) 7
1.3.2 Corleone: A HOC System for Entity Matching 7
1.4 Contributions and Outline of the Dissertation 10

2 Background and Related Work 12

2.1 Entity Matching 12
2.2 Crowdsourced Entity Matching 14
2.2.1 Blocking 15
2.2.2 Matching 16
2.2.3 Accuracy Estimation 16
2.3 Crowdsourcing Systems 17
2.3.1 Crowsourcing for Data Management 18
2.3.2 Platforms for Building a Crowdsourcing System for EM 18

3 Proposed Solution 22

3.1 Hands-Off Crowdsourcing 22
3.2 The Corleone Solution 24
3.2.1 Input to Corleone 24
3.2.2 Corleone's Workflow 25

4 Blocking To Reduce the Set of Candidate Pairs 26

4.1 Deciding Whether to Do Blocking 27

Page

4.2 Generating Candidate Blocking Rules 28
4.2.1 Taking a Small Sample 28
4.2.2 Applying Crowdsourced Active Learning 30
4.2.3 Extracting Candidate Blocking Rules 30
4.3 Evaluating Rules Using the Crowd 31
4.3.1 Selecting Blocking Rules 32
4.3.2 Evaluating the Selected Rules Using the Crowd 33
4.4 Applying Blocking Rules 36

5 Training and Applying a Matcher 41

5.1 Training the Initial Matcher 41
5.2 Consuming the Next Batch of Examples 44
5.3 Deciding When to Stop 45

6 Estimating Matching Accuracy 49

6.1 Current Methods and Their Limitations 49
6.2 Crowdsourced Estimation with Corleone 50
6.2.1 Generating Candidate Reduction Rules 51
6.2.2 Repeating a Probe-Eval-Reduce Loop 51
6.2.3 Optimizations 57

7 Iterating to Improve 61

8 Engaging the Crowd 65

8.1 Crowdsourcing Platforms 65
8.2 Combining Noisy Crowd Answers 67
8.3 Re-using Labeled Examples 68

9 Empirical Evaluation 70

9.1 Overall Performance 70
9.2 Performance of the Components 73
9.3 Additional Experimental Results 76
9.4 Sensitivity Analysis 79
9.5 Setting the System Parameters 84

Page

10 Discussion 86

10.1 Design Choices 86
10.1.1 Blocking Threshold 87
10.1.2 Sampling to Learn Blocking Rules 89
10.1.3 Labeling Scheme for Crowdsourcing 90
10.2 Opportunities for Extension 90
10.2.1 Scaling Up to Very Large Datasets 91
10.2.2 Improving Matching and Estimation 93
10.2.3 Cost Models 94
10.2.4 Other Extensions 94
10.2.5 Applying to Other Problem Settings 95

11 Conclusion 96

Bibliography 98

APPENDIX Additional Empirical Data 104

LIST OF TABLES

Table Page

2.1 Comparison of general-purpose entity matching systems. 14

9.1 Data sets for our experiment. 71

9.2 Comparing the performance of Corleone against that of traditional solutions and published works. 71

9.3 Blocking results for Corleone. 73

9.4 Corleone's performance per iteration on the data sets. 74

9.5 Execution status of Corleone at different time points during one particular run on Products. 78

10.1 Stratified sampling for blocking. 92

A.1 Attributes of input tables for the datasets described in Chapter 9. 105

A.2 Top candidate rules generated by Corleone for one of the runs for Products dataset. . . . 107

A.3 Sample blocking rules applied by developer and Corleone (multiple such rules may be applied). 108

LIST OF FIGURES

Figure Page

1.1 Product matching for comparison shopping. 2

1.2 Verifying predicted matches using crowdsourcing. 5

3.1 The Corleone architecture. 24

4.1 (a)-(b) A toy random forest consisting of two decision trees, and (c) negative rules extracted from the forest. 31

4.2 Coverage and precision of rule R over S. 32

4.3 Example illustrating joint evaluation of rules. 34

5.1 Crowdsourced active learning in Corleone. 41

5.2 Example: candidate set and feature vectors. 43

5.3 Typical confidence patterns that we can exploit for stopping. 46

6.1 Example to illustrate the estimation process. 54

8.1 A sample question to the crowd. 66

9.1 Comparing estimation cost of Corleone vs. Baseline. 75

9.2 Sensitivity analysis for parameters in learning. 81

ABSTRACT

Entity matching (EM) identifies data records that refer to the same real-world entity. Recent approaches have considered applying crowdsourcing (i.e., outsourcing parts of the problem to a crowd of workers) to EM. These approaches have clearly established the promise of crowdsourced EM. However, they are limited in that they crowdsource only parts of the EM workflow, *requiring a software developer* to execute the remaining parts. Consequently, these approaches do not scale to the growing EM need at enterprises and crowdsourcing startups, and cannot handle scenarios where ordinary users (i.e., the masses) want to leverage crowdsourcing to match entities. To address these problems, we propose the notion of *hands-off crowdsourcing (HOC)*, which crowdsources the entire workflow of a task, thus requiring no developers. We show how HOC can represent a next logical direction for crowdsourcing research, scale up EM at enterprises and crowdsourcing startups, and open up crowdsourcing for the masses. We describe Corleone, a HOC solution for EM, which uses the crowd in all major steps of the EM process. Finally, we discuss the implications of our work to executing crowdsourced RDBMS joins, cleaning learning models, and soliciting complex information types from crowd workers.

Chapter 1

Introduction

Entity matching (EM) is the problem of finding data records that refer to the same real-world entity, such as (David Smith, JHU) and (D. Smith, John Hopkins). Entity matching is a critical step in numerous applications, such as comparison shopping, knowledge base construction, citation tracking, and inventory management at enterprises. This problem has received significant attention (see a recent survey by Elmagarmid et al. [48], and books by Doan et al. [42] and Christen [35]). However, no satisfactory solution has yet been found. In particular, there is still no EM solution that is robust across different problem domains and works out-of-the-box without requiring substantial developer effort.

In the past few years, crowdsourcing has been increasingly applied to EM. In crowdsourcing, certain parts of a problem are "farmed out" to a crowd of workers to solve. As such, crowdsourcing is well suited for EM, and indeed several crowdsourced EM solutions have been proposed (e.g., [41, 86, 87, 89, 90]). While these solutions demonstrate that crowdsourced EM is highly promising, they suffer from a major limitation: they crowdsource only parts of the EM workflow, thus requiring a developer to execute the remaining parts. As a result, current crowdsourced EM solutions do not scale to the growing EM need at enterprises and crowdsourcing startups, and can not handle scenarios where ordinary users want to leverage crowdsourcing to match entities. The goal of this dissertation is to address these limitations of current crowdsourced EM solutions.

In this chapter, we begin by highlighting the importance of entity matching in real-world data processing workflows. Next we review recent work on crowdsourced entity matching and identify limitations of current solutions that severely restrict their applicability. We then introduce hands-off crowdsourcing, our novel approach to crowdsourcing entity matching which addresses these

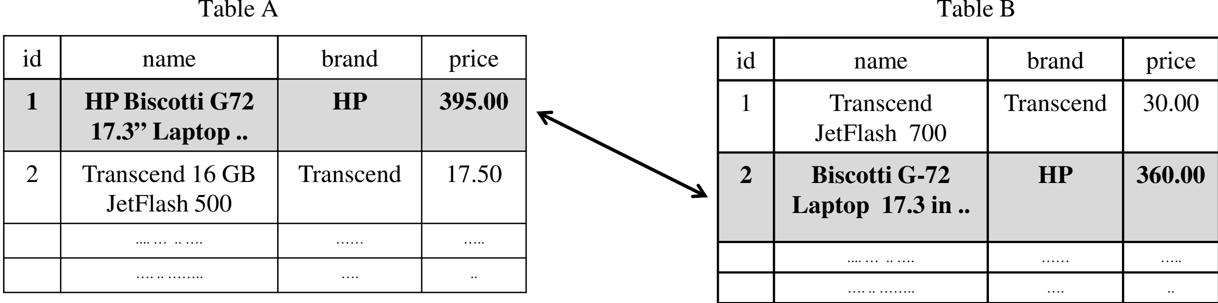


Figure 1.1: Product matching for comparison shopping.

limitations. We describe Corleone, our hands-off crowdsourcing solution for entity matching. Finally, we list the contributions and give a road map to the rest of the dissertation.

1.1 Entity Matching

Entity matching, also known as data matching, record linkage, duplicate detection, or entity resolution, has received significant attention over the past several decades [35]. Researchers have studied a variety of settings for entity matching, such as deduplicating records in a table [61, 77], matching two tables [25, 49], and collective matching [27, 44]. In this dissertation, we consider the commonly encountered setting of matching two tables, i.e., finding all tuple pairs $(a \in A, b \in B)$ from two relational tables A and B that refer to the same real-world entity. The following example illustrates this setting:

Example 1.1.1. Figure 1.1 shows two tables A and B containing product descriptions from two different retailers, say Amazon and Walmart, respectively. The two tables have identical sets of attributes. Given these tables, our goal is to return all *matching* tuple pairs from the two tables, i.e., tuple pairs that refer to the same real-world product. For instance, the tuple from table A with id = 1 and the tuple from table B with id = 2 refer to the exact same laptop, i.e., (1,2) is a matching tuple pair. □

Entity matching plays a critical role in many real-world applications. It is an essential step in building comparison shopping Web sites such as Google Shopping [4], Nextag [10], and Price-Grabber [12]. These sites allow customers to compare prices for a particular product from multiple online retailers. A big challenge in building such a site is identifying product descriptions from different retailers that refer to the same real-world product, e.g., "HP Biscotti 17.3" G72 laptop" and "Biscotti G-72 laptop 17.3 in" in Figure 1.1.1. To find all such product pairs we need to perform entity matching.

As another example, entity matching is of great value to the health-care industry. Patient data is often spread across hospitals, insurance companies, and pharmacies. Matching the patient data from all these sources provides a single view of patient data to doctors and health-care researchers. Among its many applications, this can allow researchers to identify key disease patterns, e.g., matching patient addresses to spatial data can help identify local hot-spots for diseases and correlations among diseases [46, 52]. Matching such data, however, is non-trivial due to discrepancies in the way it is represented across different sources, e.g., for the same patient two hospitals may have different addresses, or slightly different names on record. To perform such matching tasks, robust and accurate entity matching solutions are necessary.

As yet another example, national census agencies around the world collect data about various aspects of the population, such as income, health, and education. These agencies often need to collect and collate records from several sources such as past census collections, existing health and economic surveys, and administrative databases. Entity matching is an important tool used in creating highly accurate census data, e.g., matching people records to eliminate duplicates (duplicates can highly bias the census statistics) and to identify conflicting or missing information (e.g., matching income from tax, survey, and census sources) [53, 91].

Other real-world applications where EM plays a key role include knowledge base construction [51], data warehousing [24], business mailing lists for marketing [35], and master data management at enterprises [92].

1.2 Crowdsourced Entity Matching and Current Limitations

Several approaches have been proposed for entity matching, such as rule-based matching [37, 51, 57], supervised learning [30, 47], clustering [71], probabilistic models [49, 76], and collective matching [27, 44]. However, existing solutions are still far from perfect, i.e., there is still no EM solution that is robust across different problem domains and works out-of-the-box without requiring substantial developer effort. Thus, there are various ongoing efforts to improve the state of the art of entity matching [33, 41, 75, 86].

One such research effort that has gained significant momentum in recent years is applying crowdsourcing to entity matching. In crowdsourcing, certain parts of a problem are "farmed out" to a crowd of workers to solve. As such, crowdsourcing is well suited for EM, and indeed several crowdsourced EM solutions have been proposed (e.g., [41, 83, 86, 87, 89, 90]). These solutions can be broadly categorized into three groups:

- using the crowd to verify matches predicted by traditional EM solutions [41, 86, 87],
- finding the best questions to ask the crowd to minimize the total number of questions asked [83, 89], and
- finding the best user interface to pose questions to the crowd (e.g., whether to display one pair or ten pairs per page, and whether to show pairs of records or clusters of records) [66, 90].

Example 1.2.1. To illustrate how recent work uses the crowd to complement traditional EM solutions, let us consider using the crowd to verify predicted matches. We first describe the typical workflow of traditional EM solutions, then explain the role of crowdsourcing. For illustrative purposes, consider matching products from the two tables A and B shown in Figure 1.2. There are three tuples in table A and two tuples in table B. Thus, there are a total of six tuple pairs to be matched.

In practice, there could be hundreds of thousands of tuples in each table, so billions of tuple pairs to match. Matching so many pairs is very expensive or highly impractical. Hence, developers often perform *blocking* (Chapter 4, Christen [35]), a step to reduce the number of pairs to be

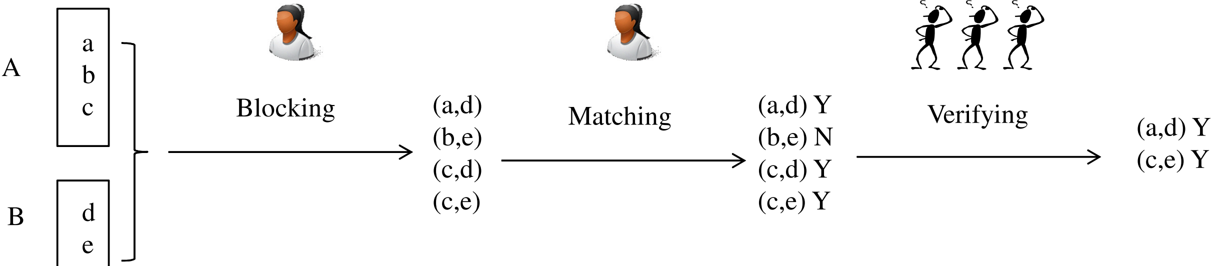


Figure 1.2: Verifying predicted matches using crowdsourcing.

matched. A commonly used blocking solution is to have a developer write and apply rules (called *blocking rules*) to remove as many obviously non-matched tuple pairs from $A \times B$ as possible. For instance, in the matching scenario in Figure 1.2, a developer applies the blocking rule "if the prices of two products differ by at least \$50, then they do not match" and removes two of the six pairs to be matched.

After blocking, the next step in a typical EM workflow is to build and apply a matcher (e.g., using hand-crafted rules, machine learning, or clustering) to predict a Yes/No label for each of the surviving pairs. In the scenario in Figure 1.2, a developer builds a rule-based matcher to match the surviving pairs. This matcher predicts the following three matching tuple pairs: (a,d), (c,d), and (c,e).

Traditional EM solutions stop at this point and return the predicted matching pairs. In practice, these predicted matches are far from perfect, and thus, often include falsely matched pairs. To improve the accuracy of these predicted matches, recent work [41, 86, 87] proposes to use the crowd to verify the predicted labels for these pairs, e.g., by asking workers from Amazon Mechanical Turk [2] (a popular crowdsourcing platform) to label the predicted matches, then taking the majority vote. Applying this approach to the product matching task in Figure 1.2, each of the three predicted matching pairs is sent to three workers from Amazon Mechanical Turk for labeling. After taking the majority vote only two pairs (a, d) and (c, e) are predicted to match. These are then returned as the final set of predicted matches. □

While recent work clearly demonstrates the promise of crowdsourced EM, it suffers from a major limitation: it crowdsources only parts of the EM workflow, thus *requiring a developer* who

knows how to code and match pairs to execute the remaining parts. For example, several recent solutions require a developer to write heuristic rules to reduce the number of candidate pairs to be matched, then train and apply a matcher to the remaining pairs to predict matches. They use the crowd only at the end, to verify the predicted matches (as discussed in Example 1.2.1). Evidently, the developer must know how to code (e.g., to write heuristic rules in Perl/Java) and match entities (e.g., create training data and select learning model). This need for a developer limits the applicability of current solutions in two important ways:

1. Current solutions do not scale to the growing EM need at enterprises and crowdsourcing startups. Many enterprises (e.g., eBay, Microsoft, Amazon, Walmart) routinely need to solve tens to hundreds of EM tasks, and this need is growing rapidly. It is not possible to crowdsource all these tasks if crowdsourcing each requires the involvement of a developer (even when sharing developers across tasks). To address this problem, enterprises often ask crowdsourcing startups (e.g., CrowdFlower) to solve the tasks on their behalf. But again, if each task requires a developer, then it is difficult for a startup, with a limited staff, to handle hundreds of EM tasks coming in from multiple enterprises.

2. Current solutions cannot help ordinary users (i.e., the "masses") leverage crowdsourcing to match entities. For example, suppose a journalist wants to match two long lists of political donors, and can pay up to a modest amount, say $500, to the crowd on Amazon's Mechanical Turk (AMT). He or she typically does not know how to code, thus cannot act as a developer and use current solutions. He or she cannot ask a crowdsourcing startup to help either. The startup would need to engage a developer, and $500 is not enough to offset the developer's cost. The same problem would arise for domain scientists, small business workers, end users, and other "data enthusiasts" [56].

1.3 Our Solution: Hands-Off Crowdsourcing for Entity Matching

To address the above limitations, in this dissertation we introduce the notion of *hands-off crowdsourcing (HOC)*. We then describe Corleone, our HOC solution for EM (named after Don Corleone, the fictional Godfather figure [74] who managed the mob in a hands-off fashion).

1.3.1 Hands-Off Crowdsourcing (HOC)

Our first contribution is to introduce the notion of hands-off crowdsourcing. Hands-off crowdsourcing, as the name suggests, crowdsources the *entire* workflow of a task, thus requiring no developers. HOC can be a next logical direction for EM and crowdsourcing research, moving from no-, to partial-, to complete crowdsourcing for EM. By requiring no developers, HOC can scale up EM at enterprises and crowdsourcing startups. For example, given a HOC system for entity matching, enterprises with hundreds of matching tasks can use such a system for each of the tasks without requiring a developer to execute any part of the EM workflow.

HOC can also open up crowdsourcing for the masses. Returning to the example of the journalist wanting to match two lists of donors, he or she can just upload the lists to a HOC Web site, and specify how much he or she is willing to pay. The Web site will use the crowd to execute a HOC-based EM workflow, then return the matches. Developing crowdsourcing solutions for the masses (rather than for enterprises) has received little attention, despite its potential to magnify many times the impact of crowdsourcing. HOC can significantly advance this direction.

1.3.2 Corleone: A HOC System for Entity Matching

Our second contribution is to design, develop, and evaluate Corleone, a HOC system for entity matching. Corleone uses the crowd (no developers) in all four major steps of the EM workflow. We now briefly describe these steps and summarize how Corleone addresses the challenges encountered in each of the steps.

Blocking To Reduce Set of Candidate Pairs: Virtually any large-scale EM workflow starts with blocking, a step that uses heuristics to reduce the number of tuple pairs to be matched. This is

because the Cartesian product $A \times B$ of the tables A and B to be matched is often very large, e.g., 10 billion tuple pairs if $|A| = |B| = 100,000$. Matching so many pairs is very expensive or highly impractical. Hence many blocking solutions have been proposed (e.g., [40, 48]).

These solutions require a developer to execute this step (e.g., to write and apply blocking rules, create training data, build indexes, etc.). Our goal however is to completely crowdsource it. To do so, we must address the challenge of using the crowd to generate machine-readable blocking rules. Most ordinary crowd workers cannot write such rules. If they write in English, we cannot reliably convert these rules into machine-readable ones. If we ask them to select among a set of rules, we often can only work with relatively simple rules and it is difficult to construct sophisticated ones.

To solve this challenge, Corleone takes a relatively small sample S from $A \times B$; applies crowdsourced active learning, in which the crowd labels a small set of informative pairs in S, to learn a matcher (a random forest [32]); extracts potential blocking rules from the matcher; uses the crowd again to evaluate the quality of these rules; then retains only the best ones.

Training & Applying a Matcher: The next step builds and applies a matcher to match the candidate tuple pairs output by the blocking solution. Given the set of candidate tuple pairs C, Corleone builds a random forest matcher M, which applies crowdsourcing to learn to match tuple pairs in C. Our goal is to maximize the matching accuracy, while minimizing the crowdsourcing cost. To do this, we use active learning [80]. Specifically, we train an initial matcher M, use it to select a small set of informative examples from C, ask the crowd to label the examples, use them to improve M, and so on.

A key challenge is deciding when to stop training M. Excessive training wastes money, and yet surprisingly can actually *decrease*, rather than increase the matcher's accuracy. To address this, Corleone uses a "confidence"-based stopping solution that monitors the confidence of the matcher and stops training when the confidence has peaked, indicating that the accuracy of the matcher has stopped improving.

Estimating Matching Accuracy: After applying the matcher, users often want to estimate the matching accuracy, i.e., the precision and recall[1] of the matcher. This step is vital in real-world EM (e.g., the estimated accuracy helps the user decide whether to continue the EM process). Surprisingly, very little work has addressed this problem, and as we show in Section 6.1, this work breaks down when the data is highly skewed, i.e., having very few matches (a common situation). Our goal is to overcome the limitations of current work, and use the crowd to estimate accuracy in a principled fashion.

To achieve this goal, Corleone incrementally samples from the set of candidate tuple pairs. If it detects data skew, i.e., too few matching tuple pairs, it performs reduction (i.e., using rules to eliminate obvious non-matching tuple pairs) to increase the positive density, then samples again. This continues until it has managed to estimate precision and recall of the matcher within a given margin of error. Corleone does not use any developer. Rather, it uses the crowd to label examples in the samples, and to generate reduction rules.

Iterating to Improve: In practice, entity matching is not a one-shot operation. Developers often estimate the matching result, then revise and match again. A common way to revise is to find tuple pairs that have proven difficult to match, then modify the current matcher, or build a new matcher specifically for these pairs. For example, when matching e-commerce products, a developer may find that the current matcher does reasonably well across all categories, except in Clothes, and so may build a new matcher specifically for Clothes products.

Corleone operates in a similar fashion. It estimates the matching accuracy (as discussed earlier), then stops if the accuracy does not improve (compared to the previous iteration). Otherwise, it revises and matches again. Specifically, it attempts to locate difficult-to-match pairs, then build a new matcher specifically for those. The challenge is how to locate difficult-to-match pairs. Our key idea is to identify *precise* positive and negative rules from the learned random forest, then remove all pairs covered by these rules (they are, in a sense, easy to match, because there already

[1] Precision is the fraction of predicted matching pairs that actually match. Recall is the fraction of actual matching pairs that are predicted to match.

exist rules that cover them). We treat the remaining examples as difficult to match, because the current forest does not contain any precise rule that covers them.

Engaging the Crowd: In all four major steps of the EM workflow, Corleone heavily uses crowdsourcing. In particular, it engages the crowd to label tuple pairs, to (a) supply training data for active learning (in blocking and matching), (b) supply labeled data for accuracy estimation, and (c) evaluate rule precision (in blocking, accuracy estimation, and locating difficult pairs).

Corleone currently uses Amazon Mechanical Turk (AMT) as the crowdsourcing platform to label the tuple pairs. To label each tuple pair (x, y), it poses a question "does x match y?" to workers on AMT and the workers can choose either *yes* or *no* as their answer. Crowdsourced answers, however, can often be *noisy*. Hence, Corleone uses majority voting to infer the label, i.e., getting multiple workers to answer the same question, and then taking the majority vote. The more workers we ask, the higher is the crowdsourcing cost. To minimize the crowdsourcing cost, Corleone exploits the fact that different steps in the EM workflow have different sensitivity to crowd noise. When using crowd-labeled data to train the matcher, crowd noise has only a marginal effect on the matcher's accuracy. Hence, for this step Corleone uses a simple majority voting scheme engaging a maximum of 3 workers per question. Crowd noise, however, can significantly affect accuracy estimation and rule evaluation. For these steps, Corleone uses a stronger voting scheme engaging up to 7 workers per question.

1.4 Contributions and Outline of the Dissertation

To summarize, in this dissertation I make the following contributions:

- I introduce the notion of hands-off crowdsourcing (HOC), which crowdsources the entire workflow of a task. I show that HOC is a next logical direction for crowdsourcing research, that it can scale up EM for enterprises and crowdsourcing startups, and that it can open up crowdsourcing for the masses.

- I describe Corleone, the first HOC solution for EM, to the best of my knowledge. I show how to crowdsource writing blocking rules, building a matcher using active learning, estimating matching accuracy given severe skew, and finding difficult-to-match tuple pairs.

- Finally, I present extensive experiments over three real-world data sets, showing that Corleone achieves comparable or significantly better accuracy (by as much as 19.8% F_1) than traditional solutions and published results, at a reasonable crowdsourcing cost of $9.20-$256.80 for the end-to-end EM workflow.

The rest of this dissertation is organized as follows. Chapter 2 describes the related work on crowdsourced entity matching. Chapter 3 introduces hands-off crowdsourcing and presents an overview of Corleone, our HOC solution for entity matching. Chapters 4-8 describe in detail how Corleone executes the different steps of the EM workflow, starting with blocking (Chapter 4), matching (Chapter 5), accuracy estimation (Chapter 6), iteration by locating difficult pairs (Chapter 7), and finally, engaging the crowd throughout the EM workflow (Chapter 8). Chapter 9 presents extensive empirical results for Corleone. Chapter 10 discusses the key design choices that went into developing Corleone and the opportunities for extending the system. Chapter 11 concludes the dissertation.

Chapter 2

Background and Related Work

In this chapter, we review work from the areas most relevant to the problem of crowdsourced entity matching. First, we survey entity matching systems (Section 2.1). Next, we review crowdsourced EM solutions (Section 2.2). Finally, we look at the larger context of crowdsourcing systems for data management problems, and platforms that help developers build crowdsourcing systems (Section 2.3).

2.1 Entity Matching

Entity matching has received extensive attention over the past few decades from researchers in the database, statistics, and AI communities (see Christen [35]). People have studied a variety of settings for entity matching, such as deduplicating records in a table [61, 77], matching two tables [25, 49], and collective matching [27, 44]. In this dissertation we consider the commonly encountered setting of matching two tables, i.e., finding all tuple pairs $(a \in A, b \in B)$ from two relational tables A and B that refer to the same real-world entity.

There is a wide variety of entity matching systems from industry as well as academia (see Chapter 10 by Christen [35] for a detailed overview). The majority of these systems belong to one of the following categories:

- Systems that provide entity matching tools as part of a bigger toolkit that is aimed at the data quality problem at enterprises, e.g., IBM InfoSphere [5], Informatica Data Quality [6], and Oracle Enterprise Data Quality [11].

- General-purpose domain-independent systems for entity matching, e.g., research and open-source systems such as Dedoop [61] and FEBRL [34], and commercial systems such as LinkageWiz [8] and Match2Lists [9].

- Special-purpose entity matching systems, e.g., Link Plus, a tool for duplicate detection in a cancer registry database [7] and D-Dupe, a tool for deduplication in social networks [31].

- Libraries for specific parts of the entity matching workflow, e.g., libraries of comparison functions such as SimMetric [17] and SecondString [16].

The systems most relevant to our work are general-purpose domain-independent entity matching systems. In particular, we survey research and open-source systems such as DuDe [45], FEBRL [34], Dedoop [61], and TAILOR [47]. These systems can be seen as collections of tools to execute different steps in the entity matching workflow. Specifically, they feature one or more of the following tools (the specific techniques available vary across systems):

1. data pre-processing tools, e.g., to support different input data formats such as CSV and XML;

2. library of comparison functions for different types of attributes such as string, numeric, date, and location;

3. methods to efficiently execute the blocking algorithm, such as indexing and canopy clustering (as discussed in Section 2.2.1);

4. methods for matching such as SVMs, decision trees, and hierarchical clustering; and

5. a GUI/Web interface to specify the EM workflow, visualize the matching process, and interact with the EM system.

There are certainly some differences across these systems. For instance, DuDe [45] does not have a GUI interface. Dedoop [61] is one of the few systems to support distributed computation over Hadoop. Table 2.1 compares these systems based on the supported features.

Supported Features	General-purpose EM Systems			
	TAILOR	DuDe	FEBRL	DeDoop
Web-based/Graphical UI	✓	✗	✓	✓
Blocking methods	✓	✓	✓	✓
Matching methods	✓	✓	✓	✓
Accuracy estimation	✗	✗	✗	✗
Methods for iteration	✗	✗	✗	✗
Support for distributed EM	✗	✗	✗	✓
Crowdsourcing support	✗	✗	✗	✗
Hands-off	✗	✗	✗	✗

Table 2.1: Comparison of general-purpose entity matching systems.

One feature common across all of these systems is that they need a developer, e.g., to write the blocking rules, to specify which methods to use to speed up blocking, to select the features to use for training, to select the matching algorithm, and to specify the end-to-end workflow. Thus, none of these systems is hands-off like Corleone.

2.2 Crowdsourced Entity Matching

Recently, crowdsourced EM has received increasing attention (e.g., [41, 81, 86, 87, 89, 90]). Most of the recent solutions can be categorized into three groups: (i) using the crowd to verify predicted matches [41, 86, 87], (ii) finding the best questions to ask the crowd [89], and (iii) finding the best user interface to pose such questions [66, 90].

Wang et al. [86, 87] and Demartini et al. [41] use the crowd to verify the matches predicted by traditional systems. Thus, the developer is needed to build solutions for blocking, matching, and accuracy estimation as in traditional EM systems, and the crowd is used only in the final step to verify the predicted labels. In particular, Wang et al. [86] focus on minimizing the cost of verification by effectively splitting the pairs into clusters. In their follow-up work [87], they further improve upon this solution by leveraging transitive relations among the pairs. Demartini et al. [41] use a probabilistic reasoning framework to decide which pairs need human verification.

Whang et al. [89] assume a limited budget for crowdsourced entity matching and propose a probabilistic framework to optimally use this budget to pose questions to the crowd. However, this

again assumes that some matcher has already assigned similarity scores to the pairs, and focuses on using the crowd to improve the accuracy. Whang et al. [90] evaluate the impact of different interfaces for matching two tuples on the accuracy of crowdsourced labels.

These works demonstrate that crowdsourcing is highly promising for entity matching. However, they suffer from a major limitation: they crowdsource only parts of the EM workflow, requiring a developer to execute the remaining parts. In contrast, Corleone crowdsources the entire EM workflow, thus requiring no developers. To compare and contrast Corleone with existing crowdsourced EM solutions, in the rest of this section we review existing solutions for the key steps in EM workflow: blocking (Section 2.2.1), matching (Section 2.2.2), and accuracy estimation (Section 2.2.3).

2.2.1 Blocking

Virtually any large-scale EM workflow starts with blocking, a step that uses heuristics to reduce the number of tuple pairs to be matched. This is because the Cartesian product $A \times B$ of the tables A and B to be matched, is often very large, e.g., 10 billion tuple pairs if $|A| = |B| = 100,000$. Matching so many pairs is very expensive or highly impractical. Hence many blocking solutions have been proposed [29, 36, 40, 54, 57, 68, 70].

The vast majority of existing works focus on efficiently executing the blocking solution [38, 54, 57, 62, 68] while requiring a developer to manually define heuristics for blocking (e.g., defining blocking keys [62] or sorting keys [57], and writing rules [68, 38]). There are some highly promising works that use supervised learning to automatically learn a blocking solution, most notably [29, 39, 40, 70]. These solutions, however, still require a developer, e.g., to sample and label training data, and to select appropriate features.

To summarize, as far as we know, the existing solutions for blocking do not employ crowdsourcing, and require a developer (e.g., to write and apply rules, create training data, build indexes, etc.). In contrast, Corleone completely crowdsources this step.

2.2.2 Matching

After blocking, the next step builds and applies a matcher to match the surviving pairs, i.e., classify each pair as matching or non-matching (e.g., using hand-crafted rules or machine learning models). Several techniques have been employed for matching ([48], Chapter 6 by Christen [35]), including rules [37, 51, 57], clustering [71], supervised learning [30, 47], active learning [77], probabilistic models [49, 76], and collective matching [27, 44].

Here the works closest to ours are those that use active learning [25, 26, 72, 77]. Arasu et al. [25] and Bellare et al. [26] use active learning to train a classifier that maximizes recall given a threshold on minimal precision. Sarawagi et al. [77] train a decision tree for matching using active learning with a committee-based approach for sampling [80]. Mozafari et al. [72] examine different active learning algorithms for crowdsourcing database problems, with a focus on understanding the trade-offs between different strategies for active sampling.

These works, however, either do not use crowdsourcing (requiring a developer to label training data) (e.g., [25, 26, 77]), or use crowdsourcing [72] but do not consider how to effectively handle noisy crowd input and to terminate the active learning process. In contrast, Corleone considers both of these problems and uses crowdsourcing with no developer in the loop.

2.2.3 Accuracy Estimation

In a typical EM workflow, the next step after matching is to estimate the matching accuracy (e.g., as precision and recall). This is vital in real-world EM (e.g., so that the user can decide whether to continue the EM process), but surprisingly has received very little attention in EM research. To the best of our knowledge, there is no prior work in entity matching literature that studies this problem in depth.

Estimating the precision and recall of a matcher is a common form of evaluating a classifier, a well-studied problem in machine learning. However, there is relatively little work on low-cost construction of a test set for highly imbalanced datasets. Here the most relevant work is by Katariya et al. [60] and Sawade et al.[78]. Katariya et al. [60] use a continuously refined stratified sampling strategy to estimate the accuracy of a classifier. However, their solution can not be used to estimate

recall, which is often necessary for EM. Sawade et al. [78] consider the problem of constructing the optimal labeled set for evaluating a classifier given the size of the sample. In contrast, we consider the different problem of constructing a minimal labeled set, given a maximum allowable error bound. Additionally, neither Katariya et al. [60], nor Sawade et al. [78] use the crowd for accuracy estimation.

Subsequent steps in the EM process involve "zooming in" on difficult-to-match pairs, revising the matcher, then matching again, and iterating until we can not improve the accuracy any further. While very common in industrial EM, these steps have received little or no attention in EM research. This concept of iteratively focusing on difficult-to-match examples and learning a new matcher to match those is similar to the idea of boosting in machine learning [79]. Corleone shows how this iterative process can be executed rigorously using only the crowd and no developer.

2.3 Crowdsourcing Systems

Crowdsourcing, as per Jeff Howe (one of the first people to coin that term) [58], refers to "the act of a company or institution taking a function once performed by employees and outsourcing it to an undefined (and generally large) network of people in the form of an open call". Crowdsourcing is a recent term, but it is not a new phenomenon, e.g., the Oxford English Dictionary (OED), first published in full in 1928, was a product of crowdsourcing[1]. However, it is only since the rise of the World Wide Web that crowdsourcing has become a highly potent tool with a wide range of applications. As a result, we have seen an explosion of crowdsourcing systems and applications in the last decade, e.g., Wikipedia [20], reCAPTCHA [13], ESP game [84], VizWiz [28], CrowdSearch [93], etc. There is a wide variety of crowdsourcing systems on the World Wide Web (see [43] for a recent survey). In this section, we only focus on work most relevant to this dissertation.

Crowdsourcing has been applied to a variety of data management problems, entity matching being just one of them. We first review recent work on crowdsourcing data management problems

[1]In 1858, an open call was made for volunteers to contribute words in the English language along with their documented usage. The editors received more than six million submissions over a period of seventy years. These became an integral part of the OED, which was first published in full in 1928.

(Section 2.3.1). Next, we describe the services offered by popular crowdsourcing platforms for building crowdsourcing systems (Section 2.3.2).

2.3.1 Crowsourcing for Data Management

In recent years, the database community has shown significant interest in leveraging crowdsourcing to perform database operations difficult to compute otherwise, e.g., filling in the missing contact information in a table of computer science professors at UW-Madison, or ranking rows in a table of movie actors based on their attractiveness.

Some of the early work proposed extensions to RDBMSs to support crowdsourced operations (CrowdDB [50], Deco [73], and Qurk [67]). This early work focuses on fundamental challenges involved in building a crowdsourced database system, such as extensions to the relational data model, extensions to the SQL language, new operators for crowdsourced operations, and opportunities for optimizing crowdsourced queries.

Besides this work on building crowdsourced RDBMSs, there are many other works focusing on crowdsourcing solutions for specific data management problems. For example, Marcus et al. [66] crowdsource "fuzzy" joins (e.g., joining a table containing celebrity names with another table containing celebrity pictures). McCann et al. [69] and Zhang et al. [94] apply crowdsourcing for schema matching by using the crowd to verify predicted matching attributes output by a machine algorithm. Trushkowsky et al. [82] focus on the challenges in collecting missing data when answering a selection query in a crowdsourced database system. Amsterdamer et al. [23] use the crowd for mining interesting association rules.

2.3.2 Platforms for Building a Crowdsourcing System for EM

There are some fundamental challenges faced by anyone building a crowdsourcing system, such as identifying the part of the workflow to crowdsource, recruiting workers, dividing the crowdsourced task among workers, and managing the workers. To enable developers to quickly build crowdsourcing systems, many crowdsourcing platforms have been built (e.g., Amazon Mechanical Turk (AMT) [2], CrowdFlower [3], WorkFusion [21], and SamaSource [15]).

A crowdsourcing platform such as AMT allows requesters (e.g., companies, research labs, and ordinary users) to post tasks that can be completed by anyone with access to the Internet (i.e., the crowd). The crowd workers, in turn, get paid by the requesters (typical payment is a few cents) on successfully completing the task. These are typically simple tasks, such as Yes/No questions and short surveys, that can be completed by most Web users without any special training.

Crowdsourcing systems such as Corleone often need to solicit the services of crowdsourcing platforms to access and recruit workers. Hence, we review here the services offered by two of the most popular commercial crowdsourcing platforms: Amazon Mechanical Turk (AMT) and CrowdFlower.

Amazon Mechanical Turk (AMT): Amazon Mechanical Turk (started in 2005) is one of the first and most popular crowdsourcing platforms. Requesters post tasks (each unit of task is called a HIT which stands for "Human Intelligence Task") on AMT either using a Web-based interface or programmatically using an API. Workers access the AMT Web site, browse the available tasks, and start working on any of the tasks that they are eligible for. When finished, workers submit their work to AMT. Requesters then access and review the submitted work, and they can either decide to approve the work and pay the worker in full, or reject the work and pay nothing.

The services offered by AMT to requesters can be broadly categorized into two types: (i) basic services available for any general task, and (ii) specialized solutions for popular tasks.

1. **Basic services:** These are the basic tools that AMT offers to help requesters post HITs and manage the crowd. AMT provides an API to programmatically post HITs and receive answers. When posting a HIT, requesters have some basic control over which workers are allowed to work on their HIT[2]. Specifically, when posting a task on AMT, a requester can specify the qualifications that a worker must have to be able to work on that task (e.g., a worker must be from the US, must have completed at least 100 HITs on AMT, and at least 95% of them have been approved). Apart from worker control, an important part of posting a HIT is designing the UI that workers will use to complete the task. To help requesters

[2] One of the biggest challenges in using crowdsourcing platforms is getting good quality answers from workers. Controlling who is allowed to work on your HIT is one way to control the quality of answers you get.

in this regard, AMT provides interface templates for the commonly posted tasks such as categorization, data collection, and image moderation.

While these basic services are very useful, to fully accomplish a task using AMT a requester still needs to perform several other tasks, such as deciding which part of the workflow to crowdsource, how to divide the work to be crowdsourced into individual HITs, how many workers to assign each HIT to, and how much to pay each worker for each HIT.

2. **Specialized solutions:** In addition to the above basic tools, AMT offers specialized solutions for two of the most popular tasks that requesters perform using AMT: a categorization application for the task of assigning categories to each item in a collection, and a sentiment application to gauge the sentiment (e.g., positive, neutral, or negative) for each item. These two applications have built-in solutions for some of the functions requesters need to perform when crowdsourcing the task, such as determining the price to pay per HIT, designing the interface for a HIT, and evaluating the workers. However, requesters still need to perform other tasks such as design the end-to-end workflow and decide which parts of the workflow to crowdsource.

CrowdFlower: CrowdFlower [3] is an emerging crowdsourcing platform that has gained significant popularity in recent years. The services offered to requesters by CrowdFlower can be categorized into two modes:

1. **Self-service mode:** In this mode requesters gain access to the CrowdFlower platform, including access to a variety of tools for posting and managing the tasks. These tools are similar to the ones provided by AMT. However, CrowdFlower provides additional functionalities, e.g., a declarative language to design the interface for tasks, fine-grained control over which workers can work on a task, and additional tools for quality control, such as worker training, automated work evaluation, and worker monitoring. The rest of the work must be done by the requesters themselves, e.g., determining which part of the workflow to crowdsource, designing the crowdsourcing workflow, how to divide the crowdsourced task into smaller tasks, etc.

2. **Managed-service mode:** In this mode, a dedicated team of professionals manages the entire crowdsourcing workflow for the requester. The requester only needs to provide the data and task requirements to this team.

To summarize, when using one of the existing platforms to develop a crowdsourced solution for a task, say entity matching, we either have to choose the do-it-yourself option (e.g., the basic services offered by AMT or the self-service mode on CrowdFlower) or the managed-service option. However, in either case a developer is required to implement the workflow. In the do-it-yourself option we would either need to hire a developer or act as one ourselves, while in the managed-service option the crowdsourcing startup provides the developer. Thus, none of the popular crowdsourcing platforms provides a hands-off option.

Chapter 3

Proposed Solution

Crowdsourcing is highly promising for entity matching, as demonstrated by recent crowdsourced EM solutions. However, as already explained (Section 1.2), the need for a developer severely limits these solutions. Specifically, they do not scale to the growing EM need at enterprises and crowdsourcing startups, and they can not handle crowdsourcing for the masses. To address these limitations, we propose the notion of hands-off crowdsourcing (HOC). We then describe Corleone, our proposed HOC solution for entity matching.

3.1 Hands-Off Crowdsourcing

Given a problem P supplied by a user U, we say a crowdsourced solution to P is *hands-off* if it uses no developers, only a crowd of ordinary workers (such as those on AMT). It can ask user U to do a little initial setup work, but this should require no special skills (e.g., coding) and should be doable by any ordinary worker. For example, Corleone only requires a user U to supply

1. two tables A and B to be matched,
2. a short textual instruction to the crowd on what it means for two tuples to match (e.g., "these records describe products sold in a department store; they should match if they represent the same product"), and
3. four examples, two positive and two negative (i.e., pairs that match and do not match, respectively), to illustrate the instruction. EM tasks posted on AMT commonly come with such instruction and examples.

Corleone then uses the crowd to match A and B (sending them information in (2) and (3) to explain what user U means by a match), then returns the matches. As such, Corleone is a hands-off solution. The following real-world example illustrates Corleone and contrasts it with current EM solutions.

Example 3.1.1. Consider a retailer that must match tens of millions of products between the online division and the brick-and-mortar division (these divisions often obtain products from different sets of suppliers). The products fall into 500+ categories: toy, electronics, homes, etc. To obtain high matching accuracy, the retailer must consider matching products in each category separately, thus effectively having 500 EM problems, one per category.

Today, solving each of these EM problems (with or without crowdsourcing) requires extensive developer's involvement, e.g., to write blocking rules, to create training data for a learning-based matcher, to estimate the matching accuracy, and to revise the matcher, among others. Thus current solutions are not hands-off. One may argue that once created and trained, a solution to an EM problem, say for toys, is hands-off in that it can be automatically applied to match future toy products, without using a developer. But this ignores the initial non-negligible developer effort put into creating and training the solution (thus violating our definition). Furthermore, this solution cannot be transferred to other categories (e.g., electronics). As a result, extensive developer effort is still required for all 500+ categories, a highly impractical approach.

In contrast, using Corleone, per category the user only has to provide Items 1-3, as described above (i.e., the two tables to be matched; the matching instruction which is the same across categories; and the four illustrating examples which virtually any crowdsourcing solutions would have to provide for the crowd). Corleone then uses the crowd to execute all steps of the EM workflow. As such, it is hands-off in that it does not use any developer when solving an EM problem, thus potentially scaling to all 500+ categories. □

We believe HOC is a general notion that can apply to many problem types, such as entity matching, schema matching, information extraction, etc. In this dissertation we will focus on entity matching. Realizing HOC poses serious challenges, in large part because it has been quite hard to figure out how to make the crowd do certain things. For example, how can the crowd write blocking rules

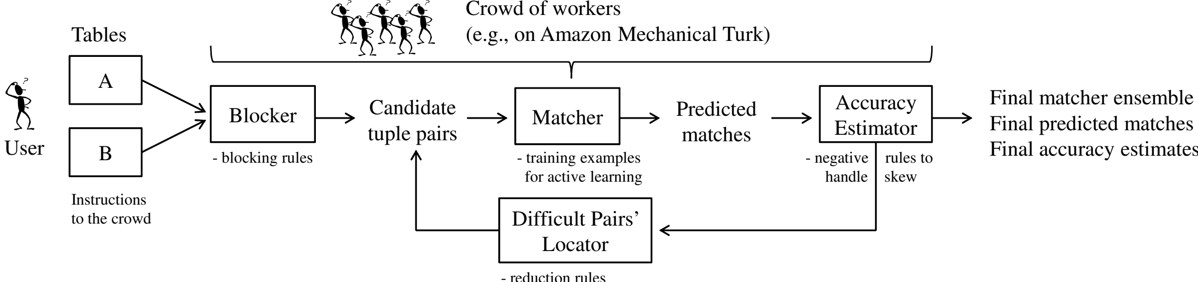


Figure 3.1: The Corleone architecture.

(e.g., "if prices differ by at least $50, then two products do not match")? We need rules in machine-readable format (so that we can apply them). However, most ordinary crowd workers cannot write such rules, and if they write in English, we cannot reliably convert them into machine-readable ones. Finally, if we ask them to select among a set of rules, we often can only work with relatively simple rules and it is hard to construct sophisticated ones. Corleone addresses such challenges, and provides an HOC solution for entity matching.

3.2 The Corleone Solution

We now present an overview of Corleone, our proposed hands-off crowdsourcing system for entity matching.

3.2.1 Input to Corleone

Figure 3.1 shows the input supplied by the user to Corleone. We describe below specific details about the input data format.

1. The two tables A and B to be matched are provided as disk-resident files in CSV format. The tables A and B must have identical schema. Each table should have a column named `id` which serves as a primary key uniquely identifying each record in the table.

2. The instructions to the crowd are provided by the user in a text file (see Section A.2 for sample instructions).

3. The four examples (two matching and two non-matching pairs) to illustrate the instructions are provided as records in a CSV file. Each record has three attributes: `(id1, id2, label)`. Here `id1` and `id2` are the id values of a pair of tuples from tables A and B, respectively. `label` is a boolean-valued attribute that takes the value `true` if the pair of tuples match, otherwise it is `false`.

Given the above input for a matching task, Corleone executes a HOC-based entity matching workflow for that task and returns the predicted matches to the user.

3.2.2 Corleone's Workflow

Figure 3.1 shows the Corleone architecture, which consists of four main modules: Blocker, Matcher, Accuracy Estimator, and Difficult Pairs' Locator. The Blocker generates and applies blocking rules to $A \times B$ to remove obviously non-matched pairs. The Matcher uses active learning to train a random forest [32], then applies it to the surviving pairs to predict matches. The Accuracy Estimator computes the accuracy of the Matcher. The Difficult Pairs' Locator finds pairs that the current Matcher has matched incorrectly. The Matcher then learns a better random forest to match these pairs, and so on, until the estimated matching accuracy no longer improves.

As described, Corleone is distinguished in three important ways. (1) All four modules do not use any developers, but heavily use crowdsourcing. (2) In a sense, the modules use crowdsourcing not just to label the data, as existing work has done, but also to "create" complex rules (blocking rules for the Blocker, negative rules for the Estimator, and reduction rules for the Locator, see Sections 4-7). And (3) Corleone can be run in many different ways. The default is to run multiple iterations until the estimated accuracy no longer improves. But the user may also decide to just run until a budget (e.g., $300) has been exhausted, or to run just one iteration, or just the Blocker and Matcher, etc.

In the rest of the dissertation we describe Corleone in detail. Chapters 4-7 describe the Blocker, Matcher, Estimator, and Locator, respectively. We defer all discussions on how Corleone engages the crowd to Chapter 8.

Chapter 4

Blocking To Reduce the Set of Candidate Pairs

In this chapter we describe the Blocker, the component of Corleone that takes the user-supplied input (two tables A & B, instructions to the crowd, and four labeled pairs), and outputs the set of candidate matching pairs. The Blocker generates and applies the blocking rules to identify the candidate matching pairs among all possible matching pairs ($A \times B$).

Blocking is critical for large-scale EM. This is because the set of tuple pairs to be matched, which is the Cartesian product $A \times B$ of the tables A and B to be matched, is often very large. For example, if $|A| = |B| = 100,000$ then we need to match 10 billion tuple pairs. Prior work requires a developer to perform blocking. Our goal however is to completely crowdsource it.

To do so, we must address the challenge of using the crowd to generate precise machine-readable blocking rules. Existing work on crowdsourced entity matching asks the crowd only to label tuple pairs as positive or negative (Section 2.2). Our work addresses this problem. Our key idea is to ask the crowd to label tuple pairs as before, and use these answers to learn precise machine-readable rules for blocking.

Here is a brief overview of how the Blocker works. First, the Blocker determines whether there is a need to perform blocking. If $A \times B$ is so small that developing a matching solution over the entire Cartesian product is very inexpensive, then clearly blocking is not needed. If blocking is required, then the Blocker proceeds to the next step of generating candidate blocking rules using the crowd. The Blocker then uses the crowd again to evaluate the quality of these rules, and then retains only the best ones. Finally, the Blocker identifies a subset of the surviving rules that can eliminate "sufficient" number of pairs, and applies them to identify the set of candidate matching pairs. We now describe these steps in detail (see Algorithms 4.1-4.4 for the pseudo code).

Algorithm 4.1 Pseudo-code for the Blocker

Input: Tables A and B ($|A| < |B|$) , Set of user-provided labeled pairs L

Output: Candidate tuple pairs C

1: /* 1. Decide whether to do blocking */
2: **if** $|A \times B| \leq t_B$ **then**
3: **return** $A \times B$ // no need to block
4: **end if**
5: /* 2. Generate candidate blocking rules (Algorithm 4.2)*/
6: $\{X, S\}$ = $generateCandidateRules(A, B, L)$
7: /* 3. Evaluate top rules using the crowd (Algorithm 4.3)*/
8: V = $evaluateTopRules(X, S)$
9: /* 4. Apply precise blocking rules (Algorithm 4.4)*/
10: C = $applyPreciseRules(V, A, B, L)$
11: **return** C

4.1 Deciding Whether to Do Blocking

Let A and B be the two tables to be matched. Intuitively, we want to do blocking only if $A \times B$ is too large to be processed efficiently by subsequent steps. Currently we deem this is the case if $A \times B$ exceeds a threshold t_B, set to be the largest number such that if after blocking we have t_B tuple pairs, then we can fit the feature vectors of all these pairs in main memory (we discuss feature vectors below), thus minimizing I/O costs for subsequent steps. The goal of blocking is then to generate and apply blocking rules to remove as many obviously non-matched pairs from $A \times B$ as possible.

Before we go on, a brief remark on minimizing I/O costs. One may wonder if the consideration of reducing I/O costs makes sense, given that Corleone already uses crowdsourcing, which can take a long time. We believe it is still important to minimize system time (including I/O time) for three reasons. First, we use active learning, so the sooner the system finishes an iteration, the faster

it can go back to the crowd, thereby minimizing total time. Second, depending on the situation, the crowd time may or may not dominate the system time. For example, if we pay one penny for a relatively complex question, it may take hours or days before we get three workers to answer the question. But if we pay eight pennies (a rate that many crowdsourcing companies pay), it may take just a few minutes, in which case the crowd time may just be a fraction of the system time. Finally, when working at the scale of millions of tuples per table, system time can be quite significant, taking hours or days (on par or more than crowd time). Section 10.1.1 has a more detailed explanation for why we use a threshold for blocking, and how we set the threshold.

4.2 Generating Candidate Blocking Rules

When it is determined that blocking is needed, the Blocker generates many machine-readable rules as candidate rules for blocking. To do that, it takes a relatively small sample S from $A \times B$; applies crowdsourced active learning, in which the crowd labels a small set of informative pairs in S, to learn a random forest matcher; and then extracts negative rules[1] from the matcher as the candidate blocking rules.

4.2.1 Taking a Small Sample

We want to learn a random forest F, then extract candidate blocking rules from it. Learning F directly over $A \times B$ however is impractical because this set is too large. Hence we will sample a far smaller set S from $A \times B$, then learn F over S. Naively, we can randomly sample tuples from A and B, then take their Cartesian product to be S. Random tuples from A and B however are unlikely to match. So we may get no or very few positive pairs in S, rendering learning ineffective.

To address this problem, we sample as follows (lines 1-3 in Algorithm 4.2). Let A be the smaller table. We randomly sample $t_B/|A|$ tuples from B (assuming that t_B is much larger than $|A|$, please see below), then take S to be the Cartesian product between this set of tuples and A. Note that we also add the four examples (two positive, two negative) supplied by the user to S. This way, S has roughly t_B pairs, thus having the largest possible size that still fits in memory, to

[1] rules that predict non-matching pairs

Algorithm 4.2 Pseudo-code for $generateCandidateRules$

Input: Tables A and B ($|A| < |B|$) , Set of user-provided labeled pairs L

Output: Set of candidate blocking rules X, Sample S of tuple pairs from $A \times B$ used to generate the candidate rules

1: /* 1. Take sample S from $A \times B$ */
2: $B_s \leftarrow$ Scan B and take a simple random sample of $t_B/|A|$ records from B
3: $S \leftarrow (A \times B_s) \cup L$
4: /* 2. Apply crowdsourced active learning to S */
5: $T \leftarrow L$, $M \leftarrow$ Train initial random forest on T
6: **repeat**
7: $E \leftarrow$ Select q most informative unlabeled examples from S
8: Label all the pairs in E using the crowd
9: $T \leftarrow T \cup E$, $M \leftarrow$ Train a random forest on T
10: **until** M has stopped improving
11: $X \leftarrow$ Generate all the negative rules from M
12: **return** X, S

ensure efficient learning[2]. Furthermore, if B has a reasonable number of tuples that have matches in A, and if these tuples are distributed uniformly in B, then the above strategy ensures that S has a reasonable number of positive pairs.

We now discuss the assumption that t_B is much larger than $|A|$. We make this assumption because we consider the current targets of Corleone to be matching tables of up to 1 million tuples each, frequently less (e.g., in the range of 50K-300K tuples per table). The vast majority of EM problems that we have seen in industry fall into this range, and we are not aware of any current publication or software that successfully matches tables of 1 million tuples each, even with using

[2]For learning, we are interested in the feature vectors of the pairs in S, rather than the attribute values of the tuples. Hence, our algorithm does not materialize S, and directly computes the feature vectors of the pairs in S as the pairs are getting sampled.

Hadoop (unless they do very aggressive blocking). For this target range, t_B, set to be 3M to 5M, is much larger than $|A|$, the smaller table of the two.

That said, our eventual goal is to scale Corleone to tables of millions of tuples. Hence, we are exploring better sampling strategies. In Section 10.2.1 we report some preliminary results in this direction.

Our experiments show that the current naive sampling method works well on the current data sets (i.e., we successfully learned good blocking rules from the samples). Briefly, they worked because there are often *many* good negative rules (i.e., rules that find non-matched pairs) with good coverage (i.e., can remove many pairs). Even a naive sampling strategy can give the blocker enough data to find some of these good negative rules, and the blocker just needs to find *some* in order to do a good job at blocking.

4.2.2 Applying Crowdsourced Active Learning

In the next step, we convert each tuple pair in S into a feature vector, using features taken from a pre-supplied feature library (see Section 5.1 for details). Example features include edit distance, Jaccard measure, Jaro-Winkler, TF/IDF, Monge-Elkan, etc. [17, 16]. Then we apply crowdsourced active learning to S to learn a random forest F (lines 4-10 in Algorithm 4.2). Briefly, we use the two positive and two negative examples supplied by the user to build an initial forest F, use F to find informative examples in S, ask the crowd to label them, then use the labeled examples to improve F, and so on. A random forest is a set of decision trees [77]. We use decision trees because blocking rules can be naturally extracted from them, as we will see, and we use active learning to minimize the number of examples that the crowd must label. We defer describing this learning process in detail to Chapter 5.

4.2.3 Extracting Candidate Blocking Rules

The active learning process outputs a random forest F, which is a set of decision trees, as mentioned earlier. Figures 4.1.a-b show a toy forest with just two trees (in our experiments each forest has 10 trees, and the trees have 8-655 leaves). Here, the first tree states that two books match

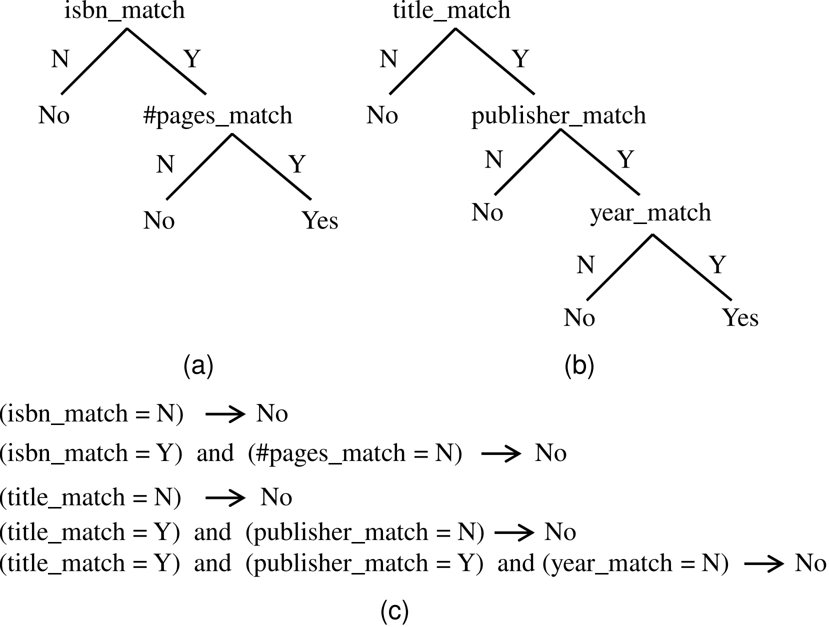


Figure 4.1: (a)-(b) A toy random forest consisting of two decision trees, and (c) negative rules extracted from the forest.

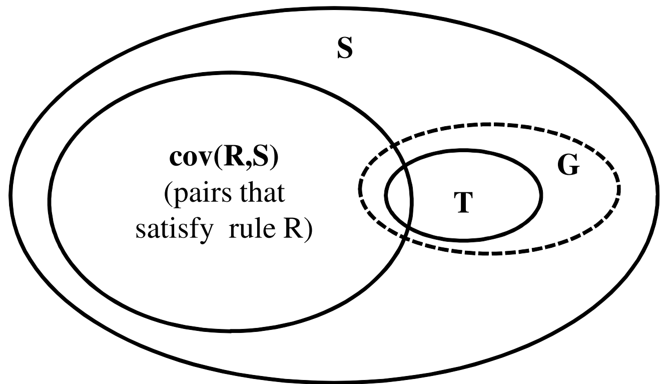
only if the ISBNs match and the numbers of pages match. Observe that the leftmost branch of this tree forms a *decision rule*, shown as the first rule in Figure 4.1.c. This rule states that if the ISBNs do not match, then the two books do *not* match. It is therefore a *negative rule*, and can clearly serve as a blocking rule because it identifies book pairs that do not match. In general, given a forest F, we can extract *all* tree branches that lead from a root to a "No" leaf to form negative rules. Figure 4.1.c show all five negative rules extracted from the forest in Figures 4.1.a-b. We return all negative rules as the set of candidate blocking rules.

4.3 Evaluating Rules Using the Crowd

The candidate blocking rules can vary widely in precision. So we must evaluate and discard the imprecise ones. Ideally, we want to evaluate *all* rules, using the crowd. This however can be very expensive money-wise (we have to pay the crowd), given the large number of rules (e.g., up

to 8,943 in our experiments). Hence, we first pick only k most promising rules, we then evaluate them using the crowd (current $k = 20$).

4.3.1 Selecting Blocking Rules



$$\text{prec}(R, S) = \frac{|\,\text{cov}(R, S) - G\,|}{|\,\text{cov}(R, S)\,|}$$

S: sample from A X B
G: actual matching pairs in S
T : pairs in S labeled as matches by the crowd during active learning

Figure 4.2: Coverage and precision of rule R over S.

To pick k rules among the candidate blocking rules, we compute two metrics for each rule, and then rank the rules based on these two metrics. Specifically, for each rule R, we compute the coverage of R over sample S, $cov(R, S)$, to be the set of examples in S for which R predicts "no.". We define the precision of R over S, $prec(R, S)$, to be the number of examples in $cov(R, S)$ that are indeed negative divided by $|cov(R, S)|$. As Figure 4.2 shows, $prec(R, S) = |cov(R, S) - G|/|cov(R, S)|$. Of course, we cannot compute $prec(R, S)$ because we do not know the true labels of examples in $cov(R, S)$, and hence, we do not know the set G. However, we can compute an upper bound on $prec(R, S)$. Let T be the set of examples in S that (a) were selected during the active learning process in Step 3, Section 4.2, and (b) have been labeled by the crowd as positive. Then clearly $prec(R, S) \leq |cov(R, S) - T|/|cov(R, S)|$, since $T \subseteq G$ as can be seen in Figure 4.2. We then select the rules in decreasing order of the upper bound on $prec(R, S)$, breaking tie using $cov(R, S)$, until we have selected k rules, or have run out of rules. Intuitively, we prefer rules with higher precision and coverage, all else being equal.

Note that ideally, we would want to compare the rules based on the precision and coverage of each rule over entire $A \times B$. However, operating on all of $A \times B$ would be extremely expensive

and would defeat the very purpose of blocking. Hence, we compute these metrics over sample S instead.

4.3.2 Evaluating the Selected Rules Using the Crowd

Let V be the set of selected rules. We now use the crowd to estimate the precision of rules in V, then keep only highly precise rules. Specifically, for each rule $R \in V$, we execute the following loop:

1. We randomly select b examples in $cov(R, S)$, use the crowd to label each example as matched / not matched, then add the labeled examples to a set X (initially set to empty).

2. Let $|cov(R, S)| = m$, $|X| = n$, and n_- be the number of examples in X that are labeled negative (i.e., not matched) by the crowd. Then we can estimate the precision of rule R over S as $P = n_-/n$, with an error margin $\epsilon = Z_{1-\delta/2}\sqrt{\left(\frac{P(1-P)}{n}\right)\left(\frac{m-n}{m-1}\right)}$ [88]. This means that the true precision of R over S is in the range $[P - \epsilon, P + \epsilon]$ with a δ confidence (currently set to 0.95).

3. If $P \geq P_{min}$ and $\epsilon \leq \epsilon_{max}$ (which are pre-specified thresholds), then we stop and add R to the set of precise rules. If (a) $(P + \epsilon) < P_{min}$, or (b) $\epsilon \leq \epsilon_{max}$ and $P < P_{min}$, then we stop and drop R (note that in case (b) with continued evaluation P may still exceed P_{min}, but we judge the continued evaluation to be costly, and hence drop R). Otherwise return to Step 1.

This incremental sampling in batches of size b eventually stops. At this point we either have determined that R is a precise rule, or we have dropped R as it may not be good enough. Currently we set $b = 20, P_{min} = 0.95, \epsilon_{max} = 0.05$. Asking the crowd to label an example is rather involved, and will be discussed in Section 8.

It is important to emphasize that in the above evaluation we do not take just a sample of size b. Instead, we go through multiple iterations, in each we take a sample of size b. Put another way, we start with b pairs. If we find that these pairs do not allow us to compute rule precisions with sufficient accuracy, then we take another b pairs, and add those to the previous ones, and so on.

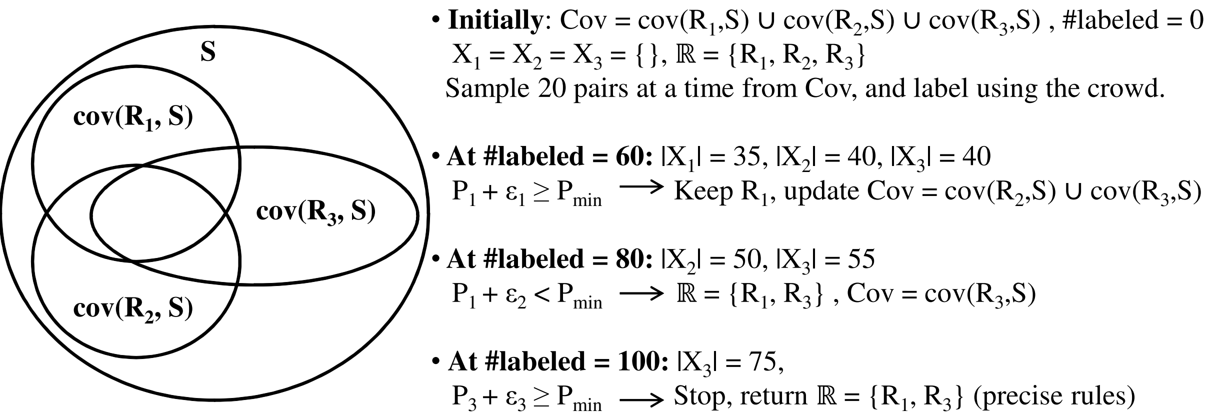


Figure 4.3: Example illustrating joint evaluation of rules.

The above procedure evaluates each rule in V in isolation. We can do better by evaluating all rules in V jointly, to reuse examples across rules. Specifically, let $R_1, \ldots, R_q$ be the rules in V. Then we start by randomly selecting b examples from the *union* of the coverages of $R_1, \ldots, R_q$, use the crowd to label them, then add them to $X_1, \ldots, X_q$, the set of labeled examples that we maintain for the $R_1, \ldots, R_q$, respectively. For example, if a selected example is in the coverage of only R_1 and R_2, then we add it to X_1 and X_2. Next, we use $X_1, \ldots, X_q$ to estimate the precision of the rules, as detailed in Step 2, and then to keep or drop rules, as detailed in Step 3. If we keep or drop a rule, we remove it from the union, and sample only from the union of the remaining rules. Lines 7-24 in Algorithm 4.3 show the pseudocode for joint evaluation of rules.

Example 4.3.1. To illustrate this joint evaluation algorithm, suppose that we have a set of three rules $V = R_1, R_2, R_3$ that need to be evaluated. Figure 4.3 shows how the joint evaluation algorithm proceeds. Initially, all the rules are "active", i.e., need to be evaluated, and thus, the set that we sample from, Cov, is set to the union of coverages of all the three rules, i.e., $cov(R_1, S) \cup cov(R_2, S) \cup cov(R_3, S)$.

At this point, we have not sampled any pairs thus, the sample X_i for each rule is empty and the number of labeled pairs (#labeled) = 0. Now we start sampling 20 pairs at a time from Cov, get labels for these pairs, and update X_i, as well as the estimated P_i and ϵ_i for each rule R_i. Suppose that after sampling and labeling a total of 60 pairs, we have 35 pairs in X_1 (the sample for R_1),

40 of the pairs in X_2, and 40 pairs in X_3. At this point, suppose that R_1 satisfies the condition for a "precise" rule (line 18 in Algorithm 4.3). Clearly, we do not need to sample to evaluate R_1 anymore. Thus, we update the set Cov to only include coverages of R_2 and R_3, i.e., Cov = $cov(R_2, S) \cup cov(R_3, S)$.

We now continue sampling from this updated set Cov. Suppose that after sampling and labeling 20 more pairs (thus, #labeled = 80) we have 50 pairs in X_2 and 55 pairs in X_3. At this point, suppose that we find that R_2 is a "bad" rule (i.e., satisfies the condition on line 20 in Algorithm 4.3). In that case, we drop R_2, and only continue the evaluation of R_3. Thus, we have $V = \{R_1, R_3\}$, and Cov = $cov(R_3, S)$. Now on sampling 20 more pairs from Cov, we will have 75 pairs in X_3. At this point suppose R_3 satisfies the condition for a "precise" rule. Clearly we are done evaluating, and we return $\{R_1, R_3\}$ as the set of precise rules. □

Correctness of the Joint Evaluation Algorithm: The straightforward solution to evaluate the rules estimates the precision of each rule in isolation. In this solution, to estimate the precision of each rule R_i we draw a uniform random sample X_i (without replacement) from the set $cov(R_i, S)$. In the joint evaluation algorithm, we sample uniformly without replacement from the set Cov, which is the union of the rules yet-to-be-evaluated, instead of drawing in isolation from the coverages of each rule. The set Cov keeps changing as the rules are getting evaluated. However, it is easy to see that for any individual rule R_i, if X_i is the sample maintained for R_i during the joint evaluation process then X_i is a uniform random sample (without replacement) drawn from $cov(R_i, S)$.

Here is a brief explanation. Note that $cov(R_i, S)$ is part of Cov until R_i gets evaluated, i.e., $cov(R_i, S) \subseteq Cov$. Thus, every time we draw a pair from Cov following the uniform distribution, every pair $p \in cov(R_i, S)$ has an equal probability of getting picked. Since we are drawing without replacement, once a pair $p \in cov(R_i, S)$ gets picked that pair would never get picked again. Thus, the sample X_i, which is the set of all pairs p drawn during joint evaluation such that $p \in cov(R_i, S)$, is a uniform random sample drawn without replacement from $cov(R_i, S)$.

Efficiency Challenge: Efficiently implementing Algorithm 4.3 is an important challenge. We implement two key optimizations here to reduce the execution cost. First, we avoid recomputation of the coverage $cov(R, S)$ for each of the top k rules when jointly evaluating them. Note that in line 7 in Algorithm 4.3 we recompute the union of the coverages of all the yet-to-be-evaluated rules each time we sample the next batch of pairs. Recomputing the coverage and the union each time would be highly inefficient. Hence, we compute the coverage of each rule only once and store it in memory.

Second, to minimize the cost of recomputing the union set, we compute the union Cov of coverages of all the rules once before we begin sampling. We then incrementally maintain this union set. Specifically, whenever we are finished evaluating any rule R among the k rules, we remove $cov(R, S)$ from Cov. To ensure fast computation of Cov we use bit vector representation to store Cov, as well as the coverage $cov(R, S)$ of each rule R.

4.4 Applying Blocking Rules

Let Y be the set of rules in V that have survived crowd-based evaluation. We now consider which subset of rules $\mathcal{R}$ in Y should be applied as blocking rules to $A \times B$.

This is highly non-trivial. Let $Z(\mathcal{R})$ be the set of pairs obtained after applying the subset of rules $\mathcal{R}$ to $A \times B$. If $|Z(\mathcal{R})|$ falls below threshold t_B (recall that our goal is to try to reduce $A \times B$ to t_B pairs, if possible), then among all subsets of rules that satisfy this condition, we will want to select the one whose set $Z(\mathcal{R})$ is the *largest*. This is because we want to reduce the number of pairs to be matched to t_B (at which point we can fit all the pairs into main memory), but do not want to go too much below that, because then we run the risk of eliminating many true positive pairs. On the other hand, if no subset of rules from Y can reduce $A \times B$ to below t_B, then we will want to select the subset that does the most reduction, because we want to minimize the number of pairs to be matched.

One may wonder why we do not want to apply all blocking rules. For example, if a rule can reduce the Cartesian product by 80%, why not apply it? The answer is that blocking rules are often not perfect. That is, they often remove not just negative pairs, but some positive pairs too.

Unfortunately, a priori there is no good way to evaluate how good a blocking rule is (our precision calculations give only *estimates* of the true precisions, of course). So if one's goal is to keep as many positive pairs as one can (because recall is important), then one may choose not to apply a blocking rule even though it can filter out a large number of negative pairs.

For the above reason, we have found that in practice people typically initiate the blocking process only if the original number of pairs is too large to be processed in a reasonable amount of time, and then they do blocking only to the extent that the resulting data set can now be processed practically. They do not apply all blocking rules that they can write, for fear of accidentally removing too many positive pairs.

Returning to our current setting, we cannot execute all subsets of Y on $A \times B$, in order to select the optimal subset. So we use a greedy solution. First, we rank all rules in Y based on the precision $prec(R, S)$, coverage $cov(R, S)$, and the tuple cost. The tuple cost is the cost of applying rule R to a tuple, primarily the cost of computing the features mentioned in R. We can compute this because we know the cost of computing each feature in Step 3, Section 4.2. Next, we select the first rule, apply it to reduce S to S', re-estimate the precision, coverage, and tuple cost of all remaining rules on S', re-rank them, select the second rule, and so on. We repeat until the set of selected rules when applied to S has reduced it to a set of size no more than $|S| * (t_B/|A \times B|)$, or we have selected all rules. We then apply the set of selected rules to $A \times B$ (using a Hadoop cluster), to obtain a smaller set of tuple pairs to be matched. This set is passed to the Matcher, which we describe in Chapter 5.

We now describe two optimizations we implement to improve the performance the algorithm described so far for applying the blocking rules.

1. An Additional Metric To Rank the Rules: In Algorithm 4.4, we use $prec(R, S)$ as our estimate for the precision of rule R, when ranking the rules (line 4). This estimate is very useful to distinguish between rules that differ significantly in their precision, e.g., if R_1 has 20% higher precision than R_2 over entire $A \times B$, then it is highly likely that the estimate $prec(R_1, S)$ will be greater than $prec(R_2, S)$. However, if R_1 and R_2 differ in precision by a small margin, say 2%, then often our estimates $prec(R_1, S)$ and $prec(R_2, S)$ will be exactly equal.

To address this problem, we compute an additional metric to distinguish between any two rules when their estimated precision is the same. To compute this metric we use the random forest F from which the blocking rules were extracted (Section 4.2.2). We apply F to match the pairs in $cov(R, S)$ and count the number of predicted matches d. These d pairs are predicted as non-matches by rule R, while the random forest F predicts them to match. If the random forest predicts a given pair to match, it implies that majority of the trees in the forest predict that the given pair of tuples match. Intuitively, if a blocking rule R "disagrees" a lot with the forest F, i.e., if d is very high for rule R, then it may indicate that rule R makes errors on a lot of actual positives by predicting them as negatives. We call this the *disagreement score* for rule R. If two rules R_1 and R_2 have exactly same estimated precision, i.e., if $prec(R_1, S) = prec(R_2, S)$, then we compare the disagreement scores d_1 and d_2, and prefer the rule with lower disagreement score.

2. Special Case of Preferring A Rule With Lower Coverage: When ranking the rules in Algorithm 4.4 (line 4), if two rules R_1 and R_2 have the same precision, then we compare their coverages, $|cov(R_1, S)|$ and $|cov(R_2, S)|$, and prefer the rule with higher coverage. However, this is not always the case. We are only interested in reducing the Cartesian product $A \times B$ to below t_B number of pairs, but not much below that. Thus, if two rules R_1 and R_2 both have sufficient coverage so that applying them will reduce the candidate set to t_B or below, then we actually prefer the rule with *lower* coverage.

Algorithm 4.3 Pseudo-code for $evaluateTopRules$

Input: Set of candidate blocking rules X, Set of sample tuple pairs S used to generate candidate rules

Output: Set of precise blocking rules V

1: /* 1. Select top k blocking rules */
2: **for all** $R \in X$ **do**
3: Compute $|cov(R, S)|$, $P_{ub} = |cov(R, S) - T|/|cov(R, S)|$
4: **end for**
5: Sort the rules in X in decreasing order of P_{ub}, $|cov(R, S)|$
6: V = Top k rules in X_{-}
7: /* 2. Jointly evaluate the rules in V */
8: $V_a \leftarrow V, \forall R \in V, X(R) = \emptyset$
9: **while** $A \neq \emptyset$ **do**
10: $Cov \leftarrow \bigcup_{R \in V_a} cov(R, S)$
11: $Q \leftarrow$ Sample b pairs from Cov (without replacement)
12: Label Q using the crowd
13: **for all** $p \in Q$ **do**
14: $\forall R \in V_a$, if $p \in cov(R, S)$, then $X(R) = X(R) \cup \{p\}$
15: **end for**
16: **for all** $R \in V_a$ **do**
17: Estimate precision P and error ϵ
18: **if** $P \geq P_{min}$ **and** $\epsilon \leq \epsilon_{max}$ **then**
19: $V_a \leftarrow V_a - \{R\}$
20: **else if** $(P + \epsilon) < P_{min}$ **or** ($\epsilon \leq \epsilon_{max}$ **and** $P < P_{min}$) **then**
21: $V_a \leftarrow V_a - \{R\}, V \leftarrow V - \{R\}$
22: **end if**
23: **end for**
24: **end while**
25: **return** V

Algorithm 4.4 Pseudo-code for $applyPreciseRules$

Input: Set of precise blocking rules V, Tables A and B, Sample pairs S

Output: Set of candidate matching pairs C

1: /* 1. Select the subset of blocking rules J */

2: $J \leftarrow \emptyset$, $s_0 \leftarrow |S|$, $Y \leftarrow V$

3: **while** $Y \neq \emptyset$ **and** $|S| > (t_B/|A \times B|) \cdot s_0$ **do**

4:     Sort the rules in Y in decreasing order of v (v = $< prec(R, S), |cov(R, S)|, cost(R) >$)

5:     Pick the topmost rule R and remove it from Y

6:     $Y = Y \cup \{R\}$, $S \leftarrow S - cov(R, S)$

7: **end while**

8: /* 2. Apply the blocking rules in J */

9: $C \leftarrow \emptyset$

10: **for all** $(a, b) \in A \times B$ **do**

11:     **while** $(R \leftarrow getNext(Y)) \neq null$ **do**

12:         if (a, b) satisfies R, continue to the next pair.

13:     **end while**

14:     $C \leftarrow C \cup \{(a, b)\}$ // if (a, b) survives all the rules in Y

15: **end for**

16: **return** C

Chapter 5

Training and Applying a Matcher

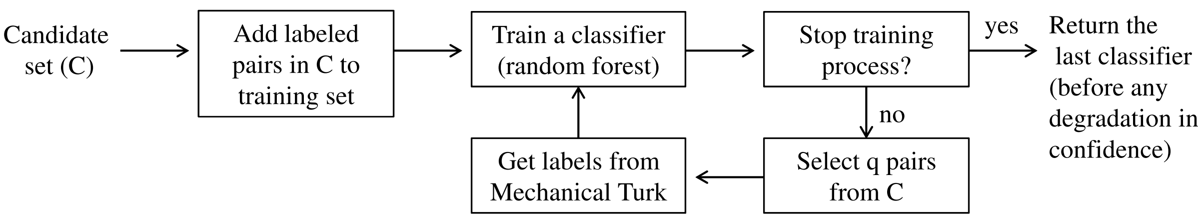


Figure 5.1: Crowdsourced active learning in Corleone.

Let C be the set of tuple pairs output by the Blocker. We now describe Matcher M, which applies crowdsourcing to learn to match tuple pairs in C. We want to maximize the matching accuracy, while minimizing the crowdsourcing cost. To do this, we use active learning. Figure 5.1 shows the overall workflow for learning the matcher. Specifically, we train an initial matcher M, use it to select a small set of informative examples from C, ask the crowd to label the examples, use them to improve M, and so on. A key challenge is deciding when to stop training M. Excessive training wastes money, and yet surprisingly can actually *decrease*, rather than increase the matcher's accuracy. We now describe matcher M and our solution to the above challenge.

5.1 Training the Initial Matcher

We begin by converting each example (i.e., tuple pairs) in C into a feature vector, i.e., a vector of numeric feature values, for learning purposes. This is done at the end of the blocking step: any surviving example is immediately converted into a feature vector, using all features that are appropriate (e.g., no TF/IDF features for numeric attributes) and available in our feature library.

The feature library contains functions that compare one or more attributes of a given pair of tuples, and return a numeric or categorical value as output. We currently have the following features available in the feature library: Jaro-Winkler (JW), Levenshtein (L), Q-gram (Q), Jaccard (J), Smith-Waterman-Gotoh (SWG), Monge-Elkan (ME), Soft TF-IDF (STF-IDF), TF-IDF over Tri-grams (Tri TF-IDF), Equals (E), and Relative difference (RD). RD is a feature function we defined to compare numeric values[1]. All the other features are standard functions used for measuring similarity of strings [17, 16].

Given the features in the library, we determine the features to be computed for the given matching task as follows. For each attribute z of the input tables A and B, we identify the type of z (currently, we compute features for three attribute types: short string, long string, and numeric). We then identify all the features in the library that are applicable for comparing attributes of this type and add them to the set of features to be computed, e.g., if z is a short string, then we have 3 applicable features in the library for comparing $A.z$ and $B.z$: JW, L, and E. Thus, we will add JW($A.z$, $B.z$), L($A.z$, $B.z$), and E($A.z$, $B.z$) to the set of features to be computed. This is repeated for each attribute to get the final set of features to compute. Note that this set of features to compute depends only on the feature library, and the types of the attributes. Thus, we determine the set of features only once at the beginning of the Blocker when the files containing the tables are parsed. Example 5.1.1 further illustrates this feature vector computation process.

In the rest of this dissertation, we use the terms example, pair, and feature vector interchangeably, when there is no ambiguity.

Example 5.1.1. Suppose we have two tables A and B to be matched, each containing book tuples (Figure 5.2.a). Each tuple contains 6 attributes: id, isbn, title, publisher, year, and #pages (number of pages). Suppose that blocking is triggered in this case. Figure 5.2.b shows the candidate set C output by the blocking step. C contains a small number of potential matching pairs of tuples from

[1] We define the function as follows: $RD(v_1, v_2) = |v_1 - v_2|/|v1 + \epsilon|$, where v_1 and v_2 are numeric values to be compared and $epsilon = 0.01$

A

id	isbn	title	publisher	year	#pages
1	981	Cosmos	Random House	1980	550
2	937	Twilight	Little, Brown	2005	320
...	-----	-------	---------	-------	-------
...	-------	-------	---------	-------	-------

B

id	isbn	title	publisher	year	#pages
1	937	Twilight	Yen Press	2010	120
2	981	Cosmos	Random	1980	550
...	-----	-------	---------	-------	
...	-------	-------	---------	-------	

(a)

Candidate set C

aid	bid	aisbn	bisbn	atitle	btitle	...
1	2	981	981	Cosmos	Cosmos	
2	1	937	937	Twilight	Twilight	
,,,	-----	-------	--------	----	-------	...
...	-----	-------	-------	---.	...	....
...	-----	-------	-------	---.	...	....
...	-----	-------	-------	---.	...	....

(b)

Feature vectors for pairs in C

aid	bid	isbn_match	title_match	publisher_match	year_match	#pages_match
1	2	1	1	0	1	1
2	1	1	1	0	0	0
,,,	-----	-------	--------	----	-------	...
...	-----	-------	-------	---.	...	...
...	-----	-------	-------	---.	...	...
...	-----	-------	-------	---.	...	...

(c)

Figure 5.2: Example: candidate set and feature vectors.

A and B. Suppose that Corleone selects only the Equals(E) feature function to compare each attribute, so a total of 5 features to compute for each pair[2]: isbn_match, title_match, publisher_match, year_match, and #pages_match. Each of these features are binary, i.e., if the two values are exactly equal the feature evaluates to 1, if not then 0. Corleone computes all the 5 features for each pair in C to get the feature vectors, as shown in Figure 5.2.c. This table of feature vectors is then used in the learning step.

After computing the feature vectors for all the examples in C, we use all labeled examples available at that point (supplied by the user or labeled by the crowd) to train an initial classifier that when given an example (x, y) will predict if x matches y. Currently we use an ensemble-of-decision-trees approach called *random forest* [32]. In this approach, we train k decision trees independently, each on a random portion (typically set at 60%) of the original training data. When training a tree,

[2]In practice, Corleone will pick many more features for each attribute. We pick only Equals (E) for illustrative purposes.

at each tree node we randomly select m features from the full set of features $f_1, \ldots, f_n$, then use the best feature among the m selected to split the remaining training examples. The values k and m are currently set to be the default 10 and $log(n) + 1$, respectively. Once trained, applying a random forest classifier means applying the k decision trees, then taking the majority vote.

To illustrate, going back to the book examples (Example 5.1.1), Figure 4.1 shows a sample random forest that could be learned from this dataset.

5.2 Consuming the Next Batch of Examples

Once matcher M has trained a classifier, M evaluates the classifier to decide whether further training is necessary (see Section 5.3). Suppose M has decided yes, then it must select new examples for labeling.

In the simplest case, M can select just a single example (as current active learning approaches often do). A crowd however often refuses to label just one example, judging it to be too much overhead for little money. Consequently, M selects q examples (currently set to 20) for the crowd to label. Intuitively, M wants these examples to be "most informative". A common way to measure the "informativeness" of an example e is to measure the disagreement of the component classifiers using entropy [80]:

$$entropy(e) = -[P_+(e) \times ln(P_+(e)) + P_-(e) \times ln(P_-(e))], \tag{5.1}$$

where $P_+(e)$ and $P_-(e)$ are the fractions of the decision trees in the random forest that label example e positive and negative, respectively. The higher the entropy, the stronger the disagreement, and the more informative the example is.

Thus, M selects the p examples (currently set to 100) with the highest entropy from set C (excluding those that have been selected in the previous iterations). Next, M selects q examples from these p examples, using weighted sampling, with the entropy values being the weights. This sampling step is necessary because M wants the q selected examples to be not just informative, but also diverse. The following example illustrates the weighted sampling step.

Example 5.2.1. To keep the example simple, suppose that p = 5 and q = 2. Suppose the top 5 pairs with the highest entropy are as follows: p_1 (0.6), p_2 (0.6), p_3 (0.4), p_4 (0.4), p_5 (0.4), where the number in parentheses shows the entropy for the pair. Now we randomly draw a total of 2 pairs, drawing one pair at a time from these 5 pairs. However, each pair does not have an equal chance of getting picked. The probability of picking a pair is proportional to its entropy. Thus, when drawing the first pair, the probability for picking p_1 will be 0.6/(0.6 + 0.6 + 0.4 + 0.4 + 0.4) = 1/4. Similarly, the probability of picking p_2 will be 1/4. However, for p_3, p_4, and p_5 the probability of being picked will be 1/6 each.

In the next step, M sends the q selected examples to the crowd to label (described in Section 8), adds the labeled examples to the current training data, then re-trains the classifier.

At this point one may wonder how expensive entropy computation is. We note that this computation requires just a linear scan over the pairs in the candidate set (i.e., those pairs surviving the blocking step). As such, its time (per iteration) grows proportional to the candidate set size, and has stayed in the range of seconds in our experiments. For example, on the Product data set, on average the candidate set's size is 200K, entropy computation time is about 2.4 seconds per iteration, and total entropy computation time is 2 minutes (for 50 iterations, see the experiment section). Of course, on large data sets (and thus larger candidate sets), this time will grow. Fortunately, this step is trivially parallelizable.

5.3 Deciding When to Stop

Recall that matcher M trains in iteration, in each of which it pays the crowd to label q training examples. We must decide then when to stop the training. Interestingly, more iterations of training not only cost more, as expected, but can actually *decrease* rather than increase M's accuracy. This happens because after M has reached peak accuracy, more training, even with perfectly labeled examples, does not supply any more informative examples, and can mislead M instead. This problem became especially acute in crowdsourcing, where crowd-supplied labels can often be incorrect, thereby misleading the matcher even more.

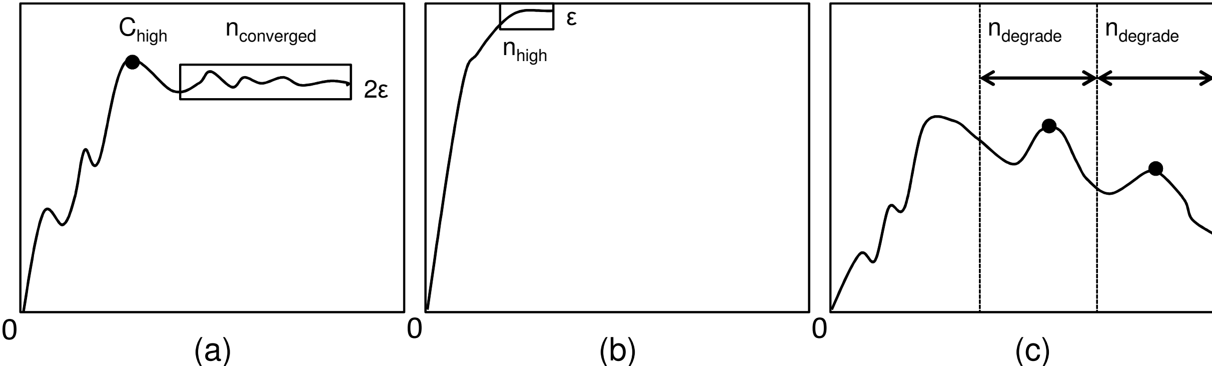


Figure 5.3: Typical confidence patterns that we can exploit for stopping.

To address this problem, we develop a solution that tells M when to stop training. Our solution defines the "confidence" of M as the degree to which the component decision trees agree with one another when labeling. We then monitor M and stop it when its confidence has peaked, indicating that there are no or few informative examples left to learn from.

Specifically, let $conf(e) = 1 - entropy(e)$, where $entropy(e)$ is computed as in Equation 5.1, be the *confidence* of M over an example e. The smaller the entropy, the more decision trees of M agree with one another when labeling e, and so the more confident M is that it has correctly labeled e.

Before starting the active learning process, we set aside a small portion of C (currently set to be 3%), to be used as a monitoring set V. We monitor the confidence of M over V, defined as $conf(V) = \sum_{e \in V} conf(e)/|V|$. We expect that initially $conf(V)$ is low, reflecting the fact that M has not been trained sufficiently, so the decision trees still disagree a lot when labeling examples. As M is trained with more and more informative examples (see Section 5.2), the trees become more and more "robust", and disagree less and less. So $conf(V)$ will rise, i.e., M is becoming more and more confident in its labeling. Eventually there are no or few informative examples left to learn from, so the disagreement of the trees levels off. This means $conf(V)$ will also level off. At this point we stop the training of matcher M.

We now describe the precise stopping conditions, which, as it turned out, was quite tricky to establish. Ideally, once confidence $conf(V)$ has leveled off, it should stay level. In practice,

additional training examples may lead the matcher astray, thus reducing or increasing $conf(V)$. This is exacerbated in crowdsourcing, where the crowd-supplied labels may be wrong, leading the matcher even more astray, thus causing drastic “peaks” and “valleys” in the confidence line. This makes it difficult to sift through the “noise” to discern when the confidence appears to have peaked. We solve this problem as follows.

First, we run a smoothing window of size w over the confidence values recorded so far (one value per iteration), using average as the smoothing function. That is, we replace each value x with the average of the w values: $(w-1)/2$ values on the left of x, $(w-1)/2$ values on the right, and x itself. (Currently $w = 5$.) We then stop if we observe any of the following three patterns over the smoothed confidence values:

- **Converged confidence:** In this pattern the confidence values have stabilized and stayed within a 2ϵ interval (i.e., for all values v, $|v - v^*| \leq \epsilon$ for some v^*) over $n_{converged}$ iterations. We use $\epsilon = 0.01$ and $n_{converged} = 20$ in our experiments (these parameters and those described below are set using simulated crowds). Figure 5.3.a illustrates this case. When this happens, the confidence is likely to have converged, and unlikely to still go up or down. So we stop the training.

- **Near-absolute confidence:** This pattern is a special case of the first pattern. In this pattern, the confidence is at least $1 - \epsilon$, for n_{high} consecutive iterations (see Figure 5.3.b). We currently use $n_{high} = 3$. When this pattern happens, confidence has reached a very high, near-absolute value, and has no more room to improve. So we can stop, not having to wait for the whole 20 iterations as in the case of the first pattern.

- **Degrading confidence:** This pattern captures the scenarios where the confidence has reached the peak, then degraded. In this pattern we consider two consecutive windows of size $n_{degrade}$, and find that the maximal value in the first window (i.e., the earlier one in time) is higher than that of the second window by more than ϵ (see Figure 5.3.b). We currently use $n_{degrade} = 15$. We have experimented with several variations of this pattern. For example, we considered comparing the average values of the two windows, or comparing the first

value, average value, and the last value of a (relatively long) window. We found however that the above pattern appears to be the best at accurately detecting degrading confidence after the peak.

Afterward, M selects the last classifier before degrading to match the tuple pairs in the input set C.

Chapter 6

Estimating Matching Accuracy

After applying matcher M, Corleone estimates M's accuracy. If this exceeds the best accuracy obtained so far, Corleone continues with another round of matching (see Section 7). Otherwise, it stops, returning the matches together with the estimated accuracy. This estimated accuracy is especially useful to the user, as it helps decide how good the crowdsourced matches are and how best to use them. We now describe how to estimate the matching accuracy.

6.1 Current Methods and Their Limitations

To motivate our method, we begin by describing current evaluation methods and their limitations. Suppose we have applied matcher M to a set of examples C. To estimate the accuracy of M, a common method is to take a random sample S from C, manually label S, then compute the precision $P = n_{tp}/n_{pp}$ and the recall $R = n_{tp}/n_{ap}$, where (a) n_{pp} is the number of predicted positives: those examples in S that are labeled positive (i.e., matched) by M; (b) n_{ap} is the number of actual positives: those examples in S that are manually labeled as positive; and (c) n_{tp} is the number of true positives: those examples in S that are both predicted positive and actual positive.

Let P^* and R^* be the precision and recall on the set C (computed in an analogous fashion, but over C, not over S). Since S is a random sample of C, we can report that with δ confidence, $P^* \in [P - \epsilon_p, P + \epsilon_p]$ and $R^* \in [R - \epsilon_r, R + \epsilon_r]$, where the error margins are defined as

$$\epsilon_p = Z_{1-\delta/2}\sqrt{\left(\frac{P(1-P)}{n_{pp}}\right)\left(\frac{n^*_{pp} - n_{pp}}{n^*_{pp} - 1}\right)}, \tag{6.1}$$

$$\epsilon_r = Z_{1-\delta/2}\sqrt{\left(\frac{R(1-R)}{n_{ap}}\right)\left(\frac{n^*_{ap}-n_{ap}}{n^*_{ap}-1}\right)}, \tag{6.2}$$

where n^*_{ap} and n^*_{pp} are the number of actual positives and predicted positives on C, respectively, and $Z_{1-\delta/2}$ is the $(1-\delta/2)$ percentile of the standard normal distribution [88].

As described, the above method has a major limitation: it often requires a very large sample S to ensure small error margins, and thus ensuring meaningful estimation ranges for P^* and R^*. For example, assuming $R^* = 0.8$, to obtain a reasonable error margin of, say $\epsilon_r = 0.025$, using Equation 6.2 we can show that $n_{ap} \geq 984$ (regardless of the value for n^*_{ap}). That is, S should contain at least 984 actual positive examples.

The example universe for EM however is often quite skewed, with the number of positive examples being just a small fraction of the total number of examples (e.g., 0.06%, 2.64%, and 0.56% for the three data sets in Section 9, even after blocking). A fraction of 2.64% means that S must contain at least 37,273 examples, in order to ensure at least 984 actual positive examples. Labeling 37,000+ examples however is often impractical, regardless of whether we use a developer or the crowd, thus making the above method inapplicable.

When finding too few positive examples, developers often apply heuristic rules that eliminate negative examples from C, thus attempting to "reduce" C into a smaller set C_1 with a far higher "density" of positives. They then randomly sample from C_1, in the hope of boosting n_{ap} and n_{pp}, thereby reducing the margins of error. This approach, while promising, is often carried out in an ad-hoc fashion. As far as we know, no strategy on how to do reduction systematically has been reported. In what follows, we show how to do this in a rigorous way, using crowdsourcing and negative rules extracted from the random forest.

6.2 Crowdsourced Estimation with Corleone

Our solution incrementally samples from C. If it detects data skew, i.e., too few positive examples, it performs reduction (i.e., using rules to eliminate certain negative examples from C) to increase the positive density, then samples again. This continues until it has managed to estimate P and R within a given margin of error ϵ_{max}. Our solution does not use any developer. Rather,

it uses the crowd to label examples in the samples, and to generate reduction rules, as described below.

6.2.1 Generating Candidate Reduction Rules

When applied to a set of examples (e.g., C), reduction rules eliminate negative examples, thus increasing the density of positive examples in the set. As such, they are conceptually the same as blocking rules in Section 4. Those rules cannot be used on C, however, because they are already applied to $A \times B$ to generate C.

Instead, we can generate candidate reduction rules exactly the way we generate blocking rules in Section 4, except for the following. First, in the blocking step in Section 4 we extract the rules from a random forest trained over a relatively small sample S. Here, we extract the rules from the random forest of matcher M, trained over the entire set C. Second, in the blocking step we select top k rules, evaluate them using the crowd, then keep only the precise rules. Here, we also select top k rules, but we do not yet evaluate them using the crowd (that will come later, if necessary). We return the selected rules as candidate reduction rules.

6.2.2 Repeating a Probe-Eval-Reduce Loop

We then perform the following online search algorithm to estimate the accuracy of matcher M over C:

1. *Enumerating our options:* To estimate the accuracy, we may execute no reduction rule at all, or just one rule, or two rules, and so on. Let $\mathcal{R} = \{R_1, \ldots, R_n\}$ be the set of candidate reduction rules. Then we have a total of 2^n possible options, each executing a subset of rules in $\mathcal{R}$.

2. *Estimating and selecting the lowest-cost option:* A priori we do not know which option is the best. Hence, we perform a limited sampling of C (the *probe* operation) to estimate the cost of each option (to be discussed below), then select the one with the lowest cost.

3. *Partially evaluating the selected option:* Without loss of generalization, suppose we have selected the option that executes rules $\mathcal{D} = \{R_1, \ldots, R_d\}$. Fully evaluating this option means (a) using the crowd to evaluate rules $R_1, \ldots, R_d$, exactly the way we evaluate blocking rules in Section 4.3 (the *eval* operation), (b) keeping only good, i.e., highly precise, rules, (c) executing these rules on C to reduce it, thereby increasing the positive density, then (d) sampling from the reduced C until we have managed to estimate P and R within the margin of error ϵ_{max}.

   Instead of fully evaluating the selected option, we do mid-execution optimization. Specifically, after executing (a)-(c), we do not do (d). Instead we return to Step 1 to re-enumerate our options. Note that now we have a reduced set C (because we have applied the good rules in $\mathcal{D}$), and also a reduced set $\mathcal{R}$ (because we have removed all rules in $\mathcal{D}$ from $\mathcal{R}$).

   The above strategy is akin to mid-query re-optimization in RDBMSs, where given a SQL query, we select a good execution plan, partially evaluate it, then use the newly gathered statistics to re-optimize to find a potentially better execution plan. Similarly, in our setting, once we have selected a plan, we perform a partial evaluation by executing Steps (a)-(c). At this point we may have gained more information, such as which rules are bad. So we skip Step (d), and return to Step 1 to see if we can find a potentially better plan. Eventually we do have to execute Step (d), but only after we have concluded that we cannot find any potentially better plan.

4. *Termination:* If we have not terminated earlier (e.g., in Step 2, after sampling of C, see below), then eventually we will select the option of using no rules (in the worst-case scenario this happens when we have applied all rules). If so, we sample until we have managed to estimate P and R within a margin of error ϵ_{max}. Algorithm 6.2 shows the pseudo-code for this estimation step that terminates the algorithm.

All that is left is to describe how we estimate the costs of the options in Step 2. Without loss of generalization, consider an option that executes rules $\mathcal{Q} = \{R_1, \ldots, R_q\}$. We estimate its cost to be (1) the cost of evaluating all rules in $\mathcal{Q}$, plus (2) the cost of sampling from the reduced set C

Algorithm 6.1 Pseudo-code for $probe$ operation

Input: Candidate set C, Matcher M, ϵ_{max}

Output: Density of actual positives d

1: Uniformly sample b examples from C, and label them using the crowd to create sample S.

2: $n_{ap} \leftarrow$ Number of actual positives in S

3: $n_{tp} \leftarrow$ Number of true positives in S

4: $n_{pp} \leftarrow$ Number of predicted positives in S

5: Compute P, R, ϵ_p, and ϵ_r (using equations 6.1)

6: if $\epsilon_p \leq \epsilon_{max}$ **and** $\epsilon_r \leq \epsilon_{max}$, then stop the estimation process.

7: **return** $d = \frac{n_{ap}}{|S|}$

after we have applied all rules in $\mathcal{Q}$ (note that we are making an optimistic assumption here that all rules in $\mathcal{Q}$ turn out to be good).

Currently we estimate the cost in (1) to be the sum of the costs of evaluating each individual rule. In turn, the cost of evaluating a rule is the number of examples that we would need to select from its coverage for the crowd to label, in order to estimate the precision to be within ϵ_{max} (see Section 4.3). We can estimate this number using the formulas for precision P and error margin ϵ given in Section 4.3.

Suppose after applying all rules in $\mathcal{Q}$, C is reduced to set C'. We estimate the cost in (2) to be the number of examples we need to sample from C' to guarantee margin of error ϵ_{max}. If we know the positive density d' of C', we estimate the above number. It is easy to prove that $d' = d * |C|/|C'|$, where d is the positive density of C (assuming that the rules are 100% precise).

To estimate d, we perform a "limited sampling", i.e., the *probe* operation, by sampling b examples from the set C (currently $b = 50$). Algorithm 6.1 shows the pseudo-code for the probe operation. We use the crowd to label these examples, then estimate d to be the fraction of examples being labeled positive by the crowd. (We note that in addition, we also use these labeled b examples to estimate $P, R, \epsilon_p, \epsilon_r$, as shown in Section 6.1, and immediately exit if ϵ_p and ϵ_r are already below ϵ_{max}.) We now present an example to illustrate the algorithm.

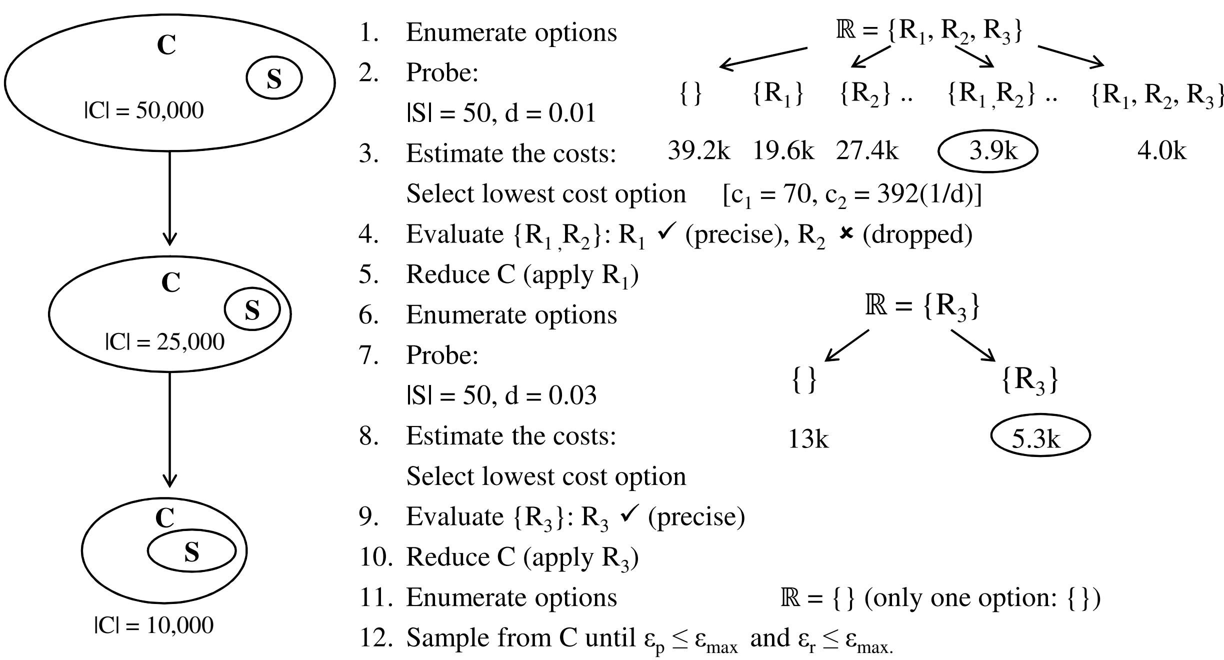


Figure 6.1: Example to illustrate the estimation process.

Example 6.2.1. Suppose that we have a candidate set C containing 50000 pairs, and we want to estimate the precision and recall of a given matcher M over the set C. Figure 6.1 shows a step-by-step execution of the crowdsourced estimation algorithm for this example. Suppose that generating top rules from M gives us a set $\mathcal{R}$ containing 3 rules $\mathcal{R} = \{R_1, R_2, R_3\}$. As the first step, Corleone enumerates all the options, i.e., lists all the subsets of $\mathcal{R}$. Next, it performs the *probe* operation, taking a sample S of size 50 to estimate the density of positives d. Suppose d is 0.01 (i.e. 1%).

It then estimates the expected cost (i.e. number of examples to be labeled) for each of the options listed, starting from $\{\}$, i.e., the option of applying zero rules to reduce, and sampling all the way from the current candidate set. As explained earlier, the estimated cost of an option $\mathcal{Q} = \{R_1, \ldots, R_q\}$ is $c(\mathcal{Q}) = c_r + c_s$, where c_r is the cost of evaluating all the rules in $\mathcal{Q}$, c_s is the cost of sampling from the reduced set after applying all the rules in $\mathcal{Q}$. Here $c_r = q \cdot c_1$ where q is the number of rules in $\mathcal{Q}$, and $c_s = c_2 \cdot (1/d')$ where c_2 is the number of positives we need in

the sample to guarantee an error margin below ϵ_{max}, and d' is the expected positive density in the reduced set we will obtain on applying all the rules in $\mathcal{Q}$.

To illustrate how the cost is computed, let us consider $\mathcal{Q} = \{\}$. In this case c_r = 0 as q = 0, and d' = d, since C is not reduced at all. Suppose, c_1 = 70 and c_2 = 392. Thus, we get: c({}) = 0 + 392(1/0.01), i.e. 39.2k.

Next, Corleone picks the option with the lowest cost, which is $\{R_1, R_2\}$ in this case, as shown in Figure 6.1. It then evaluates the selected rules. Suppose R_1 passes the test for precision, but R_2 fails. In this case, it applies only the rule R_1 to reduce C, and removes R_1 and R_2 from the set of rules $\mathcal{R}$. After the reduction, C contains 25000 pairs as shown in Figure 6.1. At this point, $\mathcal{R} = \{R_3\}$. Now Corleone again enumerates the options and performs the *probe* operation to estimate the cost of each option. We only have two possible options either to use no rules ({}) and sample all the way from current C, or $\{R_3\}$, i.e., to evaluate R_3 and if it is precise, apply it to reduce C. The second option has lower expected cost (5.3k as opposed to 13k), and thus, the option $\{R_3\}$ gets picked. Next, the algorithm evaluates R_3 to find that it is precise. It then applies R_3 to reduce C further. At this stage, we are left with a candidate set C of size 10000, and no more rules to reduce. Thus, the algorithm picks the default option {}, and samples from C until it has estimated both precision and recall, within ϵ_{max} error margin. This terminates the estimation step. $\square$

Correctness of the Estimates: In the crowdsourced estimation algorithm above, the final estimates for precision (lines 5 and 6 from Algorithm 6.2) and recall (line 16 from Algorithm 6.3) over the original candidate set C are computed in the sampling step at the end. However, the sample here is drawn only from the reduced candidate set C'. Since this sampling procedure is the same as described in 6.1, estimating precision and recall over C' is straightforward. Thus, to prove the correctness of the equations for precision and recall we simply need to show how to estimate the precision and recall over original candidate set C, using estimates over reduced set C'. This is exactly what we do next.

Proposition 1. *Let M be a given binary classifier. Let P and R be the precision and recall values of M on the candidate set of matching pairs C. Let P' and R' be the precision and recall values of M on the reduced set C' obtained at the end of crowdsourced estimation with* Corleone. *With the assumption that all the rules used for reducing C are perfect, i.e., they do not eliminate any positive pairs, and the labels provided by the crowd are perfect, we have:*

$$R = R', P = \alpha \cdot P'$$

where $\alpha = \frac{|M(C')|}{|M(C)|}$, and $M(X)$ returns the set of pairs in set X predicted as positive by M.

Proof. We first show that $Matches(C') = Matches(C)$, where $Matches(X)$ returns the set of actual matching pairs in set X. We then derive the expressions for recall and precision of f on the set C.

In the crowdsourced estimation algorithm, we begin with C as the candidate set, and then iteratively apply reduction rules on the candidate set to finally obtain C'. Since each of these reduction rules retain all the positive examples in C, C' must contain all the positive examples in C, i.e., $Matches(C') = Matches(C)$.

Let G denote this set of all the matching pairs in C. We have, $G = Matches(C') = Matches(C)$.

Given G as the set of actual positives in C (as well as C'), $M(C)$ as the set of predicted positives in C, and $M(C')$ as the set of predicted positives in C', we can show that (a) for the set C, $|Actual\,positives| = |G|$, $|Predicted\,positives| = |M(C)|$, $|True\,positives| = |M(C) \cap G|$, and (b) for the set C', $|Actual positives| = |G|$, $|Predicted positives| = |M(C')|$, $|True positives| = |M(C') \cap G|$.

Using the definitions for precision and recall, from Section 6.1, we can now write R, R', P and P' as follows:

$$R = \frac{|M(C) \cap G|}{|G|}, R' = \frac{|M(C') \cap G|}{|G|} \tag{6.3}$$

$$P = \frac{|M(C) \cap G|}{|M(C)|}, P' = \frac{|M(C') \cap G|}{|M(C')|} \tag{6.4}$$

Let $C'' = C \backslash C'$. We can write, $M(C) = M(C') \cup M(C'')$, where $M(C'')$ is the set of predicted positives in C''. Thus, $M(C) \cap G = (M(C') \cup M(C'')) \cap G$, i.e.,
$M(C) \cap G = (M(C') \cap G) \cup (M(C'') \cap G)$.

However, G is completely contained inside C'. Thus, $M(C'') \cap G = \emptyset$. Therefore,

$$M(C) \cap G = M(C') \cap G$$

Substituting this in the equations 6.3 and 6.4, we get:

$$R = R', P = \frac{|M(C')|}{|M(C)|} \cdot P'.$$

□

6.2.3 Optimizations

We now describe two key optimizations we have implemented to improve the performance of crowdsourced estimation in Corleone.

1. Pruning During Plan Search: In our online search algorithm for finding the best plan (Section 6.2.2), in Step 1 we enumerate all the possible options, each executing a subset of the reduction rules $\mathcal{R}$. There are a total of 2^n such options given n reduction rules. In Step 2 we estimate the cost of each option, and then select the lowest-cost option. However, estimating the cost of all the possible options could be expensive, e.g., if $n = 20$ we need to explore more than 1 million options.

In practice, we can often heuristically prune a significant number of these options even before estimating their cost. We achieve this as follows. We begin cost estimation with the option corresponding to no rules (i.e. apply no reduction rules, and sample directly). We then proceed to options with more and more rules such that, an option with $k + 1$ rules is explored only after exploring all the options with k rules. As we proceed, we keep track of the lowest cost option, and we consider an option with more number of rules only if there is still a possibility of finding an option with a cost lower than the current lowest.

Specifically, suppose that we have explored all options with k rules or less. Let c be the sampling cost for the lowest cost option explored so far, and m be the number of reduction rules for the lowest cost bin. Let c_r be the expected cost of evaluating an additional reduction rule. Let c_{all} be the expected sampling cost after applying all the reduction rules from $\mathcal{R}$.

The cost of any option unexplored option $\mathcal{Q}$, is the sum of (1) the cost of evaluating the rules in $\mathcal{Q}$ and (2) the cost of sampling from the reduced set after applying the rules in $\mathcal{Q}$. The cost of sampling from the reduced set, for any option $\mathcal{Q}$, can not get lower than c_{all}, i.e., c_{all} is a lower bound on component (2) of the total cost for option $\mathcal{Q}$.

Since $\mathcal{Q}$ is unexplored yet, it must contain more than k rules. The overall cost for $\mathcal{Q}$ can be lower iff, the increase in component (1) is more than offset by the potential reduction in component (2). Thus, we consider any option with $k+1$ rules, only if:

$$(k+1-m) \cdot c_r < (c - c_{all})$$

2. Reliably Estimating Error Margins for P & R: Equations 6.1 and 6.2 for estimating P and R use the standard normal approximation interval (also known as Wald interval) for binomial proportions to estimate error margins. However, this does not work well for extreme values of P or R, i.e., very close to 0 or 1, or when the sample size is very small, i.e., n_{ap} or n_{pp} is very small. Hence, we use the Agresti-Coull interval for estimation to get more reliable estimates [22].

Algorithm 6.2 Pseudo-code for the $estimate$ operation

Input: Original candidate set C, Reduced candidate set C', Matcher M

Output: P, ϵ_p, R, ϵ_r

1: /* We first sample from C' until the recall error is below ϵ_{max} */
2: $\{R, \epsilon_r, S, n_{ap}, n_{tp}\} \leftarrow estimateRecall(C, C', M)$ /* Algorithm 6.3 */
3: /* Now we check if precision error is already below ϵ_{max}, if yes we are done */
4: $S_{pp} = M(S)$, $n_{pp} = |S_{pp}|, done = false$ /* here $M(A)$ denotes $\{t \in A : M(t) = +\}$ */
5: $\alpha = \frac{|M(C')|}{|M(C)|}$, $P = \alpha \cdot \frac{n_{tp}}{n_{pp}}$
6: $\epsilon_p = \alpha \cdot Z_{1-\delta/2}\sqrt{\left(\frac{P(1-P)}{n_{pp}}\right)\left(\frac{n^*_{pp}-n_{pp}}{n^*_{pp}-1}\right)}$
7: **if** $\epsilon_p \leq \epsilon_{max}$ **then**
8: done = true
9: **end if**
10: /* If not done, then sample more from ($M(C') - S_{pp}$)*/
11: **while** (not done) **do**
12: Uniformly draw next batch B of b examples from ($M(C') - S_{pp}$)
13: Get label $l(t)$ for each example $t \in B$, from the crowd
14: $S_{pp} = S_{pp} \cup B$, $n_{pp} = n_{pp} + b$
15: **for all** $t \in B$ **do**
16: **if** ($l(t) = +$) **then**
17: $n_{tp} = n_{tp} + 1$
18: **end if**
19: **end for**
20: Compute P and ϵ_p as in 5 and 6.
21: **if** $\epsilon_p \leq \epsilon_{max}$ **then**
22: done = true
23: **end if**
24: **end while**
25: **return** P, ϵ_p, R, ϵ_r

Algorithm 6.3 Pseudo-code for $estimateRecall()$

Input: Original candidate set C, Reduced candidate set C', Matcher M

Output: R, ϵ_r, Labeled sample S, n_{ap}, n_{tp}

1: $S = \emptyset$, $n_{ap} = 0$, $n_{tp} = 0$, $n = 0$
2: $recallDone = false$
3: **while** (not recallDone) **do**
4: Uniformly draw next batch B of b examples from ($C' - S$)
5: Get label $l(t)$ for each example $t \in B$, from the crowd
6: $S = S \cup B$, $n = n + b$
7: **for all** $t \in B$ **do**
8: **if** ($l(t) = +$) **then**
9: $n_{ap} = n_{ap} + 1$
10: **if** $M(t) = +$ **then**
11: $n_{tp} = n_{tp} + 1$
12: **end if**
13: **end if**
14: **end for**
15: $max_{ap} = n_{ap} + |C'| - n$
16: $R = \frac{n_{tp}}{n_{ap}}$, $\epsilon_r = Z_{1-\delta/2}\sqrt{\left(\frac{R(1-R)}{n_{ap}}\right)\left(\frac{max_{ap}-n_{ap}}{max_{ap}-1}\right)}$
17: **if** $\epsilon_r \leq \epsilon_{max}$ **then**
18: recallDone = true
19: **end if**
20: **end while**
21: **return** R, ϵ_r, S, n_{ap}, n_{tp}

Chapter 7

Iterating to Improve

In practice, entity matching is not a one-shot operation. After blocking and matching, developers often estimate the matching result, then revise their matching solution, and repeat the process until they can not improve any further. Sometimes developers need to iterate until their solution meets a required accuracy requirement, e.g., for product matching at e-commerce companies, achieving a very high precision (above 99%) is a must as a wrongly matched pair of products can translate into actual revenue loss as well as damage to the company's reputation. This iterative development workflow is often unavoidable as it is really hard to identify the best solution (e.g., which rules to use for matching, how big a training set to use to train the matcher, which features to use, etc.) in first attempt.

A common way to revise is to find tuple pairs that have proven difficult to match, then modify the current matcher, or build a new matcher specifically for these pairs. For example, when matching e-commerce products, a developer may find that the current matcher does reasonably well across all categories, except in Clothes, and so may build a new matcher specifically for Clothes products.

Corleone operates in a similar fashion. It estimates the matching accuracy (as discussed earlier), then stops if the accuracy does not improve (compared to the previous iteration). Otherwise, it revises and matches again. Specifically, it attempts to locate difficult-to-match pairs, then builds a new matcher specifically for those. The challenge is how to locate difficult-to-match pairs. Our key idea is to identify *precise* positive and negative rules from the learned random forest, then remove all pairs covered by these rules (they are, in a sense, easy to match, because there already exist precise rules that cover them). We treat the remaining examples as difficult to match, because

the current forest does not contain any precise rule that covers them. In this chapter, we describe this idea in detail.

Before we proceed, a brief remark on revising the solution by locating difficult pairs. There are many different ways a developer could revise the solution and iterate, e.g., add or remove features used for training the matcher, modify the strategy used for selecting unlabeled examples during active learning, change the classification model for the matcher, modify the crowd management algorithm so as to improve the accuracy of crowdsourced labels, etc. Locating difficult pairs is just one way to revise.

The developer could, in principal, revise the solution by employing combinations of the above methods. However, exploring all the possible combinations would be extremely expensive and even impractical, as we will have an exponential number of combinations to explore and exploring each combination itself would cost significant time and money. We observe that in practice, developers often revise by locating difficult pairs and focusing on those. Hence, as a feasible and practical approach, Corleone currently considers only this particular method for revising the solution. We have found this to be highly effective so far (see Section 9.3 for empirical results).

1. Extract Positive and Negative Rules: Let C be the candidate set of tuples output by the Blocker. Let F be the random forest learned by matcher M. To locate the pairs in C that are difficult to match, we first identify highly precise rules to predict negative and positive pairs in C. We then use these rules to remove pairs in C, and reduce C to a smaller set of difficult to match pairs. We identify these highly precise negative and positive rules by extracting them from forest F.

In Section 4 we have discussed how to extract negative rules from F, select top rules, use the crowd to evaluate them, then keep only the highly precise ones. Here we do exactly the same thing to obtain k (k = 20) highly precise negative rules (or as many as F has). The only difference is that the coverage and precision are being computed over the candidate set C instead of the sample S. Note that some of these rules might have been used in estimating the matching accuracy (Section 6). We need not evaluate those rules again.

We then proceed similarly to obtain k highly precise positive rules (or as many as F has). A positive rule is similar to a negative rule, except that it is a path from a root node of a tree to a "yes" leaf node in F. That is, if a pair satisfies the rule, then the rule predicts that the pair of tuples match. Thus, the coverage of a positive rule R over the candidate set C, $cov(R, C)$, is the set of pairs in C for which R predicts "yes". Note that when computing the coverage for each rule R over C, each rule is applied to the set of feature vectors for the pairs in C. Since the set of feature vectors fit in memory, computation of rule coverage is entirely in-memory.

2. Apply Rules to Remove Easy-to-Match Pairs: Let $\mathcal{E}$ be the set of positive and negative rules so obtained. Recall that in the current iteration we have applied matcher M to match examples in set C. We now apply all rules in $\mathcal{E}$ to C, to remove examples covered by any of these rules. Let the set of remaining examples be C' ($C' = C \setminus W$, where $W = \bigcup_{R \in \mathcal{E}} cov(R, C)$). As mentioned earlier, we treat these examples as difficult to match, because they have not been covered by any precise (negative or positive) rule in the current matcher M.

When applying the rules, we do not need to recompute the coverage $cov(R, C)$ of each rule $R \in \mathcal{E}$, as we can reuse the coverage computed in step 1 when we evaluate the rules using crowd to identify the top k rules. Specifically, we compute the coverage of any rule R over the set C only once and store it using a bit vector representation (a single bit for each pair in C, bit is set to 1 only if the pair satisfies the rule). When applying the rules in $\mathcal{E}$ to C, we can just use standard logical operations ($\neg$ and $\vee$) on these bit vectors to compute a bit vector representation for the resultant set C'.

3. Learn a New Matcher for Surviving Pairs: In the next iteration, we learn a new matcher M' over the set C', using the same crowdsourced active learning method described in Section 5. This is then followed by the crowdsourced estimation step (Section 6), followed by the next iteration and so on. In each iteration we train a new matcher. In the end we use the so-constructed set of matchers (*matcher ensemble*) to match examples in C. For example, if we terminate after two iterations, then we use matcher M to make prediction for any example in $C \setminus C'$ and M' for any example in C'.

Deciding Whether to Revise or Terminate: At the end of each iteration of matching and estimation, Corleone decides whether to revise the solution or terminate the execution. Intuitively, we should revise and continue to the next iteration only if the accuracy of the matcher ensemble is going to improve by training a new matcher. In practice, the improvement achieved in each iteration diminishes, and after a few iterations the accuracy stops improving completely. Thus, to decide whether to revise or terminate, Corleone monitors the estimated F_1-score of the matcher ensemble and decides to terminate as soon as it observes that the F_1-score has stopped improving.

For example, suppose that at the end of first iteration f is the estimated F_1-score of matcher M and at the end of the second iteration f' is the estimated F_1-score of the matcher ensemble M and M'. At the end of second iteration, Corleone will decide to revise and continue to the third iteration only if $f' > f$. Otherwise, it will terminate, i.e., it will apply the matcher ensemble, return the predicted matches to the user, and report to the user the final accuracy estimates.

As an optimization, Corleone makes an exception to the above rule in the following cases, when it seems very unlikely that training a new matcher will result in any significant improvement in F_1-score. Specifically, if the set C' is too small (e.g., having less than 200 examples), or if no significant reduction happens (e.g., $|C'| \geq 0.9 * |C|$), then we terminate without learning a new matcher M' for C'.

Chapter 8

Engaging the Crowd

As described so far, Corleone heavily uses crowdsourcing in each of the key components in the entity matching workflow: blocker, matcher, estimator, and difficult pairs' locator. In particular, it engages the crowd to label examples, to (a) supply training data for active learning (in blocking and matching), (b) supply labeled data for accuracy estimation, and (c) evaluate rule precision (in blocking, accuracy estimation, and locating difficult pairs). We now describe how Corleone engages the crowd to label examples, highlighting in particular how we address the challenges of noisy crowd answers and example reuse.

8.1 Crowdsourcing Platforms

Currently we use Amazon's Mechanical Turk (AMT) to label the examples. However we believe that much of what we discuss here will also carry over to other crowdsourcing platforms (Section 2.3.2 gives an overview of the services offered by AMT). To label a batch of examples, we organize them into HITs (i.e., "Human Intelligence Tasks"), which are the smallest tasks that can be sent to the crowd. AMT provides an API allowing requesters to programmatically access the platform and execute various functions, such as post HITs, check whether all HITs have been answered, and retrieve answers submitted by the workers. Crowds often prefer many examples per HIT, to reduce their overhead (e.g., the number of clicks). Hence, we put 10 examples into a HIT. Within each HIT, we convert each example (x, y) into a question "does x match y?". Figure 8.1 shows a sample question for product matching task. Currently we pay 1-2 pennies per question, a typical pay rate for EM tasks on AMT.

Do these products match?		
	Product 1	Product 2
Product image		
Brand	Kingston	Kingston
Name	Kingston HyperX 4GB Kit 2 x 2GB …	Kingston HyperX 12GB Kit 3 x 4GB …
Model no.	KHX1800C9D3K2/4G	KHX1600C9D3K3/12GX
…….	… …. …	… …. …
Features	o Memory size 4 GB o 2 x 2GB 667 MHz …	o 3 x 4 GB 1600 MHz o HyperX module with …
Yes	No	Not sure

Figure 8.1: A sample question to the crowd.

When using a crowdsourcing platform like AMT, there are several challenges such as recruiting the crowd, designing the HIT, maintaining a good relationship with workers to ensure a good reputation as a requester, etc. We discuss a few of these challenges here.

Designing the HIT so as to attract workers is quite challenging, especially as a new requester on AMT. To recruit workers and get their feedback on the HIT interface, we posted some trial matching HITs on AMT, and asked workers on a popular forum, called TurkerNation [18], to provide feedback on the HIT interface. The workers provided a number of useful suggestions (e.g., put multiple questions per page, not to pay less than 1 cent per question, open links in new tabs, etc.) which we incorporated in our final HIT design.

The trial HITs helped us improve the HIT interface in yet another way. In our initial HIT interface, each worker had to choose between two options to answer each question: “Yes” (the two tuples match) and “No” (the two tuples do not match). On analyzing the answers obtained for trial HITs, we observed that for certain questions workers genuinely find it difficult to decide whether the two tuples match. In such cases, they were forced to randomly pick one of the two options.

This could adversely affect the algorithms that use these answers. Hence, to avoid such a situation, we added a third option for the workers to choose from: "Not sure" (Figure 8.1).

To avoid the possibility of dissatisfied workers writing bad reviews for us on worker forums (e.g., Turker Nation [18]), we included a comment box in each HIT so that workers can provide feedback on specific HITs. In addition, we decided not to reject payment to any worker even if that worker might have very low accuracy in labeling.

Apart from HIT design, the number of pairs labeled at a time (i.e. the size of each batch) is also an important factor that affects how many workers decide to work on our task and thus, the latency in obtaining answers from the crowd (as has been observed in prior work [50]). A large batch size can attract more workers and reduce the latency, however, it may also result in higher monetary cost. Corleone currently leans toward minimizing the monetary cost, and thus, posts 20 questions (two HITs) at a time during active learning and rule evaluation. When estimating the accuracy of matcher (Section 6.2.2), Corleone posts 50 questions (five HITs) in each batch. This is because the total number of pairs that need to be labeled for estimation is typically very high (several hundreds in our experiments) and thus, a larger batch size (50 instead of 20) has no effect on the total monetary cost of the estimation step.

8.2 Combining Noisy Crowd Answers

Several solutions have been proposed for combining noisy answers, such as golden questions [65] and expectation maximization [59]. They often require a large number of answers to work well, and it is not yet clear when they outperform simple solutions, e.g., majority voting [85]. Hence, we started out using the 2+1 majority voting solution: for each question, solicit two answers; if they agree then return the label, otherwise solicit one more answer then take the majority vote. This solution is commonly used in industry and also by recent work [50, 66, 93].

Soon we found that this solution works well for supplying training data for active learning, but less so for accuracy estimation and rule evaluation, which are quite sensitive to incorrect labels. Thus, we need a more rigorous scheme than 2+1. We adopted a scheme of "strong majority vote": for each question, we solicit answers until (a) the number of answers with the majority label minus

that with the minority label is at least three, or (b) we have solicited seven answers. In both cases we return the majority label. For example, four positive and one negative answers would return a positive label, while four negative and three positive would return negative.

The strong majority scheme works well, but is too costly compared to the 2+1 scheme. So we improved it further, by analyzing the importance of different types of error, then using strong majority only for the important ones. Specifically, we found that false positive errors (labeling a true negative example as positive) are far more serious than false negative errors (labeling a true positive as negative). This is because false positive errors change n_{ap}, the number of actual positives, which is used in estimating $R = n_{tp}/n_{ap}$ and in Formula 6.2 for estimating ϵ_r. Since this number appears in the *denominators*, a small change can result in a big change in the error margins, as well as estimated R and hence F_1. The same problem does not arise for false negative errors. Based on this analysis, we use strong majority voting only if the current majority vote on a question is positive (thus can potentially be a false positive error), and use 2+1 otherwise. We found empirically that this revised scheme works very well, at a minimal overhead compared to the 2+1 scheme.

Note that the solution discussed above only considered two possible values for the crowd provided label: positive (match) and negative (do not match). In our case, a worker can also provide a third value for the label: neutral (not sure if the tuples match). We handle this third value as follows. If the majority vote on a question is neutral, then we consider this as an indication that this pair is genuinely hard to label and hence do not use the pair at all (e.g., for training the matcher, or estimating the matcher's accuracy). Otherwise, we use the scheme described above considering only the positive and negative answers. For example, three neutral, one negative and one positive answer would return a neutral label, while one neutral, four positive answers would return a positive label.

8.3 Re-using Labeled Examples

Since Corleone engages the crowd to label at many different places (blocking, matching, estimating, locating), we cache the already labeled examples for reuse. When we get a new example,

we check the cache to see if it is there and has been labeled the way we want (i.e., with the 2+1 or strong majority scheme). If yes then we can reuse without going to the crowd. Interestingly this simple and obviously useful scheme poses complications in how we present the questions to the crowd.

Recall that during active learning (for blocking and matching) we send a batch of 20 examples at a time, packed into two HITs (10 questions each), to the crowd. What happens if we find 15 examples out of 20 already in the cache? It turns out we cannot send the remaining 5 examples as a HIT. Turkers avoid such "small" HITs because they contain too few questions and thus incur a high relative overhead.

To address this problem, we require that a HIT always contains 10 questions. Now suppose that k examples out of 20 have been found in the cache and $k \leq 10$, then we take 10 example from the remaining $20 - k$ examples, pack them into a HIT, ask the crowd to label, then return these 10 plus the k examples in the cache (as the result of labeling this batch). Otherwise if $k > 10$, then we simply return these k examples as the result of labeling this batch (thus ignoring the $20 - k$ remaining examples).

When reusing the labels during rule evaluation and accuracy estimation, we can not ignore any examples (as done during active learning) since that would affect the statistical validity of the estimated accuracy. For example, if 15 examples out of 20 are already in the cache, we can not just ignore the five unlabeled examples, and only use the 15 cached examples to estimate rule precision, as these 15 examples would not represent a uniform random sample. As a solution, instead of sampling only 20 examples at a time, we sample until we have 20 unlabeled examples and then send those 20 to the crowd. Suppose that after sampling 20 examples (only five of them being unlabeled), we sample 30 more examples, and 15 of them are unlabeled. Thus, we have a sample of total 50 examples, 20 of which are unlabeled. We send these 20 examples to the crowd, get the labels, and then return all the 50 labeled examples.

Chapter 9

Empirical Evaluation

We now empirically evaluate Corleone. Table 9.1 describes three real-world data sets for our experiments. Restaurants matches restaurant descriptions. Citations matches citations between DBLP and Google Scholar [64]. These two data sets have been used extensively in prior EM work (Section 9.1 compares published results on them with that of Corleone, when appropriate). Products, a new data set we created, matches electronics products between Amazon and Walmart. Overall, our goal is to select a diverse set of data sets, with varying matching difficulties.

We used Mechanical Turk and ran Corleone on each data set three times, each in a different week. The results reported below are averaged over the three runs. In each run we used common turker qualifications to avoid spammers, such as allowing only turkers with at least 100 approved HITs and 95% approval rate. We paid one cent per question for Restaurants & Citations, and two cents for Products (it can take longer to answer Product questions due to more attributes being involved).

9.1 Overall Performance

Accuracy and Cost: We begin by examining the overall performance of Corleone. The first five columns of Table 9.2 (under "Corleone") show this performance, broken down into P (precision), R (recall), F_1 (harmonic mean of precision and recall), the total cost, and the total number of tuple pairs labeled by the crowd. The results show that Corleone achieves high matching accuracy, 89.3-96.5% F_1, across the three data sets, at a reasonable total cost of $9.2-$256.8. The number of pairs being labeled, 274-3205, is low compared to the total number of pairs. For example,

Datasets	Table A	Table B	# of Matches
Restaurants	533	331	112
Citations	2616	64263	5347
Products	2554	22074	1154

Table 9.1: Data sets for our experiment.

Datasets	Corleone					Baseline 1			Baseline 2			Published Works
	P	R	F_1	Cost	# Pairs	P	R	F_1	P	R	F_1	F_1
Restaurants	97.0	96.1	**96.5**	$9.2	274	10.0	6.1	**7.6**	99.2	93.8	**96.4**	**92-97** [63, 86]
Citations	89.9	94.3	**92.1**	$69.5	2082	90.4	84.3	**87.1**	93.0	91.1	**92.0**	**88-92** [26, 63, 64]
Products	91.5	87.4	**89.3**	$256.8	3205	92.9	26.6	**40.5**	95.0	54.8	**69.5**	Not available

Table 9.2: Comparing the performance of Corleone against that of traditional solutions and published works.

after blocking, Products has more than 173,000 pairs, and yet only 3205 pairs need to be labeled, thereby demonstrating the effectiveness of Corleone in minimizing the labeling cost.

The total number of pairs labeled is lowest for Restaurants, followed by Citations and Products. This can be attributed to three factors:

1. Restaurants is small enough not to trigger blocking, and thus avoids the blocking cost.

2. Restaurants and Citations are both *relatively easier* to match compared to Products, i.e., they have less diverse matching pairs. As a result, they require fewer training examples to achieve similar matching accuracy.

3. Being harder to match, Products is much harder to reduce during the estimation step, e.g., Restaurants requires only 1 reduction rule during estimation, while Products requires an average of 16 rules. It also has a lower positive density than Citations. Thus, it requires many more labeled pairs to estimate precision and recall, than the other two datasets.

Run time: The total run times for Corleone are 2.3 hours, 2.5 days and 2.1 days for Restaurants, Citations and Products datasets respectively. To understand the run times for each of the components of Corleone, let us focus on Products. Here, Corleone takes 2.5 hours for blocking, 1.4

days for learning, 14 hours for estimation and 1.4 hours for reduction. If we exclude the crowd time, then the runtimes for Corleone over the three datasets are 12 seconds, 12.4 minutes and 49 minutes respectively. The total machine time taken to compute entropy for all the examples in the candidate set is only 2 minutes, which is negligible compared to the overall run time. This clearly shows that time spent to obtain labels from the crowd dominates the run time.

Comparison to Traditional Solutions: In the next step, we compare Corleone to two traditional solutions: Baseline 1 and Baseline 2. Baseline 1 uses a developer to perform blocking, then trains a random forest using the same number of labeled pairs as the average number of labeled pairs used by Corleone. Baseline 2 is similar to Baseline 1, but uses 20% of the candidate set (obtained after blocking) for training. For example, for Products, Baseline 1 uses 3205 pairs for training (same as Corleone), while Baseline 2 uses 20% * 180,382 = 36,076 pairs, more than 11 times what Corleone uses. Baseline 2 is therefore a very strong baseline matcher.

The next six columns of Table 9.2 show the accuracy (P, R, and F_1) of Baseline 1 and Baseline 2. The results show that Corleone significantly outperforms Baseline 1 (89.3-96.5% F_1 vs. 7.6-87.1% F_1), thereby demonstrating the importance of active learning, as used in Corleone. Corleone is comparable to Baseline 2 for Restaurants and Citations (92.1-96.5% vs. 92.0-96.4%), but significantly outperforms Baseline 2 for Products (89.3% vs. 69.5%). This is despite the fact that Baseline 2 uses 11 times more training examples.

Baseline 1 uses passive learning (i.e. training examples are randomly sampled once at the beginning), while Corleone uses active learning, selecting the training examples iteratively, and only those judged informative are added to the training set, until it has a satisfactory matcher. This explains why Baseline 1 performs a lot worse than Corleone, in spite of using the same number of labeled examples. For Restaurants, Baseline 1 does especially worse due to the extremely low positive density (0.06%) in the candidate set, resulting in very few (or no) positive example in the training set.

Baseline 2 also uses passive learning, but with a significantly larger training set. This explains the improved performance of Baseline 2 over Baseline 1. On Products, however, Baseline 2 does

not fare very well compared to Corleone. This again has to do with Products dataset being *harder to match*, i.e., requiring a larger and more diverse training set.

When comparing Baseline 1 and Baseline 2 against Corleone, it is important to note that Corleone not only returns the matched results, but also the estimated precision and recall, while Baseline 1 and Baseline 2 do not report any estimates for accuracy.

Comparison to Published Results: The last column of Table 9.2 shows F_1 results reported by prior EM work for Restaurants and Citations. On Restaurants, [63] reports 92-97% F_1 for several works that they compare. Furthermore, CrowdER [86], a recent crowdsourced EM work, reports 92% F_1 at a cost of $8.4. In contrast, Corleone achieves 96.5% F_1 at a cost of $9.2 (including the cost of estimating accuracy). On Citations, [26, 63, 64] report 88-92% F_1, compared to 92.1% F_1 for Corleone. It is important to emphasize that due to different experimental settings, the above results are not directly comparable. However, they do suggest that Corleone has reasonable accuracy and cost, while being hands-off.

Summary: The overall result suggests that Corleone achieves comparable or in certain cases significantly better accuracy than traditional solutions and published results, at a reasonable crowdsourcing cost. The important advantage of Corleone is that it is totally hands-off, requiring no developer in the loop, and it provides accuracy estimates of the matching result.

9.2 Performance of the Components

We now "zoom in" to examine Corleone in more details.

Datasets	Cartesian Product	Umbrella Set	Recall (%)	Cost	# Pairs
Restaurants	176.4K	176.4K	100	$0	0
Citations	168.1M	38.2K	99	$7.2	214
Products	56.4M	173.4K	92	$22	333

Table 9.3: Blocking results for Corleone.

Blocking: Table 9.3 shows the results for crowdsourced automatic blocking executed on the three data sets. From left to right, the columns show the size of the Cartesian product (of tables

Datasets	Iteration 1				Estimation 1				Reduction 1		Iteration 2				Estimation 2			
	# Pairs	P	R	F_1	# Pairs	P	R	F_1	# Pairs	Reduced Set	# Pairs	P	R	F_1	# Pairs	P	R	F_1
Restaurants	140	97	96.1	96.5	134	95.6	96.3	96	0	157								
Citations	973	89.4	94.2	91.7	366	92.4	93.8	93.1	213	4934	475	89.9	94.3	92.1	0	95.2	95.7	95.5
Products	1060	89.7	82.8	86	1677	90.9	86.1	88.3	94	4212	597	91.5	87.4	89.3	0	96	93.5	94.7

Table 9.4: Corleone's performance per iteration on the data sets.

A and B), the size of the umbrella set (i.e., the set after applying the blocking rules), recall (i.e., the percentage of positive examples in the Cartesian product that are retained in the umbrella set), total cost, and total number of pairs being labeled by the crowd. Note that Restaurants is relatively small and hence does not trigger blocking.

The results show that automatic crowdsourced blocking is quite effective, reducing the total number of pairs to be matched to be just 0.02-0.3% of the original Cartesian product, for Citations and Products. This is achieved at a low cost of $7.2-22, or just 214-333 examples having to be labeled. In all the runs, Corleone applied 1-3 blocking rules. These rules have 99.9-99.99% precision. Finally, Corleone also achieves high recall of 92-99% on Products and Citations. For comparison purposes, we asked a developer well versed in EM to write blocking rules. The developer achieved 100% recall on Citations, reducing the Cartesian product to 202.5K pairs (far higher than our result of 38.2K pair). Blocking on Products turned out to be quite difficult, and the developer achieved a recall of 90%, compared to our result of 92%. Overall, the results suggest that Corleone can find highly precise blocking rules at a low cost, to dramatically reduce the original Cartesian products, while achieving high recall.

Performance of the Iterations: Table 9.4 shows Corleone's performance per iteration on each data set. To explain, consider for example the result for Restaurants (the first row of the table). In Iteration 1 Corleone trains and applies a matcher. This step uses the crowd to label 140 examples, and achieves a true F_1 of 96.5%. Next, in Estimation 1, Corleone estimates the matching accuracy in Iteration 1. This step uses 134 examples, and produces an estimated F_1 of 96% (very close to the true F_1 of 96.5%). Next, in Reduction 1, Corleone identifies the difficult pairs and comes up with 157 such pairs. It uses no new examples, being able to re-use existing examples. At this point,

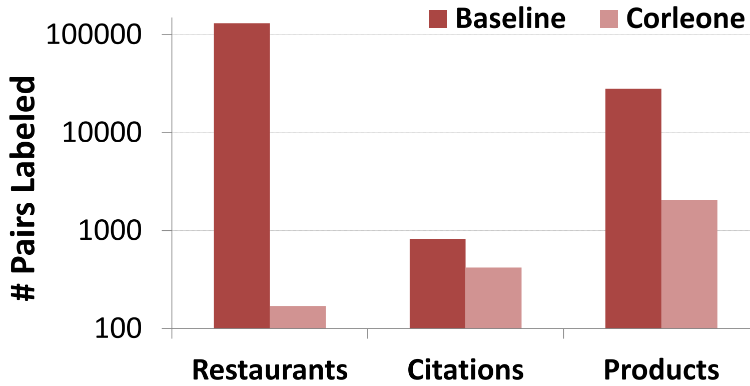


Figure 9.1: Comparing estimation cost of Corleone vs. Baseline.

since the set of difficult pairs is too small (below 200), Corleone stops, returning the matching results of Iteration 1.

The result shows that Corleone needs 1-2 iterations on the three data sets. The estimated F_1 is quite accurate, always within 0.5-5.4% of true F_1. Note that sometimes the estimation error can be larger than our desired maximal margin of 5% (e.g., Estimation 2 for Products). This is due to the noisy labels from the crowd. Despite the crowd noise, however, the effect on estimation error is relatively insignificant. Note that the iterative process can indeed lead to improvement in F_1, e.g., by 3.3% for Products from the first to the second iteration (see more below). Note further that the cost of reduction is just a modest fraction (3-10%) of the overall cost.

Crowd Workers: For the end-to-end solution, an average of 22 (Restaurants) to 104 (Citations) turkers worked on our HITs. The average accuracy of turkers was the highest for Restaurants (94.7%), and the lowest for matching citations (75.9%). For Products, it was again quite high (92.4%), which is understandable given the familiarity of turkers with products as opposed to citations. The accuracy of the labels inferred by Corleone (using majority voting) was higher than the average turker accuracy for all the datasets (96.3% for Restaurants, 77.3% for Citations, and 96% for Products). Note that in spite of the low labeling accuracy for Citations, Corleone still performs just as good as the traditional solutions and published works.

9.3 Additional Experimental Results

We have run a large number of additional experiments to extensively evaluate Corleone.

Estimating Matching Accuracy: Section 9.2 has shown that our method provides accurate estimation of matching accuracy, despite noisy answers from real crowds. Compared to the baseline accuracy estimation method in Section 6.1, we found that our method also used far fewer examples. We now compare the average cost (i.e., the number of pairs labeled) of the two methods. For a fair comparison, we use a simulated crowd that labels everything perfectly for both the methods, and we start the estimation procedure without any cached labels.

Figure 9.1 shows the number of pairs labeled for estimation for all three datasets, for the both the methods (*Baseline*) and Corleone). For Restaurants, the baseline method needs 100,000+ examples to estimate both P and R within a 0.05 error margin, while ours uses just 170 examples. For Citations and Products, we use 50% and 92% fewer examples, respectively. The result here is not as striking as for Restaurants primarily because of the much higher positive density for Citations and Products.

Effectiveness of Reduction: Section 9.2 has shown that the iterative matching process can improve the overall F_1, by 0.4-3.3% in our experiments. This improvement is actually much more pronounced over the set of difficult-to-match pairs, primarily due to increase in recall. On this set, recall improves by 3.3% and 11.8% for Citations and Products, respectively, leading to F_1 increases of 2.1% and 9.2%. These results suggest that in subsequent iterations Corleone succeeds in zooming in and matching correctly more pairs in the difficult-to-match set, thereby increasing recall.

Note that this increase in recall is a lot more pronounced for Products (11.8%) than for Citations (3.3%). This is mainly due to the lower positive density for Products (1.9% compared to 21.4%). The lower positive density results in fewer positives getting selected in the training set in the first iteration, and thus, a less representative training set. In the second iteration, we narrow down to a set with a much higher positive density, and thus, many of these previously unrepresented positives get added to the training set which leads to a higher recall.

Effectiveness of Rule Evaluation: Section 9.2 has shown that blocking rules found by Corleone are highly precise (99.9-99.99%). We have found that rules found in later steps (estimation, reduction, i.e., identifying difficult-to-match pairs) are highly precise as well, at 97.5-99.99%. For the estimation step, Corleone uses 1, 4.33, and 7.67 rules on average (over three runs) for Restaurants, Citations, and Products, respectively. For the reduction step, Citations uses on average 11.33 negative rules and 16.33 positive rules, and Products uses 17.33 negative rules and 9.33 positive rules.

The number of rules used during estimation depends on the ease of matching the dataset and the positive density, e.g., for Restaurants, which is the easiest to match among the three, just one rule is sufficient for the estimation procedure, while Products, which is the most difficult to match among the three, and has lower positive density than Citations, requires more rules (7.67) for the estimation procedure.

For the rules used in reduction step, the average accuracy is still very high (97.5% and above), but a little lower than that for estimation (99.9% and above). This is because we select only the topmost precise rules for estimation (since we need near-perfect rules here). For reduction, there is no such necessity, and thus, we consider even the not-so-precise rules.

Finally, if we look at the number of rules used, then we see a similar trend as for estimation rules, except for the positive rules. For Products, only 9.33 precise positive rules are applied during reduction, whereas for Citations we apply 16.33 positive rules. This is because Citations has almost 5 times the total number of positive examples as Products (5347 vs. 1154). With more positives, we get more positive rules, and thus, higher number of precise positive rules. Overall, the crowdsourced rule evaluation works extremely well to give us almost perfect rules.

Using Corleone up to a Pre-specified Budget: Corleone runs until the estimated matching accuracy no longer improves. However, the user can stop it at any time. In particular, he or she can run Corleone only until a pre-specified budget for crowdsourcing has been exhausted, an operation mode likely to be preferred by users in the "masses", with modest crowdsourcing budgets. We found that even with a modest budget, Corleone already delivers reasonable results (which improve steadily with more money). In the case of Products, for instance, a budget of $50,

Money	True P/R/F1	Execution status	Additional info.
\$50	79.8/78.4/79.1	Iteration#1	Finished blocking at cost = \$22.44 in 3.1 hours. So far spent \$27.04 on matching, to create a training set with 415 examples. Total time = 9.4 hours.
\$100	86.6/85.2/85.9	Estimation#1	Finished matching iteration#1 at cost = \$97.68. Estimated positive density = 4.9%. Currently evaluating rules to reduce the universe. No P/R estimates yet. Total time = 24.6 hours.
\$150	86.6/85.2/85.9	Estimation#1	Finished reduction. So far spent \$51.48 on estimation. 535 labeled examples used for estimation. Est. P = 89.8 $\pm$ 7.5 %, Est. R = 86.2 $\pm$ 8.6 %. Total time = 29.6 hours.
\$200	86.6/85.2/85.9	Reduction#1	Finished estimation #1 at total cost = \$197.03. Currently reducing the input set for matching iteration#2. Est. P = 91 $\pm$ 4.2%, Est. R = 88.7 $\pm$ 4.9%. Total time = 34.8 hours.
End (\$250.9)	89.1/88.0/88.5	Finished	Finished matching iteration#2 and estimation#2. Stopped since training set contains $>$ 25% of the input set. Est. P = 95.5 $\pm$ 3%, Est. R = 94.4 $\pm$ 3.6%. Total time = 57.2 hours.

Table 9.5: Execution status of Corleone at different time points during one particular run on Products.

\$150, and \$250.9 (the end) delivers F_1 score of 79.1%, 85.9%, and 88.5%, respectively. Table 9.5 shows the detailed execution status of Corleone for one particular run for Products, from start (\$0) to finish (\$250.9).

9.4 Sensitivity Analysis

Each of the components of Corleone has some parameters that can be used to fine tune the performance. We have run extensive sensitivity analyses for Corleone to test the robustness of the system. Overall we observe that small changes in these parameters do not affect the performance of Corleone in any significant way. However, one can certainly tune them to extract the best performance out of the system. We report here the results for the most important factors that may affect the performance of Corleone. For all the sensitivity analyses, we performed experiments on the Products dataset, since that is the most difficult one to match, as can be seen from the results reported earlier. Additionally, we used a simulated crowd for all the experiments, since performing so many experiments with real crowd is prohibitively expensive and time consuming.

Blocking Threshold t_B: The effect of t_B is most pronounced in the blocking stage, so we vary t_B from 1 million to 20 million to see how it affects the blocking time, size of the candidate set, and the recall of the candidate set.

Effect of t_B on the blocking time: The total blocking time can be broken down into 4 main components:

- Sampling and feature vector generation time (t_1)
- Rule learning time (t_2)
- Rule evaluation time (t_3)
- Applying the rule on the Cartesian product time (t_4)

t_1, t_2 and t_3 are directly proportional to t_B because higher the t_B, larger is the sample size, longer it takes to sample and compute features and longer it takes to learn and evaluate rules. From our plots we observe that t_1 + t_2 + t_3 increases linearly from $3m$ $43s$ to $1h$ $15m$ $40s$ as we increase t_B from $1M$ to $20M$. t_4, however, does not directly depend on t_B but depends on the blocking rule applied. So we do not observe any particular trend for t_4 as we vary t_B. The highest $t4$ is $34m$ $49s$, the lowest is $11m$ $38s$ and the average is $21m$ $2s$.

Effect of t_B on the size of the candidate set: We observe that, on increasing t_B the candidate set size increases in the average case. For example, we obtain candidate sets of sizes $22.4K$, $247K$ and $3.8M$ for $t_B = 1M$, $3M$ and $5M$ respectively. However, note that the size of the candidate set depends on the blocking rule that was applied. Thus, in certain cases, this trend may not be strictly followed, as we observe for $t_B = 10M$ for which we have a candidate set of size $583.6K$.

Effect of t_B on the recall of the candidate set: In general, larger the candidate set higher is the recall. We observe this behavior in our experiments where we have recalls of 92.63%, 95.67% and 99.13% for candidate sets of sizes $247K$, $3.8M$ and $5.9M$ respectively. For a very small t_B of 1 million, we get a recall of 82.8%, otherwise the recall is in the range 92.63% to 99.31%.

Number of Trees (k) and Features (m) in Random Forest: Section 5.1 describes the random forest ensemble model and its parameters: the number of trees in the forest (k) and the number of features (m) considered for splitting at each node in the tree. These parameters can only affect the training step, hence, to understand the effect of varying these parameters on Corleone, we only execute the training step in the workflow. In particular, we start with a candidate set returned by the blocking step, and the 4 user-provided labeled pairs, and then perform active learning over the candidate set until the stopping condition kicks in.

Figures 9.2a, 9.2b, and 9.2c show the effect of varying the number of trees (k) in the forest from 5 to 100, on the execution time, cost, i.e., the number of pairs labeled for training, and accuracy (F_1 score). We observe that the execution time grows linearly as we increase k, from 1 minute for 5 trees to 20 minutes for 50 (Figure 9.2a). This is expected since for every new tree added, the forest needs to apply one more tree to classify a given pair. The increasing number of trees also lead to a better F_1, however, the increase in F_1 is significant (12%) only as we go from 5 to 10 trees. Going from 10 to 50 trees in the forest, the F_1 increases by just 2% (Figure 9.2c). This marginally higher accuracy comes at a higher labeling cost as well (Figure 9.2b). From 5 to 10 trees, the labeling cost rises from 500 to almost 1400, but beyond this, the cost rises at a very slow pace, going up by just 200 as we go from 10 to 50 trees. Overall, we can see that at k = 10, the execution time is very small (2 minutes), while the F_1 is almost as good as that for k = 50. Thus, the default value of k = 10 used in Corleone is highly justified.

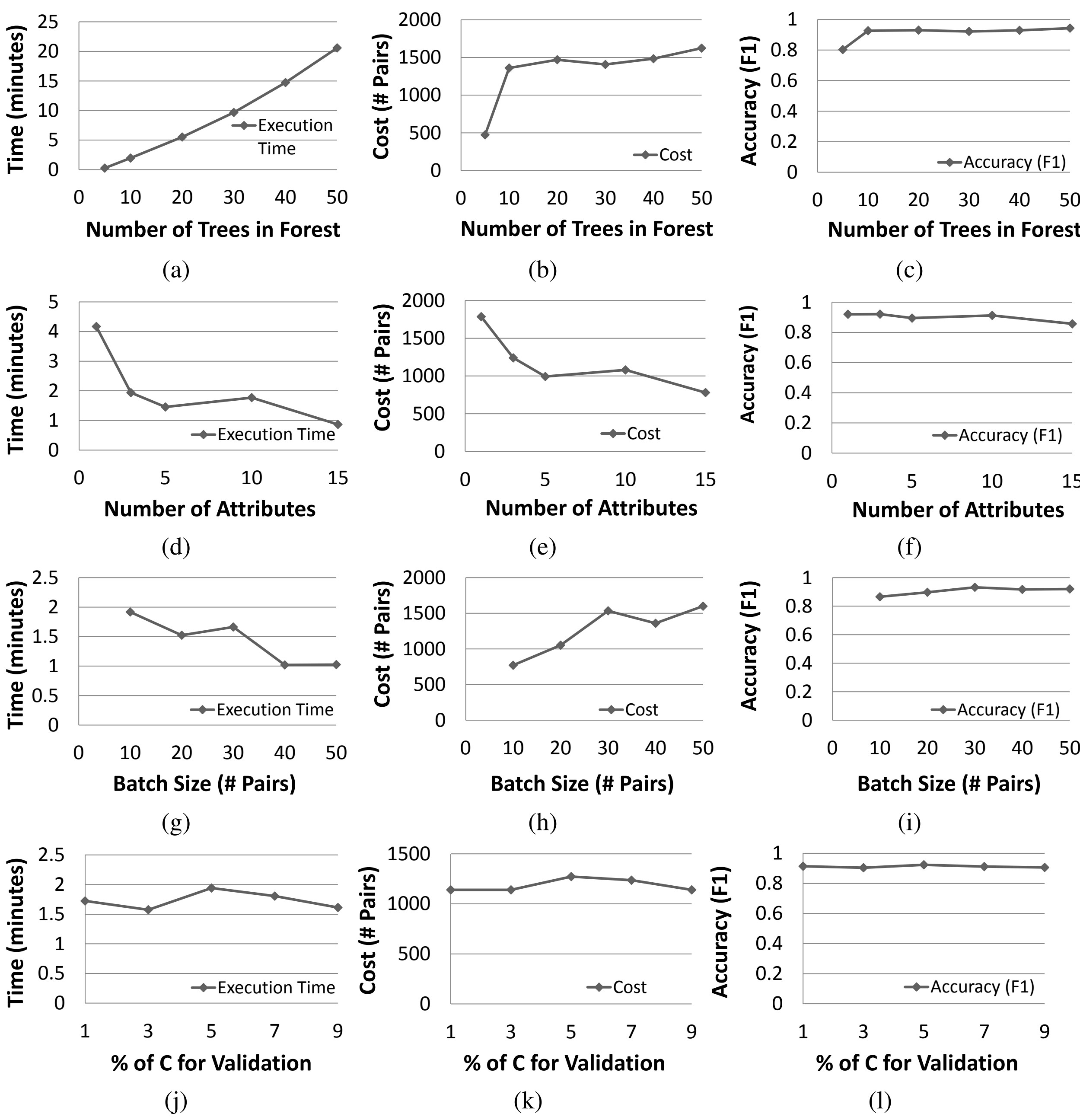


Figure 9.2: Sensitivity analysis for parameters in learning.

The small rise in F_1, and cost as we increase the number of trees can be explained. By increasing the number of trees, we get a more diverse ensemble (forest), since each tree is trained on a different portion of the training data. This results in a higher F_1. However, more diverse trees also

have more disagreement (higher entropy), and thus, the confidence of the whole forest takes longer to stabilize leading to a higher labeling cost.

Next, we describe the effect of increasing the number of features, i.e., attributes (m) considered at each node when learning the trees in the forest. Figures 9.2d, 9.2e, and **??** plot the execution time, labeling cost, and F_1 as we increase number of attributes m from 1 to 15. Note that for Products dataset, Corleone sets m = 5, since the total number of attributes n for Products is 23, and m is set to $log(n) + 1$.

We found that on increasing m, the execution time goes down from 4 minutes for m = 1, to 1.4 minutes for m = 5, and then decreases very slowly beyond this value of m (Figure 9.2d). The labeling cost goes down by more than 50% from m = 1 to m = 5, and then reduces by just 20% as m goes to 15. This reduction in time and cost, comes at a cost. The F_1 drops by 7% on increasing m from 1 to 15. With a higher m, we get a less diverse ensemble, and thus, the confidence converges sooner leading to stopping active learning sooner. This results in a lower cost as well as lower F_1. Note that at m = 5, which is the default in Corleone the F_1 is within 2% of that for m = 1, while both execution time and cost are much lower than for m = 1.

Batch Size (q) for Active Learning: In all the experiments reported so far we had set the batch size q to 20. We examine the effect of batch size on the active learning step by executing only the training step, exactly as done for varying k and m above.

Figures 9.2g, 9.2h, and 9.2i show the effect of varying the batch size (q) from 10 to 50, on execution time, the labeling cost (# examples) for learning, and F_1 score. On increasing q from 10 to 50, we found that the execution time reduced from 2 to 1 minutes, since the the algorithm required fewer learning iterations. The number of iterations dropped by 55% from 77 to 32. This is expected since the algorithm can learn more in each iteration by selecting a bigger batch. The labeling cost, however, increased by more than 100%, from 770 for q = 10, to 1600 for q = 50, while there is a small rise in F_1 from 86% to 92%. At q = 20, which is the default in Corleone we observe that F_1 is again within 2% of the maximum, while the labeling cost is within 20% of the best attainable. Given that there is a trade-off between cost, F_1, and time, we choose 20 as our "sweet spot" for batch size.

Size of Validation Set to Decide Stopping: The active learning algorithm sets aside part of the candidate set as validation set, to decide when to stop (as described in 5.3). Corleone uses 3% of the candidate set as validation set. To examine the effect of varying the size of validation set, we perform just the training step over the candidate set C, as done above for k, m, and q, and vary the validation set size from 1% of candidate set (C) to 9% of C. Figures 9.2j, 9.2k, and 9.2l show the effect of increasing the percentage of C set aside for validation.

Overall, we observe that increasing the size of validation set has almost no effect on time. cost, or F_1. The execution time stays within 1.6 and 1.9 minutes as we increase t_B. The labeling cost stays within 1140 and 1240, while F_1 stays in the range of 90% and 92%. Thus, we observe that the active learning algorithm is quite robust to a change in the size of validation set.

Parameters Used for Rule Pruning: Section 4.3 describes how we use P_{min} and maximum number of rules (k), to select at most top k rules (currently, k = 20). In blocking step, the number of rules that are finally applied is no more than 3 in all our experiments with the three datasets, thus, varying k from 10 to 50 had no effect on the system.

When evaluating the top rules, we use P_{min} as a threshold to remove the imprecise rules. On varying P_{min} from 90% to 99%, we did not observe any change in the rules that get picked, and thus, on the whole system. This can be explained by the fact that in all our experiments the top rules that got evaluated for precision were either highly precise (precision being higher than 99% for most), or had a much lower (less than 80%) precision. These low precision rules may get picked when we fail to get a tight upper bound on their precision. These factors could have more significant effect on other datasets. We would consider exploring this in future work.

Labeling Accuracy of the Crowd: To test the effect of labeling accuracy, we use the random worker model in [55, 59] to simulate a crowd of random workers with a fixed error rate (i.e., the probability of incorrectly labeling an example). We found that a small change in the error rate causes only a small change in Corleone's performance. However, as we vary the error rate over a large range, the performance can change significantly. With a perfect crowd (0% error rate), Corleone performs extremely well on all three data sets. With moderate noise in labeling (10%

error rate), F_1 reduces by only 2-4%, while the cost increases by up to $20. As we move to a very noisy crowd (20% error rate), F_1 further dips by 1-10 % for Products and Citations, and 28% for Restaurants. The cost on the other hand shoots up by $250 to $500. Managing crowd's error rates better therefore is an important topic for future research.

Number of Labels Per Pair: In Section 8, we mentioned that we use 2+1 labeling scheme during the training phase, to keep the cost low.

For the training step, we only get a maximum of three labels per pair. We now analyze the effect of using more labels per pair during training. The number of labels requested per pair is of value only in presence of a noisy crowd. Hence, to analyze the effect of per-pair labels, we used the low accuracy simulated crowd (20% error rate). This crowd performed especially worse on Restaurants. Hence, we report here the results for increasing the number of labels for Restaurants, with a crowd having 20% error rate.

On increasing the maximum requested labels from three to five to seven, we found that the F_1 improved significantly from 70% to 97%, while the cost reduced by more than $500. Intuitively, the cost reduces drastically because the quality of inferred labels improves a lot, which in turn, leads to a quick termination of the active learning algorithm. This experiment demonstrates a little non-intuitive fact that with a noisy crowd, getting more labels could not only give better performance, but sometimes it can also help to drastically lower the total cost.

9.5 Setting the System Parameters

Finally, we describe how we set the various parameters of Corleone for our experiments. In the blocker, t_B is set to be the maximal number of tuple pairs that can fit into memory (a heuristic used to speed up active learning during blocking), and is currently set to three million, based on the amount of memory available on our machine. We have experimented and found that Corleone is robust to varying t_B (e.g., as we increase t_B, the time it takes to learn blocking rules increases only linearly, due to processing larger samples). See Section 9.4 for details.

The batch size $b = 20$ is set using experimental validation with simulated crowds (of varying degrees of accuracy). The number of rules k is set to a conservative value that tries to ensure that the blocker does not miss any good blocking rules. Our experiments show that k can be set to as low as five without affecting accuracy. Similarly, experiments suggest we can vary P_{min} from 0.9 to 0.99 without noticeable effects, because the rules we learned appear to be either very accurate (at least .99 precision) or very inaccurate (well below 0.9). Given this, we current set P_{min} to 0.95. The confidence interval 0.95 and error margin 0.95 are set based on established conventions.

In the matcher, the parameters for the random forest learner are set to default values in the Weka package. The stopping parameters (validation set size, smoothing window w, etc.) are set using experiments with simulated crowds with varying degrees of accuracy.

For engaging the crowd, we solicit three labels per pair because three is the minimum number of labels that give us a majority vote, and it has been used extensively in crowdsourcing publications as well as in industry. When we need higher crowd accuracy for the estimator, we need to consider five labels or seven labels. After extensive experiments with simulated crowds, we found that five gave us too wide error ranges, whereas seven worked very well (for the estimator). Hence our decision to solicit seven labels per pair in such cases. Finally, the estimator and the difficult pairs' locator use many algorithms used by the blocker and the matcher, so their parameters are set as described above.

Chapter 10

Discussion

In this chapter we discuss the key design choices that went into building Corleone, the scope of this dissertation, and various opportunities to extend the current system.

10.1 Design Choices

Our goal behind building Corleone was to (i) have a first end-to-end HOC system for the entity matching problem that performs well on real datasets, thus demonstrating the potential of HOC, and (ii) have the first starting point for further research in building HOC systems. As a result, when designing Corleone and its components, our focus was on having a practical solution that actually works with real data and real crowd, and keeping it simple unless absolutely necessary. At the same time, we wanted the overall architecture to be very general, so that each of the components can be improved and extended.

We now take a top-down look at the choices we have made while designing the Corleone system. At the very top, Corleone is aimed at solving the entity matching problem using the crowd, starting from the two input tables, all the way to returning the matching pairs and the estimated matching accuracy. The ideal goal for such a system would be to maximize the matching accuracy (F_1 score), minimize the monetary cost of crowdsourcing, and minimize the end-to-end execution time. This is a highly challenging optimization problem, since it is very difficult to model the trade-off between matching accuracy, monetary cost, and execution time. This trade-off can be highly dependent on the particular application setting, e.g., a large company may be willing

to pay a large monetary price for a small gain in accuracy, while a domain scientist may care a lot more about the cost than the accuracy.

As a first step toward tackling this challenge, we pursue a more modest goal focusing on maximizing the accuracy while minimizing the monetary cost. Optimizing for both accuracy and cost for the entire EM workflow is again highly non-trivial, since it is very hard to estimate the accuracy and cost of any step in the workflow before executing that particular step. To illustrate, if we could predict the accuracy and cost for the matching step when we are executing the blocking step, then we could use that knowledge to stop blocking at an optimal point. However, before we perform the actual matching step, it is very hard to predict its performance. Hence, we break down the problem, focusing on optimizing the accuracy and cost for each individual step. Now we look at some key choices made in the various steps in the EM workflow.

10.1.1 Blocking Threshold

Why Need a Threshold for Blocking? We trigger blocking only if the size of the Cartesian product ($A \times B$) is above a threshold (t_B) (as described in Section 4.2). Having such a threshold has to do with the fundamental reason blocking is needed in the first place, which is to execute the EM workflow in an *acceptable* amount of time.

Building and applying a matching solution is often computationally expensive, e.g., if we are learning a classifier, then we need to enumerate all the possible pairs that may match, compute all the features for all the pairs, train the classifier using the labeled pairs, and then apply the classifier to predict each pair as matching or non-matching. If we have to do this for the entire $A \times B$, then for very large tables, it could take several days to even weeks to execute, even on a cluster of machines. Blocking is just a fast solution that reduces the original input to a small size on which we can apply the expensive matching solution. However, blocking comes at a cost as it is typically not as accurate as matching. Thus, intuitively one would want to perform blocking *only if* applying matching is going to be prohibitively expensive. If the Cartesian product is small enough that we can execute matching within an acceptable time, then it would certainly be better than to block first

and risk losing some matching pairs. To model this notion of "small enough", we use a parameter t_B that represents the threshold for blocking.

Setting the Blocking Threshold: The blocking threshold limits the size of the input to the matching step, and in turn, limits the execution time. Thus, intuitively, we should set the blocking threshold as small as we can to minimize the execution time for the matching step. On the other hand, the smaller the threshold we set, the more pairs we would need to eliminate during blocking, i.e., more matching pairs will be lost in blocking. To balance these conflicting goals, we look further into the components that dominate the execution time for matching.

The execution time T for the matching step (Chapter 5), can be expressed as $T = n \cdot T_{iter}$, where n is the number of iterations of active learning, T_{iter} is the execution time for each iteration. T_{iter} is dominated by t_1, the time to select the next batch of pairs for training, and t_2, the time for labeling the selected pairs. Now t_2 is constrained by the batch size and the maximum number of labels we can request per pair. t_2 is independent of the threshold we set. However, t_1 is very much dependent on the threshold we set, since t_1 involves computing the entropy for every pair in the candidate set. t_1 involves only the CPU cost if the feature vectors for the candidate set fit in memory and thus, is relatively small compared to t_2. However, if the feature set is larger, then t_1 also has an I/O component, which can get significant for large candidate sets. To balance our goals of (i) keeping t_B small to constrain the execution, while (ii) keeping it large enough to avoid loss in matching accuracy, we set t_B such that the feature vectors of the candidate set fit in memory, which avoids I/O cost, and constrains the execution time for matching.

A few brief remarks about the above strategy. First, we set the goal of trying to obtain a candidate set of size t_B, and we try to get as close to that goal as possible. But we cannot guarantee that we will have a candidate set size of t_B or less. For example, if no blocking rule is good, then we cannot perform any blocking and we would still have a candidate set size of $|A \times B|$. Second, the above strategy is just a reasonable heuristic; other strategies are possible and should be explored in future work. Finally, setting the value of t_B depends on the computational infrastructure used for matching, e.g., if we are performing active learning over a very large cluster then we could set t_B to a much larger value without affecting the execution time.

We would like to note that besides efficiency, there is another reason why blocking is needed in EM workflows: the extreme imbalance in the ratio of positive to negative examples in EM datasets. Learning an accurate matcher becomes much more difficult when operating directly on a highly imbalanced data set. The blocking step helps overcome this imbalance by removing a significant portion of the non-matching pairs. Thus, an alternative strategy to decide whether to perform blocking or not could be driven by the imbalance in the data set, i.e., the fraction of positives present in the Cartesian product $A \times B$. We have not explored this strategy in our current work, however, this could be a promising direction for future research.

10.1.2 Sampling to Learn Blocking Rules

Why Sample to Learn Blocking Rules? As far as we know, ours is the first work that learns the blocking rules starting from scratch, i.e. just the input tables, without requiring a developer. To learn such rules we need some labeled examples to train on. To obtain such training examples in a cost minimizing fashion, a natural choice is to use active learning. However, if we were to use the entire $A \times B$ as the input to the active learning algorithm (which is the same as the one used in the matching step), then the learning process will be prohibitively expensive. This defeats the very purpose of blocking. Hence, we take the approach of sampling a small set of pairs from $A \times B$, and using only this sample to learn the blocking rules. This way we constrain the cost as well as execution time for blocking.

Setting the Sample Size: Given that we take the sampling approach to constrain the cost and execution time for blocking, we should minimize the sample size as much as we can. On the other hand, to learn effective rules, we need to have a representative sample with a "sufficient" number of positives in it. To illustrate, if our sample contains zero positive examples, then any rule will have 100% accuracy on the sample and we will have no way to judge which rule is more effective. Thus, we also need to have a "large enough" sample. These conflicting goals are very similar to what we faced when setting the blocking threshold. In fact, just like the matching step, the execution time for blocking is also dominated by the crowdsourced active learning algorithm. Thus, following

similar reasoning we set the sample size to t_B such that the feature vectors for the sample just fit in memory.

10.1.3 Labeling Scheme for Crowdsourcing

In Chapter 8, we describe the labeling scheme used in Corleone in the Estimator component, which requires "strong majority" only if the majority label is positive. In this scheme, we get a minimum of 3 labels and a maximum of 7. Getting a minimum of 3 labels is quite straightforward as you need at least 3 labels to avoid the possibility of a tie. In fact, this is a standard value used in many works that use majority voting for combining crowd answers [50, 66, 93].

We limit the maximum number of labels that can be obtained per pair to 7. Now clearly we need some limit on the total labels for a single pair we may get as otherwise, in the worst case, the algorithm may never stop, and we would end up spending an exorbitant amount. Since we want to minimize the cost ideally we should set this limit as low as possible. The first choice for this limit would be 5. However, in our experiments with simulated noisy crowd, we found that getting 5 labels was not sufficient to estimate the matcher accuracy with low error, especially with a very noisy crowd (error rate of 20% or more). After increasing this limit to 7 labels, on the other hand, we found the total cost to increase only marginally by up to 100$, while the accuracy of estimation improved significantly (by more than 10%), very close to what we would obtain with a perfect crowd. On increasing the limit further to 9 or 11 labels, the cost continues to increase, whereas there is very little gain in estimation accuracy. Based on these experiments with simulated crowd, we set the maximum number of labels to get to 7.

10.2 Opportunities for Extension

The Corleone system is just a first step toward building HOC systems. In this dissertation, we have focused on only some of the many novel challenges involved in building a robust scalable HOC system for entity matching. As for any system, addressing all of the challenges in the first attempt is next to impossible, and thus, Corleone is designed with a clean separation between

the various components so that each of them can be easily extended. We describe here some key opportunities to further extend the system.

10.2.1 Scaling Up to Very Large Datasets

While the Corleone system is highly promising, as can be seen from the empirical results in Chapter 9, we need to ensure that it can handle datasets of any nature and size. For scaling up to very large datasets, each of the components of Corleone must scale up. Intuitively, the blocking component is the one that is most responsible to handle the scale problem. If blocking works the way it is supposed to, then in most cases the output of blocking should return a candidate set small enough for efficient execution of the matching and subsequent steps in the workflow. Hence, we now discuss how we can scale up the blocking component.

Sampling Strategy for Large Datasets: After the system has decided to block, the first step in blocking is to sample from the Cartesian product. The sampling strategy currently used (Step 2 in Section 4.3) randomly samples $t_B/|A|$ tuples from the larger table B and takes their Cartesian product with all of A. As one may suspect, this strategy may not work very well if we have very large tables or very low fraction of matching pairs in $A \times B$. In such a situation, the current sampling strategy may give us a sample with very few positives.

This possible limitation can be addressed as follows. First, we must select t_B as large as we can without hurting the execution time for matching. In our current system, we assume a very modest infrastructure for matching, and thus set t_B to 3 million. Here our goal was to demonstrate that even with very strict constraints on the size of candidate set, our solution comes up with highly precise rules. In practice, t_B can be easily increased to a few hundred million by using a cluster to speed up the active learning algorithm (as we discuss later). This will ensure that t_B is much larger than $|A|$ or $|B|$.

Secondly, we can use a non-uniform sampling strategy to select positives with a high likelihood. We already have one such strategy and our preliminary results indicate that it works very well even on large datasets or when there is very low positive density. Here is how it works. Given the sample size t_B, we randomly select t_B/m tuples from the table B ($m = 200$). For each selected

Datasets	$\|A \times B\|$	Positive density (%)	# positives in sample
Citations	168.1 M	3.20 x 10^{-3}	4123
Citations 2X	672.4 M	1.60 x 10^{-3}	4219
Citations 5X	4202.5 M	0.64 x 10^{-3}	4186
Citations 10X	16,810.0 M	0.32 x 10^{-3}	4145

Table 10.1: Stratified sampling for blocking.

tuple from B we select m tuples from table A, forming m tuple pairs, and add them to the sample. The m tuples are selected as follows. We "stratify" the tuples in A into two sets: A_1 - tuples that have at least one token in common with the B tuple, and A_2 tuples with no common token. We then randomly pick up to $m/2$ tuples from A_1, and then randomly pick tuples from A_2 until we have a total of m tuples. At the end of this process we have the desired sample with roughly t_B tuple pairs.

To demonstrate the effectiveness of this strategy, we present in Table 10.1 the results for a preliminary experiment on Citations dataset. We use the new sampling strategy to sample from different versions of the Citations dataset. To test how well it scales up to large datasets, we create larger versions of the Citations dataset by replicating the tables. Thus, Citations 2X has tables A and B with twice as many tuples as in Citations, while 5X Citations has 5 times the number of tuples. For 2X Citations, $A \times B$ is 4 times that of Citations, while the positive density is half of that for Citations (since the actual number of matching pairs is only 2 times that of Citations). Thus, as we replicate Citations more and more times, the size of $A \times B$ keeps increasing while positive density keeps dropping. In Table 10.1, we show the number of positives selected in the sample, as we create bigger and bigger versions of the Citations dataset. As we can see, the new sampling strategy gives us a consistently high number of positives, even as the size of $A \times B$ increases and the positive density drops.

Applying the Rules Efficiently: In the current system, when a blocking rule is actually applied to eliminate the obvious non-matching pairs, the rule is evaluated for every pair in $A \times B$. This may work well for now as this is computed over a Hadoop cluster, however, beyond a certain input size it will be too expensive to enumerate all the pairs. Fortunately, there is a way around this problem.

Prior work on scaling up blocking has developed techniques such as sorted neighborhood, canopy clustering, and indexing to speed up the application of blocking rules [48]. The Blocker can be extended to incorporate these techniques for applying the blocking rules.

Efficient Active Learning: The active learning algorithm for training the matcher proceeds iteratively. In each iteration, it processes all the unlabeled pairs in the candidate set and selects the most informative pairs among them (Section 5.2). This step involves computing the entropy for each unlabeled pair in the candidate set. The blocking threshold already ensures that the size of candidate set is no more than t_B. However, when scaling up to very large datasets, we might want to set t_B to a very large value (hundreds of millions).

To ensure that we can efficiently execute matching over a very large candidate set, we need a scalable solution to compute the entropy for pairs in the candidate set. Thankfully, this is a very straightforward problem to parallelize, since we can compute the entropy for each pair independently. Thus, we can easily distribute the entropy computation over a MapReduce cluster and can easily handle candidate sets with hundreds of millions of pairs. There are additional techniques we can use to further speed up this solution, e.g., as the active learning progresses, we can narrow down the set of pairs for which we need to compute the entropy in each iteration.

10.2.2 Improving Matching and Estimation

Improving the Blocking Recall: One way to improve the overall matching performance is by improving the recall for blocking. This could be achieved in two ways. First is to learn a more diverse set of blocking rules. Our current framework separates the process of generating candidate blocking rules, from the evaluation of the candidates to pick the best rules. Thus, we can easily extend the system to add new techniques to generate the rules. In addition to pulling the rules from the forest, we could use other learning techniques to obtain candidate blocking rules. We can even consider directly obtaining simple rules from the crowd, and adding them to our set of candidate rules.

Another way to improve the blocking recall is by better evaluation of the blocking rules. If we can precisely predict the number of errors each candidate rule is going to make, then we can do a better job at picking the best rules.

Improving the Estimation of Matcher Accuracy: The current solution for estimating the precision and recall of the matcher (Section 6.2) works very well, but is not perfect. It relies on the accuracy of the rules used for reduction, and also the accuracy of the labels inferred from crowd provided labels. Next step to would be to make this solution more tolerant to the errors made by the crowd, as well as the imperfection of the reduction rules.

10.2.3 Cost Models

Modeling the relationship between monetary cost of crowdsourcing, accuracy of labels provided by crowd, and labeling time can be immensely useful to improve a HOC system such as Corleone. It can help solve some key optimization challenges, e.g., given a monetary budget constraint, how to best allocate it among the blocking, matching, and accuracy estimation step? As another example, paying more per question often gets the crowd to answer faster. How can we minimize the monetary cost given a time constraint? A possible approach is to profile the crowd during the blocking step, then use the approximate crowd models (in terms of time, money, and accuracy) to help guide the subsequent steps of Corleone.

10.2.4 Other Extensions

Engaging the Crowd: There are a number of ways to improve the interaction with the crowd. First, we can improve the way we manage the workers, and infer the answers. For instance, we can test alternative solutions to infer the answers, such as estimating the worker accuracy and labels in an online fashion. Secondly, we can look at optimizing the time taken by the crowd, together with the accuracy of labels. Third, we can improve the interface used for presenting the pairs to the crowd. Finally, we can extend the system to use multiple crowdsourcing platforms, instead of using only Amazon Mechanical Turk.

Adding New Components: Another direction for extending the current system is to add new components that complement the EM workflow. One such component is a verification component at the end of the current workflow, that takes the final predicted matches and the crowd provided labels as input, and verifies the predicted matches to return only the most trustworthy matching pairs. Some of the techniques proposed in recent crowdsourced EM solutions [41, 86, 87] can be used to implement such a verification component.

Another useful addition to the Corleone system would be a pre-processing component that performs data cleaning and normalization operations. This can be extremely useful in further improving the matching accuracy. Finally, a visualization module can make the system much more user-friendly. This can also empower the user to make informed decisions in the middle of workflow execution, e.g., when to stop the execution, or whether to skip a particular step in workflow.

10.2.5 Applying to Other Problem Settings

Finally, it would be interesting to explore how the ideas underlying Corleone can be applied to other problem settings. Consider for example crowdsourced joins, which lie at the heart of recently proposed crowdsourced RDBMSs. Many such joins in essence do EM. In such cases our solution can potentially be adapted to run as hands-off crowdsourced joins. We also note that crowdsourcing typically has helped learning by providing labeled data for training and accuracy estimation. Our work however raises the possibility that crowdsourcing can also help "clean" learning models, such as finding and removing "bad" positive/negative rules from a random forest. Finally, our work shows that it is possible to ask crowd workers to help generate complex machine-readable rules, raising the possibility that we can "solicit" even more complex information types from them.

Chapter 11

Conclusion

Entity matching is the problem of finding data records referring to the same real world entity. Entity matching is a critical step in many different applications such as comparison shopping, knowledge base construction, managing healthcare data, and master data management at enterprises. This problem has been studied by researchers from databases, statistics, and machine learning communities over the past several decades. However, there is still no EM solution that is robust cross different problem domains and works out-of-the-box without requiring substantial developer effort.

In recent years, there have been increasing efforts (e.g., [41, 86, 87, 89, 90]) to apply crowdsourcing to EM. This recent work demonstrates that crowdsourcing has a strong potential to advance the state-of-the-art EM solutions. However, current crowdsourced EM solutions have a major limitation: they crowdsource only parts of the EM workflow requiring a developer to execute the remaining parts. As a result, they do not scale to the growing EM need at enterprises and crowdsourcing startups and can not handle scenarios where ordinary users want to leverage crowdsourcing to match entities.

This dissertation makes key contributions toward addressing the limitations of current crowdsourced EM work. We have proposed the concept of hands-off crowdsourcing (HOC) and showed how HOC can scale to EM needs at enterprises and startups, and open up crowdsourcing for the masses. We have presented Corleone, a HOC solution for EM, and showed that it achieves comparable or better accuracy than traditional solutions and published results at a relatively little cost, while requiring no developer effort. Our work demonstrates the feasibility and promise of HOC, and suggests many interesting research directions such as extending Corleone to handle a variety

of very large datasets, using crowdsourcing to “clean up” machine learning models, and applying HOC to other problems (e.g, information extraction, schema matching, and data analysis). We believe that HOC systems can greatly improve the accessibility of software-based solutions to ordinary users. Corleone is just a first step in that direction.

Bibliography

[1] *Amazon*. http://www.amazon.com.

[2] *Amazon Mechanical Turk*. http://www.mturk.com.

[3] *CrowdFlower*. http://www.crowdflower.com.

[4] *Google Shopping*. http://www.google.com/shopping.

[5] *IBM InfoSphere QualityStage*. http://www-03.ibm.com/software/products/en/ibminfoqual.

[6] *Informatica Data Quality*. http://www.informatica.com/us/products/data-quality/.

[7] *Link Plus*. http://www.cdc.gov/cancer/npcr/tools/registryplus/lp.htm.

[8] *LinkageWiz*. http://www.linkagewiz.net/.

[9] *Match2Lists*. http://www.match2lists.com/.

[10] *Nextag*. http://www.nextag.com/.

[11] *Oracle Enterprise Data Quality*. http://www.oracle.com/technetwork/middleware/oedq/overview/index.html.

[12] *PriceGrabber*. http://www.pricegrabber.com/.

[13] *reCAPTCHA*. http://www.google.com/recaptcha.

[14] *RIDDLE Datasets*. http://www.cs.utexas.edu/users/ml/riddle/data.html.

[15] *Samasource*. http://samasource.org.

[16] *SecondString*. http://secondstring.sourceforge.net/.

[17] *SimMetrics*. http://sourceforge.net/projects/simmetrics/.

[18] *Turker Nation*. http://www.turkernation.com/.

[19] *WalMart*. http://www.walmart.com.

[20] *Wikipedia*. http://www.wikipedia.org.

[21] *WorkFusion*. http://www.workfusion.com/.

[22] A. Agresti and B. A. Coull. Approximate is better than "exact" for interval estimation of binomial proportions. *American Statistician*, 52(2):119–126, 1998.

[23] Y. Amsterdamer, Y. Grossman, T. Milo, and P. Senellart. Crowd mining. In *SIGMOD*, 2013.

[24] R. Ananthakrishna, S. Chaudhuri, and V. Ganti. Eliminating fuzzy duplicates in data warehouses. In *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB '02, pages 586–597. VLDB Endowment, 2002.

[25] A. Arasu, M. Götz, and R. Kaushik. On active learning of record matching packages. In *SIGMOD*, 2010.

[26] K. Bellare, S. Iyengar, A. G. Parameswaran, and V. Rastogi. Active sampling for entity matching. In *SIGKDD*, 2012.

[27] I. Bhattacharya and L. Getoor. Collective entity resolution in relational data. *ACM Trans. Knowl. Discov. Data*, 1, March 2007.

[28] J. P. Bigham, C. Jayant, H. Ji, G. Little, A. Miller, R. C. Miller, R. Miller, A. Tatarowicz, B. White, S. White, and T. Yeh. Vizwiz: Nearly real-time answers to visual questions. In *Proceedings of the 23Nd Annual ACM Symposium on User Interface Software and Technology*, UIST '10, 2010.

[29] M. Bilenko, B. Kamath, and R. J. Mooney. Adaptive blocking: Learning to scale up record linkage. In *Proceedings of the Sixth IEEE International Conference on Data Mining (ICDM-06)*, pages 87–96, Hong Kong, December 2006.

[30] M. Bilenko and R. J. Mooney. Adaptive duplicate detection using learnable string similarity measures. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD-2003)*, Washington, DC, August 2003.

[31] M. Bilgic, L. Licamele, L. Getoor, and B. Shneiderman. D-dupe: An interactive tool for entity resolution in social networks. In *Visual Analytics Science and Technology (VAST)*, October 2006.

[32] L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[33] Y.-H. Chiang, A. Doan, and J. F. Naughton. Modeling entity evolution for temporal record matching. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, 2014.

[34] P. Christen. Febrl: A freely available record linkage system with a graphical user interface. In *Proceedings of the Second Australasian Workshop on Health Data and Knowledge Management - Volume 80*, HDKM '08, pages 17–25, 2008.

[35] P. Christen. *Data Matching - Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection*. Data-centric systems and applications. Springer, 2012.

[36] P. Christen. A survey of indexing techniques for scalable record linkage and deduplication. *IEEE Transactions on Knowledge and Data Engineering*, 24(9):1537–1555, 2012.

[37] W. W. Cohen. Data integration using similarity joins and a word-based information representation language. *ACM Trans. Inf. Syst.*, 18(3):288–321, July 2000.

[38] W. W. Cohen and J. Richman. Learning to match and cluster large high-dimensional data sets for data integration. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '02, 2002.

[39] N. Dalvi, V. Rastogi, A. Dasgupta, A. Das Sarma, and T. Sarlos. Optimal hashing schemes for entity matching. In *Proceedings of the 22Nd International Conference on World Wide Web*, WWW '13, 2013.

[40] A. Das Sarma, A. Jain, A. Machanavajjhala, and P. Bohannon. An automatic blocking mechanism for large-scale de-duplication tasks. In *CIKM*, 2012.

[41] G. Demartini, D. E. Difallah, and P. Cudré-Mauroux. ZenCrowd: Leveraging probabilistic reasoning and crowdsourcing techniques for large-scale entity linking. In *WWW*, 2012.

[42] A. Doan, A. Halevy, and Z. Ives. *Principles of Data Integration*. Elsevier Science, 2012.

[43] A. Doan, R. Ramakrishnan, and A. Y. Halevy. Crowdsourcing systems on the world-wide web. *Commun. ACM*, 54(4):86–96, Apr. 2011.

[44] X. Dong, A. Halevy, and J. Madhavan. Reference reconciliation in complex information spaces. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, 2005.

[45] U. Draisbach and F. Naumann. Dude: The duplicate detection toolkit. In *Proceedings of the International Workshop on Quality in Databases (QDB), Singapore*, 2010.

[46] H. L. Dunn. Record linkage*. *American Journal of Public Health and the Nations Health*, 36(12):1412–1416, 1946.

[47] M. G. Elfeky, V. S. Verykios, and A. K. Elmagarmid. Tailor: A record linkage toolbox. In *Proceedings of the 18th International Conference on Data Engineering (ICDE)*, pages 17–28, 2002.

[48] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate record detection: A survey. *IEEE Trans. on Knowl. and Data Eng.*, 19(1):1–16, 2007.

[49] I. P. Fellegi and A. B. Sunter. A Theory for Record Linkage. *Journal of the American Statistical Association*, 64(328):1183–1210, 1969.

[50] M. J. Franklin, D. Kossmann, T. Kraska, S. Ramesh, and R. Xin. CrowdDB: Answering queries with crowdsourcing. In *SIGMOD*, 2011.

[51] C. L. Giles, K. D. Bollacker, and S. Lawrence. Citeseer: An automatic citation indexing system. In *Internation Conference on Digital Libraries*, pages 89–98. ACM Press, 1998.

[52] L. Gill. OX-LINK: The oxford medical record linkage system. In *Record Linkage Techniques - 1997: Proceedings of an International Workshop and Exposition*, pages 15–33, 1997.

[53] L. Gill. *Methods for automatic record matching and linkage and their use in national statistics*. Number 25. Office for National Statistics, 2001.

[54] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate string joins in a database (almost) for free. In *Proceedings of the 27th International Conference on Very Large Data Bases*, 2001.

[55] S. Guo, A. Parameswaran, and H. Garcia-Molina. So who won?: Dynamic max discovery with the crowd. In *SIGMOD*, 2012.

[56] P. Hanrahan. Analytic DB technology for the data enthusiast. SIGMOD Keynote, 2012.

[57] M. A. Hernández and S. J. Stolfo. The merge/purge problem for large databases. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, SIGMOD '95, 1995.

[58] J. Howe. *Crowdsourcing: Why the Power of the Crowd Is Driving the Future of Business*. Crown Publishing Group, New York, NY, USA, 1 edition, 2008.

[59] P. G. Ipeirotis, F. Provost, and J. Wang. Quality management on amazon mechanical turk. In *HCOMP*, 2010.

[60] N. Katariya, A. Iyer, and S. Sarawagi. Active evaluation of classifiers on large datasets. In *ICDM*, 2012.

[61] L. Kolb, A. Thor, and E. Rahm. Dedoop: Efficient deduplication with Hadoop. *PVLDB*, 5(12):1878–1881, 2012.

[62] L. Kolb, A. Thor, and E. Rahm. Load balancing for MapReduce-based entity resolution. In *ICDE*, pages 618–629, 2012.

[63] H. Köpcke and E. Rahm. Frameworks for entity matching: A comparison. *Data Knowl. Eng.*, 69(2):197–210, Feb. 2010.

[64] H. Köpcke, A. Thor, and E. Rahm. Evaluation of entity resolution approaches on real-world match problems. *PVLDB*, 3(1-2):484–493, 2010.

[65] J. Le, A. Edmonds, V. Hester, and L. Biewald. Ensuring quality in crowdsourced search relevance evaluation: The effects of training question distribution. In *SIGIR 2010 Workshop on Crowdsourcing for Search Evaluation*, 2010.

[66] A. Marcus, E. Wu, D. Karger, S. Madden, and R. Miller. Human-powered sorts and joins. *PVLDB*, 5:13–24, 2011.

[67] A. Marcus, E. Wu, S. Madden, and R. C. Miller. Crowdsourced databases: Query processing with people. In *CIDR*, 2011.

[68] A. McCallum, K. Nigam, and L. H. Ungar. Efficient clustering of high-dimensional data sets with application to reference matching. In *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '00, 2000.

[69] R. McCann, W. Shen, and A. Doan. Matching schemas in online communities: A Web 2.0 approach. In *Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on*, pages 110–119. IEEE, 2008.

[70] M. Michelson and C. A. Knoblock. Learning blocking schemes for record linkage. In *Proceedings of the 21st National Conference on Artificial Intelligence - Volume 1*, AAAI'06. AAAI Press, 2006.

[71] A. E. Monge. Matching algorithms within a duplicate detection system. *Bulletin of the Technical Committee on Data Engineering*, 23:2000, 2000.

[72] B. Mozafari, P. Sarkar, M. J. Franklin, M. I. Jordan, and S. Madden. Active learning for crowd-sourced databases. *CoRR*, abs/1209.3686, 2012.

[73] H. Park, H. Garcia-Molina, R. Pang, N. Polyzotis, A. Parameswaran, and J. Widom. Deco: A system for declarative crowdsourcing. *PVLDB*, 5(12):1990–1993, 2012.

[74] M. Puzo. *The Godfather*. G. P. Putnam's Sons, 1969.

[75] V. Rastogi, N. Dalvi, and M. Garofalakis. Large-scale collective entity matching. *Proc. VLDB Endow.*, 4(4):208–218, Jan. 2011.

[76] P. Ravikumar and W. W. Cohen. A hierarchical graphical model for record linkage. In *Proceedings of the 20th conference on Uncertainty in artificial intelligence*, UAI '04, 2004.

[77] S. Sarawagi and A. Bhamidipaty. Interactive deduplication using active learning. In *SIGKDD*, 2002.

[78] C. Sawade, N. Landwehr, and T. Scheffer. Active estimation of F-measures. In *NIPS*, 2010.

[79] R. E. Schapire. The boosting approach to machine learning: An overview. In *Nonlinear estimation and classification*, pages 149–171. Springer, 2003.

[80] B. Settles. Active learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 6(1):1–114, 2012.

[81] M. Stonebraker, D. Bruckner, I. F. Ilyas, G. Beskales, M. Cherniack, S. B. Zdonik, A. Pagan, and S. Xu. Data curation at scale: The data tamer system. In *CIDR*, 2013.

[82] B. Trushkowsky, T. Kraska, M. J. Franklin, and P. Sarkar. Crowdsourced enumeration queries. In *ICDE*, 2013.

[83] N. Vesdapunt, K. Bellare, and N. Dalvi. Crowdsourcing algorithms for entity resolution. *Proc. VLDB Endow.*, 7(12):1071–1082, Aug. 2014.

[84] L. Von Ahn. Games with a purpose. *Computer*, 39(6):92–94, 2006.

[85] P. Wais, S. Lingamneni, D. Cook, J. Fennell, B. Goldenberg, D. Lubarov, D. Marin, and H. Simons. Towards large-scale processing of simple tasks with mechanical turk. In *HCOMP*, 2011.

[86] J. Wang, T. Kraska, M. Franklin, and J. Feng. CrowdER: Crowdsourcing entity resolution. *PVLDB*, 5(11):1483–1494, 2012.

[87] J. Wang, G. Li, T. Kraska, M. J. Franklin, and J. Feng. Leveraging transitive relations for crowdsourced joins. In *SIGMOD*, 2013.

[88] L. Wasserman. *All of Statistics: A Concise Course in Statistical Inference*. Springer, 2010.

[89] S. E. Whang, P. Lofgren, and H. Garcia-Molina. Question selection for crowd entity resolution. *PVLDB*, 6(6):349–360, 2013.

[90] S. E. Whang, J. McAuley, and H. Garcia-Molina. Compare me maybe: Crowd entity resolution interfaces. Technical report, Stanford University, 2012.

[91] W. E. Winkler and Y. Thibaudeau. An application of the Fellegi-Sunter model of record linkage to the 1990 U.S. decennial census. Technical Report Statistical Research Report Series RR91/09, U.S. Bureau of the Census, Washington, D.C., 1991.

[92] R. Wolter and K. Haselden. The what, why, and how of master data management. November 2006. http://msdn.microsoft.com/en-us/library/bb190163.aspx.

[93] T. Yan, V. Kumar, and D. Ganesan. CrowdSearch: Exploiting crowds for accurate real-time image search on mobile phones. In *MobiSys*, 2010.

[94] C. J. Zhang, L. Chen, H. V. Jagadish, and C. C. Cao. Reducing uncertainty of schema matching via crowdsourcing. *Proc. VLDB Endow.*, 6(9):757–768, July 2013.

APPENDIX
Additional Empirical Data

A.1 Additional Details for Datasets

- **Restaurants:** This is for the task of matching real listings for restaurants in Los Angeles area from two different restaurant guides (Fodor's and Zagat's). It is publicly available as part of the RIDDLE repository [14]. This dataset has been used extensively in prior EM work [86, 63].

- **Citations:** This is for the task of matching citation records extracted from two sources, DBLP and Google Scholar. This dataset was created by Kopcke et al. [64] for benchmarking different EM solutions.

- **Products:** This is for the task of matching products under Electronics category from two popular online retailers, Walmart and Amazon. We created the dataset ourselves by crawling the websites for the two retailers and extracting records for the products. Below is a detailed description of the process used to create this dataset.

  We first crawled the product pages under Electronics category from the Walmart website [19]. We extracted a total of 31,442 product tuples from Walmart. For each of these products, we used the product title to issue a keyword query to the Amazon website [1], and extracted records for the products listed on the first page of the search result. This resulted in a total of 27,991 product tuples from Amazon. We only kept the product tuples for which we have valid UPCs, so that we could use them to identify the true matching pairs. That left us with 25,535 tuples from Walmart and 22,074 tuples from Amazon. Of all these tuples, we found about 8,350 pairs of tuples for which the UPCs matched. However, on manual

Dataset	Attributes
Restaurants	`name, address, city, type(cuisine)`
Citations	`title, authors, venue, year`
Products	`title, brand, category, parentCategory, modelno, price, item_weight, shipping_weight, dimensions`

Table A.1: Attributes of input tables for the datasets described in Chapter 9.

inspection (as well as some experiments with the crowd) we found that several other tuple pairs *matched*, although the UPCs *did not match*. Thus, identifying matching pairs solely based on UPCs would lead to faulty results. However, manually labeling millions of pairs is not feasible. Hence, we uniformly sampled 10% of the products from Walmart, and we consider the tuple pairs for only these Walmart products. Thus, our final products dataset has 2,554 tuples from Walmart, and 22,074 tuples from Amazon. We then used UPC matches along with manual evaluation to identify 1154 true matching pairs for this dataset.

A.2 Sample Instructions to the Crowd

You will see two product records. Your task is to tell us whether the two records represent the same product or not. For each product, you will see one or more of the following attributes:

- A picture of the product
- Brand
- Model number
- Product name
- Features, Technical details
- Shipping weight
- Product link (Link to the original product page)

Some of the attributes may be missing for some of the products, so please make the most use of the available information. If you are very confused about whether the two products are the same or not, you may choose the third option ”Can not tell” as your answer.

GUIDELINES TO DECIDE WHETHER ”PRODUCT 1” AND ”PRODUCT 2” ARE THE SAME

- The product image, brand, model number, and name are the most helpful in deciding whether the two products are same.
- The name may be similar but not exactly same for matching products, so use your judgement in deciding.
- Model number usually has digits and letters (e.g. ”535T” for TomTom GPS). If model numbers for two products differ only by some additional character like a hyphen ”-” then they still match.
- Matching products may have brand names that may appear little different. For example, here are brand names for two product records that match: ”Apple” and ”Apple Inc.”.
- We are looking for products that exactly match. Thus, if two products have the same brand and model, but have different color, then they ARE NOT THE SAME. Example: If both the records are for ”Nikon Coolpix L18” cameras, but one is Red and the other is Blue, they ARE NOT THE SAME.
- For products in Electronics, make sure that the key specifications match, e.g., “Kingston 4GB Flash Memory” IS DIFFERENT FROM “Kingston 2GB Flash Memory”.

NOTE: IF THE INFORMATION LISTED FOR EACH PRODUCT IS NOT SUFFICIENT TO DECIDE, PLEASE CLICK ON THE PRODUCT LINKS TO VIEW THE ORIGINAL PRODUCT PAGES.

A.3 Sample Top k Candidate Rules for Blocking

title_q < 0.47
title_q < 0.44
title_q < 0.73 AND title_q < 0.44
predicate = brand_jw < 0.8
predicate = brand_swg < 2.94
brand_me < 9
modelno_mtype < 0.5 AND itemwt_score >= 0.03 AND modelno_jw < 0.96 AND atit_wmod_swg < 4.64
title_me < 15.09 AND dimensions_score IS NULL
modelno_jw < 0.96 AND shipwt_score >= 0.08 AND pcat1_jw >= 0.49
modelno_mtype < 0.5 AND price_score >= 0.08 AND pcat1_jw < 0.62
title_me < 15.09 AND dimensions_score >= 0.11
modelno_jw IS NULL AND title_q < 0.44
modelno_mtype IS NULL AND title_swg < 1.11
modelno_mtype IS NULL AND title_me < 14.5
brand_me >= 9 AND modelno_l < 0.94 AND pcat1_jw < 0.69 AND atit_wmod_swg < 2.14
modelno_jw < 0.96 AND shipwt_score >= 0.08 AND pcat1_jw < 0.49 AND atit_wmod_swg < 2.25

Table A.2: Top candidate rules generated by Corleone for one of the runs for Products dataset.

Here, brand_jw, modelno_jw, title_me, etc. are features that compare attributes from the two tables, e.g., brand_jw compares the value of brand using the Jaro-Winkler (JW) similarity function.

A.4 Sample Blocking Rules

Sample blocking rules by developer	
Citations	trigram(a.title, b.title) < 0.2
Products	overlap(a.brand, b.brand) = 0 AND cosine(a.title, b.title) $\leq$ 0.1 AND (a.price/b.price 3 OR b.price/a.price $\geq$ 3 OR a.price IS NULL OR b.price IS NULL)
Sample blocking rules by Corleone	
Citations	trigram(a.title, b.title) < 0.38
Products	jaroWinkler(a.brand, b.brand) IS NULL AND levenshtein(a.modelno, b.modelno) < 0.94 AND trigram(a.title, b.title) < 0.47

Table A.3: Sample blocking rules applied by developer and Corleone (multiple such rules may be applied).