

Human-Centric Debugging of Entity Matching

by

Fatemah Panahi

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

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2017

Date of final oral examination: 12/16/2016

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

Jeffrey F. Naughton, Professor, Computer Sciences

AnHai Doan, Professor, Computer Sciences

Paris Koutris, Assistant Professor, Computer Sciences

Bilge Mutlu, Associate Professor, Computer Sciences

C. David Page Jr., Professor, Biostatistics and Medical Informatics

© Copyright by Fatemah Panahi 2017
All Rights Reserved

*To my kind mother
my inspiring father
my beloved husband, Majid
our gorgeous son, Mohammad-Reza*

ACKNOWLEDGMENTS

In the Name of Allah, the Beneficent, the Merciful

Above all, I am thankful to Allah (God). I am thankful to him for all the blessings in my life. This work was not possible without his will and support.

It is my honor that Professor Jeffrey Naughton is my advisor for my graduate work and this thesis. I want to thank him for giving me the chance to work under his supervision, and for advising me through this journey. From Professor Naughton, not only I learned how to perform research, but also I learned many life lessons. His patience and positive attitude was always a reassurance through this long journey. He went above and beyond expectations to support me in any ways that he could.

I also want to thank Professor AnHai Doan, for he was like my second advisor and was involved in some way in most of this thesis. I want to thank both Professor Naughton and Professor Doan for their enormous encouragement and support as I was going through big changes in my life with our son's birth, searching for jobs, as well as finishing up my thesis.

I thank my other committee members, Professor David Page, Bilge Mutlu, and Paris Koutris, for kindly accepting to serve on my committee and for their precious time, input, and feedback on this thesis.

I thank Professor Mary Vernon, for her advice throughout my PhD years. I would go to her when I had to make hard decisions, and as an external eye she helped me focus on what is most important. I thank Christopher Re for I improved my programming skills in the limited time I worked with him.

I thank the students at UW-Madison, specially the students in the databases group. I enjoyed working with them day-to-day, and they provided valuable input to my research, papers, and presentations. It is not possible to name all these students but I would like to name a few. I thank Arun Kumar, Wentao Wu, Yueh-Hsuan Chiang, Ian Rae, Xi Wu, Chaitanya Gokhale, Sanjib Das, Pradap Konda, Adel Ardalan, Paul Suganthan, Haojun Zhang, Ce Zhang, Feng Nui, and Shishir Prasad. Special thanks to Thanh Do for his advice in preparing for job interviews. I also thank the group of women in the Computer Sciences department. I really enjoyed interacting with them around the department as well as in WACM events (UW-Madison's chapter of women in computing).

The Computer Sciences department was my home for my undergraduate, graduate, and PhD work. I spent significant amount of my life in this department. I thank all the professors who taught me or supported me in any form, and all the administrative staff of the department.

Throughout these years at Madison, I was lucky to have many friends who made my life pleasurable in these years. I want to thank them all.

I owe my family for my success. I thank my parents who raised me, taught me value of knowledge, and always encouraged me to learn more. I thank my older brother and sister for their support. I thank my parents-in-law for their support of my studies. I am especially indebted to my husband for his enormous support throughout these years. He believed in me even when I was discouraged and helped me stay on track for finishing my PhD. He stayed by me for all these years in Madison even though he had better career opportunities elsewhere. I was extremely happy being with him and he was the most source of encouragement for me. I also want to thank our son, for bringing hope and freshness to our lives after this long journey.

CONTENTS

Contents iv

List of Tables vi

List of Figures vii

Abstract ix

- 1 Introduction 1**
 - 1.1 *What is Entity Matching (EM)?* 1
 - 1.2 *The Entity Matching Process* 2
 - 1.3 *Human-Centric Entity Matching* 6
 - 1.4 *Debugging Entity Matching* 7
 - 1.5 *Related Work* 9
 - 1.6 *Contributions of the Thesis* 12

- 2 Experience with Real and Abstract Analysts 13**
 - 2.1 *Introduction* 13
 - 2.2 *An End-to-End Entity Matching Tool for Analysts* 14
 - 2.3 *An Abstract Model of an Entity Matching Task for Analysts* 36
 - 2.4 *Insights from Experience with Real and Abstract Analysts* 55

- 3 Towards Interactive Debugging of Rule-based EM 56**
 - 3.1 *Introduction* 56
 - 3.2 *Related Work* 58
 - 3.3 *Motivating Example* 59
 - 3.4 *Preliminaries* 61
 - 3.5 *Early Exit + Dynamic Memoing* 63
 - 3.6 *Optimal Ordering* 68
 - 3.7 *Incremental Matching* 78
 - 3.8 *Experimental Evaluation* 82
 - 3.9 *Conclusions* 90

- 4 Debugging Entity Matching Data sets 93**

4.1	<i>Introduction</i>	93
4.2	<i>Categories of Inconsistency in Data sets</i>	94
4.3	<i>A Framework for Finding and Resolving Inconsistencies</i>	98
4.4	<i>Experimental Evaluation</i>	116
4.5	<i>Discussions</i>	128
4.6	<i>Related Work</i>	130
4.7	<i>Conclusions</i>	132
5	<i>Conclusions, Limitations, and Future work</i>	134
5.1	<i>Conclusions</i>	134
5.2	<i>Limitations and Future work</i>	134
	<i>References</i>	136

LIST OF TABLES

2.1	Overview of student projects.	33
3.1	Notation used in cost modeling and optimal rule ordering study.	69
3.2	Real-world data sets used in the experiments.	83
3.3	Feature computation costs.	85
4.1	Data sets for evaluating effectiveness of rankings.	120
4.2	Analyst performance with different rankings with original data sets.	121
4.3	Analyst performance with different rankings with synthetic data sets (part 1).	122
4.4	Analyst performance with different rankings with synthetic data sets (part 2).	123
4.5	Quality of classifiers before/after cleanings.	127
4.6	Data sets for evaluating cleaning impact and spatial blocking.	128

LIST OF FIGURES

1.1	A typical entity matching process.	2
2.1	Workflow of analysts in our prototype.	15
2.2	A sample rule set with 2 rules.	17
2.3	A screenshot of statistics for table.	21
2.4	A screenshot of per-attribute statistics.	22
2.5	A screenshot of the search and query features.	23
2.6	A screenshot of the matching summary view.	26
2.7	A screenshot of the matching details view.	27
2.8	A screenshot of the skyline functionality.	29
2.9	A screenshot of the find by feature and threshold functionality.	31
2.10	Performance of simple analyst on consistent data set.	47
2.11	Performance of threshfinder analyst on consistent data set.	47
2.12	Performance of threshfinder analyst on consistent dataset after injecting non-matching pairs.	50
2.13	Number of rules for simple analyst with consistent dataset.	50
2.14	Number of rules for threshfinder analyst with consistent dataset.	50
2.15	Performance of simple analyst on inconsistent data set.	51
2.16	Performance of threshfinder analyst on inconsistent data set.	52
2.17	Performance of consistent analyst on inconsistent data set.	54
3.1	A typical matching workflow for analysts.	57
3.2	Motivating example.	60
3.3	Run time for different matching function evaluation approaches (Prod- ucts data set).	84
3.4	Sample rules from random forest.	84
3.5	(A) Actual versus estimated run time. (B) Run time as we increase number of pairs. (C) Run time as we incrementally add rules.	87
3.6	Incremental run time for matching function changes.	89
3.7	Run time for different matching function evaluation approaches (all data sets).	91
4.1	Example of incorrect label.	95

4.2	Example of missing feature.	96
4.3	Labeled sample illustration.	99
4.4	Framework for finding and resolving inconsistencies.	100
4.5	Median rank aggregation example.	104
4.6	Spatial blocking illustration.	110
4.7	Average size of random forest trees before and after cleaning.	126
4.8	Cleaning operations proposed for each data set.	127
4.9	Exhaustive search versus spatial blocking.	128

ABSTRACT

Entity matching (EM) is the problem of finding data records that refer to the same real-world entity. For example, the two records (Matthew Richardson, 206-453-1978) and (Matt W. Richardson, 453 1978) may refer to the same person. It is an important data integration problem with many applications such as in e-commerce, healthcare, and national security. Recent work on entity matching has focused on using machine learning and/or crowdsourcing in order to improve accuracy and/or scale the current matching solutions despite the fact that this task is typically done with a human analyst in the loop. Therefore, in this thesis we propose to work on solutions that acknowledge that humans are in the loop for completing an entity matching task. We focus on debugging of entity matching, which is an iterative process by which an analyst improves matching quality. Hence the title, “Human-Centric Debugging of Entity Matching”.

We build an end-to-end matching system and experiment with it in an e-commerce setting as well as with students in a graduate data modeling course at UW-Madison. We also develop an abstract model of the entity matching problem for an analyst to understand what makes an entity matching problem hard for an analyst. The insights learned in the above work lead to the following works in the rest of the thesis: First, we focus on debugging rule-based matchers and we attempt to make it an interactive process by which an analyst can quickly iterate and find a high quality matcher. We show that by optimally ordering the rules as well as incrementally running the matcher on top of previous matching output we can decrease runtime significantly. And second, we focus on debugging of entity matching data sets. We develop a framework to help an analyst quickly find and resolve inconsistencies in a data set. We experiment with seven real-world data sets and demonstrate the effectiveness of our framework in finding inconsistencies.

1 INTRODUCTION

1.1 What is Entity Matching (EM)?

Entity matching (EM) is the problem of finding data records that refer to the same real-world entity. For example, the two records (Matthew Richardson, 206-453-1978) and (Matt W. Richardson, 453 1978) may refer to the same person, and (Apple, Cupertino CA) and (Apple Corp, California) refer to the same company. Entity matching is a crucial task for data integration and data cleaning. Therefore, it has received significant attention and is becoming increasingly important in industry and among data enthusiasts. For overview surveys and a comparison of different matching platforms see [16, 23, 29, 46].

There are many use cases for matching in industry. Let us take Walmart as an example. Walmart may want to find matching products in competitor sites (for example, Amazon) in order to determine the price for a product. This is called price matching. Introduction of online marketplaces has expanded the use case for matching as well. In an online marketplace, a larger company such as Walmart sells items from smaller vendors on its online shopping website. In this scenario, different vendors may upload the same item to the marketplace, and these duplicates need to be identified. Also, companies are interested in selling products stocked at their stores through their online website. This requires matching products from the online product database with the store's product database. Finally, matching is used for data enrichment. For example, suppose a vendor has uploaded a TV to the marketplace without specifying the exact resolution. Walmart can partner with a data provider such as CNET that has complete information about that TV and fill in the missing value. This requires matching between the Walmart product database and the CNET data provider's database.

Given that today data is being collected in every occupation and field, data enthusiasts are becoming increasingly interested in entity matching [24]. *Data enthusiasts* are individuals who do not major in Computer Sciences but are interested in analyzing data and making conclusions about it. For example, a journalist may want to match two lists of donors for an election campaign and remove duplicates, or find donors that contributed to different elections.

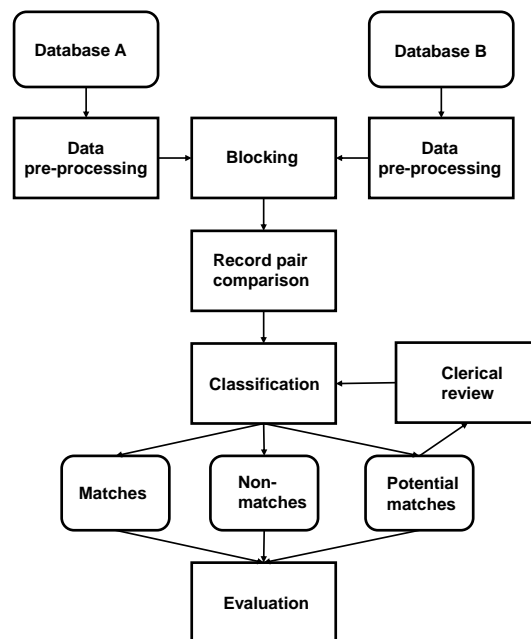


Figure 1.1: A typical entity matching process [9]

Other use cases of matching include a national census, where one would be interested in generating longitudinal datasets, by matching census data that have been collected at different instants in time (5 or 10 year intervals). Matching is also very important in the health sector, where different institutions may record data about the same individual. Another use case of matching is in national security, where different states can share information about criminals that commit crimes across states [9].

So far we have described the entity matching problem and some of its use cases in an attempt to demonstrate the importance of the problem. We will describe the steps involved in the entity matching process in the next section.

1.2 The Entity Matching Process

A typical entity matching process is shown in Figure 1.1 [9].

The input to this entity matching process are two tables to be matched. The entity matching output is the matching record pairs from the two tables. In the following subsections we explain different steps in the entity matching process (see [9] for more detailed descriptions).

1.2.1 Data Pre-processing

Data stored in the two input tables can vary in format, structure, and content. For example, in one table date of birth may be stored in one attribute and in the other table in multiple attributes corresponding to day, month, and year. Therefore, data needs to be cleaned and standardized before it can be used for matching. The goal of this step is to make sure that attributes used for matching have the same structure, and their content follows the same formats [9]. It has been shown that this is a crucial step to successful entity matching [26]. The major steps involved in data pre-processing are as follows [9]:

1. Remove unwanted characters and words.
2. Expand abbreviations and correct misspellings.
3. Segment attributes into well-defined and consistent output attributes.
4. Verify the correctness of attribute values.

1.2.2 Blocking

Given table A with m records, and table B with n records, there are $m \times n$ potential matches. Basically, each record from one table needs to be compared with all records in the other table to allow us to find all the matching pairs of records. However, even with moderate size tables that have tens of thousands of records the total number of potential matches could be very large. Suppose the two tables have one million rows each. The total number of potential matches will be one trillion. Even if we could do 100000 comparisons per second, it will take 116 days to compare these two tables.

The majority of the comparisons will be between records that are clearly non-matches. Through the blocking step we remove clear non-matches from all potential matches and reduce the number of comparisons. For example, suppose that each

product has a category attribute (for example, clothing and electronics). We could assume that products from different categories are clear non-matches. This will reduce the task to finding matching products within the same category.

We refer to the set of potential matches left after the blocking step as the “candidate record pairs” throughout the rest of this document.

1.2.3 Record Pair Comparison

In order to determine the overall similarity of two records in the candidate record pairs, we need to do detailed comparisons between those records. Record pair comparison is done in the following steps:

1. **Select attribute pairs:** A subset of the *attribute pairs* are selected to be used for matching. For example, product title from the Walmart table and product title from the Amazon table create an attribute pair. The intuition is to select attribute pairs such that the more similar the two records are across the selected attribute pairs, the more likely they refer to the same real-world entity.
2. **Select features:** For each attribute pair one or more features is created. A *feature* is specified by an attribute pair and a *similarity function*. For the above example, the Walmart and Amazon title attribute pair may be compared using the “Jaccard similarity function”. A similarity function typically generates a number in $[0, 1]$ where 0 means completely different and 1 means exactly the same. We call this number the *similarity score*.
3. **Calculate comparison vector:** For each of the candidate record pairs, the similarity score for each feature is calculated. We call this set of similarity scores the *comparison vector* for that record pair.

The comparison vector for all the candidate record pairs is the input to the next step, classification.

1.2.4 Classification

In this step we classify the compared record pairs to two or three classes [9]. In the two-class case, each compared record pair is classified as a *match* or a *non-match*. A record pair is classified as a “match” if the records refer to the same real-world

entity. If the records do not refer to the same real-world entity, then the record pair is classified as a “non-match”. Also, if the record pairs were removed in the blocking step, they will be classified as non-matches without being compared. In some classification methods there is a third class called *potential match*. These are record pairs for which we are not sure about the classification outcome. Therefore, a manual *clerical review* is needed to determine the correct match status of those record pairs.

There are many approaches to classification. For example, supervised learning, active learning, probabilistic classification, clustering-based approaches, and rule-based classification are all approaches that have been used for classification (see [9] for details of each approach).

1.2.5 Evaluation

To evaluate the quality of the entity matching output, we need to have ground-truth data, which is also known as *gold standard*. Ground-truth data should contain the true match status of all known matches between the two input tables.

Creating the gold standard is a time-consuming process. For each record pair, we need to identify the true match status manually. This is called *labeling* the record pair. If the number of candidate record pairs is very large, this may be infeasible with respect to our time constraints. Therefore, in many cases a sample of the candidate record pairs is selected and labeled, which we call the *labeled sample*. This sample is then used for evaluation purposes. With this approach, the hope is that if we can do high quality matching on the labeled subset of candidate record pairs, all the matching output should have reasonable quality.

Of course, if we have ground-truth data for the entire dataset and we do not expect to see any new data, we would be finished, and there would be no need for any additional matching. However, as we mentioned in many cases we will not be able to label the complete data set. Also, usually when we develop a classifier on existing data, the ultimate goal is to automatically classify new records that we encounter in the future using this classifier.

The standard measures for evaluation are *precision* and *recall*. Let n be the total number of candidate record pairs that should match. Assume the classifier makes

predictions for p pairs and out of those, q pairs are correctly predicted to match. Then precision is q/p and recall is q/n .

Basically, “precision” measures how many of the classified matches actually refer to the same real-world entity. “Recall” measures how many of the real-world entities that appear in both tables were correctly matched.

Normally there is a trade-off between improving recall and precision. We could get to 100% recall by simply classifying all candidate record pairs as matches. But that will lead to very poor precision. Similarly, we can simply get to 100% precision by classifying all candidate record pairs as non-matches. This will lead to 0% recall. Therefore, another measure called *F1 score* is usually reported along with precision and recall. The F1 score considers both precision and recall and is formulated as the harmonic mean of precision and recall:

$$F1 = 2 \times (\text{precision} \times \text{recall}) / (\text{precision} + \text{recall})$$

After the quality measures are calculated using ground-truth (or the labeled sample), they are compared with quality requirements set for matching. In some cases very high precision or recall is required. An example would be price matching between two vendors. In this case, if a product is incorrectly matched with a different less expensive product and the prices are matched, this will lead to profit loss. Therefore, for this example the matching output must be precise. If the quality does not meet the requirements, the problem areas in the matching process are identified and modified, and matching is run again. The quality is iteratively improved to satisfy the set requirements.

1.3 Human-Centric Entity Matching

Recent work on entity matching has focused on using machine learning and/or crowdsourcing in order to improve accuracy and/or scale the current matching solutions [22, 37, 38, 44, 45]. This is despite the fact that currently in industry this task is typically done by non-technical human analysts [7]. Analysts are easier to hire than the more technical developers and therefore more of them can be hired. Also, they can be trained for the needs of the business and therefore are more reliable than the crowd in crowdsourcing platforms. Furthermore, there is an increasing

need for humans to be able to do matching among *data enthusiasts*: non-technical individuals in various occupations who are interested in analyzing data and making conclusions about it. Currently, it is very hard for these individuals to do matching unless they know programming or pay to use crowdsourced solutions [22].

Therefore, in this thesis we propose to work on solutions that acknowledge that humans are in the loop for completing an entity matching task. Consequently, we aim to build an entity matching system that helps non-technical analysts do entity matching quickly and with high quality.

When focusing on the human aspect of the entity matching problem, these questions arise: how do we make a good entity matching system for analysts to use? What kind of classifier most suits the constraints of a human analyst? How can we guide analysts in the entity matching process? What is it that makes an entity matching task hard or easy for analysts? What is it that makes entity matching time-consuming or labor-intensive for the analysts?

To gain a better understanding of the answers to these questions we built a prototype of an entity matching system and experimented with it in an e-commerce setting as well as with students from a data modeling graduate course in UW-Madison. We also designed an abstract model of the entity matching problem for the analysts to gain insight on the parts of the entity matching problem that are challenging for the analysts. We describe the system and the abstract model in Chapter 2 of this thesis.

1.4 Debugging Entity Matching

From our experience with the entity matching system and feedback from real analysts and students that tried the system, we observed that entity matching is an iterative process where an analyst creates a matcher and then modifies the matcher and/or input tables many times to improve the quality. We refer to this process as *debugging*.

This process can be very time-consuming and labor-intensive for the analyst. Specifically, we noticed that when using a rule-based matcher with many rules each iteration can take a long time to run and thus waste analyst time waiting for the matching output. This is despite the fact that the analyst only modifies the matcher slightly at each iteration. Therefore, in Chapter 3 we attempt to find a rule ordering

such that matching is done faster. Furthermore, we investigate re-using matching output from previous iterations and running matching incrementally to make this process more interactive for the analyst.

We also developed and experimented with an abstract model of the entity matching problem for analysts. We noticed that for some data sets it was very easy for the analyst to come up with a small set of rules that lead to high quality matching output. For some other data sets, no matter how many rules the analyst wrote, she was not able to achieve high quality matching. Basically, we found that some data sets are easy to match and some are hard to match for the analyst. We define the notion of consistency for an entity matching data set, which intuitively means that matching pairs should have higher similarity values than non-matching pairs, and we show that consistent data sets are easy to match whereas inconsistent data sets are hard to match for analysts.

In Chapter 4, we follow up on the notion of consistency. We categorize different sources of inconsistency in a data set and discuss how the analyst should react to such issues. We further propose a framework for finding and resolving inconsistencies in a data set. In a sense, the analyst debugs the entity matching data sets using this framework. In this framework, we rank the pairs of records in the labeled sample such that the ones that are more problematic are ranked higher. We then present this ranking to the analyst and she interactively inspects the pairs and proposes cleaning operations for the data set. The ranking is re-generated and the process continues until the analyst inspects all record pairs or she runs out of time. We find that there are multiple approaches for ranking the record pairs and each of them can find a different set of problematic pairs. We propose two such approaches and we show that a hybrid ranking that aggregates rankings from multiple approaches can help the analyst find more issues in the data set.

In summary, in this thesis we consider two issues with human-centric debugging. First, we consider debugging a rule-based matcher, and try to make this process more interactive for the analyst. And second, we consider debugging entity matching data sets, and help the analyst to quickly find and resolve issues in the data set.

1.5 Related Work

1.5.1 Entity Matching Systems

Konda et al. [33] extensively review existing non-commercial (for example, D-Dupe, DuDe, Febrl, Dedoop, Nadeef) and commercial (for example, Tamr, Data Ladder, IBM InfoSphere) entity matching systems as of 2016, and introduce Magellan, a new kind of entity matching system.

Each of these systems provide a set of unique functionalities. For example, all of these systems have support for blocking and matching, some of them have support for data cleaning and exploration, but most of them lack support for other steps of the matching pipeline, including debugging data sets and matching, that is discussed in this thesis.

Techniques described in this thesis are inspired by our experience with an e-commerce company as well as many graduate students that have tried out Magellan for various entity matching data sets through courses at University of Wisconsin-Madison. Thus, we feel that they could be used to enhance any entity matching system. In particular, we are planning to incorporate these in Magellan. Magellan is the most recent of these systems, and it is designed to be extendable, such that new techniques can be easily added to the system to improve it. Furthermore, a main focus in Magellan is to minimize user burden and provide a rich set of tools to help users do each entity matching step, which aligns well with the goals and contributions of this thesis.

1.5.2 Human-Centric Entity Matching

Chiticariu et al. [7] conducted a survey and show that while rule-based information extraction dominates the commercial world, it is mostly regarded as a dead-end technology by the academia. Their results show that while (as of 2013) 67% percent of large vendors have implemented a rule-based information extraction solution, only 3.5% of the Natural Language Processing (NLP) papers published between 2003 and 2012 focused on solely rule-based system. They argue that this gap between academia and industry needs to be reduced so that academic research is most valuable to industry.

Based on our experience with an e-commerce enterprise we highly suspect that this is similar to the gap between academia and industry for entity matching research. Firstly, we found that similar to NLP, rule-based approaches are used in the industry due to the requirement for high precision matching outputs, while they are typically not the focus in the research community. Furthermore, even though there is typically a human in the loop for performing entity matching, most entity matching research is focused on improving accuracy of matching and blocking steps (at least 96 papers from 2009-2014 [33]) and does not address this fact. This highly motivates our research on human-centric entity matching.

1.5.3 Rule-based Entity Matching

Different works have focused on maintaining a rule-based classifier and improving the output of the classifier. [42, 43] exploit previous materialized Entity Resolution (ER) results to save redundant work when the rules change. [41] proposed using negative rules to disallow inconsistencies in the entity resolution solution. A consistent solution is then arrived using guidance from a domain expert. These works assume that a rule-based classifier already exists and address the problem of maintaining it. In this thesis, we instead focus on the problem of creating the rule-based classifier and assume a “human analyst” will iteratively generate the rule set.

1.5.4 Crowdsourcing

Recently, crowdsourced entity matching has received increasing attention in academia and industry (CrowdFlower, CrowdComputing, and SamaSource).

In academia, works such as [12, 37, 38] use the crowd to verify the matching output. [44] finds the best questions to ask the crowd, and [45] investigates the impact of user interface (UI) when asking questions from the crowd. Corleone [22] takes this concept a step further and introduces Hands-Off Crowdsourcing (HOC). HOC attempts to eliminate analysts by crowdsourcing all of the entity matching pipeline.

These solutions demonstrate that in certain scenarios we can crowdsource all or part of the entity matching task. However, our experience with an e-commerce enterprise suggests that there may always be cases where a trained non-technical

analyst needs to manually design and maintain a matching classifier. For example, in the case where extremely high precision matching output is required, current crowdsourced solutions cannot be fully trusted, and analyst involvement is required. Furthermore, there are always cases where these approaches fail to produce the correct output, and thus an analyst has to devise new rules on top of the classifier or modify the automatically generated classifier to get high quality.

In industry, companies such as CrowdFlower use crowdsourcing for collecting, cleaning, and labeling data. The customers for these companies are typically enterprises, businesses, and startups. It would not be feasible or would be very expensive for a data enthusiast to submit a one-time or small matching task to these companies. By taking a human-centric approach towards the entity matching problem, we make it easier for such users to perform entity matching on their own.

1.5.5 Debugging Entity Matching

Most entity matching solutions do not consider the issue of debugging in various steps of matching or are limited to showing what rules have fired for a matching pair [33]. Magellan has support for debugging the blocking step as well as debugging different kind of matchers such as decision tree and random forest. In this thesis we focus on debugging entity matching data sets as well as reducing the time for debugging rule-based matchers.

1.6 Contributions of the Thesis

More concretely, our accomplishments and contributions regarding this thesis are as follows:

- Developed and experimented with an entity matching system as well as an abstract model of the entity matching problem for analysts to gain insight on issues for human-centric debugging of entity matching. This is presented in Chapter 2.
- Developed algorithms for optimal ordering of rules as well as incremental matching to enable interactive debugging of rule-based matchers. This is presented in Chapter 3.
- Developed a framework for debugging entity matching data sets such that an analyst can quickly find and resolve inconsistencies. This is presented in Chapter 4.

2 EXPERIENCE WITH REAL AND ABSTRACT ANALYSTS

2.1 Introduction

Recall that what we hope to ultimately achieve through techniques proposed in this thesis is to reduce the time and effort that analysts spend on each entity matching task and help them achieve high quality matching faster. Therefore, in this chapter we try to develop an understanding of what steps in the matching process take the most time from the analysts and what is it that makes a data set hard for them for matching. To do so we experiment with real and abstract analysts.

To gain experience with real analysts, we developed an end-to-end matching system to help us better understand how analysts interact with a matching problem, and what steps in the process are labor-intensive. With real analysts we mean the author with available public data sets, analysts in an enterprise e-commerce setting with industry data sets, and students at a data modeling graduate course at UW-Madison. These students first generated an entity matching data set through crawling the web, and then performed matching on it. This system was end-to-end, meaning that analysts went through all the steps of entity matching in one tool, from browsing, to matching, to debugging. We explain this system in more detail in Section 2.2.

This experience helped us identify a key bottleneck in performing matching efficiently for analysts. We observed that finding and defining a high quality matcher is iterative by nature, and analysts have to refine their matcher many times before reaching their satisfied quality. This is despite the fact that current research is mostly focused on improving matching time when matching is done only once after a matcher is decided on. Therefore, in Chapter 3 we investigate how to reduce run time of each iteration for the analysts to enable her to find a high quality matcher faster.

We also observed that it is very easy to define a high quality matcher for some data sets, while it is very hard for other data sets. We further developed and experimented with a set of simple abstract analysts to better understand what is it that makes a matching problem hard for analysts. In defining our abstract analysts, our goal is not to fully model the entity matching problem, as we believe this is not possible due to the many variations in workflows that analysts can follow. On

the other hand, our goal was to develop a better understanding of the matching problem. In fact, we were able to get valuable insight with this simple model. We further discuss our model and insights in Section 2.3.

In particular, we noticed that in some data sets, there are certain pairs of records that are inconsistent with respect to other pairs of records. For example, we found some matching pairs that actually had lower similarity scores than non-matching pairs. Data sets that were free of inconsistencies were very easy to match with high quality while inconsistent data sets are harder to match with high quality. In Chapter 4, we investigate how we can help the analysts to identify and rectify such inconsistencies in a data set quickly.

In summary, through experimenting with real and abstract analysts, we were able to identify key issues in the matching process that decrease the time and effort of matching for analysts. This motivated us to pursue these issues more deeply which lead to works presented in Chapters 3 and 4.

2.2 An End-to-End Entity Matching Tool for Analysts

When we observed analysts perform entity matching in an e-commerce setting we noticed that the workflow of matching for analysts may involve many tools, as all steps are not covered under the same tool. For example, they may use SQL queries to generate statistics and get an understanding of the attributes in an input table. They use Excel to browse the input tables. They may write rules in a text file using a business-specific rule language. They upload the rule file in a user interface for matching and then run matching. The output is stored in a text file that they can browse through a text editor. There are two problems that we can identify here:

1. Using different tools and switching between them for different parts of the pipeline wastes a lot of time for analysts.
2. Each of these separate tools such as SQL and Excel are not tailored for matching tasks and therefore do not provide exactly what analysts need nor can they be extended to provide the features analysts need.

At the time of this thesis, the few entity matching systems that are around are limited in their functionality and they do not support all steps of the matching

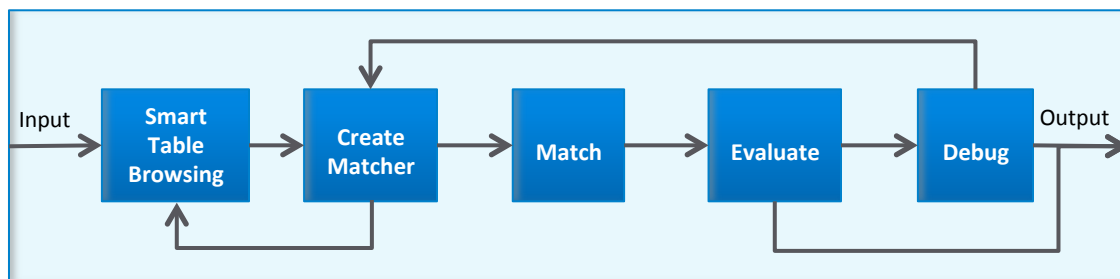


Figure 2.1: Workflow of analysts in our prototype of end-to-end matching system.

workflow. We explain these systems further in detail in Section 2.2.7. Therefore, we worked towards building a prototype of an end-to-end entity matching system.

Figure 2.1 shows the workflow that we considered for the analysts in our prototype. An analyst first tries to develop an understanding of the input tables. We help the analyst through this process in our browser module. Then the analyst creates a set of initial rules. In our prototype we only consider a specific type of matcher, a rule-based matcher. We will describe our justification for using a rule-based matcher as well as details of the mathcher in Sections 2.2.1, 2.2.2. We support rule creation/-modification through a graphical interface. The analyst then runs matching. We also support matching through our tool. Normally, the analyst is provided with a labeled subset of the candidate record pairs or ground-truth to evaluate the quality of the rule set (See Section 1.2.5 for details about how evaluation is performed). She evaluates the rule set using the labeled record pairs. We support this step through the evaluation module in our tool. It is common in industry that certain business requirements are set for matching quality. For example, precision must be greater than 95 percent and recall as high as possible. If the matching output does not meet the set of requirements, the analyst tries to find out why, fix it, and run matching again. We help the analyst to find out exactly why a certain error occurred through the debug module. Through this module, we also show her record pairs that help her quickly improve the current rule set to achieve higher precision and/or recall.

As described above, in this tool we support all the steps in this matching pipeline, which we call “modules” hereafter. In each module, we identified the operations that analysts commonly perform and implemented them. In the interest of space we will omit the details of create/modify rules, match, and evaluate modules as the operations are fairly obvious to the reader from the explanations in Chapter

1. In the next sections, we will provide details regarding the Browser and Debug modules which were in particular developed by the author.

I should note that I began the implementation of an early prototype of the end-to-end rule-based matching system in the 2013-14 academic year. The current version of the tool was implemented during an internship at WalmartLabs in the summer of 2014. I did the primary coding for the browser and debug modules, Sanjib Das did the primary coding of the other modules, and we collaborated to create the end-to-end system under supervision of Professor AnHai Doan.

We experimented with this tool in an e-commerce setting as well as with graduate students at University of Wisconsin-Madison and found that it can help analysts/students quickly perform matching with high quality. The feedback that we got from analysts/students were very positive. We will discuss this feedback in Section 2.2.6.

Our experience with building an end-to-end entity matching system gave us insights on key steps in the pipeline that are time-consuming and labor-intensive for the analysts. We try to address some of these issues in the remaining chapters of this thesis. Through this prototype we were also able to identify key issues with current entity management systems. Using these insights, the entity matching group at UW-Madison subsequently re-designed our prototype to a full-fledged entity management system, called Magellan [33], which is open-sourced and is being used by students as well as our industry partners. We will discuss Magellan in more detail in Section 2.2.8.

2.2.1 Why Rule-based Classification?

There are many classification methods that we can use for entity matching. For example, one can use supervised or unsupervised machine learning approaches or she can choose to write a set of rules for matching. For our prototype of an end-to-end matching system, we decided to explore a rule-based classification method for the following reasons:

First, rules are easy to understand and debug for non-technical analysts. Consider a machine learning algorithm. Analysts need to become familiar with how the algorithm works, which may be very technical. Moreover, the input to the machine learning algorithm is training data. Thus, in order to change the matching output, analysts need to retrain the model. This is non-trivial for two reasons. First, it is not

$r1 : p1(\text{first name}) \geq 0.9 \wedge p2(\text{last name}) = 1.0 \wedge p3(\text{address}) \geq 0.7 \Rightarrow \text{Match}$
 $r2 : p1(\text{first name}) \geq 0.7 \wedge p2(\text{last name}) = 1.0 \wedge p3(\text{phone}) = 1.0 \Rightarrow \text{Match}$

Figure 2.2: A sample rule set with 2 rules.

guaranteed that enough training data is available for every instance. For example, suppose an analyst wants to make sure that the model does not match “Peter Pan peanut butter” and “Pan”. They need to add enough instances with the same pattern to the training data such that the algorithm can learn this pattern. These training instances may simply not be available. This is not the case with rules. If you show this example to an analyst, it is highly likely that she will be able to modify the rule set such that these two records do not match. Second, upon changing the input training data, there is no guarantee that the machine learning model and the matching output will change. Therefore, debugging a classifier generated by machine learning can be very hard.

Rules are not only intuitive and easy to understand but also can be easily customized to meet high precision/recall quality requirements. High quality requirements are very common in industry. For example, for a price matching application, extremely high precision is required. Suppose you incorrectly match the price of a more expensive product with a less expensive product from the competitor’s website. This can lead to thousands of dollars of lost revenue. Therefore, even if a coarse round of machine learning is used to quickly do matching, often manually created rules are used on top of it to fine-tune the results and achieve high precision.

2.2.2 Rule-based Classifier

We implemented a commonly used rule-based classifier in our end-to-end matching system where each rule set is a disjunction of a set of rules and each rule is a conjunction of clauses/predicates. An example set of rules is shown in Figure 2.2.

Rule set: The entity matching result is generated by evaluating a rule set for each candidate pair of records. If the rule set matches a pair, then it is a match, otherwise it is a non-match. A rule set is a disjunction of one or more rules. If at least one of the rules match a record pair, then the rule set matches that record pair. Otherwise, the rule set does not match the record pair. In Figure 2.2, the rule set is a

disjunction of 2 rules that determine if two records refer to the same person or not. The disjunction symbol is not explicitly shown here and is implicit from the context.

Rule: A rule is a conjunction of one or more clauses/predicates. A rule matches a record pair if all its clauses return true for that pair.

In Figure 2.2, for the first rule, if the similarity score for first names of the two records is greater than or equal to 0.9, the last names exactly match, and the address similarity score is greater than or equal to 0.7, the record pair is considered a match.

Clause/Predicate: Each clause/predicate is a comparison between a similarity score for an attribute pair and a threshold. For example, in Figure 2.2 the first clause of the first rule checks that the similarity score for the first names of the two records is greater than or equal to 0.9.

2.2.3 Helping Analysts to Develop a Classifier

Consider a situation where an analyst is given a set of candidate record pairs and asked to write a high-quality rule set. What an analyst typically does is to look through the candidate record pairs, find patterns for matching pairs, and write a rule for it. This is when she is trying to improve recall. The question now is: In what order should the analyst look through pairs? Should she just randomly go through the pairs? Now consider the situation where the analyst has already written a rule set, and would like to get an estimate of the precision of the rules. She certainly could randomly go through the matched pairs and find out if she is making precision errors. Again, the question is can we help her do better than randomly looking through pairs? Basically, we would like to give the analyst a direction to look at pairs when she is trying to improve precision or recall. The *Skyline* approach is designed to give this direction.

The main idea here is to compute a skyline of record pairs, that is, a set of record pairs such that no other record pair differs as much as or more than any of them in every attribute pair (when trying to improve precision) or that no other record pair is as similar as or more similar than any of them in every attribute pair (when trying to improve recall).

Intuitively, when an analyst is trying to improve recall, the skyline record pairs will be the most similar record pairs that she has not matched. These are the most obvious recall errors that the analyst would want to take care of first. Also, if ground-

truth data is not available she can get an idea of what the actual recall is by looking at the most similar pairs not matched. Intuitively, if the most similar items not matched are non-matches, then this is an informal confirmation that the rule set has a reasonably high recall and most obvious matches have been covered.

Similarly, when an analyst is trying to improve precision, the skyline record pairs will be the most different record pairs that she has matched. These are the most obvious precision errors that the analyst would want to take care of first. Also, if ground-truth data is not available she can get an idea of what the actual precision is by looking at these pairs. If these pairs are matches, then this is a confirmation that no obvious non-matches have been matched by this rule set.

We compute the skyline in two steps: (1) generating comparison vectors and (2) computing skyline vectors. We describe each step below:

Generating Comparison Vectors: Given 2 records in a candidate record pair, if record1 has m attributes and record2 has n attributes there are $m \times n$ possible attribute pairs from which the analyst selects a subset of size u , $\{P_1 \dots P_u\}$ as relevant for matching.

The comparison vector $V_{i,j}$ of two records r_i and r_j is denoted as $\langle v_1 \dots v_u \rangle$ where each $v_k \in V_{i,j}$ is computed by a similarity function over the attribute values for attribute pair P_k .

Skyline Vectors: Assume that for each similarity value computed above higher values indicate higher similarity.

Given any two comparison vectors A and B , we say A dominates B ($A \succ B$) if and only if:

$$\forall k \in [1, u], a_k \geq b_k \quad \wedge \quad \exists j \in [1, u], a_j > b_j$$

Given a set SV of vectors, a vector V_i is a skyline vector if V_i is not dominated by any other vectors in SV :

$$\nexists V_d \in SV, \text{ s.t., } V_d \succ V_i$$

A skyline record pair is the pair of records corresponding to a skyline vector $V_{i,j} \in SV$. We use the SFS algorithm [8] to find the skyline vectors given a set of comparison vectors.

The skyline record pairs can be generated iteratively in case an analyst would like to browse through more pairs. To produce more record pairs, the generated

skyline record pairs are removed from the set of vectors, and the next set of skyline record pairs are generated. If we continue this process it is analogous to sorting the record pairs from most similar to least similar, where most similar is defined by the skyline approach.

The notion of a Skyline operator to assist analysts in entity resolution was originally proposed by Sun Chong in his PhD thesis [36] and we implemented it as part of the debugging module of our end-to-end system to help analysts develop rule-based classifiers.

2.2.4 Browser Module

A common matching workflow starts by an analyst browsing the data and developing an understanding of the data. Therefore, the first goal of the browser module is to help the analyst quickly develop an understanding of the dataset. We identified operations that are commonly performed by analysts to understand the data and incorporated them in this module. For example, with the “view table statistics” operation, we help them quickly identify attributes that may be effective for matching (for example, attributes containing unique values) and attributes that have almost no value for matching (for example, attributes containing many missing values).

In our observations of analysts at work at WalmartLabs, it is very common for an analyst to do simple table manipulation and normalization on the input tables before writing any rule. In the browser module, we also support simple data manipulation in case the analyst finds it necessary. For example, the analyst may want to remove attributes that she decided have no value for matching from the table and save it as a separate copy. Another example is when the input table contains large number of rows, then she may want to take a sample of the table and work with the sample.

We have 2 types of operations in the browser module:

- **Data exploration operations:** An analyst performs these operations to develop intuition about the input tables. These operations include viewing statistics for table and attributes, sorting, searching, and querying attribute values.
- **Table manipulation operations:** These operations allow an analyst to do simple manipulations on the input tables such as taking a sample of table, removing rows/attributes from table, editing attribute values, opening a copy of table, and saving changes to table.

Rank	Attribute	Unique	Missing
1	isbn	100%	0.3%
2	upc	100%	99%
3	title	93.5%	0%
4	author	82.3%	13.9%
5	series	56.4%	83.7%
6	volume	15.2%	97.2%
7	pages	12.7%	28.6%

Figure 2.3: A screenshot of statistics for table.

Next we will explain each of the operations in detail.


2.2.4.1 Data Exploration Operations

View Statistics for Table Figure 2.3 shows a screenshot of table statistics as part of the data exploration operations.

Based on talking to analysts working on matching, we found that what an analyst usually does in the matching workflow after loading a table is to get an idea of all the attributes in the table. This operation was normally done through SQL queries on the table which would be very time-consuming. Therefore, integrating this operation in the end-to-end matching tool will make the workflow faster for analysts.

The analyst is interested to know which attributes are useful for matching and what attributes have almost no value for matching. Two statistics that are frequently considered are the percentage of unique values and the percentage of missing values for each attribute. Intuitively, attributes that have a high percentage of unique values are considered good candidates for matching (for example, “isbn” in the screenshot). This is not always true but the analyst wants to be aware of these attributes. Also, attributes that are mostly missing, for example “upc” in the above screenshot, cannot be used for matching.

Therefore, we present the “percent unique” and “percent missing” statistics to the analyst, and we rank the attributes based on the following heuristic to help the analyst pinpoint good attributes for matching: Sort the attributes from higher



Statistics for binding	
Unique: 0.11% Missing: 0.25%	
Paperback	3998
Trade Cloth	3373
E-Book	1512
Perfect	178
UK-B Format Paperback	120
CD/Spoken Word	117
UK-Paperback	105
Library Binding	71

Figure 2.4: A screenshot of per-attribute statistics.

percent unique to lower percent unique; breaking ties by lower percent missing values.

View Per-attribute Statistics Figure 2.4 shows a screenshot of per-attribute statistics as part of the data exploration operations.

Analysts normally want to get an idea of the values that appear in an attribute. For example, they may want to know the kind of bindings (book covers) that exist in one table and compare it to the types of bindings that exist in the other table. This is very important for normalization. For example, one data source may store “Hard cover” as “Trade cloth”. This will cause a problem for matching as these values will not match regardless of the fact that they refer to the same type of binding. To normalize the data, the analyst can request that all the “Trade cloth” values are changed to “Hard cover” or vice versa.

With this operation analysts can see all the values that appear in an attribute. For each attribute value, we show them the frequency of that value, and we sort the values from highest frequency to lowest frequency. This is because high frequency values are more important for analysts for matching. For the bindings example above, the two values with highest frequency values are “Paperback” and “Trade Cloth”.

Sort Attribute Values Another way an analyst explores the data is by sorting the values for an attribute. For example, she may be interested to group similar values

Facets and Filters

7 rows

Show 5 10 20 50

all

id	title	author	numAuthors	binding	publisher	isbn	pubYear	pubMonth
9780217092258	History of the Third French Republic	Wright, Charles Henry Conrad	1	Paperback	General Books	9780217092258	2010	3
9780217180979	Universit�s� Chinese : Universitaire Chinois,	Source: Wikipedia	1	Paperback	General Books	9780217180979	2011	8

Figure 2.5: A screenshot of the search and query features.

together by sorting the values for an attribute. This is exactly what this operation provides.

Search for a String in Attribute Values/Query Attribute Values Figure 2.5 shows a screenshot of the search and query features as part of the data exploration operations.

The operators that we discussed so far give an analyst a global view of an input table. However, in some cases the analyst is interested to look into records that satisfy a specific condition. With the search operator, we allow the analyst to search for a string in an attribute. This operation is type agnostic and will treat every attribute value as a string. With the query operation, she can do simple queries on the attributes and this operation takes into account the type of the attribute (for example, numeric). She may stack multiple search and query conditions to perform a conjunction of the searches. In the shown example, the analyst is interested to view all books published by the publisher “General Books” (search) after 2009 (query) which results in seven rows.

2.2.4.2 Table Manipulation Operations

Take a Sample of Table If the input table is very large, an analyst may prefer to first work with a sample of the table. With this operation, the analyst specifies the number of rows to sample and the table name to save the sample rows into. We will then take a random sample of the table and save the sample rows in the table specified by the analyst.

Remove Attribute/Rows from Table In some cases an analyst may want to remove certain rows or attributes from the input table. For example, if she finds that an attribute has many missing values she can remove that attribute from the table using this operation.

Edit an Attribute Value If an analyst sees that an attribute value is recorded incorrectly, using this operation she can edit that value to a new value that she inputs. Note that if the analyst sees many such mistakes, the normal workflow is to report that to the party who provided the table and she will not attempt to fix all the errors.

Open a Copy of Table Commonly, an analyst would like to retain the original table that was provided to her, and do her manipulations in another copy. With the “open copy” operation, we make an in-memory copy of the table that she is currently browsing and open it in a new tab for her to view.

Save Changes to Table This operation allows an analyst to persist table manipulations on disk. All the manipulations are stored in memory and persisted in disk only when the analyst explicitly asks for saving it.

2.2.5 Debug Module

A common workflow for matching consists of writing an initial set of rules, evaluating the quality, and iteratively improving the quality of the rule set until business requirements are satisfied. This can be a very time-consuming process. In fact, based on our discussions with analysts we anticipate that an analyst will spend most of her time trying to iteratively improve the quality. Therefore, it is critical to save analyst’s time in this step of the workflow. Correspondingly, the goal of the debug module is to help analysts quickly and iteratively improve the precision/recall of the rule set.

We identified a set of views and operations that help an analyst in quickly improving the quality and incorporated them in this module. In summary, the views and operations in this module help the analyst to:

- Identify exactly why a candidate record pair was matched or was not matched using this rule set.
- Quickly browse through the set of candidate record pairs using different filters (for example, view the precision/recall errors).
- Quickly modify existing rules to improve precision/recall.
- View the candidate record pairs that an updated rule would match, without doing the complete matching process that uses all the rules.

This module can run in different modes. If an analyst has access to the ground-truth or a labeled subset of candidate record pairs for the matching task, we can use the ground-truth, calculate precision and recall, give her additional features for filtering the candidate record pairs, and color code the candidate record pairs using their true match status. However, they can always run the debug module without the ground-truth, and they will have access to all the operations that do not need ground-truth information (Please See Section 1.2.5 for detailed explanation on how evaluation is performed).

Next we will explain each of the views and operations in the debug module in detail.

2.2.5.1 Summary View

Figure 2.6 shows a screenshot of the matching summary view as part of the debugging module.

This view gives an analyst a quick overview of the matching status. If ground-truth is available, the evaluation summary is presented. The evaluation summary includes precision, recall, and F1 score, which are common measures for calculating the quality of matching (See Section 1.2.5 for details of how these measures are calculated). We also show number of precision and recall errors, which are clickable links and will filter the candidate record pairs by precision/recall errors so that the analyst can zoom into problem areas quickly.

The matching summary section shows number of candidate record pairs and number of matched record pairs by this rule set, along with the total number of rules in the rule set. If there is no ground-truth available, this will give the analyst an intuition about the recall.

Matching Summary	
Evaluation summary:	
Precision:	0.74
Recall:	0.34
F1 score:	0.47
Number of precision errors:	6
Number of recall errors:	33
Matching summary:	
All pairs in the candidate set:	68
Total matches:	23
Total rules:	1
Rule summary:	
title author matches	23

Figure 2.6: A screenshot of the matching summary view.

In the rule summary section, for each rule we show how many candidate record pairs were matched by this rule. The rule summary is helpful for an analyst since she can identify rules that have a higher or lower effect on matching if they are matching many or only few of the candidate record pairs.

Similar to the evaluation summary, from both the matching summary and the rule summary the analyst can filter the candidate record pairs to view all the matched record pairs, or all the matched record pairs by a particular rule.

2.2.5.2 Matching Details View

Figure 2.7 shows a screenshot of the matching details view as part of the debugging module.

In order for an analyst to be able to debug the matching output, she needs to have an understanding of exactly why a record pair was matched or did not match and the options that she has for fixing the error. Therefore, for each candidate record pair shown on the debug page, we show them the record pair details and the matching details explained below.

bowker item		walmart item		Matching Details	Feature scores
id	9780226156439	id	4086892		
title	Bedtrick : Tales of Sex and Masquerade	title	The Bedtrick: Tales of Sex and Masquerade		
author	Doniger, Wendy	author	Doniger, Wendy		
numAuthors	1	numAuthors	1		
binding	Perfect	binding	Paperback		
publisher	University of Chicago Press	publisher	Univ of Chicago Pr		
isbn	9780226156439	isbn	9780226156439		
pubYear	2005	pubYear	2005		

Figure 2.7: A screenshot of the matching details view.

Record pair details For each record pair, we show the two records from the input tables and all the attributes for each of the records. If ground-truth is not available, the analyst can determine the match status of the record pair by inspecting all attribute values of the records. Otherwise, the record pair will be color coded to show the true match status of the pair in the ground-truth.

Looking at the record pair details also helps her to identify the attributes that she should use for improving precision and/or recall. For example, suppose two books that had different publication years were incorrectly matched and thus create a precision error. She will be able to see that the publication year for the two books is different and thus add a clause to her rule to make sure that the publication years of the two books match.

Matching details In order to fix a precision or recall error, an analyst should know why exactly the current rule set made a mistake about that record pair. If it is a recall error, why is it not matched by any of the rules that she devised? If it is a precision error, which rules incorrectly matched this record pair? This helps her to modify her rules such that she can improve precision/recall. We provide this information in the “Matching details” box.

In the matching details box, for each rule we show to the analyst if it matched the record pair or not. We then dig deeper, and for each clause in the rule show if it returned true for that record pair or not. Furthermore, for each clause we show them the threshold that the analyst has specified and the calculated score for this record pair. For example, if the threshold for title similarity score is set to 0.9 and

the calculated score is 0.1, this clause returns false for this record pair and thus the rule will not match it. This information identifies exactly why a record pair was matched or was not matched by a specific rule. Since we show these details for each rule in the rule set, the analysts can identify exactly what rules matched or did not match the record pair.

Once the analyst has identified why exactly a record pair corresponding to a precision or recall error was matched or was not matched she will try to fix the error. One common way of fixing the precision/recall errors is to use a different similarity function for an attribute pair (i.e., a different feature). For example, in the title similarity example mentioned above, the analyst may have chosen an incorrect similarity function for comparing titles such that the similarity score is very low even if the actual values look similar to the analyst.

Now, if the analyst decides that she needs another similarity function, we should help her pick one. Therefore, in the matching box we give her the option of browsing all the features available for this project and their score for this particular record pair. For example, suppose she wants to compare addresses. She may find that Jaccard similarity is not producing high numbers where 2 addresses actually match. When she browses through all the available features and their scores for matching pairs, she finds that Overlap similarity function can better identify matching addresses and will decide switch to using that feature for comparing addresses.

Note that by showing these alternative features up front, we will keep the analyst from doing many rounds of matching for identifying the right feature for an attribute pair. By showing the similarity scores for all the alternative features we try to take guesswork out of the loop as much as possible.

2.2.5.3 Filter by Matching Prediction

With this filter the analyst can zoom into:

- All the candidate record pairs that matched using the current rule set.
- All the candidate record pairs that were not matched.
- All the candidate record pairs regardless of their matched status.

This filter is very critical when ground-truth is not available. In order to keep an eye on precision, an analyst would be looking into all the candidate record pairs that

Figure 2.8: A screenshot of the skyline functionality.

she matched. In order to keep an eye on recall, she will look into all the candidate record pairs that she has not yet matched. Once she applied this filter, she can get further guidance on how to improve recall and/or precision using the “find skyline pairs” operation described below.

2.2.5.4 Filter by Rule

With this filter an analyst zooms into all the matches for a particular rule. This is again very important in case ground truth data is not available. Normally, the analyst will inspect these pairs to keep an eye on the precision of a particular rule. She can get further guidance on how to improve the precision of this particular rule using the “find skyline pairs” operation described below.

2.2.5.5 Filter by Precision/Recall Errors

With this filter an analyst zooms into precision/recall errors and tries to fix them. This will be the go-to filter in case ground-truth is available. She can get further guidance on how to improve precision/recall using the “find skyline pairs” operation described below.

2.2.5.6 Find Skyline Pairs

Figure 2.8 shows a screenshot of the skyline functionality as part of the debugging module.

The idea behind this operation is to quickly focus the attention of the analyst to the most obvious precision/recall errors so that she can modify the rule set to

fix these errors. The normal workflow for using the skyline operator is to use of the filter operations such as filter by precision/recall and then use this operation to further focus the attention of the developer to a few pairs that will help her quickly improve the quality. The analyst will have the option to view the most similar or least similar record pairs out of the filtered pairs. This operation will be helpful in a number of scenarios depending upon if ground-truth data is available or not.

Skyline use cases when ground truth not available

- The analyst filters by match status, views matching pairs, and selects least similar pairs. This will point the analyst to pairs that could be precision errors.
- The analyst filters by match status, views non-matching pairs, and selects the most similar pairs. This will point the analyst to pairs that could be recall errors.
- The analyst filters by rule, views pairs matched by this rule, and selects least similar pairs. This will point the analyst to pairs that could be precision errors for this particular rule.

Skyline use cases when ground truth is available

- The analyst filters by recall errors, selects the most similar pairs. This will focus the attention of the analyst to the most obvious recall errors.
- The analyst filters by precision errors, selects the least similar pairs. This will focus the attention of the analyst to the most obvious precision errors.

2.2.5.7 Find by Feature and Threshold

Figure 2.9 shows a screenshot of the find by feature and threshold functionality as part of the debugging module.

It is very common for analysts to modify the features and/or thresholds of the clauses in existing rules or create new rules in the debugging phase. One could say that this is the main point of the debug module.

Now, one workflow could be to decide on a change, make that change, run matching again, and view the evaluation results. However, matching is an expensive and sometimes time-consuming operation. There may be tens of rules that should be applied to hundreds of thousands of candidate record pairs in each matching

The screenshot shows a software interface for finding item pairs based on features and thresholds. On the left, there are two filter panels. The top panel is for 'title_jac' with a threshold of 0.6. The bottom panel is for 'author_jac' with a threshold of 0.6. The main area displays two columns of item details: 'bowker item' and 'walmart item'. The bowker item has id 9780226156439, title 'Bedtrick : Tales of Sex and Masquerade', author 'Doniger, Wendy', numAuthors 1, binding 'Perfect', publisher 'University of Chicago Press', isbn 9780226156439, and pubYear 2005. The walmart item has id 4086892, title 'The Bedtrick: Tales of Sex and Masquerade', author 'Doniger, Wendy', numAuthors 1, binding 'Paperback', publisher 'Univ of Chicago Pr', isbn 9780226156439, and pubYear 2005. On the right, the 'Matching Details' panel shows a tree structure with rules: 'title_author', 'title_jac >= 0.8', 'author_jac >= 0.8', and 'year_exact == 1'. Below this, there is another set of item details for a different pair.

Figure 2.9: A screenshot of the find by feature and threshold functionality.

step. This is despite the fact that the change that the analyst wants to try in each iteration of debugging is usually local to one rule. Moreover, in many cases after matching is done, the analyst may find out that the change that she proposed actually was not very effective or even reduced quality. Now, to revert back to the original rule, she needs to run matching again. This makes the debugging process very time-consuming.

Therefore, the goal of this operation is to reduce the number of times that the matching operation is run, by allowing the analyst to estimate the effect of her change on the quality before running matching again. With the “find by feature and threshold” operator the new workflow would be as follows: Decide on change, investigate the effect of change on quality using the “find by feature and threshold” operation, if satisfied with the change make the change and run matching again.

With this operation, the analyst selects a feature and threshold for that feature and the candidate record pairs will be filtered by this condition. This is similar to a clause in a rule. By stacking these clauses on top of each other she can effectively make a rule composed of the clauses that she has in mind, and the candidate record pairs will be filtered to show matches by that rule. She can then investigate the pairs matched by this new rule.

Suppose the change was to improve recall using a new rule. She can browse through the record pairs found by this operation and get an idea of how many record pairs that she had not matched before are being matched by this new rule. She may find that the change has no effect on recall and she is not introducing any

new matches by this change. Then she will go back and think of another way to improve recall. But in case she is actually improving recall by this rule, she can estimate the precision of this rule using the “Skyline” operator. She can view the least similar pairs affected by this rule. If those pairs are matches then intuitively this rule will not hurt precision significantly. As you can see with this operation the analyst can estimate the effect of the change she has in mind on the quality even before running matching again.

2.2.6 Feedback from Students and Analysts

We experimented with our end-to-end entity matching system with analysts in an e-commerce setting as well as students in a graduate course at University of Wisconsin-Madison. We demoed our system to analysts at WalmartLabs and got their feedback regarding the usability of the system and features that they think would be helpful for them. Furthermore, we performed matching using our tool for industry data sets. Students worked with the system as part of their course project. They crawled the web and generated data sets for matching from different domains such as cars, movies, and books, among others. They then used our system to upload and browse the data, perform matching, evaluate and debug their rule sets until they reach a quality threshold. They reported on the rule sets that they came up with as well as different iterations that they performed for improving the matching quality.

Table 2.1 shows an overview of 12 student projects with our end-to-end matching system. It is encouraging that students using our tool were able to achieve an average precision of 98 percent and average recall of 91 percent with on average 4 rules. The overall feedback from the analysts and students were very positive as well. Some of their feedback are as follows:

- EMS has the complete pipeline for matching and it is easy for us to find our way around.
- The system is designed well in that projects are easily portable and also uses standard data formats.
- It has good interface for browsing and viewing intermediate results.

Group	Domain	Table sizes (thousands)	# Rules	Precision	Recall
1	Restaurants	25 × 3.2	4	1	0.98
2	Cars	25 × 5	5	0.97	0.88
3	Electronics	21.5 × 3.6	4	0.98	0.94
4	Movies	17 × 9	5	1	0.91
5	Video games	6.7 × 3.7	2	1	0.89
6	Movies	5.5 × 4.3	5	0.97	0.98
7	Books	5 × 6.5	3	0.9	0.96
8	Breakfast	4 × 3.6	4	1	0.79
9	Books	3 × 33.5	2	0.97	0.99
10	Books	3.4 × 3.2	2	1	0.82
11	Books	3.6 × 3.5	5	0.96	0.93
12	Cosmetics	17 × 3.7	NR	NR	NR

Table 2.1: Overview of student projects with our end-to-end entity matching system.

- Easy UI and very well descriptive debugging and refining rules interface, helped us a lot in improving precision and/or recall.
- The tool helped us to identify data cleaning or attribute extraction problems.
- System provides useful statistics.

The feedback was encouraging, and it was clear that students used all the modules in order to get their matching task done. Specifically, they used the debug module iteratively to improve precision and recall. The students also provided suggestions for improving the system. Some of their suggestions are as follows:

- The system may have a bit of learning curve for non-technical users. Specifically, it is not trivial to know which function is suitable for what attribute.
- Add support for suggesting features and functions.
- Add support for machine learning.

- Make the system more robust to user errors and provide useful hints for them to resolve the issue.
- Enable simple data cleaning inside EMS.

From the suggestions it seems that users did not feel confident in selecting features and functions, and would like to try out more automatic approaches such as machine learning. Furthermore, our system did not enable data transformations and cleanings inside the browser module, and they had to do this outside the system. This created a burden for the user.

To mitigate these issues, we can suggest features and functions to users based on attribute types and characteristics such as average length of attribute values. Also, we can enable them to run a variety of machine learning approaches through this system. Since we cannot possibly enumerate all the cleaning operations that a user may want to do on a data set, we can provide support for the major operations, and enable the users to clean the data sets using an interactive scripting environment inside the system. We try to address all these concerns in the consequent versions, with the latest one being Magellan, which we describe in more detail in Section 2.2.8.

2.2.7 State-of-the-art Entity Matching Systems

Konda et al. conduct a comprehensive review of the current entity matching systems as of early 2016 [33], which consists of 18 major non-commercial systems and 15 commercial ones.

Non-commercial systems, such as D-Dupe, DuDe, Febrl, Dedoop, and Nadeef provide support for blocking and matching steps, and limited support for data transformation and cleaning. These systems do not provide guidance to the user on how to perform matching step by step, or guidance on how to select appropriate matchers and blockers. A few of these systems support scaling up the matching and blocking steps. They lack support for other steps of matching such as sampling, labeling, or debugging.

Commercial systems, such as Tamr, Data Ladder, and Informatica Data Quality, mostly offer entity matching as part of a data integration system. They have more sophisticated support and user interface for data cleaning and transformation than

non-commercial systems. Similar to non-commercial systems they provide support mostly for blocking and matching steps, and they do not have support for other steps of the matching pipeline such as sampling, labeling, debugging, and so forth. It is notable that the commercial systems provide fewer variety of matching and blocking mechanisms. Authors suspect that this is due to the fact that they need to support these operations at scale. Indeed, almost all of the commercial systems support scaling up using Hadoop or Spark.

Through review of these commercial and non-commercial systems, it becomes apparent that there is no single entity matching system that supports all entity matching steps. In fact, in Magellan [33], we argue that we cannot possibly build such a system, and that is why it is important that an entity matching system can operate with existing tools as part of an open-source data stack. We will discuss Magellan in more detail below.

2.2.8 Magellan: Toward Building Entity Matching Management Systems

Most entity matching research so far has focused on building algorithms to improve accuracy of matching. Magellan argues that we should build end-to-end systems to make practical impact. They review 18 non-commercial and 15 commercial systems and observe the following key points:

1. While entity matching is an iterative process done in many steps, current entity matching systems only cover part of the steps such as matching and blocking, and miss some equally important steps such as debugging and sampling.
2. Current EM systems are stand-alone and do not co-exist well with other tools that provide other functionalities. They were not inherently designed to be extensible. Therefore, it is labor-intensive and time-consuming for analysts to switch between different tools to complete a matching task.
3. Because a single system cannot support all requirements of every user, it is important that a system is easy to “patch” by users. This means that they should be able to quickly write code to implement a lacking functionality. Most current EM systems do not provide such functionality.

4. Current EM systems do not provide guidance to the user on how to perform matching with high quality. For example, should they use rules or machine learning for achieving a certain quality threshold? or how to debug a selected technique?

Magellan tries to address these limitations. Through providing how-to guides Magellan tells the users what to do step by step and it supports all those steps in a single end-to-end system. Furthermore, all the tools supported by Magellan are built on top of the Python data analysis and big data stack. In particular, Magellan suggests that entity matching is done by analysts in two separate steps. In the development stage, users use data samples to come up with a good quality matching workflow. This is supported through the python data analysis tools such as pandas, scikit-learn, matplotlib, etc. In the production step, this solution is applied to the whole data set which may be much larger and require tools from the big data stack such as Pydoop, mrjob, PySpark, etc. As such, Magellan is well-integrated with the Python data eco-system that allows the users to use a wide range of techniques available to them through this eco-system. Furthermore, this allows Magellan to provide users with an interactive scripting environment such that they can quickly patch the system.

Magellan is developed by the entity matching group at University of Wisconsin-Madison, and we are planning to incorporate some of the tools and techniques proposed by this thesis into Magellan.

2.3 An Abstract Model of an Entity Matching Task for Analysts

Through our experiments with real analysts, students, and working with multiple matching data sets, we noticed that for some matching tasks it is very easy for analysts to come up with a high quality matching solution. However, for some other matching tasks, it deems almost impossible to come up with a set of rules to achieve high quality matching. If we could get insight about the reasons behind this observation we could try to build tools to help analysts with the hardest parts of the matching process. Therefore, to better understand the underlying structural difficulties of matching for analysts, we develop an abstract model of the entity

matching problem for analysts. Our goal from developing this model is not to fully model the matching problem. What we are hoping to achieve is to generate insight on what it is that makes the entity matching problem hard for the analysts.

In this section we explain the components of our matching model. In this model, we work with the rule-based matcher that we described earlier. A rule-based matcher is easier to model and specifically it is easier to model how an analyst would go about generating a set of rules. Normally, an analyst browses through the pairs of records to come up with a set of rules. Therefore, our model consists of 3 main parts. The *analyst* which specifies the rule generation approach and what the analyst does when browsing a particular pair of records. The *pair generator* that specifies the order in which the analyst browses the candidate record pairs, and the *dataset characteristics* which considers the consistency of the dataset. The notion of consistency seems vague at this point and we will define the exact definition of it for this model later. We will explain each of these parts in detail in the following sections.

2.3.1 Analysts

Different analysts generate different rules when they see a new record pair. For example, one analyst may ignore all non-matching pairs and only inspect the matching pairs in detail. Another analyst, may develop a pattern of the non-matching pairs and make sure that she will not match those pairs in the rules that she develops. Therefore, in this model we consider three abstract types of analysts: *Simple*, *ThreshFinder*, and *Consistent* analysts which will be explained in detail.

Note that these analysts are very simple and abstract, and they are not intended to be realistic representations of real analysts. Rather, they are intended to isolate and make precise certain basic aspects of real analysts' behavior. Our hope is that these abstract analysts help us develop insight into how these basic behaviors interact with some key properties of comparison vectors and data sets.

We assume that all the analysts are able to determine the true match status of a record pair when they view it. However, we do not assume that analysts have access to the ground-truth. That means that they cannot calculate precision and recall of their matching output unless they determine the match status of all candidate record pairs. These analysts will assume that all the records pairs in the candidate record pairs are classified as non-match when they start matching. They will then write

positive rules that will classify subsets of the candidate record pairs as matches. The rule set definition is described in Section 2.2.2.

2.3.1.1 The Simple Analyst

Recall that to determine the overall similarity of two records, for each record pair in the candidate record pairs we generated a comparison vector which was composed of similarity scores for selected attribute pairs (See Section 1.2.3 for details).

Given a matching record pair and its comparison vector $F : \langle f_1, f_2, \dots, f_n \rangle$, the simple analyst will make a rule as follows:

$$(s_1 = f_1) \wedge (s_2 = f_2) \wedge \dots \wedge (s_n = f_n) \Rightarrow \text{Match}$$

where $\langle s_1, s_2, \dots, s_n \rangle$ is the comparison vector for every candidate record pair that will be evaluated with this rule. This analyst will not make any rule when seeing a non-matching record pair.

The intuition is that from seeing a matching pair, the analyst concludes that other pairs with the same comparison vector are likely to be matches. Therefore, this rule will match all the record pairs with the exact same comparison vector as this record pair.

Note that the simple analyst is making an implicit assumption about the dataset. It assumes that there are no non-matching pairs with the same comparison vector as the matching pairs. The more this assumption is violated, the more precision errors that the analyst will introduce to the output.

If this assumption is not violated, the analyst will always achieve 100% precision for the matching task. This is because intuitively what this analyst is doing is writing a rule per matching record pair and thus it will never match a non-matching pair. This of course comes at a cost, in order to improve recall this analyst will be making many rules, and it has to potentially view all matching pairs to devise all the rules.

Technically speaking, the simple analyst will achieve 100% recall when it views all the distinct comparison vectors that correspond to at least one matching pair.

This analyst is very interesting because it shows that with certain assumptions about the data, it is actually not a complicated task to achieve 100% precision and recall. You just write one rule per matching pair. It may take a long time and you may write many rules but you eventually will get perfect precision and recall. However,

it is not clear to what extent this rule set will be useful if it is to be used for matching new data.

The simple analyst also highlights the point that matching might be simple if the data set is consistent with respect to the set of features that are used to generate the comparison vectors (We will formally define dataset consistency in Section 2.3.3). That is, in other words, one can view the matching process as at least in part trying to modify the set of features and data set to render them consistent. We will return to this later.

2.3.1.2 The ThreshFinder Analyst

The simple analyst can be very slow in improving recall because the rules that it makes only affects record pairs that have the same comparison vector as the matching pair it just saw. It also tends to make many rules. In order for her to get 100% recall, it will generate a rule for each distinct comparison vector corresponding to at least one matching pair.

To remedy the shortcomings of the simple analyst, we introduce the *ThreshFinder Analyst*. This analyst makes a stronger assumption about the dataset that allows her to improve recall faster. Also, to get to 100% recall it will tend to write fewer rules.

Given a record pair and it's comparison vector $F : \langle f_1, f_2, \dots, f_n \rangle$ the threshfinder analyst will make a rule as follows:

$$(s_1 \geq f_1) \wedge (s_2 \geq f_2) \wedge \dots \wedge (s_n \geq f_n) \Rightarrow \text{Match}$$

where $\langle s_1, s_2, \dots, s_n \rangle$ is the comparison vector for every candidate record pair that will be evaluated with this rule. This analyst will not make any rule when seeing a non-matching record pair.

The intuition is that from seeing a matching pair, the analyst concludes that all other pairs that are more similar than this pair (with respect to the features) are also matching pairs. This helps the analyst to improve recall faster than the simple analyst. However, this analyst is making a stronger assumption about the dataset than the simple analyst. Therefore, if there are non-matching pairs that are more similar than the matching pairs in their comparison vectors, this analyst will make precision errors. If this assumption is not violated, then the analyst can achieve

100% precision. The threshfinder analyst will achieve 100% recall when it views all the least similar matching pairs (with respect to the features).

Suppose the threshfinder analyst's assumption about the dataset is actually true. We formally call this a *consistent* dataset which will be explained in Section 2.3.3. In that case we will see that with many fewer rules and much more quickly an analyst can achieve perfect precision and recall. This suggests that trying to approach a consistent dataset will be helpful in matching.

With the rules that the threshfinder analyst makes, there is a possibility that one rule matches at least all the candidate record pairs that another rule matches. In that case, the second rule is redundant. Therefore, each time the analyst makes a rule, it modifies the rule set with this new devised rule such that there are no redundant rules. We call this a *valid* rule set. Lets formally define a valid rule set:

Suppose rule r_1 uses comparison vector $A = \langle a_1, a_2, \dots, a_n \rangle$ and rule r_2 uses comparison vector $B = \langle b_1, b_2, \dots, b_n \rangle$. Therefore r_1, r_2 are:

$$r_1 : (s_1 \geq a_1) \wedge (s_2 \geq a_2) \wedge \dots \wedge (s_n \geq a_n) \Rightarrow \text{Match}$$

$$r_2 : (s_1 \geq b_1) \wedge (s_2 \geq b_2) \wedge \dots \wedge (s_n \geq b_n) \Rightarrow \text{Match}$$

Definition 2.1. *comparison vector A dominates comparison vector B if and only if:*

$$\forall i \in [1, n], a_i \leq b_i \quad \wedge \quad \exists j \in [1, n], a_j < b_j$$

Definition 2.2. *Rule r_1 with comparison vector A dominates rule r_2 with comparison vector B if and only if A dominates B.*

Intuitively, if r_1 dominates r_2 then the set of pairs that r_1 matches is a superset of pairs that r_2 matches.

Definition 2.3. *In a valid rule set there does not exist 2 rules $\{r_1, r_2\}$ such that r_1 dominates r_2 .*

Intuitively, this means that all rules either should affect the matching output, or should be removed.

The analyst makes sure the rule set is valid in 2 steps:

1. Determine if devised rule is dominated. In that case, there is no reason to add this rule to the rule set. If not, move to step 2.
2. Remove all rules dominated by the devised rule from the rule set. This is because after adding this new rule, the dominated rules will not have any effect on the matching result.

2.3.1.3 The Consistent Analyst

So far the simple and threshfinder analyst made very strong assumptions about the dataset. The simple analyst assumes that if two record pairs have the same comparison vector, one cannot be matching and one non-matching. The threshfinder analyst assumes that a non-matching pair cannot be more similar than a matching pair. If those assumptions hold in the dataset, they will not make a single precision error while devising new rules.

However, this is far from true in the practical datasets and thus the precision drops very quickly as these analysts try to improve the recall. To remedy this situation, we introduce a new analyst: the *Consistent Analyst*, which does not make any assumptions about the dataset and tries to do her best in a dataset that may contain inconsistencies.

The rules that the consistent analyst makes are exactly the same as the threshfinder analyst. The difference is in its reasoning and when it devises a new rule. Rather than making a new rule every time it sees a matching pair, it keeps a memory of the non-matching pairs that it has seen and does not make *inconsistent decisions*.

Definition 2.4. *If an analyst decides that a record pair is a non-match, then it is an inconsistent decision to decide that a less similar pair is a match.*

If the analyst makes inconsistent decisions it means that it is willing to introduce precision errors to the output to improve recall. We configure this willingness with parameter K . By increasing K , we allow the analyst to make decisions that are inconsistent with what it has previously decided.

Now let's define what we exactly mean by *more similar* and *less similar*

Definition 2.5. *Record pair with comparison vector A is more similar than record pair with comparison vector B if and only if A dominates B . If B dominates A then record pair with comparison vector A is less similar than record pair with comparison vector B .*

Note that being *more similar* is always defined with respect to the features used to generate the comparison vectors. A record pair may be more similar than another pair with respect to a set of features, but be more different than the other pair with respect to another set of features. To illustrate this, consider two cases where the comparison vector was generated using only one feature. For the first case, the used feature was title similarity. For the second case, the used feature was price similarity. A record pair may be more similar than another record pair with respect to title. However, these same records may have very different prices, and could be less similar than the other record pair with respect to price.

The consistent analyst makes a rule with a matching pair if it passes the following acceptance criteria:

Definition 2.6. Acceptance criteria: *The analyst has not seen K or more non-matching pairs that are equally or more similar to the current pair.*

This means that if the analyst is sure that adding this rule will introduce at least K precision errors, then it will not make that rule. This way the consistent analyst tries to keep the precision high as it is improving the recall. The higher the value of K is the more the analyst tolerates precision errors.

2.3.2 Pair Generators

Generally an analyst browses record pairs and inspects their comparison vectors to make new rules. Now, the question is: what is the best way to browse the record pairs? Should she randomly look at pairs? Or can we provide her some guidance that can help her quickly come up with a good rule set? The pair generators define the order in which the analyst browses the record pairs. We consider three different pair generators: *Random*, *Sort*, and *Skyline* as described below.

Remember that the analysts in our simple model make a rule based on the match status of a record pair and the comparison vector associated with that pair. Therefore, if they see a pair with the same comparison vector and the same match status as they have seen before this will not add knowledge to them and may significantly increase the number of pairs that they view. For example, all the record pairs that are exact matches will have a comparison vector containing of all 1's. Viewing these pairs over and over again will not add any knowledge to our analysts.

To resolve this issue, the pair generators should not show record pairs that do not add any knowledge to the analysts. However, the pair generators have no way of verifying the match status before showing the pair to the analyst and can only decide based on the comparison vectors. Since we found that record pairs with the exact same comparison vector and with different match status are not common, in our model the pair generators will skip the pairs with repeated comparison vectors; regardless of the match status.

2.3.2.1 The Random Generator

The random generator chooses a record pair randomly and shows it to the analyst. This will work as a baseline for us to compare with the next generators that are designed to have a particular order for showing record pairs.

2.3.2.2 The Sort Generator

For the sort generator, the analyst defines a sort order. For example, <title similarity, year similarity>. The record pairs are sorted based on this hierarchical sort order, resolving ties in the higher level using the values in the lower level. This may be useful to the analyst if she has an idea of the relevancy of the features for matching purpose. The pairs are presented to the analyst in the sort order.

2.3.2.3 The Skyline Generator

For the skyline generator, the items are presented to the analyst from most similar to least similar. If we have only one feature, it is easy to generate this order. You just sort the items on the similarity score for that feature from higher to lower. When we have more than one attribute this is not trivial because an record pair may be more similar than another record pair in one feature but less similar in another feature in the comparison vector.

Definition 2.7. *a record pair is the most similar pair if no pair is more similar than this pair (see definition 2.5).*

Note that there may be more than one record pair that fits this definition; i.e., more than one record pair could be “most similar”. The most similar record pairs are generated iteratively. We form a first-in-first-out queue to hold record pairs to be

shown to the analyst. The record pairs are retrieved one by one from the head of the queue and shown to the analyst. If the queue is empty, the most similar record pairs are extracted, inserted in the queue, and removed from the set of all record pairs.

2.3.3 Dataset Consistency

Dataset consistency is an important factor for the matching problem. As we will see later in the evaluation section, it is actually very easy to do matching on a *consistent* dataset, and it could be very hard to do matching on an *inconsistent* dataset. Unfortunately, many datasets encountered in practice are inconsistent based on the definition below:

Definition 2.8. *A dataset is consistent with respect to a set of features if and only if for any matching pair, there does not exist a non-matching pair that is more similar than the matching pair or has the same comparison vector as that matching pair.*

Definition 2.9. *A dataset is weakly consistent with respect to a set of similarity functions if and only if for any matching pair, there does not exist a non-matching pair that has the exact same comparison vector.*

The intuition is that greater similarity score should reflect greater similarity. However, this is not always the case in reality and in that case we have a weakly consistent or inconsistent dataset. Inconsistency may arise for different reasons such as:

- **Unnormalized data:** The same value may have different formats in the two tables to be matched. For example: hard cover and trade cloth have the same meaning for books, but Walmart may store it as trade cloth and the Bowker data provider may store it as hard cover. This will result unusually low similarity scores for the binding feature for matching pairs.
- **Poor attribute extraction:** If relevant attributes for matching such as color and quantity are not extracted, a non-matching pair may have unusually high similarity measures. For example, if color is not extracted from the title, it can be the case that all words except for a single word “blue” matches with the other title which has a “red” color. This will lead to a high similarity score for the title similarity where the two items actually refer to different products.

- **Using unsuitable similarity functions:** Some functions may be insensitive to slight differences in the values that actually matter for the matching purpose. For example, an single extra character in model number for cameras is very significant for matching purposes but a single character different in the title of the product is not very significant. Therefore, the same similarity function should not be used for comparing model numbers and comparing titles.
- **Incorrect data:** Some data is stored incorrectly into the database. Even important attributes for matching such as “modelno” and “UPC” are sometimes entered incorrectly. This may lead to unusually high or low similarity scores.
- **Missing data:** If at least one of the items in a pair is missing some attribute (for example, price), then typically the similarity score will be 0 for every feature using that attribute. This could lead to unusually low similarity measures for matching pairs that have missing values.

In the next section, we evaluate our abstract model on a real-world dataset. For our evaluation, we define a set of features, convert this inconsistent dataset to consistent and experiment with both of the datasets to understand the implications that dataset consistency has on the matching task.

2.3.4 Empirical Evaluation

We now empirically evaluate our abstract model of the entity matching problem on the Products dataset. This the data set created by [22], and matches electronics products between Amazon and Walmart. We chose this dataset because it is representative of datasets that are used in the industry and is inconsistent with respect to the main features that are used for performing matching on it.

2.3.4.1 The Products Dataset

The product dataset matches 2554 items from Amazon and 22074 items from Walmart with 1154 matching pairs. After blocking there are about 297000 candidate record pairs left.

The features that we consider for matching are as follows:

1. Similarity of Amazon title with Walmart model number

2. Similarity of Amazon title with Walmart title
3. Similarity of Amazon model number with Walmart model number

We chose these features because based on our past experience with this dataset title and modelno similarity are very important for determining the match status for this dataset. The first feature may need further explanation because it searches for the Walmart modelno in the Amazon title. This feature has an important role in recall because for many of the Amazon items, the model number is recorded in the title instead of the model number attribute. All these features are calculated using a customized similarity function created at WalmartLabs for comparing string attributes.

2.3.4.2 Consistent Products Dataset

We created a consistent version of the dataset with the following steps:

1. Remove all the matches such that sum of their similarity scores is less than the threshold $T = 0.7$. This means that to be a match, at least one of the features should be fairly similar.
2. For every match, remove all the non-matches that did not have a similarity score less than this pair for at least one of the features. This means that the non-matching pair either had the same comparison vector as the matching pair, or was more similar than the matching pair.

This is of course not the only way to create a consistent version of the dataset; but we found that it suffices to illustrate the performance of the various analysts with respect to consistent data sets.

Next we present the results with the consistent and inconsistent dataset and discuss the implications of the results on the matching task for analysts.

2.3.4.3 Results with the Consistent Dataset

For the consistent data set, we only experimented with the simple and threshfinder analysts, because the consistent analyst would behave exactly like the threshfinder analyst in the absence of inconsistency. With this dataset, if the analyst sees a

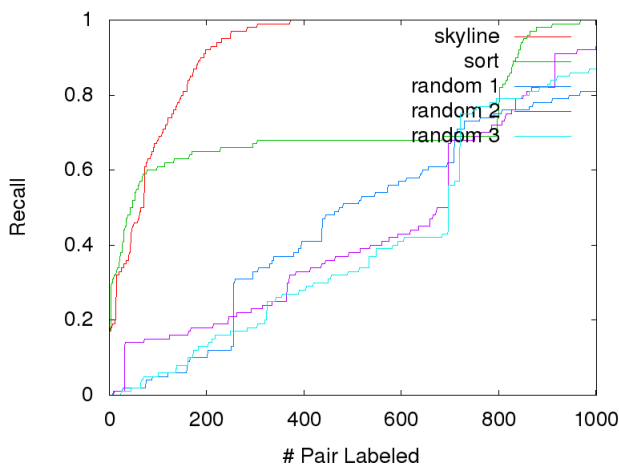


Figure 2.10: Performance of the simple analyst on the consistent dataset.

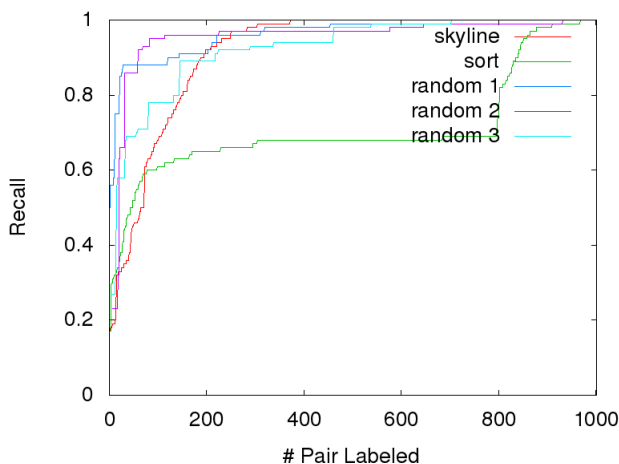


Figure 2.11: Performance of the threshfinder analyst on the consistent dataset.

matching pair and makes a rule with it, there is no chance to hurt precision. The question then becomes how fast the analyst can improve recall.

We measure the performance of the analyst by counting the number of pairs that it views before it gets to a certain precision/recall. Therefore, the fewer pairs viewed to get to a certain precision/recall, the better the performance. We also consider the number of rules the analyst generates. In practice, large rule sets are hard to maintain, and sometimes correspond to over-fitting to the sample data set. Therefore, we present the number of rules generated here to provide insight as to what generates large numbers of rules in our admittedly highly abstract setting.

Figures 2.10 and 2.11 show the performance of simple and threshfinder analysts on this dataset. Recall that when the threshfinder analyst sees a matching pair, it concludes that all pairs that have exactly the same comparison vector or are more similar than this pair match. However, when the simple analyst sees a matching pair, it only concludes that all pairs with exactly the same comparison vector are matches. Thus the simple analyst does not generalize her findings.

Note that with the consistent dataset, both analysts, no matter how they browse the pairs, can get to 100% recall. It is just a matter of how many pairs they need to see before getting to that point.

It may sound counter-intuitive that threshfinder analyst can perform very well with the random generator. In fact, within the first few pairs the analyst can get to 90% recall or above.

This is because there are a total of 1101 unique comparison vectors and 238 (22%) of them belong to matches. This means that in average out of about 5 pairs that the analyst randomly browses, one of them is a match. Since the random pairs shown do not follow any order in terms of similarity scores, the analyst quickly encounters matching pairs with low similarity scores, and the rules that it makes with them automatically cover all the pairs that are more similar than these pairs.

However, analyst's performance with the random pair generator degrades significantly as the number of non-matches increases. To show this, we injected 100000 non-matches to the dataset. All these non-matches have comparison vectors strictly lower than all the matches. In this case there are a total of 101101 unique comparison vectors and 238 (0.002%) of them belong to matches. Thus in average out of every 425 pairs that the analyst randomly browses, 1 of them is a match.

As you can see in Figure 2.12 the analyst performs much worse with the random generator than with the skyline and sort generators when there are many distinct comparison vectors corresponding to non-matches. The random generator just cannot find matches and thus the analyst is not able to improve recall. In fact, the sort generator also stops finding matches at a very low recall. This is because when the threshold on the first attribute in the sort order becomes low, most of the pairs that the analyst views are actually non-matches. The skyline is able to get the analyst to 100% recall without seeing many non-matches.

Also, in the case of the simple analyst, the random generator improves recall very slowly. The analyst sees a lot of non-matches with this generator which makes it slow.

Therefore, we can conclude that randomly browsing the record pairs will result in poor performance unless a high percentage of the distinct comparison vectors belong to matches and we make very strong assumptions about the consistency of the dataset as in the threshfinder analyst.

Another interesting result is that both simple and threshfinder analysts get to 100% recall faster with the skyline generator than sort and random. This is because it does a better job at identifying matching items in the dataset. The random generator simply does not consider any of the similarity scores for generating pairs. The sort generator emphasises the sort order and thus sees many non-matches once the attribute highest in the sort order drops below a certain threshold.

One observation regarding the performance of analysts with skyline and the sort generator is that with each new rule that they devise they improve the recall just a little. This is because these two generators search the space of matches from more similar to less similar one by one. Therefore, when they see a match they already have devised a rule for more similar matches, and thus the current rule that they devise only adds matches with exactly the same comparison vector as this match. This behavior is not necessarily true with real analysts. The real analysts, can quickly skim through the pairs that they think are too similar to the pairs that they have already seen.

As expected, the simple analyst generates many more rules than the threshfinder analyst because it only assumes that pairs with the exact same comparison vector as the matching pair that it just saw are matches. Figures 2.13 and 2.14 show the number of rules devised by the simple versus the threshfinder analyst.

As you can see, the final rule set for the simple analyst has 238 rules, which is the number of unique comparison vectors for matching record pairs. On the other hand, the final rule set for the ThreshFinder has no more than 10 rules for all the generators.

2.3.4.4 Results with the Inconsistent Dataset

In this section, we compare the performance of the analyst with the inconsistent dataset with that of the consistent dataset.

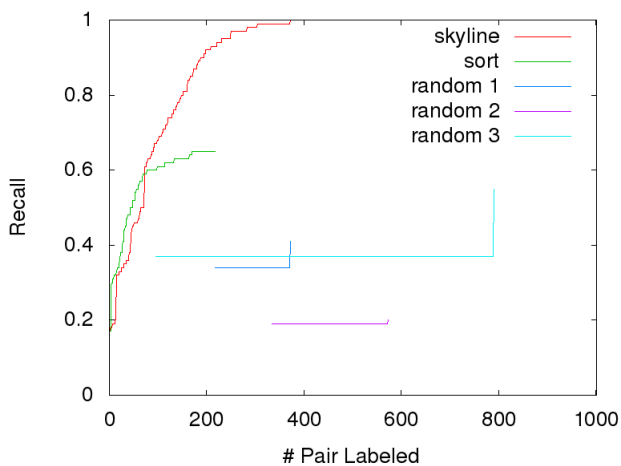


Figure 2.12: Performance of the threshfinder analyst on the consistent dataset after adding 100000 non-matching pairs.

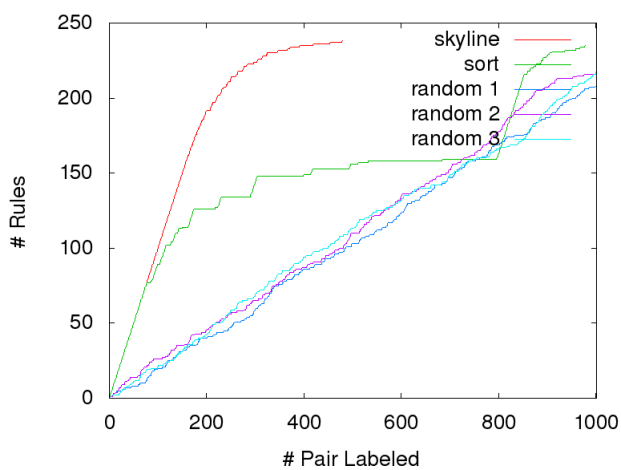


Figure 2.13: Number of rules that the simple analyst generates with the consistent dataset.

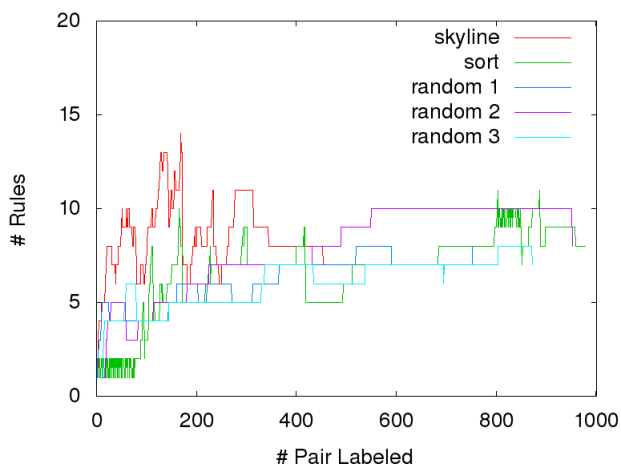


Figure 2.14: Number of rules that the threshfinder analyst generates with the consistent dataset.

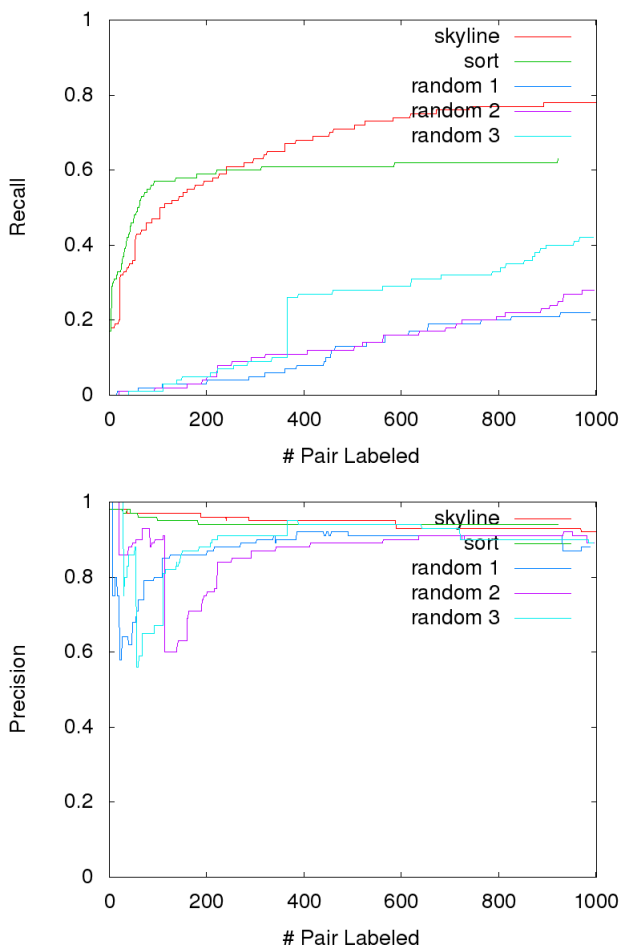


Figure 2.15: Performance of the simple analyst on the inconsistent dataset.

Unlike the consistent data set, with the inconsistent dataset the analyst may introduce precision errors, because there can exist non-matching pairs that are more similar than the matching pairs or have the exact same comparison vector as the matching pairs. This means that none of the assumptions that the simple and threshfinder analyst make hold. Therefore, we look at both precision and recall curves for analyzing performance.

Figure 2.15, 2.16 show the precision and recall for the simple and threshfinder analyst for the inconsistent dataset.

The threshfinder analyst loses precision very quickly with all the generators. Specifically, the analyst loses precision much faster with the random generator than with skyline and sort. Remember that with the consistent dataset, the random

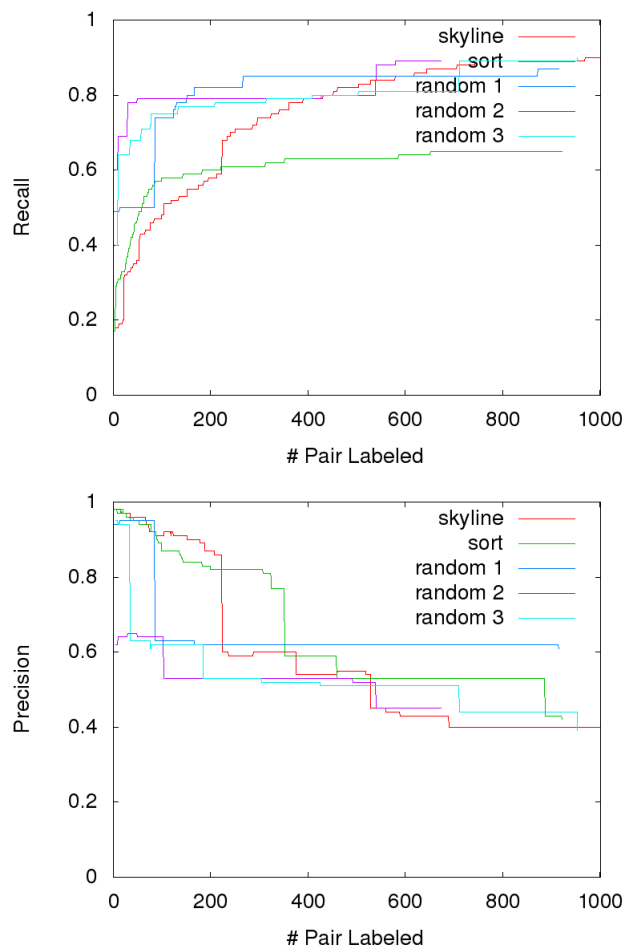


Figure 2.16: Performance of the threshfinder analyst on the inconsistent dataset.

generator in some cases performed very well because there was no chance to hurt precision. But when there are inconsistencies in the data set, it simply cannot keep precision high. This is because it is likely that the analyst views matching pairs that have low similarity scores in the comparison vector and thus are not very similar. In that case there may be many non-matching pairs that have higher similarity scores than this matching pair and will be added to the matching output by the newly devised rule. This problem is avoided with the skyline and sort because they view the matching pairs from more similar to least similar.

Note that the simple analyst is able to keep precision higher for longer. This is because the rules that it makes are very specific to the matching pairs that it sees. However, since the dataset is not even weakly consistent, there may be non-matching

pairs with exactly the same comparison vector as the matching pairs has, and thus the simple analyst can still have precision errors.

In fact, if the dataset is at least weakly consistent, the simple analyst can eventually achieve 100% recall and 100% precision. However, the simple analyst makes many rules to achieve this, basically one rule per matching pair. This rule set will be tailored to the matching pairs in this data set and it is very possible that it will not work well on new data.

With the inconsistent dataset, the notion of the consistent analyst makes sense. This analyst tries to avoid precision errors using the knowledge about the non-matching pairs that it has already seen. It will not devise a rule that it is sure will add at least K precision errors (false positives) to the matching output. In Figure 2.17 you can see the performance of the consistent analyst with inconsistency threshold $K = 10$.

With the consistent analyst, with all the generators, the analyst can improve recall and keep the precision higher than the simple and threshfinder analyst.

Also, with the random generator, even the consistent analyst has a hard time keeping precision high because it does not view the matches from most similar to least similar and thus cannot keep track of inconsistencies.

2.3.4.5 Conclusions from the experiments with abstract analysts

In this section, we would like to revisit some of the interesting and/or important findings throughout the experiments:

- With the consistent dataset, all the analysts no matter how they browse the pairs, can get to 100% recall. However, using the skyline generator they will get to 100% recall faster than with random and sort pair generators. The skyline generator does a better job at finding matching pairs.
- Randomly browsing the record pairs will result in poor performance with both consistent and inconsistent data sets for common matching tasks where a very high percentage of the candidate record pairs are non-matches. For the consistent data set, with the random generator the analyst will not encounter matching pairs and keeps looking through non-matching record pairs. For the inconsistent data set, the same problem holds plus using the random generator leads to a very quick drop in precision. The random generator is not able to

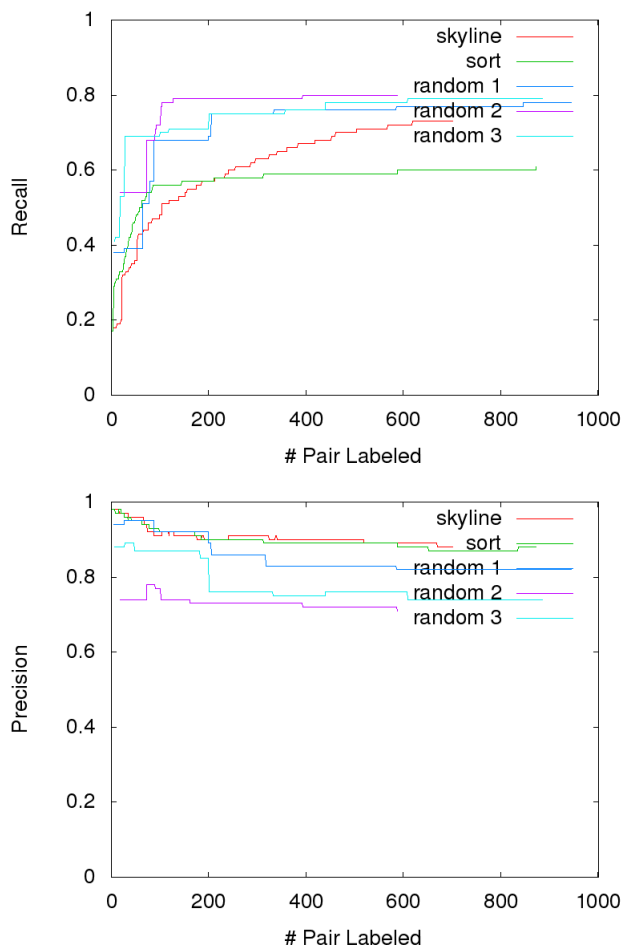


Figure 2.17: Performance of the consistent analyst on the inconsistent dataset.

keep precision high even with the consistent analyst because the analyst will not be able to detect inconsistencies through randomly browsing the pairs.

- The simple analyst, who basically writes a rule per matching pair, improves recall very slowly in all cases and generates many rules. However, with the inconsistent dataset it is able to keep precision higher for longer than the threshfinder analyst because it only assumes that the dataset is weakly consistent. However, it is not clear if the rules that it makes would result in high quality with a new set of data.
- The main challenge with the inconsistent dataset is to keep precision high while improving recall. With this dataset, the consistent analyst does a better

job than simple and threshold analysts since it is reluctant to write rules that it knows will cause precision errors.

2.4 Insights from Experience with Real and Abstract Analysts

Insights that we gained from working with real and abstract analysts helped us shape the next chapters of this thesis. In particular, we noticed that as the analyst explores the set of features and rules for matching the data set, she goes through many iterations before achieving high quality. The matching step is performed in each iteration and if we could speed it up, it means that the analyst can go through the iterations and complete the matching task faster. We will focus on this problem in Chapter 3¹.

Through working with real analysts and real-world data sets we noticed that some data sets are harder to match than others. With our abstract analysts we found out that the notion of dataset consistency has a significant impact on analyst performance such that consistent data sets could be easily matched with high precision and recall with a few rules. In Chapter 4, we investigate how we can help the analyst quickly find and eliminate inconsistencies in a data set.

¹We did not do formal user studies, so the insights from real analysts are only presented to provide intuition and insight rather than to draw concrete conclusions.

3 TOWARDS INTERACTIVE DEBUGGING OF RULE-BASED EM

3.1 Introduction

Rule-based entity matching is widely used in practice [3, 32]. This involves analysts designing and maintaining sets of rules. These analysts typically follow an iterative and time-consuming debugging process, as depicted in Figure 3.1. For example, imagine an e-commerce marketplace that sells products from different vendors. When a vendor submits products from a new category, the analyst writes a set of rules designed to match these products with existing products. He or she then applies these rules to a labeled sample of the data and waits for the results. If the analyst finds errors in the matching output, he or she will refine the rules and re-run them, repeating the above process until the result is of sufficiently high quality.

Our goal is to make this process interactive by reducing the time that an analyst is idle in the *Run EM* step. Research shows that when interacting with software, if the response time is greater than one second, the analyst’s flow of thought will be interrupted, and if it is greater than 10 seconds, the user will turn their attention to other tasks [31]. Therefore, it is imperative to reduce the idle “waiting” time as much as possible. Moreover, the faster an iteration can be finished, the more iterations that can be accomplished in a given time, which typically leads to better matching quality.

Rule-based entity matching is typically accomplished by evaluating a boolean matching function for each candidate record pair (for example, B1 in Figure 3.2). In this thesis, we follow the approach we have encountered in practice in which this matching function is in Disjunctive Normal Form (DNF). Each disjunction is a rule, and each rule is a conjunction of a set of predicates that evaluate the similarity of two records on one or more attributes, using a similarity function (such as Jaccard or TF-IDF). For example, $\text{Jaccard}(a.\text{name}, b.\text{name}) > 0.7$ is a predicate, where $\text{Jaccard}(a.\text{name}, b.\text{name})$ is a feature. A record pair is a match if it matches at least one rule.

As was pointed out by Benjelloun et al. [3], and confirmed in our experiments, computing similarity function values dominates the matching time. In view of this, our basic idea is to eliminate unnecessary similarity function computations as the analyst defines new rules and/or refines existing rules.

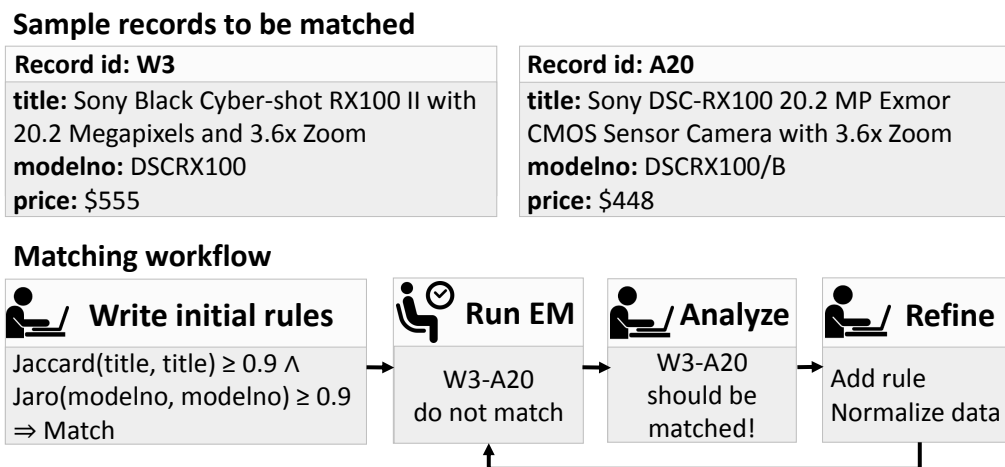


Figure 3.1: A typical matching workflow for analysts.

Specifically, we exploit properties of DNF/CNF rule sets that enable “early exit” evaluation, which can eliminate the need for evaluating many rules and/or predicates for a given candidate pair of records. Moreover, we use “dynamic memoing”: we only compute (and materialize the result of) a feature (i.e., a similarity score) if that predicate result is required by the matching function (because of early exit, not all features need to be computed.) This “lazy feature computation” strategy can thus save significant computation cost when there are many possible features but only a few of them are really required by the rule set/data set under consideration. Techniques such as “early exit” and “dynamic memoing” are of course ubiquitous in computer science, but to the best of our knowledge, ours is the first study of their efficacy in the domain of entity matching.

Meanwhile, the application of “early exit” and “dynamic memoing” implies that different evaluation orders of the predicates/rules may lead to significant differences in computational cost. This then raises the problem of optimal predicate/rule ordering. We study this problem in detail. We show that the optimization problem under our setting is NP-hard, and we propose two greedy solutions based on heuristic optimization criteria. In our experiments with six real-world datasets, we show that the greedy solutions can indeed produce orderings that significantly reduce runtime compared to random ones.

Since the elements (e.g., features, predicates, or rules) involved in matching change frequently as the analyst iteratively refines the rule set, we further develop an incremental matching solution to avoid rerunning matching from scratch after

each change. We show that our incremental solutions can reduce matching time by orders of magnitude.

3.2 Related Work

Our work differs from previous work in several ways. Previous work on efficiently running rule-based entity resolution [3] assumes that each predicate is a black box, and thus memoing of similarity function results is not possible. In our experience in an industrial setting, these predicates are often not black boxes — rather, they are explicitly presented in terms of similarity functions, attributes, and thresholds.

On the other hand, the traditional definition of the EM workflow, as described in [9, 14], assumes that *all* similarity values for *all* pairs are *precomputed* before the matching step begins. This makes sense in a batch setting in which a static matching function has been adopted, and the task is to apply this function to a set of candidate record pairs. However, in this chapter we are concerned with the exploratory stage of rule generation, where at the outset the matching function is substantially unknown. In such settings the combinatorial explosion of potential attributes pairs, potential similarity functions, and candidate pairs can render such full precomputation infeasible.

Even in small problem instances in which full precomputation may be feasible, it can impose a substantial lag time between the presentation of a new matching task and the time when the analyst can begin working. This lag time may not be acceptable in practical settings where tens of matching tasks may be created every day [22] and the analyst wants to start working on high priority tasks immediately. Finally, during the matching process, an analyst may perform cleaning operations, normalization, and attribute extractions on the two input tables. The analyst might also introduce new similarity functions. In any of these situations, it is not possible to precompute all features a priori.

Previous work on incrementally evaluating the matching function when the logic evolves assumes that we evaluate all predicates for all pairs and materialize the matching result for each predicate [43]. Because we use early exit, our information about the matching results for each predicate is not complete. As a result, this solution is not directly applicable in our setting.

In other related work, Dedoop (abbreviation for “Deduplication with Hadoop”) [27] seeks to improve performance for general, large, batch entity matching tasks through the exploitation of parallelism available in Hadoop. By contrast, our work focuses on interactive response for rule-based entity matching where the matching function is composed of many rules that evolve over time. Exploring the application of parallelism as explored in Dedoop to our context is an interesting area for future work.

Our work is also related to [39]. In that work, the user provides a set of rule templates and a set of labeled matches and non-matches, the system then efficiently searches a large space of rules (that instantiate the rule templates) to find rules that perform best on the labeled set (according to an objective function). That work also exploits the similarities among the rules in the space. But it does so to search for the best set of rules efficiently. In contrast, we exploit rule similarities to support interactive debugging.

Finally, our work is related to the Magellan project, also at UW-Madison [28]. That project proposes to perform entity matching in two stages. In the development stage, the user iteratively experiments with data samples to find an accurate EM workflow. Then in the production stage the user executes that workflow on the entirety of data. If the user has decided to use a rule-based approach to EM, then in the development stage he or she will often have to debug the rules, which is the focus of this paper. This work thus fits squarely into the development stage of the Magellan approach.

In the following we present our approach to try to achieve interactive response times, and present experimental results of our techniques on six real world data sets.

3.3 Motivating Example

To motivate and give an overview of our approach, consider the following example. Our task is to match Table A and Table B shown in Figure 3.2 to find records that refer to the same person. We have four candidate pairs of records: $\{a1b1, a1b2, a2b1, a2b2\}$. Assume our matching function is B1. Intuitively, B1 says that if the name and zipcode of two records are similar, or if the phone number and name of two records are similar, then they match. Here $p1_{name}$ and $p2_{name}$ for

Table A					Table B				
Id	Name	Street	Zip	Phone	Id	Name	Street	Zip	Phone
a1	John	Dayton	54321	123-4567	b1	John	Dayton	54321	987-6543
a2	Bob	Regent	53706	121-1212	b2	John	Bascom	11111	258-3524

Matching function evolution

B1: $(p_{1_name} \wedge p_{zip}) \vee (p_{phone} \wedge p_{2_name}) \rightarrow$
B2: $(p_{1_name} \wedge p_{zip} \wedge p_{street}) \vee (p_{phone} \wedge p_{2_name})$

Figure 3.2: Tables A, B to be matched and example matching functions. Function B1 evolves to B2.

example compute $\text{Jaccard}(a.name, b.name)^1$, then compare this value to different thresholds, respectively, as we will see below. For this example, B1 will return true for a1b1 and false for the rest of the candidate pairs.

A simple way to accomplish matching is to evaluate every predicate for every candidate pair. To evaluate a predicate, we compute the value of the similarity function associated with that predicate and compare it to a threshold. For the candidate pair a2b1, we would compute 4 similarity values.

This is unnecessary because once a predicate in a rule evaluates to false, we can skip the remaining predicates. Similarly, once a rule evaluates to true, we can skip the rest of the rules and therefore finish matching for that pair. We call this strategy “early exit,” which saves unnecessary predicate evaluations. For instance, consider the candidate pair a2b1 again. Suppose that the predicate p_{1_name} is

$$\text{Jaccard}(a.name, b.name) \geq 0.9.$$

Since the Jaccard similarity of the two names is 0, p_{1_name} will return false for this candidate pair. Further assume that p_{phone} performs an equality check and thus returns 0 as well. We then do not need to evaluate p_{zip} and p_{2_name} to make a decision for this pair. Therefore, for this candidate pair, “early exit” reduces the number of similarity computations from 4 to 2.

Since the same similarity function may be applied to a candidate pair in multiple rules and predicates, we “memo” each similarity value once it has been computed.

¹In practice we often compute Jaccard over the sets of q-grams of the two names, e.g., where $q = 3$; here for ease of exposition we will assume that Jaccard scores are computed over the set of words of the two names.

If a similarity function appears in multiple predicates, only the first evaluation of the predicate incurs a computation cost, while subsequent evaluations only incur (much cheaper) lookup costs. We call this strategy “dynamic memoing.” Continuing with our example, suppose p_{2_name} is

$$\text{Jaccard}(a.name, b.name) \geq 0.7.$$

Then for a_2b_1 this predicate only involves a lookup cost.

When using *early exit* and *dynamic memoing*, different orders of the predicates/rules will make a difference in the overall matching cost. Once again consider the candidate pair a_2b_1 . If we change the order of predicates in B_1 to

$$(p_{1_name} \wedge p_{zip}) \vee (p_{2_name} \wedge p_{phone}),$$

the output of the matching function will not change. However, it reduces the matching cost to one computation for p_{1_name} plus one lookup for p_{2_name} . This raises a novel optimization problem that we study in Section 3.6.

Finally, we take into account the fact that, as the matching function’s logic evolves, the changes to the function are often *incremental*. We can then store results of a previous EM run, and as the EM logic evolves, use those to save redundant work for the next EM iterations. As an example, imagine the case where the matching function B_1 evolves to B_2 . Since B_2 is *stricter* than B_1 , we only need to evaluate p_{street} for the pairs that were matched by B_1 to verify if they still match. For our example, this means that we only need to evaluate B_1 for a_1b_1 among the four pairs.

3.4 Preliminaries

Recall from Chapter 1 that the input to the entity matching (EM) workflow is two tables A, B with a set of records $\{a_1 \dots a_n\}, \{b_1 \dots b_m\}$ respectively. The goal of EM is to find all record pairs $a_i b_j$ that refer to the same entity. Given table A with m records and table B with n records, there are $m \times n$ potential matches. Even with moderate-size tables, the total number of potential matches could be very large. Many of these potential matches obviously do not match and can be eliminated

from consideration easily. That is the purpose of a *blocking* step, which typically precedes a more detailed matching phase.

For example, suppose that each product has a category attribute (e.g., clothing or electronics). We can assume that products from different categories are clear non-matches. This will reduce the task to finding matching products within the same category. We refer to the set of potential matches left after the blocking step as the *candidate record pairs* or *candidate pairs* throughout the rest of this chapter.

Each candidate record pair is evaluated by a Boolean *matching function* B , which takes in two records and returns true or false. We assume that B is commutative, i.e.,

$$\forall a_i b_j, B(a_i, b_j) = B(b_j, a_i).$$

We assume that each matching function is in disjunctive normal form (DNF). We refer to each disjunct as a *rule*. For example, our matching function $B1$, depicted in Figure 3.2 is composed of two rules.

Such a matching function is composed of only “positive” rules, as they say what matches, not what does not match. In our experience, this is a common form of matching function used in the industry. Reasons for using only positive rules include ease of rule generation, comprehensibility, ease of debugging, and commutativity of rule application.

Each rule is a conjunction of a set of *predicates*. Each predicate compares the value of a *feature* for a candidate pair with a threshold. A feature in our context is a similarity function computed over attributes from the two tables. Similarity functions can be as simple as exact equality, or as complex as arbitrary user-defined functions requiring complex pre-processing and logic.

Recall from Chapter 1 that the matching result is composed of the return value of the matching function for each of the candidate pairs. In order to evaluate the quality of matching, typically a sample of the candidate pairs is randomly chosen and manually labeled as match or non-match based on domain knowledge. The matching results for the sample is then compared with the correct labels to get an estimate of the quality of matching. The standard measures for quality are *precision* and *recall*.

Algorithm 1: The rudimentary baseline.

Input: B , the matching function; \mathcal{C} , candidate pairs

Output: $\{(c, x)\}$, where $c \in \mathcal{C}$ and $x \in \{M, U\}$ (M means a match and U means an unmatched)

```

1 Let  $\mathcal{R}$  be the CNF rules in  $B$ ;
2 Mark all  $c \in \mathcal{C}$  with  $U$ ;
3 foreach  $c \in \mathcal{C}$  do
4   | foreach  $r \in \mathcal{R}$  do
5   |   | foreach  $p \in \text{predicate}(r)$  do
6   |   |   | Evaluate  $p$ ;
7   |   |   end
8   |   | Evaluate  $r = \bigwedge_{p \in \text{predicate}(r)} p$ ;
9   |   end
10  | Mark  $c$  with  $M$  if  $B = \bigvee_{r \in \mathcal{R}} r$  is true;
11 end

```

3.5 Early Exit + Dynamic Memoing

In this section, we present the details of *early exit* and *dynamic memoing*.

3.5.1 Baselines

We study two baseline approaches in this section. In the following algorithms, for a given rule r , we will use $\text{predicate}(r)$ and $\text{feature}(r)$ to denote the set of predicates and features r includes, respectively.

3.5.1.1 The Rudimentary Baseline

The first baseline algorithm simply evaluates every predicate in the matching function for every candidate pair. Each predicate is considered as a black box and any similarity value used in the predicate is computed from scratch. The results of the predicates (true or false) are then passed on to the rules, and the outputs of the rules passed on to the matching function to determine the matching status. Algorithm 1 presents the details of this baseline.

3.5.1.2 The Precomputation Baseline

This algorithm precomputes all feature values involved in the predicates before performing matching. Algorithm 2 presents the details of this baseline. As noted in the introduction of this chapter, full precomputation may not be feasible or desirable in practice, but we present it here as a point of comparison. We store precomputed values as a hash table mapping pairs of attribute values to similarity function outputs.

Algorithm 2: The precomputation baseline.

Input: B , the matching function; \mathcal{C} , candidate pairs

Output: $\{(c, \chi)\}$, where $c \in \mathcal{C}$ and $\chi \in \{M, U\}$ (M means a match and U means an unmatched)

- 1 Let \mathcal{R} be the CNF rules in B ;
 - 2 Let $\mathcal{F} = \bigcup_{r \in \mathcal{R}} \text{feature}(r)$;
 - 3 Let $\Gamma = \{(c, f, v)\}$ be a $|\mathcal{C}| \times |\mathcal{F}|$ array that stores the value v of each $f \in \mathcal{F}$ for each $c \in \mathcal{C}$;
 - 4 **foreach** $c \in \mathcal{C}$ **do**
 - 5 **foreach** $f \in \mathcal{F}$ **do**
 - 6 Compute v and store (c, f, v) in Γ ;
 - 7 **end**
 - 8 **end**
 - 9 Run Algorithm 1 by looking up feature values from Γ when evaluating predicates;
-

3.5.2 Early Exit

Both baselines discussed above ignore the properties of the matching function B . Given that B is in DNF, if one of the rules returns true, B will return true. Similarly, because each rule in B is in CNF, a rule will return false if one of its predicate returns false. Therefore, we do not need to evaluate all the predicates and rules. Algorithm 3 uses this idea. While this is straightforward, we choose to retain the details here to ease our discussion on cost analysis in Section 3.5.4. The “breaks” in lines 8 and 12 are the “early exits” in this algorithm.

Algorithm 3: Early exit.

Input: B , the matching function; \mathcal{C} , candidate pairs**Output:** $\{(c, x)\}$, where $c \in \mathcal{C}$ and $x \in \{M, U\}$ (M means a match and U means an unmatched)

```

1 Let  $\mathcal{R}$  be the CNF rules in  $B$ ;
2 Mark all  $c \in \mathcal{C}$  with  $U$ ;
3 foreach  $c \in \mathcal{C}$  do
4   | foreach  $r \in \mathcal{R}$  do
5   |   |  $r$  is true;
6   |   | foreach  $p \in \text{predicate}(r)$  do
7   |   |   | if  $p$  is false then
8   |   |   |   |  $r$  is false; break;
9   |   |   | end
10  |   | end
11  |   | if  $r$  is true then
12  |   |   | Mark  $c$  with  $M$ ; break;
13  |   | end
14  | end
15 end

```

3.5.3 Dynamic Memoing

We can combine the precomputation of the second baseline with early exit. That is, instead of precomputing everything up front, we postpone the computation of a feature until it is encountered during matching. Once we have computed the value of a feature, we store it so following references of this feature only incur lookup costs. We call this strategy “dynamic memoing,” or “lazy feature computation.” Algorithm 4 presents the details.

3.5.4 Cost Modeling and Analysis

In section, we develop simple cost models to use in rule and predicate ordering decisions studied in Section 3.6. In the following discussion, we use $\text{cost}(p)$ to denote the cost of evaluating a predicate p . Let \mathcal{C} be the set of all candidate pairs. Moreover, let F be the set of all features involved in the matching function, and we use $\text{cost}(f)$ to denote the computation cost of a feature f . Furthermore, we use δ to represent the lookup cost.

Algorithm 4: Early exit with dynamic memoing.

Input: B , the matching function; \mathcal{C} , candidate pairs

Output: $\{(c, x)\}$, where $c \in \mathcal{C}$ and $x \in \{M, U\}$ (M means a match and U means an unmatched)

```

1 Let  $\mathcal{R}$  be the CNF rules in  $B$ ;
2 Let  $\Gamma$  be the feature values computed;  $\Gamma \leftarrow \emptyset$ ;
3 Mark all  $c \in \mathcal{C}$  with  $U$ ;
4 foreach  $c \in \mathcal{C}$  do
5   foreach  $r \in \mathcal{R}$  do
6      $r$  is true;
7     foreach  $p \in \text{predicate}(r)$  do
8       Let  $f$  be the feature in  $p$ ;
9       if  $f \notin \Gamma$  then
10        | Compute  $f$ ;  $\Gamma \leftarrow \Gamma \cup \{f\}$ ;
11        | else
12        |   Read the value of  $f$  from  $\Gamma$ ;
13        | end
14        | if  $p$  is false then
15        |    $r$  is false; break;
16        | end
17        | end
18        | if  $r$  is true then
19        |   Mark  $c$  with  $M$ ; break;
20        | end
21    end
22 end

```

3.5.4.1 The Rudimentary Baseline

The cost of the rudimentary baseline (Algorithm 1) can be represented as:

$$C_1 = \sum_{c \in \mathcal{C}} \sum_{r \in \mathcal{R}} \sum_{p \in \text{predicate}(r)} \text{cost}(p).$$

In our running example in the introduction, the cost of making a decision for the pair $a1b2$ is then

$$\text{cost}(p1_{\text{name}}) + \text{cost}(p_{\text{zip}}) + \text{cost}(p_{\text{phone}}) + \text{cost}(p2_{\text{name}}).$$

3.5.4.2 The Precomputation Baseline

Suppose that each feature f appears $\text{freq}(f)$ times in the matching function. Then the cost of the precomputation baseline (Algorithm 2) is

$$C_2 = \sum_{c \in \mathcal{C}} \sum_{f \in \mathcal{F}} (\text{cost}(f) + \text{freq}(f)\delta).$$

In our running example this means that, for pair $a1b2$ and matching function $B1$, we would need to precompute three similarity values and look up four. Note that this requires knowing $\text{cost}(f)$ — in our implementation, as discussed in our experimental results, we use an estimate of $\text{cost}(f)$ obtained by evaluating f over a sample of the candidate pairs.

3.5.4.3 Early Exit

To compute the cost of early exit (Algorithm 3), we further introduce the probability $\text{sel}(p)$ that the predicate p will return true for a given candidate pair (i.e., the *selectivity* of p). In our implementation, we use an estimate of $\text{sel}(p)$ obtained by evaluating p over a sample of the candidate pairs.

Given this estimate for $\text{sel}(p)$, suppose that we have a rule r with m predicates p_1, \dots, p_m . The expected cost of evaluating r for a (randomly picked) candidate pair is then

$$\begin{aligned} \text{cost}(r) = & \text{cost}(p_1) + \text{sel}(p_1) \text{cost}(p_2) + \dots \\ & + \text{sel}\left(\bigwedge_{j=1}^{m-1} p_j\right) \text{cost}(p_m), \end{aligned} \quad (3.1)$$

because we only need to evaluate p_j if p_1, \dots, p_{j-1} are all evaluated to be true. Similarly, we can define the selectivity of the rule r as

$$\text{sel}(r) = \text{sel}\left(\bigwedge_{j=1}^m p_j\right).$$

Suppose that we have n rules r_1, \dots, r_n . The expected cost of the early exit strategy (Algorithm 3) is then

$$\begin{aligned} C_3 = & \text{cost}(r_1) + (1 - \text{sel}(r_1)) \text{cost}(r_2) + \dots \\ & + (1 - \text{sel}\left(\bigvee_{i=1}^{n-1} r_i\right)) \text{cost}(r_n). \end{aligned}$$

3.5.4.4 Early Exit with Dynamic Memoing

The expected cost of early exit with dynamic memoing (Algorithm 4) can be estimated in a similar way. The only difference is that we need to further know the probability that a feature is present in the memo. Specifically, suppose that a feature can appear at most once in a rule. Let $\alpha(f, r_i)$ be the probability that a feature f is present in the memo after evaluating r_i . The expected cost of computing f when evaluating r_i is then

$$E[\text{cost}(f)] = (1 - \alpha(f, r_{i-1})) \text{cost}(f) + \alpha(f, r_{i-1})\delta. \quad (3.2)$$

The expected cost C_4 of Algorithm 4 can be obtained by replacing all $\text{cost}(p)$'s in Equation 3.1 with their expected cost expressed in Equation 3.2.

Let $\text{prev}(f, r_i)$ be the predicates in the rule r_i that appear before f . We then have

$$\alpha(f, r_i) = (1 - \alpha(f, r_{i-1})) \text{sel}\left(\bigwedge_{p \in \text{prev}(f, r_i)} p\right) + \alpha(f, r_{i-1}).$$

Based on our assumption, different predicates in the same rule contain different features. If we further assume that predicates with different features are independent, it then follows that

$$\alpha(f, r_i) = (1 - \alpha(f, r_{i-1})) \prod_{p \in \text{prev}(f, r_i)} \text{sel}(p) + \alpha(f, r_{i-1}).$$

Note that the initial condition satisfies

$$\alpha(f, r_1) = \prod_{p \in \text{prev}(f, r_1)} \text{sel}(p).$$

We therefore have obtained an inductive procedure for estimating $\alpha(f, r_i)$ ($1 \leq i \leq n$). Clearly, $\alpha(f, r_i) = \alpha(f, r_{i-1})$ if $f \notin \text{feature}(r_{i-1})$. So we only need to focus on those rules that contain f .

3.6 Optimal Ordering

Our goal in this section is to develop techniques to order rule and predicate evaluation to minimize the total cost of matching function evaluation. This may sound familiar, and indeed it is — closely related problems have been studied previously

Notation	Description
$\text{cost}(X)$	cost of X (X is a feature/predicate/rule)
δ	the lookup cost
$\text{freq}(f)$	frequency of feature f
$\text{predicate}(r)$	predicates of rule r
$\text{feature}(X)$	features of X (X can be a predicate/rule)
$\text{sel}(X)$	selectivity of X (X can be a predicate/rule)
$\text{prev}(f, r)$	features/predicates in rule r before feature f
$\text{predicate}(f, r)$	predicates in rule r that have feature f
$\text{reduction}(r)$	overall cost reduction by execution of rule r
$\text{cache}(f, r)$	chance that f is in the memo after running r

Table 3.1: Notation used in cost modeling and optimal rule ordering study.

in related settings (see, for example, [2, 25]). However, our problem is different and unfortunately more challenging due to the interaction of early exit evaluation with dynamic memoing.

3.6.1 Notation

Table 3.1 summarizes notation used in this section. Some of the notation has been used in the previous section when discussing the cost models.

3.6.2 Problem Formulation

We briefly recap an abstract version of the problem. We have a set of rules $\mathcal{R} = \{r_1, \dots, r_n\}$. Each rule is in CNF, with each clause containing exactly one predicate. A pair of records is a match if any rule in \mathcal{R} evaluates to true. Therefore, \mathcal{R} is a disjunction of rules:

$$\mathcal{R} = r_1 \vee r_2 \vee \dots \vee r_n.$$

Consider a single rule

$$r = p_1 \wedge p_2 \wedge \dots \wedge p_m.$$

We are interested in the minimum expected cost of evaluating r with respect to different orders (i.e., permutations) of the predicates p_1, \dots, p_m .

Given a specific order of the predicates, the expected cost of r can be expressed as

$$\begin{aligned} \text{cost}(r) = & \text{cost}(p_1) + \text{sel}(p_1) \text{cost}(p_2) + \cdots \\ & + \text{sel}\left(\bigwedge_{j=1}^{m-1} p_j\right) \text{cost}(p_m). \end{aligned} \quad (3.3)$$

Similarly, given a specific order of the rules, the expected cost of evaluating R , as was in Section 3.5.4.3, is

$$\begin{aligned} \text{cost}(R) = & \text{cost}(r_1) + (1 - \text{sel}(r_1)) \text{cost}(r_2) + \cdots \\ & + (1 - \text{sel}\left(\bigvee_{i=1}^{n-1} r_i\right)) \text{cost}(r_n). \end{aligned} \quad (3.4)$$

We want to minimize $\text{cost}(R)$.

3.6.3 Independent Predicates and Rules

The optimal ordering problem is not difficult when independence of predicates/rules holds. We start by considering the optimal ordering of the predicates in a single rule r . If the predicates are independent, Equation 3.3 reduces to

$$\begin{aligned} \text{cost}(r) = & \text{cost}(p_1) + \text{sel}(p_1) \text{cost}(p_2) + \cdots \\ & + \text{sel}(p_1) \cdots \text{sel}(p_{m-1}) \text{cost}(p_m). \end{aligned}$$

The following lemma is well known for this case (e.g., see Lemma 1 of [25]):

Lemma 3.1. *Assume that the predicates in a rule r are independent. $\text{cost}(r)$ is minimized by evaluating the predicates in ascending order of the metric:*

$$\text{rank}(p_i) = (\text{sel}(p_i) - 1) / \text{cost}(p_i) \quad (\text{for } 1 \leq i \leq m).$$

We next consider the optimal ordering of the rules by assuming that the rules are independent. We have the following similar result.

Theorem 3.2. *Assume that the predicates in all the rules are independent. $\text{cost}(R)$ is minimized by evaluating the rules in ascending order of the metric:*

$$\text{rank}(r_j) = -\frac{\text{sel}(r_j)}{\text{cost}(r_j)} = -\frac{\prod_{p \in \text{predicate}(r_j)} \text{sel}(p)}{\text{cost}(r_j)}.$$

Here $\text{cost}(r_j)$ is computed by using Equation 3.3 with respect to the order of predicates specified in Lemma 3.1.

Proof. By De Morgan's laws, we have

$$R = r_1 \vee \dots \vee r_n = \neg(\bar{r}_1 \wedge \dots \wedge \bar{r}_n).$$

Define $r'_j = \bar{r}_j$ for $1 \leq j \leq n$ and $R' = \neg R$. It follows that

$$R' = r'_1 \wedge \dots \wedge r'_n.$$

This means, to evaluate R , we only need to evaluate R' , and then take the negation. Since R' is in CNF, based on Lemma 3.1, the optimal order is based on

$$\text{rank}(r'_j) = (\text{sel}(r'_j) - 1) / \text{cost}(r'_j) \quad (\text{for } 1 \leq j \leq n).$$

We next compute $\text{sel}(r'_j)$ and $\text{cost}(r'_j)$. First, we have

$$\text{sel}(r'_j) = 1 - \text{sel}(r_j) = 1 - \prod_{p \in \text{predicate}(r_j)} \text{sel}(p),$$

by the independence of the predicates. Moreover, we simply have $\text{cost}(r'_j) = \text{cost}(r_j)$, because we can evaluate r'_j by first evaluating r_j and then taking the negation. Therefore, it follows that

$$\text{rank}(r_j) = \text{rank}(r'_j) = -\frac{\text{sel}(r_j)}{\text{cost}(r_j)} = -\frac{\prod_{p \in \text{predicate}(r_j)} \text{sel}(p)}{\text{cost}(r_j)}.$$

This completes the proof of the theorem. \square

Recall that in our implementation we compute feature costs and selectivity by sampling a set of record pairs and compute the costs and selectivities on the sample. So far, we have implicitly assumed that memoing is not used.

3.6.4 Correlated Predicates and Rules

We now consider the question when memoing is used. This introduces dependencies so Lemma 3.1 and Theorem 3.2 no longer hold.

Let us start with one single rule r . We introduce a *canonical form* of r by “grouping” together predicates that share common features. Formally, for a predicate p , let $\text{feature}(p)$ be the feature it refers to. Furthermore, define

$$\text{feature}(r) = \cup_{p \in \text{predicate}(r)} \{\text{feature}(p)\}.$$

Given a rule r and a feature $f \in \text{feature}(r)$, let

$$\text{predicate}(f, r) = \bigwedge_{p \in \text{predicate}(r) \wedge \text{feature}(p)=f} p.$$

We can then write the rule r as

$$r = \bigwedge_{f \in \text{feature}(r)} \text{predicate}(f, r). \quad (3.5)$$

Since we only consider predicates of the form $A \geq a$ or $A \leq a$ where A is a feature and a is a constant threshold, it is reasonable to assume that each rule does not contain redundant predicates/features. As a result, each group $\text{predicate}(f, r)$ can contain at most one predicate of the form $A \geq a$ and/or $A \leq a$. Based on this observation, we have the following simple result.

Lemma 3.3. *$\text{cost}(\text{predicate}(f, r))$ is minimized by evaluating the predicates in ascending order of their selectivities.*

Proof. Remember that $\text{predicate}(f, r)$ contains at most two predicates p_1 and p_2 . Note that, the costs of the predicates follow the pattern c, c' if memoing is used, regardless of the order of the predicates in $\text{predicate}(f, r)$. Here c and c' are the costs of directly computing the feature or looking it up from the memo ($c > c'$). As a result, we need to decide which predicate to evaluate first. This should be the predicate with the lower selectivity. To see this, without loss of generality let us assume $\text{sel}(p_1) < \text{sel}(p_2)$. The overall cost of evaluating p_1 before p_2 is then

$$C_1 = c + \text{sel}(p_1)c',$$

whereas the cost of evaluating p_2 before p_1 is

$$C_2 = c + \text{sel}(p_2)c'.$$

Clearly, $C_1 < C_2$. This completes the proof of the lemma. \square

Since the predicates in different groups are independent, by applying Lemma 3.1 we get the following result.

Lemma 3.4. *cost(r) is minimized by evaluating the predicate groups in ascending order of the following metric:*

$$\text{rank}(\text{predicate}(f, r)) = \frac{\text{sel}(\text{predicate}(f, r)) - 1}{\text{cost}(\text{predicate}(f, r))}.$$

Here $\text{cost}(\text{predicate}(f, r))$ is computed by using Equation 3.3 with respect to the order of predicates specified in Lemma 3.3.

Now let us move on to the case in which there are multiple rules whose predicates are not independent. Unfortunately, this optimization problem is in general NP-hard. We can prove this by reduction from the classic traveling salesman problem (TSP) as follows. Let the rules be vertices of a complete graph G . For each pair of rules r_i and r_j , define the cost $c(i, j)$ of the edge (r_i, r_j) to be the execution cost of r_j if it immediately follows r_i . Note that here we have simplified our problem by assuming that the cost of r_j only depends on its predecessor r_i . Under this specific setting, our problem of finding the optimal rule order is equivalent to seeking a Hamiltonian cycle with minimum total cost in G , which is NP-hard. Moreover, it is known that a constant-factor approximation algorithm for TSP is unlikely to exist unless P equals NP (e.g., see Theorem 35.3 of [11]). Therefore, in the following we seek heuristic approaches.

3.6.4.1 Greedy Algorithms

We now need to further order the rules by considering the overhead that can be saved by memoing. By Lemma 3.4, the predicates in each rule can be locally optimally ordered. Note that each order of the rules induces a global order over the (bag of) predicates. However, the selectivities of the predicates are no longer independent, because predicates in different rules may share the same feature. Furthermore,

Algorithm 5: A greedy algorithm based on expected costs of rules.

Input: $\mathcal{R} = \{r_1, \dots, r_n\}$, a set of CNF rules
Output: \mathcal{R}^π , execution order of the rules

- 1 Let Q be a priority queue $\langle (\text{cost}(r), r) \rangle$ of the rules;
- 2 $\mathcal{R}^\pi \leftarrow \emptyset$;
- 3 **foreach** $r \in \mathcal{R}$ **do**
- 4 Order predicate(r) according to Lemma 3.4;
- 5 Compute $\text{cost}(r)$ based on this order;
- 6 Insert $(\text{cost}(r), r)$ into Q ;
- 7 **end**
- 8 **while** Q is not empty **do**
- 9 $r_{\min} \leftarrow \text{ExtractMin}(Q)$;
- 10 Add r_{\min} into \mathcal{R}^π ;
- 11 **foreach** $(\text{cost}(r), r) \in Q$ **do**
- 12 Update $\text{cost}(r)$ by assuming that r immediately follows r_{\min} ;
- 13 **end**
- 14 **end**
- 15 **return** \mathcal{R}^π ;

the costs of predicates are no longer constants due to memoing. In fact, they even depend on the positions of the predicates in their global order. In other words, the costs of predicates depend on the order of the rules (recall the cost model in Section 3.5.4.4). Hence we are not able to apply Lemma 3.1 or Theorem 3.2 in this context.

Nonetheless, intuitively, a predicate should tend to have priority if it is very selective (returns true for very few pairs) and small cost, since it will eliminate many pairs cheaply. On the other hand, a rule should tend to have priority if it is not very selective (returns true for many candidate pairs) and small cost, since it contributes many matches cheaply.

Our first algorithm then uses this intuition in a greedy strategy by picking the rule with the minimum expected cost. The details of this algorithm are presented in Algorithm 5. Note that when we update $\text{cost}(r)$ at line 12, we use the cost model developed in Section 3.5.4.4, which considers the effect of memoing, by assuming that r will be the immediate successor of r_{\min} .

Algorithm 5 only considers the expected costs of the rules if they are the first to be run among the remaining rules. Some rules may have slightly high expected costs but significant long-term impact on overall cost reduction. Algorithm 5 does

not consider this and thus may overlook these rules. We therefore further consider a different metric that is based on the rules that can be affected if a rule is executed. This gives our second greedy algorithm.

In the following, we use $\text{reduction}(r)$ to represent the overall cost that can be saved by the execution of the rule r , and use $\text{cache}(f, r)$ to represent the probability that a feature f is in the cache (i.e., memo) after the execution of r . For two features f_1 and f_2 in r , we write $f_1 < f_2$ if f_1 appears before f_2 in the order of predicate groups specified by Lemma 3.4. Following Section 3.5.4.4, we redefine $\text{prev}(f, r)$ to be the *features* that appear before f in r , namely,

$$\text{prev}(f, r) = \{f' \in \text{feature}(r) \wedge f' < f\}.$$

If we write r as it is in Equation 3.5, then

$$\text{sel}(\text{prev}(f, r)) = \prod_{f' \in \text{prev}(f, r)} \text{sel}(\text{predicate}(f', r)) \quad (3.6)$$

is the selectivity of (conjunction of) the predicates appearing before f in r . Here we have abused notation because $\text{prev}(f, r)$ is a set of features rather than a predicate. Basically, $\text{sel}(\text{prev}(f, r))$ is the chance that the feature f needs to be computed (by either direct computation or cache lookup) when executing r . We further define $\text{prev}(r)$ to be the rule executed right before r . It then follows that

$$\begin{aligned} \text{cache}(f, r) &= (1 - \text{cache}(f, \text{prev}(r))) \text{sel}(\text{prev}(f, r)) \\ &\quad + \text{cache}(f, \text{prev}(r)). \end{aligned}$$

Next, define $\text{contribution}(r', r)$ to be the reduced cost of r' by executing the rule r before the rule r' . Further define $\text{contribution}(r', r, f)$ to be the reduced cost due to the feature f . Let $\text{feature}(r', r) = \text{feature}(r') \cap \text{feature}(r)$. Clearly,

$$\text{contribution}(r', r) = \sum_{f \in \text{feature}(r', r)} \text{contribution}(r', r, f).$$

We now consider how to compute $\text{contribution}(r', r, f)$. If we do not run r before r' , the expected cost of evaluating f in r' is then

$$\begin{aligned} \text{cost}_1(f, r') &= \text{sel}(\text{prev}(f, r')) [\text{cache}(f, \text{prev}(r))\delta \\ &\quad + (1 - \text{cache}(f, \text{prev}(r))) \text{cost}(f)], \end{aligned}$$

whereas if we run r before r' the cost becomes

$$\begin{aligned} \text{cost}_2(f, r') &= \text{sel}(\text{prev}(f, r')) [\text{cache}(f, r)\delta \\ &\quad + (1 - \text{cache}(f, r)) \text{cost}(f)]. \end{aligned}$$

It then follows that

$$\begin{aligned} \text{contribution}(r', r, f) &= \text{cost}_1(f, r') - \text{cost}_2(f, r') \\ &= \text{sel}(\text{prev}(f, r')) \Delta(\text{cost}(f) - \delta), \end{aligned}$$

where $\Delta = \text{cache}(f, r) - \text{cache}(f, \text{prev}(r))$.

Based on the above formulation, we have

$$\text{reduction}(r) = \sum_{r' \neq r} \text{contribution}(r', r).$$

Our second greedy strategy simply picks the rule r that maximizes $\text{reduction}(r)$ as the next rule to be executed. Algorithm 6 presents the details of the idea. It is more costly than Algorithm 5 because update of $\text{reduction}(r)$ at line 21 requires $O(n)$ rather than $O(1)$ time, where n is the number of rules.

Note that the computations of $\text{cost}(r)$ and $\text{reduction}(r)$ are still based on local decisions, namely, the *immediate* effect if a rule is executed. The actual effect, however, depends on the actual ordering of all rules and cannot be estimated accurately without finishing execution of all rules (or alternatively, enumerating all possible rule orders).

3.6.4.2 Discussion

If we only employ early exit without dynamic memoing, the optimal ordering problem remains NP-hard when the predicates/rules are correlated. However, we can have a greedy 4-approximation algorithm [2, 30]. The difference in this

Algorithm 6: A greedy algorithm based on expected overall cost reduction.

Input: $\mathcal{R} = \{r_1, \dots, r_n\}$, a set of CNF rules
Output: \mathcal{R}^π , execution order of the rules

- 1 Let Q be a priority queue $\langle (\text{reduction}(r), r) \rangle$ of the rules;
- 2 $\mathcal{R}^\pi \leftarrow \emptyset$;
- 3 **foreach** $r \in \mathcal{R}$ **do**
- 4 | Order predicate(r) according to Lemma 3.4;
- 5 **end**
- 6 **foreach** $r \in \mathcal{R}$ **do**
- 7 | $\text{reduction}(r) \leftarrow 0$;
- 8 | **foreach** $r' \in \mathcal{R}$ *such that* $r' \neq r$ **do**
- 9 | | **foreach** $f \in \text{feature}(r')$ **do**
- 10 | | | **if** $f \in \text{feature}(r)$ **then**
- 11 | | | | $\text{reduction}(r) \leftarrow \text{reduction}(r) + \text{contribution}(r', r, f)$;
- 12 | | | **end**
- 13 | | **end**
- 14 | **end**
- 15 | Insert $(\text{reduction}(r), r)$ into Q ;
- 16 **end**
- 17 **while** Q *is not empty* **do**
- 18 | $r_{\max} \leftarrow \text{ExtractMax}(Q)$;
- 19 | Add r_{\max} into \mathcal{R}^π ;
- 20 | **foreach** $(\text{reduction}(r), r) \in Q$ **do**
- 21 | | Update $\text{reduction}(r)$ by assuming that r immediately follows r_{\max} ;
- 22 | **end**
- 23 **end**
- 24 **return** \mathcal{R}^π ;

context is that the costs of the predicates no longer depend on the order of the rules. Rather, they are constants so approximation is easier. One might then wonder if combining early exit with precomputation (but not dynamic memoing) would make the problem even tractable, for now the costs of the predicates become the same (i.e., the lookup cost). Unfortunately, the problem remains NP-hard even for uniform costs when correlation is present [20].

3.6.4.3 Optimization: Check Cache First

We have proposed two greedy algorithms for ordering rules and predicates in each rule. The order is computed before running any rule and remains the same during

matching. However, the greedy strategies we proposed are based on the “expected” rather than actual costs of the predicates. In practice, once we start evaluating the rules, it becomes clear that a feature is in the memo or not. One could then further consider dynamically adjusting the order of the remaining rules based on the current content of the memo. This incurs nontrivial overhead, though: we basically have to rerun the greedy algorithms each time we finish evaluating a rule. So in our current implementation we do not use this optimization. Nonetheless, we are able to reorder the predicates inside each rule at runtime based on the content of the memo, if lookups are much cheaper than direct computations (which is the common case in practice). Specifically, we first evaluate predicates for which we have their features in the memo, and we still rely on Lemma 3.4 to order the remaining predicates.

3.6.5 Putting It All Together

The basic idea in this section is to order the rules such that we can decide on the output of the matching function with lowest computation cost for each pair. To order the rules we use a small random sample of the candidate pairs and estimate feature costs and selectivities for each predicate and rule. We then use Algorithm 5 or Algorithm 6 to order the rules. These two algorithms consider two different factors that affect the overall cost: 1) the expected cost of each rule, and 2) the expected overall cost reduction that executing this rule will have if the features computed for this rule are repeated in the following rules. In our experiments we show results for both algorithms.

3.7 Incremental Matching

So far we have discussed how to perform matching for a fixed set of fixed rules. We now turn to consider incremental matching in the context of an evolving set of rules.

3.7.1 Materialization Cost

To perform incremental matching, we materialize the following information during each iteration:

- For each pair: For each feature that was computed for this pair, we store the calculated score. Note that because we use lazy feature computation, we may not need to compute all feature values.
- For each rule: Store all pairs for which this rule evaluated to true.
- For each predicate: Store all pairs for which this predicate evaluated to false.

We show in our experiments that if we use straightforward techniques such as storing bitmaps of pairs that pass rules or predicates, the total memory needed to store this information for our data sets is less than 1GB.

3.7.2 Types of Matching Function Changes

An analyst often applies a single change to the matching function, re-runs EM, examines the change in the output, then applies another change. We identify different types of changes to the matching function and present our incremental matching algorithm for each type.

3.7.2.1 Add a Predicate / Make a Predicate More Strict

If a matching result contains pairs that should not actually match, the analyst can make the rules that matched such a pair more “strict” by either adding predicates, or making existing predicates more strict. (We consider the alternative of deleting a rule separately.) For example, consider the following predicate

$$\text{Jaccard}(a.\text{name}, b.\text{name}) \geq 0.7.$$

We can make this more strict by changing it to

$$\text{Jaccard}(a.\text{name}, b.\text{name}) \geq 0.8.$$

In this case, we can obtain the new matching results incrementally by evaluating this modified predicate only for the pairs that were evaluated and matched by the rule we made stricter. Consider such a previously matched pair:

- If the modified predicate returns true, the pair is still matched.

- If the modified predicate returns false, the current rule no longer matches this pair. However, other rules in the matching function may match this pair, so we must evaluate the pair with the other rules until either a rule returns true or all rules return false.

We can use the same approach for adding a new predicate to a rule, because that can be viewed as making an empty predicate that always evaluates to true more strict. Algorithm 7 illustrates the procedure for adding a predicate.

Algorithm 7: Add a predicate.

Input: \mathcal{R} , the set of CNF rules; r , the rule that was changed; p , the predicate added to r

- 1 Let $M(r)$ be the previously matched pairs by r ;
- 2 Let X be the unmatched pairs by p ; $X \leftarrow \emptyset$;
- 3 **foreach** $c \in M(r)$ **do**
- 4 **if** p returns false for c **then**
- 5 $X \leftarrow X \cup \{c\}$;
- 6 **end**
- 7 **end**
- 8 Let \mathcal{R}' be the rules in \mathcal{R} after r ;
- 9 **foreach** $c \in X$ **do**
- 10 Mark c as an unmatched;
- 11 **foreach** $r' \in \mathcal{R}'$ **do**
- 12 **if** r' returns true for c **then**
- 13 Mark c as a match; **break**;
- 14 **end**
- 15 **end**
- 16 **end**

3.7.2.2 Remove a Predicate / Make a Predicate Less Strict

In the case where pairs that should match are missing from the result, we might be able to fix the problem by either removing a predicate or making an existing predicate less strict. (We consider the alternative of adding a new rule separately.) Consider again the predicate

$$\text{Jaccard}(a.\text{name}, b.\text{name}) \geq 0.7.$$

We can make it less strict by changing it to

$$\text{Jaccard}(a.\text{name}, b.\text{name}) \geq 0.6.$$

In both cases, all pairs for which this predicate returned false need to be re-evaluated. Consider such a previously unmatched pair:

- If the new predicate returns false, the pair then remains unmatched.
- If the new predicate returns true, we will evaluate the other predicates in the rule.² If any of these predicates returns false, then the pair will remain a non-match. Otherwise, this rule will return true for this pair, and it will be declared a match.

Algorithm 8 illustrates the details of the procedure for updating the matching result after making a predicate less strict. Removing a predicate follows similar logic and is omitted for brevity.

3.7.2.3 Remove a Rule

We may decide to remove a rule if it returns true for pairs that should not match. In such a case, we can re-evaluate the matching function for all pairs that were matched by this rule. Either another rule will declare this pair a match or the matching function will return false. Algorithm 9 illustrates this procedure.

3.7.2.4 Add a Rule

One way to match pairs that are missed by a current matching function is to add a rule that returns true for them. In this case, inevitably, all non-matched pairs need to be evaluated by this rule. However, note that only the newly added rule will be evaluated for the non-matched pairs, which can be substantial savings over re-evaluating all rules. Algorithm 10 demonstrates this procedure.

²Note that, because we use the “check-cache-first” optimization, the order of the predicates within the rule is no longer fixed. In other words, different pairs may observe different orders. So we cannot just evaluate predicates that “follow” the changed one.

Algorithm 8: Make a predicate less strict.

Input: r , the rule that was changed; p , the predicate of r that was made less strict

- 1 Let $U(p)$ be the pairs for which p returned false;
- 2 Let Y be the pairs p now returns true; $Y \leftarrow \emptyset$;
- 3 **foreach** $c \in U(p)$ **and** c was an unmatch **do**
- 4 | **if** p returns true for c **then**
- 5 | | $Y \leftarrow Y \cup \{c\}$;
- 6 | **end**
- 7 **end**
- 8 **foreach** $c \in Y$ **do**
- 9 | Mark c as a match;
- 10 | **foreach** $p' \in \text{predicate}(r)$ **and** $p' \neq p$ **do**
- 11 | | **if** p' returns false for c **then**
- 12 | | | Mark c as an unmatch; **break**;
- 13 | | **end**
- 14 | **end**
- 15 **end**

Algorithm 9: Remove a rule.

Input: \mathcal{R} , the set of CNF rules; r , the rule removed

- 1 Let $M(r)$ be the previously matched pairs by r ;
- 2 Let \mathcal{R}' be the rules in \mathcal{R} after r ;
- 3 **foreach** $c \in M(r)$ **do**
- 4 | Mark c as an unmatch;
- 5 | **foreach** $r' \in \mathcal{R}'$ **do**
- 6 | | **if** r' returns true for c **then**
- 7 | | | Mark c as a match; **break**;
- 8 | | **end**
- 9 | **end**
- 10 **end**

3.8 Experimental Evaluation

In this section we explore the impact of our techniques on the performance of various basic and incremental matching tasks. We ran experiments on a Linux machine with eight 2.80 GHz processors (each with 8 MB of cache) and 8 GB of main memory. We implemented our algorithms in Java. We used six real-world data sets as described below.

Algorithm 10: Add a rule.

Input: \mathcal{R} , the set of CNF rules; r , the rule added

- 1 Let $U(r)$ be the previously unmatched pairs by \mathcal{R} ;
- 2 **foreach** $c \in U(r)$ **do**
- 3 Mark c as a match;
- 4 **foreach** $p \in \text{predicate}(r)$ **do**
- 5 **if** p returns false for c **then**
- 6 Mark c as an unmatched; **break**;
- 7 **end**
- 8 **end**
- 9 **end**

Data set	Source 1	Source 2	Table1 size	Table2 size	Candidate pairs	Rules	Used features	Total features
Products	Walmart	Amazon	2554	22074	291649	255	32	33
Restaurants	Yelp	Foursquare	3279	25376	24965	32	21	34
Books	Amazon	Barnes&Noble	3099	3560	28540	10	8	32
Breakfast	Walmart	Amazon	3669	4165	73297	59	14	18
Movies	Amazon	Bestbuy	5526	4373	17725	55	33	39
Video games	TheGamesDB	MobyGames	3742	6739	22697	34	23	32

Table 3.2: Real-world data sets used in the experiments.

3.8.1 Datasets and Matching Functions

We evaluated our solutions on six real-world data sets. One data set was obtained from an industry EM team. The remaining five data sets were created by students in a graduate-level class as part of their class project, where they had to crawl the Web to obtain, clean, and match data from two web sites. Table 3.2 describes these six data sets.

We extensively experiment with the first and largest data set and present the results for the other data sets in Section 3.8.7. We obtained the Walmart/Amazon products data set used in [22] from the authors of that paper. The dataset domain is electronics items from Walmart.com and Amazon.com. After the blocking step, we have 291,649 candidate pairs. Gokhale et al. [22] have generated the labels for these pairs.

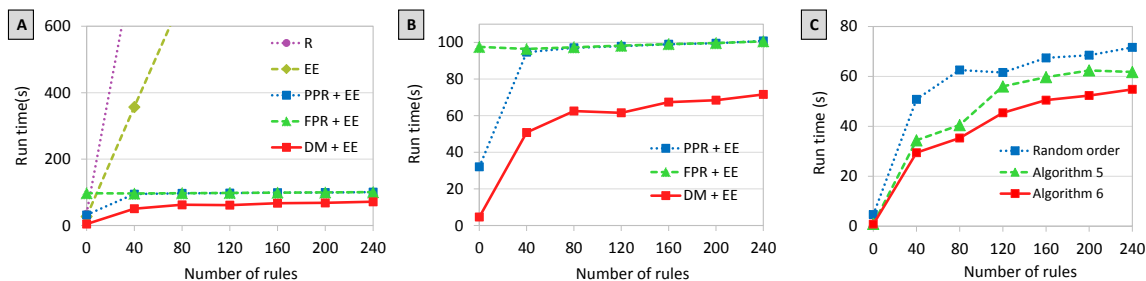


Figure 3.3: **(A)** Run time for different sizes of matching function for rudimentary baseline (R), early exit (EE), production precomputation baseline + early exit (PPR + EE), full precomputation baseline + early exit (FPR + EE), and dynamic memoing + early exit (DM + EE). **(B)** Zoom-in of A to compare methods that use precomputation/dynamic memoing. **(C)** Run time for different orderings of the set of rules/predicates: Random ordering, order by Algorithm 5, and order by Algorithm 6.

$$\begin{aligned}
 R1 \quad & \text{Jaro Winkler}(m, m) \geq 0.97 \wedge \\
 & \text{Jaro}(m, m) \geq 0.95 \\
 & \wedge \text{Soft TF-IDF}(m, t) < 0.28 \\
 & \wedge \text{TF-IDF}(m, t) < 0.25 \wedge \text{Cosine}(t, \\
 & t) \geq 0.69
 \end{aligned}$$

$$\begin{aligned}
 R2 \quad & \text{Jaccard}(t, t) < 0.4 \wedge \text{TF-IDF}(t, t) < \\
 & 0.55 \\
 & \wedge \text{Soft TF-IDF}(t, t) \geq 0.63 \wedge \text{Jaccard} \\
 & \geq 0.34 \\
 & \wedge \text{Levenshtein}(m, m) < 0.72 \\
 & \wedge \text{Jaro Winkler}(m, m) < 0.05
 \end{aligned}$$

Figure 3.4: Sample rules extracted from the random forest. m, t stand for modelno and title respectively.

We generated 33 features using a variety of similarity functions based on heuristics that take into account the length and type of the attributes. Table 3.3 shows a subset of these features and their associated average computation times. The computation times of features vary widely.

Using a combination of manual and semi-automatic approaches, analysts from the EM team that originally created the data set have created a total of 255 matching rules. We will use this rule set as a basis from which to create and evaluate a variety of matching functions. Figure 3.4 shows two sample rules for this data set.

Function	Walmart	Amazon	μ s
Exact Match	modelno	modelno	0.2
Jaro	modelno	modelno	0.5
Jaro Winkler	modelno	modelno	0.77
Levenshtein	modelno	modelno	1.22
Cosine	modelno	title	3.37
Trigram	modelno	modelno	4.79
Jaccard	modelno	title	6.75
Soundex	modelno	modelno	8.77
Jaccard	title	title	10.54
TF-IDF	modelno	title	12.18
TF-IDF	title	title	18.92
Soft TF-IDF	modelno	title	21.89
Soft TF-IDF	title	title	66.06

Table 3.3: Computation costs for a subset of features in the products data set

3.8.2 Early Exit + Dynamic Memoing

Figure 3.3A shows the effect of early exit and precomputing/memoing feature values on matching time as we use an increasingly larger rule set. For example, to generate the data point corresponding to 20 rules, we randomly selected 20 rules and measured the time to apply them to the data set. For each data point we report the average running time over three such random sets of rules.

We compare the run time for baseline, early exit, production precomputation + early exit, full precomputation + early exit, and dynamic memoing + early exit. For production precomputation, which we described as one of our baselines in Section 3.5.1.2, we assume that we know all the features that are used in the rules. We call this “production precomputation” because it would be feasible only if the set of rules for matching is already finalized. In full precomputation, we know a superset of features that the analyst will choose from to make the rule set. In such a case, we may precompute values for features that will never be used. We compare these approaches with dynamic memoing + early exit proposed in this chapter.

We can see that the rudimentary baseline has a very steep slope, and around 20 rules, it takes more than 10 minutes to complete. The early exit curve shows significant improvement over baseline, however, it is still slow compared to either the precomputation baseline or early exit with dynamic memoing. Figure 3.3B zooms in and shows the curves for the full and production precomputation baselines and dynamic memoing. We can see that using dynamic memoing + early exit can significantly reduce matching time.

In this section, we have not considered the optimal ordering problem, and we ran dynamic memoing with a random ordering of the rules and predicates in each rule. In the next section, we further study the effectiveness of our greedy algorithms on optimizing orderings of predicates/rules.

3.8.3 Optimal Ordering

Figure 3.3C shows runtime as we increase the number of rules for “dynamic memoing + early exit” with random ordering of predicates/rules, as well as that with orderings produced by the two greedy strategies presented in Algorithm 5 and Algorithm 6. Each data point was generated using the same approach described in the previous section. We used a random sample consisting of 1% of the candidate pairs for estimating feature costs and predicate selectivities. We can see that the orderings produced by both of these algorithms are superior to the random ordering.

We further observe that Algorithm 6 is faster than Algorithm 5, perhaps due to the fact that its decision is based on a global optimization metric that considers the overall cost reduction by placing a rule before other rules. As the number of rules increases, the impact is less significant, because most of the features have to be computed. Nonetheless, matching using Algorithm 6 is still faster even when we use 240 rules in the matching function.

3.8.4 Memory Consumption

We store the similarity values in a two dimensional array. We assign each pair an index based on their order in the input table. Similarly, we assign each feature a random order and an index based on the order. In the case of the precomputation baseline, this memo is completely filled with feature values before we start matching. In the case of dynamic memoing, we fill in the memo as we run matching and the

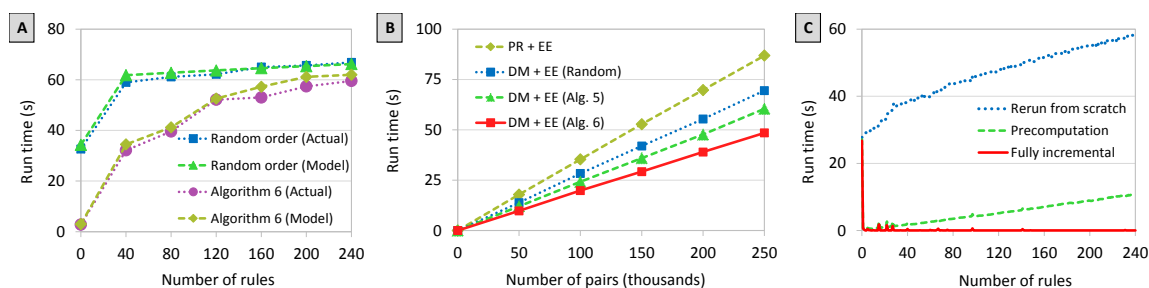


Figure 3.5: **(A)** Actual run time versus run time estimated by the cost model for random ordering of rules and rules ordered by Algorithm 6. **(B)** Run time as we increase number of pairs for production precomputation + early exit (PPR + EE), and dynamic memoing + early exit (DM + EE) for random ordering of rules, rules ordered by Algorithm 5, and ordered by Algorithm 6. **(C)** Run time with dynamic memoing + early exit as we incrementally add rules one by one to the matching function in three cases: 1) Rerun matching from scratch, 2) Precomputation: lookup memoed feature values but evaluate all rules 3) Fully incremental: lookup memoed feature values but only evaluate the newly added rule.

analyst makes changes to the rule set. Therefore, the memory consumption of both approaches is the same. For this dataset, if we use all rules, the two-dimensional array takes 22 MB of space, which includes the space for storing the actual floats as well as the bookkeeping overhead for the array in Java. For incremental matching, we store a bitmap for each rule as well as for each predicate. In our implementation, we use a boolean array for each bitmap. For this dataset, we have 255 rules and a total of 1,688 predicates. These bitmaps occupy 542 MB.

For our dataset, the two-dimensional array and bitmaps fit comfortably in memory. For a data set where this is not true, one could consider avoiding an array and using a hash-map for storing similarity values. Since we do not compute all the feature values, this would lead to less memory consumption, although the lookup cost for hash-maps would be more expensive.

3.8.5 Cost Modeling and Analysis

To illustrate accuracy of our cost models, in Figure 3.5A we compare the actual run time of “dynamic memoing + early exit” versus run time estimated by the cost model for random ordering of rules as well as rules ordered by Algorithm 6. The two curves follow each other closely.

To compute the selectivity of each predicate, we select a sample of the candidate pairs, evaluate each predicate for the pairs in the sample and compute the percentage of pairs that pass each predicate. In our experiments, we observed that using a 1% sample can give relatively accurate estimates of the selectivity, and increasing the sample size did not change the rule ordering in a major way. We used the same small sample approach to estimate feature costs.

Figure 3.5B shows the actual matching time when we use all the rules for the data set as we increase number of pairs. As we assumed in our cost modeling, the matching cost increases linearly as we increase number of pairs. Given this increase proportional to the number of pairs (which is itself quadratic in the number of input records), the importance of performance enhancing techniques to achieve interactive response times will increase with larger data sets.

3.8.6 Incremental Entity Matching

Our first experiment in this section examines the “add rule” change to a rule set. Adding a rule can be expensive for incremental entity matching because we need to evaluate the newly added rule for all the unmatched pairs.

To test how incremental matching performs for adding a new rule, we conducted an experiment in which we start from an empty matching function without any rules. We then add the first rule to the matching function, run matching with this single-rule matching function, and materialize results. Next, we add the second rule and measure the time required for incremental matching. In general, we run matching based on k rules, and then run incremental matching for the $(k + 1)$ -th rule when it is added. We repeat this for $1 \leq k \leq 240$.

We consider two variations of incremental algorithm. In the *precomputation* variation, all the rules in the matching function are evaluated. Note that we use early exit and the optimization discussed in Section 3.6.4.3 with this variation to reduce unnecessary lookups. The second variation is *fully incremental*. In this case we not only lookup the stored feature values, but also only evaluate part of the matching function for the subset of candidate pairs that will be affected by this operation. In particular, for the “add rule” operation, all the non-matched pairs need to be evaluated by just the new rule that is added, and all the rules in the matching function do not need to be evaluated.

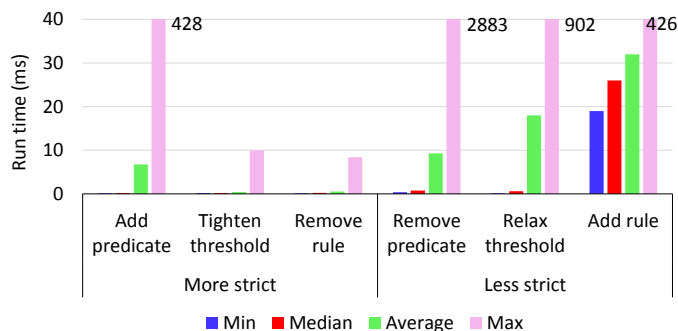


Figure 3.6: Incremental EM run time for different changes to the matching function that make it more/less strict.

Figure 3.5C shows the results for the add-rule experiment. We can see that in the first iteration, both variations are slow. This is because there is no materialized result to use (i.e. the memo is empty). However, from the second iteration onwards we can see that the cost of the precomputation baseline steadily increases whereas the cost of fully incremental is mostly constant and significantly smaller than that of the precomputation baseline. This is because the precomputation baseline performs unnecessary lookups and evaluates all the rules in the matching function. The incremental approach just evaluates the newly added rule and thus it does not slow down as the number of rules increases.

In certain runs both of the variations experience a sudden increase in the running time. These are the cases in which the new rule requires many feature computations, because either there was a new feature, or the feature was not in the memo, and this feature was “reached” in the rule evaluation (it might not be reached, for example, if a predicate preceding the feature evaluates to false.)

Figure 3.6 shows run times for incremental EM for different changes to the matching function. To illustrate how the numbers were generated, assume that we want to measure the incremental run time for adding a predicate. We randomly selected 100 predicates, removed the predicate, ran EM and materialized the results, then added the predicate to the rule, and measured the run time. The rest of the numbers in the table were generated in a similar manner.

For tightening the thresholds, we randomly selected a predicate, and for that predicate we randomly chose one of the values in $\{0.1, 0.2, 0.3, 0.4, 0.5\}$ that could be

applied to the predicate. For example, assume that the predicate is

$$\text{Jaccard}(a.\text{name}, b.\text{name}) \geq 0.6.$$

For tightening the threshold, we add a value to the current threshold. For this predicate we select a random value from $\{0.1, 0.2, 0.3, 0.4\}$, because adding 0.5 to the current threshold makes it larger than 1. If the predicate uses a \geq operation we add the value to the current threshold, and if it uses a \leq operation we subtract the value from the current threshold. The procedure is similar for relaxing thresholds.

We can see that making the matching function more strict by adding a predicate, tightening the threshold, and removing a rule on average takes no more than about 6 milliseconds. On the other hand, making the function less strict could take up to 34 milliseconds on average. This cost is due to the fact that we may need to calculate new features for a fraction of candidate pairs.

3.8.7 Other data sets

In this section, we show the results of our experiments using the other five real-world data sets along with the largest data set that we used in the above experiments (Figure 3.7). The features/rules for each data set is defined using a combination of manual decisions based of domain knowledge, and an automatic approach where we extract rules from a random forest using a labeled sample of candidate pairs. The candidate sets were generated using a standard blocking technique (Q-gram). Table 3.2 describes the details about each data set. In summary, we observe no meaningful difference between the performance of our techniques on these data sets and the performance on the data set presented above.

3.9 Conclusions

We have focused on scenarios where an analyst iteratively designs a set of rules for an EM task, with the goal of making this process as interactive as possible. Our experiments with six real-world data sets indicate that “memoing” the results of expensive similarity functions is perhaps the single most important factor in achieving this goal, followed closely by the implementation of “early-exit” techniques that

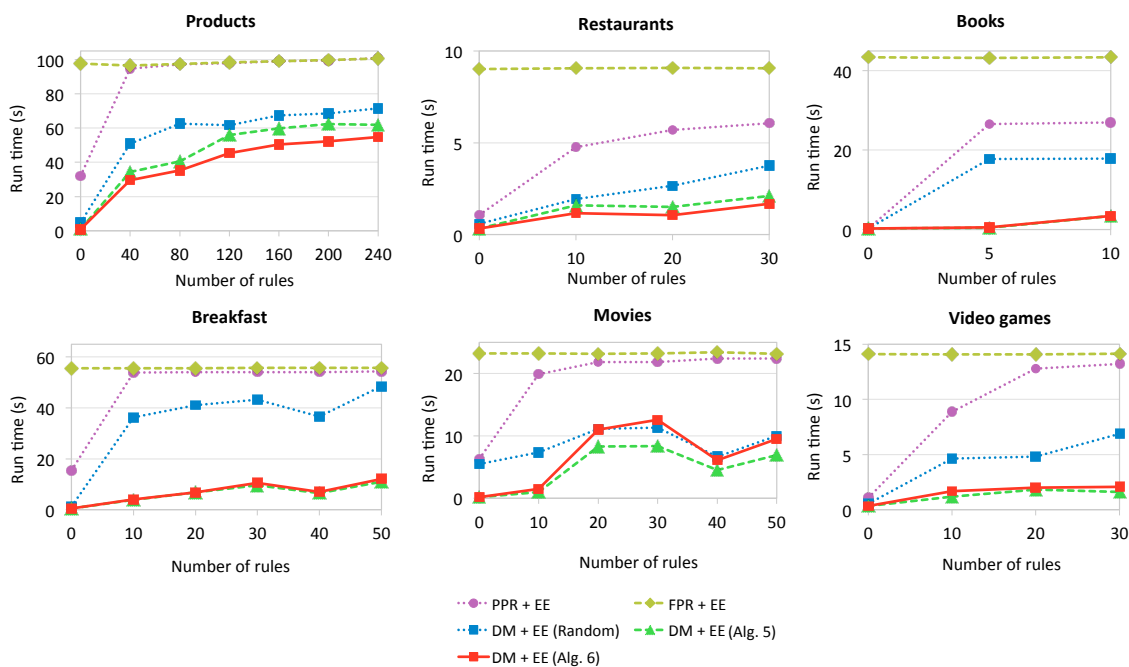


Figure 3.7: Run time for different sizes of matching function for production pre-computation + early exit (PPR + EE), full precomputation + early exit (FPR + EE), and dynamic memoing + early exit (DM + EE) with random ordering and rules ordered by Algorithm 5 and Algorithm 6 for real-world data sets.

stop evaluation as soon as a matching decision is determined for a given candidate pair.

In the context of rule creation and modification it may not be desirable or even possible to fully precompute similarity function results in advance. Our just-in-time “memoing” approach solves this problem, dynamically storing these results as needed; however, the interaction of the on-demand memoing and early-exit evaluation creates a novel rule and predicate ordering optimization problem. Our heuristic algorithms to solve this optimization problem provide significant further reductions in running times over more naive approaches.

Finally, in the context of incremental rule iterative development, we show that substantial improvements in running times are possible by remembering the results of previous iterations and on the current iteration only computing the minimal delta required by a given change.

From a broader perspective, this work joins a small but growing body of literature which asserts that for matching tasks, there is often a “human analyst in the

loop,” and rather than trying to remove that human, attempts to make that human more productive. Much room for future work exists in integrating the techniques presented here with a full system and experimenting with its impact on the analyst.

4 DEBUGGING ENTITY MATCHING DATA SETS

4.1 Introduction

In this chapter, we re-visit the insights that we got from developing abstract analysts in Chapter 2. With the abstract model of the entity matching problem, we asked abstract analysts to create rules for matching for two kinds of data sets. The first kind was the original product data set from an e-commerce industry which included many inconsistencies. We formally defined an inconsistency, but intuitively, inconsistencies are matching pairs of records with low similarity values or non-matching pairs of records with high similarity values. We then created a synthetic and clean version of this data set by removing the inconsistencies. When comparing the abstract analysts performance with these two data sets we had a very interesting (and intuitive) observation. With a clean data set all our analysts can find a set of rules with high precision and recall for matching. However, the matching task becomes hard when we have inconsistencies in the data set. In particular, it becomes very hard to keep precision high while improving recall. In fact, we show that with some data sets it is impossible to achieve perfect precision and recall even if the analyst creates a rule for every matching pair of records.

Following this observation, in this chapter we try to answer the following questions:

1. What causes inconsistency in a data set and how should the analyst act on an inconsistency that she finds?
2. How can we help the analyst to reduce inconsistency in a data set?

Answering the first question helps us focus our attention to different kinds of inconsistency so that we can develop techniques to help the analyst resolve them. Answering the second question helps the analyst in finding and resolving such inconsistencies. It may not be possible to fully identify all causes of inconsistency in data sets. Therefore, in Section 4.2 we will describe the main categories that we have come across working with many data sets and suggest a decision process for the analyst for acting on each category in Section 4.2.6. Following that in Section 4.3 we will propose a framework for identifying and presenting record pairs to the analyst

that helps her in identifying inconsistency causes so that she can resolve them. In Section 4.4 we turn our attention to a particular kind of inconsistency, incorrect labels, and evaluate our proposed framework with respect to this particular cause of inconsistency. We review related work in Section 4.6 and conclude this chapter in Section 4.7.

4.2 Categories of Inconsistency in Data sets

In this section we will describe the categories of inconsistencies that we have come across working with many entity matching data sets. Even though it is not possible to enumerate every cause of inconsistency in matching data sets, we believe that this set of categories occur enough that warrants further attention to them.

4.2.1 The Restaurants Data set

This data set is commonly used for evaluation in entity matching research. It compares restaurant listings from two sources, Fodors and Zagat, containing 528 and 329 records respectively. 4 attributes are available for matching: restaurant name, addr, city, and type. The type of a restaurant identifies its cuisine. There are 111 matching records between the two sources.

4.2.2 Incorrect Label in Ground Truth

As described in Chapter 1, typically, an analyst uses a labeled sample of data in the matching process. In the case of rule-based matching, the analyst views this labeled sample to get a sense of what rules to create. In the case of matchers that are created using machine learning, the labeled sample is used as training and test data. Furthermore, the labeled sample is used for performing evaluation and generating precision and recall numbers as the analyst iterates to generate a high quality matcher. Therefore, the labeled sample is used in different parts of the matching pipeline.

Taking a representative sample of the candidate pairs is a non-trivial task by itself that is discussed in [22, 33] in more detail. What we are interested in this thesis is the labeling process itself. The typical way of labeling is by manually going through the sample and labeling it. If the sample is large multiple analysts may

Record a		Record b		Feature vector	
name	cafe roma	name	palace court	name_JACCARD	0
addr	3570 las vegas blvd. s	addr	3570 las vegas blvd. s.	addr_JACCARD	1
city	las vegas	city	las vegas	city_TRIGRAM	1
type	coffee shops/diners	type	french (new)	type_TRIGRAM	0

Figure 4.1: Incorrect label in the labeled sample for restaurants data set. This pair was labeled as a match but the records do not match.

be assigned to label different chunks of the sample. Similarly, the sample can be generated via crowdsourcing. In the case of very large samples, the labels could be generated via semi-automatic approaches such as using rules.

In all these approaches of labeling a sample, there are opportunities for error. For example, different analysts may have different logic for deciding if a pair of records is a match or not. This problem is even more pronounced if we use crowdsourcing or semi-automatic approaches to label the sample.

Consider the example in Figure 4.1 from the restaurants data set. The two records represent two restaurants that are in the same building and have the exact same address. However, they have different names and types. An analyst may have devised a rule that two records with the exact same address are matching even though these records do not match, leading to the incorrect label.

4.2.3 Missing Feature

Consider the interesting example in Figure 4.2 from the restaurants data set. These two records are actually matching but because the restaurant is located at the corner of Horn Avenue and Sunset Boulevard it has two addresses that appear completely different. In fact, this is a pattern and there are other restaurants in this data set in New York City that have two valid addresses. The analyst, after viewing this pair of records, can resolve this issue by adding a feature that captures the similarity of the addresses using latitude and longitude coordinates of the addresses.

Record a	Record b	Feature vector
name spago	name spago (los angeles)	name_JACCARD 0.33
addr 1114 horn ave.	addr 8795 sunset blvd.	addr_JACCARD 0
city los angeles	city w. hollywood	city_TRIGRAM 0
type californian	type californian	type_TRIGRAM 1

Figure 4.2: Example of missing feature. These restaurants are the same with two valid addresses. The analyst can add a feature comparing latitude and longitude of the addresses.

4.2.4 Inconsistent Extraction Logic or Extraction Error

In some cases, the attributes are extracted with different logic across the sources. For example, consider the record pair in Figure 4.2 again. In one data source the city attribute is extracted with finer granularity, west hollywood, compared to the other data source that has los angeles as the city. Furthermore, the data sources may include extraction error. For example, in one data source, the city attribute from some records may be missing due to an error in the code that extracted the attribute. In such cases, if the analyst has access to the extraction code or the developer that has written the extraction code, she can take action to fix the extraction error or inconsistent extraction logic.

4.2.5 Normalization Opportunity

Consider the following example from a data set that includes books from different sources. Suppose in one data source a book's cover type is specified as "hard cover", and in the other source cover type is "trade cloth" for the same book. These two values refer to the same cover type. This is a normalization opportunity. The analyst then can fix this issue by changing the representations of this value to be the same in both data sources.

4.2.6 Analyst Actions on Inconsistencies

Now, how should the analyst react to each record pair that she inspects? If she performs extensive cleaning on the labeled sample (train) such that the cleanings do not apply to all the data (test) she has wasted time and as we show in the experiments

Algorithm 11: Possible analyst decision process

Input: r , record pair to inspect

- 1 **if** *Incorrect label in ground truth* **then**
- 2 | Fix label in sample.
- 3 **else if** *Systematic extraction error* **then**
- 4 | Modify the extraction code or contact developer.
- 5 **else if** *Systematic normalization opportunity* **then**
- 6 | Modify data sources to have the same representation for the value.
- 7 **else if** *Missing feature* **then**
- 8 | Add feature to set of features.
- 9 **else**
- 10 | Do not make any change to the data or features.

this may in fact lead to lower quality matching output. This notion is similar to over-fitting in machine learning literature. If the model learns too much about patterns in the training data set, it may not perform well on the test data. Therefore, we loosely define *systematic* errors as follows:

Definition 4.1. *Systematic errors are ones for which the analyst can define a fix such that it can be applied to all pairs, training and test, to resolve the issue.*

In simpler terms, a systematic error is one that you see it in the training data, then you can fix it in all data. Recall the example of normalization opportunity above where in one data source a book’s cover type is specified as “hard cover” and in the other as “trade cloth”. These two values refer to the same cover type. It makes sense to suspect that this applies for the rest of the data as well. In fact, the analyst can quickly browse the tables to verify that this is the case. The analyst then can fix this issue by changing the representations of this value to be the same in both tables.

Given the definition of systematic errors, Algorithm 11 shows a possible decision process for the analyst. If she encounters an incorrect label in the ground truth, she can safely fix the error. If an attribute is not extracted and is required for matching, the analyst will modify the extraction mechanism to fix this error or contact the developer responsible for extraction. Also, if she finds that normalizing different representations of a value can help the matching quality, she will modify the data sources accordingly. In some cases, adding a new feature will capture some aspect of similarity that was not captured through another feature. In such a case, the analyst will add this feature. Any record pair that does not fit into these categories

will be left untouched with the hope that the matching algorithm will be able to deal with it.

We speculate that this might be a reasonable algorithm for an analyst to use. We have not evaluated this whole algorithm, and think this is very fertile ground for future work. We will focus on the very first step - "Fix label in sample" - later in this chapter.

4.3 A Framework for Finding and Resolving Inconsistencies

To date, most of the literature on matching algorithms has focused on improving accuracy and reducing run time given that the data to be matched is "fixed" or "frozen." In this thesis, we consider a situation in which data cleaning/debugging and matching is done in an interleaved fashion. That is, can you use information gathered in the matching phase to provide insight into opportunities for cleaning, and then use this in turn to improve the quality of the matching? Specifically, our goal is to identify record pairs that we suspect are causing an inconsistency in the data set and require analyst action and show them to the analyst for inspection. We may refer to the process of finding and resolving inconsistencies as "debugging" or "cleaning" hereafter.

To find suspect pairs of records and show them to the analyst, we leverage the existence of a set of labeled examples for matching. Given two records, denoted as a "record pair", the label states if the record pair is a "matching pair" or a "non-matching pair". Typically a sample of record pairs are labeled for evaluating the matching output and we propose to use the same set of labels in the debugging step. Figure 4.3 shows a labeled sample for a set of record pairs along with the similarity values between the records for two attributes, regarded as *feature vectors*.

To find all debugging opportunities, the analyst can inspect all the pairs of records in the labeled sample. This requires a lot of effort from the analyst, and may be infeasible when the sample is very large. On the other hand, if she randomly selects a subset of the labeled sample and inspects it, she may lose many debugging opportunities, specially if most of the data is clean and consistent. Therefore, we propose to rank the record pairs such that the ones that provide more clues for

Record Pair	Feature Vector	Label
p1:(a1, b1)	(0.5, 1.0)	Match
p2:(a10, b5)	(0.1, 0.1)	Non-match
p3:(a15, b100)	(1.0, 1.0)	Match
p4:(a1002, b13)	(0.75, 1.0)	Non-Match
p5:(a300, b542)	(0.33, 1.0)	Match

Figure 4.3: Labeled sample for pairs of records and their feature (similarity) vector.

debugging come first such that the analyst can find more issues in the data set with less effort (i.e. inspecting fewer number of record pairs).

There are multiple ways of generating rankings, and we found that they can reveal different kinds of issues. We will illustrate this with an example. Let us assume that matches have positive labels and non-matches have negative labels. Consider a matching pair that has a negative label, which is incorrect. Now, how does the analyst go about finding an incorrect label?

One way an analyst finds incorrect labels is after she has performed matching and come up with a matcher. When she evaluates the output of the matcher, she finds that the matcher predicts this pair as a match, but the label is negative. Let us call this approach *False Positives False Negatives (FPFN)*. In this approach, the analyst inspects the false positives and false negatives from the matching output. False positives are pairs of records that are labeled as non-matching but declared as match by the matcher. False negatives are pairs of records that are labeled as matching but predicted non-match by the matcher. What if the matcher also predicts this pair as a non-match? Basically, the matcher agrees with the incorrect label. In such a case, the analyst will not have a chance to view this pair and correct the label.

As the above example showed, relying on the errors of the matcher may not reveal all the debugging opportunities. Therefore, we introduce another approach for finding such opportunities, which we call *Mono* hereafter. In *Mono*, we make use of the following property of the matching problems: matching record pairs typically have higher similarity values than non-matching record pairs. Let us call this property *monotonicity*, and we will define it precisely later. Now for the above example, it is likely that the matching pair with negative label will violate monotonicity with respect to other matching pairs. This means that it is labeled as

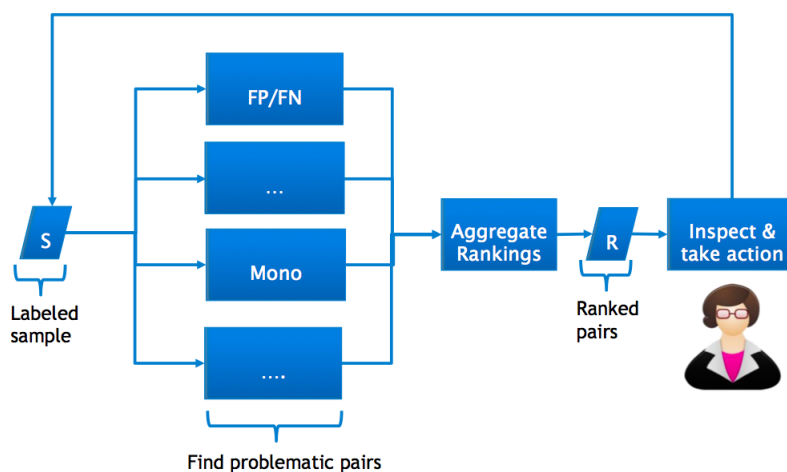


Figure 4.4: A framework for finding and resolving inconsistencies.

a non-match and has higher similarity values than some other matches. In such a case, the analyst can find the incorrect label by inspecting the pairs of records that violate monotonicity.

We can present the output of each of these methods separately to the analyst, but that will waste analyst time because they may overlap. Therefore, we propose to aggregate the rankings from multiple approaches and present the aggregate ranking to the analyst. The hope is that the aggregate ranking will effectively combine the information revealed by each of the rankings and record pairs that reveal a debugging opportunity will be ranked higher in the aggregate ranking.

Figure 4.4 summarizes our approach. Basically, given a labeled sample, there are multiple ways of ranking the pairs of records to reveal debugging opportunities. We then aggregate the individual rankings to produce a final ranking that will be presented to the analyst. The analyst then iteratively and interactively inspects the records pairs, applies fixes to the data set, features, and/or labels in the labeled sample until she runs out of time or cannot find more issues.

Motivating Example As an anecdotal motivating example of the power of this approach, consider the Restaurants data set. This data set has been used often in the matching literature; to the best of our knowledge, no paper has reported any data cleaning or synonym representation problems in this data set. The data set has over 1 million potential matching pairs for which we know the label. The approach we present in this paper, when applied to this data set, immediately returns as the first pair to consider the record pair shown in Figure 4.2. As we discussed in Section 4.2

these two records are actually matching but because the restaurant is located at the corner of Horn Avenue and Sunset Boulevard they have two addresses that appear completely different. The analyst, after viewing this pair of records, can easily clean the addresses by adding a feature that captures the similarity of the addresses using latitude and longitude coordinates of the addresses. It would be very unlikely that the analyst spots this cleaning issue just by randomly looking at over 1 million pairs of records.

In the following sections we will formally describe the ranking and rank aggregation problems. Then we will discuss details of *FPFN* and *Mono*. Following that we will describe the interactive/iterative process that the analyst will go through for inspecting the records in the ranking and resolving the inconsistencies.

4.3.1 Ranking and Rank Aggregation Background

Ranking and rank aggregation has been studied extensively in the literature. Some of these works include [5, 15, 19, 17, 18, 34, 35]. Here we draw from this existing literature and discuss preliminaries regarding ranking and rank aggregation to the extent relevant to this chapter.

Given a universe U of candidates, a ranking with respect to U is a list $\tau = [x_1 > x_2 > \dots > x_d]$ where $x_i \in S \subset U$ and $>$ denotes an ordering relation between the elements.

τ can be a full list, partial list, or a top-k list: If τ contains all the elements in U then it is a full list. In other words, a full list is a permutation of the elements in U . On the other hand, if $|\tau| < |U|$ then τ is a partial list. Top-k lists are a special case of partial lists. In this case, only the first k elements from the top of the ranking are reported and all the other elements are assumed to have a ranking lower than k .

To illustrate further, consider a search engine over the web. The engine is returning a full list if it has access to all pages in the web and it is returning an ordering of the pages to the user. This is rarely the case, since typically an engine has indexed only a subset of the pages in the web. In that case, it has access to a subset S of U and if it returns an ordering of elements in S , we call it a partial list. However, due to performance reasons and the fact that a user typically browses only a limited number of top results, the search engine returns the top k ranked pages from S , which we call a top-k list.

4.3.1.1 Distance measures

Given a set S we can have multiple rankings $\tau_1, \tau_2, \dots, \tau_k$. The question that we address in this section is: how do we measure the distance between two rankings with respect to a set S ? First, let us assume that we only consider full lists and then we will extend the discussion to partial lists. Two popular distance measures for rankings are *Spearman footrule distance* and *Kendall tau distance*. Let $\tau(i)$ denote the ranking of element x_i in τ . Given two full lists σ, τ , the Spearman footrule distance sums up the absolute difference in the rank of element i in the two lists:

$$F(\sigma, \tau) = \sum_i |\sigma(i) - \tau(i)|$$

The Kendall tau distance counts the number of pairwise disagreements between two lists, as follows:

$$K(\sigma, \tau) = |\{(i, j) : i < j, \sigma(i) < \sigma(j) \quad \tau(i) > \tau(j)\}|$$

Basically, the Kendall distance is the number of pairwise swaps required to transform one list to the other list.

Suppose we have several lists $\tau_1, \tau_2, \dots, \tau_k$ and we want to compute the distance of one list σ with respect to the others. We can extend the definition above and compute the normalized footrule distance as follows:

$$F(\sigma, \tau_1, \tau_2, \dots, \tau_k) = (1/k) \sum_i F(\sigma, \tau_i)$$

We can compute the normalized Kendall distance in a similar manner.

Now, let us consider the case where we have partial lists. In such a case, there may be elements that are in one set but not in the other set. Therefore, we cannot apply the formulas above directly. We first need to define the concept of projection:

Given a list τ and a subset S of the universe U , the projection of τ with respect to S , denoted $\tau|_S$, will be a new list that contains only elements from S .

If $\tau_1, \tau_2, \dots, \tau_k$ are partial lists, let U denote the union of elements in $\tau_1, \tau_2, \dots, \tau_k$ and let σ be a full list with respect to U . Now, given σ , the idea is to consider the distance between τ_i and the projection of σ with respect to τ_i . Then, we have the induced footrule distance:

$$F(\sigma, \tau_1, \tau_2, \dots, \tau_k) = (1/k) \sum_i F(\sigma | \tau_i, \tau_i)$$

The induced Kendall distance can be defined in a similar manner.

4.3.1.2 Optimal rank aggregation

In rank aggregation we want to come up with a single ranking that best represents the information we obtain from different ranking methods (for example, search engines). Typically, in doing so we would like to minimize the distance of the final ranking from each individual ranking. Therefore, the term optimal is defined with respect to a particular distance metric. For example, suppose we use the Kendall distance. The question is that given full or partial lists $\tau_1, \tau_2, \dots, \tau_k$ find a σ such that σ is a full list with respect to the union of the elements of $\tau_1, \tau_2, \dots, \tau_k$, and σ minimizes $K(\sigma, \tau_1, \tau_2, \dots, \tau_k)$.

σ obtained from solving the above optimization problem is called the Kemeny optimal aggregation. This particular aggregation has the property of eliminating noise from various ranking schemes. More importantly, it is the only aggregation that simultaneously satisfies natural and important properties of rank aggregation functions, called neutrality, consistency, and the Condorcet property. Neutrality in a voting system means that the system itself does not favor any candidate. Consistency says that if the voters in two arbitrary groups in separate elections select the same candidate the result should not change if the groups are combined. An election method satisfies the Condorcet property if it elects the candidate that would win by majority rule in all pairings against the other candidates, whenever one of the candidates has that property. Unfortunately, computing the Kemeny optimal aggregation is known to be NP-Hard. Therefore, there are many approximations have been suggested in the literature.

Specifically, it is shown that the footrule optimal aggregation can reasonably approximate the Kenemy optimal aggregation [15]:

If σ is the Kemeny optimal aggregation of full lists $\tau_1, \tau_2, \dots, \tau_k$, and σ' optimizes the footrule aggregation, then

$$K(\sigma', \tau_1, \tau_2, \dots, \tau_k) < 2K(\sigma, \tau_1, \tau_2, \dots, \tau_k)$$

rank	τ_1	τ_2	τ_3	σ
1	D	A	C	A
2	A	B	D	D
3	C	C	{A,B}	C
4	{B}	{D}		B

Figure 4.5: Median rank aggregation example.

Fagin et al. [17] propose *Median Rank Aggregation*, which approximates the footrule aggregation by a constant factor of three and consequently approximates the Kemeny optimal aggregation by a constant factor. They found that it works reasonably well in practice and is simple and fast to compute. We will use this method for rank aggregation and will describe it in detail in the following section.

4.3.1.3 Median Rank Aggregation

Median rank aggregation is an approach for finding a constant factor approximation of the optimal aggregation ranking of multiple partial rankings. The algorithm is surprisingly simple: Sort all the candidates based on the median of the ranks they receive from the rankings. Break ties arbitrarily.

Figure 4.5 shows an example with three partial rankings τ_1, τ_2, τ_3 . The last group of candidates in each ranking are ties in each ranking. σ is the optimal ranking. In σ A, D are ties with median ranking of 2, and B, C are ties with median ranking of 3. We break the ties in σ arbitrarily.

4.3.2 False Positives False Negatives (FPFN)

4.3.2.1 Pair selection

One way of finding problematic pairs is to use the labeled sample and learn a matcher using machine learning approaches such as decision trees, random forest, naive bayes and so on. Recall that a false positive is a non-matching pair that is reported as match by the matcher. A false negative is a matching pair that is reported non-match by the matcher. We found that pairs of records that this matcher gets

wrong in terms of false positive and false negatives can reveal issues in the data set and/or set of features used for learning the matcher, hence the name FPFN.

To learn a machine learning model, typically we split the labeled sample to train and test data. Train a model using training data (seen data) and test the accuracy of the model using the test data (unseen data) which was not revealed at training. To reduce variability of the prediction, a well-known approach for evaluating a model is k-fold cross validation. In this case, the labeled sample is divided to k folds and k models are learned using k-1 folds as training and 1 fold as test data in k iterations. The accuracy of the model is then the average accuracy of all the models. k is typically set to 10, meaning that at each iteration 90% of the data is used as training data and 10% is used as test data.

For us to use false positives and false negatives as pairs to show to the analyst, we would like to find such pairs in all the labeled sample and not only in test data. If we split the data into train and test, we will get false positives and negatives from only the test data, since prediction is done only on test data. To achieve this we can use 10-fold cross validation, and union the false positives and false negatives from each iteration. Since every data point is used at least in one fold as test data, every record pair in the labeled sample will have a chance to appear in the final ranking as false positive or false negative.

4.3.2.2 Ranking

Recall that we would like record pairs that reveal more debugging issues ranked higher so that the analyst can quickly debug the most important/prevalent issues. In FPFN we select pairs that appear in false positives or false negatives after learning a model for predicting if a pair is a match or a non-match. Suppose a pair of records shows in the false positives, meaning that it must not be a match but it is predicted as a match by the model. The more confident the model is in predicting this pair as a match, the more suspicious this pair looks like. That is, this is a non-match that has very high similarity values and that is probably the reason the model has learned to predict it as a match with high confidence.

Therefore, if the machine learning model that we learn reports a confidence score for each of its predictions, we can use this score to rank the pairs returned by the pair selection method. Pairs with higher confidence score are ranked on top, and we break ties arbitrarily.

Random forest is one of the models that is amenable to this requirement for a confidence score. In a random forest, multiple decision trees are learned using the same training data and the final prediction is the majority vote between the trees. Basically, if more trees agree on a prediction for a pair, then the model is more confident of its prediction. Therefore, we can use the mean predicted class probabilities of the trees in the forest as the confidence score for each pair. Using a random forest model has the additional advantage that the model is composed of rules that can be understood by humans. Therefore, if necessary the analyst can try to decipher why the model made a decision for a particular pair. Many other learning algorithms lack this property and are like a black box for the analyst. For the above reasons, we will use random forests to generate the rankings for FPFN.

4.3.3 Mono

In Mono, we find suspect record pairs based on the observation that a matching record pair typically has higher similarity values than non-matching record pairs. Particularly, it would be non-intuitive for a non-matching pair to be more similar than a matching pair in all dimensions. This observation was reported in [6] and this is what we found in practice as well.

One way to capture this observation is through the *monotonicity* property for matching data sets introduced in [6] and defined below:

Definition 4.2. Monotonicity *A matching record pair p_1 and non-matching record pair p_2 satisfy monotonicity with respect to a set of defined features \mathcal{F} if there exists at least a feature $f \in \mathcal{F}$ such that $f(p_1) > f(p_2)$.*

Intuitively, the matching pair must at least be more similar than the non-matching pair in one feature. If a data set is monotonic or highly monotonic with respect to a set of features, we can perform matching using a few simple rules [6]. However, we have observed that in many data sets used in practice, specially if they are not cleaned or missing features, monotonicity is violated. Based on this observation, our idea is that monotonicity violations can point to cleaning opportunities or missing features for matching.

The above definition of monotonicity only requires the matching pair to be more similar than the non-matching pair in one dimension. We generalize the notion of monotonicity introduced in [6] as follows:

Definition 4.3. *k-monotonicity* A matching record pair p_1 and non-matching record pair p_2 satisfy *k-monotonicity* with respect to a set of defined features \mathcal{F} if there exists at least k features $f \in \mathcal{F}$ such that $f(p_1) > f(p_2)$.

Note that 1-monotonicity resolves to the original definition of monotonicity, and thus throughout this paper we use 1-monotonicity and monotonicity interchangeably. We have observed that if a non-matching pair is more similar than the matching pair in every dimension, violating 1-monotonicity, it is a strong signal for cleaning opportunities. However, we found this definition rather strict in the general case, specifically if we have many features. As the number of features increase, it is more likely that a matching and non-matching record pair satisfy 1-monotonicity even if they do require cleaning. Suppose we have 15 features for a data set, it seems counter-intuitive if a matching pair is not more similar than a non-matching pair even in 2 dimensions, violating 2-monotonicity. We found that in such cases, record pairs that violate 2-monotonicity can also be a good signal for a cleaning opportunity, although a weaker signal than 1-monotonicity violations. In general, as k increases, violations of k -monotonicity are weaker and weaker signals of cleaning issues.

In practice, the analyst can only look at a limited number of record pairs before moving forward with the matching process. Therefore, we would like to rank record pairs such that ones that need cleaning issues come first. Therefore, given a limit for the number of pairs L that the analyst will inspect at each round, our problem is to include as many record pairs that need cleaning as possible in the top- L record pairs. We use monotonicity violations as a signal that a record pair needs cleaning.

To do so, we first need to find all monotonicity violations. If we have a large labeled sample finding all monotonicity violations may take a long time or be prohibitively expensive. For example, assume that we have 100000 labeled record pairs out of which 40000 are matches and 60000 are non-matches. With an exhaustive search we need to compare 2400 million pairs of matching and non-matching records which can take hours to complete. Therefore, we need to devise a mechanism where we can quickly find all monotonicity violations. We develop a spatial blocking mechanism to find violations quickly and describe it in Section 4.3.3.1.

The notion of k -monotonicity helps us find and rank violations from stronger signals to weaker signals. However, there may be thousands of violations. Not all record pairs involved in a violation are interesting and require cleaning. For example, assume a matching record pair that has similarity zero in all dimensions.

Algorithm 12: Exhaustive search for finding monotonicity violations.

Input: $\mathcal{P} = \{p_1, p_2, \dots\}$ list of positive pairs
 $\mathcal{N} = \{n_1, n_2, \dots\}$ list of negative pairs
 \mathcal{FV} hash table containing feature vector for each pair
Output: $\mathcal{V} = \{(p_i, n_j) \mid (p_i, n_j) \text{ violate monotonicity}\}$

```

1  $\mathcal{V} = [];$ 
2 foreach  $p \in \mathcal{P}$  do
3   | foreach  $n \in \mathcal{N}$  do
4   |   |  $\text{pos\_feature\_vec} = \mathcal{FV}.\text{get}(p);$ 
5   |   |  $\text{neg\_feature\_vec} = \mathcal{FV}.\text{get}(n);$ 
6   |   | if  $\text{dominates}(\text{neg\_feature\_vec}, \text{pos\_feature\_vec})$  then
7   |   |   |  $\mathcal{V}.\text{append}((p,n));$ 
8   |   | end
9   | end
10 return  $\mathcal{V};$ 

```

This may be due to a ground truth error, meaning that this record pair is actually not a match and the label was incorrect. This record pair may violate monotonicity with respect to thousands of non-matching record pairs that are more similar than this pair. Now, which record pairs should we propose to the analyst to look at? Intuitively, this particular record pair should be presented to the analyst, and the rest of the pairs that violate monotonicity with respect to this record pair do not necessarily need cleaning. So here the question is how do we select a subset of record pairs to show to the analyst?

Here we borrow ideas from the data repair literature where we have a set of data quality rules (DQRs) that need to be satisfied, and we want to make the minimum number of changes to the data such that all violations are resolved [4, 10, 47]. Here, our data quality rule or constraint is that matching pairs must be more similar than non-matching pairs in at least k dimensions (satisfy k -monotonicity). Consequently, we want to make minimal changes to the data sets such that all monotonicity violations are resolved. We use this idea to select record pairs to show the analyst which is described in Section 4.3.3.3.

4.3.3.1 Spatial blocking

So far we have established that we are interested in monotonicity violations because they can point us to record pairs that we suspect will point us to cleaning opportu-

Algorithm 13: Spatial blocking for finding monotonicity violations.

Input: $\mathcal{P} = \{p_1, p_2, \dots\}$ list of positive pairs
 \mathcal{FV} hash table containing feature vector for each pair
 \mathcal{S} hash table containing negative pairs in each square
Output: $\mathcal{V} = \{(p_i, n_j) \mid (p_i, n_j) \text{ violate monotonicity}\}$

```

1 foreach  $p \in \mathcal{P}$  do
2    $p\_square = \text{getSquare}(p)$ ;
3   foreach  $s \in \mathcal{S}$  do
4     if strongly_dominates( $s, p\_square$ ) then
5       foreach  $n \in \mathcal{S}.get(s)$  do
6          $\mathcal{V}.append((p,n))$ ;
7       end
8     if  $s == p\_square$  or dominates( $s, p\_square$ ) then
9       foreach  $n \in \mathcal{S}.get(s)$  do
10         $pos\_feature\_vec = \mathcal{FV}.get(p)$ ;
11         $neg\_feature\_vec = \mathcal{FV}.get(n)$ ;
12        if dominates( $neg\_feature\_vec, pos\_feature\_vec$ ) then
13           $\mathcal{V}.append((p,n))$ ;
14        end
15      end
16 end
17 return  $\mathcal{V}$ ;

```

nities or missing features. The problem that we attempt to address in this section is: how do we efficiently find all monotonicity violations?

To find monotonicity violations in a sample of labeled pairs, we can evaluate monotonicity of all matching record pairs with respect to all non-matching record pairs. We can implement this process as an exhaustive search presented in Algorithm 12. This algorithm has complexity of $\mathcal{O}(N \times P)$ where N is the number of non-matching examples and P is number of matching examples in the labeled sample. This solution is suitable for small labeled samples but can be prohibitively expensive in cases where we have samples containing a large number of matching and non-matching pairs.

Fortunately, many matching and non-matching record pairs are obviously monotonic. For example, consider Figure 4.6. For the labeled sample in this figure, we have two features represented by the horizontal and vertical axis. Record pairs labeled as matches are denoted by a star and non-matches are denoted by crosses.

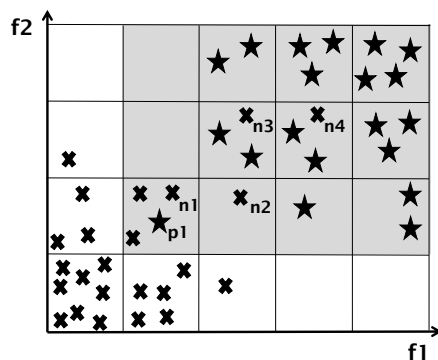


Figure 4.6: Labeled sample shown in the space of two features f_1 , f_2 . Stars denote matching pairs and crosses denote non-matching pairs. The space is divided into squares. Only negative points in the gray area can violate monotonicity with respect to matching pair p_1 .

Matching pairs on the top-right of the figure are obviously monotonic with respect to record pairs on the bottom-left of the figure since they have higher similarity in both features f_1 , f_2 . We can use this special structure of the matching problem to avoid unnecessary checks for monotonicity violations. We call this approach *Spatial Blocking*. It is “spatial” because it uses the feature space to avoid comparing obviously monotonic pairs. We also call it “blocking” because it is similar to blocking in the context of matching where we want to avoid computing expensive similarity functions on obviously non-matching record pairs.

Details of the algorithm are presented in Algorithm 13. Assume that we have n features defined by the analyst. Intuitively, we divide the feature space into equal-volume squares. We then assign each negative point to a square, and keep track of the negative points in each square in a hash table. We denote a square s in d dimensions with its lower left and upper right corner $s = (ll, ur)$ where $ll, ur \in [0, 1]^d$.

Definition 4.4. A point in space $p_1 \in [0, 1]^d$ dominates $p_2 \in [0, 1]^d$ if $\forall d_i \in dp_1[d_i] \geq p_2[d_i]$

Definition 4.5. A point in space $p_1 \in [0, 1]^d$ strongly dominates $p_2 \in [0, 1]^d$ if $\forall d_i \in dp_1[d_i] \geq p_2[d_i], \exists d_i \in dp_1[d_i] > p_2[d_i]$.

Definition 4.6. A square $s_1(ll_1, ur_1)$ dominates/strongly dominates another square $s_2(ll_2, ur_2)$ if ur_1 dominates/strongly dominates ll_2 .

For each positive pair like p_1 in Figure 4.6, we only need to check monotonicity with respect to negative pairs in the same square, here n_1 , and negative pairs in squares that dominate this square, here n_2 . Negative pairs in squares strongly dominating this square, here n_3, n_4 , will obviously violate monotonicity with respect to this positive pair. Negative pairs in squares dominated by this square will be obviously monotonic with respect to the positive pair.

For each square, we will store all the negative record pairs that fall into that square, and this requires $O(N)$ storage since we do not have overlapping squares. If we are memory-constrained, an alternative is to create the hash table over positive pairs, which are typically much less than negative pairs, and loop through negative pairs to find violations.

We define the granularity of the squares using the parameter $slength \in (0, 1]$ which specifies the length of each side of the square. The larger $slength$ the less points in each square, and this will reduce number of unnecessary monotonicity checks. On the other hand, this increases the complexity of finding all dominating squares. The number of features also affects the decision of $slength$. Suppose we have only one feature. In that case $slength = 1$ reduces to exhaustive search. The more features that we have, the larger we can have $slength$. In our experiments we set $slength$ to 0.25 and we found that it performed well with respect to a variety of data sets with different number of features.

4.3.3.2 Finding k -monotonicity violations

We may also be interested in finding violations of 2-monotonicity, 3-monotonicity and so on as they can point to a cleaning issue for the record pair. Given the maximum K that we are interested in finding the violations for, denoted as K , we can find all violations of $1 \dots K$ -monotonicties in one path using both exhaustive search or spatial blocking. We will have K separate lists to store violations for each k -monotonicity and insert violations to the lists as we find them. Note that a non-matching and matching pair that violate k -monotonicity will definitely violate $(k + 1)$ -monotonicity. We only store violations in one list and do not repeat them in lists with higher k .

Algorithm 14: Select and rank record pairs given monotonicity violations

Input: $\mathcal{V} = \{(p_i, n_j) \mid (p_i, n_j) \text{ violate monotonicity}\}$
Output: $\mathcal{R} = \{r_1, r_2, \dots\}$ list of selected and ranked record pairs

- 1 $\mathcal{P} \leftarrow \{p_i \mid \exists (p_i, n_j) \in \mathcal{V}\}$
- 2 $\mathcal{N} \leftarrow \{n_j \mid \exists (p_i, n_j) \in \mathcal{V}\}$
- 3 $\text{Vertices} = \{\mathcal{P} \cup \mathcal{N}\}$
- 4 $\text{Edges} = \{(p_i, n_j) \mid (p_i, n_j) \in \mathcal{V}\}$
- 5 $\mathcal{G} = (\text{Vertices}, \text{Edges})$
- 6 $\mathcal{M} \leftarrow$ Minimum vertex cover of \mathcal{G} .
- 7 $\mathcal{R} \leftarrow$ Rank \mathcal{M} such that records involved in more violations are on top.
- 8 **return** \mathcal{R} ;

4.3.3.3 Pair selection

In the previous section, we described how to efficiently find all monotonicity violations in a matching data set. There may be thousands of matching and non-matching record pairs involved in monotonicity violations, but not all of them are record pairs that are worth the analyst time for inspection. For example, suppose that we have a ground truth error, such that a non-matching record pair which has low similarity values is labeled as a match. In such a case, this record pair may violate monotonicity with respect to thousands of non-matching record pairs, which may be perfectly fine record pairs that do not need inspection. So the question is how do we select/rank a set of record pairs to show to the analyst for inspection?

We make use of similar concepts in the data repair literature [10, 47]. For data repair, given a database D , and a set of data quality rules (DQRs), we would like to minimally repair D such that all violations of the DQRs are resolved. Following the same concept, we define the following data quality rule for our matching data set:

Definition 4.7. Matching Data Quality Rule (DQR): *All matching and non-matching record pairs must satisfy the monotonicity property.*

Therefore, all monotonicity violations violate this matching DQR. Similarly, we would like to minimally change our data set such that all monotonicity violations are resolved. Intuitively, the minimal change hints to minimum analyst effort for cleaning the matching data sets.

In order to achieve this goal, we would like to find the *Maximal Consistent Dataset*, which is basically the largest set of record pairs for which there is no monotonicity

violations. In other words, we want the analyst to inspect the minimum number of record pairs such that if they are removed from the data set all of monotonicity violations are resolved.

We formulate this problem as finding the *Minimum Vertex Cover* of a bipartite graph generated from the set of all monotonicity violations. This solution is shown in Algorithm 14.

Definition 4.8. *The Minimum Vertex Cover of a graph is the minimum set of vertices such that each edge of the graph is incident to at least one of these vertices.*

Given the set of negative pairs \mathcal{N} and a set of positive pairs \mathcal{P} that are involved in at least one violation, $\{\mathcal{P} \cup \mathcal{N}\}$ are vertices in our graph \mathcal{G} . We then add an edge between every matching pair p_i and non-matching pair n_j that violate monotonicity. Since there are no edges among matching pairs or non-matching pairs, we have a bipartite graph.

For our problem, finding the minimum vertex cover of our bipartite graph means that we find the minimum set of pairs that if removed from the data set are monotonicity violations are resolved. Finding the minimum vertex cover a general graph is an NP-Hard optimization problem. However, for the special case of a bipartite graph a polynomial solution exists.

4.3.3.4 Pair selection using k-monotonicity violations

Recall from Section 4.3.3 that violations of k-monotonicity can signal cleaning issues, with lower k's being a stronger signal and higher k's weaker signals. Also, recall that an analyst will look at a limited number of pairs L in each iteration before making a cleaning decision. In the previous section, we discussed how to modify the spatial blocking algorithm to return lists of $\{1 - K\}$ monotonicity violations and that in some cases we can update these lists incrementally as the analyst performs cleaning operations.

Now, the question is how to consolidate and use all these signals in finding the best pairs to show to the analyst, given that the analyst will inspect only a maximum of L record pairs. In algorithm 14 we only have one type of signal, and that is 1-monotonicity violations. Here we would like to use all $\{1 - K\}$ monotonicity violations as a signal.

To do so, we will give these lists one by one, from lower k to higher k , as input to Algorithm 14. For each k , the algorithm will return to us the set of record pairs to be inspected by the analyst corresponding to those violations. We will repeat this process until we have processed all K lists or we have reached the maximum number of pairs L that the analyst will inspect. Note that with this process, we have ordered the record pairs such that record pairs with stronger signals (lower k 's) are shown to the analyst before weaker signals (higher k 's).

4.3.3.5 Ranking

Just as with FPFN, all pairs selected by Mono are not equally interesting. We expect some to reveal more debugging opportunities for the analyst. Therefore, we still need to decide on the ranking of the pairs. To do so, we will use the bipartite graph generated based on the monotonicity violations for ranking the selected record pairs. Pairs are ranked such that the ones with highest number of connections are ranked higher. The intuition behind this method is that record pairs that are involved in many monotonicity violations are more suspect and need to be inspected by the analyst higher in the ranking. Algorithm 14 shows the process of selecting a set of record pairs and ranking them for showing to the analyst.

4.3.4 Other heuristics

We can use other heuristics for ranking the record pairs as well. For example, using the summation of the values of feature vectors is a simple heuristic to find problematic pairs. Basically, matching pairs with lowest summation and non-matching pairs with highest summation could be suspect record pairs.

One limitation of this approach is that it considers matching pairs and non-matching pairs in isolation. For example, non-matching pairs with highest summation are not necessarily suspect record pairs. If there is not any matching pair that is less similar than this non-matching pair, it is likely that there is no issue with this pair. Even if there are matches that are less similar than this pair, it is not clear that the issue is with this non-matching pair and not those matching pairs. Also, since matching and non-matching pairs are treated in isolation, this approach generates two rankings that should be inspected by the analyst and may increase analyst effort.

Furthermore, we empirically found that this approach can be sensitive to the data and the set of features that are selected and produce inferior rankings to FPFN and Mono. For example, suppose that our set of features contains redundant features. Intuitively, two features are redundant if they provide the same information. For example, if one of the features shows high similarity the other feature also shows high similarity. In such a case, these set of redundant features can drive the summation of the feature values high or low and negatively affect the ranking. This is avoided in FPFN and Mono to some extent because the score for each pair for ranking is not decided on in isolation and is relative to the other pairs of records.

4.3.5 Hybrid

The idea behind the Hybrid approach is that each of the methods for ranking the pairs could surface different issues in the data set. Therefore, in the Hybrid ranking we make use of multiple rankings, here FPFN and Mono. We first generate the ranking for each, and then aggregate them to generate a single ranking to present to the analyst. We use median rank aggregation as described in Section 4.3.1.3. The median of two values is the average of the two values. Therefore, the median rank aggregation for two rankings is equivalent to taking the average ranking for each pair, and then sorting the pairs from higher to lower average ranking, breaking ties arbitrarily.

4.3.6 Interactive Debugging

So far we have generated a ranking to present to an analyst for inspection. Now, how does the analyst interact with this ranking? Specifically, what happens after the analyst proposes a fix to the data set and/or features? After the analyst makes a change to the dataset it may not make sense for the analyst to go through the same ranking again. Consider the following example. After viewing a matching pair that has low similarity values in the book type attribute, the analyst proposes to normalize the values in table A from “trade cloth” to “hard cover” to be consistent with table B. After this fix, not only this pair will have higher similarity values, but also many other pairs with the same issue will have higher similarity values. If such pairs are shown to the analyst in the ranking, it will be a waste of analyst time.

Algorithm 15: Interactive_Clean(\mathcal{L} Labeled Sample, \mathcal{F} Features)

```

1  $\mathcal{FV} \leftarrow$  Generate feature vectors
2  $\mathcal{R} \leftarrow$  Generate the Hybrid ranking
3  $\mathcal{J} \leftarrow \emptyset$  Inspected records
4 while User wants to continue and  $\mathcal{R}$  not empty do
5   User inspects next top-k record pairs  $r_{1-k} \in \mathcal{R}$ 
6    $\mathcal{J} \leftarrow \mathcal{J} \cup r_{1-k}$ 
7   if User performs cleaning or changes feature set then
8      $\mathcal{FV} \leftarrow$  update features vectors
9      $\mathcal{R} \leftarrow$  Generate the Hybrid ranking
10     $\mathcal{R} \leftarrow \mathcal{R} - \mathcal{J}$ 

```

Therefore, we re-generate the ranking after each fix that the analyst proposes. When we re-generate the ranking the new and old rankings may overlap. Suppose the analyst proposes a fix to the data set after viewing 10 pairs. The new ranking may rank the same 10 pairs on top. Thus we want to avoid presenting them again to the analyst. Therefore, in the new ranking we will eliminate the pairs that the analyst has already viewed. This way, the analyst can start again from top of the list and inspect the pairs.

We show this process in Algorithm 15. The analyst is presented ranked pairs of records to inspect. She will inspect the pairs and propose fixes. Once a fix is applied, the ranking is re-generated and pairs already inspected are removed from the ranked list. She continues this process until she runs out time or she inspects all the pairs. Given that the number of pairs in the labeled sample can be large, we suspect that the analyst will stop after she does not find any other issue to fix after inspecting a number of record pairs.

4.4 Experimental Evaluation

4.4.1 Ranking methods

In this section, we would like to evaluate our framework for ranking and presenting pairs of problematic record pairs to the analyst. We do not have access to real analysts for evaluation purposes, and so we will mimic the analyst behavior by the following procedure.

According to our interactive cleaning process described in Algorithm 15 we simulate an analyst looking at the first k pairs, fixing issues found in those pairs, re-ranking, then continuing. k is a parameter in our experiments and we set it to 20 for the results reported here. This will continue until the analyst is available and there are more pairs to view. Typically, there are hundreds of pairs in the labeled sample, and so we suspect that the first condition will terminate the loop.

Now, in our experiments, how do we simulate if an analyst is still available? We assume that the analyst will stop if she finds no more problematic pairs. The main point of ranking record pairs is that the ones that are problematic are ranked higher than non-problematic ones. Therefore, after a certain point, the analyst will run out of problematic pairs and start inspecting mostly non-problematic pairs. In that case, she will stop inspecting more pairs. Specifically, we will stop inspecting more pairs if we go through d iterations without finding any more problematic pairs. For smaller data sets generated by students, we set d to 3. For Cora which is larger and has more errors, we set this to 15.

When evaluating a ranking, we are interested to know how much the ranking was able to help the analyst in cleaning a data set. This is a challenge for evaluation, especially without access to real analysts. The challenge arises from two facts: First, there are many categories of issues in a data set that require action from the analyst, and a pair of records may hint to multiple cleaning issues. For example, one pair might point to a missing feature and also need normalization on an attribute. How do we know what cleaning issues do a pair of records suggest to an analyst?

The second challenge for evaluating a ranking is to measure how much the data set has been cleaned based on an action that the analyst takes. For example, if the analyst normalizes a value across two tables, how important is this cleaning for matching? It is challenging to measure this importance directly. One proxy for measuring it is to measure the quality of the matcher that an analyst comes up with after cleaning the data set. Intuitively, the better the data is cleaned, the higher the matching quality is. We show this correlation in Section 4.4.2.2. For evaluation purposes we need to decide on the kind of matcher that the analyst uses among many possibilities, which adds another dimension to the problem.

As it has become apparent, fully evaluating the effectiveness of a ranking in revealing issues in a data set is non-trivial. Therefore, in order to make evaluation tractable, in this thesis we focus on one category of inconsistency in the data set and

will consider evaluating the ranking for the rest of the categories for future work. In particular, we are interested to evaluate our framework with respect to finding incorrect labels in the labeled sample. As we mentioned in Section 4.2, there can be multiple causes for incorrect labels in the labeled sample, such as dividing the labeling work between multiple analysts or using semi-automatic approaches. We also discussed that incorrect labels in the labeled sample may lead to learning a lower-quality matcher as well as misleading us when evaluating the precision and recall of a matcher. Therefore, it is important to fix the labels as much as possible.

Focusing on finding pairs of records with incorrect labels will make evaluation tractable. First, when viewing a pair of records, either it is a pair of record with an incorrect label or a correct label. Therefore, it is clear if the pair is problematic or not. Second, in order to evaluate effectiveness of a ranking and measure how much the ranking has helped the analyst in cleaning the data set, we can count the number of incorrect labels that the ranking revealed to the analyst. Basically, a ranking will be more effective if it ranks more records with incorrect labels on top.

4.4.1.1 Introducing errors in data sets

For each of the data sets, we have manually created a version of the labeled sample that is free of errors. We need this version because we would like to know how many errors are revealed by each ranking method. The samples that were generated by the students were small and contained only a few labeling errors. Therefore, we were not able to meaningfully compare Mono, FPFN, and the hybrid rankings in terms of how many errors they revealed. Therefore, in order to better understand the difference between these approaches, we introduce synthetic errors to the labels and repeat the experiments.

To inject synthetic errors in the labels we flip the labels in the corrected version of the labeled sample to be incorrect. We initially selected a random set of pairs and flipped their labels. We found that this method did not reveal many differences in the rankings. Let us illustrate this further. Recall that most of our pairs of records in the labeled sample actually are clean and do not need analyst action. For example, consider a matching pair in the restaurant data set with exactly the same name, address, city and type. Now, if we flip the label for this pair from match to non-match, it will be a very obvious problematic pair. Basically, it is a non-matching

pair with high similarity value in all the attributes. This will be picked up by all our ranking mechanisms and ranked high.

Randomly flipping the labels for a pair of records in order to inject error does not make a lot of sense in practice as well. Consider the above example, the analysts will probably not make a mistake on such an obviously matching pair. Therefore, it makes sense to flip the label for pairs that their match status is not obvious. We suspect that the analyst would have a harder time on deciding a label for this pair.

In order to find pairs of records that are hard or tricky for analysts to label, we use state-of-the-art active learning methods. Active learning is typically used for generating a labeled sample for training a machine learning algorithm. In such a scenario, the advantage of machine learning is that the user needs to label fewer data points for learning a model. In active learning, the user labels a few seed data points and feeds that to the model. The model then will query the user to label more data points that it thinks would be helpful for learning an accurate model. There are multiple strategies for selecting a data point to show to the user label. In our experiments we use “uncertainty sampling”, which means that the model asks the user to label points that it is most uncertain about. We assume that if the machine learning model cannot confidently decide on a label for a pair, it is hard for the analyst to label that pair as well. That means that we take the points that the model is most uncertain about from active learning and flip their label to incorrect to inject labeling errors into the data set.

4.4.1.2 Results

To evaluate effectiveness of different ranking mechanisms in finding incorrect labels we use six real-world entity matching data sets generated by students at a graduate-level data modeling course at UW-Madison. We also use the Cora data set that is used for evaluations in entity matching literature. Details of the data sets can be found in Table 4.1.

For each of the seven data sets, we report how many incorrect labels were found once our code stops according the stopping criteria defined above. We also report how many pairs were inspected before stopping inspecting more pairs. We consider three rankings in our experiments, Mono, FPFN, and Hybrid. Hybrid is generated by aggregating the rankings in Mono and FPFN via median rank aggregation described in Section 4.3.1.3.

Data set	Source A	Source B	Table A size	Table B size	Candidate Set Size	Labeled sample size (+, -)
Beer*	Beer Advocate	Rate Beer	4345	3000	4334961	450 (68, 382)
Bikes*	Bikedekho	Bikewale	4785	9002	8009	450 (130, 320)
Books*	Amazon	Barnes & Noble	3506	3508	2017	374 (232, 142)
Citations*	Google Scholar	DBLP	3122	12418	5882623	1417 (92, 308)
Movies*	Rotten Tomatoes	IMDB	7390	6407	78079	600 (190, 410)
Restaurants*	Yellow Pages	Yelp	11840	5223	5278	400 (130, 270)
Cora	Cora search engine	Cora search engine	1295	1295	379748	20000 (932, 19068)

Table 4.1: Data sets used in the experiments for evaluating effectiveness of rankings in finding incorrect labels. Data sets generated by students are marked with a *. The number of positive and negative labels for each data set is listed in the paranthesis after the sample size.

As described, we created a corrected version of the labeled sample for our data sets. Table 4.2 for each data set shows size of the labeled sample and the total number of incorrect labels that we found in the labeled sample. For the data sets generated by the students there are a few hundred pairs in the labeled sample and only a few of their labels were incorrect. Cora data set is larger and we found many more labeling errors in this data set.

In the same table we also report the number of errors that our program found with each ranking and the number of pairs that our program inspected before it stopped. We find that for all data sets we find more incorrect labels using Hybrid than each of the methods alone. For Bikes and Cora, FPFN and Mono find a different set of labels even though they may overlap. This is encouraging as it supports our assumption that multiple rankings can reveal different errors and that aggregating them will generate a ranking that can be more effective than each alone.

Data set	Sample size	Incorrect labels	Mono	FPFN	Hybrid
Beer	450	2	0, 8	0, 10	0, 13
Bikes	450	8	2, 25	4, 132	5, 153
Books	374	4	2, 14	3, 15	3, 21
Citations	400	0	0, 4	0, 7	0, 8
Movies	600	1	0, 8	0, 7	0, 12
Restaurants	400	0	NA	NA	NA
Cora	20000	116	51, 104	78, 180	90, 194

Table 4.2: Performance of simulated analyst with different rankings for errors in the original labeled sample. For each of the rankings the first number indicates number of errors found and the second number indicates number of inspected pairs.

For the six data sets generated by students, since there were only a few errors in the original labeled samples, we use active learning and select a subset of the pairs to flip their label to be incorrect. The active learning mechanism may choose matching or non-matching pairs for us to flip their labels. Therefore, we also report in each case how many matching (positive, denoted by +) or non-matching (negative, denoted by -) labels were chosen by active learning and flipped. For each of the data sets, we report five trials with different seed data points to the active learning method. For each of the rankings Mono, FPFN, and Hybrid we report number of incorrect labels found as well as number of inspected pairs to find those errors. We report these results in tables 4.3 and 4.4.

One issue to note is that we suspect that this method of selecting pairs for flipping their labels using active learning may favor the FPFN approach over Mono, and this could explain why in many cases the FPFN approach is able to find more errors than Mono. Recall that using active learning we will select pairs of records that are not obvious for the machine learning model if they are matches or non-matches. This means that the model was not able to classify these pairs of records easily to start with. Therefore, it is likely that they will show up as false positives and false negatives and thus be included in the FPFN ranking, and consequently we will be able to find them in that ranking.

Experiment	Flipped labels (+, -)	Mono	FPFN	Hybrid
Beer				
1	3, 19	10, 14	17, 24	17, 25
2	3, 19	7, 8	16, 25	16, 25
3	7, 15	12, 14	15, 21	17, 28
4	5, 17	16, 21	15, 21	17, 24
5	3, 19	17, 23	12, 17	17, 27
Bikes				
1	8, 14	3, 26	12, 118	13, 122
2	9, 13	5, 28	13, 112	14, 132
3	8, 14	3, 25	11, 100	15, 131
4	9, 13	2, 24	10, 93	12, 117
5	10, 12	4, 26	11, 100	13, 124
Books				
1	13, 5	8, 14	13, 24	13, 27
2	13, 5	5, 11	12, 23	12, 23
3	12, 6	5, 10	13, 20	13, 26
4	15, 3	3, 11	14, 23	13, 26
5	13, 5	2, 11	14, 22	13, 28

Table 4.3: Performance of simulated analyst with different rankings with synthetic errors introduced using active learning (for Beer, Bikes, and Books data sets). For each ranking we report number of incorrect labels found and number of inspected pairs separated by a comma.

On the other hand, this method of selecting pairs does not favor the Mono approach. Recall that the idea behind the Mono approach is that matching pairs typically have higher similarity values than non-matching pairs. Therefore, using Mono, we are hoping to find matching pairs with high similarity values or non-matching pairs with low similarity values. On the other hand, the pairs selected by active learning are likely to be borderline cases where it is not clear that they

Experiment	Flipped labels (+, -)	Mono	FPFN	Hybrid
Citations				
1	7, 13	7, 14	17, 20	17, 22
2	4, 16	11, 19	18, 21	18, 23
3	4, 16	10, 12	18, 21	18, 22
4	4, 16	8, 11	18, 21	18, 26
5	3, 17	8, 17	19, 23	19, 31
Movies				
1	12, 18	2, 3	28, 32	30, 40
2	15, 15	12, 15	25, 33	29, 41
3	13, 17	9, 10	28, 35	28, 37
4	11, 19	8, 13	28, 36	28, 41
5	14, 16	10, 14	28, 37	28, 38
Restaurants				
1	3, 14	5, 7	13, 19	13, 19
2	4, 16	4, 7	7, 11	11, 16
3	6, 14	11, 15	17, 20	20, 26
4	7, 13	6, 8	17, 20	17, 21
5	7, 13	10, 11	16, 18	15, 16

Table 4.4: Performance of simulated analyst with different rankings with synthetic errors introduced using active learning (for Citations, Movies, and Restaurants data sets). For each ranking we report number of incorrect labels found and number of inspected pairs separated by a comma.

are matches or non-matches and thus have a mix of high and low similarity values. This makes it harder for the Mono approach to find these pairs of records.

Furthermore, we can see that in some cases with Mono the analyst stops earlier than with the FPFN approach and is able to find fewer number of errors. Recall that with Mono, we select the minimum number of pairs to show to the analyst and thus some suspect pairs may have been removed by the minimum vertex cover algorithm that chooses a subset of pairs to show to the analyst.

Nevertheless, in 12 out of 30 trials Mono and FPFN find different sets of errors and using the Hybrid approach the analyst is able to find more incorrect labels than any individual approach. This confirms our observations with the original data set that the analyst can find more errors by aggregating multiple rankings.

Results presented in this section were joint work with Haojun Zhang and with guidance of Professor AnHai Doan.

4.4.2 Impact of cleaning

We have observed that when the data is clean, matching can be done with higher quality with “simpler” matchers. To illustrate this observation we propose a set of cleaning operations for three datasets commonly used in the entity matching literature, Cora, Restaurants, and Products. Please refer to Figure 4.6 for details of the data sets. We find these cleaning operations by closely inspecting the record pairs in these data sets. In fact, we used the Mono approach to generate a ranking of suspect record pairs and help us with finding cleaning opportunities. Figure 4.8 shows these cleaning operations.

To evaluate impact of cleaning on different matchers we consider one rule-based matcher and one machine learning matcher. Both of these approaches use a set of labeled samples to create a matcher. Rule-based approaches are popular because they are easy to understand and debug for analysts. For rule-based, we explore SJU operator trees that contain similarity joins and unions [6]. For machine learning, we use random forest that is shown to generate accurate models for matching in previous work [22]. We will explain these matchers below.

Rule-based SJU operator trees were proposed in [6]. Suppose we have defined z features over the attribute pairs \mathcal{M} . The set of all defined features is denoted as $\mathcal{F} = \{f_1, \dots, f_z\}$. Recall that we name the set of feature values for all features for a record pair a *feature vector*.

We have a set of rules $\mathcal{R} = \{r_1, \dots, r_n\}$. Each rule is in Conjunctive Normal Form (CNF), with each clause containing exactly one predicate in the form of $f_i \geq t_i$ where t_i is a threshold $\in [0, 1]$

A single rule is shown below:

$$\text{title_exact_match} \geq 1.0 \wedge \text{author_exact_match} \geq 1.0$$

A pair of records is a match if any rule in \mathcal{R} evaluates to true. Therefore, \mathcal{R} is a disjunction of rules:

$$R = r_1 \vee r_2 \vee \dots \vee r_n.$$

A rule set will output a set of matching pairs $O \subseteq P$ (all candidate pairs).

Given a set of feature vectors for the labeled examples, we can search the feature space to find a SJU operator tree with high matching quality. The algorithm has three tuning parameters:

- d , maximum number of predicates in each rule
- K , maximum number of rules
- B , maximum number of false positives for each rule on labeled examples

They show that for three data sets commonly used in the literature we can achieve high precision and recall by setting $d = 4, K = 4$. We implemented their approach and achieved comparable results on these data sets.

Random Forest Random forest is a popular machine learning model for classification. Random forests are an ensemble learning method for classification in which multiple decision trees are learned over the data. Classification decision is made by majority voting over the trees. An advantage of random forest over trees is that it avoids over-fitting to examples through the majority voting procedure. Even though it loses some interpretability compared to single decision trees, it is still convertible to a set of rules that are human-readable. We are specifically interested in random forests because we would like to compare the complexity of learned trees between the original and cleaned data sets. Gokhale et al. [22] used random forest to learn a matching classifier and show that it can achieve high quality matching results.

4.4.2.1 Cleaning options

Table 4.5 shows the effect of proposed cleanings on the quality of the rule-based and machine learning matchers. We report quality with three different options:

1. **None.** No cleaning applied to training or test data.
2. **Training and test.** Cleaning operations applied to both training and test data.

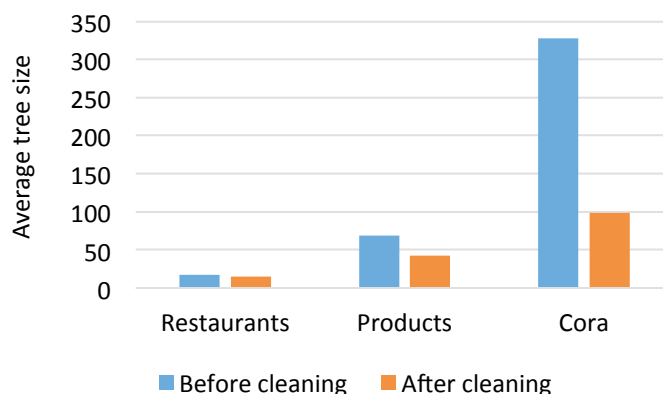


Figure 4.7: Average size of random forest trees before and after cleaning.

3. **Training but not test.** Cleaning operations only applied to training data.

When comparing the quality with no cleaning (option 1), and when both training and test data are cleaning (option 2), we can see that quality of the classifiers increase with cleaning the data sets. However, if we only normalize the training data and not the test data (option 3), it can have a detrimental effect on quality, such that quality is decreased compared to not doing any cleaning. This test is not applicable to the products data set since the reference cleaning operations do not have any normalizations. This shows that it is important that the analyst performs “systematic” cleanings, those that will be able to fix errors in both train and test data.

4.4.2.2 Classifier complexity

We find that simpler matchers are easier to understand/debug for the analysts. We observed that cleaning a data set can simplify the matcher as we will show here. For the purpose of our experiments we estimate the complexity of the model with average tree size of the trained random forest (i.e. number of nodes in the tree). In Figure 4.7, we report the average size of the random forest trees before and after applying the cleaning operations to both training and test data. We can see that cleaning can lead to significantly simpler random forest models, which is most obvious for the case of the Cora data set that required the most cleanings.

Dataset	Cleaning Functions
Restaurant	Normalize type Add address latlong feature Add name_contains feature Fix ground truth error (1)
Products	Add modelno_in_title feature Add modelno_contains_modelno feature Fix ground truth errors (4)
Cora	Extract and normalize year Normalize author Normalize venue Extract and normalize pages Fix ground truth errors (260)

Figure 4.8: Cleaning operations identified by inspecting the record pairs using the Mono ranking.

Data set	Classifier	F1 before	F1 after	Increase in F1	F1 after (only train)	Decrease in F1 (only train)
Restaurants	Rule-based	94	96	2	94	0
	Random Forest	94	96	2	91	-3
Products	Rule-based	76	80	4	NA	NA
	Random Forest	74	78	4	NA	NA
Cora	Rule-based	86	97	11	94	8
	Random Forest	91	98	7	81	-10

Table 4.5: Quality of classifiers before/after cleanings proposed by the analyst.

4.4.3 Spatial blocking

The goal of spatial blocking is to reduce the runtime of the Mono approach for larger samples. Basically, we want to find all monotonicity violations efficiently. The exhaustive approach will compare all pairs with a positive label with all pairs with a negative label. With spatial blocking we eliminate comparing matching and non-matching pairs that are obviously monotonic.

For this experiment we use the full candidate sets for Cora, Restaurants, and Products. We have access to the labels for all the candidate pairs for these data sets. We set `slength`, the length of each side of the squares in spatial blocking, to 0.25 for all the data sets. Number of positive and negative record pairs in each table and the cross product is reported in Figure 4.6. In Figure 4.9, we can see that with all three

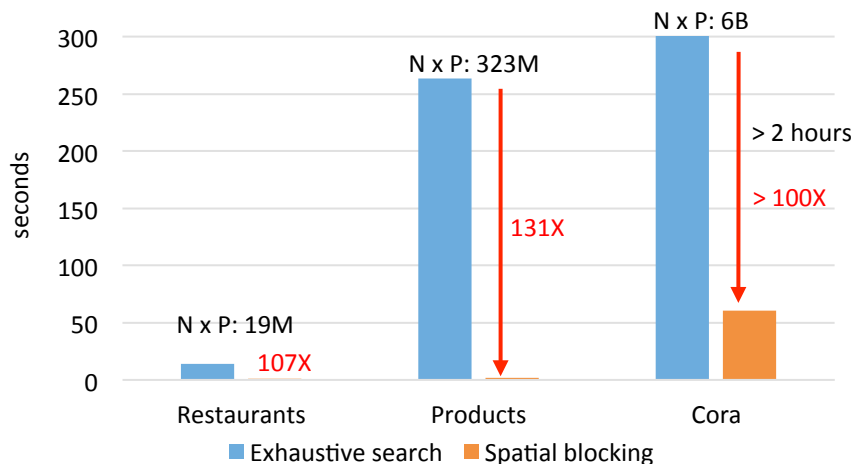


Figure 4.9: Time for finding all monotonicity violations with exhaustive search versus spatial blocking.

Data set	Source A	Source B	Table A size	Table B size	Candidate set size	Cross product (N, P) N × P
Restaurants	Fodors	Zagats	528	329	173712	(111, 173601) 19269711
Products	Walmart	Amazon	2554	22074	291649	(1112, 290537) 323077144
Cora	Cora search engine	Cora search engine	1295	1295	379748	(17184, 362564) 6230299776

Table 4.6: Data sets used in the experiments for evaluating cleaning impact and spatial blocking.

data sets there is orders of magnitude improvement in run time. This improvement is specially noticeable for the analyst with the Cora data set. For this data set the exhaustive search would take hours to complete, however, with spatial blocking, this time is reduced to about 60 seconds.

4.5 Discussions

Continuing from our abstract analysts in Chapter 2 and throughout this chapter we observe that resolving inconsistencies in a data set can lead to simpler and

more accurate matchers and reduce analyst effort for matching. We saw that there can be many causes for inconsistency in a data set such as normalization issues, incorrect labels, attribute extraction error and so forth. Even though we evaluated our framework for finding and resolving inconsistencies in a data set for the case of incorrect labels, we anticipate this framework to reveal other kinds of inconsistencies in a data set. In fact, when inspecting the exact same rankings that we inspected for finding incorrect labels, we can identify other causes of inconsistency. This is how we found and proposed cleaning operations for Cora, Restaurants, and Products data sets in the previous section.

Even though we empirically observed that the rankings generated by Mono, PPFN, and Hybrid are helpful in finding a variety of inconsistency causes, it remains an open challenge how to precisely show this in an experimental setting. This is due to the fact that every pair in the ranking can point to multiple causes of inconsistency and so it is not clear what action the analyst will take when viewing that pair. For the case of the incorrect label, we assumed that the analyst will fix the incorrect label, which intuitively makes sense. Even if we know exactly how the analyst reacts when viewing a pair, it is not clear how helpful that action is in terms of cleaning the data set.

One way to mitigate this issue is to come up with an objective metric for cleanliness of a data set that can be measured after analysts' cleaning actions are performed. One can think of multiple metrics that can be indicative of cleanliness. For example, we hypothesise that the number of monotonicity violations can be correlated with cleanliness. If there are no monotonicity violations, typically matching can be done with a set of simple rules. However, even without monotonicity violations, there could be cases where the analyst would need to write many rules for matching.

Similarly, complexity of the matcher can be correlated with cleanliness. The cleaner the data set, the simpler the matcher can be. We showed in our experiments that as we clean the data set, our random forest model got much simpler. Furthermore, the quality of matching can be correlated with cleanliness. The cleaner the data set, the higher we can get with respect to precision and recall. We showed this with abstract analysts in Chapter 2 to some extent. When the data set included inconsistencies, it was hard for the analyst to increase recall by keeping the precision high. However, in some cases, you can achieve high quality matching even though

your data set includes many inconsistencies by having a very complex matching model.

As has become apparent, even though these metrics can be correlated with cleaning, they are not absolute metrics for data set cleanliness. In fact, there may not exist an absolute metric for data cleanliness since the notion of cleanliness can be vague itself. One could argue the the ultimate goal is to have high quality matching, and cleaning the data set is only relevant when it can improve matching. Consider the case of fixing incorrect labels in the labeled sample. It seems very intuitive that if the analyst finds an incorrect label, she should fix it. However, in our experiments we found cases where fixing the incorrect label actually reduced the quality of the machine learning model. In such a case should the analyst fix this label or not?

Finding a metric for data set cleanliness remains an open challenge and an interesting area for future work. In the meantime, we believe that it is important to resolve inconsistencies in a data set regardless of quality of matching. First, it is counter-intuitive that cleaning a data set will reduce quality of matching. In that case, perhaps the matcher is very complex and is over-fitting to the data. This kind of matcher will be very hard for the analyst to debug and understand. And secondly, typically the same data has multiple use cases in an organization that could be different from matching. In that case, cleaning a data set can improve quality of other usecases of the same data and thus desired.

4.6 Related Work

The notion of monotonicity for entity matching data sets was introduced in [6]. They observed that for most entity matching data sets matching pairs are more similar than non-matching pairs in at least one dimension. They then propose a heuristic algorithm that uses this observation, searches the feature space and finds a set of simple rules for matching. We also have observed that “clean” entity matching data sets satisfy monotonicity. However, data sets that are not properly cleaned or enough features are not defined for capturing their similarity often violate monotonicity. We use this observation to find record pairs that we suspect require cleaning and ranking them.

Our work is related to the literature regarding data repair [4, 10, 47]. In these works, given database D , we have a set of Data Quality Rules (DQRs) that need to be

satisfied, and we would like to resolve the constraints with minimal change to the database. For example, one DQR can state that if zipcode is 53706 then city must be Madison and state must be WI. Cong et al. [10] propose a heuristic approach to automatically repair the database to satisfy the DQRs while keeping the accuracy of the repairs high. Yakout et al. [47] investigate a solution where an analyst should inspect the automatically suggested repairs for correctness. They propose a ranking mechanism to show repairs to the analyst that will improve the quality most. We use similar concepts in the entity matching context, where our DQR is that there is no monotonicity violation between matching and non-matching record pairs. We then rank the record pairs such that the ones we suspect will be most helpful to the analyst for resolving the violations come first.

In order to find monotonicity violations quickly, we use a *spatial blocking* method to avoid comparing record pairs that are obviously monotonic. To do so we map each feature vector to a point in the feature space. This is similar to the work done for clustering high dimensional and spatial data in [1, 40]. In these works, the feature space is divided into rectangular cells and points are assigned to each cell based on their feature vector. This information is then used to find high density cells adjacent to each other and form clusters. We leverage the spatial structure to quickly find record pairs that are obviously monotonic or non-monotonic with respect to a particular record pair.

We can regard the task of finding violations of monotonicity as a non-equijoin over two data bases composed of only matching and only non-matching pairs and their feature vectors. A non-equi (or theta) join is a join statement that uses an unequal operation such as \leq , \geq [13]. Suppose we have two features f_1, f_2 , to find all non-matching pairs that are more similar than matching pairs in all dimensions we will write the join:

```
SELECT M.pair,N.pair FROM MATCHING AS M, NONMATCHING AS N WHERE N.f1
> M.f1 AND N.f2 > M.f2
```

Frenay et al. [21] survey the state-of-the-art research on finding incorrect labels in training data for machine learning. They enumerate reasons for such mislabelled data points such as insufficient information for labeling, data of poor quality, mistake of labelers, and so on. They also discuss various literature that have shown that labeling errors can decrease prediction performance as well as increase the

complexity of the learned model. Other related tasks such as feature selection or feature ranking may be impacted by label noise as well.

They also categorize state of the art methods to deal with label noise. The first approach is to use learning methods that are robust to label noise. For example, bagging achieves better results than boosting in presence of label errors. Data cleansing methods remove data points that appear to be mislabelled, for example, using model predictions to detect suspicious data points and removing them. Another approach is to make learning algorithms noise-tolerant. For example, prevent data points to take too large weights in neural networks.

They further discuss the issue that there are only a few data sets where mislabelled instances have been identified. This is similar to our experiment setting where we had to manually verify the correctness of the labels in our seven data sets to find the incorrect labels. This makes it hard to verify effectiveness of the approaches to deal with noise in labels. Thus, most of the experiments are done with injecting synthetic errors. We also had to inject synthetic errors to six of our data sets where there were only a few errors as well.

Our goal in this chapter is broader than just finding incorrect labels in the labeled sample. We propose a framework for finding and resolving inconsistencies in a matching data set, and there could be many causes for such inconsistencies. Furthermore, in matching we may use manual rule-based matchers or matchers that are learned from the labeled sample. The research on dealing with noise in labels for machine learning models can help us learn higher quality machine learning matchers if incorrect labels exist in the labeled sample, but does not eliminate the need for fixing such errors in the labeled sample.

4.7 Conclusions

In this chapter we categorize the causes of inconsistency in a matching data set and propose a framework for finding and resolving such inconsistencies for the analyst. Our goal is to reduce analyst effort in this process. Therefore, we attempt to rank pairs of records such that the ones that are more problematic come on top so that the analyst can easily find them. We observe that there could be multiple methods for ranking the pairs of records and each could find a different set of problematic pairs. Therefore, we propose to use state-of-the-art rank aggregation methods and present

a single aggregated ranking to the analyst. We propose two different rankings, FPFN and Mono, and aggregate them to get a third ranking, Hybrid. The analyst then interactively inspects the pairs in the ranking and proposes cleaning operations on the data set. We experiment with seven data sets and show that the rankings can in fact find different problematic pairs and using the Hybrid approach the analyst is able to pinpoint more errors. Furthermore, we propose cleaning operations for three data sets used in the entity matching literature and show that cleaning the data sets will improve matching quality and reduce complexity of matchers.

5 CONCLUSIONS, LIMITATIONS, AND FUTURE WORK

5.1 Conclusions

In this thesis we worked on the problem of “Human-Centric Debugging of Entity Matching”. By “Human-Centric” we acknowledged the fact that typically there is a human analyst in the loop for performing entity matching. We then turned our attention to “Debugging of Entity Matching” when it is done by a human analyst, where debugging is the process by which an analyst iteratively improves quality of matching.

To better understand the problem space we developed an end-to-end matching system and experimented with it in an e-commerce setting as well as with students from a graduate data modeling course at UW-Madison. We also developed and experimented with an abstract model of the entity matching problem for the analyst. These experiences helped us identify two key challenges in debugging of entity matching for analysts (Chapter 2). First, we found that as an analyst is iteratively defining a set of rules for matching, she spends unproductive time waiting for matching results to come back after each change to the matcher. And second, we found that some data sets are harder for analysts to match due to inconsistencies in them.

In Chapter 3, we address the first challenge by developing algorithms to order rules as well as performing matching incrementally to reduce runtime of rule-based matchers. We show that we can reduce matching runtime significantly by using these methods. In Chapter 4, we address the second challenge by developing a framework for helping analysts to find and resolve inconsistencies in a data set. We show that the analyst is able to find inconsistencies in a data set using this framework, and that removing these inconsistencies lead to simpler and higher quality matchers.

5.2 Limitations and Future work

Throughout this thesis, we have used a set of simple measures, such as time to perform matching and number of browsed pairs, to suggest that we can reduce the time and effort that an analyst spends in debugging for completing a matching task.

These measures are intuitive and we found them important through interacting with analysts in an e-commerce setting and students that used our end-to-end entity matching system on data sets with various domains. Intuitively, if the analyst can iterate faster through the matching process, this is likely to save the analyst time. Similarly, if we rank the pairs of records so that the problematic ones are ranked higher, the analyst needs to browse through fewer number of pairs to find inconsistencies, and this saves analyst time and effort. However, there may be more variables that come into play in practice when the analyst is performing matching, and we have not addressed them in this thesis.

Fully understanding how an analyst interacts with a matching problem, and what parts of the debugging process are more challenging to the analyst, would require performing formal field studies. Field studies are conducted in the actual location and context where the work is getting done. For example, one can observe real analysts whose job is performing entity matching to understand their work process and find points in the process that requires the most time and/or effort from the analyst. Even though we tried to develop intuition by interacting with analysts in an e-commerce setting and students that used our end-to-end system, we did not perform formal studies.

Furthermore, validating that our proposed approaches actually help analysts would require conducting controlled laboratory experiments using human participants, preferably real analysts. Such experiments would require measuring metrics such as time and number of browsed pairs for analysts when performing matching, with and without employing our techniques, and with data sets from different domains. We would then be able to examine if we observe a statistically significant difference in the metrics under these conditions.

Conducting field studies to better understand analyst challenges in debugging of entity matching as well as conducting controlled laboratory experiments to confirm effectiveness of our approaches are interesting areas for future work.

REFERENCES

- [1] Agrawal, Rakesh, Johannes Gehrke, Dimitrios Gunopulos, and Prabhakar Raghavan. 2005. Automatic subspace clustering of high dimensional data. *Data Mining and Knowledge Discovery* 11(1):5–33.
- [2] Babu, Shivnath, Rajeev Motwani, Kamesh Munagala, Itaru Nishizawa, and Jennifer Widom. 2004. Adaptive ordering of pipelined stream filters. In *SIGMOD*, 407–418.
- [3] Benjelloun, Omar, Hector Garcia-Molina, David Menestrina, Qi Su, Steven Euijong Whang, and Jennifer Widom. 2009. Swoosh: a generic approach to entity resolution. *The VLDB Journal* 18(1):255–276.
- [4] Bohannon, Philip, Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. 2007. Conditional functional dependencies for data cleaning. In *ICDE*, 746–755.
- [5] Brancotte, Bryan, Bo Yang, Guillaume Blin, Sarah Cohen-Boulakia, Alain Denise, and Sylvie Hamel. 2015. Rank aggregation with ties: Experiments and analysis. In *VLDB*, 1202–1213.
- [6] Chaudhuri, Surajit, Bee Chung Chen, Venkatesh Ganti, and Raghav Kaushik. 2007. Example driven design of efficient record matching queries. In *VLDB*, 327–338.
- [7] Chiticariu, Laura, Yunyao Li, and Frederick R Reiss. 2013. Rule-based information extraction is dead! long live rule-based information extraction systems! In *EMNLP*, 827–832.
- [8] Chomicki, Jan, Parke Godfrey, Jarek Gryz, and Dongming Liang. 2003. Skyline with presorting. In *ICDE*, 717–719.
- [9] Christen, Peter. 2012. *Data matching concepts and techniques for record linkage, entity resolution, and duplicate detection*. Springer.
- [10] Cong, Gao, Wenfei Fan, Floris Geerts, Xibei Jia, and Shuai Ma. 2007. Improving data quality: Consistency and accuracy. In *VLDB*, 315–326.

- [11] Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2001. *Introduction to algorithms, second edition*. The MIT Press and McGraw-Hill Book Company.
- [12] Demartini, Gianluca, Djellel Eddine Difallah, and Philippe Cudré-Mauroux. 2012. Zencrowd: leveraging probabilistic reasoning and crowdsourcing techniques for large-scale entity linking. In *WWW*, 469–478.
- [13] DeWitt, David J, Jeffrey F Naughton, and Donovan A Schneider. 1991. An evaluation of non-equijoin algorithms. In *VLDB*, 443–452.
- [14] Doan, AnHai, Alon Halevy, and Zachary Ives. 2012. *Principles of data integration*. Elsevier.
- [15] Dwork, Cynthia, Ravi Kumar, Moni Naor, and Dandapani Sivakumar. 2001. Rank aggregation methods for the web. In *WWW*, 613–622.
- [16] Elmagarmid, Ahmed K, Panagiotis G Ipeirotis, and Vassilios S Verykios. 2007. Duplicate record detection: A survey. *IEEE Transactions on Knowledge and Data Engineering* 19(1):1–16.
- [17] Fagin, Ronald, Ravi Kumar, Mohammad Mahdian, D Sivakumar, and Erik Vee. 2004. Comparing and aggregating rankings with ties. In *SIGMOD*, 47–58.
- [18] ———. 2006. Comparing partial rankings. *SIAM Journal on Discrete Mathematics* 20(3):628–648.
- [19] Fagin, Ronald, Ravi Kumar, and D Sivakumar. 2003. Comparing top k lists. *SIAM Journal on Discrete Mathematics* 17(1):134–160.
- [20] Feige, Uriel, László Lovász, and Prasad Tetali. 2002. Approximating min-sum set cover. In *APPROX*, 94–107.
- [21] Frénay, Benoît, Ata Kabán, et al. 2014. A comprehensive introduction to label noise. In *ESANN*, 667–676.
- [22] Gokhale, Chaitanya, Sanjib Das, AnHai Doan, Jeffrey F Naughton, Narasimhan Rampalli, Jude Shavlik, and Xiaojin Zhu. 2014. Corleone: Hands-off crowdsourcing for entity matching. In *SIGMOD*, 601–612.

- [23] Gu, Lifang, Rohan Baxter, Deanne Vickers, and Chris Rainsford. 2003. Record linkage: Current practice and future directions. Tech. Rep., CSIRO Mathematical and Information Sciences.
- [24] Hanrahan, Pat. 2012. Analytic database technologies for a new kind of user: the data enthusiast. In *SIGMOD*, 577–578.
- [25] Hellerstein, Joseph M. 1998. Optimization techniques for queries with expensive methods. *ACM TODS* 23(2):113–157.
- [26] Herzog, Thomas N, Fritz J Scheuren, and William E Winkler. 2007. *Data quality and record linkage techniques*. Springer.
- [27] Kolb, Lars, Andreas Thor, and Erhard Rahm. 2012. Dedoop: efficient deduplication with hadoop. In *VLDB*, 1878–1881.
- [28] Konda, Pradap, Sanjib Das, Paul Suganthan GC, AnHai Doan, Adel Ardalan, Jeffrey R Ballard, Han Li, Fatemah Panahi, Haojun Zhang, Jeff Naughton, et al. 2016. Magellan: Toward building entity matching management systems. In *VLDB*, 1197–1208.
- [29] Köpcke, Hanna, and Erhard Rahm. 2010. Frameworks for entity matching: A comparison. *Data & Knowledge Engineering* 69(2):197–210.
- [30] Munagala, Kamesh, Shivnath Babu, Rajeev Motwani, and Jennifer Widom. 2005. The pipelined set cover problem. In *ICDT*, 83–98.
- [31] Nielsen, Jakob. 1994. *Usability engineering*. Elsevier.
- [32] Paul Suganthan, GC, Chong Sun, K Krishna Gayatri, Haojun Zhang, Frank Yang, Narasimhan Rampalli, Shishir Prasad, Esteban Arcaute, Ganesh Krishnan, Rohit Deep, et al. 2015. Why big data industrial systems need rules and what we can do about it. In *SIGMOD*, 265–276.
- [33] Pradap Konda, Paul Suganthan G.C. AnHai Doan Adel Ardalan Jeffrey R. Ballard Han Li Fatemah Panahi Haojun Zhang Jeff Naughton Shishir Prasad Ganesh Krishnan Rohit Deep Vijay Raghavendra, Sanjib Das. 2016. Toward building entity matching management systems. Tech. Rep., UW-Madison.

- [34] Schalekamp, Frans, and Anke van Zuylen. 2009. Rank aggregation: Together we're strong. In *Proceedings of the meeting on algorithm engineering & experiments*, 38–51.
- [35] Stoyanovich, Julia, Sihem Amer-Yahia, Susan B Davidson, Marie Jacob, Tova Milo, et al. 2013. Understanding local structure in ranked datasets. In *CIDR*.
- [36] Sun, Chong, and Jeffrey Naughton. 2012. Multi-filter string matching and human-centric entity matching for information extraction.
- [37] Wang, Jiannan, Tim Kraska, Michael J Franklin, and Jianhua Feng. 2012. Crowder: Crowdsourcing entity resolution. In *VLDB*, 1483–1494.
- [38] Wang, Jiannan, Guoliang Li, Tim Kraska, Michael J Franklin, and Jianhua Feng. 2013. Leveraging transitive relations for crowdsourced joins. In *SIGMOD*, 229–240.
- [39] Wang, Jiannan, Guoliang Li, Jeffrey Xu Yu, and Jianhua Feng. 2011. Entity matching: How similar is similar. In *VLDB*, 622–633.
- [40] Wang, Wei, Jiong Yang, Richard Muntz, et al. 1997. Sting: A statistical information grid approach to spatial data mining. In *VLDB*, 186–195.
- [41] Whang, Steven Euijong, Omar Benjelloun, and Hector Garcia-Molina. 2009. Generic entity resolution with negative rules. *The VLDB Journal* 18(6):1261–1277.
- [42] Whang, Steven Euijong, and Hector Garcia-Molina. 2010. Entity resolution with evolving rules. In *VLDB*, 1326–1337.
- [43] ———. 2014. Incremental entity resolution on rules and data. *The VLDB Journal* 23(1):77–102.
- [44] Whang, Steven Euijong, Peter Lofgren, and Hector Garcia-Molina. 2013. Question selection for crowd entity resolution. In *VLDB*, 349–360.
- [45] Whang, Steven Euijong, Julian McAuley, and Hector Garcia-Molina. 2012. Compare me maybe: Crowd entity resolution interfaces. Tech. Rep., Stanford InfoLab.

- [46] Winkler, William E. 2006. Overview of record linkage and current research directions. Tech. Rep., Bureau of the Census.
- [47] Yakout, Mohamed, Ahmed K Elmagarmid, Jennifer Neville, Mourad Ouzzani, and Ihab F Ilyas. 2011. Guided data repair. In *VLDB*, 279–289.