Effective Instance Matching for Heterogeneous Structured Data

Zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften (Dr.-Ing.)

von der Fakultät für Wirtschaftswissenschaften
des Karlsruher Instituts für Technologie (KIT)
genehmigte Dissertation von

Yongtao Ma

Tag der mündlichen Prüfung: 05. Juni 2014
Referent: Prof. Dr. Rudi Studer, Karlsruhe Institute of Technology
Korreferent: Prof. Dr. Christian Bizer, Universität Mannheim

Abstract

Structured data is abundantly available in enterprises and also largely increasing in the Web setting. Generally speaking, it can be conceived as structured descriptions of real-world entities. One main problem towards the effective usage of structured data is instance matching, where the goal is to find instance representations referring to the same real-world thing. However, the structured data on the Web is heterogeneous, e.g. type information of instances is missing or too general to be useful. Besides, the challenges that lie ahead for typical instance matching approaches also include dealing with the low-quality data and high computation complexity.

In this thesis, we tackle these challenges in different steps of the instance matching process. The first step is *typification*, in which the type semantics is derived by an unsupervised approach. The second step, *blocking*, aims to reduce the quadratic complexity of the instance matching process through the efficient and effective generation of match candidates. We propose an unsupervised approach to learn the most representative attributes of instances called keys, based on which two instances are considered as a match candidate if they share the same value of the key. The third step *classification*, aims to deal with the low quality of data, for which we propose an almost-parameter-free approach for learning instance matching rules to classify candidate instance pairs into matches and non-matches. In the last *filtering* step, we propose a parameter-free solution that leverage only simple Boolean functions and exploits fine-grained word-level dissimilarity evidences to further filter out the non-matches. We evaluate our approaches against the latest baselines. The results show advances beyond the state-of-the-art.

Acknowledgements

Without the support and advices from so many people, this thesis would not have been possible. First, I would like to thank my supervisor Prof. Dr. Rudi Studer for providing me the opportunity to work at institute AIFB and in particular the support and guidance to complete this dissertation. I thank Dr. Duc Thanh Tran, my co-advisor, for his advices, motivation, enthusiasm, and patience during the work on this thesis. I also thank Prof. Dr. Christian Bizer for taking the Korreferat of my dissertation and for providing me numerous important advices to improve it.

In particular, I would like to gratefully acknowledge the funding that I received towards my PhD from Siemens and Germany Academic Exchange Service (DAAD) PhD fellowship. I also thank Dr. Steffen Lamparter, my supervisor in Siemens, for his supervisory role and encouragement.

Furthermore, I would like to thank my colleagues at AIFB, who are professional, conscientious and warm-hearted, taken together, make it a great team. Especially, I am thankful to Dr. Andreas Wager and Dr. Günter Ladwig for suggestions and technical assistance, to Dr. Daniel M. Herzig for usually sharing delicious food, and to Lei Zhang for many discussions and Brainstorming.

Also, I would like to thank my friends and family for their support and encouragement. Especially, I thank my parents, Jiazhong Ma and Wenzhen Deng, for their unconditional love and care. Without them I cannot be where I am today.

Most of all, I want to thank my beloved Talatuoni for her tremendous encouragement, support and patience during the days spent for this dissertation. I cannot forget the many late evenings that she accompanied me in front of the computer.

Contents

1 Introduction 1
1.1 Structured Data . . . 1
1.2 Instance Matching . . . 2
1.3 Challenges . . . 2
1.4 Research Question . . . 3
1.5 Contribution of this Thesis . . . 5
1.6 Organization of this Thesis . . . 7

2 Foundations 9
2.1 Data Model . . . 9
2.2 Attribute Matching . . . 12
2.3 Preprocessing . . . 14
2.3.1 Typical Preprocessing Tasks . . . 14
2.3.2 Typification . . . 17
2.4 Candidate Generation . . . 18
2.5 Classification . . . 20
2.5.1 Supervised Learning of Instance Matching Schema . . . 21
2.5.2 Efficient Execution of Instance Matching Schema . . . 22
2.5.3 Attribute-threshold Instance Matching Rule . . . 24
2.5.4 Mapping-threshold Instance Matching Rule . . . 24
2.5.5 Collective Instance Matching . . . 25
2.6 Filtering . . . 26
2.7 Metrics . . . 26

3 Typification: Inferring the Type Semantics of Structured Data 29
3.1 Introduction . . . 29
3.2 Research Question and Contributions . . . 30
3.3 Overview . . . 31
3.4 Clustering Solutions . . . 32
3.4.1 Features and Similarities . . . 33
3.4.2 Techniques . . . 33
3.5 Value-level schema Features . . . 35
3.6 TYPifier . . . 38
3.6.1 Clusters and Cluster Relations . . . 38
3.6.2 Relation-based Hierarchical Clustering . . . 40
3.7 Experimental Evaluation . . . 44
3.7.1 Datasets . . . 45
3.7.2 Efficiency of Typification . . . 46

3.7.3 Effectiveness of Typification 47
3.7.4 Parameter Sensitivity 48
3.8 Related Work 51
3.9 Conclusion 53

4 Blocking: Learning Type-specific Blocking Key and Key Value 55
4.1 Introduction 55
4.2 Research Question and Contributions 56
4.3 Overview 57
4.4 Learning Types 59
4.5 Learning Keys and Values 60
4.5.1 Blocking Key Selection 60
4.5.2 Key Value Selection 62
4.6 Experimental Evaluation 64
4.6.1 Datasets and Matching Tasks 64
4.6.2 Experimental Setting 65
4.6.3 Efficiency of Blocking 66
4.6.4 Effectiveness of Blocking 67
4.6.5 Efficiency of Instance Matching 68
4.6.6 Effectiveness of Instance Matching 70
4.7 Related Work 71
4.8 Conclusion 73

5 Classification: Learning Rules for Effective Almost-parameter-free Instance Matching 75
5.1 Introduction 75
5.2 Research Question and Contribution 77
5.3 Instance Matching 78
5.4 Algorithm for Learning mIR 81
5.4.1 Certainty for Instance Matching 81
5.4.2 Estimate (Non-)Matching Certainty 83
5.4.3 Learn Threshold 85
5.4.4 Evaluate mIR Candidate 87
5.4.5 Learn Single mIR 89
5.4.6 Learn a Set of mIRs 90
5.5 Algorithm for Executing mIR 91
5.6 Experimental Evaluation 93
5.6.1 Dataset and Matching Task 93
5.6.2 Experimental Setting 94
5.6.3 Efficiency 94
5.6.4 Effectiveness 95
5.6.5 Parameter Sensitiveness 97
5.7 Related Work 98
5.8 Conclusion 99

6 Filtering: Effective Parameter-free Boolean Instance Matching 101
6.1 Introduction 101
6.2 Research Question and Contribution 102
6.3 Overview 103
6.3.1 Thresholded Instance Matching 104
6.3.2 Boolean Instance Matching. 105
6.4 Learning Word-Level Dissimilarity Evidences 106
6.4.1 Word Co-occurrence Based Evidences 106
6.4.2 Learning CDE from Positive Examples 107
6.4.3 Using Self-Matches as Examples 110
6.4.4 Enriching Examples with Self-learning 112
6.4.5 On the Combination of Thresholded and Boolean Matching . . 114
6.4.6 Multiple Attributes 115
6.5 Implementation 115
6.6 Experimental Evaluation 119
6.6.1 Parameter Analysis 120
6.6.2 Efficiency of Instance Matching 122
6.6.3 Effectiveness of Instance Matching 123
6.6.4 Labeling Effort 124
6.7 Related Work 126
6.8 Conclusion 127

7 Conclusion 129
7.1 Summary 129
7.2 Outlook 130
7.3 Conclusion 131

Bibliography 133

List of Figures 145

List of Tables 147

A Algorithm Analysis 149
A.1 Number of aIR candidates 149
A.2 Number of mIR candidates 149
A.3 Time Complexity Analysis 150
A.4 Set-Based Analysis 150

Chapter 1

Introduction

In this first chapter, this thesis motivates the topic of instance matching on heterogeneous structured data. Section 1.1 and 1.2 briefly introduce the structured data and instance matching problem, respectively. Section 1.3 motivates the main topic of the thesis with the challenges including heterogeneity, complexity and the low quality of the data. Resulting from research questions in Section 1.4, we present the main contributions of this thesis in Section 1.5. Finally, Section 1.6 presents the organization of this thesis.

1.1 Structured Data

Structured data is abundantly available in enterprises and also largely increasing in the Web setting. Generally speaking, it can be conceived as structured descriptions of real-world entities. A popular standard for making such descriptions available on the Web is RDF [65, 80]. It is a graph-structured data model that can be flexibly used to compose entity descriptions as sets of $\langle subject, predicate, object \rangle$ triples, or in other words, edges of an RDF data graph. Every triple captures either (1) an entity's *attribute* value, (2) its *type*, (3) or a *relation* between an entity and another. In the last few years, large number of structured data (formerly stored in relational and XML databases) have been converted to RDF and OWL [53], and published on the Web. Prominent examples include DBpedia[1] (the structured data counterpart of Wikipedia), as well as entity descriptions embedded in Web pages in the form of microformats[2] and RDFa[3]. The semantics captured by this data have been exploited in various tasks such as Web search: RDFa and mircoformats are used by Google and Yahoo! to provide rich snippets [4] for Web search results, which are based on structured descriptions of the entities embedded in the results.

[1] http://dbpedia.org
[2] http://microformats.org/
[3] http://www.w3.org/TR/xhtml-rdfa-primer/
[4] http://www.google.com/webmasters/tools/richsnippets

1.2 Instance Matching

Given the increasing amount of structured data, one main problem towards the effective usage of them is integration. Besides schema matching, the other main problem in this regard is *instance matching* (also known as merge/purge [54], deduplication [112], reference reconciliation [39], or record linkage [89]), which is to resolve differences in the data representation for finding the same real-world object (matches) [23, 33, 39, 42, 45, 54, 62, 82].

Instance matching is an important part for a variety of applications. Most commonly, the data to be matched across two or more datasets, or to be deduplicated in a single data source (named self-match) may correspond to different domains, such as people, businesses, productions, and publications. For example, with the increasing number of online shopping sites, instance matching has become the essential technology to allow accurate and comprehensive price comparisons for a certain product. A comparison shopping site should be able to accurately identify the products that are actually the same. For applications such as Web search, it is important that the duplicate results being removed from the search results. Besides, instance matching technologies are also widely applied to applications such as national census, national security, bibliographic databases to enable the data being prepared for statistical analysis and data mining to discover new and potentially knowledge.

1.3 Challenges

Instance matching is a difficult task for several reasons. We highlight some of the major challenges that will be further discussed in the relevant chapters later in this thesis.

Challenge 1: Heterogeneity. Instance matching solutions may not perform well, when *type information is either missing or too general to be useful*. The proliferation of RDF data on the Web is mainly due to the flexibility of the RDF model. Without imposing constraints on the schema and data integrity, it makes it easy for data providers to assert, to publish, and to link triples to other datasets. While it seems to be well suited for dealing with the dynamics on the Web, the downside is that many structured descriptions available on the Web today do not contain type triples. Besides, in enterprises, entity types mostly correspond to tables in which tuples representing entity descriptions are stored. However, big data tables are often used to store a variety of entity descriptions. The type captured by a table is often too general to be useful.

Challenge 2: Complexity. Intuitively, when matching instances between two data sources, each instance in one data source need to be compared with all the instances in the other data source to determine if a pair of instances refers to the same entity.

The cost of instance matching is therefore very expensive because the computation complexity grow quadratically as the data sources to be matched get larger.

Challenge 3: Low data quality. The quality of the structured data exhibited on the Web is generally low. On the one hand, the data sources to be matched lack of unique identifiers, e.g. ISBN of publications or social security numbers of people. Otherwise, if the identifiers are available, the instance matching task can be simply implemented by comparing the values of the identifiers. On the other hand, even if identifiers are contained in the data sources, it is unlikely that the contents of the identifiers are consistent and accurate, in particular on the Web where datasets may change frequently and contain various errors such as spelling mistakes and missing values.

1.4 Research Question

The principal research question of this thesis is:

How to match instances effectively and efficiently for heterogeneous structured data?

This broad research question entails the three main challenges as stated above. Towards tackling these challenges, there are four corresponding research questions related to different steps of the instance matching process:

Research Question 1. *How to derive type semantics of instances?*

In the pre-processing step, the question is how can we derive type semantics of instances. The existing instance matching approaches are designed to treat instances as belonging to one single type, i.e. instances of the same or similar types. When the two data sources to be matched contain instances of different types, the state-of-the-art approaches solve the problem by cross-fertilizing instance matching results with schema matching results [43, 102], such that only instances of the same type are matched. However, these solutions do not perform well, when type information is missing on the schema level or too general to be useful, as stated in Challenge 1. Therefore, to restore type information is important for instance matching problem. In Chapter 3, we formulate this problem of inferring type semantics as a clustering problem, where the goal is to identify a particular kind of clusters that represents the type of entities. We also discuss the features that are used for structured and semi-structured data.

Research Question 2. *How can match candidates be efficiently and effectively generated?*

In the second step, the question is how can match candidates be generated to reduce the quadratic complexity of the instance matching process, as stated in Challenge 2. *Blocking* is a typical approach to boost efficiency, which is designed to efficiently filter out non-matches and generate instance pairs that are most likely to be matches. The blocking approaches split the original data sources into small blocks according to some criteria called *blocking key*, e.g. the `title` of two `products` data sources. Only the instances that are in the same block, i.e. the instances that have the same value of the blocking key, will be further compared in detail. In Chapter 4, we investigate how to learn the type-specific blocking keys and key values that enable the efficient and effective generation of match candidates.

Research Question 3. *How can the match candidates be effectively classified to matches and non-matches?*

In the classification step, we need to answer the question that how can the match candidates be effectively classified to matches and non-matches. Due to the low quality of data, as stated in Challenge 3, the calculated match candidates may include a large number of incorrect results, i.e. false positives. Existing approaches employ instance-matching rules to find the real matches. For example, "two products are considered as the same if they have similar `Title` and `Manufacturer`". However, the question is actually to determine how to calculate similarities and how similar is similar. To address the question, different combinations of attributes and similarity functions are considered to calculate similarities. Thresholds are also incorporated into the rules so that instances are considered similar if their similarity is higher than a threshold. For example, the rule above can be specified as "two `products` are the same, if the `Jaccard` similarity of `Title` is greater than 0.8 and the `Cosine` similarity of `Manufacturer` is greater than 0.4". In Chapter 5 we discuss how to learn instance-matching rules for effective instance matching.

Research Question 4. *How to identify non-matching instance pairs by simple Boolean functions?*

In the last step, filtering, we will answer the question whether we can further filter out the non-matches by simply using Boolean functions? This research question is also derived from the Challenge 3 *Low quality of data*. The approaches based on the use of thresholds are designed to obtain more "sophisticated" similarity evidences because they compare instances at the more coarse-grained level of attribute values. However, the use of thresholds is sensitive to training data and always requires some manual efforts in parameter tuning to be applicable in practice. Therefore, the instance matching results that are output based on the use of thresholds may still contain a large number of non-matches. We will investigate this research question in Chapter 6.

1.5 Contribution of this Thesis

In general, this thesis is concerned with the efficient and effective instance matching solutions for heterogeneous structured data. It comprises four main contributions, each of which results from the investigation of one research question stated above and is detailed in its own chapter:

- **Contribution 1.** *Inferring the Type Semantics of Structured Data*

  We tackle the problem of deriving missing type information in structured data. We propose to deal with this novel problem of inferring the type semantics of structured data, named *typification*. We formulate this as a clustering problem, where the goal is to identify a particular kind of clusters that represent the types of entities. We discuss the main requirements, quality metrics and how existing clustering techniques can be applied. We present the experimental results using data from domain-specific product data in the enterprise to cross-domain encyclopedic data in DBpedia up to heterogeneous Linked Data on the Web. We discuss why for this particular clustering problem, schema-level features, i.e. labels of attributes and relations, are most effective in obtaining high quality results. The main intuition here is while entities of the same type are often associated with the same attributes and relations, their concrete values are quite different. However, the amount of these features is often limited, especially in the one large table setting (e.g. entities in a product table are associated with only few attributes). To deal with this, we also propose a solution for automatically computing value-level schema features from data. Optimized for the use of (value-level)schema features, we propose TYPifier, a novel clustering algorithm for the typification problem. Compared to the baselines based on existing clustering techniques that use the same (schema) features, this algorithm is comparable to the best baseline in terms of efficiency, but produces higher quality results than the baseline that performed best in terms of effectiveness. Moreover, it is able to determine the number of types (clusters) automatically. We have discussed this contribution in previous published paper [76] and present a revised version of them in Chapter 3.

- **Contribution 2.** *Type-specific Unsupervised Learning of Keys and Key Values for Heterogeneous Web Data Integration*

  For the problem of blocking, we provide a solution to solve the subtasks of selecting discriminative attributes (blocking keys) and representing their values (key values). This solution is derived for every type learned from the data. We show in experiments that our approach of using type-specific keys and values improves both blocking and instance matching. In experiments, we show how this approach can be used for blocking and instance matching. Compared to

state-of-the-art instance matching approaches, our solution greatly improves result quality (up to 201.56% improvement in terms of F-measure). Results are also promising when considering the blocking task only. Our approach yields up to 32.62% improvement in terms of reduction ratio over existing solutions for blocking [108]. It is also time efficient when compared against approaches that achieve similar result quality. Compared to the approach that yields second best result quality, our approach is up to several times faster w.r.t. blocking, and achieves similar performance w.r.t. instance matching. We have discussed this contribution in previous published paper [75] and present a revised version of them in Chapter 4.

- **Contribution 3.** *Learning Rules for Effective Almost-parameter-free Instance matching*

  We propose an efficient approach to learn attributes, similarity functions and thresholds, called instance-matching rules, for finding matches. For example, we learn that "two products are the same, if the average of the `Jaccard` similarity of `Title` and the `Cosine` similarity of `Manufacturer` is greater than 0.6". We observe that the average similarities of matches (and non-matches), that are calculated according to different attributes and similarity functions, are subjected to different probability distributions. We propose to calculate the matching and non-matching certainties, as the certainties to assign an instance pair to either a match or a non-match, from the cumulative probability of (dis)similarities of (non-)matches. Then a pair of instances will be considered as the same if its certainty to be a match is greater than that to be a non-match. Then the decision boundary can be calculated as the only one threshold, such that when a pair of instances has similarity that is greater than the threshold, its matching certainty is always greater than the non-matching certainty. Based on this observation, we further propose an efficient algorithm that searches the best rule from all rule candidates, and an efficient algorithm to execute the rule. Compared to the state-of-the-art rule learning approaches, our solution greatly improves the effectiveness as well as efficiency by up to 87% reduction of learning time. Moreover, the approach is also effective in the way that it can achieve stable results when the parameters are set with a large range of different values.

- **Contribution 4.** *Effective Parameter-free Boolean Instance Matching*

  We propose an effective parameter-free instance matching approach that relies only on relatively simple Boolean functions to filter out non-matches. Being different from existing approaches that compare instances at the more coarse-grained level of attribute values, our Boolean approach extracts evidences at the more fine-grained level of words (tokens in general) found in attribute values. Our approach learns what we call word-level dissimilarity evidences, such as

"`Apple` and `ASUS` are dissimilar words, one is in the `Title` of n_i and the other is in the `Title` of n_j". It is a dissimilarity evidence because it leads to the inference that n_i and n_j are not the same (are non-matches). We show that using this type of evidences has the following merits: (1) due to their Boolean nature, the learning of these evidences is more simple, i.e. does not require parameters and is not sensitive to training data. (2) At the word level, a large number of these evidences can be learned to identify non-matches (high recall). (3) Also due to the use of fine-grained words, the learned evidences are more discriminative in identifying non-matches (high precision). Using benchmark matching tasks, we show our Boolean solution greatly outperforms state-of-the-art approaches in terms of result quality and is superior in terms of sensitivity to training data and parameters. While our focus is to explore the direction of Boolean matching, we also discuss in the thesis how our Boolean approach can be combined with existing works that use thresholds. In the experiment, we show that when used as a Boolean filtering mechanism, our approach consistently improves the results of the underlying matching approach.

1.6 Organization of this Thesis

This thesis comprises seven chapters. Besides Chapter 2, which provides the foundations and the definitions of the main concepts used through the thesis, all the following chapters discuss one of the contributions of this thesis stated above. Each chapter starts with an introduction of the problem, restates the investigated research question, thereof derived hypotheses, and the contribution elaborated in this chapter. Then an overview of the proposed approach is presented before it is discussed in detail and investigated in experiments. Related work and existing approaches are discussed in each chapter. The thesis is structured as follows:

- **Chapter 2.** In this chapter, we introduce the areas of instance matching and the basic concepts. After an introduction of the data model, we describe the attribute matching, which is the basis of instance matching. We then show that the instance matching process in this thesis can be achieved in four sequential steps: typification, blocking, classification, and filtering. We provide the formal definition of each step before discussing the evaluation metrics that are used in the experiments throughout the thesis.

- **Chapter 3.** In this chapter, we propose to deal the novel problem of inferring the type semantics of structured data, called typification. We formulate it as a clustering problem and discuss the features needed to obtain based on existing clustering solutions. Because schema features perform best, but are not abundantly available, we propose an approach to automatically derive them from

data. Optimized for the use of schema features, we present TYPifier, a novel clustering algorithm that in experiments, yields better typification results than the baseline clustering solutions.

- **Chapter 4.** In this chapter, we discuss the method to learn type-specific blocking keys and key values for match candidates generation. We show that for the problem of learning blocking keys and key values, both generic techniques that do not exploit type information and supervised learning techniques optimized for one single predefined type of instances do not perform well on heterogeneous Web data capturing instances for which the *predefined type is too general*. That is, they actually belong to some types that are not explicitly specified in the data. We propose an unsupervised approach for *learning the type-specific blocking keys and key values*. Compared to state-of-the-art supervised and unsupervised learning approaches that are optimized for one general type, our approach improves efficiency as well as result quality.

- **Chapter 5.** In this chapter, we describe the approach to classify the match candidates to the classes of match and non-match. we propose an efficient approach to learn attributes, similarity functions, and thresholds, called instance matching rules, for finding matches. We show that our solution greatly improves the effectiveness as well as the efficiency on both real and synthetic datasets.

- **Chapter 6.** In this chapter, we propose an approach to filter out non-matches from the calculated results, which uses simpler *boolean* similarity functions rather than thresholds. We show that the simple boolean nature of the employed rules allows for a *parameter-free* learning approach. For high effectiveness, we propose to incorporate fine-grained word-level dissimilarity evidences into rule learning. That is, instead of capturing the similarity of entire attribute values in the rules, our approach employes words extracted from attribute values to capture the dissimilarity. Using benchmark matching tasks, we show the proposed solution greatly outperforms state-of-the-art approaches in terms of result quality and most importantly, is not sensitive to the choice of training data and parameters.

- **Chapter 7.** This chapter summarizes the thesis, concludes with a discussion on the results with respect to the addressed research questions and gives an outlook on future research directions.

The cited references are given at the end of the thesis in the bibliography followed by the lists of Figures, and Tables. The appendix contains the additional analysis presented in the main part of the thesis.

Chapter 2

Foundations

In this chapter we introduce the areas of instance matching and the basic concepts, clarify how certain terms are used in this thesis. Definitions that are only used within one chapter and specific to the chapter's topic are given in the corresponding chapter. After introducing the data model for the structured data in Section 2.1, we describe the techniques for attribute matching in Section 2.2. Then we show that the matching of two datasets in this thesis can be achieved in four sequential steps: preprocessing, candidate calculation, classification and filtering, as shown in Fig. 2.1. First, we preprocess each dataset that is to be matched, such as removing the stop words to make the attribute value more discriminative and segmenting the attributes to make them comparable. Second, we quick calculate the matching candidates that exclude most non-matches so that we can improve the efficiency by only comparing the candidates in details. Then we further compare the candidates to classified them to the class of match and non-match. Finally, the matching instance pairs are filtered using dissimilarity evidences to further remove the non-matches that are not identified in the classification step. The formal definitions of each step are given in Sections 2.3 - 2.6, respectively. Finally we discuss the metrics used in the experiments throughout the thesis.

2.1 Data Model

We consider general Web data, which includes different types such as relational, XML and RDF data. In literatures, this data is often conceived as a graph:

Definition 2.1. *The* data graph *G is a tuple (U, E, L) where*

- *U is a finite set of nodes. Thereby, U is conceived as the disjoint union $U_N \uplus U_C \uplus U_V$ with U_N representing* instances*, U_C stands for* classes*, and U_V stands for* attribute values*.*

- *L is a finite set of labels, subdivided by $L = L_U \uplus L_E \uplus type$, where L_U are node labels and $L_E \uplus type$ are edge labels.*

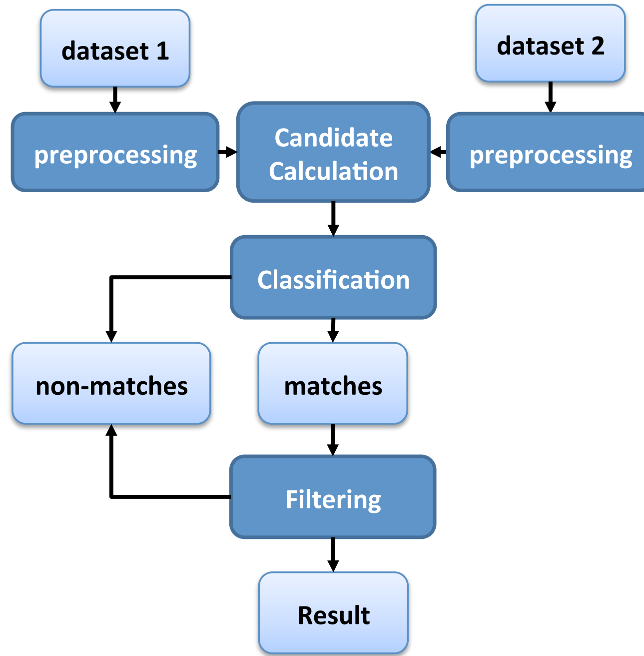


Figure 2.1: The Instance Matching Process Applied in This thesis.

- *E is a finite set of edges $e(u_1, u_2)$ with $u_1, u_2 \in U$ and $e \in L$. In particular, we distinguish* attribute edges *$E_A \subset E$, with $e(u_1, u_2) \in E_A$ iff $u_1 \in U_N$ and $u_2 \in U_V$ from* type edges, *$type(u_1, u_2) \in E$, $u_1 \in U_N$ and $u_2 \in U_C$.*

This graph-structured model resembles the RDF model, (omitting special features such as RDF blank nodes), where classes correspond to RDF classes, instances are RDF resources, and attribute values are RDF literals. A record in a relational database captures an instance and its attribute values. For clarity, we omit relationships, e.g. relations between RDF resources and foreign key relationships between records, because they are not used for instance matching here. The graph for our data example is shown in Fig. 2.2 according to the data in Tab. 2.1.

Example 2.1. *As a running example, Tab. 2.1 shows real product data. Using four attributes, the original product table captures* `electronic products`*, which actually belong to subtypes such as* `Computer`*,* `TV` *and* `software` *(e.g. n_8). The data graph in Fig. 2.2 captures the same products, albeit using a more complex schema with many more attributes. The dotted edges in Fig. 2.2 and the lack of a type column in Tab. 2.1, indicate missing type information.*

For the ease of presentation of the thesis, we simply use N and A to denote all the instances and attributes in the data source respectively. Each instance $n \in N$ is represented by a set $\{a_1, a_2, \cdots a_m\}, a_i \in A$ of attributes. The value of an attribute a_i

Table 2.1: Product table; some words later used as features for illustration purposes, are highlighted.

ID	Title	Price	Manufacturer	Description
n_1	Sony KDL-60R550A	1252	Sony	Standby Mode 0.1 **W**. Full **HD 1080p** gives high picture quality over standard HDTV via Sony **LED**... Sony's 60-inch Smart **TV** is a revolutionary...
n_2	Sony KDL-40R450A	849	Sony	Sony 40-inch Slim **LED HDTV**. Never miss a moment with Sony Smart TV...This **LED HDTV** allows you to immerse yourself in a **1080p** full high-definition viewing experience
n_3	Sony KDL-60R550A	1179	Sony	65 **W** total power. The **LED** series provides a fantastic Smart **TV** experience and features a 3D **LED** panel, **1080p** Full **HD** resolution, and a new narrow metal frame.
n_4	Sony Fit Series VAIO SVF14214CXW laptop	665	Sony	Sony SVF14214CXW laptop... Intel Core i5-3337U **CPU**, 6GB **RAM**, 750 GB 5400 rpm HDD, 14-Inch Screen, 65 **W** 3.3A Power Supply
n_5	Sony VAIO SVF14212CXW	586	Sony	the stylish Sony VAIO Fit **laptop** is built with Intel Core i3 **CPU**, 4GB **RAM**, 500GB Hard Drive... with power supply of 65 **W**
n_6	Sony VAIO SVF14214CXW Laptop	680	Sony Inc.	Sony VAIO Fit Series 14-inch **laptop**; AC 65 **W** adapter, 1.8 GHz Intel Core i5-3337U ultra-low voltage **CPU**, 6 GB of installed DDR3 **RAM**...
n_7	MacBook Pro ME664LL	999	Apple Inc.	Apple MacBook Pro ME664LL **Laptop** with Retina Display (Intel Core i7, 2.4GHz, 8GB **RAM**, 256G hard disk, NVIDIA GeForce GT 650M)
n_8	MadMaps Pacific	8	SpotItOut	**Windows** Vista / 7 / **XP**. Media: **DVD**. It's a snap to load Pacific Coast **GPS Travel** Directory by MAD **Maps** into your GPS device.
n_9	Garmin nüMaps	99	Garmin	**Windows** Vista / 7 / **XP**. Media: **DVD**. Compatible with **GPS** Garmin Colorado, Dakota, eTrex...Coverage includes detailed **maps** for **traveling** in Australia.
n_{10}	Rosetta Spanish	399	Rosetta Stone	**Windows** Vista / 7 / **XP**. Media: **DVD**. Build your **vocabulary** and **language** abilities... Discover how to speak, read, write, and understand...
n_{11}	Learn German	10	Innovative	**Windows** Vista / 7 / **XP**. Media: **DVD**. Learn level 9 German **vocabulary** with the audio playback tool, Listen to the lesson dialog and master the **language**...

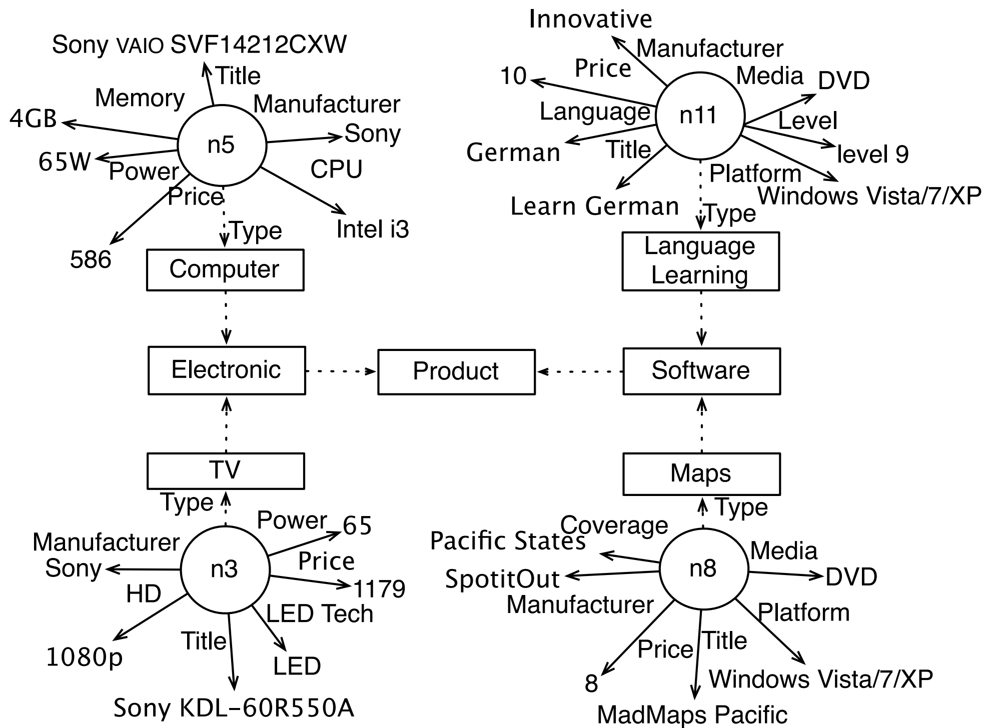


Figure 2.2: A data graph. The solid and dotted line denote the attribute edges and type edges respectively.

of the instance n (or a set of values of instances in N) is denoted by $n[a_i]$ ($N[a_i]$). For example, the `Title` of the instance n_1 is denoted as $n_1(\texttt{Title})$.

2.2 Attribute Matching

Attribute matching is the basis of instance matching, because comparing two instances comes down to comparing each attribute of both instances. However, the data to be matched can usually be of low quality with errors and typographical variations of attribute values. The nosies in attributes raise two questions. The first question is that instead of identical comparing two attributes, how to develop string comparison techniques, called *similarity functions*, to calculate the value that indicate how similar two attribute values are. And the second question is how to select the best similarity function for a specific type of attribute. We discuss the approaches that solve these questions in the following part of this section and section 2.5,respectively.

A similarity function maps two strings to a similarity value between 0 and 1. The larger the value, the similar the two strings. We consider two types of similarity

functions in this thesis, which are formally defined as follows:

Definition 2.2 (Similarity Function)**.** *A* Boolean function *same* : $N[a] \times N[a] \rightarrow \{0,1\}$ *maps a pair of attribute values to the score* 0 *or* 1*. A* thresholded similarity function *sim* : $N[a] \times N[a] \rightarrow [0,1]$ *maps a pair of attribute values to a score in the range* $[0,1]$*. A larger similarity score indicates a higher similarity between two attribute values.*

A boolean function only outputs binary values, in which 1 indicates two string are matching and 0 indicates they are not. Boolean functions typically depend on identical comparison. For example, let the prefix of a string s be the first x characters that is denoted with $s[1..x]$, we can define a boolean function that outputs 1 for two strings if they have the same prefix:

$$same(s_1,s_2) = \begin{cases} 1 & if s_1[1..x] = s_2[1..x] \\ 0 & otherwse \end{cases} \tag{2.1}$$

The use of identical comparisons in boolean functions allows computing matching candidates efficiently. For example, we can build inverted index using the prefix of attribute `Title` of each product, so that every two products who have the same inverted index will have the similarity 1 according to the equation 2.1.

The other type of similarity functions compare two strings and output normalized similarity values between 0 and 1 if the two strings are somewhat similar to each other. The similarity of 1 corresponds to an exact match between two attribute values, while 0 corresponds to a total non-match. A value between 0 and 1 indicate some degrees of similarity between two attribute values. We name these functions with threshold similarity function, because we still need a threshold value to determine whether two strings are matching. We will discuss several typical strategies of designing threshold similarity functions in the following part of this section.

Overlap-based similarity function. Because there are errors and typographical variations, comparing two matching attribute values as a whole would result in very low similarity. One strategy to solve the problem is splitting the whole string into a set of substrings, and then compute the similarity value based on the overlap of the two substring sets. The higher the overlap, the higher the similarity. Effective overlap-based similarity functions involve determining 1) computing substring set, 2) the weight indicating the importance of each substring, and 3) calculating the similarity. The substring set are usually generate with each words in the string or the n-grams that are contiguous sequence of n character (or words) [51, 68, 110, 114, 115] (task 1). And tf-idf are most commonly used in evaluating the importance of each substring (task2). Finally, given two sets of substring, the similarity is calculated using set-based approaches such as Jaccard [111] and Cosine [111]. Current approaches differ in the way to implement each of these three tasks. For example, `WHIRL` [15] splits

each string into `words`. And then every word is assigned with a weight using tf-idf. Then it calculates the similarity using cosine function. An extension of `WHIRL` is to use n-gram instead of words to generate the substring set [32].

Another type of thresholded similarity functions are designed based on measuring the cost of converting one string to another. An typical example is `edit distance` [69, 70, 85, 86, 98] that count the smallest number of edit operations in the conversion, including character insertions, deletions and substitutions. `Smith-Waterman distance` is an extension of `edit distance` which allows for gaps as well as character-specific match scores and costs [15].

Besides these general-purpose similarity functions, there are also similarity functions that are designed for specific types of data. For example, `Soundex` [87] is a phonetic similarity function, which is used for comparing names to find attribute values that are phonetically similar even if they are not similar in character or token level. `Jaro` and `Winkler` are mainly used for comparison of last and first names. There are also functions for dates, numbers and locations to compare attributes that contain such data. The readers who are interested in similarity functions can find the more detailed introduction from the surveys [28, 34, 40, 67]. There are also open-source projects that implement these similarity functions, such as SecondString[1] and SimMetrics[2].

2.3 Preprocessing

In this section, we first introduce the typical preprocessing tasks that are used in instance matching. Then we give the definition of `Typification`, a new preprocessing task that groups instances according to their types when the types of instances are unknown or too general to be useful.

2.3.1 Typical Preprocessing Tasks

Due to the variations and errors in the data, the attributes are usually not directly comparable. For example, we cannot directly compare two attributes, if one of them contains the whole name of a person and the other one contains only the family name. Moreover, note we always want the values being representative for instances in instance matching tasks. It is also necessary to preprocess the data to remove unwanted words. Finally, before matching two datasets, we should also guarantee that the attributes are all consistent to each other. For example, to ensure the gender does not conflict with the name, if the name is obviously uniquely for male or female. In sum, the preprocessing task refers to modifying the raw data for the purpose of efficient

[1] http://secondstring.sourceforge.net/
[2] http://sourceforge.net/projects/simmetrics/

and effective instance matching. Tab. 2.2 provides a summary of the preprocessing methods according to their purpose, each of which is discussed in details below:

Table 2.2: Preprocessing methods. The methods that start with asterisks are proposed in this thesis.

Purpose	Method
Attributes values being comparable	1. Word segmentation 2. Stemming and lemmatization 3. Standardization 4. Processing missing value 5. Attribute segmentation *6. Typification
Attribute values being discriminative	1. Removing stop words *2. Removing type-specific words
Attribute values being consistent	1. Attribute verification

- Attributes values being comparable:
  - **Word segmentation**: most similarity functions, e.g. Jaccard and Cosine functions, assume that the attribute values are segmented, i.e. values that are separated by word delimiter like whitespace character. However the word delimiter is not found in all data, such as data in the language of Chinese and Japanese, where sentences but not words are delimited. In practice, we can take use of nature language processing (NLP) technology to segment the text into words [79].
  - **Stemming and lemmatization**: even the same words can have different forms, e.g. the words "stemmer", "stemming" and "stemmed" are all based on the same root "stem". Note instance matching algorithms are mostly designed based on comparing the attribute values. The different forms of the same word provide false dissimilarity evidences, i.e. indicate incorrectly that two attributes are non-matching. To solve the problem, stemming is to reduce inflected words to their word stem. For instance, it will replace the tokens "stemmer", "stemming" and "stemmed" with the base form "stem". Lemmatization is more complex than stemming, which first determines the part of speech of a word, and then apply different normalization rules for each part of speech. For example, it will replace the word "better" and "best" with their lemma "good".
  - **Standardization**: for the same reason as stemming and lemmatization, standardization is to convert the tokens in attributes value to their stan-

dard forms, such as expanding the abbreviations and correcting the typographical errors or variations. For example, replacing the city name "CA" with the full name "California", or replacing the nicknames of people with their standard names. Standardization can be achieved by looking up dictionaries that contain the standard forms of tokens and their known typographical errors and variations. Standardization ensures that all the datasets to be matched using the same vocabulary, so that the attributes are comparable.

  - **Processing missing value**: we can simply remove the attributes that contain missing values but are rarely used by instance matching. However, if the attributes are important for the instance matching, we should keep them and fill the missing values using domain knowledge. For example, filling the missing values in attribute "brand" with the words that refer to brands in attribute "title" by looking up the brand dictionary. The domain knowledge can be obtained either from domain experts or with the help of machine learning technologies. For instance, we can train a classifier to infer the gender of people using the values of the attributes such as "first name" and "hobby".
  - **Attribute segmentation**: the attributes, such as address, name, date and telephone number, are usually a combination of some more fine-grained attributes. For example, a name can be further split into first name and last name. And an address can be split into stress, house number, postcode, city and country. Instances from two different datasets to be matched can be incomparable if one of them uses single attribute "full name" and the other one use two attributes "first name" and "last name". To solve the problme, we can segment the coarse-grained attribute into multiple fine-grained attributes, which can be achieved with rule-based approaches, statistical approaches and hidden Markov model based approaches [28]. Attribute segmentation can also improve the efficiency of comparing two instances. For example, rather than comparing the full name, we can terminate the name comparisons earlier without comparing first names if two last names are non-matching.
  - **Typification**: when the types of instances are unknown or too general to be useful, we proposed a new preprocessing task called `Typification`, which will automatically group instances according to their types. The definition of Typification is provided in Sec. 2.3.2.

- Attribute values being representative for instance:
  - **Removing stop words**: while stop words such as short function words like "a" and "the" are useless in discriminating instances, they provide false

similarity evidences when compute the attribute similarity, i.e. indicate incorrectly that two attributes are matching. Therefore, stop words should usually be removed before attribute matching.

- **Removing type-specific words**: we distinguish the instance-specific and type-specific words when compute attribute similarity. For example, given two strings like "`Sony VAIO SVF13213CXW laptop`" and "`Sony VAIO SVF14212CXW laptop`", it is easy for human to infer that they are titles of two different products because the words "`SVF13213CXW`" and "`SVF14212CXW`" indicate two different models. We treat the words like "`SVF13213CXW`" as instance-specific words since they are representative for instances. However, machines may identify the two stings as a match because of the overlap of the words that representative for types rather than instances, such as "`Sony`" and "`laptop`", that result in high similarity. We treat these words as type-specific words, and remove them from attribute values when compare attributes of the same-type instances. We will discuss removing type-specific words in Sec. 4.5.2

- Attribute values being consistent:
  - **Attribute verification**: before applying instance matching, one optional preprocessing step is to verify the existence or correctness of the attribute values. For example, ensuring the birthday is a valid date or ensuring the existence of an address. Again, domain knowledge, such as a database of address, is usually required in this step to verify the attribute values.

2.3.2 Typification

Currently, most instance matching approaches assume the data to be matched is homogeneous, i.e. the type and the schema of the data are known. However, these approaches do not perform well, when the type information is missing or too general to be useful. Therefore, in this thesis we propose another preprocessing step, called typification, whose purpose is to restore the type information of data. Typification can be formulated as a clustering problem:

Definition 2.3 (Typification)**.** *Given a set of instances $N = \{n_1, \ldots, n_j, \ldots, n_{|N|}\}$, typification attempts to seek a K-*partition *of N ($K \leq |N|$) into* clusters $\mathbf{C} = \{C_1, \ldots, C_K\}$*, such that (1) $C_i \neq \emptyset, i = 1, \ldots, K$, (2) $\bigcup_{i=1}^{K} C_i = N$ and (3) $C_i \cap C_j = \emptyset, i, j = 1, \ldots, K$ and $i \neq j$. Moreover, when missing subtype edges are also considered, it seeks* hierarchical clusters*, i.e., a tree-like nested structure partition of N (called* hierarchy tree*), $\mathbf{H} = \{H_1, \ldots, H_Q\}$ ($Q \leq N$), s.t. $C_i \in H_m$, $C_j \in H_l$ and $m > l$ imply $C_i \subseteq C_j$ or $C_i \cap C_j = \emptyset$ for all $i, j \neq i, m, l = 1, \ldots, Q$.*

Related to the purpose of typification is the work on *deriving schema information from semi-structured data*, such as DataGuide [50]. Applied to this setting, a DataGuide is

basically a summary that groups entities together, which exhibit the same structure (as captured by incoming and outgoing paths). For instance, the entities x of the type *Product* in the data, $\langle x, type, Product \rangle$, are represented by one single node corresponding to *Product* in the summary. Instead of computing a summary based on the given structure in the data, the problem here is to infer elements in the data that are missing. More related is recent work on *recovering the semantics of Web tables* [73, 116]. For this, a database of type (and attribute) labels is used. A type (attribute) label is associated with the column of a table if values in that column can be matched to that label. Instead of assigning a column to a particular type label, the problem here is to assign entities (tuples in a table) to possibly different types. In other words, the problem here is not the one of recovering the particular type (and attribute) semantics of a table but the different *type semantics of entries in tables*.

2.4 Candidate Generation

A brutal force algorithm for instance matching is to compare all the possible instance pairs. Such an algorithm is very inefficient, because for two dataset A and B, there are at most $|A| \cdot |B|$ instance pairs to be compared. There are generally two strategies to improve the efficiency: 1). reducing the cost of single instance comparison, and 2). reducing the number of instance pairs to be compared. Note one single instance comparison actually requires multiple attribute comparisons, i.e. calculate attribute similarity using certain similarity functions. So the first strategy actually refers to either comparing less attributes or using cheap similarity functions. We will discuss the technologies that can lead to less attribute comparisons in section 2.5.2. And the use of cheap similarity functions is applied in the canopies approaches [33, 81, 84] that adopt the second strategy to generate matching candidates. In the following part of this section, we will further introduce the approaches that focus on the second strategy to reduce the number of instance comparisons.

Generally, to reduce the number of instance comparisons, the approaches that are designed for the second strategy will generate matching candidates by assigning the instances that are similar to each other to the same group. Then, only the instances in the each group need to be further compared to find the true matches. Typical candidates generation approaches are *sorted neighborhood approach*, *Canopy*, and *blocking*. These approaches differ in how to generate the instance group.

The **sorted neighborhood approach** consists of three steps [127]. First, a key value for each instance is extracted from certain attributes. Then all the instances are sorted according to the key value. Finally a fixed size window is slide through the sorted instances, so that the instances inside the window form a group and only the instances inside the group are compared.

There are two main drawbacks of the sorted neighborhood approach. The first

drawback is that the method is sensitive to the selection of key values. When there are noises in the data, it is possible that two matching instances are not close enough to each other to be included in the window after instance being sorted. This problem can be solved by multiple iterations of the method using different key values in each of the iterations [54]. The other drawback is that the fixed-size window may not cover all the matches if there are a large number of matches or the matches are not close enough in the sorted instances. This problem can be solved by using dynamic window size [127].

Canopies approach [33, 81, 84] forms a cluster by random selecting a instance and assign into its cluster with all the other instances whose similarity that calculated with a cheap similarity function is above a certain threshold. After the grouping step, the instances in each cluster are compared pairwise using a more expensive similarity function. For example, if the length difference of two strings is 4, then their edit distance is at least 4. However, the cost to calculate the length difference of two strings is much cheaper than that of the edit distance. In this way, the string length difference can be used as a cheaper version of edit distance. A canopy cluster can be formed by randomly select an instance and put in its cluster with instances that have attribute value length difference below 4. Then the instance in the same cluster can be further compared using edit distance [52].

In this thesis, we focus on the technique named ***blocking***, which subdivide the original data into a set of mutually exclusive subsets, named blocks, assuming that there are no matches across different blocks [54, 71, 104, 123]. As the result, only the instances in the same block will be further compared in detail. Blocking is typically limited to finding attributes also called *blocking keys* and their *value representation* to calculate the so called blocking key values (BKV) [16, 83, 93, 123]. Previous works [93, 108] use the best-ranked attribute [46, 56, 58, 108] as key and all features from its values as key values (schema-based), or the values of attributes combined [93] (omitting attribute information, hence called schema-agnostic). The value representation is the set of tokens that can be extracted from the key value. Then instances are considered as matching candidate and placed into one block when there is an overlap in their BKV. In practice, there are different value representation methods to generate BKV, such as using the prefix [124], the suffix [35] or the q-gram [52] of the blocking key. Resembling the blocking problem, we focus on finding attributes and their value representation in this blocking context:

Definition 2.4 (Find Blocking Keys and Key Values)**.** *Given the data graph $G(U, E, L)$, we find a conjunction of Boolean function predicates (called* blocking scheme*) $\bigwedge_{e \in L^*} \sim^e_V$ where $\sim^e_V$: $U^e_V \times U^e_V \rightarrow \{true, false\}$, $U^e_V \subseteq U_V \subset U$ denotes the values of some* keys *$e \in L^*$, and $L^* \subset L$ is the set of* blocking keys*. Blocking maps every instance $n_i \in N$ to a subset $N_{B_i} \subseteq N$, an equivalence class of instances (instances, that according to the blocking scheme, are equivalent to n_i) called* block*: $N_{B_i} = [n_i] = \{n_j \in N : n_i, n_j \in N, \bigwedge_{e \in L^*} \sim^e_V (n_i, n_j) =$

Table 2.3: Example blocks for different subtypes using attribute Title as blocking key and the words Sony and VAIO as key values.

Block	ID	Title	Subtype
$block_1$	n_1	**Sony** KDL-60R550A	TV
	n_2	**Sony** KDL-40R450A	TV
	n_3	**Sony** KDL-60R550A	TV
$block_2$	n_4	Sony **VAIO** SVF14214CXW	Computer
	n_5	Sony **VIVO** SVF14212CXW	Computer
	n_6	Sony **VAIO** SVF14214CXW Laptop	Computer

true}.

Example 2.2. *Tab. 2.3 shows two blocks obtained using* `Title` *as key and words in* `Title` *as key values. Note instances in the same block overlap on the word* `Sony` *and* `VAIO`. *In addition, based on the result of typification, it recognizes that instances belong to the two types* `TV` *and* `Computer`.

Additionally, an additional matching predicate (Jaccard similarity) has been applied on top to filter those candidates that do not exceed the predefined threshold θ [108]. Thus, candidates that remain in one block overlap on some key tokens and their key value similarity exceeds θ.

There are multi-pass approaches for blocking, where at every pass, a conjunction of keys is used to generate candidates (e.g. match the last two letters of `Manufacturer` and first 3 tokens of `Title`). The union of all results is used because different passes cover different matches. Accordingly, blocking can be seen as a disjunction of conjunctions.

2.5 Classification

In the classification step, matching candidates are further compared to be classified to the class of match or non-match. The classification task can be solved by a general binary classifier with the help of machine learning. Let $G = \{g_1, g_2, ...\}$ be a set of similarity functions and $A = \{a_1, a_2, ...\}$ be a set of comparable attributes of two datasets to be matched, we can present a matching candidate as a vector $x = \{x_1, x_2, ..., x_m\}$, where $m = |G| \cdot |A|$. Each x_i in the vector shows a similarity calculated by applying a certain similarity function on a certain attribute. Many approaches also use binary values for the x_i in the vector and set $x_i = 1$ if the corresponding attribute is matching and set $x_i = 0$ if the attribute is non-matching. Then provided a set of matching and non-matching instance pairs as positive and negative examples, we can solve the

classification task by converting the training examples to feature vectors, and use the feature vectors to train classifiers such as Bayes classifier [44] or SVM [14].

The main drawback of using general binary classifiers for instance matching is that it is inefficient to calculate the feature vector of the length $|G| \cdot |A|$ since similarity calculation is expensive. Although feature selection can reduce the dimension of the feature vector, it is not guarantee that the number of similarity computations is minimized. Note not all attributes are important for instance matching task, and not all similarity functions should be used for each attribute. It is possible to select a subset $A' \subseteq A$ of all attributes for instance matching, and for each attribute the best similarity function. In this way, each instance pair comparison requires only $|A'|$ similarity computations, which is far less than that need for a general binary classifiers. In practice, the combination of attribute and similarity functions can either be assigned manually by experts or learned using training examples. In general, the instance matching approaches in the classification step need to answer two questions: 1). how to learn instance matching schema, i.e. the combination of attributes and similarity functions, as well as the thresholds and weight for each attribute, and 2). how to efficiently execute the instance matching schema.

2.5.1 Supervised Learning of Instance Matching Schema

The instance matching schema can be characterized through the tuple $(\sim_N, \sigma)$, where $\sim_N$: $N \times N \rightarrow \mathbb{R}^+$ maps a pair of instances to a similarity value, and they are considered as being the same when that similarity value is higher than the threshold θ [23, 42, 49, 72]. The instance matching schema learning usually involve determining (1) (combinations of) attributes (`Title`, `Price` etc.) and for each of them (2) the similarity measures (e.g. edit distance and Jaccard similarity), (3) the similarity thresholds [14, 24, 83, 112, 113], and (4) the weights [55, 61, 88].

Let G be a set of similarity functions, A be a set of attributes, and M be a set of training examples. We should consider all the possible combinations of attributes and similarity functions to learn the instance matching schema. For example, let $\{a_1, a_2\}$ be attributes and $\{g_1, g_2\}$ be similarity functions to use, and $g_i(a_j)$ be an association of similarity function g_i with attribute a_j, then there are overall eight combinations including: $g_1(a_1)$, $g_2(a_1)$, $g_1(a_2)$, $g_2(a_2)$, $g_1(a_1)g_1(a_2)$, $g_1(a_1)g_2(a_2)$, $g_2(a_1)g_1(a_2)$, and $g_2(a_1)g_2(a_2)$. Then, for each instance pair in the training data, if we use the similarities that are calculated by these combinations of attributes and similarity functions as threshold, we can get eight candidates of instance matching schema. Finally, the instance matching schema learning problem is converted to finding the candidate that can maximize a quality function from all the candidates that are converted from all the training examples. The quality function can usually be F-measure or any others that are used to evaluate instance matching. The overall number of instance matching schema candidates is about $|G|^{|A|} \cdot |M|$. However, since we may consider more

than twenty similarity functions in practice, even a small set of training examples can result in a huge set of schema candidates. Actually, research showed that this problem is NP hard [24]. The problem can be solved by detecting and avoiding processing invalid candidates. Chaudhuri et al. [24] proposed a Hyper Rectangle algorithm to learn the combination of attributes, similarity functions and thresholds. Wang et al. [119] further improve the efficiency by eliminating candidates that are composed of redundant similarity functions and thresholds. Robert et al. [61] proposed an approach that can also learn the weight of attributes using genetic programming. Thomas et al. [55] learn the weight of an attribute by considering the probability that matches and non-matches having the same value on the attribute. The frequencies of attribute values are also considered when computing the weight for a specific instance pair [88].

2.5.2 Efficient Execution of Instance Matching Schema

Similar as that discussed in blocking, the efficiency of executing instance matching schema can also be improved by either improving efficiency of *single* instance pair comparison or reduce the number of matching candidates. In this section, we first show that the so-called rule-based instance matching and threshold-based instance matching can compare two instances efficiently by using less attribute comparisons [40]. Then we discussed how to generate less matching candidates with the help of instance matching schema.

The rule-based instance matching approaches [24, 61] can be efficiently executed because they need less attribute comparisons. These approaches employ rules that are written as a conjunction of attribute matching. For example, for the product dataset, it can be written as Equation. 2.2.

$$Jaccard(title) \geq 0.9 \wedge Cosine(brand) \geq 0.8 \tag{2.2}$$

This type of rules have the very similar form as the output of decision tree learning. For example, Equation. 2.2 can be converted to a decision tree as shown in Fig. 2.3. Because only the instance pairs that satisfy each attribute matching in the rule can be classified as matches, we call it *attribute-threshold instance matching rule* in this thesis, whose formal definition is given in Sec. 2.5.3. Since an input instance pair will be classified to non-match when it cannot satisfy any one of the attribute matching in the rule, we are able to early terminate comparing attributes if an attribute matching returns false. This ability that allows the program to stop comparing attribute earlier results in less attribute comparisons.

Another type of classification approach is the so-called **threshold-based instance matching**, which uses a single threshold for the sum of attribute similarities instead of assigning different thresholds for every attribute. In this thesis we call it *mapping-*

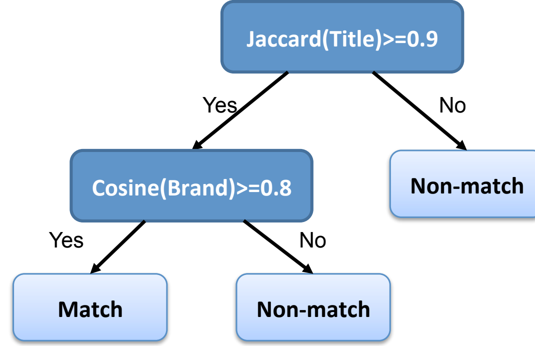


Figure 2.3: The Instance Matching Progress Applied in This thesis.

threshold instance matching rule, whose formal definition is given in Sec. 2.5.4. Consider the rule in Equation 2.3 as an example. Although it seems that executing rule 2.3 needs iterating all the attribute similarities to calculate the sum, in Sec. 5.5 we will show that it is also possible to early terminate calculating attribute similarities by checking whether the maximum values of the similarities that are not calculated can satisfy the threshold. For example, if we already calculate $Jaccard(Title) = 0.5$ for a pair of instances and we know $0 \leq Cosine(Brand) \leq 1$, then we can predicate that the instance pair cannot satisfy the rule even if $Cosine(Brand)$ reach the maximum value.

$$Jaccard(title) + Cosine(brand) \geq 1.7 \tag{2.3}$$

Currently, we have introduced how instance matching schema are used to efficiently classify the matching candidates. The instance matching schema can also feed back into the candidate generation step to output less candidates. No matter the Canopy or the blocking approach, the candidates are usually generated using an inverted index. The key of the index can be *all* the tokens extracted from the a certain attribute, and the value of the index is the instances that share the token. Then the matching candidates are generated as pairs of instances that indexed by the same token. Therefore, the less the token that are used as keys in the inverted index, the less matching candidates generated. Once the instance matching schema, i.e. the combination of attributes, similarity functions and thresholds, is determined, we can use the schema as a constraint to restrict the number of tokens to be used in the index. Take Jaccard similarity function as an example. We have a constraint $Jaccard(a) \geq t$, where a is an attribute and t is a threshold. Let the tokens in attribute value x be sorted, and the p-prefix of x be the first p tokens of x. It can be shown that we only need to index a prefix of length $|x| - \lceil t \cdot x \rceil + 1$ for every instances to ensure that the candidate generation task does not miss any pair of instances that satisfy the constraint $Jaccard(a) \geq t$ [124]. For Cosine constraint $Cosine(a) \geq t$, the length of the

tokens to be indexed can also be optimized to $|x| - \lceil t \cdot x \rceil + 1$ [124]. The edit distance constraint $EditDistance(a) \leq t$ can be converted to the constraint on the overlap between the q-gram sets of the two strings. Then the length of the tokens to be indexed can be optimized to $qt + 1$ [124].

2.5.3 Attribute-threshold Instance Matching Rule

We explicitly distinguish two type of instance matching rules: *attribute-threshold instance matching rule* (aIR) and *mapping-threshold instance matching rule* (mIR). Let $\{g_1, g_2, \cdots\}$ be a set of similarity functions such that $0 \leq g(n[a], n'[a]) \leq 1$. A higher score indicates a higher similarity between $n[a]$ and $n'[a]$. For the convenience of expression, we loosely use $g(a)$ to denote an association of a similarity function g with an attribute a. For example, we use $Jaccard(\texttt{Title})$ to denote the similarity of attribute `Title` that is calculated by *Jaccard* similarity function. Then aIR can be defined as follows:

Definition 2.5 (Attr.-thresh. Inst. Match. Rule). *Given a set $\{a_1, a_2, \cdots, a_d\}$ of attributes, and a set $\{g_1, g_2, \cdots, g_d\}$ of similarity functions, let $g_i(a_i) \geq \theta_i$ denote a* similarity function predicate *where $0 \leq \theta_i \leq 1$. For any two instances n and n', the similarity function predicate returns true if $g_i(n[a_i], n'[a_i]) \geq \theta_i$. An attribute-threshold instance matching rule is a conjunction of similarity function predicates as $\bigwedge_{i=1}^{d} g_i(a_i) \geq \theta_i$. A pair of instances n and n' are considered as a match if they satisfy all the similarity function predicates in the rule.*

Example 2.3. *Suppose an aIR $Jaccard(\texttt{Title}) \geq 0.80 \wedge Cosine(\texttt{Manufacturer}) \geq 0.70$ that is designed for the data in Tbl. 2.1. Because $Jaccard(n_4[\texttt{Title}], n_6[\texttt{Title}]) = 0.82 > 0.80$ and $Cosine(n_4[\texttt{Manufacturer}], n_6[\texttt{Manufaccture}]) = 0.71 > 0.70$, the instance pair (n_4, n_6) is identified as a match.*

Existing aIR-based approaches are designed according to *attribute monotonicity*, which requires that "any pair of matching records have a higher similarity value than a non-matching pair on at least one attribute" [24]. If an instance matching problem is attribute monotonic, these approaches are able to learn aIR to correctly identify all the matching instances. Comparing to the general machine learning techniques (e.g. SVM and decision tree [24, 119]), aIR can be executed more efficiently because of less calculations of similarity function predicates. Especially, it is executed efficiently in the way that, when any one of the similarity function predicates in a rule return false, all the rest predicates need not to be tested any more.

2.5.4 Mapping-threshold Instance Matching Rule

However, the attribute monotonicity may not always exist in real-world data due to various errors in a single attribute. Therefore, instead of exploiting attribute monotonicity, we consider a more general property for the instance matching problem,

called *mapping monotonicity*. We say an instance matching problem is *monotonic* if, for any matches (n_1, n'_1) and non-matches (n_2, n'_2), there exists a set of attributes such that the average similarity of these attributes of (n_1, n'_1) is greater than the average similarity of (n_2, n'_2). We observe that the mapping monotonicity exists more generally, because even though errors might occasionally occur in one attribute, it is unlikely that they happen in all attributes of a match. Based on the mapping monotonicity, we formally define the *mapping-threshold instance matching rule* (*mIR*) as follows:

Definition 2.6 (Map.-thresh. Inst. Match. Rule). *Given a set* $\{a_1, a_2, \cdots, a_d\}$ *of attributes and correspondingly a set* $\{g_1, g_2, \cdots, g_d\}$ *of similarity functions, a* rule function $f : N \times N \to [0,1]$ *calculates the similarity between two instances as the average of every attribute similarity, i.e.* $f(n, n') = \frac{\sum_{i=1}^{d} g_i(n[a_i], n'[a_i])}{d}, n, n' \in N$. *Given a threshold* θ, *a mapping-threshold instance matching rule (mIR) is defined as a tuple* $\lambda(f, \theta)$, *such that two instance* $n \in N$ *and* $n' \in N$ *are considered as a match if* $f(n, n') \geq \theta$, *where* $0 \leq \theta \leq 1$.

Example 2.4. *Assuming a mIR that is designed for the data in Tbl. 2.1 as* $\frac{Jaccard(\texttt{Title}) + Cosine(\texttt{Manufacturer})}{2} \geq 0.70$. *The instance pair* (n_4, n_6) *is identified as a match, because* $\frac{Jaccard(n_4[\texttt{Title}], n_6[\texttt{Title}]) + Cosine(n_4[\texttt{Manufacturer}], n_6[\texttt{Manufacturer}])}{2} = 0.76 > 0.70$.

2.5.5 Collective Instance Matching

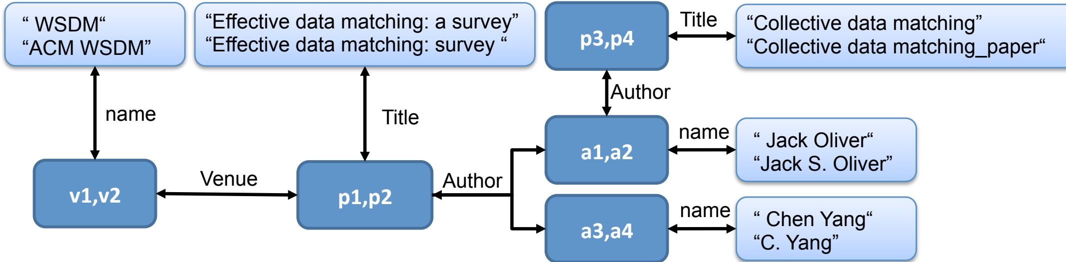


Figure 2.4: A Relationship Graph for Collective Instance Matching.

Currently we have introduced the techniques that are used to compare two instances locally based on attribute comparisons. Different from the local approaches, collective approaches make match decisions holistically by also considering the matching of related instances [12, 39, 82, 96]. In other words, not only attributes but also relations are utilized in instance matching based on the assumption that the similar instances should also have similar related instances. One example is to match two bibliographic datasets collectively. Typically, a relationship graph is usually built first, which is as shown in Fig. 2.4 where nodes "p1,p2" and "p3,p4" are pairs of papers, "a1,a2" and "a3,a4" are pairs of authors and "v1,v2" is a venue pair. The initial

similarity of each nodes is from the aggregation of their attribute similarities. Then the approach propagate the similarities through the edges of the graph. As a result, a pair of instances can gain more similarity if it has more neighbors with higher similarities in the graph.

2.6 Filtering

As discussed, because the use of thresholds are sensitive to training data and parameter tuning, the instance matching results may still include a large number of non-matches. Comparing to the *thresholded similarity functions* that computes a similarity score based on boolean evidences for matching, i.e. *similarity evidences*, we apply another type of *boolean functions* to filter out those that are non-matches. Since they are based on evidences for non-matching, i.e. *dissimilarity evidences*, we refer to this type as *dissimilarity function*. As an example, the fact that two given products are produced by different manufacturer can be taken as a dissimilarity evidence to infer that they are non-matches. Applying this type of functions to the results is referred to as the *filtering* step:

Definition 2.7 (Filtering)**.** *Given a matching candidate set M,* filtering *returns a subset* $M^+ = M \setminus M^-$*, where* M^- *represents non-matching candidates computed by the dissimilarity function* $\neg same$*, i.e.* $M^- = \{(n, n') \in M | \neg same(n[a_i], n'[a_i]) = 1\}$.

There are several works related to the problem of filtering [10, 37, 38, 122]. Doan et al. [37, 38] discussed constraints to apply sanity checks for instance matching. Bhattacharya [10] exploited negative relational evidences to improve the accuracy of the results. Whang et al. [122] introduced binary negative rules, which require a global view of records for detecting inconsistencies. Being different from these works that need experts to manually design the constraints for non-matches, the problem here is to automatically learn the dissimilarity functions for filtering.

2.7 Metrics

We use standard metrics proposed for evaluating blocking and instance matching results. Let S be the result generated by blocking or instance matching approaches, M be the matching pairs, i.e. the ground truth GT, and D the non-matching pairs in the dataset, i.e. all possible pairs of instances but the ones in GT. To evaluate the blocking approaches, pair-wise completness (PC, which is the same to the metric recall R that is used for evaluating instance matching approaches) captures the ratio between correct matches as captured in the blocks and all matches in the ground truth, i.e. $PC = \frac{|S \cap M|}{|M|}$. Reduction ratio ($RR$) measures the relative reduction in the

size of the comparison space accomplished by blocking, i.e. $RR = 1 - \frac{|S|}{|M \cup D|}$. To evaluate the instance matching approaches, precision (P) captures the ratio between correct matches and all generated matches, i.e. $P = \frac{|S \cap M|}{|S|}$. F-measure ($F$) considers both P and R, calculated as $F = \frac{2PR}{P+R}$.

Chapter 3

Typification: Inferring the Type Semantics of Structured Data

Structured data representing instance descriptions often lacks precise type information. That is, it is not known to which type an instance belongs to, or the type is too general to be useful. In this work, we propose to deal with this novel problem of inferring the type semantics of structured data, called typification. We formulate it as a clustering problem and discuss the features needed to obtain several solutions based on existing clustering solutions. Because schema features perform best, but are not abundantly available, we propose an approach to automatically derive them from data. Optimized for the use of schema features, we present TYPifier, a novel clustering algorithm that in experiments, yields better typification results than the baseline clustering solutions.

Outline. The introduction and the contributions are presented in Sec. 3.1 and Sec. 3.2. In Sec. 3.3, we introduce typification as a clustering problem. In Sec. 3.4, we discuss how techniques proposed for that can be applied. Then, we continue with the discussion of how value-level schema features can be computed from data in Sec. 3.5. Sec. 3.6 reports our algorithm for typification. Experimental results are presented in Sec. 3.7 before related work in Sec. 3.8, and conclusion in Sec. 3.9.

3.1 Introduction

The semantics captured by structured data have been exploited in various tasks such as Web search: RDFa and mircoformats are used by Google and Yahoo! to provide rich snippets [1] for Web search results, which are based on structured descriptions of the instances embedded in the results. Especially the semantics of instance *types* provide valuable information here as they enable Web designers to construct presentation templates for specific types of instances (e.g. customized snippet templates). Type information is also useful for machine learning algorithms. One concrete problem, which is used as a representative use case in this thesis, is the one of finding

[1] http://www.google.com/webmasters/tools/richsnippets

structured descriptions that refer to the same real-world entity, also known as instance resolution. State-of-the-art solutions learn a combination of matching predicates, which are used to compute the similarity between instance descriptions. They work well when applied to instances from a single table, i.e. instances of the same or similar type. These type-specific solutions however do not perform well, when *type information is difficult to be inferred*. We have identified two main scenarios where this is the case:

Dynamic Web Data. The proliferation of RDF data on the Web is mainly due to the flexibility of the RDF model. Without imposing constraints on the schema and data integrity, it makes it easy for data providers to assert, to publish, and to link triples to other datasets. While it seems to be well suited for dealing with the dynamics on the Web, the downside is that many structured descriptions available on the Web today do not contain type triples. Based on a 3 million sample of instance descriptions extracted from several Linked Data datasets crawled from the Web, we found that 393,503 of them lack type information, i.e. the type of these instances are not known.

Heterogeneous Enterprise Data. In enterprises, instance types mostly correspond to tables in which tuples representing instance descriptions are stored. However, big data tables are often used to store a variety of instance descriptions. The type captured by a table is often too general to be useful. For instance, in a data integration project with a product management company[2], we face the situation where 7M descriptions acquired from different sources are stored in one single product table. The products captured by this data actually belong to many different subtypes.

3.2 Research Question and Contributions

Given the need to tackle the heterogeneity of Structured data on the Web, the main research question of this chapter is:

Research Question 1. *How to derive type semantics of instances?*

To address this research question, we propose an clustering algorithm, named *Typifier*, in this chapter and examine it with respect to the following hypothesis:

Hypothesis 1.1. *Although the concrete values are quite different, instances of the same type are often associated with the same labels of attributes and relations. The type semantics of structured data can be inferred by clustering instances that have same or similar labels of attributes and relations.*

We address the above research question and hypotheses and perform the study towards a principled solution, providing three main contributions in this chapter:

[2] http://www.reposito.com/

- **Clustering Solutions for Typification.** Firstly, we formulate Typification as a clustering problem, where the goal is to identify a particular kind of clusters that represent the types of instances. We discuss the main requirements, quality metrics and how existing clustering techniques can be applied. We present the experimental results using data from domain-specific product data in the enterprise to cross-domain encyclopedic data in DBpedia up to heterogeneous Linked Data on the Web.

- **Computing value-level Schema Features.** We discuss why for this particular clustering problem, schema-level features, i.e. labels of attributes and relations, are most effective in obtaining high quality results. The main intuition here is while instances of the same type are often associated with the same attributes and relations, their concrete values are quite different. However, the amount of these features is often limited, especially in the one large table setting (e.g. instances in a product table are associated with only few attributes). To deal with this, we also propose a solution for automatically computing value-level schema features from data.

- **Schema-based Typification.** Optimized for the use of (value-level)schema features, we propose TYPifier, a novel clustering algorithm for the typification problem. Compared to the baselines based on existing clustering techniques that use the same (schema) features, this algorithm is comparable to the best baseline in terms of efficiency, but produces higher quality results than the baseline that performed best in terms of effectiveness. Moreover, it is able to determine the number of types (clusters) automatically.

3.3 Overview

In this chapter, we use the definitions of data and its example that was first given in Section 2.1. For ease of reading, we repeat the definition of Typification here.

Definition 2.3 (Typification). *Given a set of instances $N = \{n_1, \ldots, n_j, \ldots, n_{|N|}\}$, typification attempts to seek a K-*partition *of N ($K \leq |N|$) into* clusters $\mathbf{C} = \{C_1, \ldots, C_K\}$*, such that (1) $C_i \neq \emptyset, i = 1, \ldots, K$, (2) $\bigcup_{i=1}^{K} C_i = N$ and (3) $C_i \cap C_j = \emptyset, i, j = 1, \ldots, K$ and $i \neq j$. Moreover, when missing subtype edges are also considered, it seeks* hierarchical clusters*, i.e., a tree-like nested structure partition of N (called* hierarchy tree*),* $\mathbf{H} = \{H_1, \ldots, H_Q\}$ *($Q \leq N$), s.t. $C_i \in H_m$, $C_j \in H_l$ and $m > l$ imply $C_i \subseteq C_j$ or $C_i \cap C_j = \emptyset$ for all $i, j \neq i, m, l = 1, \ldots, Q$.*

Quality of Computed Types. The goal here is to obtain clusters that correspond to missing classes in the data. Just like in the classification setting, class labels are assumed to exist for all instances in the evaluation dataset. However, the results obtained from typification are not class labels but clusters. Thus, we need to adopt

existing metrics to compare the given class extensions with the computed clusters, instead of using the given and predicted class labels. For assessing the quality of typification results, we employ the method used for the evaluation of hierarchical clustering algorithms [129]:

Let the computed clusters be $\mathbf{C} = \{C_1, \ldots, C_i, \ldots, C_K\}$, and type labels be given for all instances in N such that for every class $u_c \in U_C$, we know its *extension*, $E(u_c) = \{n : type(n, u_c) \in E\}$, as the set of all instances that have type of u_c. Further, let $\mu : \mathbf{C} \rightarrow U_C$ be the function which maps a cluster to a corresponding class. Then, for a cluster C and the extension of its corresponding class $E(\mu(C))$,

- *true positives* (tp) $= |C \cap E(\mu(C))|$,
- *false positives* (fp) $= |C| - |C \cap E(\mu(C))|$,
- *false negatives* (fn) $= |E(\mu(C))| - |C \cap E(\mu(C))|$.

Based on these metrics, precision and recall for every cluster C can then be computed as usual, namely $Precision = \frac{tp}{tp+fp}$ and $Recall = \frac{tp}{tp+fn}$. The overall score, i.e., precision or recall, of the entire clustering solution is the sum of the individual clusters scores weighted according to the cluster size. The corresponding class $\mu(C)$ of a cluster C is chosen to be the one that maximizes the score of C. Given a cluster, this best matching class is found through an exhaustive search over the set N_C.

Quality of Type Hierarchy. While these metrics capture the quality of individual clusters, they do not consider the quality of the computed hierarchy. To do that, we compute the distance between the computed cluster hierarchy tree $\mathbf{H}$ and the given class hierarchy U_C. In particular, we use the *unordered tree edit distance* [17] as a metric, which is a generalization of the string edit distance: $\mathbf{H}$ and U_C are treated as node-labeled trees and the quality of $\mathbf{H}$ w.r.t. U_C is defined as the number of edit operations to transform $\mathbf{H}$ to U_C.

Main Requirements. Besides *quality* aspects, *performance* is another criteria for designing a solution for typification. It is desirable to compute not only the missing types but also the type hierarchy. Furthermore, a solution shall be *parameter-free* in the sense that K, the number of types, can be automatically determined to match exactly the number of classes in the data.

3.4 Clustering Solutions

A clustering solution aims to maximize intra-cluster homogeneity and inter-cluster separation such that elements in the same cluster are similar, while those in different clusters are not. We refer to the survey from Xu and Wunsch for an overview of the different types of clustering *techniques* and *similarity measures* that have been proposed [126].

3.4.1 Features and Similarities

Applied to this problem, the elements to be clustered are the instances $n \in N$, which are represented by the feature function $F(n)$. Instances in the same cluster should belong to the same type and those in different clusters should not.

Naturally, attribute values of instances can be used as features to group them according to a similarity metric in this feature space. We observe that when instances have similar values, they mostly do belong to the same type (or even denote the same real-word object). However, instances may also belong to the same type when they have very different values. More reliable for this problem are schema features: instances that have the same edge labels (attributes or relations) belong to the same type and those, which have no or only few such labels in common, do not.

Thus, for every instance n_i, labels of associated nodes and edges are extracted to obtain two types of features:

- *data feature* $F_D(n_i) = \{l(n) : \langle n_i, l, n \rangle \in E_A\}$, where $l(n)$ denotes the label of n, and
- *schema feature* $F_S(n_i) = \{l : \langle n_i, l, n \rangle \in E\}$

While more customized similarity functions can be used for different types of values, in the experiments, we only use *cosine similarity* as the similarity function, because the schema and value-level schema features are derived from the data that is mainly composed of text. Let V be the vocabulary of all possible features that can be extracted from data and schema labels. An instance n_i is represented as a vector $F(n_i) = \{f_1, \ldots, f_i, \ldots, f_{|V|}\}$ over the features in V. The similarity between two instances n_i and n_j is the cosine of the angle between $F(n_i)$ and $F(n_j)$.

Example 3.1. *Taking the instance n_5 in the data graph that is shown in Fig. 2.2 as example. The schema features of n_5 are the labels of its attributes, such as* `Title`, `CPU`, `Memory` *and* `Power`. *Therefore, n_5 is represented as a vector of all the tokens in the schema features, e.g. $F(n_5) = \{\texttt{Title,CPU,Memory,Power} \cdots\}$. On the other hand, the data features of n_5 are the values of its attributes, such as* `Sony`, `Intel i3` *and* `65 W`. *Then n_5 can be represented as a vector of all the tokens in the data features, e.g. $F(n_5) = \{\texttt{Sony,Intel,i3,W,65} \cdots\}$.*

3.4.2 Techniques

One main distinction that can be made is *hierarchical* and *partitional* clustering.

Hierarchical. This type fits our problem well because it yields a tree of clusters, where the root contains all instances, while the leaves represent single instances. Hierarchical agglomerative methods start from single instance clusters and iteratively merge them. Divisive methods proceed in a top-down fashion, splitting the root

cluster and continue until all clusters contain only one instance. For the merge and split operations, a similarity metric for clusters is needed. Typically, cluster similarity is defined based on the similarity between particular elements in the clusters, e.g. single-linkage (minimum similarity between elements), complete-linkage (maximum similarity), average-linkage clustering (average similarity). The main problem of applying hierarchical clustering to typification is the *identification of clusters representing the types*. The evaluation method proposed previously searches through the classes and selects the class that maximize the score. The resulting score does not reflect the problem that in practice, this has to be done manually. The other shortcoming is *performance*. The computational complexity of most algorithms is at least $O(M^2)$, where M is the number of input instances. Aiming at this and other problems such as sensitivity to noise and outliers, a state-of-the-art approach called *BIRCH* [128], utilizes a summary of the data called clustering feature tree, with each node storing clustering information of the data such as the number or linear sum of some sets of data points. Complexity of clustering, when applied to the summary, is reduced to $O(M)$.

Partitional. Instead of splitting or merging to form a hierarchy, this type only considers the problem of finding the optimal assignment of data points to K clusters, where the optimality criterion is often the sum of squared error. A popular algorithm is K-means, which starts from a random choice of K centers, then iteratively assigns data points to the nearest cluster, updates the centers, and terminates when the criterion converges. The time complexity of K-means is $O(MKI)$ where I is number of iterations. Since K and I are usually much smaller than N, K-means is fast and widely used in practice for dealing with large datasets. Based on a strategy for choosing the starting centers, *K-means++* is an improved algorithm that exhibits both high runtime performance and accuracy [5]. While this type of algorithms is fast and easy to implement, the obvious problem is the choice of K, which has to be done manually.

Kernel-based. Using kernels, nonlinearly separable instances can be transformed into a high-dimensional feature space, where linear separation is possible. This idea has been combined with K-means to obtain kernel K-means algorithms that compute similarities between data points in the high-dimensional space. A different type of kernel-based algorithms, which is K- parameter-free in the sense discussed before, is Support Vector Clustering (*SVC*) [9]. In the high-dimensional space obtained via a Gaussian kernel, it searches for the minimal enclosing sphere. When mapped back to the data space, this sphere reveals contours corresponding to clusters of points. The drawback of this algorithm is high complexity, which is at least $O(M^2)$.

Density-based. This is another type that also does not directly require the manual tuning of K. It considers clusters as regions in the data space in which the objects are dense and separated by sparse regions treated as cluster borders or noises. One typical approach is DBSCAN, which requires that, for a point to be part of a cluster, the neighborhood of a given radius has to contain at least a minimum number of points. *OPTICS* extends DBSCAN by removing the need for an assigned density

value and is able to generate hierarchical clusters [1]. The main problems here are complexity, which is $O(M^3)$, and recall, because relevant instances and classes might be hidden in the sparse regions.

TYPifier. We perform experimental studies with BIRCH, K-means++, SVC and OPTICS using both data and (value-level)schema features automatically derived from real-world datasets. Based on the same features, we compare them with TYPifer, an approach we propose to exploit the characteristics of schema features. It can be seen as a *divisive hierarchical clustering algorithm*, which proceeds top-down starting from the root node. However, schema features are used not only to compute similarities between instances and clusters to perform a split, but also to obtain an ordering over atomic clusters so that *several other operations are possible*, namely to merge clusters and to add clusters as child or sibling nodes. TYPifier is a K-parameter-free approach, which provides similar performance but superior result quality compared to the best results achieved by the baseline clustering approaches.

3.5 Value-level schema Features

Features characterize the membership of instances to a type well when they are (1) shared by most instances of that type and (2) not in the feature sets of other instances that belong to other types. In this sense, schema features are better type indicators than data values. As an example, for instances captured by Linked Data that actually belong to the type `maps`, the information that is missing, we find attributes such as `coverage` with values such as `USA`. For typification, `coverage` is more specific to instances of the type `maps`, while `USA` appears also as values of attributes of many other types. While in the experiments, these schema features work well for Linked Data, whereas their availability and abundance cannot be assumed in the general case. We discuss how to derive value-level schema features, i.e. those that behave like schema features, from the data.

TF-IDF. This is a popular way to measure the importance of a term t for a document d relative to other documents in the corpus D. TF measures whether the term is frequent in d while IDF indicates whether t is common or rare in D. Let $F(n)$ be the bag of all features that can be extracted from attribute values of an instance n (note we overload notation here and use bag instead of set to consider counts). Adopted to this case,

$$TF - IDF(f,n) = |\{f \in F(n)\}| \times log \frac{|N|}{|\{n_i \in N : f \in F(n_i)\}|}$$

where $|\{f \in F(n)\}|$ is the term frequency that is calculated by the raw frequency of a feature, i.e. the number of times that feature f occurs in $F(n)$, and $log \frac{|N|}{|\{n_i \in N: f \in F(n_i)\}|}$

is the inverse document frequency, in which $|N|$ is the total number of instances, and $|\{n_i \in N : f \in F(n_i)\}|$ is the number of instaces where the feature f appears.

Feature Co-Occurrence Graph. The feature with high TF-IDF score characterizes an instance but not necessarily the type it belongs to. We propose a way to consider the importance of features across instances using a feature co-occurrence graph (FCG). A feature co-occurs with another when there is an instance feature set that contains both. Based on the feature co-occurrence count, we then define the conditional co-occurrence probability (CCP):

Definition 3.1 (Conditional Co-Occ. Probability)**.** *The probability that f_i and f_j co-occur, given f_j is*

$$p(f_i|f_j) = p(f_i, f_j)/p(f_j) = count(f_i, f_j)/|N^{f_j}|,$$

where $N^{f_j} = \{n \in N : f_j \in F(n)\}$ denotes all instances having f_j as feature, and

$$count(f_i, f_j) = |N^{f_i} \cap N^{f_j}|,$$

the co-occurrence count *of the two features f_i and f_j.*

These co-occurrence relations between features are finally used to construct a graph:

Definition 3.2 (Feature Co-Occurrence Graph)**.** *Given the vocabulary of features V, the* feature co-occurrence graph *is a labeled directed graph $G' = (U', E', L')$, with nodes $u \in U'$ stand for features in V and edges $w(u_i, u_j) \in E'$ capture the co-occurrence between the two features f_i and f_j, where edge labels stand for the conditional co-occurrence probabilities, $w = p(f_i|f_j)$.*

Example 3.2. *The FCG according to the example in Tab. 2.1 is shown in Fig. 3.1. For example, because the features* `CPU` *and* `W` *co-occur, there are corresponding edges $w(CPU, W)$ and $w(W, CPU)$ in the graph. Note $count(CPU, w) = |\{n_4, n_5, n_6\}| = 3$ and there are six instances that have* `W` *as feature. Thus, we have $0.5(CPU, W)$ because $p(CPU|W) = count(CPU, W)/|N^W| = 0.5$. In the same way, we obtain $1.00(W, CPU)$ (and other edge weights not shown in the figure).*

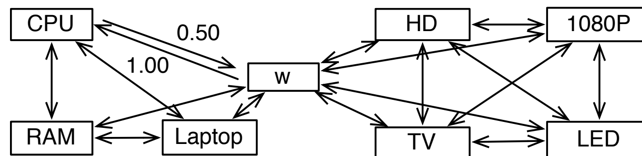


Figure 3.1: FCG for data in Tab. 2.1.

Algorithm 1: Computing Value-level Schema Features

```
Input: G(U,E,L),V.
Data: Undirected FCG G'(U',E',L').
Result: F*_S.
1 foreach f_i ∈ V do
2     foreach f_j ∈ V do
3         p(f_i|f_j) := count(f_i,f_j)/|N^{f_j}|;
4         //Construct undirected edges.
5         if p(f_i|f_j) ≥ θ ∧ p(f_j|f_i) ≥ θ then
6             G' := G' ∪ θ(f_i,f_j);
7 //Extract features from maximal cliques.
8 F*_S :=Features(MaxCliques(G'));
9 return F*_S;
```

Note that compared to TF-IDF, CCP is a frequency-based metric for importance that is computed across instances. We propose to identify features that are important for groups of instances by searching for maximal cliques in the FCG, i.e. clusters of features that pairwise, highly co-occur as measured in terms of CCP. We found out in the experiments that while these clusters do not directly correspond to types of instances, because a number of overlaped clusters usually refer to the same type. However, these clusters capture discriminative features that yield better results than the TF-IDF method.

Algorithm. Computing value-level schema features can be performed in two steps, namely computing G' from G (*FCG* step) and extracting maximal cliques from G' (*MaxCliques* step) as shown in Alg. 1. For computing G', it firstly calculates the probabilities $p(f_i|f_j)$, which requires a maximum of $|N| \times |N|$ steps (simple counting) for computing co-occurrence of f_i and f_j (because $|N|$ is the maximum number of instances that can have f_i or f_j). This has to be performed for all possible feature pairs, a number bounded by $|V| \times |V|$. An undirected edge $\theta(f_i, f_j)$ is created for G' only when both $p(f_i|f_j)$ and $p(f_j|f_i)$ are greater then the given parameter θ. Note that this is not the same as using the joint probability $p(f_i, f_j)$, which is often high when either $p(f_i)$ or $p(f_j)$ is high. This two-ways conditional probability gives a smoother estimate of co-occurrence.

Then, an existing maximal cliques algorithm [91] is applied to this undirected version of the FCG. While MaxCliques is known to be NP-hard, optimized algorithms such as this one can provide fast runtime performance. The amount of edges in the FCG is bounded by $|V|^2$. The size of V is reduced through text processing techniques such as stop words elimination. Further, a large amount of feature pairs is pruned

during the construction of the FCG, especially when θ is high.

Finally, only features corresponding to nodes captured by the resulting maximal cliques are kept.

3.6 TYPifier

Existing clustering solutions represent instances as sets of features, e.g. those remaining in V considered as value-level schema features. Then, clusters are treated as *sets of instances*. Accordingly, cluster similarity (hence cluster operations such as merge) is determined based on the similarity between the instance feature sets. We depart from this view to model clusters directly as *sets of (value-level)schema features*, and compare these feature sets to determine hierarchical relations (including similarity) between clusters.

Definition 3.3 (Cluster). *A cluster is a tuple $C(F, N, S)$, with $F = \{f_1, \ldots, f_i\}$ being the set of features (f_i is an element in the given set of all (value-level)schema features V, note we overload notation here to use V as either value-level schema or schema features), N the set of all instances that have an element in F as feature, i.e. $N = \{n : f \in F(n), f \in F\}$, and S the set of clusters that are either child or descendant nodes of C (representing subtypes of the type captured by C) in the hierarchy tree.*

Example 3.3. *Taking the data in Tab. 2.1 as an example. Assuming the schema features set is $F = \{$`Platform`,`Media`$\}$, then we have $N = \{n_8, n_9, n_{10}, n_{11}\}$. Obviously, the cluster corresponds to F and N refers to the type of* `Software` *that have two subtypes* `Language Learning` *and* `Maps`*. Therefore, we can construct a cluster $C(F, N, S)$, where S includes two child nodes as the clusters corresponding to the types* `Language Learning` *and* `Maps`*.*

We firstly discuss how to compute relations, and then show how to obtain the (value-level)schema feature representation of clusters in the presentation of the algorithm.

3.6.1 Clusters and Cluster Relations

Relations between clusters are introduced to infer the type hierarchy. Given two clusters C_i and C_j, C_i is either

- a *parent* (*ancestor*) of C_j, denoted as $C_i > C_j$ ($C_i \gg C_j$), or
- a *child* (*descendant*) of C_j, $C_i < C_j$ ($C_i \ll C_j$), or
- C_i and C_j represent the *same* cluster, $C_i = C_j$, or

Table 3.1: Cluster relations based on distance.

Relation	Distance
$C_i > (\gg) C_j$	$d(C_i, C_j) > \epsilon$ and $d(C_j, C_i) < \epsilon$
$C_i < (\ll) C_j$	$d(C_i, C_j) < \epsilon$ and $d(C_j, C_i) > \epsilon$
$C_i = C_j$	$d(C_i, C_j) > \epsilon$ and $d(C_j, C_i) > \epsilon$
$C_i \neq C_j$	$d(C_i, C_j) < \epsilon$ and $d(C_j, C_i) < \epsilon$

- there is *no relation* between C_i and C_j, $C_i \neq C_j$.

Previously, we discussed how to use *feature co-occurrence* to determine the ones that are characteristic for type clusters. To compute cluster relations from the data, we leverage a similar intuition: we use pairwise *cluster co-occurrence*, i.e. *feature sets co-occurrence*, to determine the distance between clusters, and based on that, their relations.

Definition 3.4 (Cluster Distance)**.** *Let $count(f_i, f_j)$ be the co-occurrence count of f_i and f_j and $|N_E^f|$ be the count of instances having f as feature, the distance $d(C_i, C_j)$ between the clusters C_i and C_j is the conditional probability of co-occurrence of its feature sets F_i and F_j,*

$$d(C_i, C_j) = p(F_i|F_j) = \frac{\sum_{f_i \in F_i, f_j \in F_j} count(f_i, f_j)}{\sum_{f \in F_j} |N^f|} = \frac{|N_i \cap N_j|}{|N_j|}, \tag{3.1}$$

where N_i (N_j) is the instance set associated with C_i (C_j).

Two clusters are considered related when their *feature sets F_i and F_j co-occur*. Co-occurrence here intuitively means that there are some instances, which have at least one element in both F_i and F_j as feature. Equivalently, this means there exist some instances that are in the intersection set of N_i and N_j. Thus, in other words, clusters are related when their *instance sets N_i and N_j overlap*. Further, the conditional co-occurrence probability $p(F_i|F_j)$ captures the likelihood some instances to have also some features in F_i when they have some features in F_j. Because a subtype inherits the features of its parent and ancestors, a high value for $p(F_i|F_j)$ ($p(F_j|F_i)$) can be seen as an evidence that the corresponding cluster C_i is a parent or ancestor (child or descendant) of C_j. Likewise, a low value for $p(F_i|F_j)$ ($p(F_j|F_i)$) can be interpreted as a counter-evidence for that parent or ancestor (child or descendant) relation.

Given the evidences that C_i is (1) a parent or ancestor and (2) not a child or descendant of C_j, the relations $C_i > (\gg) C_j$ is derived. Along the same line, other relations are determined from this probability-based distance metric as shown in Tab. 3.1 (ϵ is a given threshold parameter).

Example 3.4. *Taking the data in Tab. 2.1 as an example. Assuming two clusters* C_1 *and* C_2 *that correspond to the schema feature sets* $F_1 = \{\texttt{Platform}, \texttt{Media}\}$ *and* $F_2 = \{\texttt{Language}\}$*, and the instance sets* $N_1 = \{n_8, n_9, n_{10}, n_{11}\}$ *and* $N_2 = \{N_{10}, N_{11}\}$*, respectively. According to Def. 3.4, we can calculate distances* $d(C_1, C_2) = 1.0$ *and* $d(C_2, C_1) = 0.5$*. Then if we set* $\epsilon = 0.6$*, we can get the relation* $C_1 > (\gg) C_2$ *(or* $C_2 < (\ll) C_1$*) according to Tab. 3.1.*

3.6.2 Relation-based Hierarchical Clustering

The relations between clusters decide their positions in the hierarchy tree. Exploiting this, we propose a top-down hierarchical clustering procedure that in a depth-first search (DFS) manner, constructs the tree starting from the *root* node. The root node represents a virtual cluster that is associated with all features and comprises all instances. In every iteration, a node from a fixed set of *atomic clusters* are merged with or added as a child to the current node. An atomic cluster is constructed for every feature $f \in V$ such that the number of all atomic clusters is $|V|$. The procedure simply terminates when all atomic clusters have been assigned their positions in the tree. In this way, clusters and the hierarchy tree naturally form without relying on the parameter K.

Algorithm Overview. The algorithm implementing this DFS procedure is shown in Alg. 2. Firstly, the *root* node and atomic clusters $\mathbf{C}$ are initialized. Then, from every child node of *root*, it performs DFS using the recursive `formTree` subprocedure. It constructs a branch by merging and adding atomic nodes until no further child and descendant nodes can be found for that branch. The set of input nodes $\mathbf{C}$ is pruned to keep only those that are child or descendant of, or represent the same type as the current *root*, denoted as S^*_{root}. In every iteration, (1) a cluster C is removed from S^*_{root} and *merged* with or *added* as a child to the current *root*. The following additional steps are needed in the case of addition: (2) *splitting the clusters* into those that are relevant for branching from root, S^*_{root}, and those that are relevant for its siblings, $\mathbf{C} \setminus S^*_{root}$, and (3) *splitting the instances* into those that can be associated with C and those that belong to its siblings.

Add or Merge. In every iteration, we use the following relations to focus on those atomic nodes that are relevant for the current *root*:

Corollary 3.1. *Let root be a node in the cluster hierarchy. A node C is a child or a descendant of, or a node that represents the same type as root if* $C \leq root$*.*

Corollary 3.2. *Given* S^*_{root}*, the set of all possible child and descendant nodes of root,* $C \in S^*_{root}$ *is a child of root if* $C < root$ *and* $C \geq C_i$ *for all* $C_i \in S^*_{root}$*.*

Because all elements $C \leq root$ are included, Corollary 3.1 guarantees that S^*_{root} captures all candidates needed to be considered for branching from *root*. Further, because S^*_{root} is sorted, it can be derived from Corollary 3.2 that the top element must

Algorithm 2: Typify

```
   Input: N_E, V, maxDepth.
   Result: root
1  //Initialize root node.
2  root := (V, N, ∅);
3  C := ∅;
4  foreach f ∈ V do
5  |  //Construct a cluster for every feature.
6  |  C_f := (f, N^f, ∅);
7  |_ C := C ∪ C_f;
8  formTree (root, C, 1 maxDepth);
9  return root;
```

represent the same type or a *child* of C because there are no other elements in S^*_{root} that is a parent of C:

Theorem 3.1. *If S^*_{root} is sorted according to the distance to root, i.e. for the top element $C \in S^*_{root}$ it must hold that $d(C, root) \geq d(C_i, root)$ for all $C_i \in S^*_{root}$, then $C \geq C_i$ for all $C_i \in S^*_{root}$.*

Proof. We have to show there is no element in S^*_{root} that is a parent of C, i.e. there is no $C_i \in S^*_{root}$, where $d(C_i, C) > \epsilon$ and $d(C, C_i) < \epsilon$.

Since S^*_{root} is sorted, we have $|C| \geq |C_i|$. Hence, we obtain $count(C, C_i)/|C_i| \geq count(C, C_i)/|C|$ and $d(C, C_i) \geq d(C_i, C)$, respectively. From this follows that when $d(C_i, C) > \epsilon$, then it must hold that $d(C, C_i) \geq d(C_i, C) > \epsilon$. Thus when $d(C_i, C) > \epsilon$, $d(C, C_i) < \epsilon$ does not hold. □

Theorem 3.1 indicates that the top element C of S^*_{root} is either of the same type as the *root* or the direct child of the *root*. This result enables the reconstruction of the hierarchy tree based on two operations, namely to add C as child to *root* or to merge the two clusters. To perform the merge, i.e. $C = root$, the feature and instance sets of C are combined with the corresponding feature and instance sets of *root*. The subtypes of the resulting cluster, S_{root}, do not change because C is an atomic cluster. When $C < root$, C is a child and thus, can be added to S_{root} (which as opposed to S^*_{root}, does not contain elements representing the same type as *root*).

Splitting Instances. When performing this addition, elements from C, N_C, are removed from N_{root}, and later, added back to N_{root}. Finally, the instance set of a cluster *root* is computed as the union of the instance sets of all its children and descendants in S_{root}. Note that in the while clause, `formTree` keeps trying to add siblings of C to *root*. The removal of N_C from N_{root} is necessary to ensure that instances already

Algorithm 3: FormTree

Input: $root$, $\mathbf{C}$, d, $maxDepth$

Result: $\mathbf{C}$.

1 //Initialize the subtypes of root.
2 $S^*_{root} := \emptyset$;
3 **foreach** $C \in \mathbf{C}$ **do**
4   **if** $C \leq root$ **then**
5   //Keep only candidate subtypes.
6     $S^*_{root} := S^*_{root} \cup C$;
7     $\mathbf{C} := \mathbf{C} \setminus \{C\}$;
8 //If the recursion depth exceeds the $maxDepth$, the recursion is terminated
9 **if** $d > maxDepth$ **then**
10   **return C**
11 //While more children can be added to root.
12 **while** $S^*_{root} \neq \emptyset \wedge N_{root} \neq \emptyset$ **do**
13   //Make sure next element is either same as or is CHILD of root.
14   sort $C \in S^*_{root}$ by $d(C, root)$;
15   C :=pop (S^*_{root});
16   //Merge.
17   **if** $C = root$ **then**
18     $F_{root} := F_{root} \cup F_C$;
19     $N_{root} := N_{root} \cup N_C$;
20   //Add as child.
21   **if** $C < root$ **then**
22   //Construct tree for every child of C.
23   S^*_{root} :=formTree ($C, S^*_{root}, d+1, maxDepth$);
24     $S_{root} := S_{root} \cup C$;
25   //Make sure siblings of C capture only instances not already covered by C.
26     $N_{root} := N_{root} \setminus N_C$;
27 **foreach** $C \in S_{root}$ **do**
28   //Construct instance set of C as union of instance sets of its children/descendants.
29   $N_{root} := N_{root} \cup N_C$;
30 //Return clusters that are no subtype of root.
31 **return C**;

covered by C no longer have an effect on the construction of siblings of C. In particular, when sorting candidate siblings according to distance to *root*, only the instances $N_{root} = N_{root} \setminus N_C$ are employed for the distance computation.

Splitting Clusters. S^*_{root} serves as input for subsequent branching from C, which becomes the *root* in the next iteration. Again, S^*_{root} will be further refined to keep only candidates for the new root C. On the other hand, $\mathbf{C} \setminus S^*_{root}$ (the result of `formTree`) serves as input for branching from siblings of C, i.e. S^*_{root} :=`formTree`(C,S^*_{root}). The `formTree` procedure keeps trying to find siblings of C and add them as children to *root* while the candidate set of clusters S^*_{root} (and instances N_{root}) are not empty.

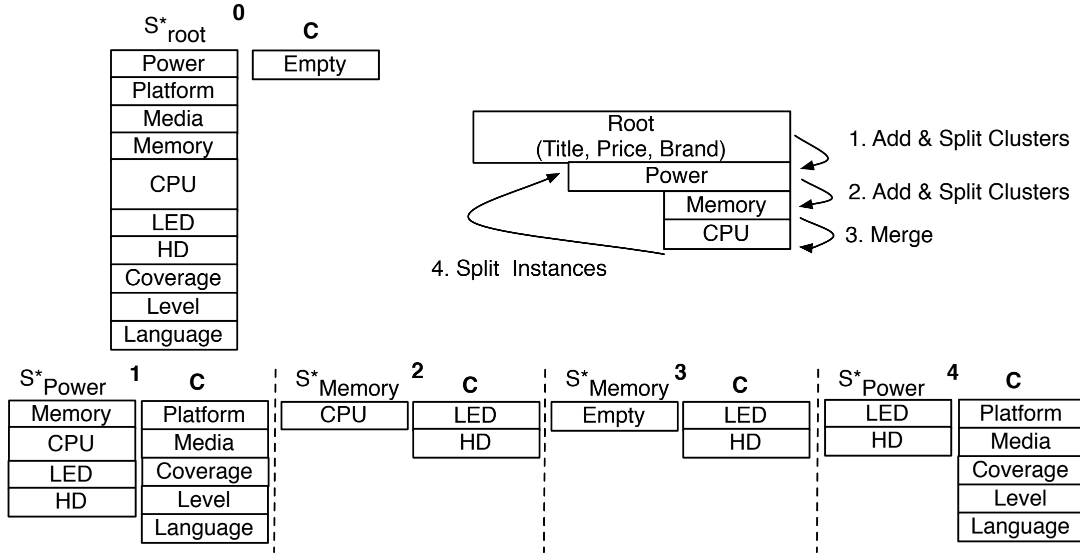


Figure 3.2: Algorithm.

Example 3.5. *Fig. 3.2 illustrates four representative steps of the algorithm. Assume* `Title`, `Price` *and* `Brand` *have already been merged to form the cluster* `Product` *as* `Root`. *Since all the other atomic clusters are children or descendants of this type, they are removed from* $\mathbf{C}$ *and stored in* S^*_{root} *(step 0). Then we have the following steps:*

1. Add and split clusters. `Power` *is at the top of* S^*_{root}, *thus it is processed first. Because* `Power<Root`, *it is added as a child to* `Root`. *Then, the algorithm continues to construct the hierarchy for* `Power` *through a new recursion of* `FormTree` *with* S^*_{root} *as input. Children and descendants of* `Power` *are removed from* $\mathbf{C}$ *to form* S^*_{Power}. *Now,* $\mathbf{C}$ *only contains children or descendants of the siblings of* `Power`.

2. Add and split clusters. Now, `Memory` *is child of* `Power`, *which is added and split once again.* `FormTree` *goes into a new recursion to form the hierarchy for* `Memory`, *where* S^*_{Memory} *only contains* `CPU` *as child and* $\mathbf{C}$ *contains* `LED` *and* `HD` *as siblings of* `Memory`.

3. Merge. `CPU` *and* `Memory` *are merged. This does not result in a new recursion.*

4. Splitting instances. When S^*_{Memory} *is empty, the recursive construction of the hierarchy for* `Memory` *ends. The algorithm goes back to the hierarchy construction of* `Power`. $N_{Memory} \cup N_{CPU}$ *has to be removed from* N_{Power} *to ensure that other children of* `Power` *have no instance overlap with* `CPU` *and* `Memory`. `Memory` *and* `CPU` *have already been removed from* S^*_{Power}*, so it now contains only* `LED` *and* `HD`*. Then, the algorithm proceed to form hierarchies for another child in* S^*_{Power}.

Termination. Clustering terminates when (1) S^*_{root} or (2) N_{root} is empty, i.e. there are no further clusters or instances in clusters to be processed. Further, (3) it stops when the top cluster C has no relation with the root, i.e. $C \neq root$ (this relation implies that neither merging, $C = root$, nor adding it as a child, $C < root$, is possible). When this top cluster has no relation with the root, then it follows (because S^*_{root} is sorted by the distance to the *root*) that all the other clusters in S^*_{root} must also have no relation with the root (i.e. $C_i \neq root, \forall C_i \in S^*_{root}$). In other words, no elements in S^*_{root} require further processing (thus, this has the same effect as the first termination condition, $S^*_{root} = \emptyset$). While it is not necessary for termination (which is ensured by the conditions discussed before), a parameter *maxDepth* can be optionally employed to control the depth of the hierarchy. (4) The algorithm also terminates when $d > maxDepth$.

Complexity. The time complexity of this procedure depends on the number of operations needed for addition, merge, and splitting clusters as well as for splitting instances. For every atomic cluster, every such operation is applied at most once. Since an atomic cluster is generated for every feature, there is a total of $|V|$ atomic clusters. With addition or merge, elements in S^*_{root} have to be sorted, which has as complexity $O(|V|\log|V|)$ (because $S^*_{root} = V$ in worst case). For splitting clusters, we iterate through all clusters in $\mathbf{C}$ ($\mathbf{C} = V$ in worst case) to find the children or descendants of *root*, hence we have $O(|V|)$. For splitting instances, all instances in S_{root} are added back and then removed from N_{root}. Because $S_{root} = V$ in worst case, complexity of this operation is the same as splitting clusters, $O(|V|)$. Because all these operations add or merge, splitting clusters and splitting instances may have to be performed for every atomic cluster, total complexity is $O(|V|^2\log|V| + |V|^2 + |V|^2)$. Note that while the complexity of the proposed algorithm is dependent on $|V|$, the amount of (value-level)schema features, the complexity of existing clustering solutions is based on $|N_E|$, the amount of instances. The problem of typifying structured data involves a large amount of instances and especially in the Linked Data case, a much smaller amount of (value-level)schema features.

3.7 Experimental Evaluation

Based on real-world datasets representing different scenarios, we present a systematic evaluation of the clustering techniques and compare them with TYPifier. In this

work, we focus on assessing the quality of the type semantics inferred and if applicable, also the type hierarchy learned by the studied systems.

3.7.1 Datasets

We choose three datasets to capture the settings of (1) multiple-source heterogeneous Linked Data on the Web, (2) single-source hierarchical and (3) schema-less data in the enterprise (see Tab. 3.2 for overview). We keep only the instances for which type information is known. This type information is then used as the ground truth for evaluation.

Linked Data (BTC). The data used for this experiment is drawn from the datasets prepared for the Billion Triple Challenge[3]. The data sample we use for the experiment contains 334,661 instances from 5 datasets in the first 5 BTC chunks, namely `www.uniprot.org`, `my.opera.com`, `www4.wiwiss.fu-berlin.de`, `www.ebusiness-unibw.org` and `www.fao.org`.

DBpedia (DBP). This dataset contains structured information extracted from Wikipedia. It describes millions of things categorized in a hierarchy of types such as `person` and its subtypes `Scientist` and `Philosopher`. We use a sample of 3,600 instances of 16 types, including 49,751 triples from DBpedia Infobox.

Product Data (P). This is a product dataset from Reposito, which crawls from different product Web sites and acquires from different data providers. It is made available to us via a company called Reposito. The sample we use includes 22,331 instances of 6 types, which includes the types `printer`, `mobile`, `vacuum cleaner`, `speaker`, `monitor`, and `notebook`. Although these instances are represented in 111,647 triples, instances of these types are associated with the same set of five attributes. These schema features are not sufficient, hence we generate value-level schema as well as TF-IDF features from a total of 18,917 words. In Tab. 3.2, we can see that three corresponding versions of this dataset are used, P_{PS}, P_{TFIDF} and P_D that contain value-level schema and TF-IDF features, as well as all words, respectively (the PS column in Tab. 3.2 denotes the number of these features).

Parameters. For all systems, we sweep over parameter configurations to obtain the optimal parameters for every system as shown in Tab. 3.3. For K-Means++, the number of clusters is set to the number of types as given in the ground truth. OPTICS requires the two parameters $minPts$ and γ to control the density that determines the clusters. SVC relies on q, the kernel parameter. For BIRCH, we use different settings for the diameter threshold T (with branching factor set to 10). With $\epsilon = 0.8$, TYPifier produces the best results. For DBP, for which a hierarchy of types is produced, the $maxDepth$ of TYPifier was set to 2. For generating value-level schema features, which are used by all systems (the P_{PS} dataset), the optimal parameter is $\theta = 0.35$.

[3]http://challenge.semanticweb.org/

Table 3.2: Number of instances, triples, schema features (S), value-level schema features (PS) and types and the depth of the hierarchy tree for each dataset.

Dataset	Instance	Triple	S	Type	Hierarchy	PS
BTC	334,661	2,991,411	537	163	0	-
DBP	3,600	49,751	146	16	5	-
P_{PS}	22,331	111,647	5	6	0	136
P_{TFIDF}	22,331	111,647	5	6	0	7,211
P_D	22,331	111,647	5	6	0	18,917

Table 3.3: Optimal parameter settings.

Dataset	TYPifier	K-Means++	BIRCH	OPTICS	SVC
BTC	$\epsilon = 0.8$	k=18	$T = 1.5$	$\gamma = 0.2$ $minPts = 12$	$q = 0.7$
DBP	$\epsilon = 0.8$	k=10	$T = 2.9$	$\gamma = 0.1$ $minPts = 48$	$q = 0.5$
P_{PS}	$\epsilon = 0.8$	k=6	T=0.73	$\gamma = 0.1$ $minPts = 24$	$q = 0.45$
P_{TFIDF}	$\epsilon = 0.8$	k=6	T=4.5	$\gamma = 0.1$ $minPts = 36$	$q = 0.3$
P_D	$\epsilon = 0.8$	k=6	T=5.3	$\gamma = 0.1$ $minPts = 42$	$q = 0.2$

3.7.2 Efficiency of Typification

As shown in Tab. 3.4, BIRCH, K-Means++ and TYPifier are the systems with the best performance. Because the number of instances in BTC is more than ten times the number of instances in the second largest dataset, the time needed for BTC is higher than for DBP and P_{PS}. While BIRCH is slightly faster than K-Means++ for DBP, BTC and P_{PS}, it is much slower for the other two datasets. The same observation can be made for TYPifier, which is also slow in these two cases. Because these two datasets contain much more features than the others, these results suggest that K-Means++ is the most suitable one for dealing with high dimensionality. However, we will show that schema and value-level schema features yield much better results than data-level features. Since the number of features at the schema level is relatively small, the ability to cope with high dimensionality is not crucial for this problem.

While BIRCH, K-Means++ and TYPifier provide performance in the range of minutes, OPTICS and particularly SVC belong to a different performance class, requiring several days for computing results. In particular, while BIRCH, K-Means++ and TYPifier scale linearly with the number of instances, the performance of OPTICS and SVC drastically worsen as this number increases. This scalability is important because datasets for which type information is needed may contain a very large number of instances (e.g. Linked Data).

In summary, TYPifier is slower than K-Means++ especially when the number of

features is high. However, for low-dimensionality datasets, it performs relatively well and most importantly, scales linearly with the number of instances just like K-Means++ and BIRCH. Compared to this class of fast and scalable algorithms, TYPifier has the benefit of producing better results as discussed in the following.

Table 3.4: Efficiency of typification in ms.

	TYPifier	K-Means++	BIRCH	OPTICS	SVC
DBP	3,992	3,756	**3,395**	88,825	7,405,503
BTC	190,542	126,983	**97,763**	>1day	>1day
P_{PS}	21,125	19,560	**14,169**	>1day	>1day
P_{TFIDF}	212,632	**17,378**	134,952	>1day	>1day
P_D	253,943	**21,141**	213,207	>1day	>1day

3.7.3 Effectiveness of Typification

As shown in Tab. 3.5, TYPifier consistently outperforms other systems. Comparing to the second best system in every dataset setting, we have the following improvements in terms of precision, recall and F-measure respectively: +11.59%, +16.60% and +20.02% increase for BTC, -4.32%, +45.47% and +43.94% for P_{PS}, and -2.56%, +27.00% and +37.81% for DBP. On overage, this translates to +1.57%, +26.10% and +33.92% increase in precision, recall and F-measure, respectively. There are two exceptions where the precision of TYPifier is slightly lower than SVC and OPTICS. However, the poor recall levels exhibited by these systems for the other datasets suggest that the achieved performance is not stable. We observe that the high precision in these two cases (for DBP and P_{PS}) results from the general tendency of these two solutions to prefer fine-grained clusters. This strategy however, does not work for BTC.

It is important to note that the high quality results obtained by TYPifier (and other systems) are due to the use of (value-level)schema features as proposed. This is apparent in the differences between results obtained for P_{PS}, P_{TFIDF} and P_D. Clearly, P_{PS} yields the best performance. This suggests the computed value-level schema features are better indicators for types. For instance, for the types `PC` and `mobile phone`, the corresponding sets of computed value-level schema features are {`WXGA`, `Windows`, `DVD`, `ethernet`, `ram`, `GB`, `wlan`,...} and {`GSM`, `WCDMA`, `mobiltelefon`, `umts`, `nokia`,...}, respectively. While these features correctly capture the types, P_{TFIDF} contains features that have high importance for only some particular instances. For instance, the features `T400` have high TF-IDF scores for some instances, but do not characterize the type `PC` that these instances belong to well. P_D contains all features, hence also type indicators as well as non-type indicators.

The quality of the hierarchy tree obtained for DBpedia is indicated in Tab. 3.6. Measured in terms of tree edit distance, the hierarchy produced by TYPifier is closest to

Table 3.5: Effectiveness in terms of precision (P), recall (R) and F-measure (F). * indicates the statistically significant improvements of TYPifier over the best result achieved by the baselines (based on paired t-test with significance at $p < 0.05$)

	TYPifier			K-Means++			BIRCH		
	P	R	F	P	R	F	P	R	F
DBP	95.02	**80.11**	**86.93**	62.68	63.08	62.88	90.69	33.25	48.66
BTC	**77.22**	**79.41**	**78.30**	44.47	68.10	53.80	47.92	22.69	30.80
P_{PS}	95.56	**71.95**	**82.09**	52.38	49.46	50.88	77.02	17.31	28.27
P_{TFIDF}	**99.83**	0.99	1.96	30.45	30.83	30.64	77.52	26.93	**39.97**
P_D	99.79	1.32	2.61	43.20	43.33	43.26	42.78	**45.50**	**44.10**

	OPTICS			SVC			chg% over second best		
	P	R	F	P	R	F	P	R	F
DBP	93.43	39.82	55.84	**97.52**	29.90	45.77	-2.56*	+27.00*	+37.81*
BTC	53.80	75.04	62.67	69.20	61.70	65.24	+11.59*	+16.60*	+20.02*
P_{PS}	**99.88**	39.91	57.03	86.77	24.10	37.72	-4.32*	+45.47*	+43.94*
P_{TFIDF}	96.70	9.16	16.73	95.56	13.58	23.78	+3.23*	-96.79*	-95.09*
P_D	**1.00**	24.43	39.27	93.43	12.91	22.69	-0.21*	-97.09*	-94.08*

Table 3.6: Quality (tree edit distance) of hierarchy tree for DBpedia.

Typification	OPTICS	BIRCH
12	14	24

the ground truth. Fig. 3.3, 3.4, 3.5 and 3.6 show the given hierarchy tree as well as the trees computed by TYPifier, OPTICS and BIRCH, respectively. Note that these algorithms actually output clusters without labels. As discussed, for every cluster node, the best matching class from the given class hierarchy is determined. In these figures, the labels of these corresponding classes are used as cluster labels. Since a class might be the best match for several clusters, it may appear several times in the computed hierarchy tree. Clearly, there are qualitative differences that could not be captured by the tree edit distance: while the quality difference in terms of this metric is only 2, the result produced by TYPifier seems to be a much better match to the given hierarchy tree than BIRCH's result.

3.7.4 Parameter Sensitivity

Fig. 3.7 and Fig. 3.8 show the effect of θ. Higher θ means that the computed value-level schema features more frequently co-occur with each other. Thus with higher θ, value-level schema features become more representative. The counter effect is that a lesser amount of features is used as value-level schema features. Benefiting from more representative features, precision improves as θ increases. However, while re-

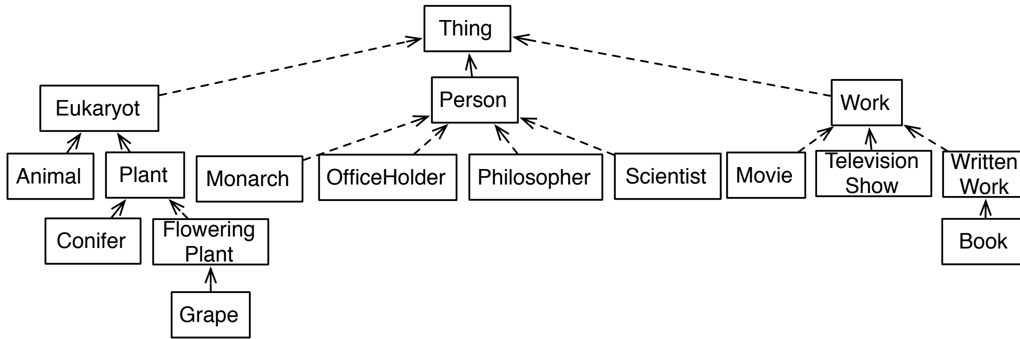


Figure 3.3: Given Hierarchy (Ground Truth).

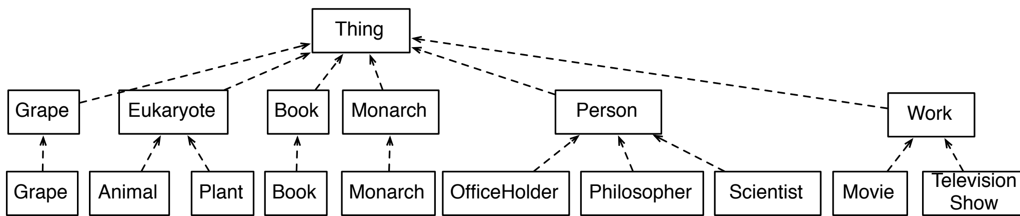


Figure 3.4: Hierarchy Generated by TYPifier.

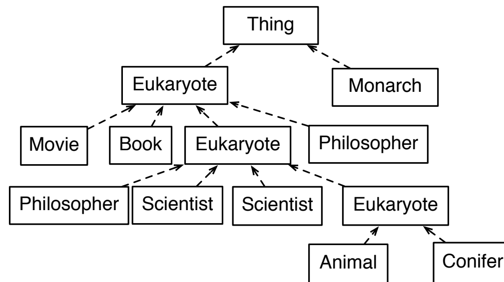


Figure 3.5: Hierarchy Generated by OPTICS.

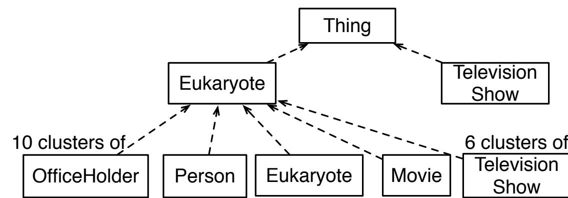


Figure 3.6: Hierarchy Generated by BIRCH

call improves at low levels of θ, it evidently drops as θ becomes large and larger. The number of features becomes too small to cover all the instances. When θ is larger than 0.5, there are no maximal cliques, hence also no value-level schema features. The best results for all systems could be obtained with $\theta = 0.35$.

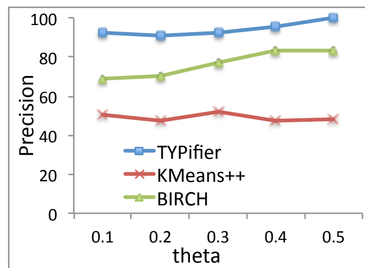


Figure 3.7: The effect of theta on precision.

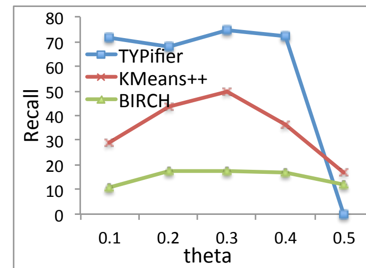


Figure 3.8: The effect of theta on recall.

Fig. 3.9 and Fig. 3.10 show results for different values of ϵ. BTC and P_{TFIDF} are not sensitive to changes in ϵ. This is mainly because features in these two datasets are less correlated. Then, changes in ϵ have less influence on the decision whether one cluster should be added as child or merged with another. Otherwise, as ϵ increases, TYPifier more aggressively adds clusters as children rather than merging them with the current one. This strategy seems to work well for DBP and P_{PS} because recall consistently increases with higher ϵ, and precision is also high at a high value for ϵ. However, this strategy has a negative effect on the quality of the hierarchy, which is not captured by Figs. 3.9+3.10 . The larger ϵ, the less likely clusters are merged. Using $\epsilon = 0.9$ in particular yields a hierarchy containing many parent and child nodes that actually represent the same type. The quality of the hierarchy is much higher at $\epsilon = 0.8$, which is one reason why it is the optimal parameter for TYPifier.

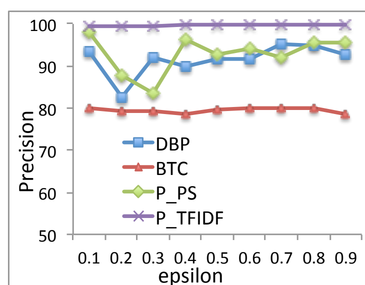


Figure 3.9: The effect of epsilon on precision.

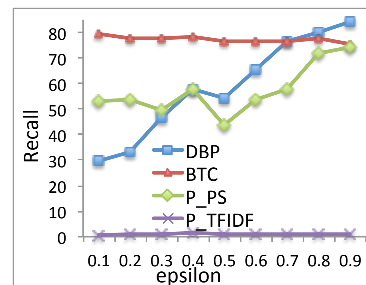


Figure 3.10: The effect of epsilon on recall.

3.8 Related Work

DataGuide [50] and *structure indexes* [20, 63] in general, are derived through the process of summarizing labeled paths in the data. A DataGuide is established by grouping together nodes sharing edge label sequences of incoming paths starting from the root nodes. The size of this summary can get exponentially larger than that of the original data graph. The structure index proposed in [20] can avoid this worst-case exponential blow-up. Further size reduction can be obtained by considering only paths up to a maximum predefined length k [63]. In principle, computing structure summaries from data and inferring the types of instances in the data are two different problems. Yet, grouping elements together based on schema features is actually similar to using edges ($k = 1$) for constructing structure indexes. However, the latter relies on strict equivalences between edges while for typification, different distance measures for the similarity between schema features may apply. Further, schema features are often too limited, hence might have to be derived from the data.

Conceptual clustering approaches are capable of generating hierarchical category structures that group instances according to their class. One best-known approach of conceptual clustering is COBWEB [47] which use category utility to evaluate the quality of the hierarchy. The algorithm supports four operations which are merging two cluster, splitting a cluster, inserting a new cluster and passing an instance down the hierarchy. It choses the operation for each instances by evaluating which operation can maximize the category utility. The main differences between Typifier and conceptual clustering is that Typifier process the (value-level) schema features rather than instances. And in practice, the number of (value-level) schema features is expected to be much less than the instances to be clustered.

For recovering the *semantics of Web tables* [73, 116], values in columns have to be matched against a database of possible column labels. While the "column semantics" is inferred from column values, type semantics are derived for entire tuples. When there is a database of type labels, classification techniques are more appropriate. In this work, we address the setting where type labels are unknown, hence clustering techniques are more relevant.

The Quartet method [31] is a hierarchical clustering method, which uses a distance matrix to generate a binary tree of minimum quartet tree cost. Further, a parameter-free divisive hierarchical clustering approach has been proposed recently [125], which is able to cluster categorical data and has the capability of subspace clustering. Schema features can be seen as categorical information and hence, this approach can also be used for typification. The drawback of this and the Quartet method is that the output is a binary tree. Thus, they cannot generate the correct hierarchy when there are more than two types that appear as siblings in the tree. Also, partitional and hierarchical clustering have been combined [74]. This method partitions the input into small subclusters and then merges the subclusters in a hierarchical way. It is tolerant

to outliers but still requires the number of clusters as a core parameter. Also, arbitrary shaped clusters can be found through single-link based clustering [94]. There is a recent method that improves the efficiency of DBSCAN [117]: first, it applies clustering to derive prototype clusters called leaders and then it speeds up DBSCAN by using these leaders to derive the density-based clusters. As discussed in the survey [126], other kinds of approaches including those based on graph theory, neural networks and heuristic search-based optimization exist. For typification, we show that good candidates include BIRCH because of its ability to find hierarchal clusters and SVC since it is able to deal with high-dimensional data. Although SVC cannot form hierarchies, it is able to automatically detect the number of types.

Another interesting direction is graph clustering. While the approaches studied in this work use a flat vectorial representation of features, this type considers rich structures in the graph. In this work, we focus on recovering types in the single table setting and regard the use of information captured by foreign key relationships between tables, or the combination of attribute and structure similarity [130], as future work.

Further, there are also clustering techniques proposed to detect overlapping clusters. The mathematical formulation of determining clusters that may possibly be overlapping was introduced in [6], which defines a cluster as a locally optimal subgraph with respect to a given metric. For instance, a cluster is locally optimal if the its density cannot be improved with the removal or addition of elements. Connected Iterative Scan is proposed [6] to find clusters with this local optimality. Clique Percolation is proposed in [92], which attempts to discover clusters by identifying cliques of size k. Two cliques are adjacent if they share $k-1$ instances. Overlapping clusters have also been detected by using the Gaussian Mixture Model [77], where elements are assigned to multiple clusters with different probabilities. These techniques can be used to support typification where instances may belong to different types. In this work, we focus on the case where instances are associated with exactly one type (the most probable type). Especially for use cases such as instance matching, we consider this type information as most important. The fact that some instances may also belong to other types could also be useful in some applications, which we will study as future work.

For learning value-level schema features, we use maximal clique algorithm to find the words that co-occur together to describe a certain type of instances. The other option to do this is using frequent itemsets mining [90]. Let the support of an itemset (a set of words) be the proportion of instances in the dataset that contain the itemset in the attribute values. We can learn the value-level schema features by finding all the frequent itemsets with a certain minimum support. The main drawback of using frequent itemsets mining is that it is difficult to find the correct value of the minimum support. In practice we should set the minimum support to bellow the smallest proportion of the size of every type. Otherwise, we cannot learn the value-level schema

features for the smallest type. However, since it is difficult to estimate the number of types and the size of each type in the dataset, it is difficult to find the correct value of the minimum support. Further, even if we know the value of minimum support, we can still not learn the correct words for a type that contains much more instances than the smallest type. For example, if 90% of instances are of type A and the rest are of type B, we should set the minimum support to be below 10%, so that we will not miss any features for B. However, using 10% as the minimum support to learn the value-level schema features for type A can result in the problem that a number of words that are not representative for A are also included in the result. Note in this thesis we construct FCG using CCP that is not depended on the size of types. Therefore we can avoid the drawback of frequent itemsets mining by applying maximal clique algorithm on FCG to learn value-level schema features.

3.9 Conclusion

For the typification problem, we present solutions based on existing clustering algorithms and TYPifier. In experiments, we show they perform best when using schema and especially, automatically generated value-level schema features. TYPifier is slightly slower but much closer to the class of the fastest clustering algorithms than some other types of clustering algorithms that do not scale to large number of instances. The benefit of TYPifier is that it provide superior results for the typification problem: it yields better types as well as a higher quality hierarchy tree.

Chapter 4

Blocking: Learning Type-specific Blocking Key and Key Value

Instance matching and blocking, a preprocessing step used for selecting candidate matches, require determining the most representative attributes of instances called keys, based on which similarities between instances are computed. We show that for the problem of learning blocking keys and key values, both generic techniques that do not exploit type information and supervised learning techniques optimized for one single predefined type of instances do not perform well on heterogeneous Web data capturing instances for which the *predefined type is too general*. That is, they actually belong to some types that are not explicitly specified in the data. We propose an unsupervised approach for *learning the type-specific blocking keys and key values*. Compared to state-of-the-art supervised and unsupervised learning approaches that are optimized for one general type, our approach improves efficiency as well as result quality. In particular, we show that the proposed strategy of learning type-specific blocking keys and key values improves both blocking and instance matching results.

Outline. We provide the introduction and contribution in Sec. 4.1 and Sec. 4.2. Then we give an overview of the problem and solutions in Sec. 4.3. The discussion of how Typifier is applied for blocking is shown in Sec. 4.4. The learning of keys and key values specific to these types is discussed in Sec. 4.5. We present experiments in Sec. 4.6, related work in Sec. 4.7, and conclusions in Sec. 4.8.

4.1 Introduction

To deal with the high computation complexity of instance matching, blocking approaches are designed to efficiently filter out non-matches, while generate instance pairs that are most likely to be matches. One main challenge behind the blocking is to find a few representative attributes (e.g. `title`), based on which the similarity between instances can be computed. These attributes and their values (so-called *keys* and *key values*) are not only needed for *blocking*, which uses a few simple *blocking keys* to quickly determine candidates and group them together in blocks, but also for the

instance matching task in general. Blocking is usually performed as a quick candidate selection step that involves boolean matching, while instance matching employs fuzzy similarity matching with fine-tuned similarity thresholds to further refine the candidates in the blocks.

State-of-the-art approaches for finding blocking keys assume a given schema, which describes the types and attributes associated with instances of these types. Given a particular type, its attributes, and training data for that type, existing approaches derive keys or more complex similarity matching function predicates that capture the keys as well as the similarity metrics and thresholds [14, 24, 83, 112, 113]. Not relying on training data, an unsupervised approach has been proposed recently to select attributes as keys based on their discriminability and coverage [108]. Basically, the discriminability of an attribute is measured by the diversity of its values such that low-discriminability means that many instances have the same values on that attribute. The coverage of an attribute refers to the number of instances that have that attribute. Intuitively, attributes shall be selected and used as keys when they are covered by many instances and also, discriminate them well. Finding keys in this supervised or unsupervised fashion is however problematic when the given instances actually belong to *multiple unknown types*.

The type information in the structured data maybe either not specified or too general to be useful. The current supervised solution [14, 24, 83, 112, 113] to that problem is to treat instances as belonging to one single type and learn blocking keys that are applicable to all of them, e.g. keys for all `Product` instances. Intuitively, when dealing with instances of the type `Computer` (n_4, n_5, n_6, and n_7 in Tab. 2.1), keys specific to `Computer` might be more useful (e.g. `Model`) than keys learned for `Product` (e.g. `Title`). However, when learned for a single big `Product` table, such computer-specific keys might not be selected because they are not applicable to many table records.

As an alternative to these schema-based supervised learning solutions, a schema- and type-agnostic approach has been recently proposed to deal with heterogeneous Web data. This approach does not exploit attributes (of specific types) for building keys, but instead, uses unstructured bags of features that can be extracted from the attribute values as key values [93]. They are simply composed of features that do not come with attribute information. Instances are considered similar when their key values overlap, i.e. they have some features in common.

4.2 Research Question and Contributions

In this chapter, we study the problem of the generation of match candidates and address the following research question:

Research Question 2. *How can match candidates be efficiently and effectively generated?*

On of the typical approaches to generate match candidates is blocking, which splits the original data sources into small blocks according to some criteria called blocking key, e.g. the title of two products data sources. Only the instances that are in the same block, i.e. the instances that have the same value of the blocking key, will be further compared in detail. We propose an unsupervised approach to learn the type-specific blocking keys and key values in this chapter and examine it with respect to the following hypothesis:

Hypothesis 2.1. *Match candidates can be efficiently and effectively generated by a blocking approach that uses discriminative type-specific attributes and values as blocking keys and key values.*

In this chapter, we show that considering the type of instances and the attributes the type captures significantly improves the quality of the results achievable with the schema-agnostic approach [93]. In particular, the quality improves when instead of using the general type [24, 108, 109], blocking keys are learned for specific types of instances in the data. To this end, we propose an *unsupervised approach* that relies entirely on the data for *learning keys and key values specific to these types*:

- **Learning type-specific keys.** We propose to leverage the *dependencies among attributes at the level of values*. We infer that an attribute a provides more similarity evidences than attribute b, when the value of b are more depended on the value of a, i.e. more instances have the same value of b, given they all have the same value of a.
- **Learning type-specific key values.** We propose an approach to select words which are discriminative for instances as key values. Those key values are able to lead to high quality blocking result.

In experiments, we show how this approach can be used for blocking and instance matching. Compared to state-of-the-art instance matching approaches, our solution greatly improves result quality (up to 201.56% improvement in terms of F-measure). Results are also promising when considering the blocking task only. Our approach yields up to 32.62% improvement in terms of reduction ratio over existing solutions for blocking [108]. It is also time efficient when compared against approaches that achieve similar result quality. Compared to the approach that yields second best result quality, our approach is up to several times faster w.r.t. blocking, and achieves similar performance w.r.t. instance matching.

4.3 Overview

In this chapter, we use the definitions of data and its example that was first given in Section 2.1. For ease of reading, we repeat some important definitions here.

Definition 3.1 (Conditional Co-Occ. Probability). *The probability that f_i and f_j co-occur, given f_j is*

$$p(f_i|f_j) = p(f_i, f_j)/p(f_j) = count(f_i, f_j)/|N^{f_j}|,$$

where $N^{f_j} = \{n \in N : f_j \in F(n)\}$ denotes all instances having f_j as feature, and

$$count(f_i, f_j) = |N^{f_i} \cap N^{f_j}|,$$

the co-occurrence count *of the two features f_i and f_j.*

Definition 3.2 (Feature Co-Occurrence Graph). *Given the vocabulary of features V, the* feature co-occurrence graph *is a labeled directed graph $G' = (U', E', L')$, with nodes $u \in U'$ stand for features in V and edges $w(u_i, u_j) \in E'$ capture the co-occurrence between the two features f_i and f_j, where edge labels stand for the conditional co-occurrence probabilities, $w = p(f_i|f_j)$.*

Definition 3.3 (Cluster). *A cluster is a tuple $C(F, N, S)$, with $F = \{f_1, \dots, f_i\}$ being the set of features (f_i is an element in the given set of all (value-level)schema features V, note we overload notation here to use V as either value-level schema or schema features), N the set of all instances that have an element in F as feature, i.e. $N = \{n : f \in F(n), f \in F\}$, and S the set of clusters that are either child or descendant nodes of C (representing subtypes of the type captured by C) in the hierarchy tree.*

Definition 3.4 (Cluster Distance). *Let $count(f_i, f_j)$ be the co-occurrence count of f_i and f_j and $|N_E^f|$ be the count of instances having f as feature, the distance $d(C_i, C_j)$ between the clusters C_i and C_j is the conditional probability of co-occurrence of its feature sets F_i and F_j,*

$$d(C_i, C_j) = p(F_i|F_j) = \frac{\sum_{f_i \in F_i, f_j \in F_j} count(f_i, f_j)}{\sum_{f \in F_j} |N^f|} = \frac{|N_i \cap N_j|}{|N_j|}, \tag{3.1}$$

where N_i (N_j) is the instance set associated with C_i (C_j).

Definition 2.4 (Find Blocking Keys and Key Values). *Given the data graph $G(U, E, L)$, we find a conjunction of Boolean function predicates (called* blocking scheme*) $\bigwedge_{e \in L^*} \sim_V^e$ where $\sim_V^e : U_V^e \times U_V^e \rightarrow \{true, false\}$, $U_V^e \subseteq U_V \subset U$ denotes the values of some* keys *$e \in L^*$, and $L^* \subset L$ is the set of* blocking keys*. Blocking maps every instance $n_i \in N$ to a subset $N_{B_i} \subseteq N$, an equivalence class of instances (instances, that according to the blocking scheme, are equivalent to n_i) called* block*: $N_{B_i} = [n_i] = \{n_j \in N : n_i, n_j \in N, \bigwedge_{e \in L^*} \sim_V^e (n_i, n_j) = true\}$.*

In this chapter, we focus on finding blocking keys and their value representation in the blocking context. We consider the case where the data might contain instances belonging to multiple unknown types. Our solution is that only the best ranked attribute determined for every type is used as key and non-value-level schema features

are extracted from its value as key values. As value-level schema features characterize a set instead of individual instances, they are considered less useful for finding similar instances and thus, are excluded from the value representation.

For finding blocks, there are two strategies: one is to iterate through the instances, as proposed recently [108]; using the features in the key value of a given instance, candidates are retrieved for that instance (*instance-based*). The alternative is to iterate through all features that appear in the instances' key values, and to retrieve all the candidates for a given feature (*feature-based*). For our approach, we use the feature-based strategy and apply value overlap as similarity measure.

Example 4.1. *Take the data in Tab. 2.1 as an example. Consider the attribute* `Title` *as the blocking key and the attribute value as the key value. Instance-based approaches iterate all the instances to find blocks. For example, for instance n_5, there are three features* `Sony`, `VAIO` *and* `SVF14212CXW` *in the key value. Then we find n_1 - n_6 are in the same block because they all have at least one of these features in their key values. On the other hand, feature-based approaches iterate all the words that appear in the values of* `Title` *to find blocks. For example, for the feature* `VAIO`*, it find n_4, n_5 and n_6 are in the same block because they all have the feature in* `Title`.

Additionally, we show that the type-specific attributes and key values are also useful for instance matching. Similar to blocking, different attributes values are used for different types. We use Jaccard similarity as measure where the threshold is learned using an existing technique [119].

4.4 Learning Types

We note that the hierarchy of types introduced in Chapter 3 may not be suitable for blocking. Not only does the learning of hierarchy require more computational effort, it also yields types too fine-grained that have only low coverage of the instances. Here, we introduce the technique used in this chapter to learn a set of types, as opposed to a hierarchy of types.

Note the value-level schema features are identified by searching for maximal cliques in the feature co-occurrence graph, i.e. clusters of features that pairwise, highly co-occur as measured in terms of conditional co-occurrence probability. Every maximal clique represents a cluster of features (i.e. a set of features) that are specific to a particular set of instances. We observe that these clusters are too fine-grained, hence do not directly correspond to the types given in the data. Hence we merge them to form larger clusters capturing types.

Two clusters F_i and F_j are merged when they highly co-occur, i.e. are close to each other in terms of cluster distance. Equivalently, this co-occurrence means there is a

strong overlap between the instances N_i that have some element in F_i and the instances N_j that have some element in F_j as features. Using this distance metric, we merge two clusters F_i and F_j when the $d(F_i, F_j) > \epsilon$ and $d(F_j, F_i) > \epsilon$, where ϵ is a parameter used to control the granularity respectively the number of clusters. Clusters resulting from this merging are used as types. In particular, given the resulting cluster F_i, the set of instances that belong to the type captured by F_i is simply N_i, i.e. instances that have some elements in F_i as features.

We note that just like the types that are learned in Chapter 3, the learning of types above also makes use of value-level schema features. However, in Chapter 3 it requires complex hierarchical relations between and operators on clusters to perform hierarchical clustering. The method above only uses the distance metric and applies cluster merging as the only operator. When doing hierarchical clustering, not only the features but also the instances representing the clusters have to be processed. For this, features and instances not only have to be merged but also might be added or spitted during the hierarchy construction process. Here, we only need to compute a set of clusters, where the computation stops after merging the clusters represented by the features. Only then, instances have to be considered, i.e. we simply group instances together that are associated with value-level schema features representing the same cluster.

We will now introduce the method to learn keys for a specific cluster and use them for blocking instances that belong to that cluster.

4.5 Learning Keys and Values

The keys are selected based on their ability to discriminate instances, measured in terms of a notion we call instance similarity, while the value representation excludes the value-level schema features computed before because they characterize a set (the type of) instances rather than individual instances.

4.5.1 Blocking Key Selection

Intuitively, attributes that are more useful in discriminating instances shall be selected as keys. Because keys capture similarity (thus are called *similarity attributes*), we introduce a notion of instance similarity that distinguishes attributes by their ability to discriminate instances, measured by the number of similarity evidences they provide:

Definition 4.1 (Attr.-specific Instance Similarity)**.** *Given $G(U, E, L)$ and the similarity relations $\sim_N$ and $\sim_V$ that indicate any two instances in N and any two attribute values in $U_V \subseteq U$ are similar, respectively, the* instance similarity *of an attribute e is measured as the number of instances in N that are similar, given they all have a similar value for attribute a,*

i.e. $s(a) = |\{n : n,m \in N, m \sim_N n, v(a,m) \sim_V v(a,n)\}|$, *where* $v(a,x)$ *refers to the value of the attribute* a *of the instance* x.

Note that this notion reflects the intuition behind existing learning-based approaches [14, 24, 83, 112, 113], which aim to learn keys that maximally separate positive examples from negative ones, i.e. keys that lead to a large amount of correct matches (high $s(a)$) when applied against training data. However, obtaining representative examples (for unknown types) is difficult. Thus, we propose to estimate instance similarity based on the *dependencies among attribute values* that can be observed in the data:

Definition 4.2 (Dependency)**.** *The strength of* dependency *of an attribute* a_i *from an attribute* a_j*, denoted* $d(a_i,a_j)$*, is inverse to the amount of information carried by values of* a_i*, given the values of* a_j *are known. Given* $G = (U,E,L)$*,* $d(a_i,a_j)$ *is measured as the entropy of the values of* a_i*,* $U_V^{a_i} = \{u_y | a_i(u_x,u_y) \in E\}$*, conditional on values of* a_j*,* $U_V^{a_j} = \{u_y | a_j(u_x,u_y) \in E\}$*:*

$$d(a_i,a_j) = (1 + H(U_V^{a_i} | U_V^{a_j}))^{-1} \tag{4.1}$$

$$= \left(1 + \sum_{u_j \in N_V^{a_j}} p(u_j) H(U_V^{a_i} | U_V^{a_j} = u_j)\right)^{-1}$$

$$= \left(1 + \sum_{u_i \in U_V^{a_i}, u_j \in U_V^{a_j}} p(u_i,u_j) log \frac{p(u_i)}{p(u_i,u_j)}\right)^{-1} \tag{4.2}$$

Example 4.2. *For the attributes of the type* `TV` *given in Tab. 2.1, we obtain* $d($`Manufacturer`$,$`Title`$) = 1$ *and* $d($`Title`$,$`Manufacturer`$) = 0.73$*. It can be easily observed in the data that given* `Title`*, it is easier to predict the value for* `Manufacturer`*, while given* `Manufacturer` *there are many products with different values for* `Title`*.*

When an attribute a_j is strongly dependent on an attribute a_i ($d(a_j,a_i)$ is high), it means that if instances have the same value for a_i then the probability they also have the same value for a_j is high. In particular, the conditional entropy used above implies that for any two instances n_i and n_j, $d(a_x,a_i) > d(a_x,a_j)$ means the probability n_i and n_j have the same value for a_x when they have the same value for a_i, is higher than the probability n_i and n_j have the same value for a_x when they have the same value for a_j, i.e. $p\left(v(n_i,a_x) = v(n_j,a_x) | v(n_i,a_i) = v(n_j,a_i)\right) > p\left(v(n_i,a_x) = (n_j,a_x) | v(n_i,a_j) = v(n_j,a_j)\right)$.

We note this notion of dependency correlates with instance similarity: namely, the more other attributes are dependent on a, the higher the amount of evidences a can

provide (higher $s(a)$) and hence, the less the amount of additional evidences the other attributes can contribute. We exploit this correlation to estimate $s(a)$ as follows:

Proposition 4.1. *Let $Agg : 2^{\mathbb{R}^+} \to \mathbb{R}^+$ be a monotonic aggregation function and L_E^F be the set of attributes associated with the type F. For two attributes $a_i, a_j \in L_E^F$, if $Agg_{a \in L_E^F}(d(a, a_i)) \geq Agg_{a \in L_E^F}(d(a, a_j))$ then we have $s(a_i) \geq s(a_j)$.*

Proposition 4.1 enable us to rank the instance similarity of an attribute a via the rank of the aggregation strength of the dependencies of all the other attributes from a. Alg. 4 computes the instance similarity for every attribute in L_E^F that is associated with a given type F, and stores it in S. Selecting the set of blocking keys L^* is simply done by sorting S and keeping the top ones with highest instance similarity. In the experiments, we find that we can achieve high quality blocking result by only using the top-1 attribute as the blocking key, i.e. L^* consists of exactly one attribute.

First, Alg. 4 computes $d(a_i, a_j)$ for every pair of attributes to construct the matrix M. Every combination of values for a_i and a_j has to be considered to derive the probabilities $p(u_i)$ and $p(u_i, u_j)$. At most, these steps of probability computation have to be performed $|N_V^{a_{max}}| \times |N_V^{a_{max}}|$ times (a_{max} denotes the attribute in the data that is associated with the largest number of values). We choose the *sum of dependencies* as an aggregation function for selecting similarity attributes. Then, instance similarities stored in S can be derived from the sums of row values in M. Since probabilities as well as the sums of dependency values in M can be computed in time linear to the size of G, we have this as overall complexity:

Theorem 4.1. *Given $G = (U, E, L)$, S can be computed in $\mathcal{O}((|U_V^{a_{max}}| \times |U_V^{a_{max}}|)^{|L_E^F| \times |L_E^F|})$.*

Thus, the selection of blocking key can be performed in polynomial time, bounded by the maximum number of instances and values associated with an attribute, and the amount of attributes. In practice, these bounds can be reduced substantially: not all attributes are relevant for blocking. For instance, strategies for ranking and filtering attributes (e.g. attributes too general or too type-specific to be useful for blocking) can be applied. Further, we note that just like other learning algorithms, a smaller amount of representative samples may be chosen instead of using all instances and values. Even without these strategies, the proposed algorithms scale well to large datasets in our experiment.

4.5.2 Key Value Selection

The idea behind selecting keys also applies for the selection of key values: they should help to discriminate instances. Value-level schema features characterize a set of instances well but are less useful in discriminating individual instances. As a result, while they help to identify types, they are less useful in identifying blocks of similar instances, as illustrated by the following example:

Algorithm 4: Computing instance similarities

Input: $G(G, E, L)$.
Data: $H(U_V^{a_i}, U_V^{a_j})$, $|L_E^F| \times |L_E^F|$ matrix M.
Result: S.

```
1 foreach a_i ∈ L_E^F ⊂ L do
2     foreach a_j ∈ L_E^F ⊂ L do
3         foreach u_i ∈ U_V^{a_i} = {u_y | a_i(u_x, u_y) ∈ E} do
4             foreach u_j ∈ U_V^{a_j} = {u_y | a_j(u_x, u_y) ∈ E} do
5                 H(U_V^{a_i}, U_V^{a_j}) = H(U_V^{a_i}, U_V^{a_j}) + p(u_i, u_j) log (p(u_i) / p(u_i, u_j));
6         M_ij = (1 + H(i, j))^{-1};
7 foreach a ∈ L_E^F ⊂ L do
8     S_a = s(a) = Σ_{1≤i≤|L_E^F|} M_{a_i};
9 return S;
```

Example 4.3. *There are two* `Product` *instances with the* `Title` *"Sony Fit Series VAIO SVF14214CXW laptop" and "Sony VAIO SVF14212CXW". When using all the words in* `Title` *as key values, these two instances exhibit an overlap and thus, would be placed in one block. However, after excluding value-level schema features such as* `Sony` *and* `VAIO`*, features that remain such as* `SVF14214CXW` *and* `SVF14212CXW` *are more specific to these two instances (rather than the class they belong to).*

Thus, value-level schema features associated with a type are excluded from the value representation of keys that have been selected for that type.

In summary, the output of the two steps discussed before is a blocking scheme for every type F, which consists of the attribute as key that has highest instance similarity. The key value representation contains all features from its value except those that are in the set of value-level schema features.

Example 4.4. *Take the data in Tab. 2.1 as an example. We can find the instances n_4, n_5, n_6 and n_7 are of the same type* `Computer`*. Then among all the four attributes, we use* `Title` *as the blocking key, sincie it is the best in term of instance similairity. Further, we can calculate that the words* `Sony` *and* `VAIO` *are value-level schema features, so they are excluded from the key values. Finally, we find instances N_4 and n_6 are in the same block because they share the same key value* `SVF14214CXW`.

4.6 Experimental Evaluation

Our work mainly addresses scenarios where type information is missing. To study the proposed solution, we employ a recent instance matching benchmark [66] that captures data from enterprise databases. We show that recognizing subtypes and learning type-specific attributes and values improve both efficiency and the quality of results. Compared against the best blocking baseline, our approach is up to several times faster and improves result quality measured in terms of reduction ratio (RR) by up to 32.62%, while maintaining similar results for pair completeness (PC). Considering the instance matching task, our solution, when used in combination with standard solutions for other tasks, improves F-measure result by up to 201.56%.

4.6.1 Datasets and Matching Tasks

We now briefly describe the datasets and matching tasks in the benchmark [66] that are used here. They cover the bibliographic and e-commerce domains. Tab. 4.1 provides an overview of these datasets.

Table 4.1: For each dataset pair: number of instances, words that appear in attribute values, value-level schema features (PS), and mappings indicating two instances are same (ground truth, GT).

Task	Instances		Words	PS	GT
	Dataset1	Dataset2			
AB	1,081	1,092	7,040	163	1,097
DS	2,616	64,263	89,189	386	5,347

DBLP-Scholar (DS). These datasets include data from DBLP (2,616 instances) and Google Scholar (64,263 instances). The attributes and values in Google Scholar are automatically extracted from full-text documents, hence contain a number of misspellings and heterogeneous representations of authors and venues. Because one instance in DBLP can be mapped to multiple instances in Google Scholar, the number of grount truth is larger than the number of instances in DBLP.

Abt-Buy (AB). This matching task is performed between instances of the product dataset from `http://abt.com` (1,081 instances) and `http://buy.com` (1,092 instances). Although all the products are stored in the same tables, they actually represent different types of products, including `computer`, `cell phone`, and `wash machine`. Compared to the other matching tasks featured by this benchmark, the one captured by this pair of datasets is in fact most representative for our scenario of heterogeneous data with multiple unknown types.

4.6.2 Experimental Setting

In the experiments, we select the four approaches [24, 93, 108, 109] as discussed before as baselines.

Systems. The first two approaches are recent approaches proposed for blocking, while the other twos are state-of-the-art solutions targeting the instance matching task. The first one is the type- and schema-agnostic approach (**Agnostic**), which treats instances as unstructured bags of words extracted from attribute values [93]. The second approach is the unsupervised solution for learning keys based on ranking attributes by discriminability and coverage [108] (**Unsupervised**). It requires setting two parameters: α is used to control the discriminability of a key and β determines if a key should be removed. We swept over parameters and used the optimal configurations with $\alpha = 0.9$ and $\beta = 0.5$. For instance matching, we use a supervised learning approach, which infers the attributes, similarity thresholds etc. from positive and negative examples [24] (**Supervised**). We use 10% of the matching instance pairs as positive examples, and negative examples are given by randomly generating pair of instances that apper in positive examples [60]. Finally, we use a recent approach, which iteratively cross-fertilizes instance matching results with schema matching results [109] until convergence (**PARIS**). The maxinum number of iterations is set to 5, while PARIS always terminated in less than four iterations in the experiments. PARIS's implementation is available for download. We use that PARIS Java implementation for the experiment, while the other systems were implemented in Java 5.

Figure 4.1: The Effect of θ on Effectiveness. Figure 4.2: The Effect of ϵ on Effectiveness.

We compare these approaches against our solution, called **TYPiMatch**. To solve the task of finding similarity threshold, TYPiMatch applies an existing mechanism that exploits redundancy between thresholds [119]. Also for TYPiMatch, we swept over parameters to find the optimal configuration such that the results presented in these experiments, represent best performances of the studied approaches. As parameters, TYPiMatch uses θ and ϵ to control the amount of value-level schema features and clusters, respectively. The effect of these parameters are shown in Figs. 4.1 and 4.2. Higher θ results in less value-level schema features. We observed that as θ increases, precision decreases and while recall decreases in the beginning (for small values of

θ), it also increases with larger values for θ. With less value-level schema features available for pruning key values, instance representations contain more features for classes. Using them to find similarity yields more matches (high recall) including incorrect ones (low precision). High ϵ prevents clusters from being merged. Thus, as ϵ increases, the number of clusters increases. To accommodate the increasing number of clusters that are also more fine-grained, TYPiMatch employs a large number of schemes. As a result of this, precision increases while recall decreases. Thus, these two parameters can be used to control the tradeoff between precision and recall. The results presented in the following are for the configuration $\epsilon = 0.07$ and $\theta = 0.1$.

All experiments were run on a server with two Intel Xeon 2.8GHz Dual-Core CPUs, using 8GB of main memory, running Linux with kernel version 2.6.18.

4.6.3 Efficiency of Blocking

Table 4.2: Performance of learning blocking keys in ms.

	TYPiMatch-Type	TYPiMatch-Key	Unsupervised
Abt	4,572	**1,041**	1,069
Buy		**895**	1,192
DBLP	42,262	**1,679**	1,952
Scholar		10,639	**9,661**

Tab. 4.2 shows learning times only for TYPiMatch and Supervised because Agnostic simply uses all attributes as keys. Both approaches yield good and fairly comparable performances, requiring only a few seconds to output the keys. We can see that TYPiMatch was slightly faster in learning keys (TYPiMatch-Key), which excludes the times needed to infer the types in the data (TYPiMatch-Type). Unsupervised iterates through the data graph and considers all instances to calculate the discriminability and coverage for each property. Thus, learning time for Unsupervised increases with the size of the dataset. This is also the case with TYPiMatch. It uses dependencies for key selection. Given an attribute, it inspects values associated with instances to calculate dependencies. However, only values owned by more than one instance are relevant. Thus, processing was faster for datasets that contain less duplicate values. Also, we can see that inferring types could be done efficiently. As shown in the TYPiMatch-Type column, this could be done in 4,572 ms for AB, and 42,262 ms for DS.

Tab. 4.3 shows blocking times for all three approaches. Note that blocking performance depends on two tasks: (1) retrieving candidates from the index and (2) performing pair-wise similarity computation to filter these candidates. As discussed, an instance-based [108] or a feature-based strategy can be used for candidate retrieval. While Unsupervised uses the former, the other two approaches employ the latter.

Table 4.3: Performance of blocking in ms.

	TYPiMatch	Agnostic	Unsupervised
AB	**9,635**	27,208	18,243
DS	**95,254**	882,626	491,569

TYPiMatch yields best performance while Agnostic is clearly worst. The latter is slow because it employs all words extracted from all attribute values, whereas the other two approaches only use words extracted from the keys. Compared to the baselines, TYPiMatch benefits from decomposing the dataset into n types. Because blocking is performed separately for every type, the number of retrieval operations needed equals the number of distinct features associated with every type. The total number of features for all types combined is higher than the number of distinct features for the single general type. Accordingly, the total number of retrieval operations is higher for TYPiMatch. However, the total number of candidates retrieved for all type features was lower than the total number of candidates retrieved for the general type features. Therefore in total, TYPiMatch spends less time in computing matches. In fact, this can be seen as a "blocking within blocking" strategy, where the data is partitioned into blocks corresponding to types, and subsequently, is further decomposed into more fine-grained blocks. However, we already pointed out in Section 2 that types and blocks resulting from standard blocking are different in many aspects.

4.6.4 Effectiveness of Blocking

Table 4.4: Effectiveness of blocking in terms of PC and RR, * indicates statistically significant improvements of TYPiMatch over the best baseline, Unsupervised (paired t-test, p<0.05).

	TYPiMatch		Agnostic		Unsupervised		chg% over second best	
	PC	RR	PC	RR	PC	RR	PC	RR
AB	86.14	**94.17**	85.96	26.45	**87.05**	71.01	-1.04*	+32.62*
DS	98.33	**99.64**	98.43	93.16	**99.93**	96.20	-1.60*	+3.58*

Tab. 4.4 presents total results for blocking effectiveness for the comparison between our solution and the other two unsupervised approaches. TYPiMatch largely outperforms these two in terms of RR and achieves comparable results for PC. Compared to the second best approach in every setting, TYPiMatch results in -1.32% decrease in PC and and +18.10% increase in RR on average. Intuitively, instances which re-

fer to the same real-world object should belong to the same type. Thus, learning the blocking scheme for every type should be possible without losing PC. However, the scheme obtained for every type is more specific and thus, can improve RR. The results show this is indeed true for the matching tasks in the benchmark, as TYPiMatch yields high quality types that can be used to preserve PC and improve RR. In particular, we note it produces the largest improvement for the AB task. This is because for products captured by these datasets, there exist many different types that exhibit different vocabularies of features. TYPiMatch is able to infer these types and to adopt distinguished words for matching instances with these types. For the DS task, which involves datasets that are well maintained and fairly homogeneous (contain only instances of type `Publication`), TYPiMatch achieves a small improvement.

4.6.5 Efficiency of Instance Matching

Table 4.5: Performance of learning instance matching schemes and executing them in ms.

	Learning		Instance Matching		
	TYPiMatch	Supervised	TYPiMatch	Supervised	PARIS
AB	**2,369**	2,784	24,527	66,856	**18,373**
DS	6,281	**5,726**	101,070	229,833	**70,479**

For instance matching, this version of TYPiMatch also includes the mechanism for learning thresholds [119]. The overall time needed by TYPiMatch and Supervised include the time for learning learning keys, key values and thresholds. Additionally, the time need by Supervised also includes that for capturing the weights, are shown in Tab. 4.5. These two approaches achieve similar performance. However, Supervised's learning process is non-deterministic. In our experiments, it often did not finish after several hours. This problem has been discussed before [119], which occurs especially when there is a large numbers of attributes that are considered to form the schema. The results presented for Supervised is an average over five successful runs.

Tab. 4.5 also shows the performance for instance matching. The process implemented by TYPiMatch for this is similar to the one used for blocking: it consists of feature-based retrieval of candidates and then, for matching them, Jaccard similarity as well as the learned thresholds are employed. As discussed, due to the use of types, this processing resembles a "blocking within instance matching" strategy. TYPiMatch retrieves candidates for type features and matches these type candidates. We can see this strategy yields better performance when analyzing the differences between TYPiMatch and Supervised. While the learning techniques they employ are

different, the resulting scheme employed by both these approaches for matching are of the same type, i.e. keys, values and thresholds (and weights). For example, both approaches can output a schema in the form like "two instances in DB are the same if the Jaccard similarity of `Title` is greater than 0.95". Just like TYPiMatch, Supervised applies feature-based candidate retrieval and uses such a scheme for matching. However, TYPiMatch needs only half of the time taken by Supervised. For example, it only cost TYPiMatch 24,527 ms to match the instances in AB, while Supervised spends 66,856 ms for instance matching. This is because TYPiMatch retrieves and matches instances belonging to types (using different schemes), Supervised applies its scheme to all candidates. Even though TYPiMatch requires a higher number of retrieval operations, it yields better overall performance because it reduces the number of pairwise similarity computation. PARIS is the fastest. We observed that this is mainly because PARIS is very aggressive in pruning candidates. Often, candidates are only kept when they have identical attribute values. While this largely increases performance, it leads to low recall when the data contains more noises, as discussed in the following.

Table 4.6: Number of matching candidates and RR.

	Full input mappings (Cartesian product)	TYPiMatch		Benchmark	
		Blocking result	RR(%)	Blocking result	RR(%)
AB	1.2 million	70,000	**94.17**	164,000	86.67
DS	16.8 million	605,000	**99.64**	607,000	96,39

We further compare the efficiency of instance matching based on blocking against the benchmark [66]. Because we only take use one attribute in the experiment, we compare our method with the non-learning approaches that also use one attribute in the benchmark. For the task AB, the instance matching time in the benchmark is between 600 ms and 53,500 ms that are achieved by PPJoin+ using Jaccard similarity function and FellegiSunter [45] using Winkler similarity function, respectively. And for the task DS, the instance matching time is between 3,400 ms and 164,000 ms that are achieved by PPJoin+ [124] using Cosine similarity function and FellegiSunter using Winkler similarity function. We can see that TYPimatch needs more time than PPJoin+ in the benchmark. As discussed in Sec. 2.5, the instance matching time is mainly determined by the number of instance comparisons and the cost of single instance pair comparison. Tab. 4.6 shows the number of candidates that are used for TYPiMatch and benchmark. Since both TYPiMatch and PPJoin+ compare only one attribute for every candidate using similar similarity function, the cost of singe instance comparison is similar. However, because TYPiMatch only compare instances that belong to the same type, it compares less than half of candidates that are processed by the benchmark for AB. Therefore, TYPiMatch should have achieved similar

(or better) efficiency compared to PPJoin+.

Besides the difference of machines that execute the instance matching, we observe another two reasons that explain the difference. First, the difference of time may results from the difference of the index that are used by two approaches. As discussed, TYPiMatch applies feature-based candidate retrieval, while PPJoin+ applies instance-based strategy. So TYPiMatch requires a higher number of retrieval operations. Especially, since TYPiMatch visits hard disk to retrieve the matching candidates and attribute values, the time difference of retrieval can be even bigger if the benchmark load all the data into the memory before instance matching. However, it is unknown that how the benchmark build and use the inverted index in practice.

Second, PPJoin+ use thresholsd as a restrictions to reduce the number of blocking key values; hence the less number of matching candidates. Besides, it can take use the positions of blocking key values in the attribute to further reduce the cost of single instance pair comparisons. Note TYPiMatch is compatible with PPJoin+. Thus we can combine the both technologies to achieve the performance that is better than using each of them along.

4.6.6 Effectiveness of Instance Matching

Table 4.7: Effectiveness of instance matching in terms R, P and F; * indicates statistically significant improvements of TYPiMatch over the best baseline, Supervised for AB, and PARIS for DS (paired t-test, $p<0.05$).

	TYPiMatch			Supervised			PARIS			chg% over second best		
	R	P	F	R	P	F	R	P	F	R	P	F
AB	**56.70**	**72.49**	**63.63**	55.79	13.01	21.10	2.55	12.44	4.24	+1.63*	+457.19*	+201.56*
DS	75.28	76.49	**75.88**	**79.66**	5.76	10.74	41.18	**89.08**	56.32	+82.81*	-14.13*	+34.73*

Tab. 4.7 presents results for instance matching effectiveness. TYPiMatch largely outperforms the other two approaches in terms of F-measure. We can see from the results that the product matching task AB is more difficult, where F-measures values obtained by all approaches are lower than for the other task. This is because the product descriptions contain a large amount of noisy text, while the bibliographic data employed in that benchmark is more structured.

Because Supervised randomly picks positive examples, the schemes it learns vary in different runs. We calculate the average for five runs. We observed that Supervised's strategy of learning the threshold from the skyline of negative examples tends to produce low values. As the result, Supervised covers most positive matches and thus, leads to high recall. However, this comes at the cost of very low precision.

The poor performance on the AB matching task suggests that PARIS is not successful with dealing with noises in textual descriptions. PARIS can propagate

evidences when the same values at the data or schema level can be found. However, this task involves many products, which refer to the same object, but due to noises in the data, these matching instances greatly vary in their values for `name` and `price`. PARIS's performance was better for the DS task. As a general observation, PARIS tends to be more aggressive in pruning results compared to the other approaches. This leads to high precision but low recall.

We further compare the effectiveness with the approaches in the benchmark [66]. COSY achieves the best result. However, because COSY is an commercial system, and it is unknown about the details like how similarity functions and thresholds are applied, it is difficult to compare TYPiMatch that focus on improving instance matching using type-specific blocking with such an full-fledge system. For the task AB, TYPiMatch outperforms all the other systems including non-learning approach and complex learning-based approaches that take use only one attribute for matching, where the best one achieves F-measures 54.8. For the task DS, TYPiMatch achieves similar result as the non-learning approaches, but performs worse than the learning-based approach. We observe two reasons that explains the performance difference for the DS. First, while the task AB refers to the datasets that contain products belonging to many different types, the DS task involves datasets that are fairly homogeneous. Therefore, TYPiMatch that are designed for heterogeneous data matching cannot improve the matching quality on DS as well as on AB. Second, while TYPiMatch only learns the threshold of one attribute in the experiment, the learning-based approaches in the benchmark take use SVM and decision tree to learn a more complex instance matching schema involving combinations of similarity functions and thresholds. This also explains why the learning-based approaches in the benchmark outperform TYPiMatch when two attributes are used.

4.7 Related Work

Throughout the paper, we have discussed the most related approaches applicable to the problem of learning blocking keys and key values [93, 108] as well as more complex instance matching schemes that also include similarity functions and thresholds [14, 24, 83, 119]. We pointed out that as opposed to existing solutions, we learn keys and key values that are specific for types derived from the data. Here, we provide a broader overview of solutions for the candidate generation problem.

Blocking approaches aim to reduce the number of similarity comparisons [16, 83, 93, 123]. These approaches typically associate each instance with blocking keys values, such that instances with the same key values are assigned to the same block. Examples, such as q-gram blocking [52], PPJoin+ [124] and suffix arrays based blocking [35], derive two instances as a match candidate if they share at least one q-gram,

prefix, and suffix respectively. The blocking keys and key values are either manually designed or learned from training examples with the help of machine learning. For example, [16, 83] model this problem as learning disjunctive sets of blocking predicate conjunctions that consist of blocking keys and key values.

Blocking as performed by our approach is different in terms of the procedure, which is applied only to every type of instances. The only technique specifically designed for dealing with heterogeneous Web data that may belong to multiple unknown types is the schema-agnostic approach [93] discussed throughout the paper. We show in the experiment that our approach of learning specific keys yields better results than that, which uses all attribute value tokens.

For the purpose of candidate calculations, there are also methods such as Sorted Neighborhood approach and Canopy clustering. The Sorted Neighborhood approach [54] first sorts the instances according to certain attributes, and then slide a fixed window through the sorted list of instances in order to limit the number of comparisons by only matching instances in the window. Canopy clustering [7, 22, 33, 52, 81] employs a cheap string similarity metric to group similar instances into the same clusters.

There are also several similarity join algorithms for efficient similarity comparison [2, 2, 23, 78, 101, 107]. They can be used to compute candidate matches based on the blocking scheme learned by our approach, where the conjunction of similarity function predicates acts as the join predicate.

Besides the conditional entropy that we use to evaluate the ability of an attribute to determine the values of other attributes, functional dependency approaches [41] can also be used fur the similar purpose. A set of attributes X is said to functionally determine another set of attributes Y, if and only if each value of X is associated with precisely one value of Y. A typical approach of functional dependency is GORDIAN[106], which can efficiently discover composite keys of a data schema. The main difference between the approach applied in this thesis and functional dependency approaches is that we do not restrict to the one-to-one mapping of attribute values, which is important for blocking. Consider a dataset with attributes `firat name` and `last name` for an example. Because one value of `first name` can be mapped to multiple values of `last name`, functional dependency approaches learn the only key of the dataset that consists of both `firat name` and `last name`. However, because of the errors and variations of attributes, we should not assume that two instances of the same person have exactly the same values of `firat name` and `last name`. Using the vlaue of `firat name` and `last name` together as blocking key value can result in missing true matches in the blocking result. Therefore, we choose a more flexible solution in this thesis, which is to sort the attributes according to their performance of blocking and use every top-rank attributes as blocking keys in multiple iterations.

4.8 Conclusion

For the problem of matching instances, we provide a solution to solve the subtasks of selecting discriminative attributes and representing their values. This solution is derived for every type learned from the data to recognize that in some datasets, instances cannot be assumed to be of one particular type but may actually belong to multiple unknown types. We showed in experiments that our approach of using type-specific keys and values improved both blocking and instance matching. In experiments, we show how this approach can be used for blocking and instance matching. Compared to state-of-the-art instance matching approaches, our solution greatly improves result quality (up to 201.56% improvement in terms of F-measure). Considering the blocking task, our approach yields 32.62% improvement in terms of reduction ratio. It is also time efficient when compared against approaches that achieve similar result quality. Currently, we focus on selecting keys and their values. One direction for future work is to apply this type-specific strategy to other tasks in instance matching such as learning similarity metrics and thresholds.

Chapter 5

Classification: Learning Rules for Effective Almost-parameter-free Instance Matching

In this chapter, we propose an efficient approach to learn attributes, similarity functions, and thresholds, called instance-matching rules, for finding matches. Existing rule-based approaches calculate similarity of each attribute separately, and identify an instance pair as a match if each of the similarities is high enough. They may fail to identify matching instance pairs if any one of the attributes fails to be matched. Besides, these approach cannot effectively learn the rules without the fine-tuning of parameters. These approaches are also expensive in learning, because they learn the best rule from a large number of candidates whose number depends on the number of attributes, similarity functions, and especially training examples. In this chapter, we address these three problems. We measure two instances as a whole by calculating the average similarity of a set of attributes to balance the errors in a single one. The approach we propose is almost free of parameters. The parameters need not fine-tuning and can be directly estimated from the training data. We then propose an efficient algorithm to learn the instance-matching rules from a significantly smaller set of candidates whose size only depends on the number of attributes and similarity functions. The experiments on both real and synthetic datasets show that our solution greatly improves the effectiveness and efficiency. Moreover, the approach is also effective in the way that it can achieve stable results when the parameters are set with a large range of different values.

Outline We organize the chapter as follows. The introduction and the contributions are presented in Sec. 5.1 and Sec. 5.2. We formally define the problem of learning mapping-threshold instance matching rules in Sec. 5.3. The methods for rule learning and execution are provided in Sec. 5.4 and Sec. 5.5 respectively. We present experiments in Sec. 5.6, related works in Sec. 5.7, and conclusions in Sec. 5.8.

5.1 Introduction

Because a number of calculated match candidates are actually incorrect, they need be further compared in detail to be classified to the class of matches and non-matches.

For this purpose, state-of-the-art approaches employ instance-matching rules to find instances that are the same. For example, n_1 and n_2 in Tab. 5.1 may form a match, if we consider a instance-matching rule "two product are identified as the same if they have similar `Title` and similar `Manufacturer`". Typical approaches for learning instance-matching rules involve selecting a set of attributes, and for each attribute determining the best similarity function (e.g. Jaccard, Cosine, QGram etc. [40]) and threshold. For example, the rule above can be specified as "two products are the same, if the *Jaccard* similarity of `Title` is greater than 0.4 and the *QGram* similarity of `Manufacturer` is equal to 1.0". We call this type of rules *attribute-threshold instance matching rule* (*aIR*), because it is satisfied only when the similarity of each attribute is higher than the threshold.

However there are three problems of aIR-based approaches. First, aIR-based approaches may fail to identify matching instance pairs if there are errors that occur in a single attribute leading to the similarity being incorrectly low. Secondly, the learning cost is expensive because these methods are designed to search a large number of candidates for the best rule. The number of the candidates is decided by the number of attributes, similarity functions, and especially training examples. While more training data may lead to higher effectiveness of the learned rules, it also results in much more learning time. And finally, these approaches cannot effectively learn the correct rules without fine-tuning of various parameters.

We propose an approach to learn instance-matching rules, which can solve these three problems. Firstly, we observe that although there may be errors in a single attribute, it is unlikely that the errors happens on every attribute. Therefore, we consider the similarity of two instances as the average similarity of a set of attributes. The two instances are considered as the same only when the average similarity is greater than the threshold. For example, instances n_1 and n_2 in Tab. 5.1 are identified as a match, if we consider a instance-matching rule "two products are the same, if the average of the *Jaccard* similarity of `Title` and the *QGram* similarity of `Manufacturer` is greater than 0.7". We call this type of instance-matching rule *mapping-threshold instance matching rule* (*mIR*).

Secondly, the approach we propose is efficient in the way that the number of rule candidates is irrelevant to the number of training examples. We observe that the average similarities of matches (and non-matches), that are calculated according to different attributes and similarity functions, follow different probability distributions. We propose to calculate the matching and non-matching certainties, as the certainties to assign an instance pair to either a match or a non-match, from the cumulative probability of (dis)similarities of (non-)matches. Then a pair of instances will be considered as the same if its certainty to be a match is greater than that to be a non-match. The decision boundary can be calculated as the only one threshold, such that when a pair of instances has similarity that is greater than the threshold, its matching certainty is always greater than the non-matching certainty. Since there is only one mIR can-

didate for a specific combination of attributes and similarity functions (because we calculate only one threshold for the candidate), the number of mIR candidates is only relevant to the number of attributes and similarity functions.

Finally, the parameters required by the approach can be easily estimated from the training data. Without fine-tuning of the parameters, the approach can result in stable and high-quality results.

5.2 Research Question and Contribution

In this chapter we address the following research question:

Research Question 3. *How can the match candidates be effectively classified to matches and non-matches?*

This question is derived from the challenge of low quality data, in which there are various errors of attribute values. We observe that to deal with the errors in a single attribute, the similarity evidences that are obtained based on a set of attributes are more reliable. This observation leads to the following hypothesis:

Hypothesis 3.1. *The instance matching problem is* monotonic, *if all the matches have higher average similarities than all the non-matches for a set of attributes. And if the instance matching problem is monotonic, then there exists an instance-matching rule which can correctly identify all the matching instance pairs.*

Based the above research question and hypothesis, we propose an approach for effective almost-parameter-free instance matching, providing three main contributions as follows:

- **Almost-parameter-free Instance Matching.** The approach we propose in this chapter is almost free of parameters. The parameters need not fine-tuning and can be directly estimated from the training data.
- **Efficient Algorithm to Learn Instance-matching Rule.** We propose an efficient algorithm to learn instance matching rules, which search the best mIR from a significantly smaller set of candidates compared to existing aIR learning approaches. The number of candidates depends only on the number of attributes and similarity functions, while it is irrelevant to the number of training examples.
- **Efficient algorithm to Execute Instance-matching Rule.** We propose an efficient algorithm to execute the mIR, which is able to make a decision only based on a part of similarities in the rule.

Comparing to the state-of-the-art aIR learning approaches, our solution greatly improves the effectiveness as well as efficiency by up to 87% reduction of learning time. Moreover, the approach is also effective in the way that it can achieve stable results when the parameters are set within a large range of different values.

5.3 Instance Matching

The problem tackled in this chapter is to classify the match candidates to the classes of matches $\mathbf{M}^+$ and non-matches $\mathbf{M}^-$. The class $\mathbf{M}^+$ contains all the mappings that refer to the same real-world entities (matches), and the class $\mathbf{M}^-$ contains all the mappings that refer to different entities (non-matches).

We reuse the definitions of data that was first given in Section 2.1. For the ease of explaining the approaches, we assume the data has already been processed in the typication step so that all the instances are of the same type. The examples instances of the type `Computer` are shown in Tab. 5.1

Table 5.1: A sample of product instances taken from a real E-commerce database; matching instance pairs are (n_1, n_2), (n_3, n_4), (n_5, n_6), (n_5, n_7), (n_6, n_7).

ID	Title	Manufacturer	Description
n_1	MacBook Air MD231D	Apple	Apple MacBook Air 33,8 cm (13,3 inches) Notebook (Intel Core i5, 1,8GHz, 4GB RAM, 128GB hdd, Intel HD 4000)
n_2	Apple MacBook Air MD231D 13 inch laptop (newest version)	Apple	MacBook Air MD231D laptop Core i5-13,3 inches
n_3	Apple MacBook Pro ME664LL Notebook	Apple Inc.	MacBook Pro with Retina Display 33.8 cm (13.3 inches) laptop Intel Core i7, 2.4GHz, 8GB RAM, 256G SSD, NVIDIA GeForce GT 650M, Mac OS
n_4	Apple MacBook Pro ME664LL	Apple	Apple MacBook Pro ME664LL Notebook with Retina Display (Intel Core i7, 2.4GHz, 8GB RAM, 256G hard disk, NVIDIA GeForce GT 650M)
n_5	ASUS UX31A-R4003V Notebook	ASUS Computer Inc.	Asus Prime Laptop UX31A-R4003V 33.8 cm (13.3 inches) Ultrabook (Intel Core i7-3517U, 1.9 GHz, 4GB RAM, 256GB HDD, Intel HD 4000)
n_6	ASUS UX31A-R4003V Notebook	ASUS	ASUS UX31A R4003V Notebook - Core i7 1.9 GHz - 13.3 inch - 4 GB RAM - 256 GB HDD
n_7	Asus prime Laptop UX31A-R4003V 13.3 inch laptop	ASUS	ASUS Laptop Core i7 1.9 GHz, 13.3 inch, 4 GB RAM, 256 GB HDD
n_8	ASUS N550JV-DB72T Notebook	ASUS Inc.	ASUS Core i7 notebook, with 15 inch, 4 GB RAM, 256 GB SSD

For the ease of reading, we repeat the definition of attribute-threshold instance matching rule and mapping-threshold instance matching rule here.

Definition 2.5 (Attr.-thresh. Inst. Match. Rule). *Given a set* $\{a_1, a_2, \cdots, a_d\}$ *of attributes, and a set* $\{g_1, g_2, \cdots, g_d\}$ *of similarity functions, let* $g_i(a_i) \geq \theta_i$ *denote a* similarity function predicate *where* $0 \leq \theta_i \leq 1$*. For any two instances* n *and* n'*, the similarity function predicate returns true if* $g_i(n[a_i], n'[a_i]) \geq \theta_i$*. An attribute-threshold instance matching rule is a conjunction of similarity function predicates as* $\bigwedge_{i=1}^{d} g_i(a_i) \geq \theta_i$*. A pair of instances* n *and* n' *are considered as a match if they satisfy all the similarity function predicates in the rule.*

Definition 2.6 (Map.-thresh. Inst. Match. Rule). *Given a set* $\{a_1, a_2, \cdots, a_d\}$ *of attributes and correspondingly a set* $\{g_1, g_2, \cdots, g_d\}$ *of similarity functions, a* rule function $f : N \times N \to [0,1]$ *calculates the similarity between two instances as the average of every attribute similarity, i.e.* $f(n, n') = \frac{\sum_{i=1}^{d} g_i(n[a_i], n'[a_i])}{d}, n, n' \in N$*. Given a threshold* θ*, a mapping-threshold instance matching rule (mIR) is defined as a tuple* $\lambda(f, \theta)$*, such that two instance* $n \in N$ *and* $n' \in N$ *are considered as a match if* $f(n, n') \geq \theta$*, where* $0 \leq \theta \leq 1$*.*

Table 5.2: Similarities calculated according to the similarity function $Jaccard(\texttt{Title})$ and $QGram(\texttt{Manufacturer})$, and the rule function $f = \frac{Jaccard(\texttt{Title}) + QGram(\texttt{Manufacturer})}{2}$ for the data in Tbl.2.1.

Mapping	(Non-)Matching	$Jaccard(\texttt{Title})$	$QGram(\texttt{Manufacturer})$	f	$1-f$
(n_1, n_2)	$\mathbf{M}^+$	0.42	1.00	0.71	0.29
(n_5, n_6)	$\mathbf{M}^+$	1.00	0.32	0.66	0.34
(n_5, n_8)	$\mathbf{M}^-$	0.50	0.62	0.56	0.44
(n_3, n_8)	$\mathbf{M}^-$	0.14	0.48	0.31	0.69

Comparing to the general machine learning techniques (e.g. SVM and decision tree [24, 119]), aIR can be more efficiently executed resulting from less calculation of similarity function predicates. Especially, it does not have to test all the similarity function predicates, if any one of the predicates in the rule returns false. However, there are also drawbacks of aIR-based approaches:

- **aIR approach is sensitive to errors in attributes.** Even if most attributes are the same, an aIR might still fails to identify a matching instance pair, due to a variety of errors (e.g. spelling errors, missing values etc.) that occur in a single attribute. An attribute may fail to be matched due to such errors.
- **aIR approach depends on fine-tuning of parameters.** Existing approach requires fine-tuning of various parameters. For example, the method proposed by Chaudhuri et al. [24] needs the number of similarity function predicates and the number of rules to learn as parameters. And the approach proposed by Wang et al. [119] requires experts to manually select attributes, and for some of

the attributes the similarity functions and thresholds, so that it can learn similarity functions and thresholds for the other attributes. Without the fine-tuning of such parameters, these approaches cannot learn effective rules.

- **The cost for learning aIR is expensive**: Existing approaches [24, 119] are designed to search a large number of candidates for the best rule. The number of the candidates is decided by the number of attributes, similarity functions, and especially training examples (see details in Appendix A.1). While more training data results in higher quality of the learned rules, it also leads to much more time for learning. Even though various methods being proposed to boost efficiency (e.g. greedy algorithm [24] or removing candidates composed of redundant thresholds and similarity functions [119] etc.), the cost for learning is still expensive.

Example 5.1. *Consider an aIR* $Jaccard(\texttt{Title}) \geq \theta_1 \land QGram(\texttt{Manufacturer}) \geq \theta_2$ *for the data in Tab. 5.1. As the similarities that are shown in Tab. 5.2,* θ_1 *and* θ_2 *must be set with small values that are not greater than 0.42 and 0.32 respectively, if the aIR can identify both matches* (n_1,n_2) *and* (n_5,n_6) *that are with spelling errors in* `Title` *and* `Manufacturer` *respectively. However, the aIR designed in this way is incorrect, because it is unavoidable to assign the non-matching instance pair* (n_5,n_8) *to the class of match. In this circumstance, it is impossible to define an aIR that can correctly distinguish all matches and non-matches.*

Due to these drawbacks, aIR learning may not yield to good results in terms of both effectiveness and efficiency. We adopt a different form of mapping-threshold instance-matching rule that performs better when there are errors in attributes, and can be efficiently learned by an approach that is almost free of parameters.

Example 5.2. *Consider the average similarities that are calculated according to the rule function* $f = \frac{Jaccard(\texttt{Title})+QGram(\texttt{Manufacturer})}{2}$ *in Tbl. 5.2. Since both matching instance pairs* (n_1,n_2) *and* (n_5,n_6) *have greater similarities than the non-matching instance pairs* (n_5,n_8) *and* (n_3,n_8)*, we can now correctly identify all matches using the mIR* $\frac{Jaccard(\texttt{Title})+Cosine(\texttt{Manufacturer})}{2} \geq 0.60$.

Consider a set of mappings M as training examples, which are composed of positive examples $M^+ \subseteq \mathbf{M}^+$, i.e. instance pairs that are known to be correct, and negative examples $M^- \subseteq \mathbf{M}^-$, i.e. instance pairs that are known to be incorrect. Obviously there is $M = M^+ \cup M^-$. Let Ψ be a set of mIRs that are learned from M, and let $M_\Psi \subseteq M$ be the instance pairs in M that satisfies any of the mIR in Ψ. Ideally we hope M_Ψ is exactly equal to M^+. However, in reality, M_Ψ may not only contain non-matching instance pairs, but also miss matching instance pairs that fail to be identified by Ψ. To evaluate the quality of Ψ, we consider a general objective

function $Q(\Psi, M^+, M^-)$. The less false positives, i.e. the less non-matching instance pairs that are included in M_Ψ, the higher $Q(\Psi, M^+, M^-)$; the less false negatives, i.e. the less matching instance pairs that are failed to be identified by Ψ, the higher $Q(\Psi, M^+, M^-)$. For example, F-measure is used as an general objective function $\frac{2pr}{p+r}$, where $p = \frac{|M_\Psi \cap M^+|}{|M_\Psi|}$ is precision and $r = \frac{|M_\Psi \cap M^+|}{M^+}$ is recall.

Now we formalize the problem of learning mIR for effective instance matching problem as follows:

Definition 5.1 (mIR-learning problem)**.** *Given a set of positive examples M^+ and a set of negative examples M^-, the goal in the mIR learning problem is to learn Ψ, a set of mIRs, to maximize a pre-defined objective function $Q(\Psi, M^+, M^-)$.*

5.4 Algorithm for Learning mIR

In this section, we describe an efficient algorithm for learning a set of mIR. We first introduce an approach for the restricted version of the problem in which only one mIR is learned. For a given rule function, we propose to calculate the matching and non-matching certainties, as the certainties to assign an instance pair to either a match or a non-match, from the cumulative probability of (dis)similarities of (non-)matches (Section 5.4.1). A pair of instances would be classified to matches when its matching certainty is greater than the non-matching certainty. Based on the estimation of (non-)matching certainty (Section 5.4.2), we learn the only threshold as the decision boundary, that if the similarity of an instance pair is greater than the threshold, its matching certainty is always greater than the non-matching certainty (Section 5.4.3). We then introduce the method for fast evaluation of the quality of a mIR candidate (Section 5.4.4), based on which an hill-climbing algorithm is proposed to learn the best mIR from a significantly smaller set of candidates compared to the number of candidates in existing aIR-based approaches (Section 5.4.5). Finally we describe an approximation algorithm to learn a set of mIRs which maximize the pre-defined objective function (Section 5.4.6). The algorithm for the restricted version that learns the best mIR is one of our main contributions in this chapter.

5.4.1 Certainty for Instance Matching

Let f be a rule function, henceforth, we refer to the *similarity* of a pair of instances (n, n') as the value of the rule function $x = f(n, n')$, and the *dissimilarity* as $y = 1 - f(n, n')$. Obviously, there is $0 \leq x \leq 1$, $0 \leq y \leq 1$, and $x + y = 1$. For a pair of instances, the higher the similarity x, the more likely it is assigned to $\mathbf{M}^+$; and the higher the dissimilarity y, the more likely it is assigned to $\mathbf{M}^-$. However, even

though the similarity (or dissimilarity) serves as evidences of how similar (or dissimilar) two instances are, we are still not certain to make a decision that whether they are similar enough to be a match. For examples, given a dataset with seldom errors, most matches may have very high similarity around 1. Therefore, when there are two instances with similarity 0.7, we may have low certainty to infer them as a match. On the other hand, given a dataset with various errors, we can have every high certainty to classify the instance pair as a match, because we know most matches in the dataset have similarity that are less than 0.7.

We exploit positive and negative examples to calculate the certainty of assigning a pair of instances to either $\mathbf{M}^+$ or $\mathbf{M}^-$. Intuitively, for an unlabeled instance pair with similarity value x (or dissimilarity value y), the more positive (or negative) examples we have observed that have similarities (or dissimilarities) less than x (or y), the more certainty we have to assign it to $\mathbf{M}^+$ (or $\mathbf{M}^-$). For example, when two instances have similarity value 0, there should be the lowest certainty to assign them to $\mathbf{M}^+$, because no (or few) positive examples have similarities that are less than or equal to 0. On the other hand, when they have similarity 1, the certainty of assigning it to $\mathbf{M}^+$ is the highest because all positive examples have similarities that are not greater than 1.

Definition 5.2 (Matching Certainty)**.** *Let $X = \{x|0 \leq x \leq 1\}$ be a random variable of similarities calculated by a rule function f. The matching certainty is the certainty of assigning an instance pair with similarity x to $\mathbf{M}^+$, which is calculated from the cumulative distribution function (CDF) of X as $F(x, \mathbf{M}^+) = Pr(X \leq x, \mathbf{M}^+)$, i.e. the cumulative probability of a pair of instances that belongs to $\mathbf{M}^+$, and has similarity that is less than or equal to x.*

Obviously, the higher the value of x, the higher $F(x, \mathbf{M}^+)$, and hence the higher matching certainty. The value of $F(x, \mathbf{M}^+)$ can also be viewed as the probability of a match that has similarity not exceeding x. Then if the similarity of an instance pair exceeds x, we can derive that the matching certainty of the instance pair should be also greater than $F(x, \mathbf{M}^+)$.

Similarly, we can define the non-matching certainty as follows:

Definition 5.3 (Non-matching Certainty)**.** *Let $Y = \{y|0 \leq y \leq 1\}$ be a random variable of dissimilarities calculated by $1 - f$. The non-matching certainty is the certainty of assigning an instance pair with dissimilarity y to $\mathbf{M}^-$, which is calculated from the CDF of Y as $F(y, \mathbf{M}^-) = Pr(Y \leq y, \mathbf{M}^-)$, i.e. the cumulative probability of a pair of instances that belongs to $\mathbf{M}^-$, and has similarity that is less than or equal to y.*

We can classify a pair of instances into either $\mathbf{M}^+$ or $\mathbf{M}^-$ by comparing the (non-)matching certainties. The pair is classified to $\mathbf{M}^+$ if the matching certainty is greater than the non-matching certainty, otherwise it is classified to $\mathbf{M}^-$.

Example 5.3. *We calculate the matching and non-matching certainty for an instance pair with similarity* 0.70 *(and dissimilarity* 0.30*), according to training examples that are listed in*

Tab. 5.2. Because only the positive example (n_5, n_6) among all four examples has similarity lower than 0.70, the matching certainty is 0.25. Similarly, because none of negative examples has dissimilarity lower than 0.30, the non-matching certainty is 0. As a result, we classify the instance pair to $\mathbf{M}^+$ because the matching certainty is greater than the non-matching certainty.

5.4.2 Estimate (Non-)Matching Certainty

In this section, we describe the approach to estimate matching certainty $F(x, \mathbf{M}^+)$ and non-matching certainty $F(y, \mathbf{M}^-)$. Since these two types of certainty can be estimated in the very similar way, we focus on introducing the estimation of matching certainty.

Let $q(x, \mathbf{M}^+)$ be the density distribution function of $F(x, \mathbf{M}^+)$ and $q(\mathbf{M}^+)$ be the prior probability of the class $\mathbf{M}^+$, we can rewrite $F(x, \mathbf{M}^+)$ as follows:

$$\begin{aligned} F(x \cap \mathbf{M}^+) &= \int_0^x q(t, \mathbf{M}^+) dt \\ &= \int_0^x q(t|\mathbf{M}^+) q(\mathbf{M}^+) dt \\ &= q(\mathbf{M}^+) \int_0^x q(t|\mathbf{M}^+) dt \\ &= q(\mathbf{M}^+) F(x|\mathbf{M}^+) \end{aligned} \tag{5.1}$$

where $F(x|\mathbf{M}^+)$ is the class-conditional cumulative probability function (CCF) over similarity. Note that the class prior $q(\mathbf{M}^+)$ can often be estimated simply from the fraction of positive examples in the training data. Eq. 5.1 converts the matching certainty estimation problem to estimate the CCF $F(x|\mathbf{M}^+)$. Obviously there are $F(0|\mathbf{M}^+) = 0$ and $F(1|\mathbf{M}^+) = 1$

Observation 5.1. *Suppose $X = \{x | 0 \le x \le 1\}$ be the random variable of similarities calculated from matches, X follows different CCF if the observed data of X is calculated according to different rule functions.*

Example 5.4. *We sampled positive and negative examples from a real bibliographic database provided by the benchmark [66]. The database has the same schema as the data in Tab. 5.1. As illustrated in Fig 5.1, consider the dissimilarities of negative examples, that are calculated according to two different rule functions $f_1 = \frac{Jaccard(\texttt{Title}) + QGram(\texttt{Manufacturer})}{2}$ and $f_2 = \frac{Jaccard(\texttt{Title}) + Cosine(\texttt{Description})}{2}$, as an example. Different rule functions refer to different distributions. Compared to the dissimilarities calculated according f_2, those calculated according to f_1 distribute more widely in the interval [0.2, 1], as shown in Fig. 5.1(b) and Fig. 5.1(d) respectively.*

Given a rule function f, we can obtain a set of observed similarities $\{x_1, x_2, \cdots x_k\}$ on the variable X, which are calculated from positive examples. Then the value of

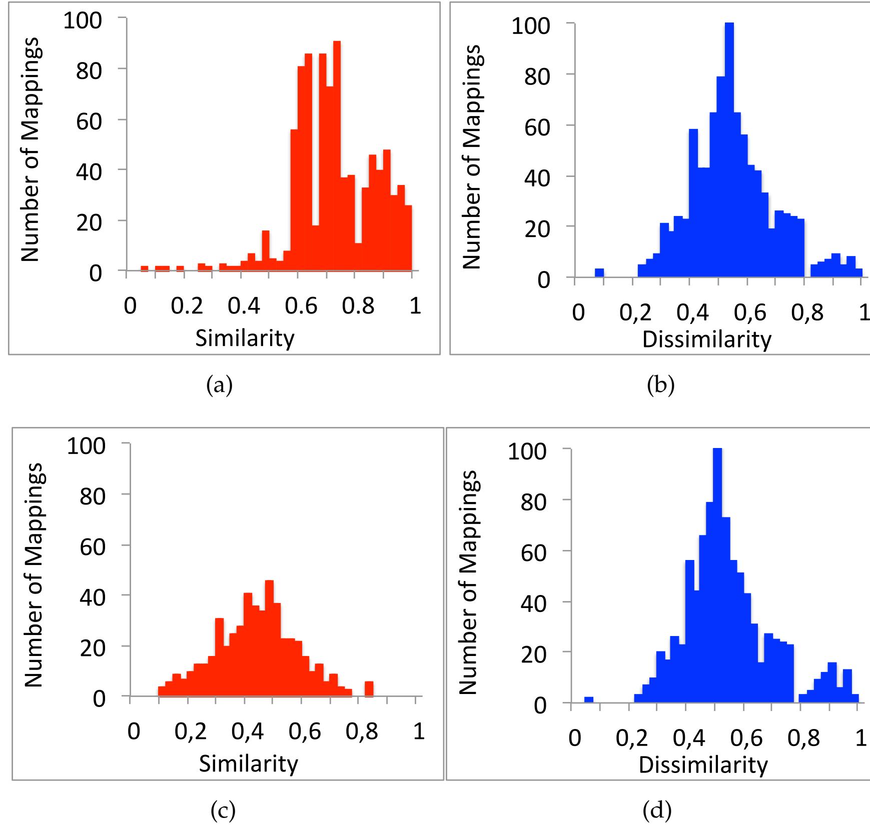


Figure 5.1: Consider two rule functions $f_1 = \frac{Jaccard(\texttt{Title})+QGram(\texttt{Manufacturer})}{2}$ and $f_2 = \frac{Jaccard(\texttt{Title})+Cosine(\texttt{Description})}{2}$. Figure (a) and (c) show the histograms for similarities of positive examples calculated according to f_1 and f_2 respectively. And Fig. (b) and (d) show the histograms for dissimilarities of negative examples calculated according to f_1 and f_2 respectively.

.

$F(x_i|\mathbf{M}^+)$ at the point x_i can be simply estimated as follows:

$$F(x_i|\mathbf{M}^+) = \frac{|\{x|x \leq x_i\}|}{k}, \text{for } x \in \{x_1, x_2, \cdots x_k\} \tag{5.2}$$

where $|\{x|x \leq x_i\}|$ is the amount of the observed data that is less than or equal to x_i.

The Eq. 5.2 can be calculated more efficiently by ranking the observed similarities. When the observed similarities from X are sorted in an ascending order, and let O_i be

the rank number of x_i, the value of $F(x_i|\mathbf{M}^+)$ is estimated as follows:

$$F(x_i|\mathbf{M}^+) = \frac{O_i}{k} \tag{5.3}$$

We are then able to estimate $F(x|\mathbf{M}^+)$ for any unobserved similarity x in $[0,1]$ by using Eq. 5.3. Let x_i and x_{i+1} be two adjacent observed similarities, for any unobserved similarity $x_u \in [x_i, x_{i+1}]$, $F(x_u|\mathbf{M}^+)$ can be estimated via linear interpolation as follows:

$$F(x_u|\mathbf{M}^+) = F(x_i|\mathbf{M}^+) + \frac{(F(x_{i+1}|\mathbf{M}^+) - F(x_i|\mathbf{M}^+))(x_u - x_i)}{x_{i+1} - x_i} \tag{5.4}$$

We call the CCF estimation by Eq. 5.4 as *interCCF*. The time complexity of interCCF is $O(\log k)$, where k is the number of training examples (see analysis in Appendix A.3). We observe that when there are a number of training examples available, the interCCF can reach high accuracy. Otherwise, because the real CCF usually does not simply follow a linear function, this estimation by the linear interpolation may largely deviate from the true value.

As a solution, we can also fit $F(x|\mathbf{M}^+)$ to the known cumulative probability distribution functions. Because the range of similarity is $[0,1]$, we only consider the cumulative distributions that are defined on the same interval, such as Beta distribution, power-law distribution. Among all these distributions, we find that power-law distribution is the most suitable for the instance matching problem in experiments, which is generally defined as follows:

$$F(x|\mathbf{M}^+) = ax^b, x \in [0,1] \tag{5.5}$$

where a and b are two parameters that need to be estimated from the training data. Since there is $F(1|\mathbf{M}^+) = 1$, we can directly figure out $a = 1$ by substituting $x = 1$ into Eq. (5.5). Then provided with the observed similarity set $\{x_1, x_2, \cdots, x_k\}$, we can estimate $b = \frac{\sum_1^k x_i F(x_i|\mathbf{M}^+)}{\sum_1^k x_i^2}$ via the least square method, where $F(x_i|\mathbf{M}^+)$ is calculated by Eq. (5.3). We call the CCF estimation by Eq. 5.5 as *powerCCF*. The time complexity of powerCCF is $O(k)$, where k is the number of training examples (see analysis in Appendix A.3).

5.4.3 Learn Threshold

Given a rule function f and an unlabeled instance pair (n, n') with similarity x and dissimilarity $y = 1 - x$, we can now classify (n, n') using a instance-matching decision rule that is defined based on the comparison of the matching certainty $F(x, \mathbf{M}^+)$ and

the non-matching certainty $F(1-x,\mathbf{M}^-)$, as follows:

$$(n,n') \in \begin{cases} \mathbf{M}^+ \text{ if } F(x,\mathbf{M}^+) \geq F(1-x,\mathbf{M}^-) \\ \mathbf{M}^- \text{ else} \end{cases} \tag{5.6}$$

This decision rule indicates that, if the matching certainty is greater than the non-matching certainty, (n,n') is assigned to $\mathbf{M}^+$, and vice versa. By substituting Eq. (5.1) into formula (5.6), the previous decision rule can finally be stated via CCF as follows:

$$(n,n') \in \begin{cases} \mathbf{M}^+ \text{ if } \frac{F(x|\mathbf{M}^+)}{F(1-x|\mathbf{M}^-)} \geq \frac{q(\mathbf{M}^-)}{q(\mathbf{M}^+)} \\ \mathbf{M}^- \text{ else} \end{cases} \tag{5.7}$$

where the ratio $\frac{F(x|\mathbf{M}^+)}{F(1-x|\mathbf{M}^-)}$ can be calculated by either interCCF or powerCCF, and the *prior ratio* $\frac{q(\mathbf{M}^-)}{q(\mathbf{M}^+)}$ is required as a parameter, which can be simply estimated from the fractions of the training examples in each of the classes. In experiments, we will show that our technique can achieve high quality of the result without fine-tuning of this parameter, even if the prior ratio is set with a large range of different values.

However, the cost of executing the decision rule in Eq. 5.7 is still expensive, since the ratio $\frac{F(x|\mathbf{M}^+)}{F(1-x|\mathbf{M}^-)}$ has to be repeatedly calculated for every unlabeled instance pair. A simpler manner to make a decision according to Eq. 5.7 is to figure out the decision boundary θ, i.e. the threshold, so that any instance pair that has similarity greater than θ must be assigned to $\mathbf{M}^+$. In the example of Fig. 5.2, it corresponds to finding the value of x shown by the vertical dotted line.

Proposition 5.1. *The threshold θ is calculated as the solution of the equation as follows:*

$$\frac{F(x|\mathbf{M}^+)}{F(1-x|\mathbf{M}^-)} = \frac{q(\mathbf{M}^-)}{q(\mathbf{M}^+)} \tag{5.8}$$

Proof. The ratio $\frac{F(x|\mathbf{M}^+)}{F(1-x|\mathbf{M}^-)}$ is a monotonically increasing function, since $F(x|\mathbf{M}^+)$ and $F(1-x|\mathbf{M}^-)$ are monotonically increasing and monotonically decreasing respectively. Therefore, as the solution of Eq.5.8, θ must be the decision boundary for Eq. 5.7, since for any similarity value $x \geq \theta$ there must be $\frac{F(x|\mathbf{M}^+)}{F(1-x|\mathbf{M}^-)} \geq \frac{q(\mathbf{M}^-)}{q(\mathbf{M}^+)}$, and vice versa. □

Alg. 5 shows the algorithm to find the threshold. Since it is difficult to directly calculate the solution of Eq. (5.8), Alg. 5 searches for an approximate solution $\bar{\theta}$ that satisfies $|\theta - \bar{\theta}| < \epsilon$, where θ is supposed to be the exact solution of Eq. (5.8) and ϵ is the *difference restriction* that restricts the difference between the exact and the approximate solutions. In reality, we can set a relative small value of the difference restriction to

guarantee that instance matching result will not be affected by it. For example, when the difference restriction is set to 0.001, an approximate threshold $\bar{\theta} = \theta \pm 0.001$ may result in the same instance-matching result as the exact threshold θ.

We apply a recursive binary search algorithm to find the approximate threshold $\bar{\theta}$. The algorithm returns if the approximate threshold equals to the exact threshold, or the difference between them is less than the pre-defined difference restriction ϵ (line 2). The complexity of Alg. 5 is $O(\log \frac{1}{\epsilon} \log k)$ for interCCF and $O(\log \frac{1}{\epsilon})$ for powerCCF, where k is the number of training examples (see analysis in Appendix A.3).

Algorithm 5: Learn Threshold

Input: lower bound x_l, upper bound x_u, difference ϵ, estimated distribution $F(x|\mathbf{M}^+)$ and $F(1-x|\mathbf{M}^-)$

Result: Estimated threshold $\bar{\theta}$

1 $\bar{\theta} := \frac{(x_l + x_u)}{2}$;

2 **if** $\frac{F(\bar{\theta}|\mathbf{M}^+)}{F(1-\bar{\theta}|\mathbf{M}^-)} = \frac{q(\mathbf{M}^-)}{q(\mathbf{M}^+)}$ *or* $x_u - x_l < \epsilon$ **then**

3   **return** $\bar{\theta}$;

4 **else if** $\frac{F(\bar{\theta}|\mathbf{M}^+)}{F(1-\bar{\theta}|\mathbf{M}^-)} > \frac{q(\mathbf{M}^-)}{q(\mathbf{M}^+)}$ **then**

5     $x_u := \bar{\theta}$;

6 **else**

7     $x_l := \bar{\theta}$;

8 **return** $LearnThreshold(x_l, x_u, \epsilon, F(x|\mathbf{M}^+), F(1-x|\mathbf{M}^-))$;

5.4.4 Evaluate mIR Candidate

For a given rule function f, we can now generate a mIR candidate $\lambda(f, \theta)$ by combining f with the learned threshold θ. As discussed, mIR candidates can be directly evaluated by a pre-defined objective function, e.g. the F-measure. Then the best-evaluated mIR can be selected as the result. However, since it is required to enumerate all the training examples to calculate the matches, the cost of executing the objective function is expensive. In this section, we introduce the method for a fast evaluation of mIR candidates.

From the view of probability, the calculated mIR candidate can be evaluated by the probability of identifying the true positives. The higher this probability is, the better the mIR candidate is evaluated.

Since all the instance pairs with similarities in the interval $[\theta, 1]$ are considered as matches, the probability of identifying the true positives is the joint probability of a instance pair (n, n') being a match and the similarity x of (n, n') is in the interval $[\theta, 1]$, as follows:

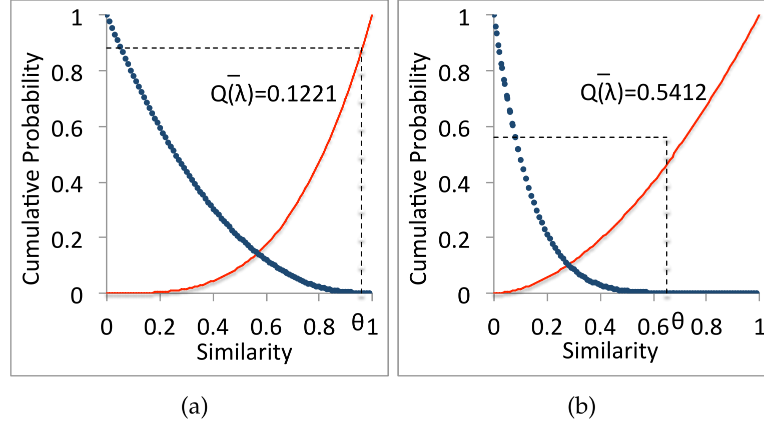


Figure 5.2: PowerCCF estimation for matching certainty $F(x|\mathbf{M}^{+})$ (solid line) and non-matching certainty $F(1-x|\mathbf{M}^{-})$ (dotted line), where x is similarity. Fig.(a) and (b) illustrate the certainty distributions that refer to $f_1 = \frac{Jaccard(\texttt{Title})+QGram(\texttt{Manufacturer})}{2}$ and $f_2 = \frac{Jaccard(\texttt{Title})+Cosine(\texttt{Description})}{2}$ respectively.

$$Pr(\theta \le x \le 1, \mathbf{M}^{+}) = F(1, \mathbf{M}^{+}) - F(\theta, \mathbf{M}^{+}) \tag{5.9}$$

Then substitute Eq. 5.1 into Eq. 5.9, the probability to identify true positives can be expressed as:

$$Pr(\theta \le x \le 1, \mathbf{M}^{+}) = q(\mathbf{M}^{+})(1 - F(\theta|\mathbf{M}^{+})) \tag{5.10}$$

Considering the prior $q(\mathbf{M}^{+})$ is the same for all mIRs, a mIR is evaluated as follows:

Proposition 5.2. *The quality of a given mIR $\lambda(f,\theta)$ is evaluated by $\bar{Q}(\lambda) = 1 - F(\theta|\mathbf{M}^{+})$. The higher $\bar{Q}(\lambda)$, the higher quality of $\lambda(f,\theta)$.*

The time complexity of mIR evaluation is $O(\ln k)$ for the interCCF and is $O(1)$ for the powerCCF, where k is the number of training examples (see analysis in Appendix A.3).

Example 5.5. *Taking rule function $f_1 = \frac{Jaccard(\texttt{Title})+QGram(\texttt{Manufacturer})}{2}$ for a example. Suppose we have already estimated $F_1(x|\mathbf{M}^{+}) = x^{3.39}$ and $F_1(y|\mathbf{M}^{-}) = y^{2.33}$ by powerCCF, where x is the value of similarity and y is the value of dissimilarity. Note $y = 1-x$, we can rewrite $F_1(y|\mathbf{M}^{-})$ as $F_1(1-x|\mathbf{M}^{-}) = (1-x)^{2.33}$. The curves of $F_1(x|\mathbf{M}^{+})$ and $F_1(1-x|\mathbf{M}^{-})$ are illustrated in Fig. 5.2(a). Then suppose the prior ratio $\frac{q(\mathbf{M}^{-})}{q(\mathbf{M}^{+})} = 600$, we can calculate the threshold $\theta = 0.9623 \pm 0.0001$ by Alg. 5 with a difference restriction $\epsilon = 0.0001$. Finally we get a mIR $\lambda_1(f_1, 0.9623)$, whose quality is evaluated as $\bar{Q}(\lambda_1) = 1 - F_1(0.9623|\mathbf{M}^{+}) = 1 - 0.9623^{3.39} = 0.1221$.*

Similarly, for the rule function $f_2 = \frac{Jaccard(\texttt{Title})+Cosine(\texttt{Description})}{2}$, *we have* $F_2(x|\mathbf{M}^+) = x^{1.78}$ *and* $F_2(1-x|\mathbf{M}^-) = (1-x)^{6.93}$, *which are plotted in Fig. 5.2(b). Then we can learn the corresponding mIR as* $\lambda_2(f_2, 0.6455)$, *whose quality is evaluated as* $\bar{Q}(\lambda_2) = 1 - F_2(0.6455|\mathbf{M}^+) = 1 - 0.6455^{1.78} = 0.5412$.

By comparing $\bar{Q}(\lambda_1)$ *and* $\bar{Q}(\lambda_2)$, λ_2 *is selected as the mIR with better quality.*

5.4.5 Learn Single mIR

In this section, we introduce the algorithm to learn only one mIR, which searches the best one from a significantly smaller set of mIR candidates compared with existing aIR-based approaches. More specifically, let l, m and k be the number of attributes, similarity functions, and training examples respectively, the total number of mIR candidates is m^l in our work, while the total number of aIR candidates is $(m \cdot k)^l$ in state-of-the-art aIR-based approaches [24, 119](see details in Appendix A.1 and Appendix A.2). For example, assuming an instance-matching problem with 4 attributes, 20 similarity, and 200 training examples, there would be 20^4 mIR candidates in our method, but $4,000^4$ aIR candidates in current aIR-based approaches.

A brute-effort algorithm is to enumerate all candidates and then select the one that maximizes the evaluation score. However even for the relative small size of candidates, this approach is still expensive. As shown in algorithm 6 we propose a hill-climbing algorithm to search the best mIR candidate, which mainly consists of two steps: mIR initialization and mIR optimization.

Step 1-mIR initialization: The algorithm starts from the mIR with the rule function that is composed of all attributes and for each attribute, the best-performed similarity function. Given two rule functions $f_i = g_i(a)$ and $f_j = g_j(a)$, if $\bar{Q}(\lambda(f_i, \theta_i))$ is greater than $\bar{Q}(\lambda(f_j, \theta_j))$, then the similarity function g_i is said to perform better than g_j on attribute a. For each attribute a_i, Alg. 6 (line 1-13) sort all similarity functions, that are stored in G_i, in the descending order, the best one at first and the worst one at last. The array I is used as an index to record the current selection of similarity function for each attribute. For example, when $I[2] = 3$, we bind attribute a_2 with the third similarity function in G_2[1]. Then the function `ConstructRulefunction` can construct a rule function by combining the attributes with the selected similarity function according to I(Alg. 7). At beginning, all the values in I are initialized with 1, so that we can select the best-performed similarity function for each attribute to initialize mIR. Considering a mIR may *not* include all attributes, we add a *null* to the end of G_i to allow the attribute a_i being ignored when it is combined with similarity function *null*.

Example 5.6. *Considering two similarity functions* $g_1 = Jaccard$ *and* $g_2 = QGram$, *and two attributes* $a_1 = \texttt{Title}$ *and* $a_2 = \texttt{Manufacturer}$. *After sorting similarity functions for each attribute, we have* $G_1 = \{Jaccard, QGram, null\}$ *and* $G_2 = \{QGram, Jaccard, null\}$,

[1]Note we use 1 as the index of the first element in an arraly.

which means Jaccard performs the best on `Title`*, and QGram performs the best on* `Manufacturer`*. In the mIR initialization step, we have $I = \{1,1\}$ such that the top similarity functions in G_1 and G_2 are combined with attributes a_1 and a_2 respectively. In this way, we get the initial rule function $f = \frac{Jaccard(\texttt{Title})+QGram(\texttt{Manufacturer})}{2}$.*

Moreover, attribute a_2 is ignored if we set $I = \{2,3\}$, since a_2 is combined with $null$, the third similarity function in G_2. In this way, we get a rule function $f' = QGram(\texttt{Title})$ by combining the second similarity function in G_1 with a_1.

step 2-mIR optimization: We adopt a hill-climbing algorithm to search the best mIR. Intuitively, starting from the initial mIR, we adjust the similarity function for each attribute for achieving a higher evaluation score $\bar{Q}$. We chose one attribute a_i in each iteration (iteration in line 15-27), and construct different rule functions by associating a_i with each of the similarity functions in G_i (iteration in line 18-27 of Alg. 6). We then evaluate the mIR candidate that is created according to each rule function (line 19-23 of Alg. 6), and finally record the one that achieves better evaluation score $\bar{Q}$ (line 24-27 of Alg. 6). The iteration (iteration in line 15-27 of Alg. 6) terminates if the evaluation score cannot be improved any more.

Example 5.7. *Assuming in Step 1, we have $I = \{1,1\}$, $G_1 = \{Jaccard, QGram, null\}$ and $G_2 = \{QGram, Jaccard, null\}$ for attribute $a_1 =$ `Title` and $a_2 =$ `Manufacturer` respectively. In the first iteration we choose the attribute a_1 and change the value of the first element in index I to $I = \{2,1\}$ and $I = \{3,1\}$ to construct two rule functions $f = \frac{QGram(\texttt{Title})+QGram(\texttt{Manufacturer})}{2}$ and $f' = QGram(\texttt{Manufacturer})$. We then create the mIR candidates according to f and f', and select the one that is better evaluated. In the next iteration, we will select the other attribute a_2, and repeat above process again. This process will continue until the evaluation score cannot be improved any more.*

The time complexity of Alg. 6 is $O(tlm \log \frac{1}{\epsilon} \log k)$ for interCCF, and $O(tlmk)$ for powerCCF, where t, l, m, and k are the number of iteration, attributes, similarity functions, and training examples respectively, and ϵ is the difference restriction for threshold learning (see analysis in Appendix A.3).

5.4.6 Learn a Set of mIRs

Since one mIR may not cover all the positive examples, more mIR have to be used to maximize the objective function score. As discussed in previous research[24], the problem to learn a set of mIRs which maximize the objective function is NP-hard. To deal with this problem, we apply the similar greedy algorithm as previous solution[24]. The algorithm greedily selects the best mIR each time, removes the positive and negative examples that are identified by the learned mIR, and then repeats this procedure on the remained examples until the objective function score cannot be improved anymore.

Algorithm 6: Learn single mIR

```
   Input: difference ε, a set A of attributes, a set G of similarity functions
   Result: best mIR λ_best
1  G = {∅,∅,···};
2  for i := 1 to |A| do
3      G_i := ∅;
4      G_i := G_i ∪ G;
5      Sort(G_i);
6      G_i := G_i ∪ {null};
7      G[i] := G_i;
8  I := {1,1,···};
9  f := ConstructRuleFunction(A,G,I);
10 Estimate F(x|M+) and F(1 − x|M−);
11 θ := LearnThreshold(0,1,ε,F(x|M+),F(1 − x|M−));
12 Q_best := 1 − F(θ|M+);
13 λ_best := λ(f,θ);
14 improved := true;
15 while improved = true do
16     improved := false;
17     for i := 1 to |A| do
18         for j := 1 to |G| + 1 do
19             I[i] := I[i] + 1;
20             f := ConstructRuleFunction(A,G,I);
21             Estimate F(x|M+) and F(1 − x|M−);
22             θ := LearnThreshold(0,1,ε,F(x|M+) ,F(1 − x|M−));
23             Q̄ := 1 − F(θ|M+);
24             if Q̄ > Q_best then
25                 λ_best := λ(f,θ);
26                 Q_best := Q̄;
27                 improved := true;
28 return λ_best;
```

5.5 Algorithm for Executing mIR

As discussed before, aIR can be efficiently executed mainly because it involves less similarity calculations. It does not have to test all the similarity function predicates, if any one of the predicates in the rule returns false. Compared to aIR, since we calculate the similarity of two instances in a mIR as the average similarity of a set of attributes,

Algorithm 7: Construct Rule Function

Input: difference ϵ, a set A of attributes, a set G of similarity functions, Index I
Result: best mIR λ_{best}

```
1 N := 0;
2 for i := 1 to |A| do
3     a_i := A[i];
4     G_i := G[i];
5     g_i := G_i[I[i]];
6     if g_i ≠ null then
7         N := N + 1;
8 return f := (Σ_{i:=0}^{|A|} g_i(a_i)) / N;
```

the mIR may not be efficiently executed if all the corresponding similarities have to be calculated. However, we observe that some similarity calculations in mIR are unnecessary, since it is possible to make a decision only based on a part of similarities.

Let $\lambda : \frac{\sum_{i=1}^{d} g_i(a_i)}{d} \geq \theta$ be a mIR that involves d similarity functions, and assume we have already finished computing j ($j < d$) similarity functions for a pair of instances (n, n'). The original mIR can be rewritten as follows:

$$\lambda : \sum_{i=1}^{j} g_i(a_i) + \sum_{i=j+1}^{d} g_i(a_i) \geq d\theta \tag{5.11}$$

where $sum_c = \sum_{i=1}^{j} g_i(a_i)$ is the sum of similarities that have already been computed, $sum_u = \sum_{i=j+1}^{d} g_i(a_i)$ is the sum of similarities that have not been calculated, and $d\theta$ is the threshold for the sum of all similarities. We observed that it is possible to make a decision only based on sum_c in two circumstances:

1. If $sum_c \geq d\theta$, (n, n') is classified to $\mathbf{M}^+$. Obviously, if we know the mIR has already been satisfied, it is not necessary to calculated sum_u any more (line 4-5 of Alg.8).

2. If $sum_c + d - j < d\theta$, (n, n') is classified to $\mathbf{M}^-$. Assuming each of the similarity functions in sum_u can achieve the highest value of 1, the maximal value of sum_u is $d - j$. We can then estimate that the maximal sum of similarities for (n, n') is $sum_c + d - j$. Based on this estimation, we can directly classify (n, n') to $\mathbf{M}^-$ when $sum_c + d - j$ is less than the threshold $d\theta$ (line 6-7 of Alg.8).

Algorithm 8: Executing mIR

Input: mIR $\lambda = \frac{\sum_{i=1}^{d} g_i(a_i)}{d} \geq \theta$, instance pairs (n, n')
Result: whether (n, n') satisfies the mIR

```
1 sum := 0;
2 for j:=1 to d do
3     sum+ = g_j(n[a_i], n'[a_j]);
4     if sum ≥ dθ then
5         return true;
6     if sum + d − j < dθ then
7         return false;
8 return sum ≥ dθ;
```

5.6 Experimental Evaluation

To study the proposed solution, we employ a recent instance matching benchmark [66] that captures data from enterprise databases as well as synthetic data Restaurant. Compared against the state-of-the-art approaches *SiFi* [119] and *SVM* [21], our approach greatly improves the result quality and efficiency.

5.6.1 Dataset and Matching Task

We now briefly describe the datasets which cover the restaurant, bibliography and products domains. Tabel 5.3 provides an overview of the datasets.

Table 5.3: For each dataset pair: number of instances, all matches $\mathbf{M}^+$, and all non-matches $\mathbf{M}^-$.

Task	Datasets		Mapping Candidates	
	Dataset1	Dataset2	$\mathbf{M}^+$	$\mathbf{M}^-$
Rest	864		112	5,270
AB	1,081	1,092	1,097	7,040
AD	2,616	2,294	2,224	3,140

Restaurant(Rest). The restaurant dataset is available at OAEI 2010[2]. We manually removed the attribute `telephone` from the dataset because the dataset with it is too easy for instance matching task to be useful for comparing algorithms.

[2] `http://oaei.ontologymatching.org/2010`

ACM-DBLP (AD). These datasets in the benchmark [66] include well-structured bibliographic data from DBLP and the ACM digital library. This one is manually created and thus, is of higher quality among all datasets. As a result, the matching task represented by this is of low difficulty.

Abt-Buy (AB). This matching task in the benchmark [66] is performed between instances of the product dataset from `http://abt.com` and `http://buy.com`. This dataset contains the most noises in the attribute values. Therefore, the matching task for this is the most difficult.

We adopt a preprocessing step, named blocking [40], to select candidate instance pairs that are most likely to be the same, as shown in Tbl 5.3. In the experiments, we classify these candidates to either $\mathbf{M}^+$ of $\mathbf{M}^-$, and compare the efficiency and effectiveness among different approaches.

5.6.2 Experimental Setting

System. In the experiments, we select two baseline methods, `SiFi` [119] and `SVM` (using LIBSVM libarary [21]). **Sifi** is a recent aIR learning approach that learns similarity functions and thresholds from positive and negative examples. It requires attributes to be manually set for each rule. Also, it requires the similarity functions and thresholds to be manually set for some attributes so that they can be learned for other attributes. `SVM` requires features as similarities of all attributes calculated by all similarity functions. We compare these approaches against our solutions using interCCF and powerCCF, called **interMIR** and **powerMIR** respectively.

Similarity. We selected 20 similarity functions in the experiments provided by an open-source Java package `SimMetrics`[3]. In case there is null value when comparing two values, we suppose the similarity is 0.

All experiments were run on a computer with one 2.4GHz Intel Core 2 Duo CPU, using 4GB of main memory, running Linux with kernel version 2.6.18.

5.6.3 Efficiency

We first compare against baseline approaches in term of efficiency. Table 5.4 shows the training and testing time for all four approaches, which use all the mapping candidates as training and testing data. We can see that `powerMIR` and `interMIR` run the fastest in the training process. For example, on dataset `AB`, `powerMIR` requires only 59 seconds for training, which is only 13% of the time of `SiFi` and half of the time of `SVM`. `interMIR` slightly outperforms `powerMIR` because it is more efficient in CCF estimation (see time complexity analysis in Appendix A.3). The cost of `SiFi` is the most expensive in training because of the large number of aIR candidates. Even

[3] `http://www.dcs.shef.ac.uk/~sam/simmetrics.html`

though SiFi proposed to eliminate candidates with redundant similarity functions and thresholds, the process of removing redundancy itself is still expensive in reality.

In the testing process, SVM is clearly the worst comparing to other methods. For example, powerMIR only spends 5 seconds on the task of AD, while SVM costs 78 seconds. As a general model, SVM spends most of the time on similarity calculations, since it require similarities of all attributes that are calculated by all similarity functions as features. Especially, because we avoid unnecessary similarity calculations in mIR execution, our approach achieves very similar testing time as the aIR approach SiFi.

Compared to the approaches in the benchmark [66], powerMIR and interMIR are more efficient than the learning-based approaches, but cost more matching time than the non-learning approaches that compare only one or two attributes. This is because powerMIR and interMIR usually compare more than one attribute on executing one instance matching rule. And they also match instances in a multi-iteration manner, where multiple rules are executed to ensure not missing any true matches. Therefore, powerMIR and interMIR need more attribute comparisons than non-learning approaches in the benchmark.

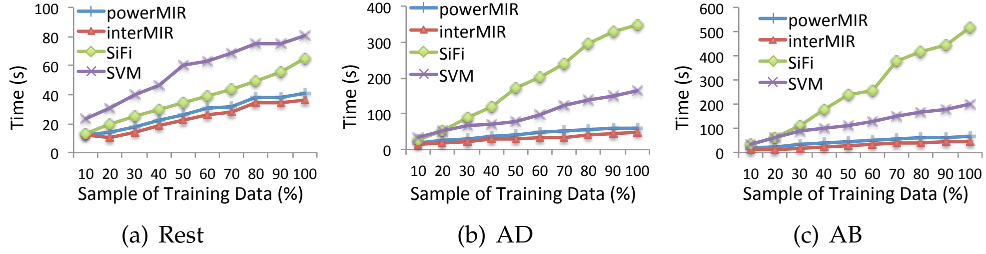
Table 5.4: Performance of training and testing in seconds.

	Rest		AD		AB	
	train	test	train	test	train	test
powerMIR	37	3	55	4	59	8
interMIR	**36**	3	**50**	4	**57**	8
SiFi	54	4	342	4	518	9
SVM	59	60	85	78	109	103

We further vary the proportion of training data and plot the total running time of different methods in Fig. 5.3. The running time of our techniques is the stablest when the size of the training data increases, mainly resulting from the number of mIR candidates is irrelevant to the number of training examples. However, since the number of candidates in SiFi increases fast with the increase of training examples, the time of SiFi increases the fastest.

5.6.4 Effectiveness

Table 5.5 shows the results of effectiveness based on the average of F-measure in 10 runs. For each run, we randomly select 30% of the mapping candidates as training data and use the rest as testing data. Our techniques achieve the best result on Rest. and AD. Note because we remove the attribute Telephone from Rest, this task becomes more difficult resulting in the worse result of SiFi compared to the result reported in the original paper. We also compare our solutions with SiFi(auto),



(a) Rest (b) AD (c) AB

Figure 5.3: Comparison for the total running time with different proportion of training data.

in which SiFi automatically learn the rules without manual tuning of parameters, such as the pre-defined attributes and similarity functions. We can see that its results becomes much worse than our approaches as well as SiFi with fine-tuning of parameters.

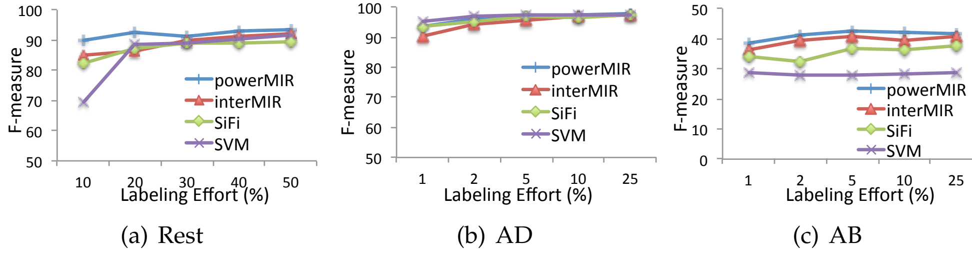
Because of the noisy data, AB is the most difficult task and a non-linear classification problem. So SVM performs the best on AB comparing to other approaches that compare only one or two attributes. We also observe the similar result in the benchmark [66] where FEBRL-SVM [27] is the best system with F-measure 71.3. In sum, instance matching is a very data-specific problem. In other words, there are not best approach for all types of data. For example, the task AD is very much easier than AB because it involves the datasets that are well maintained. Therefore, both powerMIR and the FEBRL-SVM can achieve high F-measure that is above 97.0. However, because powerMIR costs only 4 seconds which is much more efficient than FEBRL-SVM that spends 20 seconds, powerMIR should be more suitable for the task AD.

Table 5.5: Effectiveness of instance matching in terms of F-measure.

	Rest	AD	AB
powerMIR	**93.16**	**97.47**	41.69
interMIR	90.33	97.21	40.87
SiFi	88.92	96.20	37.52
SiFi(auto)	70.55	95.32	33.49
SVM	90.28	97.30	**51.20**

We further compare the F-measure against the baselines provided with different labeling effort, as shown in Fig. 5.4. The proportion of labeling effort varies between 10% and 50% in Rest, and between 1% and 25% in AD and AB. For AD, all the approaches can soon achieve stable results for a small size (2%) of training data. And in Rest and AB, powerMIR is managed to maintain high quality of results for all different labeling efforts, while SVM performs the worst given small size of training

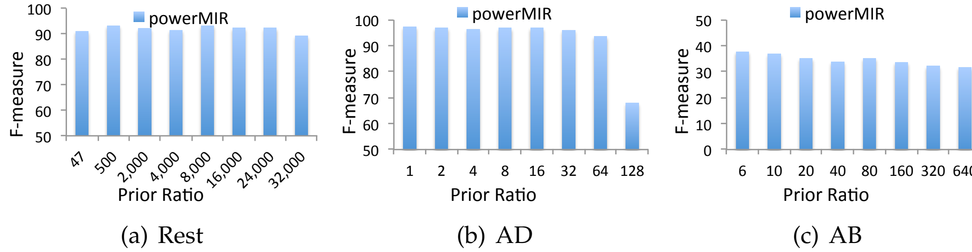
data. We also observe that, `interMIR` is worse than `powerMIR` when it is provided with less training data. As discussed before, it is because `interCCF` leverages linear interpolation, which may deviate form the true values if the real CCF does not follow a linear function. However, when there are more training data available, this estimation becomes more accurate leading to `interMIR` achieving almost the same result as `powerMIR`. Therefore, we can directly apply `interMIR` when we are not clear about the data distribution, but have large amount of training data available.



(a) Rest (b) AD (c) AB

Figure 5.4: Evaluation result for different labeling effort

5.6.5 Parameter Sensitiveness

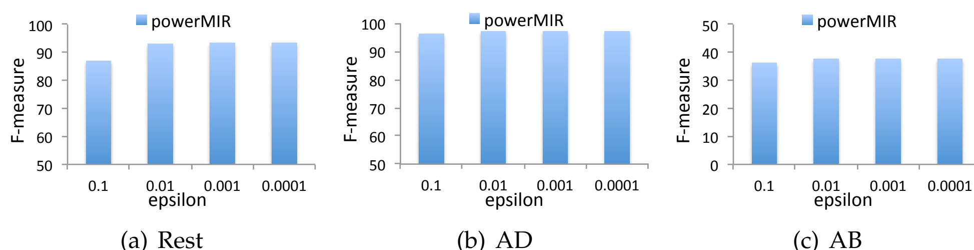


(a) Rest (b) AD (c) AB

Figure 5.5: Influence of prior ratio

We now analyze the parameters. Throughout the paper, we have two parameters: the prior ratio $\frac{q(\mathbf{M}^-)}{q(\mathbf{M}^+)}$ for threshold learning according to Eq. (5.8); and ϵ as the difference restriction between the approximate and the real threshold in Alg. 5. We show that our approach can achieve stable result without fine-tuning of these parameters.

Figure 5.5 shows the result for different values of prior ratio. The results are stable when the prior ratio is set between 47 and 24,000 for `Rest` (the real prior ratio is 47), between 1 and 64 for `AD` (the real prior ratio is 1.4), and between 6 and 160 for `AB` (the real prior ratio is 6.4). Basically, a higher prior ratio determines a higher



(a) Rest (b) AD (c) AB

Figure 5.6: Influence of ϵ

threshold. Therefore, a mIR with higher threshold may result in higher precision but less identified matches. However, since we keep adding the best evaluated mIR into the rules until the objective score cannot be improved, a higher prior ratio may result in a larger set of higher-precision mIRs. In this way, a matching instance pair that fails to be identified by one mIR can be identified by another instead. On the other hand, since a lower prior ratio may result in lower thresholds, the prior ratio that is lower than the real value may leads to lower precision but higher recall. Therefore, an applicable strategy is to set the prior ratio with the value higher than that estimated from the training data, so that we can achieve stable results.

Figure 5.6 shows the results for different values of ϵ. Intuitively, the smaller ϵ, the higher accuracy. However, in reality it makes no differences if ϵ is set less than 0.01 for all the matching tasks.

5.7 Related Work

Instance matching (also known as entity resolution, entity co-reference, or record linkage), is about finding instances that refer to the same object. A high quality of instance matching result require the fine tunning of parameters selection, which are usually achieved with the help of machine learning techniques [100, 113]. Here, we provide a broader overview of machine-learning-based solutions for the instance matching problem.

Instance matching problem can be solved with the help of standard machine learning techniques. Common to all approaches is the observation that instance matching can be formulated as a standard classification task, where instances are classified as matching or non-matching [44]. There are supervised machine learning approaches, which use decision trees [112, 113], Bayes decision rule [44] or SVM [14] to train classifier from the provided training examples. Among these techniques, SVM is known as the most effective approach.

Probabilistic graphical models were also used for unsupervised instance matching.

For instance, latent binary variables in a hierarchical graphical model were used to model whether attributes match or not [97]. Using the Latent Dirichlet Allocation model, both instances and groups of instances were captured through latent variables [11]. Similar to that, the Dirichlet Process was also employed to model the number of clusters and instances [59].

In contrast to those general machine learning techniques, rule-based approaches are designed for specific instance matching problems [3]. The instance-matching rules have advantages that they are explainable, and can be efficiently executed because of less similarity calculations. Existing rule-based approaches usually involve four subtasks to determine (1) (combinations of) attributes [16, 24, 57, 75, 83, 119, 120], and for each of them, (2) the value representation function [16, 24, 57, 75, 83, 113], (3) the similarity functions [24, 119] and (4) the similarity thresholds [24, 112, 113, 119]. Recently Chaudhuri et al. [24] propose an algorithm for these four tasks based on a given set of positive and negative examples. Wang et al. [119] further improve the learning efficiency by eliminating rule candidates that are composed of redundant similarity functions and redundant thresholds. In this chapter, we focus on task (1) (2) and (4). We compared our work with SiFi, an advanced aIR learning approach, and SVM, which adopts a linear combination of similarity functions and attributes that is similar to the form of mIR used in this chapter. We show that our approach, when be compared to SiFi, can learn the rules much faster and require less parameter tuning, and can be executed much faster than SVM.

5.8 Conclusion

For the problem of instance matching, we proposed an efficient approach to learn instance-matching rules by estimating matching and non-matching certainties. Comparing to state-of-the-art instance matching approaches, our solution greatly improves the efficiency. At the same time, it is also effective, that when compared against SVM, that is currently known as the most effective approach for instance matching, we gain comparable or even better quality results in term of F-measure. Moreover, the approach can also achieve stable results when the parameters are set with a large range of different values. As future work, we aim to learn the weights of attributes, and the value representation functions for instance-matching rules.

Chapter 6

Filtering: Effective Parameter-free Boolean Instance Matching

State-of-the-art instance matching methods use training data to learn combinations of attributes, similarity functions and thresholds, called instance matching rules, for finding matches. The learning of complex rules with thresholds is however complex and sensitive to training data. In this chapter, we explore a different avenue, proposing an approach that does not use thresholds but more simple *boolean* similarity functions. We show that the simple boolean nature of the employed rules allows for a *parameter-free* learning approach. For high effectiveness, we propose to incorporate fine-grained word-level evidences into rule learning. That is, instead of capturing the similarity of entire attribute values in the rules, our approach employes words extracted from attribute values. Using benchmark matching tasks, we show the proposed solution greatly outperforms state-of-the-art approaches in terms of result quality and most importantly, is not sensitive to the choice of training data and parameters.

Outline The introduction and contribution are provided in Sec. 6.1 and Sec. 6.2. We provide an overview in Sec. 6.3. The learning of dissimilarity evidences is presented in Sec. 6.4. An implementation of our approach is discussed in Sec. 6.5. We present experiments in Sec. 6.6, related works in Sec. 6.7 and conclusions in Sec. 6.8.

6.1 Introduction

As discussed in previous chapter, state-of-the-art approaches employ instance-matching rules to solve the instance matching problem [24, 119]. For instance, with the rule "two product instances are the same if they have similar values for `Title` and `Manufacturer`" for the data in Tab. 5.1, the instances n_1 and n_2 can be considered as the same. However, a big challenge is actually to determine how similar is similar [119]. To address this, existing approaches incorporate *thresholds* into the rules so that instances and their attribute values are only considered similar if their similarity is higher than a threshold. Using different thresholds for different combinations of attributes and similarity functions greatly improves the quality of instance

matching. However, the search space for learning these parameters is very complex and is sensitive to the training data.

In this chapter, we explore a different avenue, proposing an instance matching approach that relies only on relatively simple *Boolean similarity functions*. Existing approaches use thresholds to obtain more "sophisticated" evidences because they compare instances at the more coarse-grained level of attribute values. As an example, one evidence might be "the similarity on `Title` for the two instances n_i and n_j is greater than 0.9". Our Boolean approach extracts evidences at the more fine-grained level of words (tokens in general) found in attribute values. From the words `13.3`, `Apple`, `ASUS` etc. found in the `Title` values, our approach learns what we call *word-level dissimilarity evidences*, such as "`Apple` and `ASUS` are dissimilar words, one is in the `Title` of n_i and the other is in the `Title` of n_j". It is a dissimilarity evidence because it leads to the inference that n_i and n_j are not the same (are non-matches). We show that using this type of evidences has the following merits: (1) due to their Boolean nature, the learning of these evidences is more simple, i.e. does not require parameters and is not sensitive to training data. (2) At the word level, a large number of these evidences can be learned to identify non-matches (high recall). (3) Also due to the use of fine-grained words, the learned evidences are more discriminative in identifying non-matches (high precision).

While our focus is Boolean matching, we also discuss in the chapter how our Boolean approach can be combined with existing works that use thresholds. We show that when used as a Boolean filtering mechanism, our approach consistently improves the results of the underlying matching approach.

6.2 Research Question and Contribution

In this chapter, we address the following research question:

Research Question 4. *How to identify non-matching instance pairs by simple Boolean functions?*

This question is derived from the use of thresholds that may sensitive to the training data and parameters. Therefore, the instance matching results that are output based on the use of threshold, may still contain a large number of non-matches. We assume the simpler Boolean matching algorithm can solve this problem and hence state the following hypothesis:

Hypothesis 4.1. *Due to the Boolean nature of the word-level dissimilarity evidence, the learning of these evidences is more simple, i.e. does not require parameters and is not sensitive to training data. At the word level, the large-number evidences are more discriminative in identifying non-matches, which lead to effective Boolean instance matching.*

Regarding the above research question and hypothesis, we propose an effective parameter-free instance matching approach that relies on Boolean functions, providing three main contributions:

- **Boolean instance matching.** The approach we propose in this chapter is based on only simple Boolean similarity functions. The Boolean similarity function explores word co-occurrence based dissimilarity evidences (CDE) in the form of word co-occurrences, namely those pairs of words that when observed in an instance pair, render it as a non-match.
- **Parameter-free instance matching.** The approach we propose in this chapter is also less sensitive to training data. Provided with different samples of training data, the approach can achieve stable results.
- **Filtering based on thresholded instance matching.** We also discuss how the approach can be combined with existing thresholded instance matching methods. We show that our approach can further improve the results of thresholded instance matching methods by filtering out non-matching instance pairs.

Using benchmark matching tasks, we show our Boolean solution greatly outperforms state-of-the-art approaches in terms of result quality (up to 206.34% improvement in terms of F-measure), is comparable in terms of time performance and is superior in terms of sensitivity to training data and parameters.

6.3 Overview

We reuse the definition of data that is first given in Section 2.1. We reuse the data example that are provided in Chapter 5, assuming the data are of the same type as the result of typification, as shown in Tab. 5.1. A summary of the main notations used in this chapter is shown in Tab. 6.1

Table 6.1: A summary of notations used in this chapter.

Notation	Description
N	a set of instances
a	an attribute
$\mathbf{M}^+$, M^+, M'^+, M_i^+	all matches, examples, self-matches, matches in M_i
$\mathbf{C}$, C, C', C_i	Cart. product of words in $\mathbf{M}^+$, M^+, M'^+, M_i^+
$\mathbf{C}^+$, C^+, C'^+, C_i^+	word co-occurrences in $\mathbf{M}^+$, M^+, M'^+, M_i^+
$\mathbf{C}^-$, C^-, C'^-, C_i^-	CDEs for $\mathbf{M}^+$, M^+, M'^+, M_i^+
$W_{N[a]}$ ($W_{n[a]}$)	words in a value of all instances (of instance n)

6.3.1 Thresholded Instance Matching

Existing approaches utilize similarity functions for instance matching, which map a pair of attribute values $(n[a], n'[a])$ to a similarity score in $[0,1]$. We explicitly distinguish between these two types of similarity functions:

Definition 2.2 (Similarity Function)**.** *A* Boolean function $same : N[a] \times N[a] \rightarrow \{0,1\}$ *maps a pair of attribute values to the score* 0 *or* 1*. A* thresholded similarity function $sim : N[a] \times N[a] \rightarrow [0,1]$ *maps a pair of attribute values to a score in the range* $[0,1]$*. A larger similarity score indicates a higher similarity between two attribute values.*

Based on that, we further distinguish two types of instance-matching rules according to the types of similarity functions that are used:

Definition 6.1 (Instance-matching Rule)**.** *An* instance-matching rule Λ *is a conjunction of boolean predicates, i.e.* $\Lambda = \bigwedge \lambda_i$*, where* $\lambda_i : [0,1] \rightarrow \{true, false\}$*. Each* λ_i *is either based on*

- *a thresholded similarity function* sim_j *such that* $\lambda_i(sim_j(n[a_i], n'[a_i]), \theta_{i,j})$ *returns true iff* $sim_j(n[a_i], n'[a_i]) \geq \theta_{i,j}$ *and* $false$ *otherwise,*
- *or a boolean function* $same_j$ *such that* $\lambda_i(same_j(n[a_i], n'[a_i]))$ *returns true iff* $same(n[a_i], n'[a_i]) = 1$ *and* $false$ *otherwise.*

A pair of instances are considered as the same, called a match*, if it satisfies all the boolean predicates in the rule.*

State-of-the-art approaches employ thresholded similarity functions [24, 57, 119, 124]. Compared to the boolean outputs $\{0,1\}$, scores in the $[0,1]$ range produced by these approaches are more fine-grained and thus, help to further distinguish the quality of the matching candidates. However, we observe the following drawbacks:

Sensitivity to Training Data. Threshold candidates are generated from the similarities observed in the examples. Intuitively, the goal is to find a threshold that is higher than all similarity scores computed for negative examples and smaller than all scores computed for positive examples. The threshold learned this way is not only sensitive to differences in the training data sample provided as input but also the mix of positive and negative examples. Besides, existing approaches require at least some of the other parameters, i.e. the number of rules, the attributes and the similarity functions, to be selected and tuned manually. The learned thresholds are also sensitive to differences in the choice of these parameters.

Unstable Performance. Often, there is no optimal threshold that can perfectly separates positive from negative examples. Using Jaccard similarity and the data shown in Tab. 5.1 for instance, we obtain for `Description` the similarity scores 0.19 and

0.42 for (n_1,n_2) and (n_1,n_5), respectively. However, (n_1,n_2), which has a lower score, is actually correct while the other is incorrect. A threshold that identifies all candidates with a score higher than 0.19 as matches can deal with this case. However, it is too low for many other cases, i.e. would return many non-matches. In fact, many thresholds may exist that largely vary but are equally optimal w.r.t. the objective function used for learning. Each of them can provide good results for some cases but does not generalize over all cases. In other words, it is hard to find a threshold that provides stable performance.

6.3.2 Boolean Instance Matching.

We explore the use of simple boolean functions that are more easy in that learning does not require or is less sensitive to training data. In order to address the second drawback, i.e. to achieve good and stable performance, we employ two strategies. (1) Leveraging the more fine-grained level of words that are contained in attribute values, a large quantity of evidences can be learned. (2) Further, not only the initial set of training examples is used. We propose methods for training data enrichment such that in the end, the entire dataset is employed to derive evidences. In fact, we show that with these methods, our approach also performs well when only automatically computed pseudo-examples are used.

Two types of boolean functions are used in our approach. The first computes a similarity score based on boolean evidences for matching, i.e. *similarity evidences*. Examples for boolean similarity evidences used in literature are value overlap or substring match. We use this type to quickly find match candidates, a commonly employed pre-processing step as discussed in Chapter 4.

We then apply another type of boolean functions to the resulting candidates to filter out those that are non-matches. Since they are based on evidences for non-matching, i.e. *dissimilarity evidences*, we refer to this type as *dissimilarity function*. As an example, the fact that two given products are produced by different manufacturers can be taken as a dissimilarity evidence to infer that they are non-matches. Applying this type of functions to the candidate set is referred to as the *filtering* step:

Definition 2.7 (Filtering)**.** *Given a matching candidate set M,* filtering *returns a subset $M^+ = M \setminus M^-$, where M^- represents non-matching candidates computed by the dissimilarity function $\neg same$, i.e. $M^- = \{(n,n') \in M | \neg same(n[a_i],n'[a_i]) = 1\}$.*

The focus of this chapter lies in filtering these candidates based on fine-grained dissimilarity evidences captured at the level of words. Considering attribute values as a whole, few or no dissimilarity evidences can be derived. As an example, while it makes intuitive sense, values for `Manufacturer` actually cannot be used as dissimilarity evidences. The dissimilarity function based on that would miss the correct match (n_3,n_4) in Tab. 5.1 because n_3 and n_4 have different values for `Manufacturer`.

However, specific words in the values can serve as evidences. For instance, we define that `Apple` and `ASUS` form a dissimilarity evidence, while `Apple` and `Inc.` do not. Thus, given two instances with values for `Manufacturer` that contain `Apple` and `ASUS` as words, they will be correctly filtered as a non-match, while the match (n_3, n_4) is preserved because `Apple` and `Inc.` do not represent a dissimilarity evidence.

6.4 Learning Word-Level Dissimilarity Evidences

In this section, we first introduce evidences that are based on word-level co-occurrences. Then, we discuss the problems of learning them from (the possible lack of) training data and finally, our solution for these learning problems.

6.4.1 Word Co-occurrence Based Evidences

We find dissimilarity evidences in the form of word co-occurrences, namely those pairs of words that when observed in a instance pair, render it as a non-match. We capture this notion of word co-occurrences as follow:

Definition 6.2 (Word Co-occurrence)**.** *Given an attribute value pair* $(n[a], n'[a])$ *and their bags of words* $W_{n[a]}$ *and* $W_{n'[a]}$*, a word pair* $(w_i, w_j) \in W_{n[a]} \times W_{n'[a]}$ *co-occurs in* $(n[a], n'[a])$*, denoted by* $(w_i, w_j) \in (n[a], n'[a])$*, if* $w_i \in W_{n[a]}$ *and* $w_j \in W_{n'[a]}$*, or* $w_i \in W_{n'[a]}$ *and* $w_j \in W_{n[a]}$*. Let* M *be a set of instance pairs. For a given attribute* a*, the word pair* (w_i, w_j) *is said to co-occur in* M*,* $(w_i, w_j) \in M$*, if there exists an instance pair* $(n, n') \in M$ *such that* $(w_i, w_j) \in (n[a], n'[a])$*. We also use* $(w_i, w_j) \notin (n[a], n'[a])$ *(*$(w_i, w_j) \notin M$*) to denote that* (w_i, w_j) *does not co-occur in* $(n[a], n'[a])$ *(in* M*).*

Now we introduce the dissimilarity function that uses word pairs as dissimilarity evidences to filter non-matches:

Definition 6.3 (Diss. Evidence/Function)**.** *Let* $C = \{(w_i, w_j) \in W_{N[a]} \times W_{N[a]} | w_i \neq w_j\}$ *be all possible word pairs in the values of the attribute* a*, i.e. values of* a *for all instances* N *(denoted by* $N[a]$*).* $C^- \in C$ *is a set of word pairs representing dissimilarity evidences, called* co-occurrence based dissimilarity evidences *(CDE), such that given* C^-*, the function* $\neg same : N[a] \times N[a] \rightarrow \{0, 1\}$*, called* CDE-based dissimilarity function*, maps a pair of attribute value* $(n[a], n'[a])$ *to 1, if there exists a word pair* $(w_i, w_j) \in C^-$ *that co-occurs in* $(n[a], n'[a])$*, i.e.* $(w_i, w_j) \in (n[a], n'[a])$*, and 0 otherwise.*

Intuitively, this dissimilarity function determines a pair of values $(n[a], n'[a])$ to be a non-match when they contain a dissimilarity evidence (a word pair $(w_i, w_j) \in C^-$).

6.4.2 Learning CDE from Positive Examples

The CDE and functions introduced above are more easy to learn because they require only positive examples. This is because by Def. 6.3, positive examples representing matches must not contain any CDEs. In other words, CDEs cannot co-occur in positive examples. Further, we can even show that all words pairs that do not co-occur in positive examples must be CDEs such that for computing them, we only need to (1) find all word pairs co-occurring in positive examples, then (2) derive its complement set containing all those word pairs that do not co-occur in positive examples. That complement set must contain all CDEs. We establish the following theorem to enable the learning of CDEs:

Theorem 6.1. *Let* $\mathbf{M}^+$ *be the set of* all *matches,* $\mathbf{C}^+$ *be all word pairs that co-occur in* $\mathbf{M}^+$ *and* $\mathbf{C}^-$ *the set of all CDEs. If a word pair* (w_i, w_j) *is not in* $\mathbf{C}^+$*, it is a CDE, i.e.* $(w_i, w_j) \notin \mathbf{C}^+ \Rightarrow (w_i, w_j) \in \mathbf{C}^-$.

Proof. Given the word pair $(w_i, w_j) \notin \mathbf{C}^+$, we firstly show that all instance pairs containing (w_i, w_j) must be non-matches. Let $M = \{(n, n') | (w_i, w_j) \in (n[a], n'[a])\}$ be all the pairs of instances in which (w_i, w_j) co-occur. We prove $M \cap \mathbf{M}^+ = \emptyset$ by contradiction: assuming M contains some matches, i.e. $M \cap \mathbf{M}^+ \neq \emptyset$. Then, because $\mathbf{C}^+$ contains all the word pairs that co-occur in $\mathbf{M}^+$, $(w_i, w_j) \in (n, n')$ for all $(n, n') \in M \cap \mathbf{M}^+$, then $\mathbf{C}^+$ must contain (w_i, w_j), i.e. $(w_i, w_j) \in \mathbf{C}^+$. This however, cannot be satisfied because as given, $(w_i, w_j) \notin \mathbf{C}^+$.

Because all the instance pairs containing (w_i, w_j) are non-matches, i.e. $M \cap \mathbf{M}^+ = \emptyset$, (w_i, w_j) is a CDE and thus $(w_i, w_j) \in \mathbf{C}^-$. □

Previously, we use $\mathbf{M}^+$ to denote the set of all matches. It is used to define the concept of CDE and shall be computed through instance matching. When training examples are given, we have a subset of all matches. To facilitate the following discussion, we introduce M^+ to refer to such a subset of matches ($M^+ \subset \mathbf{M}^+$). Further, W_{M^+} denotes the bag of words model of the values of instances in M^+, $C = W_{M^+} \times W_{M^+}$ is the Cartesian product representing all possible pairs of words in W_{M^+}, $C^+ \in C$ denotes all word pairs that co-occur in M^+ and $C^- \in C$ are word pairs not co-occurring in M^+.

We employ a word pair graph (WPG) as an intuitive way to capture word pairs in positive examples and its complement set. Intuitively, two words co-occur in positive examples if they are connected by an edge in the WPG, and they do not if there are no edges connecting them. Then, a WPG can be defined as:

Definition 6.4 (Word Pair Graph). *The* word pair graph *is an undirected graph* $G = (V, E)$. *Every node* $v \in V$ *stands for a word in* W_{M^+}*, and every edge* $e(v_i, v_j) \in E$ *captures the co-occurrence between two words* w_i *and* w_j *such that there is an edge* $e(v_i, v_j) \in E$ *iff there is a word pair* $(w_i, w_j) \in C^+$.

Clearly, when the set $\mathbf{M}^+$ of all matches is given, we can then compute a complete WCG that captures the set $\mathbf{C}^+$ of all word pairs that co-occur in $\mathbf{M}^+$, and finally, derive the set $\mathbf{C}^-$ of all and correct CDEs from "missing edges" in that complete WCG according to Theorem 6.1. However, we only have a subset M^+ of positive examples for CDE learning. Based on Theorem 6.1, we compute CDEs as

$$C^- = C \setminus C^+ \tag{6.1}$$

The solution C^- derived from an incomplete set M^+ (incomplete C and C^+) might be incorrect and incomplete.

Incorrect CDE. When M^+ does not contain all matches, then some word pairs in $\mathbf{C}^+$ might not be contained in C^+. Because CDEs are derived as the complement set of C^+, these word pairs would be included in the solution C^-. However, they actually co-occur in matches (i.e. they co-occur in $\mathbf{C}^+$) and thus shall not be used as CDEs.

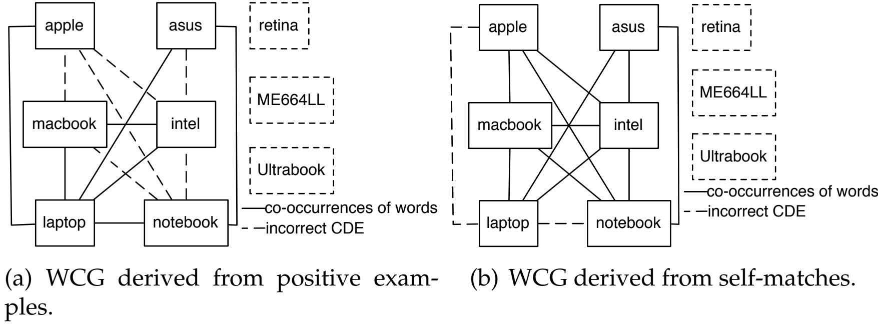
Incomplete CDE. The solution is incomplete when there are word pairs in $\mathbf{C}$ that are not in C, i.e. the values in $\mathbf{M}^+$ contain more unique words than the values in M^+. CDEs that belongs to $\mathbf{C} \setminus C$ cannot be derived.

Example 6.1. *Consider the WCG in Fig. 6.1(a) that is computed based on the attribute* `Description` *of instances in* $M^+ = \{(n_1,n_2),(n_6,n_7)\}$*. The solid lines indicate word pairs co-occurring in* M^+*, such as* (`asus`,`laptop`) *and* (`laptop`,`notebook`)*. All the missing edges are then derived as CDEs, such as* (`apple`,`asus`) *and* (`macbook`,`asus`)*.* M^+ *is incomplete because it does not include the matches* (n_3,n_4)*,* (n_5,n_6) *and* (n_5,n_7)*. As a result,* (`macbook`,`notebook`) *and* (`asus`,`intel`) *for instance (indicated by the dotted lines), are derived as CDEs because they do not co-occur in* M^+*. However, they co-occur in* (n_3,n_4) *and* (n_5,n_6)*, which are matches not included in* M^+*. Further, correct CDEs such as* (`ME664LL`,`Ultrabook`) *and* (`asus`,`retina`) *for instance, cannot be derived because* `retina`*,* `ME664LL` *and* `Ultrabook` *are not in* M^+*.*

The goal of this work is to derive a solution C^- that contains all and only correct CDEs from positive examples.

Definition 6.5 (Completeness and Soundness)**.** *Let* $\mathbf{C}^-$ *be the ground truth that contains all CDEs and* C^- *the learned solution, the learning goals are to minimize* $\mathbf{C}^- \setminus C^-$ *(*$\mathbf{C}^- \setminus C^- = \emptyset$ *means the solution is* complete*) and to minimize* $C^- \setminus \mathbf{C}^-$ *(*$C^- \setminus \mathbf{C}^- = \emptyset$ *means all CDEs in the solution are* sound*).*

Note that $\mathbf{C}^- \setminus C^-$ and $C^- \setminus \mathbf{C}^-$ stand for false negatives and false positives, respectively. Using false positives (incorrect CDEs) for filtering, instance pairs are pruned that are actually correct. This would result in a decrease in *recall*. On the other hand, when filtering is based on a solution with false negatives (an incomplete set of CDEs), some non-matches cannot be pruned, resulting in a decrease in *precision*. By maximizing the completeness and soundness of CDE learning, we aim to maximize the recall and precision of instance matching.



(a) WCG derived from positive examples.

(b) WCG derived from self-matches.

Figure 6.1: WCG in Fig. 6.1(a) that is computed based on the attribute `Description` of instances in $M^+ = \{(n_1, n_2), (n_6, n_7)\}$ according to data in Tab. 5.1. Solid lines indicate word pairs co-occurring in examples. Missing lines between any two nodes capture correct CDEs and dotted lines represent incorrect CDEs. Dotted squares indicate words that are not in the examples.

We provide the following theorems to reduce the false positives and the false negatives:

Theorem 6.2. *The set of false positives* $C^- \setminus \mathbf{C}^-$ *equals* $(\mathbf{C}^+ \cap C) \setminus C^+$.

Proof. Because $C^- \subseteq C$, we have $C^- \setminus (\mathbf{C}^- \setminus C) = C^-$. Then,

$$\begin{aligned} C^- \setminus \mathbf{C}^- &= C^- \setminus ((\mathbf{C}^- \cap C) \cup (\mathbf{C}^- \setminus C)) \\ &= C^- \setminus (\mathbf{C}^- \setminus C) \setminus (\mathbf{C}^- \cap C) \\ &= C^- \setminus (\mathbf{C}^- \cap C). \end{aligned} \tag{6.2}$$

Because $\mathbf{C}^- \cap C^+ = \emptyset$, we can write $\mathbf{C}^- \cap C = (\mathbf{C}^- \cap C) \setminus C^+$. Then,

$$\begin{aligned} C^- &= C \setminus C^+ \\ &= ((\mathbf{C}^+ \cap C) \cup (\mathbf{C}^- \cap C)) \setminus C^+ \\ &= ((\mathbf{C}^+ \cap C) \setminus C^+) \cup (\mathbf{C}^- \cap C). \end{aligned} \tag{6.3}$$

Finally, replacing C^- in Eq. 6.2 with the final rewrite obtained for C^- in Eq. 6.3 yields $C^- \setminus \mathbf{C}^- = (\mathbf{C}^+ \cap C) \setminus C^+$. □

Theorem 6.3. *The set of false negatives* $\mathbf{C}^- \setminus C^-$ *equals* $\mathbf{C}^- \setminus C$.

Proof. Because $C^+ \cap \mathbf{C}^- = \emptyset$, we can obtain the rewrite

$$\begin{aligned} \mathbf{C}^- \setminus C^- &= \mathbf{C}^- \setminus (C^- \cup C^+) \\ &= \mathbf{C}^- \setminus C \end{aligned}$$

□

These two theorems are useful for CDE learning because they reveal the relationships between false positives and the set of word pairs co-occurring in positive examples C^+ and between false negatives and the set of word pairs C. In particular, they imply that false positives and false negatives can be reduced by *enlarging* C^+ and C, respectively.

Intuitively, Theorem 6.2 captures that when using the set C to derive CDEs, the set of incorrect CDEs comprises elements in the intersection of $\mathbf{C}^+$ and C. That is, elements in C are considered as CDEs even though they actually co-occur in $\mathbf{C}^+$. However, this set is further reduced using elements in C^+. Namely, those CDEs found to co-occur in C^+ are discarded because they are incorrect. In other words, C^+ is used to filter incorrect CDEs. The larger this set, the larger is the number of incorrect CDEs (false positives) that can be filtered. Likewise, Theorem 6.3 can be interpreted as follows: CDEs are derived from C. Elements, whether correct or not, cannot be recognized as CDEs when they are not in C. Thus, enlarging C reduces the number of correct CDEs that cannot be derived (false negatives).

To achieve the learning goals, we now discusss the use of self-matches and self-training to enlarge C^+ and C, respectively.

6.4.3 Using Self-Matches as Examples

We enlarge C^+ by taking into account the following observation: every pair of instances formed by one instance and itself, i.e. instance pairs of the form (n,n) called *self-matches*, is correct.

Based on this observation, the given set of positive examples M^+, is augmented with self-matches, denoted as $M'^+ = \{(n_i,n_i)|n_i \in N\}$. Word pairs co-occurring in M'^+, referred to as C'^+, help to complement the word pairs C^+ extracted from M^+ because M'^+ and M^+ represent complementary sources. Intuitively, attribute values of the same instance contain *related words* whereas two different instances referring to the same thing, might used *similar words* (e.g. synonyms). That is, C^+ is a source of word pairs representing similar words that might be not available in C'^+, and C'^+ contains related words not captured by C^+.

Example 6.2. *Figure 6.1(a) and Fig. 6.1(b) show two WCGs that are constructed from the positive examples* $M^+ = \{(n_1,n_2),(n_6,n_7)\}$ *and the self-matches* $M'^+\{(n_1,n_1),\cdots,(n_8,n_8)\}$*, respectively. They capture different sets of word pairs co-occurring in* M^+ *and* M'^+ *(solid lines) and different false positives (dotted lines). For example, note the pair of related words* (`apple`,`intel`) *and the pair of similar words* (`laptop`,`notebook`) *only co-occur in* M'^+ *and* M^+*, respectively. These word pairs will be incorrectly derived as CDEs (false positives) if only* M^+ *or* M'^+ *is used.*

Instead of using $C^- = C \setminus C^+$ (Eq. 6.1), we thus compute CDEs by considering the union of C^+ and C'^+ as

$$C^- = C \setminus (C^+ \cup C'^+), \quad (6.4)$$

where as defined before, C is the Cartesian product representing all possible pairs of words in W_{M^+}, and C^+ and C'^+ are all word pairs co-occurring in M^+ and M'^+, respectively.

In this way, the set of false positives is reduced from $C^- \setminus \mathbf{C}^- = (\mathbf{C}^+ \cap C) \setminus C^+$ to

$$C^- \setminus \mathbf{C}^- = (\mathbf{C}^+ \cap C) \setminus (C^+ \cup C'^+). \quad (6.5)$$

Note that according to Theorem 6.3, false negatives can be reduced if we change C in Eq. 6.4. Instead of using all the words contained in the values of instances in M^+, we could replace C by C' to calculate the CDEs as

$$C^- = C' \setminus (C^+ \cup C'^+), \quad (6.6)$$

where C' is the Cartesian product of words in $W_{M'^+}$ (words in M'^+). Note that M'^+ captures **all** instances, which include the sets of instances covered by M^+ (and $\mathbf{M}^+$). Thus, the set of all word pairs derived from M'^+, C', must be a superset of C (and $\mathbf{C}$). In other words, C' is larger than C, thus computing CDEs via Eq. 6.6 helps to reduce the number of false negatives.

However, Eq. 6.6 can also result in a larger set of false positives, which can be calculated according to Theorem 6.2 as:

$$C^- \setminus \mathbf{C}^- = (\mathbf{C}^+ \cap C') \setminus (C^+ \cup C'^+) = \mathbf{C}^+ \setminus (C^+ \cup C'^+) \quad (6.7)$$

In fact, computing the difference between Eq. 6.5 and Eq. 6.7, we obtain $(\mathbf{C}^+ \setminus C) \setminus C'^+$, which represents the additional set of false positives produced by Eq. 6.6.

Intuitively, Eq. 6.1 uses all possible pairs of words contained in the values of positive examples as candidate CDEs (C). Then, word pairs that actually co-occur in positive examples (C^+) are used to discard incorrect CDEs. Eq. 6.4 goes one step further to also discard those related word pairs that co-occur in the same instance (C'^+). In Eq. 6.6, we also enlarge the set of candidates (C' instead of C). This set is however too large, including all word pairs that co-occur in positive examples C (in all matches $\mathbf{C}$) as well as those that do not. Hence, a larger number of correct CDEs can be learned, but the result also includes more incorrect CDEs. Since the effect of using Eq. 6.6 is not unambiguously positive, we only use self-matches to reduce false negatives, i.e. Eq. 6.4.

6.4.4 Enriching Examples with Self-learning

The previous discussion suggests that in order to reduce false positives as well as false negatives, candidate word pairs cannot be chosen randomly but shall co-occur in matches. Self-learning is employed to enrich the given set of positive examples with more matches, so that C is "selectively" enlarged to reduce the number of false negatives.

For this purpose, we assume a black-box matcher for candidate selection. It could be based on boolean matching, which given the instances N, produces the candidate set M_i at iteration i. The only property we assume is that this matcher can be parametrized to produce different sets of matches $\{M_1, \ldots, M_m\}$ that vary in size. In particular, we assume a list in which these sets are sorted:

Definition 6.6 (Sorted Candidate Sets)**.** *Let C_i be the Cartesian product of words in M_i. There is a list of candidate sets $\{M_1, \ldots, M_m\}$, where $M_i \subseteq M_{i+1}$ such that $C_i \subseteq C_{i+1}$, $\forall 1 \leq i < m$.*

With this list, we can guarantee that by using a larger set of candidates (M_{i+1} instead of M_i), more words can be taken into account ($C_{i+1} \supseteq C_i$). Clearly, the computation of the sorted candidate sets can be naturally supported by a thresholded similarity function. It is typically monotonic w.r.t. the threshold such that a smaller threshold always yields a larger set of candidates. In our boolean matching approach, we use a boolean similarity function that is based on the word overlaps between values, $same(n[a], n'[a]) = 1$ iff $\exists w.w \in W_{n[a]} \land w \in W_{n'[a]}$. Since the number of word overlaps is monotonic w.r.t. the size of the resulting candidate sets, we vary this measure to obtain the sorted list.

Self-learning is an iterative process, where the following components are computed in every iteration i:

$$C_i^- = C_i \setminus (C_i^+ \cup C'^+) \quad (6.8)$$

$$M_{i+1}^+ = Filtering(M_{i+1}, C_i^-) \quad (6.9)$$

where $0 \leq i < m$, C_i is the Cartesian product of words contained in values of instances in M_i^+, C_i^+ is the set of word pairs co-occurring in M_i^+, and C'^+ is the set of word pairs co-occurring in self-matches. The result of every iteration i comprises the set of CDEs, C_i^-, the candidate set M_{i+1} and the positive examples M_{i+1}^+. The positive example M_{i+1}^+ captures the set of results obtained by filtering the candidate set M_{i+1} using the CDEs obtained from the previous iteration, C_i^-. That is, we employ two boolean functions. Whereas similarity evidences are used for candidate selection in the first step to obtain M_{i+1}, CDEs are used as dissimilarity evidences to perform a subsequent candidate filtering step to compute M_{i+1}^+. In particular, $Filtering(M_{i+1}, C_i^-)$ is based on our CDE-based dissimilarity function: for a given

attribute a, an instance pair $(n[a], n'[a]) \in M_{i+1}$ is considered a non-match when there exists a CDE $(w_i, w_j) \in C_i^-$ that co-occurs in $(n[a], n'[a])$. Take the CDE (`Apple`,`ASUS`) and the attribute `Title` of the data in Tab. 5.1 for an example. Because `Apple` and `ASUS` appear in the `Title` of n_4 and n_5 respectively, the instance pair (n_4, n_5) is considered as a non-match.

In the beginning, the initial set of CDEs, C_0^-, is derived directly from the provided positive examples M_0^+ and self-matches M'^+ using Eq. 6.4. Then, we perform candidate selection to produce the candidate set M_1. Filtering this set using C_0^-, we obtain the refined set of examples M_1^+. These steps of CDE learning, candidate selection and candidate filtering are iteratively performed until we reach the last iteration m, which yields the final set of instance pairs M_m^+.

We provide the following theorem to show that this self-learning reduces the number of false negatives, i.e. the number of correct CDEs learned in iteration $i+1$ is a superset of the correct CDEs learned in iteration i:

Theorem 6.4. *For any two sets of CDEs C_i^- and C_{i+1}^- learned from M_i^+ and M_{i+1}^+, respectively, we have*

$$M_i^+ = M_{i+1}^+ \cap M_i \quad (6.10)$$

$$C_{i+1}^- = C_i^- \cup (C_{i+1}^- \setminus C_i). \quad (6.11)$$

Proof. Firstly, note all instance pairs that are filtered from M_i must contain at least one CDE in C_i^- and all instance pairs in M_i^+ must contain no CDEs in C_i^- such that the result of filtering on M_i using CDEs in C_i^- is M_i^+. Given M_i is a subset of M_{i+1} ($M_i \subseteq M_{i+1}$), we can write $M_{i+1} = M_i \cup (M_{i+1} \setminus M_i)$. Thus, the result of filtering on M_{i+1} using C_i^- can be calculated as $M_{i+1}^+ = M_i^+ \cup (M_{i+1} \setminus M_i)^+$, where $(M_{i+1} \setminus M_i)^+$ captures the result of filtering on $(M_{i+1} \setminus M_i)$. As a result, we have $M_i^+ = M_{i+1}^+ \cap M_i$ and $C_i^+ \subseteq C_{i+1}^+$ such that

$$C_{i+1}^+ \cap C_i^+ = C_i^+. \quad (6.12)$$

Secondly, since no CDEs in C_i^- can co-occur in M_{i+1}^+, the intersection of C_i^- and C_{i+1}^+ is empty:

$$C_{i+1}^+ \cap C_i^- = \emptyset \quad (6.13)$$

Then, we can rewrite C_{i+1}^- as the union of two terms:

$$C_{i+1}^- = (C_{i+1}^- \cap C_i) \cup (C_{i+1}^- \setminus C_i) \quad (6.14)$$

where $C_{i+1}^- \cap C_i$ are CDEs in C_i and $C_{i+1}^- \setminus C_i$ are CDEs not in C_i. Next, $C_{i+1}^- \cap C_i$ can

be calculated as the complement set of word pairs that co-occur in C^+_{i+1} and C'^+ as:

$$C^-_{i+1} \cap C_i = C_i \setminus ((C^+_{i+1} \cap C_i) \cup C'^+) \quad (6.15)$$

Note $C_i = C^+_i \cup C^-_i$, then Eq. 6.15 can be further rewritten as

$$C^-_{i+1} \cap C_i = C_i \setminus ((C^+_{i+1} \cap C^+_i) \cup (C^+_{i+1} \cap C^-_i) \cup C'^+). \quad (6.16)$$

Substituting Eq. 6.12 and Eq. 6.13 into Eq. 6.16, we can rewrite Eq. 6.16 according to Eq. 6.8 as follows:

$$\begin{aligned} C^-_{i+1} \cap C_i &= C_i \setminus ((C^+_{i+1} \cap C^+_i) \cup (C^+_{i+1} \cap C^-_i) \cup C'^+) \\ &= C_i \setminus (C^+_i \cup C'^+) \\ &= C^-_i \end{aligned} \quad (6.17)$$

Finally, substituting Eq. 6.17 into Eq. 6.14, we prove that $C^-_{i+1} = C^-_i \cup (C^-_{i+1} \setminus C_i)$ □

Intuitively, Theorem 6.4 captures that the CDEs and filtering results that are derived from a larger candidate set M_{i+1} are composed of two parts: *all* the CDEs and filtering result that could already learned with M_i and some new CDEs and result that are possible with the new words and instance pairs in M_{i+1}.

For a better understanding of the effects of using self-matches and self-learning, we refer to our set-theoretic analysis in the Appendix A.4.

6.4.5 On the Combination of Thresholded and Boolean Matching

We completely rely on boolean functions to obtain a simple, threshold-free approach that does not require fine-tuning. However, there are two natural ways to combine this boolean approach with thresholded instance matching.

A threshold similarity function can be used to generate the sorted list of candidate sets as previously discussed. Candidate sets are computed for and sorted according to the given threshold. In this case, thresholded matching is used for the candidate selection step, which is then followed by a boolean candidate filtering step based on CDEs. As a whole, this combination represents a matching solution that implements our iterative embedded process of CDE learning, thresholded candidate selection and boolean candidate filtering.

This whole solution can also be treated as a black box for *boolean filtering*. Given candidates computed by a thresholded approach, it can be used to identify and filter dissimilar matching candidates.

6.4.6 Multiple Attributes

So far, we have discussed CDE learning and instance matching always for a given attribute a. That is, when some word pairs are said to co-occur in values of some instances, it actually means they co-occur in the values of a specific attribute a of some instances. Because CDEs learned from different attributes capture dissimilarity evidences from different aspects, a non-matching instance pair that cannot be identified by CDEs learned from one attribute could be identified by CDEs learned from other attributes. In the multiple attributes setting, we apply the self-learning procedure to every attribute. That is, CDEs are learned for any attribute. A candidate match is considered correct only if it cannot be filtered by CDEs that are learned from every attribute. Let $\{a_1, a_2, \cdots, a_k\}$ be a set attributes and M_{ij}^+ be the result obtained by applying the filtering step on the candidate set M_i using CDEs learned from a_j, we calculate the result as the intersection of the results obtained for every attribute, i.e. $M_i^+ = \bigcap_{j=0}^{k} M_{ij}^+$.

6.5 Implementation

We previously focused on the main ideas behind CDE learning. Here, we also consider the aspect of efficiency, presenting a solution that aims to address the following two problems:

Duplicate Instance Pairs: Self-learning is an iterative procedure in which the steps of CDE learning, candidate selection and filtering are iteratively executed. Clearly, candidate sets processed in different iterations largely overlap in the instance pairs they contain. It is thus not efficient to repeatedly process the same instance pairs over several iterations. We can leverage Theorem 6.4 to solve this problem: for any two adjacent sorted candidates sets M_i and M_{i+1}, we know that the CDEs and filtering result computed for the same set of instance pairs in M_i and $M_{i+1} \cap M_i$ are the same, i.e. $M_i^+ = M_{i+1}^+ \cap M_i$, $C_i^- = C_{i+1}^- \cap C_i$. Since the computed results for instance pairs in M_i in each iteration are the same, we can simply remove the instance pairs in M_i from M_{i+1}, i.e. only process the instance pairs in the set $M_{i+1} \setminus M_i$.

Expensive WCG Calculation: To learn CDEs according to Eq. 6.8, the Cartesian product of all words in the candidate set, C, is taken into account. Then, we consider C^+ to derive the CDEs as the complement set C^-. While using these different sets help to explain the idea, we observe that for CDE-based filtering, only C^+ is actually needed. According to Theorem 6.1, a word pair is a CDE if it is not in C^+. Thus, an instance pair can be considered a non-match if one of its word pairs is not in C^+ (i.e. that word is a CDE). In this way, we can check for CDEs and perform filtering simply by looking at word pairs that co-occur in a given instance pair and word pairs in C^+. To capture this idea, we can redefine the CDE-based dissimilarity function as follows:

Definition 6.7 (C-Based Diss. Function)**.** *Let M^+ be a set of positive examples and C^+ be the set of word pairs that co-occur in M^+, the* CDE-based dissimilarity function $\neg same : N[a] \times N[a] \rightarrow \{0,1\}$ *maps a pair of attribute value $(n[a], n'[a])$ to 1, if there exists a word pair (w_i, w_j) that co-occurs in $(n[a], n'[a])$ and $(w_i, w_j) \notin C^+$, and 0 otherwise.*

Since the number of matches is usually much smaller than the number of non-matches, we can expect that most word pairs do not co-occur in matches, i.e. the size of C^+ is much smaller than C^- and C. Thus, focusing on C^+ could largely improve efficiency.

Algorithm 9: Self-learning based instance matching

Input: M_0^+, the initial examples; M'^+, self-matches; $\{M_1, M_2, \cdots, M_m\}$, sorted candidates sets; and the attributes $\{a_i, a_2, \cdots, a_k\}$.
Data: C^+_{i,a_j}, word pairs co-occurring in the values of a_j of instances in M_i^+
Result: M^+.

```
1  //"Learn CDEs" with self-matches + examples;
2  for j = 1 to k do
3      C+_{0,a_j} := LearnCDE(a_j, M+_0 ∪ M'+);
4  //Self-training;
5  for i = 1 to m do
6      M+_i = M_i \ M_{i-1};
7      for j = 1 to k do
8          M+_i := Filtering(a_j, M+_i, C+_{i-1,a_j});
9      for j = 1 to k do
10         C+_{i,a_j} := C+_{i-1,a_j} ∪ LearnCDE(a_j, M+_i);
11     M+ := M+ ∪ M+_i;
12 return M+;
```

The procedure for instance matching, which assumes a sorted list of candidate sets obtained via candidate selection and iteratively performs CDE learning and candidate filtering, is shown in Alg. 9. Due to the change discussed above, CDE learning and candidate filtering now involves the use of C^+ instead of C^-. That is, Alg. 11 actually does not learn CDE but C^+, which is then used to identify CDE in instance pairs and to filter them.

First, C^+ is constructed using self-matches and the given training examples (line 1-3). Then filtering (Alg. 10) and CDE learning (Alg. 11) are applied iteratively on each of the sorted candidates sets (line 5-11). As discussed, in Alg. 9, in each iteration, the instance pairs in M_{i-1} can be removed from M_i first to avoid duplicate processing. The resulting set of candidates is used to initiate M_i^+ (line 6). Then, non-matches from

Algorithm 10: Filtering

```
Input: Attribute a_j; and candidates set M_i, word pairs C^+_{i-1,a_j}.
Result: A set of instance pairs M^+_i.
1 foreach (n,n') ∈ M_i do
2     foreach w ∈ W_{n[a_j]} do
3         foreach w' ∈ W_{n'[a_j]} do
4             //if (w,w') is a CDE;
5             if (w,w') ∉ C^+_{i-1,a_j} then
6                 M_i := M_i \ {(n,n')} ;
7 return M^+_i := M_i;
```

Algorithm 11: Learning CDEs

```
Input: attribute a_j; instance pairs M^+_i.
Result: C^+_{i,a_j}, word pairs co-occurring in the values of a_j of instances in M^+_i.
1 //Find word pairs co-occurring in M^+_i;
2 foreach (n,n') ∈ M^+_i do
3     foreach w ∈ W_{n[a_j]} do
4         foreach w' ∈ W_{n'[a_j]} do
5             C^+_{i,a_j} := C^+_{i,a_j} ∪ {(w,w')} ;
6 return C^+_{i,a_j};
```

this candidate set are filtered according to Def. 6.7 (line 8). After filtering using all attributes, enrichment is performed to collect more words and C_i^+ from the filtering result M_i^+ (line 10). Finally, the filtering result for every iteration is added to the result set (line 11).

Example 6.3. *Fig. 6.2 illustrates the use of* `Description` *for the data in Tab. 5.1. Lets have a look at iteration 1: we have already collected the word pairs C_0^+ and C'^+ that co-occur in the examples M_0^+ and the self-matches M'^+ respectively. Now the algorithm applies filtering to the larger candidate set M_1, and then learns CDE from M_1^+, as follows:*

*1. **Removing duplicate instance pairs**: The pairs in M_0 are removed from M_1 to avoid duplicate processing.*

*2. **Filtering**: Filtering is applied to the remaining instance pairs in M_1. For example, (n_1,n_3) is derived as a non-match according to Def. 6.7 because the word pair* `(i5,i7)` *co-occurs in (n_1,n_3) but not in C_0^+ and C'^+. Finally, we can infer (n_3,n_4) as a match, which is*

then used as a positive example for CDE learning in the next step.

*3. **Learning CDEs**: In Iteration 2, enrichment is performed with the pairs of new words taken from M_1^+ such as* (`apple`,`retina`) *and* (`ME664LL`,`pro`) *that co-occur in* (n_3,n_4). *As a result, C_1^+ contains all the word pairs that co-occur in M_0^+, M'^+ and M_1^+.*

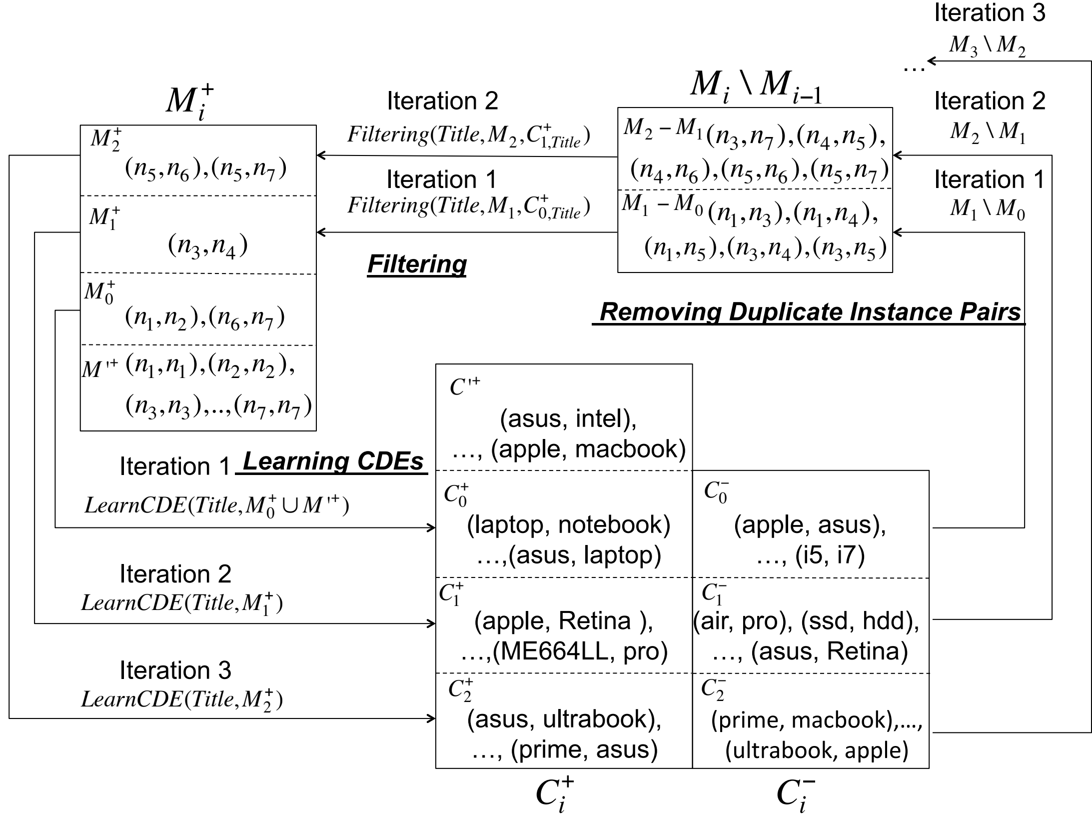


Figure 6.2: An example of the algorithm

The time complexity for every iteration in Alg. 9 depends on the number of word pairs that are compared. The algorithm needs to check all the instance pairs in M_0^+, M'^+ and M_m, where M_0^+ denotes the initial set of positive examples, M'^+ is the set of self-matches, and M_m denotes the largest one among all the sorted candidate sets (note that M_m contains all the examples added to the initial set as part of the iterative enrichment). For each instance pair, let k be the number of attributes and w^2 be the maximum number of word pairs that co-occur in the value of an attribute. In worst case, the number of word pairs that the algorithm processes is $(|M_0^+ \cup M'^+ \cup M_m|)kw^2$. With m as the number of iterations, the total time complexity is $O((|M_0^+ \cup M'^+ \cup M_m|)kw^2m)$.

6.6 Experimental Evaluation

To study the proposed solution, we employ a recent instance matching benchmark [66] that captures data from enterprise databases as well as synthetic data. Tab. 6.2 provides an overview of the datasets.

Table 6.2: Instances and ground truth (GT).

Task	Instances		GT
	Dataset1	Dataset2	
AD	2,616	2,294	2,224
AB	1,081	1,092	1,097
Rest	864		112

ACM-DBLP (AD). These datasets in the benchmark [66] include well-structured bibliographic data from DBLP and ACM digital library. This one is manually created and thus, is of higher quality among all datasets. Therefore, the task represented by this is of low difficulty.

Abt-Buy (AB). This matching task in the benchmark [66] is performed between instances collected from `http://abt.com` and `http://buy.com`. There are many errors and noises that cannot be avoided during automatic data collection. This matching task is in fact the most difficult one in this experiment.

Restaurant(Rest). This dataset is made available through the OAEI 2010[1] benchmark. We removed the `telephone` attribute because with it, previous results published by the benchmark show that it is too easy of an matching task (all systems perform well).

Systems. We compare our techniques with three recent approaches [57, 109, 119] proposed for instance matching. Two of them are thresholded instance matching approaches, denoted as `SiFi` [119] and `ST` [57]. `SiFi` learns similarity functions and thresholds from positive and negative examples. `ST` learns the most discriminative attributes and values to be used for matching. It uses a given similarity function `ISub` and thresholds manually set by experts. The third, named `Paris`, is a boolean instance matching approach that outputs true for two instances, if they have the same attribute values. The idea here is to iteratively cross-fertilize instance matching results with schema matching results until convergence is reached [109].

Quality Metrics. We use the standard metrics for comparing instance matching results: Precision (P) and Recall (R) and F-measure (F).

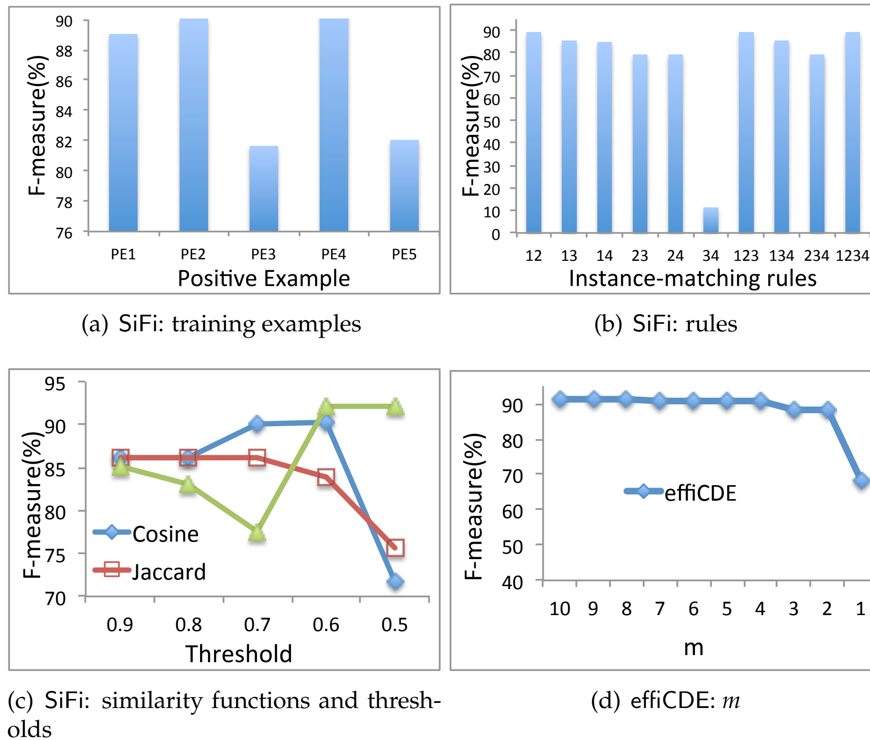
Setting. All experiments were run on a server with two Intel Xeon 2.8GHz Dual-Core CPUs, using 8GB of main memory, running Linux with kernel version 2.6.18. The presented results are computed as an average over five runs.

[1] `http://oaei.ontologymatching.org/2010`

6.6.1 Parameter Analysis

Compared to thresholded instance matching approaches, our solution is parameter-free in this sense: we simply use all the attributes, $m = 10$ as the number of iterations, and the overlap of values as the boolean function for candidate selection, i.e. instances that have the same value are considered as a match candidate. As illustrated in Fig 6.3(d) (average F-measure for all tasks), our approach achieves fairly stable results for different values of m. The only exception is when m is set to 1, which means the CDEs are learned only based on the provided examples and self-matches. Self-learning improves the results (changing $m = 1$ to any value great than 1). However, the results suggest improvements converge quickly after $m > 4$.

Now, we study the sensitivity of existing approaches (`SiFi` and `ST`) with respect to training data and parameters. We will focus our discussion on the task `Rest`.



(a) SiFi: training examples

(b) SiFi: rules

(c) SiFi: similarity functions and thresholds

(d) effiCDE: m

Figure 6.3: Parameter analysis for SiFi and effiCDE.

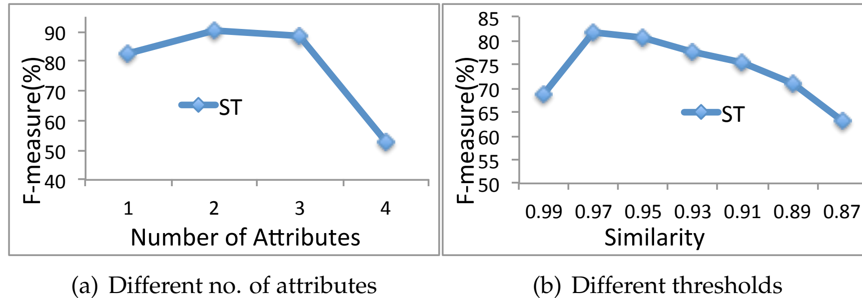
SiFi. In the experiments, we adopt the four instance-matching rules as reported by the authors in their original paper [119]. SIFI requires attributes to be manually set for each rule. Also, it requires the similarity functions and thresholds to be manually set for some attributes so that they can be learned for other attributes. Besides this

problem of manual tuning, the sensitivity of the approach w.r.t. parameters and training data is as follows.

Firstly, different sets of positive examples of the same size (30% of the ground truth) were randomly selected to analyze the sensitivity of training data. As illustrated in Fig 6.3(a), we can see results achieved by `SiFi` were not stable. Depending on the set of training examples, results vary between 81% and 9% in term of F-measure.

Secondly, different combinations of the four original instance-matching rules were used. For example, in Fig. 6.3(b), we use `13` to denote the usage of the first and the third rules. We can see that also with respect to parameters, the results are not stable (vary between 78% and 90%, not counting outlier 34). Considering 34, F-measure result might be as low as 10%. We note the results are the same for the settings `12` and `1234`, indicating that the third and the fourth rules are actually not necessary for the task at hand. However, it is difficult for experts to determine and select the rules (especially when not only result quality but also efficiency is a relevant factor).

Finally, we analyze the effect of using different thresholds and similarity functions that we manually set for the attribute `name`. As illustrated in Fig. 6.3(c), results greatly vary when either the similarity function or the threshold changes. Note in practice, experts have to chose the best setting for multiple attributes, selecting from more than tens of relevant similarity functions and all candidate thresholds in the $[0,1]$ interval.



(a) Different no. of attributes (b) Different thresholds

Figure 6.4: Parameter analysis for ST

ST. With this approach, experts have to select the attributes used for instance matching so that instance pairs, whose similarity on any attribute is higher than the threshold can be considered as matches. Intuitively, more matching instance pairs can be found when we consider more attributes (high recall). However, a high number of attributes may also result in too many non-matches (low precision). In Fig. 6.4(a), we see this for the data in our experiment. F-measure result initially improves as the number of attributes increases (the effect of improved recall) and then drops sharply as many more attributes are added (the effect of reduced precision).

We also analyze the results computed using different thresholds that are between 0.87 and 0.99. As shown in Fig. 6.4(b), `ST` is extremely sensitive to the chosen thresh-

old. For example, F-measure decreases sharply from 81% to 68% when the threshold is slightly changed from 0.97 to 0.99.

In summary, these experiments show that while good results can be achieved using thresholded approaches (approaches that require fine tuning, often much more than just the thresholds), they are very sensitive to the choice of parameters and data. Result quality depends not only on the underlying approach but also the effectiveness of the experts applying it. In what follows, we now compare these thresholded approaches with boolean approaches, including our approach and Paris, which aim to reduce the efforts needed for manual tuning.

6.6.2 Efficiency of Instance Matching

In this section, we compare the efficiency of our solution with existing approaches. As discussed in Sec.6.4.5, it can be combined with the use of thresholds in two different ways. When not explicitly mentioned, experimental results presented for our solution here are entirely based on boolean matching as presented in the chapter.

Because it is sometime difficult to separate the training and matching processes in instance matching approaches (they are often intertwined as results from matching are used to improve learning), we evaluate the efficiency of all methods w.r.t. total running time as shown in Tab. 6.3.

Table 6.3: Performance of learning and instance matching in ms.

	effiCDE	CDE	SiFi	ST	PARIS
AD	42,552	2,139,949	502,450	48,362	**25,955**
AB	14,068	699,031	33,418	12,202	**7,180**
Rest	**4,508**	52,798	58,491	5,471	10,517

CDE and effiCDE represent our solutions with and without efficiency improvements as proposed in Sec. 4. Clearly, effiCDE achieved much better performance: it needs only 2% of the time taken by CDE. Especially, because the size of C^+ (used by effiCDE) is only 1% - 2% of the size of C^-(used by CDE) in the experiments, effiCDE is also much more efficient in terms of memory usage.

Comparing to the thresholded instance matching approaches SiFi (ST), effiCDE exhibits better (similar) overall performance. In the learning step, effiCDE is efficient because it avoids the learning of complex instance matching rules: it only has to focus on word pairs that co-occur in a relatively small-amount of matches. It is also efficient in the matching step because instead of possibly complex similarity calculations (depending on the similarity functions), it only requires boolean matching on words. For the dataset AD as an example, effiCDE requires only 42,552 ms, which is only 10% of the time required by SiFi. SiFi exhibits worst performance because it has to search the best rules from a large search space of rule candidates. Even though SiFi implements a strategy for eliminating candidates with redundant similarity

functions and thresholds, we observe that the operations needed for determining and removing redundancy itself is still too costly. `ST` is fast because compared to `SiFi`, it pursues a more easy learning task, which does not require the learning of similarity functions and thresholds (hence, much smaller search space).

`Paris` is the fastest among all approaches. This is because just like our approach, PARIS only employes boolean matching. It is faster than our solution mainly because it is much more aggressive in pruning candidates. In our approach, candidates are recognized whenever they match on some words contained in their values. Further, before filtering, some are actually "re-examined" in several rounds of our iterative process. `Paris` keeps only candidates that match on their values and filters them more aggressively throughout the iterations. While this strategy largely increases performance, it also leads to lower recall when the data contains more noises, as discussed in the following.

Compared to the benchmark for the task AD, `effiCDE` costs more time than the learning-based approaches including SVM and decision tree, which use three similarity functions and two attributes for matching. SVM and decision tree spend 20 seconds and 12 seconds, respectively. However, because we count both learning and testing time, and process all the four attributes for matching, it is difficult to compare with the benchmark in details. On the other hand, for the task AB, `effiCDE` that use only 14 seconds for execution is much more efficient than SVM and decision tree, which cost at least 232 seconds and 551 seconds, respectively.

6.6.3 Effectiveness of Instance Matching

Tab. 6.4 presents results for our three quality metrics. Overall, we can see that `effiCDE` outperforms all the other approaches in terms of F-measure. Compared to the second best result achieved by any of the studied approaches, it improves F-measure by +3.66%, +206.47% and +1.34% for `AD`, `AB` and `Rest`, respectively. Especially, we note `effiCDE` is most effective in filtering non-matching instance pairs. This is reflected in the higher precision results: the improvements over the second best are +6.51%, 181.64% and 1.56% for `AD`, `AB` and `Rest`, respectively. We note that in fact, our filtering approach primarily aims at precision. Regarding recall, the boolean matching as implemented for the candidate selection step in our approach is too strict as it only selects exact matches. This however, can be remedied by using thresholded matching for candidate selection as discussed in Sec. 6.4.5 and in what follows.

Regarding specific tasks, we observe `effiCDE` is particularly effective in dealing with the most difficult task, i.e. finding product pair matches on the relatively noisy `AB` dataset. The results achieved for this task are much worse than for other tasks. This is because the automatically extracted product descriptions largely vary, while the bibliographic data `AD` and restaurant data `Rest` are more structured and uniform. As a result, when entire attribute values are considered, matches are hard to

Table 6.4: Effectiveness of instance matching in terms of P, R and F

	effiCDE			SiFi			ST			PARIS		
	P	R	F	P	R	F	P	R	F	P	R	F
AD	**96.58**	**97.84**	**97.21**	90.68	97.10	93.78	85.19	96.99	90.71	93.20	93.62	93.41
AB	**74.24**	**60.43**	**66.63**	26.36	18.51	21.75	20.77	18.69	19.67	12.01	2.55	4.21
Rest	**88.33**	**94.64**	**91.37**	86.77	93.75	90.13	87.83	90.17	88.98	5.90	45.53	10.45

find on AB. In fact, at this attribute-level, AB cannot provide sufficient similarity evidences such that the thresholded instance matching approaches SiFi and ST fail to find the thresholds that cover the given example matches. At this level, there are also not sufficient evidences for a boolean approach: PARIS incorrectly prunes candidates because only few product pairs entirely match on their attribute values. More evidences can however be collected at the word level. Products that have low similarities on entire attribute values, often share very distinctive word pairs. Our CDE-based solution leverages this, providing a better solution for noisy data by exploiting more fine-grained dissimilarity evidences. The F-measure improvement for AB is +206.34%.

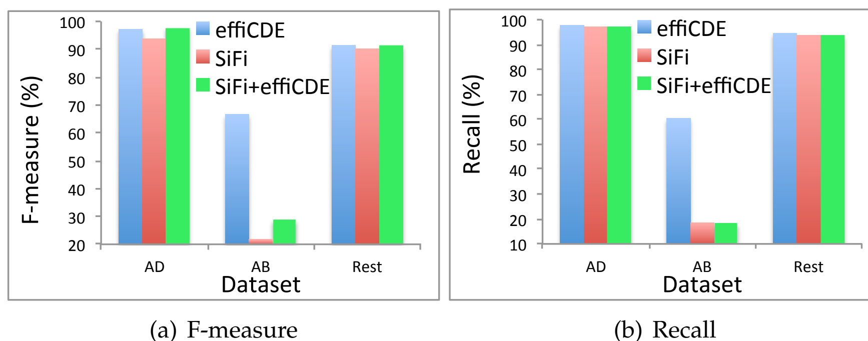
Compared to the benchmark, effiCDE achieves the very similar result as the best system SVM. Especially for the task AB, we can observe that effiCDE outperforms all the other systems. This is because the data in the AB is so noisy that AB is not a linear classification problem. While SVM can solve the non-linear classification problem by mapping the inputs into high-dimensional feature spaces, effiCDE solves this problem in another direction by using dissimilarity features in the word level.

Combining Boolean and Thresholded Matching. Instead of using a pure boolean approach, we also experiment with the combination of effiCDE and thresholded matching. It is applied for filtering results computed by other approaches. Fig. 6.5(a) shows results obtained by applying effiCDE as a filter on SiFi results. Compared to SiFi, F-measures consistently increase for all tasks after applying effiCDE as a filter. We can see in Fig. 6.5(b) that recall does not change. Thus, this F-measure improvement is entirely due to the positive effect of filtering on precision. As a filter, effiCDE helps to reduce non-matches while preserving all matches produced by the underlying approach.

However, this combination is not always better than effiCDE. SiFi could not find many candidates for the noisy AB datasets. Using effiCDE as a filter could not solve this problem of recall.

6.6.4 Labeling Effort

Because SiFi is the only approach that requires training examples (other approaches used “pseudo-examples”, i.e. matches computed in the first run), we use it as a com-



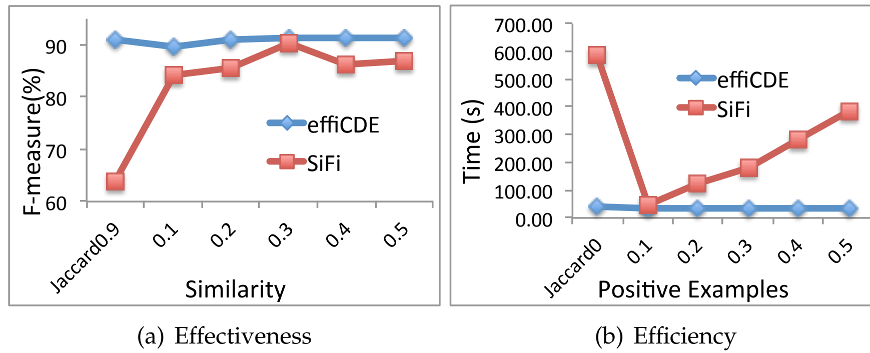
(a) F-measure (b) Recall

Figure 6.5: effiCDE as Filters

parison to study the labeling efforts needed to obtain examples. As example, we use different sets of matches (between 10% and 50%) taken from the ground truth. To consider sensitivity w.r.t. noises in the training data, we also use the set of instance pairs whose *Jaccard* similarity is greater than 0.9 as pseudo-examples, denoted as `Jaccard0.9`. For `Jaccard0.9`, precision, recall, and F-measure are 89.89%, 79.46% and 84.36%, respectively.

Figure. 6.6(a) shows the F-measures results. Clearly, `effiCDE` manages to maintain high quality results for varying amount of labeling efforts, including for `Jaccard0.9` that requires no efforts. This is because training examples are largely enriched by self-matches as well as matching obtained via self-training. Thus, the initial set of seeds is not critical for our approach. Note that `effiCDE` performs well for `Jaccard0.9`, while `SiFi` provides very low quality results. The non-matches in these pseudo-examples represent false similarity evidences, which have a large negative impact on the thresholds learned by `SiFi`. With `effiCDE`, word pairs that should be CDEs are incorrectly recognized as word pairs co-occurring in matches, due to the existence of non-matches in the examples. This only results in a smaller set of initial CDEs that can be learned. Even with this lack of CDEs, a non-match can still be filtered for two reasons: 1) there are multiple CDEs existing in various attributes of a non-match and any attribute can be used for filtering; 2) most importantly, many more CDEs can be derived later through the enrichment via self-matches and self-training.

As shown in Fig. 6.6(b), `effiCDE` achieves stable running time given varying amount of labeling efforts while `SiFi` takes more time when the number of training examples increases. As discussed, the time complexity of `effiCDE` depends on the number of instance pairs in the initial set of examples, self-matches and sorted candidates sets. The running time of our solution is less sensitive to the initial training examples because its size is relatively small compared to the other two factors. `SiFi` however directly depends on it. An increase in size leads to a much larger search space that has to be considered for rule learning.



(a) Effectiveness (b) Efficiency

Figure 6.6: Evaluation result for different labeling efforts.

6.7 Related Work

For fine-grained matching and filtering based on thresholds, different similarity functions have been incorporated into matching rules, including character-based metrics (e.g. edit distance) and token-based metrics that consider the rearrangements of words [40]. `Paris` [109] measures the degrees of matching based on probability estimates (and cross-fertilizes them through evidence propagation between the data and schema level).

For learning the rules, there are supervised techniques based on existing machine learning methods (such as SVM [14]) or specific ones designed for instance matching (e.g. learning similarity function predicates from training data can be reduced to the maximum rectangle problem [24]). As shown for `SiFi` [119], highest quality can be achieved when combinations of attributes, similarity functions and thresholds are considered for rule learning. Further, self-learning, which incorporates previous learning results into the loop can yield improvements. Recent approaches for this include `Paris` that employes evidence propagation [109] and `ST` [57].

There are also semi-supervised techniques based on probabilistic graphical models. It has been shown that many existing instance matching (and blocking) approaches can be captured as Markov Logic formulas and reformulated as a Markov Logic learning problem [105]. However, this involves learning both the structure (formulas) and their weights. Existing works focus on weight learning such that the formulas have to be specified manually.

Our idea of using dissimilarity evidences for filtering is most related to the use of constraints also called negative rules. It has been shown that using manually designed constraints can help to effectively filter non-matches [4, 10, 37, 38, 103, 122]. The main differences to our work are: (1) these constraints (just like instance matching rules) capture evidences at the level of attributes, i.e. they are based on comparing the whole attribute values. (2) Constraints (which are relative small in number) shall

capture the domain knowledge of experts while our dissimilarity functions (a large number of them) are driven by the data. They are manually designed by experts while our functions are learned from the data.

6.8 Conclusion

For the problem of instance matching, we provide a solution that instead of threshold functions that capture attribute-level evidences, employs more simpler boolean functions but also more fine-grained word-level evidences. We show that with this boolean approach, a parameter-free procedure can be designed that when combined with word-level evidences, can be highly effective. Compared to state-of-the-art instance matching approaches, this solution greatly improves result quality and most importantly, is not sensitive to the choice of training data and parameter. We show that it can be employed as a standalone solution for instance matching or combined with a matcher as a boolean filter. As future work, we will explore the usage of more complex boolean functions as well as a tighter combination of the boolean and threshold approaches.

Chapter 7

Conclusion

We conclude this thesis by summing up the research question and the achieved results. Afterwards, we provide an outlook on further research as well as conclusions.

7.1 Summary

One main problem towards the effective use of an increasing amount of structured data is instance matching as described in Chapter 1 and detailed in Chapter 2. Focusing on the topic of instance matching, we raised the following principal question and investigated it in this thesis:

How to match instances effectively and efficiently for heterogeneous structured data?

Concerning this question, we identified three challenges in Chapter 1. We tackled all the challenges in four sequential steps of the instance matching process, where each step corresponds to a specific research question. We investigated these questions in depth through experiments and provided a scientific contribution on each question, as stated in Section 1.5.

Considering the challenge of heterogeneity of structured data on the Web, the first step is to group instances that are of the same type. We showed this question is that *How to derive type semantics of instances* in Chapter 3. We discussed the features that are need to solve this problem. Because schema features perform best, but are not abundantly available, we proposed an approach to automatically derive them from data. Optimized for the use of schema features, we presented TYPifier, a novel clustering algorithm that in experiments, yields better typification results than the baseline clustering solutions.

In the second step, blocking, the question is *How can match candidates be efficiently and effectively generated*, which is answered in Chapter 4. Aiming to deal with the challenge of high complexity, we proposed an unsupervised approach to learn the

type-specific representative attributes called keys, based on which match candidates are generated by finding instances that share the same value of the key.

Due to the low quality of structured data, as stated in the third challenge in Section 1.3, a large number of calculated candidates are actually non-matches. In the classification step, the question *How can match candidates be effectively classified to matches and non-matches* is answered by a rule-based instance matching approach. In Chapter 5, we proposed an almost-parameter-free approach to learn combinations of attributes, similarity functions and thresholds, called instance-matching rule, for finding matches. Two instances are classified to the class match if the average similarity of the learned attributes is greater than the threshold.

In Chapter 6, we showed that since the use of threshold is sensitive to training data and parameters, the approaches that are based on the use of thresholds and comparing instances at the coarse-grained level of attribute values may not always perform well and hence output a lot of instance pairs that are actually non-matches. This problem results in the question *How to identify non-matching instance pairs by simple Boolean functions* in the filtering step. To answer this question, we proposed a parameter-free solution that exploits fine-grained word-level dissimilarity evidences to identify the non-matching instance pairs. Due to the Boolean nature of the word-level dissimilarity evidence, the learning of these evidences is more simple, i.e. does not require parameters and is not sensitive to training data. At the word level, the large-number evidences are more discriminative in identifying non-matches, which leads to high precision and recall of the instance matching result.

7.2 Outlook

In this section, we provide an outlook on the future development of instance matching.

Matching Unstructured Data. We described instance matching techniques that are applied on structured data. In reality, there is a large volume of unstructured data that is stored as free texts without any attributes, such as the content extracted from emails, blogs and news articles [18, 26]. Information extraction techniques are generally need for getting information that can be used for the instance matching approaches proposed in this thesis [99]. In this way, the quality of data preprocessing is crucial in order to achieve high quality results.

Real-time Instance Matching. Our approaches are designed to be applied as an offline process. However, practical applications such as search engines and security systems may need the data to be matched in real time [8, 19, 25, 64, 95, 118, 131]. Such applications involve a trade-off between matching accuracy and matching speed as well as the scalability to very large databases [13, 36]. The real-time processing may also need specialized indexes of instances to enable efficient generation of match can-

didates and calculation of similarities [29, 30].

Matching Multi Data Sources. In this chapter, we discussed how instances in two data sources can be matched. In certain applications, such as online products comparisons, instances from more than two sources need to me matched. The current techniques should be reconsidered carefully for matching instances that are from multi data sources. [48, 121].

Instance Matching Framework. Currently, various approaches that correspond to different steps of the instance matching process have been proposed. Then a unifying framework for instance matching is required that allows different approaches being integrated together towards better performance and easier evaluation. Such framework should also allow new algorithms to be easily plugged into the system.

7.3 Conclusion

Overall, we have shown in different settings from different perspectives how heterogeneous structured data can be efficiently and effectively matched. We extracted the type semantics of instances, and learned the type-specific blocking keys for each type of instances to generate match candidates. Then, we presented solutions to learn similarity evidences at the level of attributes and dissimilarity evidences at the level of words to identify matching and non-matching instance pairs, respectively. Further, the proposed approaches are all validated through extensive experiments that show they improve upon the state of the art. Finally, we gave an outlook on the future development of instance matching in the end of the thesis.

Bibliography

[1] Mihael Ankerst, Markus M. Breunig, Hans-Peter Kriegel, and Jörg Sander. Optics: Ordering points to identify the clustering structure. In *Proceedings ACM SIGMOD International Conference on Management of Data*, pages 49–60, 1999.

[2] Arvind Arasu, Venkatesh Ganti, and Raghav Kaushik. Efficient exact set-similarity joins. In *Proceedings of the 32nd international conference on Very large data bases*, pages 918–929. VLDB Endowment, 2006.

[3] Arvind Arasu, Surajit Chaudhuri, and Raghav Kaushik. Transformation-based framework for record matching. In *Proceedings of the 24th International Conference on Data Engineering (ICDE)*, pages 40–49, 2008.

[4] Arvind Arasu, Christopher Ré, and Dan Suciu. Large-scale deduplication with constraints using dedupalog. In *Proceedings of the 25th International Conference on Data Engineering (ICDE)*, pages 952–963, 2009.

[5] David Arthur and Sergei Vassilvitskii. k-means++: the advantages of careful seeding. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1027–1035, 2007.

[6] Jeffrey Baumes, Mark K. Goldberg, Mukkai S. Krishnamoorthy, Malik Magdon-Ismail, and Nathan Preston. Finding communities by clustering a graph into overlapping subgraphs. In *Proceedings of the IADIS International Conference on Applied Computing*, pages 97–104, 2005.

[7] Rohan Baxter, Peter Christen, and Tim Churches. A comparison of fast blocking methods for record linkage. In *Proc. ACM SIGKDD âĂŹ03 Workshop Data Cleaning, Record Linkage, and Object Consolidation*, pages 25–27, 2003.

[8] Roberto J. Bayardo, Yiming Ma, and Ramakrishnan Srikant. Scaling up all pairs similarity search. In *Proceedings of the 16th International Conference on World Wide Web (WWW)*, pages 131–140, 2007.

[9] Asa Ben-Hur, David Horn, Hava T. Siegelmann, and Vladimir Vapnik. Support vector clustering. *Journal of Machine Learning Research*, 2:125–137, 2001.

[10] Indrajit Bhattacharya and Lise Getoor. Relational clustering for multi-type entity resolution. In *ACM SIGKDD Workshop on Multi Relational Data Mining (MRDM)*, 2005.

[11] Indrajit Bhattacharya and Lise Getoor. A latent dirichlet model for unsupervised entity resolution. In *Proceedings of the Sixth SIAM International Conference on Data Mining (SDM)*, 2006.

[12] Indrajit Bhattacharya and Lise Getoor. Online collective entity resolution. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence, July 22-26, 2007, Vancouver, British Columbia, Canada*, pages 1606–1609, 2007. URL `http://www.aaai.org/Library/AAAI/2007/aaai07-255.php`.

[13] Indrajit Bhattacharya, Lise Getoor, and Louis Licamele. Query-time entity resolution. In *Proceedings of the Twelfth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 529–534, 2006.

[14] Mikhail Bilenko and Raymond J. Mooney. Adaptive duplicate detection using learnable string similarity measures. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 39–48, 2003.

[15] Mikhail Bilenko, Raymond J. Mooney, William W. Cohen, Pradeep D. Ravikumar, and Stephen E. Fienberg. Adaptive name matching in information integration. *IEEE Intelligent Systems*, 18(5):16–23, 2003.

[16] Mikhail Bilenko, Beena Kamath, and Raymond J. Mooney. Adaptive blocking: Learning to scale up record linkage. In *Proceedings of the 6th IEEE International Conference on Data Mining (ICDM)*, pages 87–96, 2006.

[17] Philip Bille. A survey on tree edit distance and related problems. *Theor. Comput. Sci.*, 337(1-3):217–239, 2005.

[18] Andrei Z. Broder, Steven C. Glassman, Mark S. Manasse, and Geoffrey Zweig. Syntactic clustering of the web. *Computer Networks*, 29(8-13):1157–1166, 1997.

[19] Andrei Z. Broder, David Carmel, Michael Herscovici, Aya Soffer, and Jason Y. Zien. Efficient query evaluation using a two-level retrieval process. In *Proceedings of the 2003 ACM CIKM International Conference on Information and Knowledge Management (CIKM)*, pages 426–434, 2003.

[20] Peter Buneman, Susan B. Davidson, Mary F. Fernandez, and Dan Suciu. Adding structure to unstructured data. In *Proceedings of the 6th International Conference on Database Theory (ICDT)*, pages 336–350, 1997.

[21] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, 2011. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

[22] Surajit Chaudhuri, Kris Ganjam, Venkatesh Ganti, and Rajeev Motwani. Robust and efficient fuzzy match for online data cleaning. In *Proceedings of the 22th ACM SIGMOD International Conference on Management of Data*, pages 313–324, 2003.

[23] Surajit Chaudhuri, Venkatesh Ganti, and Raghav Kaushik. A primitive operator for similarity joins in data cleaning. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE)*, page 5, 2006.

[24] Surajit Chaudhuri, Bee-Chung Chen, Venkatesh Ganti, and Raghav Kaushik. Example-driven design of efficient record matching queries. In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB)*, pages 327–338, 2007.

[25] Hsinchun Chen, Wingyan Chung, Jennifer Jie Xu, Gang Wang, Yi Qin, and Michael Chau. Crime data mining: A general framework and some examples. *IEEE Computer*, 37(4):50–56, 2004.

[26] Junghoo Cho, Narayanan Shivakumar, and Hector Garcia-Molina. Finding replicated web collections. In *Proceedings of the 19th ACM SIGMOD International Conference on Management of Data*, pages 355–366, 2000.

[27] Peter Christen. Febrl -: an open source data cleaning, deduplication and record linkage system with a graphical user interface. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Las Vegas, Nevada, USA, August 24-27, 2008*, pages 1065–1068, 2008. doi: 10.1145/1401890.1402020. URL `http://doi.acm.org/10.1145/1401890.1402020`.

[28] Peter Christen. *Data Matching - Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection*. Data-Centric Systems and Applications. Springer, 2012. ISBN 978-3-642-31163-5. doi: 10.1007/978-3-642-31164-2. URL `http://dx.doi.org/10.1007/978-3-642-31164-2`.

[29] Peter Christen and Ross W. Gayler. Towards scalable real-time entity resolution using a similarity-aware inverted index approach. In *Proceedings of the 7th Australasian Data Mining Conference (AusDM)*, pages 51–60, 2008.

[30] Peter Christen, Ross W. Gayler, and David Hawking. Similarity-aware indexing for real-time entity resolution. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management (CIKM)*, pages 1565–1568, 2009.

[31] Rudi Cilibrasi and Paul M. B. Vitányi. A fast quartet tree heuristic for hierarchical clustering. *Pattern Recognition*, 44(3):662–677, 2011.

[32] William W. Cohen. Integration of heterogeneous databases without common domains using queries based on textual similarity. In *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA.*, pages 201–212, 1998. doi: 10.1145/276304.276323. URL `http://doi.acm.org/10.1145/276304.276323`.

[33] William W. Cohen and Jacob Richman. Learning to match and cluster large high-dimensional data sets for data integration. In *Proceedings of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 475–480, 2002.

[34] William W. Cohen, Pradeep D. Ravikumar, and Stephen E. Fienberg. A comparison of string distance metrics for name-matching tasks. In *Proceedings of IJCAI-03 Workshop on Information Integration on the Web (IIWeb-03)*, pages 73–78, 2003.

[35] Timothy de Vries, Hui Ke, Sanjay Chawla, and Peter Christen. Robust record linkage blocking using suffix arrays. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management (CIKM)*, pages 305–314, 2009.

[36] Debabrata Dey, Vijay S. Mookerjee, and Dengpan Liu. Efficient techniques for online record linkage. *IEEE Trans. Knowl. Data Eng.*, 23(3):373–387, 2011.

[37] AnHai Doan, Ying Lu, Yoonkyong Lee, and Jiawei Han. Object matching for information integration: A profiler-based approach. In *Proceedings of IJCAI-03 Workshop on Information Integration on the Web*, pages 53–58, 2003.

[38] AnHai Doan, Ying Lu, Yoonkyong Lee, and Jiawei Han. Profile-based object matching for information integration. *IEEE Intelligent Systems*, 18(5):54–59, 2003.

[39] Xin Dong, Alon Y. Halevy, and Jayant Madhavan. Reference reconciliation in complex information spaces. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 85–96, 2005.

[40] Ahmed K. Elmagarmid, Panagiotis G. Ipeirotis, and Vassilios S. Verykios. Duplicate record detection: A survey. *IEEE Trans. Knowl. Data Eng.*, 19(1):1–16, 2007.

[41] Ronald Fagin and Moshe Y. Vardi. The theory of data dependencies - an overview. In *Automata, Languages and Programming, 11th Colloquium, Antwerp, Belgium, July 16-20, 1984, Proceedings*, pages 1–22, 1984. doi: 10.1007/3-540-13345-3_1. URL http://dx.doi.org/10.1007/3-540-13345-3_1.

[42] Wenfei Fan, Xibei Jia, Jianzhong Li, and Shuai Ma. Reasoning about record matching rules. *Proceedings of 35th International Conference on Very Large Data Bases (VLDB)*, 2(1): 407–418, 2009.

[43] Nicola Fanizzi, Claudia d'Amato, and Floriana Esposito. A hierarchical clustering procedure for semantically annotated resources. In *Artificial Intelligence and Human-Oriented Computing, 10th Congress of the Italian Association for Artificial Intelligence (AI*IA)*, pages 266–277, 2007.

[44] Ivan Fellegi and Alan Sunter. A Theory for Record Linkage. *Journal of the American Statistical Association*, 64(328):1183–1210, 1969. doi: 10.2307/2286061. URL http://dx.doi.org/10.2307/2286061.

[45] Ivan P Fellegi and Alan B Sunter. A theory for record linkage. *Journal of the American Statistical Association*, 64(328):1183–1210, 1969.

[46] Alfio Ferrara, Davide Lorusso, and Stefano Montanelli. Automatic identity recognition in the semantic web. In *Proceedings of the 1st IRSW2008 International Workshop on Identity and Reference on the Semantic Web*, 2008.

[47] Douglas H. Fisher. Knowledge acquisition via incremental conceptual clustering. *Machine Learning*, 2(2):139–172, 1987. doi: 10.1007/BF00114265. URL http://dx.doi.org/10.1007/BF00114265.

[48] Zhichun Fu, Jun Zhou, Peter Christen, and Mac Boot. Multiple instance learning for group record linkage. In *Advances in Knowledge Discovery and Data Mining - 16th Pacific-Asia Conference*, pages 171–182, 2012.

[49] Helena Galhardas, Daniela Florescu, Dennis Shasha, Eric Simon, and Cristian-Augustin Saita. Declarative data cleaning: Language, model, and algorithms. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB)*, pages 371–380, 2001.

[50] Roy Goldman and Jennifer Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *Proceedings of 23rd International Conference on Very Large Data Bases (VLDB)*, pages 436–445, 1997.

[51] Luis Gravano, Panagiotis G. Ipeirotis, H. V. Jagadish, Nick Koudas, S. Muthukrishnan, Lauri Pietarinen, and Divesh Srivastava. Using q-grams in a dbms for approximate string processing. *IEEE Data Eng. Bull.*, 24(4):28–34, 2001.

[52] Luis Gravano, Panagiotis G. Ipeirotis, H. V. Jagadish, Nick Koudas, S. Muthukrishnan, and Divesh Srivastava. Approximate string joins in a database (almost) for free. In *Proceedings of 27th International Conference on Very Large Data Bases (VLDB)*, pages 491–500, 2001.

[53] W3C OWL Working Group. OWL 2 Web Ontology Language Document Overview (Second Edition), 2012. URL `http://www.w3.org/TR/owl2-overview/`. W3C Recommendation 11 December 2012.

[54] Mauricio A. Hernández and Salvatore J. Stolfo. The merge/purge problem for large databases. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 127–138, 1995.

[55] Thomas N. Herzog, Fritz J. Scheuren, and William E. Winkler. *Data quality and record linkage techniques*. Springer, 2007. ISBN 978-0-387-69502-0.

[56] Aidan Hogan, Axel Polleres, Jürgen Umbrich, and Antoine Zimmermann. Some entities are more equal than others: statistical methods to consolidate linked data. In *Workshop on New Forms of Reasoning for the Semantic Web: Scalable Dynamic (NeFoRS10)*, May 2010. URL `http://www.polleres.net/publications/hoga-etal-2010NeFoRS.pdf`.

[57] Wei Hu, Jianfeng Chen, and Yuzhong Qu. A self-training approach for resolving object coreference on the semantic web. In *Proceedings of the 20th International Conference on World Wide Web (WWW)*, pages 87–96, 2011.

[58] Ykä Huhtala, Juha Kärkkäinen, Pasi Porkka, and Hannu Toivonen. Tane: An efficient algorithm for discovering functional and approximate dependencies. *Comput. J.*, 42(2):100–111, 1999.

[59] Hal Daumé III and Daniel Marcu. A bayesian model for supervised clustering with the dirichlet process prior. *Journal of Machine Learning Research*, 6:1551–1577, 2005.

[60] Robert Isele and Christian Bizer. Learning linkage rules using genetic programming. In *Proceedings of the 6th International Workshop on Ontology Matching (OM)*, 2011.

[61] Robert Isele and Christian Bizer. Learning expressive linkage rules using genetic programming. *PVLDB*, 5(11):1638–1649, 2012. URL `http://vldb.org/pvldb/vol5/p1638_robertisele_vldb2012.pdf`.

[62] Matthew A Jaro. Advances in record-linkage methodology as applied to matching the 1985 census of tampa, florida. *Journal of the American Statistical Association*, 84(406): 414–420, 1989.

[63] Raghav Kaushik, Pradeep Shenoy, Philip Bohannon, and Ehud Gudes. Exploiting local similarity for indexing paths in graph-structured data. In *Proceedings of the 18th International Conference on Data Engineering (ICDE)*, pages 129–140, 2002.

[64] Mohammad Reza Keyvanpour, Mostafa Javideh, and Mohammad Reza Ebrahimi. Detecting and investigating crime by means of data mining: a general crime matching framework. *Procedia CS*, 3:872–880, 2011.

[65] Graham Klyne and Jeremy J. Carroll. Resource Description Framework (RDF):Concepts and Abstract Syntax, February 2004. URL `http://www.w3.org/TR/rdf-concepts/`. W3C Recommendation 10 February 2004.

[66] Hanna Köpcke, Andreas Thor, and Erhard Rahm. Evaluation of entity resolution approaches on real-world match problems. *Proceedings of the 36th International Conference on Very Large Data Bases (VLDB)*, 3(1):484–493, 2010.

[67] Nick Koudas, Sunita Sarawagi, and Divesh Srivastava. Record linkage: similarity measures and algorithms. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 802–803, 2006.

[68] Karen Kukich. Techniques for automatically correcting words in text. *ACM Comput. Surv.*, 24(4):377–439, 1992.

[69] Gad M. Landau and Uzi Vishkin. Fast parallel and serial approximate string matching. *J. Algorithms*, 10(2):157–169, 1989.

[70] VI Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707, 1966.

[71] Chen Li, Liang Jin, and Sharad Mehrotra. Supporting efficient record linkage for large data sets using mapping techniques. *World Wide Web*, 9(4):557–584, 2006.

[72] Ee-Peng Lim, Satya Prabhakar, Jaideep Srivastava, and James Richardson. Entity identification in database integration. In *in Proceedings Ninth International Conference on Data Engineering*, pages 294–301. IEEE Computer Society Press, 1993.

[73] Girija Limaye, Sunita Sarawagi, and Soumen Chakrabarti. Annotating and searching web tables using entities, types and relationships. *Proceedings of the 36th International Conference on Very Large Data Bases (VLDB)*, 3(1):1338–1347, 2010.

[74] Cheng-Ru Lin and Ming-Syan Chen. Combining partitional and hierarchical algorithms for robust and efficient data clustering with cohesion self-merging. *IEEE Trans. Knowl. Data Eng.*, 17(2):145–159, 2005.

[75] Yongtao Ma and Thanh Tran. Typimatch: type-specific unsupervised learning of keys and key values for heterogeneous web data integration. In *Sixth ACM International Conference on Web Search and Data Mining (WSDM)*, pages 325–334, 2013.

[76] Yongtao Ma, Thanh Tran, and Veli Bicer. Typifier: Inferring the type semantics of structured data. In *29th IEEE International Conference on Data Engineering (ICDE)*, pages 206–217, 2013.

[77] Malik Magdon-Ismail and Jonathan T. Purnell. Ssde-cluster: Fast overlapping clustering of networks using sampled spectral distance embedding and gmms. In *rivacy, Security, Risk and Trust (PASSAT)/IEEE Third International Conference on Social Computing*, pages 756–759, 2011.

[78] Nikos Mamoulis. Efficient processing of joins on set-valued attributes. In *Proceedings of the 22th ACM SIGMOD International Conference on Management of Data*, pages 157–168, 2003.

[79] Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. The Stanford CoreNLP natural language processing toolkit. In *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 55–60, 2014. URL `http://www.aclweb.org/anthology/P/P14/P14-5010`.

[80] Frank Manola and Eric Miller. RDF Primer, February 2004. URL `http://www.w3.org/TR/rdf-primer/`. W3C Recommendation 10 February 2004.

[81] Andrew McCallum, Kamal Nigam, and Lyle H. Ungar. Efficient clustering of high-dimensional data sets with application to reference matching. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining (KDD)*, pages 169–178, 2000.

[82] Sergey Melnik, Hector Garcia-Molina, and Erhard Rahm. Similarity flooding: A versatile graph matching algorithm and its application to schema matching. In *ICDE*, pages 117–128, 2002.

[83] Matthew Michelson and Craig A. Knoblock. Learning blocking schemes for record linkage. In *Proceedings of the 21th National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference (AAAI)*, 2006.

[84] Alvaro E. Monge and Charles Elkan. An efficient domain-independent algorithm for detecting approximately duplicate database records. In *DMKD*, page 0, 1997.

[85] Gonzalo Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, 2001.

[86] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Mol. Biol.*, 48:443–453, 1970.

[87] Howard B. Newcombe. Record linking: The design of efficient systems for linking records into individual and family histories. *American Journal of Human Genetics*, 19(3): 335–359, May 1967.

[88] Howard B. Newcombe and James M. Kennedy. Record linkage: making maximum use of the discriminating power of identifying information. *Commun. ACM*, 5(11):563–566, 1962. doi: 10.1145/368996.369026. URL `http://doi.acm.org/10.1145/368996.369026`.

[89] Howard B. Newcombe, James M. Kennedy, S.J. Axford, and A.P. James. Automatic linkage of vital records. *Science*, 130(3381):954–959, October 1959.

[90] Terry Ngo. Data mining: practical machine learning tools and technique, third edition by ian h. witten, eibe frank, mark a. hell. *ACM SIGSOFT Software Engineering Notes*, 36(5):51–52, 2011. doi: 10.1145/2020976.2021004. URL `http://doi.acm.org/10.1145/2020976.2021004`.

[91] Patric R. J. Östergård. A fast algorithm for the maximum clique problem. *Discrete Applied Mathematics*, 120(1-3):197–207, 2002.

[92] Gergely Palla, Imre Derenyi, Illes Farkas, and Tamas Vicsek. Uncovering the overlapping community structure of complex networks in nature and society. *Nature*, pages 814–818, 2005.

[93] George Papadakis, Ekaterini Ioannou, Claudia Niederée, and Peter Fankhauser. Efficient entity resolution for large heterogeneous information spaces. In *WSDM*, pages 535–544, 2011.

[94] Bidyut Kr. Patra, Sukumar Nandi, and P. Viswanath. A distance based clustering method for arbitrary shaped clusters in large datasets. *Pattern Recognition*, 44(12):2862–2870, 2011.

[95] Clifton Phua, Kate Smith-Miles, Vincent C. S. Lee, and Ross W. Gayler. Resilient identity crime detection. *IEEE Trans. Knowl. Data Eng.*, 24(3):533–546, 2012.

[96] Vibhor Rastogi, Nilesh N. Dalvi, and Minos N. Garofalakis. Large-scale collective entity matching. *PVLDB*, 4(4):208–218, 2011. URL `http://www.vldb.org/pvldb/vol4/p208-rastogi.pdf`.

[97] Pradeep D. Ravikumar and William W. Cohen. A hierarchical graphical model for record linkage. In *Proceedings of the 20th Conference in Uncertainty in Artificial Intelligence (UAI)*, pages 454–461, 2004.

[98] Eric Sven Ristad and Peter N. Yianilos. Learning string edit distance. In *Proceedings of the Fourteenth International Conference on Machine Learning (ICML)*, pages 287–295, 1997.

[99] Sunita Sarawagi. Information extraction. *Foundations and Trends in Databases*, 1(3):261–377, 2008.

[100] Sunita Sarawagi and Anuradha Bhamidipaty. Interactive deduplication using active learning. In *Proceedings of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 269–278, 2002.

[101] Sunita Sarawagi and Alok Kirpal. Efficient set joins on similarity predicates. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 743–754, 2004.

[102] Anish Das Sarma, Xin Dong, and Alon Y. Halevy. Bootstrapping pay-as-you-go data integration systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 861–874, 2008.

[103] Warren Shen, Xin Li, and AnHai Doan. Constraint-based entity matching. In *the 20th National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference (AAAI)*, pages 862–867, 2005.

[104] Hung sik Kim and Dongwon Lee. Harra: fast iterative hashed record linkage for large-scale data collections. In *13th International Conference on Extending Database Technology (EDBT)*, pages 525–536, 2010.

[105] Parag Singla and Pedro Domingos. Entity resolution with markov logic. In *Proceedings of the 6th IEEE International Conference on Data Mining (ICDM)*, pages 572–582, 2006.

[106] Yannis Sismanis, Paul Brown, Peter J. Haas, and Berthold Reinwald. GORDIAN: efficient and scalable discovery of composite keys. In *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*, pages 691–702, 2006. URL `http://www.vldb.org/conf/2006/p691-sismanis.pdf`.

[107] Aya Soffer, David Carmel, Doron Cohen, Ronald Fagin, Eitan Farchi, Michael Herscovici, and Yoëlle S. Maarek. Static index pruning for information retrieval systems. In *Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*, pages 43–50, 2001.

[108] Dezhao Song and Jeff Heflin. Automatically generating data linkages using a domain-independent candidate selection approach. In *International Semantic Web Conference*, pages 649–664, 2011.

[109] Fabian M. Suchanek, Serge Abiteboul, and Pierre Senellart. Paris: Probabilistic alignment of relations, instances, and schema. *Proceedings of the 37th International Conference on Very Large Data Bases (VLDB)*, 5(3):157–168, 2011.

[110] Erkki Sutinen and Jorma Tarhio. On using q-gram locations in approximate string matching. In *the 3th Annual European Symposium (ESA)*, pages 327–340, 1995.

[111] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining*. Addison-Wesley, 2005. ISBN 0-321-32136-7.

[112] Sheila Tejada, Craig A. Knoblock, and Steven Minton. Learning object identification rules for information integration. *Information System*, 26(8):607–633, 2001.

[113] Sheila Tejada, Craig A. Knoblock, and Steven Minton. Learning domain-independent string transformation weights for high accuracy object identification. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 350–359, 2002.

[114] Esko Ukkonen. Approximate string matching with q-grams and maximal matches. *Theor. Comput. Sci.*, 92(1):191–211, 1992.

[115] Julian R. Ullmann. A binary n-gram technique for automatic correction of substitution, deletion, insertion and reversal errors in words. *Comput. J.*, 20(2):141–147, 1977. URL `http://dblp.uni-trier.de/db/journals/cj/cj20.html#Ullmann77`.

[116] Petros Venetis, Alon Y. Halevy, Jayant Madhavan, Marius Pasca, Warren Shen, Fei Wu, Gengxin Miao, and Chung Wu. Recovering semantics of tables on the web. *Proceedings of the 37th International Conference on Very Large Data Bases (VLDB)*, 4(9):528–538, 2011.

[117] P. Viswanath and V. Suresh Babu. Rough-dbscan: A fast hybrid density based clustering method for large data sets. *Pattern Recognition Letters*, 30(16):1477–1488, 2009.

[118] Gang Wang, Hsinchun Chen, and Homa Atabakhsh. Automatically detecting deceptive criminal identities. *Commun. ACM*, 47(3):70–76, 2004.

[119] Jiannan Wang, Guoliang Li, Jeffrey Xu Yu, and Jianhua Feng. Entity matching: How similar is similar. *Proceedings of the 37th International Conference on Very Large Data Bases (VLDB)*, 4(10):622–633, 2011.

[120] Steven Whang and Hector Garcia-Molina. Entity resolution with evolving rules. *Proceedings of the 36th International Conference on Very Large Data Bases*, 3(1):1326–1337, 2010.

[121] Steven Euijong Whang and Hector Garcia-Molina. Joint entity resolution. In *IEEE 28th International Conference on Data Engineering (ICDE)*, pages 294–305, 2012.

[122] Steven Euijong Whang, Omar Benjelloun, and Hector Garcia-Molina. Generic entity resolution with negative rules. *Proceedings of the 35th International Conference on Very Large Data Bases (VLDB)*, 18(6):1261–1277, 2009.

[123] Steven Euijong Whang, David Menestrina, Georgia Koutrika, Martin Theobald, and Hector Garcia-Molina. Entity resolution with iterative blocking. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 219–232, 2009.

[124] Chuan Xiao, Wei Wang, Xuemin Lin, and Jeffrey Xu Yu. Efficient similarity joins for near duplicate detection. In *Proceedings of the 17th International Conference on World Wide Web (WWW)*, pages 131–140, 2008.

[125] Tengke Xiong, Shengrui Wang, André Mayers, and Ernest Monga. Dhcc: Divisive hierarchical clustering of categorical data. *Data Min. Knowl. Discov.*, 24(1):103–135, 2012.

[126] Rui Xu and Donald C. Wunsch II. Survey of clustering algorithms. *IEEE Transactions on Neural Networks*, 16(3):645–678, 2005.

[127] Su Yan, Dongwon Lee, Min-Yen Kan, and C. Lee Giles. Adaptive sorted neighborhood methods for efficient record linkage. In *ACM/IEEE Joint Conference on Digital Libraries (JCDL)*, pages 185–194, 2007.

[128] Jiang-She Zhang and Yiu-Wing Leung. Improved possibilistic c-means clustering algorithms. *IEEE T. Fuzzy Systems*, 12(2):209–217, 2004.

[129] Ying Zhao and George Karypis. Evaluation of hierarchical clustering algorithms for document datasets. In *Proceedings of the 2002 ACM CIKM International Conference on Information and Knowledge Management (CIKM)*, pages 515–524, 2002.

[130] Yang Zhou, Hong Cheng, and Jeffrey Xu Yu. Graph clustering based on structural/attribute similarities. *Proceedings of 35th International Conference on Very Large Data Bases (PVLDB)*, 2(1):718–729, 2009.

[131] Justin Zobel and Alistair Moffat. Inverted files for text search engines. *ACM Computing Surveys (CSUR)*, 38(2), 2006.

List of Figures

2.1 The Instance Matching Process Applied in This thesis. 10
2.2 A data graph. The solid and dotted line denote the attribute edges and type edges respectively. 12
2.3 The Instance Matching Progress Applied in This thesis. 23
2.4 A Relationship Graph for Collective Instance Matching. 25

3.1 FCG for data in Tab. 2.1. 36
3.2 Algorithm. 43
3.3 Given Hierarchy (Ground Truth). 49
3.4 Hierarchy Generated by TYPifier. 49
3.5 Hierarchy Generated by OPTICS. 49
3.6 Hierarchy Generated by BIRCH 49
3.7 The effect of theta on precision. 50
3.8 The effect of theta on recall. 50
3.9 The effect of epsilon on precision. 50
3.10 The effect of epsilon on recall. 50

4.1 The Effect of θ on Effectiveness. 65
4.2 The Effect of ϵ on Effectiveness. 65

5.1 Consider two rule functions $f_1 = \frac{Jaccard(\texttt{Title})+QGram(\texttt{Manufacturer})}{2}$ and $f_2 = \frac{Jaccard(\texttt{Title})+Cosine(\texttt{Description})}{2}$. Figure (a) and (c) show the histograms for similarities of positive examples calculated according to f_1 and f_2 respectively. And Fig. (b) and (d) show the histograms for dissimilarities of negative examples calculated according to f_1 and f_2 respectively. 84
5.2 PowerCCF estimation for matching certainty $F(x|\mathbf{M}^+)$ (solid line) and non-matching certainty $F(1-x|\mathbf{M}^-)$ (dotted line), where x is similarity. Fig.(a) and (b) illustrate the certainty distributions that refer to $f_1 = \frac{Jaccard(\texttt{Title})+QGram(\texttt{Manufacturer})}{2}$ and $f_2 = \frac{Jaccard(\texttt{Title})+Cosine(\texttt{Description})}{2}$ respectively. 88
5.3 Comparison for the total running time with different proportion of training data. 96
5.4 Evaluation result for different labeling effort 97
5.5 Influence of prior ratio 97
5.6 Influence of ϵ 98

6.1 WCG in Fig. 6.1(a) that is computed based on the attribute `Description` of instances in $M^+ = \{(n_1, n_2), (n_6, n_7)\}$ according to data in Tab. 5.1. Solid lines indicate word pairs co-occurring in examples. Missing lines between any two nodes capture correct CDEs and dotted lines represent incorrect CDEs. Dotted squares indicate words that are not in the examples. 109
6.2 An example of the algorithm 118
6.3 Parameter analysis for SiFi and effiCDE. 120

6.4 Parameter analysis for ST 121
6.5 effiCDE as Filters 125
6.6 Evaluation result for different labeling efforts. 126

A.1 The upper (lower half) of the circle captures word pairs in matches, $\mathbf{C}^+$ (CDEs, $\mathbf{C}^-$). The whole square (circle) represents C ($\mathbf{C}$). The solid (lined) area contains word pairs in the examples (self-matches). Because the learned CDEs are bounded by the square, the lower area outside the square and the upper blank area insides the square captures the false negatives and the false positives, respectively. 151

List of Tables

2.1 Product table; some words later used as features for illustration purposes, are highlighted. 11
2.2 Preprocessing methods. The methods that start with asterisks are proposed in this thesis. 15
2.3 Example blocks for different subtypes using attribute Title as blocking key and the words Sony and VAIO as key values. 20

3.1 Cluster relations based on distance. 39
3.2 Number of instances, triples, schema features (S), value-level schema features (PS) and types and the depth of the hierarchy tree for each dataset. 46
3.3 Optimal parameter settings. 46
3.4 Efficiency of typification in ms. 47
3.5 Effectiveness in terms of precision (P), recall (R) and F-measure (F). * indicates the statistically significant improvements of TYPifier over the best result achieved by the baselines (based on paired t-test with significance at $p < 0.05$) . 48
3.6 Quality (tree edit distance) of hierarchy tree for DBpedia. 48

4.1 For each dataset pair: number of instances, words that appear in attribute values, value-level schema features (PS), and mappings indicating two instances are same (ground truth, GT). 64
4.2 Performance of learning blocking keys in ms. 66
4.3 Performance of blocking in ms. 67
4.4 Effectiveness of blocking in terms of PC and RR, * indicates statistically significant improvements of TYPiMatch over the best baseline, Unsupervised (paired t-test, p<0.05). 67
4.5 Performance of learning instance matching schemes and executing them in ms. 68
4.6 Number of matching candidates and RR. 69
4.7 Effectiveness of instance matching in terms R, P and F; * indicates statistically significant improvements of TYPiMatch over the best baseline, Supervised for AB, and PARIS for DS (paired t-test, p<0.05). 70

5.1 A sample of product instances taken from a real E-commerce database; matching instance pairs are (n_1,n_2), (n_3,n_4), (n_5,n_6), (n_5,n_7), (n_6,n_7). 78
5.2 Similarities calculated according to the similarity function $Jaccard(\texttt{Title})$ and $QGram(\texttt{Manufacturer})$, and the rule function $f = \frac{Jaccard(\texttt{Title})+QGram(\texttt{Manufacturer})}{2}$ for the data in Tbl.2.1. 79
5.3 For each dataset pair: number of instances, all matches $\mathbf{M}^+$, and all non-matches $\mathbf{M}^-$. 93
5.4 Performance of training and testing in seconds. 95
5.5 Effectiveness of instance matching in terms of F-measure. 96

6.1 A summary of notations used in this chapter. 103
6.2 Instances and ground truth (GT). 119

6.3 Performance of learning and instance matching in ms. 122
6.4 Effectiveness of instance matching in terms of P, R and F 124

Appendix A

Algorithm Analysis

A.1 Number of aIR candidates

For aIR-based approaches, the threshold for an attribute is learned from the similarities of *all* training examples that are calculated by *all* similarity functions. Assuming there are m similarity functions, and k training examples, there would be $m \cdot k$ different threshold candidates, which form $m \cdot k$ similarity function predicates for each attribute.

aIR candidates can be generated as conjunctions of similarity function predicates of all attributes. For examples, assume there are two attributes a_1 and a_2, and for each attribute there are $m \cdot k$ similarity function predicates. We can generate aIR candidates by first selecting one of the similarity function predicate of a_1, and then creating a conjunction with each of the similarity function predicates of a_2. In this way, we can generate $(m \cdot k)^2$ different aIR candidates. In general, assume there are l attributes, the total number of aIR candidates is $(m \cdot k)^l$.

A.2 Number of mIR candidates

For our technique, the number of mIR candidates equals to the number of rule functions, because for each rule function we construct only one mIR candidate.

A rule function involves two different elements: (1) a set of similarity functions, and (2) a set of attributes. For example, if we consider a set $\{g_1, g_2\}$ of two similarity functions, and a set $\{a_1, a_2, a_3\}$ of three attributes, then we get a total of 2^3 rule functions by combing each attribute with every similarity functions, such as $f_1 = \frac{g_1(a_1)+g_1(a_2)+g_1(a_3)}{3}$. In general, assume there are l attributes and m similarity functions, the number of rule functions is l^m, which is the same as the number of mIR candidates.

A.3 Time Complexity Analysis

Assume there are l attributes, m similarity functions, and k training examples, we analyze the time complexity of our approach as follows:

CCF estimation: Because interCCF only requires the observed similarities being sorted, the time complexity of interCCF is equal to the time complexity of sorting, which is $O(\log k)$. And because powerCCF requires enumerating all training examples to estimate the parameters, its time complexity is $O(K)$.

CCF value calculation: When calculate the value of CCF for an unobserved similarity, interCCF requires a binary search to find the nearest observed data for interpolation. Therefore, the time complexity of CCF value calculation for interCCf equals to the time complexity of binary search, as $O(\log k)$. For powerCCF, because CCF is directly calculated according to Eq. 5.5, the time complexity is $O(1)$.

Learning threshold: Given a difference restriction ϵ, Alg. 5 actually searches the approximate threshold from maximal $\frac{1}{\epsilon}$ different values. The number of comparisons required by Alg. 5 is $\log\frac{1}{\epsilon}$. Noting the ratio $\frac{F(\bar{\theta}|M^+)}{F(1-\bar{\theta}|M^-)}$ is calculated in each comparison, we also take the complexity of CCF value calculation into account. Therefore, the overall complexity of learning threshold is $O(\log\frac{1}{\epsilon}\log k)$ for interCCF and $O(\log\frac{1}{\epsilon})$ for powerCCF.

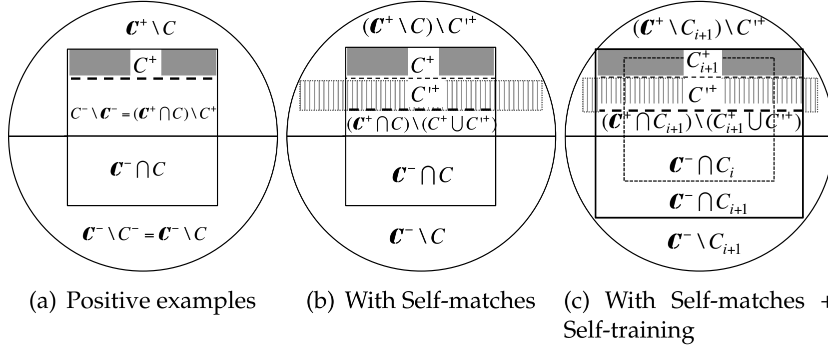
mIR Evaluation: Since the evaluation score of a mIR is calculated as $\bar{Q} = 1 - F(\theta|\mathbf{M}^+)$, the time complexity of mIR evaluation equals to the time complexity of CCF value calculation.

Learn single mIR: For each iteration of Alg. 6, because each of the l attributes will be combined with all the m similarity functions, there are lm mIR candidates to be evaluated in each iteration. Assuming there are t iterations before the termination condition is satisfied, there are overall tlm mIR candidates to be processed. Since for each mIR we execute CCF estimation, threshold learning, and mIR evaluation sequentially, the time complexity for this process is determined by the highest time complexity of each step, which is $O(\log\frac{1}{\epsilon}\log k)$ for interCCF and $O(k)$ for powerCCF. In sum, the overall time complexity of learning a single mIR is $O(tlm\log\frac{1}{\epsilon}\log k)$ for interCCF and $O(tlmk)$ for powerCCF.

A.4 Set-Based Analysis

Here, we provide a set-based analysis and illustration of the effects of using self-matches and self-training.

Only positive examples. Fig. A.1(a) shows CDE learning only using positive examples. The areas representing false positives ($C^- \setminus \mathbf{C}^- = (\mathbf{C}^+ \cap C) \setminus C^+$) and false negatives ($\mathbf{C}^- \setminus C^- = \mathbf{C}^- \setminus C$) are relatively large.



(a) Positive examples (b) With Self-matches (c) With Self-matches + Self-training

Figure A.1: The upper (lower half) of the circle captures word pairs in matches, $\mathbf{C}^+$ (CDEs, $\mathbf{C}^-$). The whole square (circle) represents C ($\mathbf{C}$). The solid (lined) area contains word pairs in the examples (self-matches). Because the learned CDEs are bounded by the square, the lower area outside the square and the upper blank area insides the square captures the false negatives and the false positives, respectively.

Self-Matches. Fig. A.1(b) shows CDE learning with positive examples and self-matches. More word pairs can now be added using self-matches. This is reflected in Fig. A.1(b) by the lined area, indicating that the upper blank area inside the square, i.e. the number of false positives, becomes smaller: $C^- \setminus \mathbf{C}^- = (\mathbf{C}^+ \cap C) \setminus (C^+ \cup C'^+)$. Because the square area is only determined by the positive examples, it does not change with the use of self-matches. Thus, using self-matches has no effect on false negatives.

Self-Matches and Self-Training. With self-training the number of positive examples increases. As a result, the square area in Fig. A.1(c) is enlarged. This translates to a reduced amount of false negatives, i.e. from $\mathbf{C}^- \setminus C_i$ to $\mathbf{C}^- \setminus C_{i+1}$, $C_i \subseteq C_{i+1}$. The set of false positives is $C^- \setminus \mathbf{C}^- = (\mathbf{C}^+ \cap C_{i+1}) \setminus (C^+_{i+1} \cup C'^+)$.