TOWARD ROBUST ENTITY MATCHING SOLUTIONS FOR STRUCTURED AND TEXTUAL DATA

by

Han Li

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

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2019

Date of final oral examination: 06/26/2019

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

AnHai Doan, Professor, Computer Sciences, UW-Madison

Paul Hanson, Professor, Center for Limnology, UW-Madison

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

Theodoros Rekatsinas, Assistant Professor, Computer Sciences, UW-Madison

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to many people who have helped, supported, and encouraged me throughout my Ph.D. journey. This dissertation would not have been finished without them.

First and foremost, I would like to give my forever thanks to my advisor, Professor AnHai Doan, for his never-ending encouragement, support, and guidance. He is such a brilliant, devoted, persistent, and rigorous person, which shows me an exemplar on how a world-class researcher should be. As an academic advisor, he has taught me a lot on how to think, communicate, write, and present. When I got stuck, he always had patience, and inspired me with insights for new research directions. His high standards prompted me to keep improving myself and conduct rigorous research. As a life coach and friend, he taught me how to manage career, time, and marriage. I have to say, he is the best advisor I could have ever asked for.

Next, I would like to thank my thesis committee members, Professors Paul Hanson, Paraschos Koutris, and Theodoros Rekatsinas for their invaluable comments and advice. Beyond the committee, I have been very fortunate to collaborate with Theodoros for the last two years of my graduate study. I have learned a lot from him about machine learning and deep learning, which has been very useful in my research.

Many thanks must also go to my collaborators, friends, and many wonderful people in the database group throughout my graduate study: Adel Ardalan, Jeff Ballard, Sanjib Das, Shaleen Deep, Harshad Deshmukh, Zhiwei Fan, Yash Govind, Pradap Konda, Sidharth Mudgal, Derek Paulsen, Paul Suganthan, Haojun Zhang, and Zuyu Zhang. I have benefited tremendously from the discussions, assistance, and feedback from them.

I would like to thank my parents Wei and Fengxia for their love, patience, and support. When I encountered difficulties in my study, they believed in me, encouraged me, and unconditionally supported me as always.

Finally, I would like to thank my wife Xiaoxue. It is the most wonderful thing to marry her in my life. She never complained for not accompanying her during my study, and always gave me love and support when I needed. I am indebted to her a lot for her love, support and being my best friend in my life.

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

Figure Page

ABSTRACT

Entity matching (EM) is the task of finding data records that refer to the same real-world entity. There have been many EM solutions proposed over the decades. In real applications, oftentimes the developed solutions provide reasonable results. However, there are still many limitations that prevent these solutions from delivering good performance for certain application scenarios. In this dissertation we focus on three limitations: limited support for debugging blocking, difficulties in handling structured but dirty data, and difficulties in handling textual data. To address the first limitation, I develop MatchCatcher for debugging blocking, and implement an open-source tool. The tool has been integrated into the Magellan EM system and used extensively with overwhelmingly positive feedback. To address the second and third limitations, I explore deep learning (DL) for EM. Specifically, I first focus on the matching step in EM, by exploring a DL design space for matching and conducting extensive evaluation. The results show that DL does not outperform the current EM solutions on structured data, but can significantly outperform them on textual and dirty data. Then I apply DL to blocking, by exploring a DL design space for blocking. Comparing against non-DL state-of-the-art solutions, the results show that it is not clear whether DL will help blocking on structured data, but it provides better blocking results on textual and dirty data. Finally, I show that with GPU acceleration, the proposed DL solutions can be executed efficiently.

Chapter 1

Introduction

This dissertation studies entity matching (EM), which is the task of finding data records that refer to the same real-world entity. For example, the two records (Dave Smith, Atlanta, 18) and (David Smith, Atlanta, 18) refer to the same real-world person, while (Dave Smith, Atlanta, 18) and (Joe Wilson, NY, 25) do not.

We begin this chapter by defining the entity matching problem in Section 1.1. Next, we briefly discuss the prior work on EM and three limitations of the existing solutions in Section 1.2. We describe solutions to these limitations in Section 1.3. Finally, we give a roadmap to the rest of the dissertation in Section 1.4.

1.1 Entity Matching

Entity matching (EM) is the task of finding data records that refer to the same real-world entity. Figure 1.1 shows an example on matching two tables A and B that contain person information. The EM task here is to find tuple pairs across A and B that refer to the same real-world person. In this example, there are three matching tuple pairs across A and B, which are connected with red arrows. This

Table A

Name	City	Age
Dave Smith	Altanta	18
Daniel Smith	LA	18
Joe Welson	New York	25
Charles Williams	Chicago	45
Charlie William	Atlanta	28

Table B

Name	City	Age
David Smith	Atlanta	18
Joe Wilson	NY	25
Daniel W. Smith	LA	30
Charles Williams	Chicago	45

Figure 1.1: An EM example.

problem is critical for many Big Data and data science applications. Thus, it has received much attention and is becoming more and more important in the data science era. See [18, 43] for recent books and surveys on EM.

1.2 Prior Entity Matching Work and Its Limitations

EM has received significant attention. Most of the existing EM works develop *algorithmic solutions* for two fundamental steps, blocking and matching (see below), by exploiting rules, learning, clustering, crowdsourcing, external data, etc. [18, 43, 36]. Figure 1.2 shows an example of the typical EM workflow with blocking and matching. Specifically, given the two tables A and B, the goal of blocking is to use domain heuristics to quickly remove obvious non-matching pairs across A and B. In this example, by observing the data we choose to block on the attribute "City" to consider only pairs that have the same city value for matching. This give us three pairs of potential matches, which form a *candidate set*. Blocking can greatly reduce the number of pairs considered in the matching step, drastically reducing the total EM time. As a result, virtually all real-world EM applications use blocking. Once blocking is done, in the next step, we perform matching with a more refined comparison method over the pairs in the candidate set, and classify each of them as a match or non-match. Then all pairs predicted as matches will be returned as the EM result. The focus of the algorithmic EM solutions is on improving accuracy, minimizing runtime, and minimizing cost (e.g., crowdsourcing fee), among others [43].

In real applications, oftentimes the developed solutions provide fairly reasonable results. However, there are still many limitations that prevent these solutions from delivering good performance for certain application scenarios. In particular, we focus on three limitations in this dissertation: *limited support for debugging blocking*, *difficulties in handling structured but dirty data*, and *difficulties in handling textual data*.

Limited Support for Debugging Blocking: The first limitation is that the current EM systems have very limited support for debugging the blocking step. As mentioned earlier, a fundamental step in the EM workflow is blocking, which uses domain heuristics to quickly remove obvious

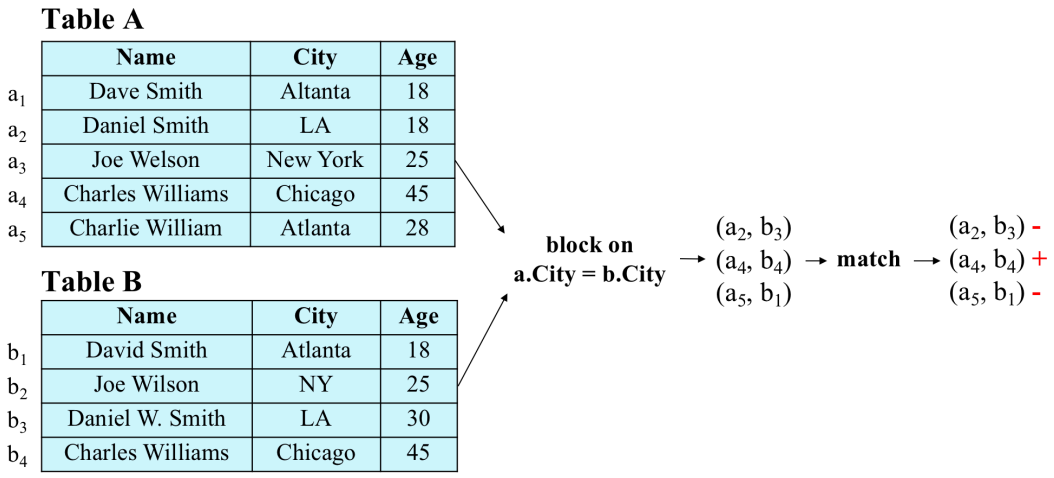

Figure 1.2: An entity matching workflow.

non-matching pairs across A and B. To get good EM results, we prefer to have a blocker that includes as many true matches in the candidate set as possible while minimizing the size of the candidate set. Typically the design of a blocker relies on the domain expertise of the user, and in practice, many different blockers can be applied. For example, in Figure 1.3, we show two different blockers. The first blocker X is a Jaccard similarity blocker defined on the attribute "title", which will only consider pairs for matching if the corresponding Jaccard similarity on the titles of a pair is no less than 0.8. The second blocker Y is an edit distance blocker which will consider pairs for matching if the corresponding edit distance of the titles of a pair is no larger than 2. Suppose the user choose the blocker X, as we do not know the true matches at this point, how does the user know that X will provide better blocking result (e.g., including more true matches in the candidate set) than Y? In fact, the performance of the same blocker, especially regarding the number of true matches included in the candidate set, can vary drastically in different application scenarios, and the case can be even worse if the tables are full of textual,

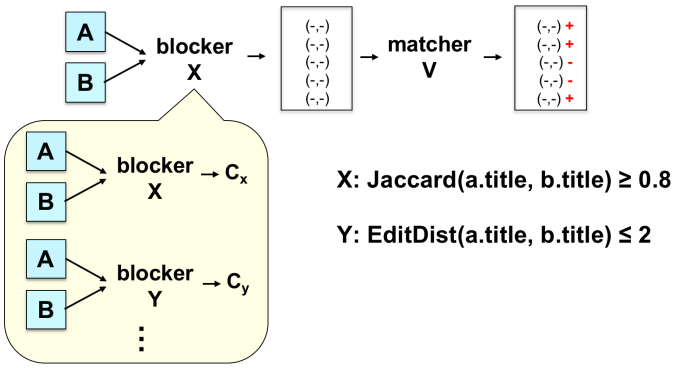

Figure 1.3: An example of debugging blocking.

dirty, misspelt, missing values or synonyms. Therefore, to better understand the performance of a blocker and select a good one, there is a need to debug blocking. While crucial to the blocking performance, debugging blocking has received very little attention to date.

Difficulties in Handling Structured but Dirty Data: The second limitation is that current EM systems have difficulties in handling structured but dirty data. In our work on EM with real users, we have seen many cases where the data to be matched is structured but dirty. Specifically, one type of dirtiness, called *attribute value misplacement*, has been commonly observed in real-world data. Figure 1.4 shows an example of matching a pair of products with this kind of dirtiness. In this example, while the first tuple is clean and structured, the brand value for the second tuple is missing from the cell for that attribute but appears in the cell for name. This commonly arises due to inaccurate information extraction. Current EM solutions cannot match this type of data well, as they generate matching rules or features attribute-wise, and cannot provide good rules or features given this misplacement of attribute values.

Name	Brand	Price
Acrobat 8	Adobe	299.99

Name	Brand	Price
Adobe Acrobat 8		299.99

Figure 1.4: An example of a dirty tuple pair where the second tuple is dirty.

Difficulties in Handling Textual Data: In recent years there is also an increasing demand for matching textual data. Figure 1.5 shows an example of a textual tuple pair about product descriptions. In this example, each description is a blob of raw text. Other examples include matching company homepages with Wikipedia pages that describe companies, matching Twitter user descriptions, and so on. Similar to the structured but dirty data mentioned above, current EM solutions also have difficulties in handling this type of data, because there are few meaningful features that we can create given only the raw text.

Description
Kingston 133x high-speed 4GB compact flash card ts4gcf133, 21.5 MB per sec data transfer rate, dual-channel support, multi-platform compatibility.

Description
Kingston ts4gcf133 4GB compactflash memory card (133x).

Figure 1.5: An example of a textual tuple pair.

1.3 Goals of This Dissertation

In this dissertation, I address the three limitations discussed in Section 1.2 for the current EM systems. Specifically, I have accomplished three projects, which are described below.

A Solution for Debugging Blocking: To address the first limitation, I developed the first blocking debugging solution called *MatchCatcher*. Given two tables to be matched and a blocker, instead of estimating the recall of the blocker, which is difficult, MatchCatcher uses a more practical way that finds likely matches killed off by the blocker, so that the user can examine these matches to understand how well the blocker does recall-wise and what can be done to improve its recall. To be able to quickly find such matches, I proposed a solution that uses string similarity joins, iterative user engagement, rank aggregation, and active/online learning. Extensive experiments show that MatchCatcher is highly effective in helping users develop blockers, and can help improve accuracy of even the best blockers manually created or automatically learned.

This work on debugging blocking has appeared at EDBT'18. I have also implemented the MatchCatcher package in Python. It has been open sourced and used by 300+ students in data science class projects and 7 teams at 6 organizations, and has received overwhelmingly positive feedback.

Exploring Deep Learning for Matching: To address the second and third limitations, I explored using deep learning (DL) for EM. Specifically, as the first step, I focused on applying DL for the matching step. In the past few years, DL has become a major direction in machine learning [88, 132, 57, 163], and yields state-of-the-art results for tasks over data with complex structure, e.g., text, image, and speech. On such data, using labeled examples, it has been shown that DL can automatically construct important features. Given this, DL is a good candidate for handling textual and dirty data.

Thus, I explored using DL for matching to understand the benefits and limitations of DL. Specifically, after reviewing many DL solutions that have been developed for related matching tasks in text processing (e.g., entity linking, textual entailment, etc.), I categorized these solutions and defined a space of DL solutions for matching, as embodied by four solutions with varying

representational power: SIF, RNN, Attention, and Hybrid. Next, I investigated three types of EM problems for which DL can be helpful: structured data instances, textual instances, and dirty instances, respectively. Comparing the above four DL solutions with Magellan [79], a state-of-the-art learning-based EM solution, I showed that DL produces competitive performance on structured EM problems, but it can significantly outperform Magellan on textual and dirty ones. For practitioners, this suggests that they should seriously consider using DL for textual and dirty EM problems. This is a joint work with Sidharth Mudgal, and has appeared at SIGMOD'18.

Exploring Deep Learning for Blocking: So far I have explored using DL for matching, and the results are promising: with little manual feature engineering, DL produces competitive performance against state-of-the-art non-DL work on structured EM problems, and much better performance on textual and dirty EM problems. This suggests that we can effectively use DL to address the second and third limitations for matching. Given this, a natural follow-up question is whether we can use DL for blocking as well.

Following the style of the DL exploration for matching, in the third project I also explored a DL design space for blocking. Specifically, I first defined a DL architecture template, by summarizing existing DL work on blocking as well as distilling and adapting non-DL work. This template consists of three modules: *a word embedding module* to convert each of the given input tuples to a sequence of word embeddings, *a tuple embedding module* to summarize each embedding sequence into an embedding vector, and *a vector-based pairing module* to pair embedding vectors with high similarity scores across two tables, to produce a candidate set. Then I explored a space of DL solutions with multiple choices for each module. Realizing the unsupervised nature of blocking, the DL techniques proposed above for matching cannot be used for tuple embedding summarization. Instead, I selected a set of unsupervised DL approaches for the tuple embedding module choices.

To see if DL can help blocking, I selected five solutions as representatives in the proposed design space, and compared them with RBB, a state-of-the-art non-DL solution. The results showed that while it is not clear whether DL can help blocking on structured data, it does provide better

blocking results on textual and dirty data. Finally, I also show how to execute the proposed DL solutions efficiently using GPU acceleration.

1.4 Roadmap of the Rest of the Dissertation

The next chapter provides background on EM and DL. The following three chapters - Chapter 3-5 - describe the three projects summarized in Section 1.3. Specifically, I describe the Match-Catcher solution to debugging blocking in Section 3, our DL exploration for matching in Section 4, then the solution using DL for blocking in Section 5. I conclude this dissertation in Chapter 6.

Chapter 2

Preliminaries

This chapter provides background materials for EM and DL. Specifically, we first give a more detailed description of the EM workflow (Section 2.1). Then we discuss EM evaluation (Section 2.2). Since we explore DL for EM in Chapters 4 and 5, we introduce the DL basics in this chapter, by giving a brief history lookback (Section 2.3) and describing the DL building blocks (Section 2.4). Finally, we give a brief introduction of GPUs and using GPUs for DL training acceleration (Section 2.5).

2.1 The Entity Matching Workflow

Figure 1.2 shows an example of a typical EM workflow. Given two tables to be matched as input, the EM procedure outputs a set of matching tuple pairs across the two tables. In practice this workflow often consists of two major steps: blocking and matching.

2.1.1 Blocking

Suppose we have two tables A and B with m and n tuples respectively. To find all matching tuple pairs, a naive way is that for each tuple a in A, we compare a with each tuple in B. This brings us a total of $m \times n$ pair comparisons. This may work well when tables A and B are small (e.g., tens or hundreds of tuples in each table). However, even with moderate-size tables that contain thousands of tuples each, it may already produce a large number of comparisons (e.g. millions or tens of millions) which is time consuming, not to mention matching tables with millions of tuples which are quite common in real-world EM tasks.

Realizing that most of the tuple pair comparisons across A and B are non-matches, the goal of the blocking step is to devise a blocker that quickly removes obvious non-matching tuple pairs to greatly reduce the number of pairs considered later for matching. For example, given two tables about person information shown in Figure 1.2, the same person should have the same city value. So we define an Attribute Equivalence (AE) blocker to compare only on city. All pairs that do not have the same city value are unlikely to match and therefore directly removed from a later more expensive comparison. The remaining pairs that survive the blocking step constitute the set of potential matches, and we refer to it as the *candidate set*.

Over the past few decades blocking has received much attention. The focus has been on developing different blocker types and scaling up blockers, e.g., [169, 77, 41, 113, 55] (see [19] for a survey). Many blocker types have been developed. In what follows we briefly discuss the most important types.

Well-known blocker types are attribute equivalence, hash, and sorted neighborhood. *Attribute equivalence (AE)* outputs a pair of tuples if they share the same values of a set of attributes (e.g., blocker $a.City = b.City$ in Figure 1.2). *Hash blocking* (also called *key-based blocking*) is a generalization of AE, which outputs a pair of tuples if they share the same hash value, using a pre-specified hash function (e.g., blocker $lastword(a.Name) = lastword(b.Name)$ performs hashing over the last words of names). *Sorted neighborhood* outputs a pair of tuples if their hash values (also called *key values*) are within a pre-defined distance.

More complex types of blockers include similarity-based and rule-based [18, 55, 30]. *Similarity-based blocking (SIM)* is similar to AE, except that it accounts for dirty values, mispellings, abbreviations, and natural variations by using a predicate involving string similarity measures, such as edit distance, Jaccard, overlap, etc. [177]. Examples include blocker $ed(lastword(a.Name), lastword(b.Name) \leq 2$, which outputs tuple pairs where the last words of their names have an edit distance of at most 2, and blocker $jaccard(a.title, b.title) \geq 0.4$, which outputs pairs of books whose titles have a Jaccard similarity score of at least 0.4. *Rule-based blocking* outputs a tuple pair satisfying a rule or a set of rules encoding domain heuristics. Such blockers can be viewed as the union of multiple blockers, one per rule.

Other types of blockers include phonetic (e.g., soundex), suffix-array, canopy, schema-agnostic blocking [114, 115, 113] etc. (see [18, 43] for an extensive discussion).

Often the design of a blocker mostly relies on the domain expertise of the person conducting the EM task. Therefore, from the machine learning point of view, it is often considered as an unsupervised procedure. That is, we do not take advantage of a set of human labeled data instances to devise the blocker. Recently, there have been some works [55, 59] focusing on learning-based blocking methods with human-in-the-loop, to reduce the requirement on domain knowledge especially for lay users.

2.1.2 Matching

Given a candidate set C of potential matching tuple pairs from the blocking step, we classify each pair in C into two classes, *match* or *non-match*, in this step. A tuple pair is classified as a "match" if the tuples of the pair refer to the same real-world entity, and "non-match" otherwise. In some application scenarios there can be a third class called *potential match* when we are not sure if a pair is a match or not. In this case, a domain expert will manually determine the correct classification for the pair. This manual verification is called "clerical review". In this dissertation we focus on the two-class classification setting as it is more common. Once we finish the classification on the candidate set, all pairs that are classified as matches will be returned as the output of the EM task.

Many different approaches for matching have been proposed, such as rules, supervised learning, active learning, crowd-sourcing etc. (see [18] for a detailed description). Rule-based solutions [45, 139] are interpretable but require the heavy involvement of a domain expert. To address this problem, some work has focused on learning matching functions [9, 79, 140]. For example, a typical supervised learning procedure works as follows. First, a subset S of the candidate set C is randomly selected. Then, we ask humans to manually label each pair in S as match or non-match. Next, given the labeled S as training data, we train a learning-based matcher M (e.g., Random Forest, SVM, etc.). Finally, we apply M to all pairs in C, and output those pairs predicted as match.

A different line of work develops methods to leverage human experts [55, 159, 147] to guide EM and help refine the learned matching functions.

2.2 Entity Matching Evaluation

2.2.1 Evaluation for Blocking

To evaluate the quality of the blocking process, three measures are commonly used: *recall*, *candidate set size ratio*, and *blocking time*.

Recall measures the percentage of true matches caught in the candidate set. Formally, given a candidate set C and a *gold-standard* set $G \subseteq A \times B$ of all true matches across A and B, recall is defined as $|G \cap C|/|G|$. As we want to include all true matches in the candidate set, a higher value of recall is preferred. However, notice that in order to calculate the recall, we need to know the gold-standard G which contains all true matches. To get G, we need to literally go over all pairs in $A \times B$ to identify all true matches, and by doing that we are solving the EM task itself! For this reason, it is infeasible for us to obtain the set G. Therefore in practice we cannot calculate the real value of recall for a blocker.

Candidate set size ratio measures how large the candidate set is. Formally, it is defined as $|C|/|A \times B|$. Since a smaller candidate set often makes the matching step easier (e.g., easier to sample true matches, less time to train, etc.), a smaller value of candidate set size ratio is preferred.

There is a trade-off between improving recall and candidate set ratio. We can easily get 100% recall by simply including all tuple pairs in the candidate set. However the candidate set will be too large for us to handle in the matching step. On the other hand, we can get zero candidate size ratio by having an empty candidate set, but none of the true matches can be caught. Thus we should balance these two measures to get the best performance. In practice, different cases can be made given the requirement of a certain application. For example, in some tasks recall is crucial such that we want to include as many true matches as possible in the candidate set. In this case a larger candidate set is acceptable and often generated. For some other tasks, one may want to finish the EM workflow quickly and lower recall is acceptable, then a smaller candidate will be generated. However, in this case one may run the risk of getting too many true matches "killed

off" by an "aggressive" blocker such that the recall is significantly below the expectation. In real applications, it is often important to know if a blocker is too "aggressive" such that it "kills off" too many matches. This allows us to guarantee (at least) an acceptable recall. While many different blockers have been developed, little attention has been paid to this problem.

Blocking time measures how long it takes for a blocker to produce the candidate set. Typically we want to have the blocking step finished as quickly as possible, therefore a smaller value of blocking time is preferred.

2.2.2 Evaluation for Matching

To evaluate the quality of the matching process, two measures are commonly used: *recall* and *precision*. Recall measures how many true matches in the candidate set are correctly classified as matches. Formally, suppose G_C contains all true matches in the candidate set C and P_m are the pairs in C predicted as match by the matcher. Then recall is calculated as $|P_m \cap G_C|/|G_C|$. As we want to catch all true matches in C, a higher value of recall is preferred.

Precision measures how many of the pairs predicted as match by the matcher are true matches. Formally, precision is calculated as $|P_m \cap G_C|/|P_m|$. A higher value of precision is preferred.

Note that to calculate recall and precision, we need to first get the set G_C of all true matches in C. To do so, we need to manually match each pair in C. Thus we are solving the matching step itself. For this reason, it is typically not feasible to obtain G_C. To address this problem, often a subset E from C is selected and labeled. Then we approximate the recall and precision values by evaluating on the subset E. Specifically, let G_E contain all true matches in E and P_m be the pairs in E predicted as match, then the recall is estimated by $|P_m \cap G_E|/|G_E|$ and precision by $|P_m \cap G_E|/|P_m|$, often with some statistical guarantees.

Similar to blocking, there is a trade-off between improving recall and precision. We can easily get 100% recall by predicting all pairs as matches, but the precision will be very low (assuming only a small number of matches in the candidate set). On the other hand, we can get 100% precision by refusing to make any match prediction, but the recall is zero in this case. Thus we should balance these two measures in order to get the best overall performance. For this reason, another measure

called the F_1 score is often reported in addition to precision and recall. The F_1 score takes into account both precision and recall, and is calculated as the harmonic mean: $F_1 = 2 \cdot \text{precision} \cdot \text{recall}/(\text{precision} + \text{recall})$.

2.3 Deep Learning: A Brief History

The term deep learning was coined to refer to deep *artificial neural networks* (ANNs), i.e., networks with multiple layers of artificial neurons. ANNs trace back to 1943 when McCulloch and Pitts proposed a model for artificial neurons [98] in an effort to explain the inner mechanisms of biological neurons. The McCulloch-Pitts (MCP) neural model is considered the ancestor of modern ANNs.

Despite its prosperity now, deep learning has suffered its ups and downs. The recent survey by Wang et al. [158] describes in detail how deep learning has already suffered two "deep winters". It was not until 1958—15 years after the MCP model—that Rosenblatt introduced the Perceptron model [124] within the context of the human vision system. However, in 1969, Minsky et al. [103] showed that the Perceptron model could not learn simple functions, such as XOR, due to its linear nature. Such results led to the first "deep winter" and little work was conducted on deep learning until the 1980s. However, during that time, the idea of backpropagation [167], the main methodology for training ANNs, was introduced. Yet it did not receive much attention.

Later, in the 80's, complex neural network architectures were introduced. In 1982, Fukushima and Miyake proposed the Neocognitron [49], which later inspired the convolutional neural network (CNN). In 1982, Hopfield designed Hopfield Network, which is a form of Recurrent Neural Network (RNN). In 1985, Ackley et al. introduced the Boltzmann Machine [1] and one year later Smolensky introduced Harmonium [141] which is later known as the Restricted Boltzmann Machine. The same year, backpropagation was reinvented by Rumelhart et al. [126], who demonstrated empirical success in applications like character recognition. Many important prototypes and methods in deep learning were developed in this period, and they laid the foundation for modern deep learning.

Starting in 1990s, statistical learning, a branch of machine learning, was getting popular with the development of Support Vector Machines [26] and Graphical Models [78]. At that time, due to hardware limitations and the small size of available labeled datasets, trained deep neural networks was not outperforming statistical learning methods. The success of statistical learning almost killed deep learning, and research in deep learning was limited. This is known as the second "deep winter". Nonetheless, several influential works were introduced at this time. In 1997, Schmidhuber et al. [66] introduced the Long Short-Term Memory (LSTM) networks, the de-facto standard network in modern NLP. In 1998, LeCun et al. developed deep convolutional neural networks (LeNet) [89]. In 2003, Bengio et al. introduced distributed representations of words for language modeling [7] and presented the first word embeddings with artificial neural networks.

Later, in 2006, the work of Hinton et al. on efficient training of Deep Belief Networks (DBNs) [65] in a layer-wise manner marks the beginning of the modern deep learning era. At the same time, the advances in hardware and the appearance of large-scale labeled datasets offered a breath of fresh air in deep learning. In 2007, NVIDIA releases the CUDA (Compute Unified Device Architecture) programming platform, which allows developers to take advantage of the massive parallel computing power of GPUs for computation-intensive tasks such as matrix algebra. GPUs, with CUDA support, enabled people to scale up training of deeper ANNs leading to several breakthroughs in deep learning. In 2009, Fei-Fei Li et al. launched ImageNet [31], an image database containing over 14 million labeled images that soon became the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) for Computer Vision (CV). Apart from being a benchmark dataset for CV research, ImageNet was the first large-scale labeled dataset that helped propel deep learning to an unprecedented success [149]. Three years later in 2012, Krizhevsky et al. [85] proposed AlexNet, a deep convolutional neural network using GPUs for fast training that won ILSVRC-2012 by outperforming the runner-up with more than 10% gap in test error rate. This was a major breakthrough in deep learning and had tremendous influence on later important works in CV such as GoogLeNet [153], ResNet [62], etc. Finally, important advances started emerging in unsupervised training of deep learning models. The work that stands out is the cat experiment by Google in 2012 [86]: a neural network taught itself to recognize cats by watching Youtube videos.

Given the astonishing results, people started to realize the potential of deep learning and pay attention not only in research but also in industry. Deep learning started being adopted in other areas like Natural Language Processing (NLP). Perhaps what made deep learning (or generally machine learning) well-known to the whole world is the DeepMind AlphaGo [138] in 2016, a Go program which for the first time defeated a top professional human player by a large margin. A few months later, in 2017 Google announced a shift to being an "AI-first" company and soon the focus on AI (especially on deep learning) spread across the industry. Also, the research in deep learning experienced explosive growth. Given the remarkable achievements, deep learning has been dominating the research in AI, and widely used in real world. Deep learning is also influencing other areas including speech recognition [64, 128], self-driving cars [14, 11], databases [165, 180, 162, 172, 93, 95, 83, 40, 107], security [32], etc. Even more classical disciplines like Chemistry [54], Astronomy [131] have adopted the use of deep learning. We now provide more details about the history of deep learning for NLP, as it is the most relevant to the theme of this dissertation.

Deep learning is the de-facto solution to many complex tasks in NLP. After the proposal of distributed word representations by Bengio et al. [7], Mikolov et al. proposed word2vec [100] in 2013. Specifically, Mikolov et al. proposed to use neural networks to learn word embeddings, i.e., to learn a fixed size vector representation of real numbers for each text word. The learned embeddings were shown to capture both syntactic and semantic relationships between similar words in terms of word-tense, gender, etc. This line of work sparked the interest in representation learning and the application of word embeddings to NLP. Word embeddings including GloVe [119] and fastText [10], have become the de facto standard for word representation, and have been widely used in NLP.

Word embeddings form the basis of several types of deep learning models used for NLP tasks. The most prominent model used in NLP corresponds to Recurrent Neural Networks (RNNs) and specifically the Long Short-Term Memory (LSTM) model [66] and its variants. Other popular

models in NLP correspond to Recursive Neural Networks (RvNNs) [142] that like RNNs can capture the hierarchical syntactic structures of sentences. More recently, in 2014, the use of Convolutional Neural Networks (CNNs)—the most successful model in Computer Vision—was proposed as a solution to the problem of sentence classification [74]. Additional models such as encoder-decoder networks also started being used in NLP in 2014. Sutskever et al. [152] proposed the sequence-to-sequence learning framework to map one sentence to another using neural networks, which revolutionized machine translation.

Later, the attention mechanism started becoming a core part of state-of-the-art deep learning architectures for NLP tasks. In 2015, Bahdanau et al. [3] proposed the attention mechanism for machine translation, and soon the idea of attention became very popular. Many different variants of attention mechanisms were developed, and they have been widely used in a wide range of NLP tasks including machine translation, textual entailment, question answering, and so on. In 2016, Google announced the Google Neural Machine Translation model (GNMT) [173] based on the sequence-to-sequence model and attention mechanism. The new model outperformed the traditional sequence-to-sequence model by a large margin, and in some relatively simplistic cases GNMT translations are nearly indistinguishable with human translations [173]. Finally, it was not until recently that model pre-training [25, 33] became popular in the NLP community. For example, a pre-trained encoder will be trained context-agnostically and later fine-tuned for specific tasks. Specifically, the Bidirectional Encoder Representations from Transformers (BERT) [33] based on this idea has been reported to achieve the state of the art on eleven NLP tasks and outperform human performance by 2.0% on the SQuAD v1.1 benchmarking dataset [122] for question answering.

2.4 Deep Learning Building Blocks

2.4.1 Artificial Neural Networks

The term Artificial Neural Networks (ANNs) refers to computational networks represented as graphs of interconnected artificial neurons. The McCulloch-Pitts (MCP) neural model [98] is considered the first artificial neuron model. The idea behind the MCP neuron is to abstract the

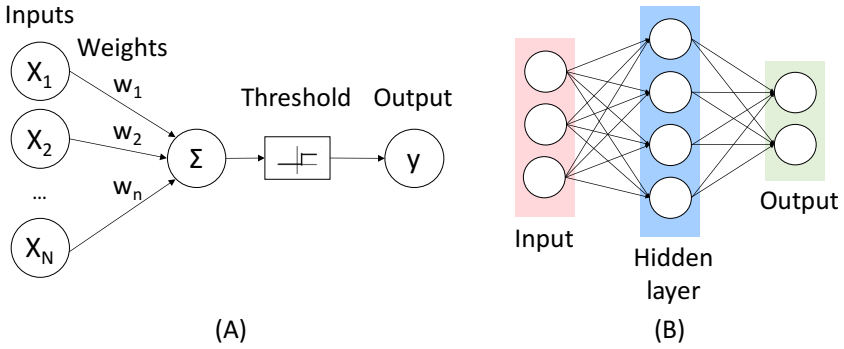

Figure 2.1: (A) The McMulloch-Pitts model for artificial neurons. (B) A simple artificial neural network with one layer of hidden notes. Each node corresponds to an artificial neuron.

biological neuron into a simple mathematical model. The neuron receives incoming signals as 1s and 0s, takes a weighted sum of these signals and outputs a 1 if the weighted sum is at least some threshold value or a 0 otherwise. A diagram of the MCP neuron is shown in Figure 2.1(A). Given this artificial neuron model one can emulate the networks formed by biological neurons by combining multiple artificial neurons to form feedforward networks as the one shown in Figure 2.1(B). This simple artificial neural network model is the predecessor of the modern artificial neural network models we review next.

The Perceptron and Single-Layer ANNs: The Perceptron is a linear classifier (binary) that takes an input $\mathbf{x}$, performs an affine transformation of the input $\mathbf{w}\mathbf{x} + b$, then applies a non-linear *activation* function such as a sigmoid function ($\sigma(x) = 1/(1 + \exp(-x))$) to produce the final output $\sigma(\mathbf{w}\mathbf{x} + b)$. The weights $\mathbf{w}$ and bias b form the parameters of this neural network. Figure 2.2(A) shows an example of Perceptron. It is easy to generalize the simple Perceptron to support multi-dimensional outputs by having multiple output nodes, where each one connects to all input nodes. In this case, one ends up with a layer that connects N input units to M output units. When all input units are connected to all output units, we call this a *fully connected layer* (see Figure 2.2(B)).

Multi-Layer ANNs: Multi-layer ANNs, also know as *multi-layer Perceptrons*, are simply a generalization of the Perceptron where ANN layers are stacked in sequence. In other words, the outputs of each layer become the input to the next layer. The layers between the inputs and outputs

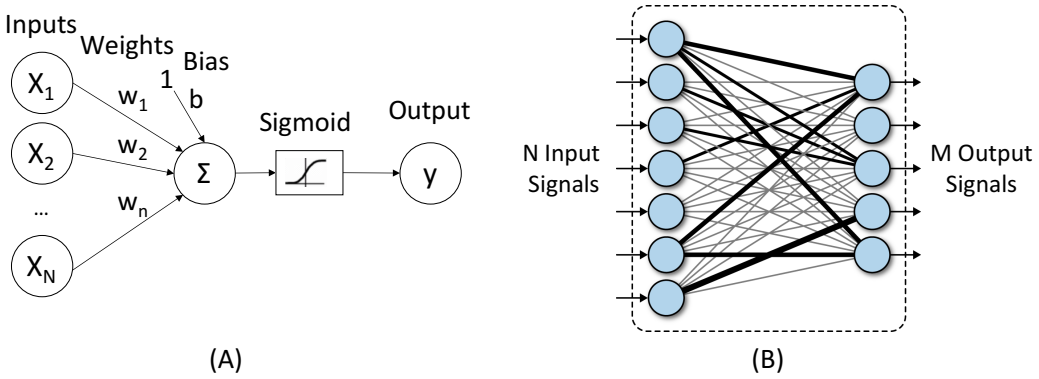

Figure 2.2: (A) The Perceptron model. (B) A fully connected layer with N input units and M output units. Each output unit with the N inputs corresponds to one instance of Perceptron.

are called *hidden layers*. The hidden layers of an ANN enable it to learn increasingly more abstract representations of the data that are "disentangled", i.e., amenable to linear separation. The number of layers in a neural network is what makes it "deep".

In 1989, George Cybenko showed that a feed-forward network with a single hidden layer containing a finite number of neurons can approximate continuous functions on compact subsets of R^n for sigmoid activation functions [27]. Later, in 1991, Kurt Hornik showed that it is not the activation function rather the multi-layer feedforward architecture that gives ANNs the potential of being universal approximators [68]. Function approximation is a powerful capability of neural networks. In theory any function can be approximated, but constructing such a network is not so trivial. One of the key lessons with neural networks is that you cannot blindly create networks and expect them to yield something useful. This is why there are many different activation functions that are being used in modern neural networks. Popular activation functions include the Sigmoid function, the Hyperbolic Tangent (tanh) function, the Rectified Linear Unit (ReLU) [108], etc.

ANN Training and Inference: The training of ANNs corresponds to adjusting the values of the network parameter (or weights). To train an ANN, we first need to define a loss function (or optimization objective) that measures the distance between the predicted values and the target ones. The choice of loss function is dictated by the underlying task the ANN is trained for. For instance,

the mean squared error (MSE) loss can be used for a regression task, or the cross-entropy loss can be used for a classification task. To demonstrate the training procedure, consider the MSE loss as an example. Given n input samples $\mathbf{x_1}, \ldots, \mathbf{x_n}$ and the targets $t_1, \ldots, t_n$, suppose the ANN with the parameters θ is represented by the function $f_\theta(\mathbf{x})$, then the MSE loss with respect to θ is:

$$L(\theta) = \frac{1}{n} \sum_{i=1}^{n} (f_\theta(\mathbf{x_i}) - t_i)^2 \tag{2.1}$$

ANNs are typically trained using mini-batch stochastic gradient descent with backpropagation [88], to minimize the loss. This involves three steps: First, given each input sample, we forward-propagate each instance through the network to compute the target value approximated by the network and to evaluate the loss function with respect to the approximated target and the true target. Second, for each input sample, we compute the gradients of the loss function with respect to all the parameters of the network using backpropagation, i.e., by repeatedly applying the chain rule of differentiation. Third, we update the parameters using an update rule such as mini-batch stochastic gradient descent (SGD). Suppose $L(\theta)$ is the loss function of an input batch, the SGD update is computed as

$$\theta \leftarrow \theta - \alpha \nabla_\theta L(\theta) \tag{2.2}$$

where α is the *learning rate*. We repeat this process for every mini-batch in the dataset. One such pass over the entire training set is called one *epoch*. We train the model for as many epochs as needed until the model converges (e.g., when the accuracy on a holdout set of labeled samples, referred to the *validation set*, stops improving).

Once the training is finished, the inference given a new input sample is pretty straight-forward. We only need to feed the new instance to the input layer and let it run through the whole network to get the output.

The above procedure commonly happens in one kind of learning tasks called supervised learning, where each input sample has the corresponding annotated label. Parameters in the network are optimized in a way to minimize distance between the predicted values and the annotated labels. There are other types of learning tasks like unsupervised learning (e.g., word embedding learning

[100, 119, 10]) and deep reinforcement learning (e.g. used in self-driving cars [129], game playing [138, 5]).

As there are many parameters in the network, the model capacity can be large to easily memorize the whole training dataset, which in turns leads to *overfitting*. Adding regularizations (e.g., Dropout [144], batch normalization [70], early stopping, etc.) has been shown to effectively alleviate this problem to make the trained model generalize well on unseen data. In addition, different optimization procedures have been proposed for training ANNs. Some include stochastic gradient descent with momentum [125], the Adam optimization algorithm [76], the Adagrad optimization method [38] and many more.

2.4.2 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) [89, 85, 153, 62] are designed to process data that exhibit some form of local structure that forms hierarchical representations: For example, in text (1D) sequences of letters form words, sequences of words form sentences that form paragraphs and so on, in images (3D due to multiple color channels) neighboring pixels form edges that form shapes, etc. As inputs can be high-dimensional we refer to them as *volumes*.

Using multi-layer ANNs to represent volumes with such local structure can be impractical to use fully connected layers. For instance, a multi-layer ANN for an 100px $\times 100$px with one fully connected layer with 20k hidden units contains around 200 million parameters. To promote scalability, a CNN relies on the operation of *convolution* to connect each neuron to only a local region of the input [57]. Multiple-layers of convolution-based neurons (referred to as *Convolutional Layers*) are combined to learn the hierarchical representation of the input data. Finally, it is common to use a Pooling Layer, i.e., a layer that aggregates neighboring inputs, in-between successive convolutional layers to progressively reduce the size of the representation to reduce the amount of parameters and computation in the network. A schematic overview of a simple CNN architecture is shown in Figure 2.3. We now review these two core components of CNNs.

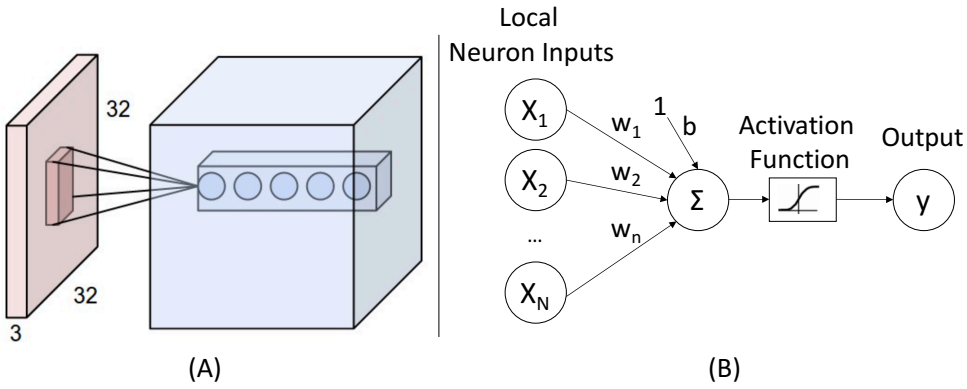

Figure 2.3: (A) An example convolutional layer for an image. Each neuron in the convolutional layer is connected only to a local region in the input, but to all three color channels. There are five neurons looking at the same region in the input. (B) The neuron model remains the same as that of the Perceptron. The neuron still computes a dot product of its weights with the input followed by a non-linearity (activation function), but the neuron connectivity is restricted to be local.

Convolutional Layer: The role of the Convolutional Layer is to extract features from an input volume over its local neighborhoods by convolving the corresponding input with different, learnable *convolution filters or kernels*. Let the dimensions of the input volume be $h \times w \times d$. For example for a 10×10 RGB image we have that $h = w = 10$ and $d = 3$. Each convolution filter is a $p \times q \times d$ volume of learnable parameters (weights) that convolves with each $p \times q \times d$ sub-volume of the input image to produce a volume that is fed as input to an activation function (see Figure 2.3). If we apply the convolution to each possible segment the size of the output matrix for a filter of dimensions $p \times q \times d$ will be $(h - p + 1) \times (w - q + 1)$. In general, convolutional layers consist of multiple learnable filters of the same dimensions $p \times q \times d$. Given f such filters the output of the convolutional layer for an input volume $p \times q \times d$ will be $(h - p + 1) \times (w - q + 1) \times f$. Each cell of this volume can be then given as input to an activation function [108] and the final output corresponds to an output of an artificial neuron. For example, Figure 2.3 shows a convolutional layer with five filters. Overall, the role of the different filters is to extract different features of the input volume. The number of filters used in a Convolutional Layer is a typical hyper-parameter of the layer. This is typically referred to as the *depth* of the convolutional layer. Another important

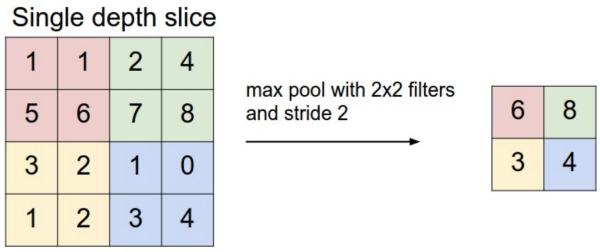

Figure 2.4: An example of a pooling layer filter of size 2×2 using the Max function.

hyper-parameter that control the number of neurons and how the neurons are arranged in the output volume of a Convolutional Layer is *stride*. Stride specifies how slide each filter over the input volume. When the stride is one then we move each filters one position (pixel in an image) at a time. When the stride is two (or uncommonly three or more) then the filters jump two positions at a time as we slide them around. Higher-stride produces output volumes of smaller dimensions.

Pooling Layer: The role of a Pooling Layer is to progressively reduce the size of the representation to reduce the number of parameters and computation in a CNN. The Pooling Layer operates independently on every depth slice of an input volume and resizes it by aggregating a local neighborhood of inputs from the corresponding depth slice. Similar to a Convolutional Layer, a Pooling Layer consists of a filter that defines the local neighborhoods over the input volume. In contrast to Convolutional Layers filters in the Pooling Layer focus on a single depth slice. Each filter corresponds to a matrix $r \times s$ and is associated with an aggregation function (e.g., max or average) that is applied in a segment of size $r \times s$ of each depth channel of the input volume. The most common form is of Pooling is a Pooling Layer with filters of size 2×2 with a Max function. An example of a pooling filter is shown in Figure 2.4. Stride is also a common hyper-parameter of the Pooling Layer.

Pooling can be viewed as adding an infinitely strong prior that the learned Convolutional Layer must be invariant to small modifications to the input. When this assumption is correct, it can greatly improve the statistical efficiency of the network. We refer the reader to [57] for more details.

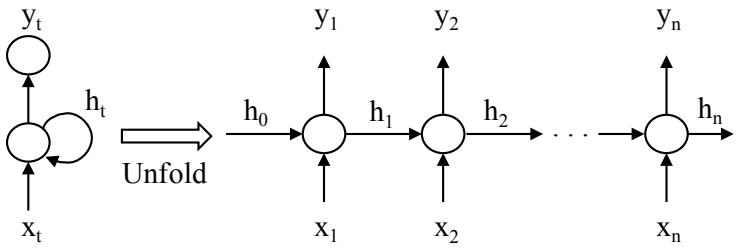

Figure 2.5: (A) An RNN unit with input state x, output state y, and a hidden state h. (B) An unfolded RNN that maps to a feed-forward neural network.

Multi-Layer CNNs: To construct a deep convolutional neural network we proceed similar to Multi-layer Artificial Neural Networks. One can stack convolutional layers followed by a non-linear activation, and then a pooling layer. The idea of deep CNN is shown to be very successful in computer vision. Model variants with stacking tens or even hundreds of CNN layers have been proposed, including the breakthrough work like AlexNet [85], GoogLeNet [153], ResNet [62], etc., and they achieve great performance on standard benchmarking datasets such as ImageNet [127], CIFAR-10 [84], etc. In this dissertation, we do not go into the details of deep CNNs as we do not cover methods using deep CNNs.

2.4.3 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) [94] are designed to process data that is sequential in nature, especially for cases when the sequences have variable length, such as a natural language sentence or a document. Given a sequence of m vectors $\mathbf{x}_1, \cdots, \mathbf{x}_m$ as input with dimension n, an RNN processes it one vector at a time, and produces a sequence of m vectors $\mathbf{y}_1, \cdots, \mathbf{y}_m$ as output with dimension l. At *time-step* t it processes the t^{th} input $\mathbf{x}_t$ to produce the t^{th} output $\mathbf{y}_t$, taking into account all the inputs it has seen so far.

To model sequential structure, RNNs use a *recurrent unit*, which is a neural network that is shared between all time-steps. Figure 2.5(A) shows an example of the recurrent unit (in an RNN). The recurrent unit contains a *hidden state* vector which is updated at every time-step. At time-step t, the recurrent unit takes as input the t^{th} sequence token $\mathbf{x}_t$ and the output of the hidden state from

the previous time-step $\mathbf{h}_{t-1}$. The recurrent unit uses these to inputs to produce the output of the hidden state $\mathbf{h}_t$ for the current time-step t. The hidden state vector of the t^{th} time-step $\mathbf{h}_t$ can be seen as a summary of the inputs $\mathbf{x}_1, \cdots, \mathbf{x}_t$. The output of the RNN for each time-step $\mathbf{y}_t$ is an affine transformation over hidden state output at each time-step $\mathbf{h}_t$.

Concretely, assume that the hidden state at time-step t is a k-dimensional vector $\mathbf{h}_t$. Then, the hidden state and output in $t - th$ step are calculated as:

$$\mathbf{h}_t = \sigma(\mathbf{W}\mathbf{x}_t + \mathbf{U}\mathbf{h}_{t-1}) \tag{2.3}$$

$$\mathbf{y}_t = \mathbf{V}\mathbf{h}_t \tag{2.4}$$

where $\mathbf{W}$ is a matrix of size $k \times n$, $\mathbf{U}$ is a matrix of size $k \times k$, and $\mathbf{V}$ is a matrix of size $l \times k$.

RNNs can also be visualized by *unfolding* them over time, as a multi-layer fully connected network with one layer per input, such that all the layers share their weights and biases. The input to each layer then consists of both the input at that time-step and also the hidden state at that time-step. Figure 2.5(B) shows an example of the unfolded sequence of RNN.

RNNs are trained using a technique called *backpropagation through time* (BPTT) [168]. The key idea behind this is quite simple: we consider the RNN unfolded over time as a regular multi-layer neural network and simply backpropagate through each layer as usual. The only difference is that since the parameters of each layer are shared, we accumulate the gradients of each parameter from all the layers.

One problem with the vanilla RNN described above is that it is difficult to capture long range dependencies between words in the input sequence. This is because an RNN essentially is a traditional multilayer NN with the difference that all the layers share their parameters. As a result it suffers from the problem of vanishing gradients [117] during training, making it difficult to effectively propagate gradient signals through more than a few layers.

To address this, Long-Short Term Memory recurrent units or LSTMs [66] were invented. LSTMs are designed to allow gradient signals to flow through unimpeded across long time-step ranges. This is done by carefully controlling updates to the hidden state by using dedicated sub-neural netwroks called *gates*, and with the help of a *memory* vector which, in a way, serves as long

term memory for the RNN. At each time-step, the LSTM unit takes into account the contents of the memory vector to produce the new hidden state, and also updates the memory vector.

Another important RNN variant is the *Gated Recurrent Units* (GRUs) [17], which is a simplified version of LSTM. GRU has been shown to be comparable to LSTM, and it is computationally cheaper as it has fewer parameters compared to LSTM [20, 164].

An issue with the above RNNs is that they only generate hidden representations from the previous steps. That is, the hidden state $\mathbf{h}_t$ is generated based on the current input $\mathbf{x}_t$ as well as the previous hidden states $\mathbf{h}_1, \ldots, \mathbf{h}_{t-1}$ (which are generated using $\mathbf{x}_1, \ldots \mathbf{x}_{t-1}$). Sometimes we need to learn the representations by looking at future inputs to better understand the context and remove ambiguity. To achieve this, Bidirectional RNNs (BiRNNs) are proposed [133]. In a BiRNN, two separate RNNs are used with one RNN to process the inputs from beginning to the end in the original sequence order (this is called the forward RNN), and the other process the inputs from the end back to the beginning (this is called the backward RNN):

$$\begin{aligned} \overrightarrow{\mathbf{h}_t} &= \overrightarrow{RNN}(\mathbf{x}_t, \overrightarrow{\mathbf{h}_{t-1}}) = \sigma(\mathbf{W}_f \mathbf{x}_t + \mathbf{U}_f \overrightarrow{\mathbf{h}_{t-1}}) \\ \overleftarrow{\mathbf{h}_t} &= \overleftarrow{RNN}(\mathbf{x}_t, \overleftarrow{\mathbf{h}_{t+1}}) = \sigma(\mathbf{W}_b \mathbf{x}_t + \mathbf{U}_b \overleftarrow{\mathbf{h}_{t+1}}) \end{aligned} \tag{2.5}$$

Here $\overrightarrow{RNN}$ represents the forward RNN and $\overleftarrow{RNN}$ represents the backward RNN. By using BiRNN according to Eq. 2.5, for each time step we get two hidden state representations $\overrightarrow{\mathbf{h}_t}, \overleftarrow{\mathbf{h}_t}$, and the final hidden state $\mathbf{h}_t = [\overrightarrow{\mathbf{h}_t}, \overleftarrow{\mathbf{h}_t}]$ can be generated by concatenating the two, and it will summarize the context around $\mathbf{x}_t$ by looking to the past and future.

The BiRNN can be easily adapted to LSTM and GRU mentioned above, to get BiLSTM and BiGRU. And they have been widely used in NLP.

2.4.4 Word Embeddings

Word embeddings form the basis of all state-of-the-art solutions in language modeling and feature learning in NLP (for tutorials and surveys on word embedding, see [47, 13]). A word embedding maps words or phrases from a vocabulary to vectors of real numbers which form the necessary input to machine learning algorithms for NLP tasks. Early work includes one-hot vectors and co-occurrence matrix embeddings. For example, the one-hot vector embedding maps each

word into a vector (with size of number of words in the vocabulary) that contains only 0's and one 1 at the index of that word in the vocabulary. Such embeddings suffers from problems like the embedding vectors do not capture the syntactic or semantic of the words well, and the embedding size grows as the vocabulary expands.

To address the limitations, modern word embeddings learn distributed-similarity-based representations based on the idea that semantically related words are likely to appear in the same context and the meaning of a word therefore can be represented by the neighbors. Each word embedding will be represented by a fixed-dimension vector with the size (e.g., 100) much smaller than the vocabulary size. The reason it is called "distributed" is that certain dimensions may represent some concepts from the training corpus. Therefore modern word embeddings are also known as *distributed representations* of words.

Various word embedding methods have been proposed. Mikolov et al.[100, 101] proposed *word2vec*, which can be trained using either Skip-gram (predicting surrounding context words given a center word) or Continuous Bag of Words (CBOW, predicting a center word form the surrounding context). Another important work is *GloVe* [119] based on the word co-occurrence matrix.

Training on the corpus of billions of words, it has been shown that words that are linguistically related are likewise related in the embedding space. The classic example is the observation in [102] that $E(king) - E(man) + E(woman)$ produces an embedding vector that is close to $E(queen)$. The embeddings have also shown to perform fairly well in answering analogy questions such as Athens : Greece $\approx$ Oslo : Norway and possible : impossible $\approx$ ethical :: unethical.

Word2vec and GloVe are word-level embedding methods, meaning that each word will be associated with an embedding vector. And this makes the learned word embeddings difficult to handle cases like morphologically rich languages (e.g. Turkish) and out-of-vocabulary words. To address this limitation, Bojanowski and Grave proposed a character-level embedding method [10], *fastText*, utilizing the subword information. In character-level embedding, each word is decomposed into a bag of n-grams. Then a word embedding will be learned on each n-gram, and the final embedding for a word will be the sum of the embeddings of its n-grams.

2.4.5 The Attention Mechanism

Traditional RNNs compress the information contained in the input sequence to a fixed-length vector representation. During this process they consider all parts of the input sequence to be equally important. This representation technique can make it difficult for RNNs to learn summarizations from long and possibly noisy input sequences. Recent work [3] has introduced *attention mechanisms* to overcome the aforementioned limitation. An attention model is a method that takes n arguments $y_1, ..., y_n$ and a context c. It returns a vector z which is supposed to be the summary of the y_i, focusing on information that is relevant to c.

Combining attention mechanisms with RNNs allows the latter to "attend" to different parts of the input sequence at each step of the output generation. Importantly, attention mechanisms let the model learn what to attend to based on the input sentence.

Interestingly, the attention mechanism is also getting more and more attention from the researchers in machine learning. Many attention methods [156, 3, 33, 166] have been proposed based on the inputs and network structures. The aforementioned procedure requires a context. Vaswani et al. proposed the self-attention mechanism known as *Transformer* [156], where a summary of the sequence can be obtained by internally attending to the important parts within the sequence. A follow-up work based on self-attention, BERT [33], has reported to achieve the state-of-the-art performance on a series of NLP tasks.

2.5 Graphics Processing Units (GPUs)

A GPU, or graphics processing unit, is a specialized hardware for rapid arithmetic operations. As the name suggests, it was originally designed for display functions like image rendering, video decoding, and so on. Different from a CPU (central processing unit) which has a few complicated cores and runs processes sequentially with few threads, a GPU is composed of hundreds or even thousands of simple cores that allows parallel computing through thousands of threads simultaneously. Given the great capabilities of parallelization, GPU has been increasingly used for general

purpose computing (rather than image processing) for applications that require repetitive, easy-to-parallelize computation, such as matrix algebra, physical modeling, etc. (see [112] for more details on general purpose computation on GPUs, or GPGPU). Nowadays, GPUs play a key role in modern DL.

GPU-accelerated DL Training: In order to achieve great model representation power, it is common to build DL models of tens of millions or even hundreds of millions of learnable parameters. Take the AlexNet [85], a deep convolutional neural network, as an example. There are 60 millions parameters in AlexNet, and it takes months if we train it on a single CPU with at most tens of cores. So one of the major bottlenecks for DL is how to train the DL model efficiently such that the model converges with respect to some optimization objective quickly. To address this, GPUs have been used to accelerate the training of DL models. Since AlexNet, which took advantage of GPUs for fast training, using GPUs to accelerate DL training has gained much attention and now is becoming the de-facto mode. Essentially, both the forward pass for DL model inference and the backward pass for DL model parameter update can be viewed as matrix operations, such as matrix addition, matrix multiplication, Hadamard product, etc., and can indeed be easily broken into a set of small computations that are independent of one another. Therefore it has been shown that DL models are *embarrassingly parallel* (where little or no effort is needed to separate the problem into a number of parallel tasks). This nature makes GPUs so useful in DL training acceleration.

GPU Computing Stack for DL: A GPU itself only provides the parallel computing capability at the hardware level. In order to achieve the acceleration of DL training using GPUs, we also need software support. In this part, we discuss the GPU computing stack for DL training, which consists of three layers: hardware, software and application.

The bottom of the computing stack is the hardware layer that uses the GPU as the hardware to provide the capability of parallel computing, which I described earlier.

Sitting on top of the GPU is the software layer, where software architectures like CUDA have been developed to provide APIs for GPU programming. CUDA, or Compute Unified Device Architecture, is a programming platform released by NVIDIA in 2007, which allows developers to

take advantage of the massive parallel computing power of GPUs (by NVIDIA) for computation-intensive tasks such as matrix algebra. Building up on the software platforms, libraries have been implemented toward DL training optimization. For example, cuDNN, or CUDA Deep Neural Network library, is a GPU-accelerated library for the basic building blocks in DL, such as convolution, pooling, normalization, activation layers, etc.

Finally, on top of the software layer is the application layer. In this layer, DL frameworks like Tensorflow, PyTorch, Caffe, etc. have integrated the libraries like cuDNN from the software layer, to provide high-level abstraction over the DL primitives, to help developers and researchers build DL applications and train them easily with GPU acceleration, without knowing the low-level GPU performance tuning.

Chapter 3

MatchCatcher: A Debugger for Blocking in Entity Matching

3.1 Introduction

Entity matching (EM) finds data instances referring to the same real-world entity [18, 43], such as tuples (Dave Smith, San Francisco, CA) and (David Smith, S.F., CA). This problem is critical for many Big Data and data science applications.

When doing EM, we often must perform blocking. Consider for example matching two tables A and B. Real-world tables often have hundreds of thousands, or millions, of tuples. Trying to match all tuple pairs in $A \times B$ is practically infeasible. So we often perform a step called *blocking* which uses domain heuristics to quickly drop many pairs judged obviously non-matched (e.g., person tuples that do not have the same state). The next step, called *matching*, matches the remaining pairs, using rule- or learning-based techniques. Blocking can greatly reduce the number of pairs considered in the matching step, drastically reducing the total EM time. As a result, virtually all real-world EM applications use blocking.

Numerous blocking methods have been developed [18]. For example, *hash blocking* drops all tuple pairs that do not have the same hash value, using a predefined hash function. This method is popular because it is easy to understand and fast. Other methods include sorted neighborhood, overlap, phonetic, rule-based, etc. (see Section 3.2).

Given two tables A and B to match, we often want a blocker Q that is *fast*, *selective*, and *accurate*. "Fastness" is measured by the time to apply Q to A and B to produce a set of tuple pairs C. "Selectivity" is typically measured as the ratio $|C|/|A \times B|$. "Accuracy" is typically measured

as the fraction of *true matches* surviving blocker Q, i.e., $|M \cap C|/|M|$, where M is the set of (unknown) true matches in $A \times B$. As such, it is also referred to as *recall*.

In practice, blockers can vary drastically in recall, and using a blocker with high recall is critical for many EM applications (see Section 3.2). Yet today there is still no good way to develop such blockers. For example, given the popularity of hash blockers, suppose we have decided to use a hash blocker Q on two tables. While fast, Q may have low recall if the attribute values to be hashed are dirty, misspelt, missing, or have many natural variations (e.g., "New York", "NY", "NYC"). A common way to address this problem is to use multiple hash blockers and take the union of their outputs, to maximize recall. However, even in this case, the recall can still be quite low. For instance, a recent work [30] describes two real-world datasets where extensive effort at combining hash blockers achieves only 38.8% and 72.6% recall. Such low recalls are simply unacceptable for many EM applications. To improve recall, we can revise the current hash blockers, replace some of them, or adding more blockers (of the non-hash types). *To do any of these, however, we need a way to understand whether the current blocker has low recall, and if so, then what the possible problems are, so that we can improve it.*

The MatchCatcher Solution: In this project we take the first step toward solving the above problems. We describe MatchCatcher, a solution to debug blocker accuracy. Given two tables A and B to be matched and a blocker Q, MatchCatcher attempts to find matches that are "killed off" by Q, i.e., those that do not survive the blocking step. We can examine these matches to see if they are indeed true matches, and if so, then why they get killed off by Q. This tells us whether Q has low recall, and if so, then how to improve it. The following example illustrates our solution:

Example 3.1.1. *Consider matching tables A and B in Figure 3.1.a. Suppose a user U begins by creating a blocker Q_1 that keeps only tuple pairs sharing the same value for "City". Figure 3.1.b shows this blocker as $Q_1 : a.City = b.City$. (This is attribute-equivalence blocking, a special type of hash blocking.) Applying Q_1 to A and B produces a set of tuple pairs C_1 (see Figure 3.1.b).*

User U wants to know if blocker Q_1 kills off too many true matches. To answer this, U applies MatchCatcher*, which operates in iterations. In the first iteration,* MatchCatcher *shows the user*

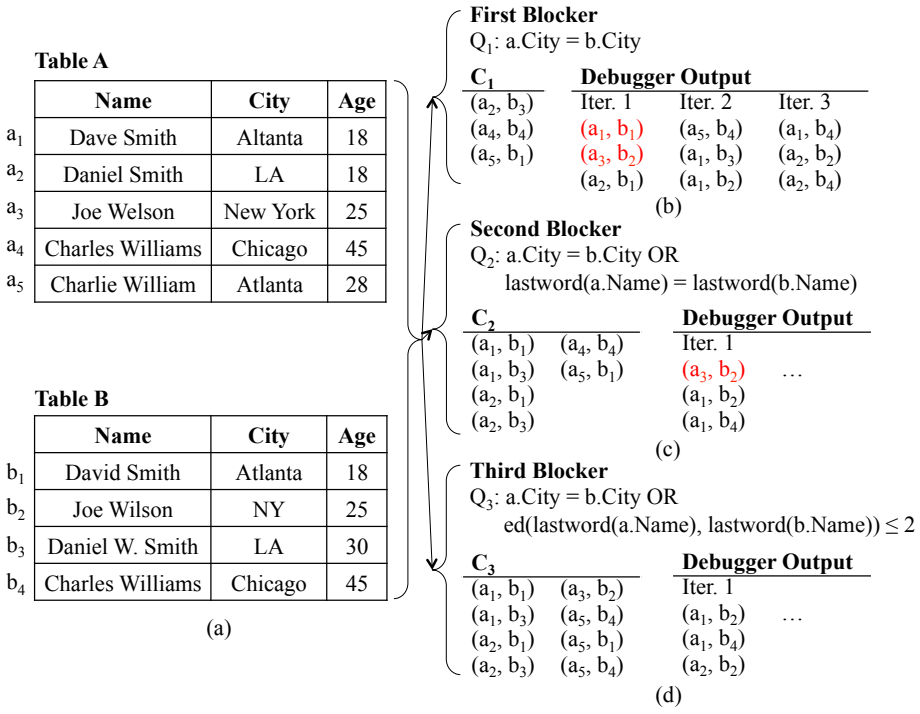

Figure 3.1: An example to illustrate MatchCatcher

n tuple pairs judged most likely to be matches killed off by Q_1. These pairs are listed on Figure 3.1.b, under "Debugger Output, Iter 1" (here $n = 3$).

User U finds that the first two pairs, (a_1, b_1) and (a_3, b_2), are indeed true matches (shown in red color on the figure). A closer examination reveals that they do not survive blocking because their "City" values do not match due to misspellings and abbreviation, e.g., "Altanta" vs. "Atlanta", "New York" vs. "NY".

Next, U wants to know if there are any more true matches. Toward this goal, U flags the true matches in the first iteration (i.e., the above two pairs). MatchCatcher *uses this feedback to find the next n pairs judged most likely to be killed-off matches, then shows those pairs in the second iteration (see Figure 3.1.b, under "Iter 2"). U finds no true matches in this iteration, as well as in the third iteration.*

At this point, U decides to stop looking for more killed-off matches, to focus on revising blocker Q_1 to improve its recall. U observes that the problem with pair (a_1, b_1), which disagree on "City", can be fixed by adding a new hash blocker that blocks on the last word of "Name", i.e., keeps a

tuple pair if they agree on this word (which is typically the last name). Figure 3.1.c shows Q_2, the revised blocker, which is the union of two hash blockers.

Invoking MatchCatcher *for Q_2 produces the list shown under "Debugger Output, Iter 1" in Figure 3.1.c. This list shows that while the new blocker Q_2 successfully keeps (a_1, b_1), it still kills off (a_3, b_2), a true match. A closer examination reveals that this is due to a mispelt last word: "Welson" vs. "Wilson".*

To fix such misspelling problems, U decides to keep a tuple pair if the last words of "Name" are very similar, e.g., within an edit distance of 2. This produces blocker Q_3 in Figure 3.1.d. Here, the hash blocker $lastword(a.Name) = lastword(b.Name)$ has been replaced by the more general blocker $ed(lastword(a.Name), lastword(b.Name) \leq 2$, where ed computes the edit distance (Section 3.2 shows how to execute this blocker efficiently). Invoking MatchCatcher *for Q_3 brings back no true matches, even after several iterations. Thus, user U stops, deciding to use Q_3 as the final blocker for A and B.*

It is important to emphasize that MatchCatcher works with any of the current blocker types. Indeed, it requires as input only the two tables A and B and the set C resulting from applying the target blocker to the tables. Further, MatchCatcher does not estimate the *actual* recall, i.e., the fraction of matches surviving blocking. Doing so would require it to know the set of true matches in $A \times B$, which would be solving the EM problem itself! Indeed, MatchCatcher does *not* attempt to match A and B. Instead, *its goal is to quickly find a large set of plausible matches killed off by the blocker and bring them to the user's attention*, so that the user can examine them to find true matches, get a sense about whether the blocker kills off too many such matches, and if so, what the problems are, so that he/she can fix them. Section 3.6 shows that real-world users indeed find MatchCatcher very helpful in answering these questions.

Challenges: While promising, developing MatchCatcher raises difficult challenges. First, we must quickly search the vast space $D = A \times B - C$ (where C is the blocker's output) to find plausible matches killed off by the blocker, and we must do so without materializing D. This search is further complicated by the fact that at this point MatchCatcher does not even know what it means to be a match (only the user knows). To address these problems, we observe that

matching tuples tend to have similar values for certain attributes (e.g., Name, City). So we convert each tuple into a string that concatenates these attributes, e.g., converting tuple a_1 of Table A in Figure 3.1.a into "Dave Smith Altanta". We then perform a *top-k string similarity join (SSJ)* to find the k tuple pairs with the highest score with respect to these strings, and output these pairs as plausible matches. The state-of-the-art solution for top-k SSJs [174] proves too slow for our interactive setting. So we develop a new solution that is significantly faster.

Second, to find as many plausible matches as possible, we need to repeat the above procedure, but for different sets of attributes (e.g., find tuple pairs that are similar with respect to Name only, City only, both Name and City, etc.). We cannot consider all such sets, called *configs*, as there are too many. So we develop a solution to find a good set of configs.

Third, we must perform multiple related top-k SSJs, one for each config. This raises the challenge of how to perform them *jointly* across the configs. We develop an efficient solution that perform them in parallel on multiple cores yet reuse computations across the joins.

Finally, top-k SSJs over the configs produce a large set E of plausible matches (e.g., in the thousands). We cannot realistically expect the user to examine all of these matches. So we develop a solution that uses rank aggregation and active/online learning to rank the pairs in E, show the top n pairs to the user, ask him/her to identify the true matches, use this feedback to rerank the pairs, and so on, until the user has been satisfied or a stopping condition is reached. In summary, we make the following contributions:

- We show that debugging blocker accuracy is critical for EM.
- We describe MatchCatcher. As far as we know, this is the first in-depth solution to address the above problem. Our solution advances the state of the art in top-k string similarity joins, and exploits active/online learning to effectively engage with the user.
- Over the past two years, MatchCatcher has been successfully used by 300+ students in data science projects and by 7 teams at 6 organizations. We briefly report on this experience. We also describe extensive experiments showing that MatchCatcher is highly effective in

helping users develop blockers, and that it can help improve the accuracy of even the best blockers manually created or automatically learned.

3.2 Debugging Blocker Accuracy

In this section we show that debugging blocker accuracy is critical for EM, discuss the limitations of current solutions, then provide an overview of the MatchCatcher solution.

Types of Blockers: As discussed in the introduction, for large tables A and B we typically perform EM by creating a blocker Q, apply Q to A and B to produce a relatively small set of tuple pairs C, then apply a matcher to pairs in C. Over the past few decades blocking has received much attention. The focus has been on developing different blocker types and scaling up blockers, e.g., [169, 77, 41, 113, 55] (see [19] for a survey).

Many blocker types have been developed. MatchCatcher works with all of them. In Section 2.1 we discussed the most important types, as Section 3.6 experiments with many of them.

Efficient Execution of Blockers: Efficient techniques have been developed to execute the above blocker types, both on a single machine and a cluster of machines (e.g., [77, 41, 113, 55, 30]). To execute hash/AE blocking, we partition the tuples in A and B into *blocks*, such that all tuples in each block share the same hash value, then output only pairs of tuples that are in the same block.

To execute a SIM blocker, such as $ed(lastword(a.Name), lastword(b.Name) \leq 2$, we build an index I (e.g., prefix filtering index [177]) on the tuples in A, say. Next, for each tuple $b \in B$, we consult I to identify all tuples $a \in A$ such that the pair (a, b) can possibly satisfy $ed(lastword(a.Name)$, $lastword(b.Name) \leq 2$. We check if (a, b) indeed satisfies this predicate, and if yes, then output the pair. Many efficient string indexing techniques [177] can be used to implement SIM blockers. Recent work [30] has also discussed efficient techniques (e.g., using indexing and MapReduce) to execute rule-based blockers.

Accuracy of Blockers: Blocker accuracy is typically measured using recall, which has been defined in Section 2.2. Due to dirty data, misspellings, natural variations, synonyms, missing

values, etc., no single blocker type produces the highest recall on all datasets. In fact, on any particular dataset, blockers can vary drastically in recalls (e.g., 2.5-98.2% in our experiments).

Finding a blocker with high recall (ideally 100%), however, is critical for many EM applications. Counter-terrorism EM applications often need very high coverage., i.e., finding *all* person descriptions that match, and thus want 100% blocking recall. Similar high-coverage examples arise in fraud detection, e-commerce, law, medicine, insurance, and pharmaceutical industry, among others. EM applications with inherently small numbers of matches naturally do not want the blocker to kill off many of these. Finally, EM applications often compute statistics over the matches (e.g., the percentage of patients attending both hospitals), which can be seriously distorted by blockers with low recall.

Limitations of Current Work: As a result, the topic of blocker accuracy has received growing attention. Proposed solutions include combining multiple blockers to maximize recall (e.g., [63, 77, 42]), and using a sample of tuple pairs labeled as match/no-match to learn blockers with high recall [8, 99, 55, 30].

While promising, these solutions can still produce blockers with varying recalls, oftentimes falling short of 100%. For example, a recent work [30] shows that extensive manual effort to combine hash blockers achieves only 38.8% and 72.6% recall on two datasets. (Obviously we cannot combine *all* possible blockers as there are too many of them.) Another recent work [55] learns blockers using samples labeled by crowdsourcing, but achieves only 92% recall on a data set. In general, due to the difficulties in obtaining a good sample, sampling flukes, etc., today there is still no guarantee that a blocker learned on a *sample* provably achieves high recall when applied to the original tables.

Since there is still no "fool-proof" method to develop a blocker with high recall, it follows that given a blocker Q (either created manually or learned), it is still highly desirable to know how well Q does recall-wise, and what the possible problems are, so that we can improve it. MatchCatcher helps answer these questions, and thus can be considered *complementary* to the above solutions. For example, Section 3.6 describes a scenario where after the solution in [30] had been used to learn

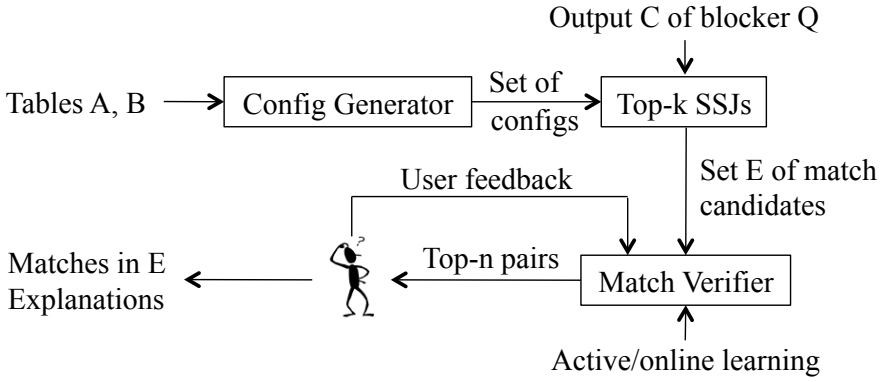

Figure 3.2: The MatchCatcher architecture

a blocker, we applied MatchCatcher to this blocker and uncovered multiple problems, which can be addressed to further improve the blocker recall.

Overview of MatchCatcher: As discussed, MatchCatcher addresses the following problem:

Definition 1 (Finding killed-off matches)**.** *Let C be the output of applying blocker Q to tables A and B. Then $D = A \times B - C$ is the set of all pairs killed off by Q. Quickly find as many true matches as possible in D (without materializing it). This helps the user understand how well Q does recall-wise, and what can be done to improve its recall.*

Figure 3.2 shows the architecture of MatchCatcher. Given two tables A and B, the Config Generator examines the two tables to generate a set of configs, each of which is a set of attributes (e.g., $\{Name, City\}$). For each config g, the Top-k SSJs module performs a top-k string similarity join to find the k tuple pairs that (a) have the highest score with respect to the attributes in g, and (b) are killed off by blocker Q. Note that to check Condition (b), this module does not need to know Q. It only needs to know C, the output of applying Q to A and B. Hence MatchCatcher works independently of the blocker type.

The Top-k SSJs module sends all top-k lists (one per config) to the Match Verifier. This module uses a rank aggregator to combine the lists into a single global list, shows the top n pairs to user U, asks U to identify true matches, uses this feedback together with active and online learning to rerank the pairs, and so on, until U is satisfied or a stopping condition is met. U can examine the true matches to understand how well blocker Q does recall-wise, and to obtain explanations for

why these matches are killed off. This helps U decide if Q should be revised, and if so, then how. The next few sections describe MatchCatcher in detail.

3.3 Generation of Configurations

We now describe the Config Generator, which outputs a set of configs, each being a set of attributes. We cannot consider all possible configs, so the key challenge is to select a good subset of configs. We show how to do so, by carefully managing attributes with many missing values, few unique values, or long string values.

3.3.1 How Configurations Are Used

We first motivate the notion of configurations (or "configs" for short) and explain how they are being used. Later we build on this to discuss how to select a good set of configs.

Recall that we want to quickly search $D = A \times B - C$, the set of tuple pairs killed off by blocker Q, to find pairs that can be matches. This raises three problems. First, D is not materialized, we only have A, B, and C. Second, even if D is materialized, it would be too large to search quickly. Finally, we do not even know what to search for, since at this point MatchCatcher does not know what a match is (only the user knows).

To address these problems, we begin by assuming that tables A and B share the same schema S (MatchCatcher can be trivially extended to the case of different schemas). We observe that matching tuples tend to share similar values in a set of attributes, say g (e.g., $\{Name, City\}$). So we want to quickly find tuple pairs in D that share similar values for g and return those as possible matches.

To do so, we convert each tuple a in A into a string $str_g(a)$ that concatenates the values of all attributes in g. For example, if a is (David Smith, Atlanta, 43) and $g = \{Name, City\}$, then $str_g(a)$ is "David Smith Atlanta". This converts Table A into a set A_g of such strings. We convert Table B into a set B_g of strings similarly.

Let $h(x, y)$ be a string similarity measure which computes a score in $[0, 1]$ between two strings x and y. Examples of such measures are Jaccard, cosine, overlap, edit distance, etc. [177]. Then

next we perform a top-k string similarity join (SSJ) between A_g and B_g to find the k tuple pairs in $A \times B$ with the highest $h(x, y)$ score. Techniques have been developed to quickly perform top-k SSJs [174, 178]. Of course, our goal is not to find pairs in $A \times B$, but rather in $D = A \times B - C$. We can modify the above techniques slightly to ensure this, by dropping a found pair if it is in C. We then return the k pairs in D with the highest $h(x, y)$ score as possible matches.

The above procedure does not require a materialized D, only tables A, B, and C (the output of blocker Q). It can quickly search D using a modified version of top-k SSJs to return possible matches. Of course, at this point we still do not know if these are indeed matches. But later we can work with the Match Verifier to quickly shift through them to find true matches, if any. We now discuss several important aspects of the above procedure.

Why Concatenating the Attributes? We can use a variety of methods to find tuples that share similar values for attributes in g, e.g., finding pairs that share similar values for *each* attribute in g, then taking their intersection, say. However, given the interactive nature of debugging, we want this step to be as fast as possible. Hence we decide not to treat the attributes in g separately, but concatenate all of them into a single string, then compare them using SSJs. Section 3.4 shows that this method can quickly search a very large set D. But a drawback is that we can return *false positives* such as tuple pair (Jim Madison, Smithville, 32) and (Jim Smith, Madison, 32), because their concatenated strings are very similar given certain similarity measures. Such false positives, however, can be "weeded out" in the Match Verifier, using user feedback and active/online learning (see Section 3.5).

Which String Similarity Measure to Use? Given that similar attribute values can still vary significantly (e.g., "Dave Smith" vs "David Frederic Smith"), we believe that measures that treat strings as sets (e.g., Jaccard, cosine, etc.) work better than those that treat strings as sequences of characters (e.g., edit distance). So for MatchCatcher, we use the well-known Jaccard measure that tokenizes two strings x and y into two sets of words P_x and P_y, then returns $|P_x \cap P_y|/|P_x \cup P_y|$ [174]. However, Theorem 2 shows that our solution can also work with other set-based similarity measures, namely overlap, cosine, and Dice [174].

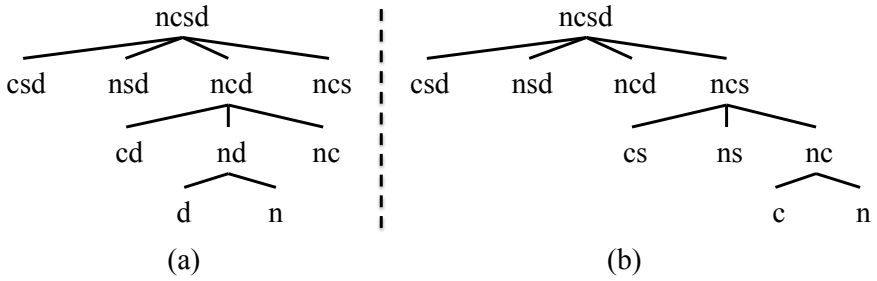

Figure 3.3: An example of generating config trees.

Why Multiple Configurations? So far we have used just one config g to find match candidates. Using multiple configs, however, can produce more matches. For example, config $\{Name, City\}$ may not return the pair (David Smith, Seattle) and (Dave Smith, Redmond) because the cities are different. Config $\{Name\}$ however can. Conversely, config $\{Name\}$ may not return the pair (Chuck Smith, San Francisco) and (Charles F. Smith, San Francisco) because the names are too different, but config $\{Name, City\}$ can. Together, these two configs can return more matches than either of them in isolation. Generating a good set of configs however is a major challenge, which we address next.

3.3.2 Generating Multiple Configurations

As a baseline, we can use all subsets of attributes in S (the schema of A and B) as configs. But this generates too many configs even for a moderate size (e.g., $|S| = 10$ produces $2^{|S|} - 1 = 1023$ configs). We cannot use all of them because the total SSJ time would be too high. So we must select a smaller set of configs.

To do so, we select a set of promising attributes in S, then use them to generate configs, in a top-down fashion. In each step of the process, we select which configs to generate next by carefully considering the impact of attributes with many missing values, few unique values, or long string values. The end result is a *config tree* consisting of multiple configs. Later the Top-k SSJs module will traverse this tree to perform top-k SSJs on the configs in a joint fashion. We now elaborate on these steps.

Selecting the Most Promising Attributes: We first classify attributes in S as string, numeric, categorical, and boolean, using a rule-based classifier. Next, we drop numeric attributes (e.g.,

Salary, Price) because matching tuples still often differ in their values (e.g., the same product having different prices). Finally, we drop categorical and boolean attributes whose appearances in tables A and B are different. For example, if Gender has values $\{Male, Female\}$ in A but $\{M, F, U\}$ in B, then we drop it as these two sets share no value (in general if the Jaccard score of these two sets is less than a pre-specified threshold then we drop the attribute). The remaining attributes are string, or categorical/boolean but with similar sets of values. We return these as T, the set of the most promising attributes to be used for config generation. (Of course, the user can also manually curate schema S to generate T. The experiments in Section 3.6 however do not involve any manual curation.)

Generating a Config Tree: Given the set T of promising attributes, we generate a *config tree* in a top-down fashion, then return all configs in the tree. Specifically, we start with T as the config at the root of the tree. Next, we "expand" this node by removing each attribute from T to obtain a smaller config of size $|T| - 1$. This produces $|T|$ new configs, which form the nodes at the *next level* of the tree. We then select just one node at this level to "expand" further, and so on (we will discuss how to select shortly). This continues until we have reached configs of just one node. Figure 3.3.a shows an example config tree, assuming $T = \{n, c, s, d\}$ (which stand for Name, City, State, and Description, respectively).

Intuitively, this strategy ensures that we generate a diverse set of $|T|(|T| + 1)/2$ configs of varying size $|T|, |T| - 1, \ldots, 1$. The config tree will also be used to guide the joint execution of top-k SSJs on the configs (see Section 3.4.2). We now turn to the challenge of how to select a node to expand in the config tree.

Managing Many Missing Values and Few Unique Values: Consider again the config tree in Figure 3.3.a. Suppose we are currently at the second level of the tree, and need to select one node among the four nodes csd, nsd, ncd, and ncs, to expand. This selection is equivalent to *selecting an attribute to exclude from subsequent config generation*. Indeed, if we exclude attribute s, then we select node ncd to expand (as shown in the figure). Otherwise if we exclude d, then we select the rightmost node ncs to expand, and so on.

So which attribute should we exclude? We observe that if an attribute has many missing values, then keeping it for subsequent config generation is not desirable, because we will end up with configs that produce similar top-k lists. For example, suppose we have selected config ncd to expand (as shown in Figure 3.3.a), and suppose that d has many missing values, then many strings for config ncd and config nc will be identical, potentially leading to similar top-k lists. In the extreme case, if all values for d are missing, then these two top-k lists are identical. Clearly, we want different configs to produce substantially different top-k lists, to avoid redundant work and to maximize the number of matches we can retrieve.

Another observation is that if an attribute has more unique values than another, e.g., c vs s (which stand for City and State, respectively), then it is better to exclude s, the one with fewer unique values, because intuitively, if two tuples agree on c, they are more likely to match than if they agree on s, all else being equal. Thus, to maximize the number of matches we can retrieve, we should strive to keep the "more specific" attributes, i.e., the ones with more unique values.

Combining the above two observations, we define the e-score (shorthand for "expected benefit") of an attribute as follows:

Definition 2 (E-score of an attribute)**.** *Let $n_A(f)$ be the ratio of the number of non-missing values of attribute f in A over the number of tuples in A, and $u_A(f)$ be the ratio of number of unique values of f in A over the number of non-missing values of f in A. We define $n_B(f)$ and $u_B(f)$ similarly. Define $e_A(f) = 2n_A(f)u_A(f)/[n_A(f) + u_A(f)]$ and define $e_B(f)$ similarly. Then we define the e-score of attribute f as $e(f) = e_A(f)e_B(f)$.*

We then select the attribute with the lowest e-score to exclude at each level of the config tree. For example, suppose $e(n) > e(d) > e(c) > e(s)$. Then at the second level of the tree in Figure 3.3.a, we exclude attribute s, which means selecting node ncd to expand. At the third level of the tree, we exclude c, which means selecting node nd to expand.

Managing Long String Attributes: Many datasets contain attributes with long string values, e.g., Comment, Desc, etc. Figure 3.4 shows two tuples where attribute Desc has such long values. Such long attributes can cause two problems. First, they can cause multiple configs to generate very similar top-k lists.

Example 3.3.1. *Consider again the config tree in Figure 3.3.a. Suppose attribute d has long string values (such as those shown in Figure 3.4). Then all seven configs involving d can generate similar top-k lists because the long values of d "overwhelm" the short values of the remaining attributes. So when moving from a config involving d to another (e.g., from ncd to nd), the strings do not change much, and therefore their similarity scores also do not change much (we formalize this notion below), leading to similar top-k lists.*

The second problem is that if the long string values are different for matching tuples, then a config involving this long attribute will fail to return the match. For example, the two tuples in Figure 3.4 match, but any config involving attribute Desc will not return this match, because the values for Desc here are very different, and so the score between the two tuples will be low.

Name: Bryan Lee, **City**: Austin, **State**: TX,
Desc: Joined in 8/2003, promoted to team lead 5/2005, promoted to director of sales 4/2009. Currently on unpaid leave until 1/2013.

Name: Bryan M. Lee, **City**: Austin, **State**: TX,
Desc: Outstanding customer service record 03-05. Achieved sales of $2M/year 05-09. Shortlisted for VP of sales 2011. Shortlisted for VP of marketing 2012.

Figure 3.4: Examples of tuples with long string attributes.

To address this, we modify our config-tree generation procedure as follows. Suppose we need to select a config node in the tree to expand. Before, we select $g_{default}$, the one without the attribute with the smallest e-score. Now, we first run a procedure FindLongAttr to see if there is any attribute that is "too long" (i.e., likely to adversely affect selecting good configs). If such an attribute f_{long} exists, then we select the config without f_{long} to expand. Otherwise we select $g_{default}$, as usual.

Example 3.3.2. *Consider again Figure 3.3.a, which shows the "default" config tree with root $ncsd$. To handle long attributes, once we are at the second level, we do not automatically select ncd (the config without s, the attribute with the smallest e-score) for expansion. Instead, we run* FindLongAttr *at this level. Suppose it returns d (thus judging d to be too long). Then we select ncs, the config without d, for expansion. This produces new configs cs, ns, and nc (see Figure 3.3.b). Suppose running* FindLongAttr *at the level of these new configs returns no attribute. Then*

we select config nc (the config without s, the attribute with the smallest e-score) for expansion (see Figure 3.3.b).

We now explain procedure FindLongAttr. The key challenge is to formalize what it means to be "too long". Let p be a node in the config tree. Suppose that when running the default config generation procedure (the one that does not consider long attributes), we end up selecting q, a child node of p, for expansion, and that we eventually generate a subtree T_q rooted at q (see Figure 3.5.a).

We say that an attribute f is too long if it "overwhelms" many config nodes in subtree T_q, specifically if it overwhelms at least half of the configs in $F(T_q)$, the set of configs in T_q that contain f. In turn, we say that f overwhelms a config $r \in F(T_q)$ (see Figure 3.5.a) *if the top-k list obtained from config r is "roughly the same" as the top-k list obtained from config q (we formalize this below).* Intuitively, we want to avoid such cases, because we want each config to return a different top-k list, to maximize the number of true matches that we will find. So if we find that f overwhelms at least half of the configs in $F(T_q)$, then we judge f to be too long and should be removed. That is, instead of selecting q for expansion, we will select the config (in the same tree level as q) that does not contain f.

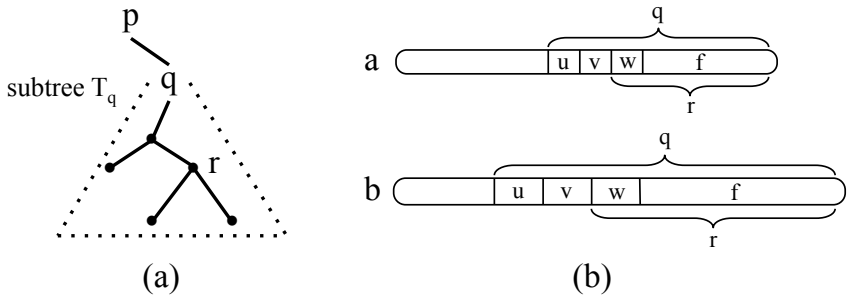

Figure 3.5: Finding attributes judged too long.

Of course, we do not have access to the top-k lists of r and q. So we develop a condition which if true would suggest that the two lists are "roughly the same". Specifically, let $sim_g(a, b) = h(str_g(a), str_g(b))$ be the string similarity function between the string values of two tuples a and b, for config g. Suppose that for all tuple pairs (a, b) in $D = A \times B - C$ we have

$$Condition\ 1 : |sim_q(a, b) - sim_r(a, b)|/sim_q(a, b) \leq \alpha,$$

for a small pre-specified α value, say 0.2. Then we can say that when we switch config from q to r, the score of each tuple pair does not change much, so the top-k list for r will stay roughly the same as that of q.

Checking Condition 1 for all pairs (a, b) in D is not feasible. Hence we perform a theoretical analysis for an idealized scenario (described below). Of course, this idealized scenario rarely happens in practice. But understanding it helps us come up with an efficient heuristic to check Condition 1.

Let $L_f(a)$ be the length (i.e., the total number of words) of attribute f in tuple a, $L_q(a)$ be the sum of the lengths of all attributes in q, for tuple a, and so on. The idealized scenario assumes that (a) attribute f takes the same proportion of the total length of q in both a and b, i.e., $L_f(a)/L_q(a) = L_f(b)/L_q(b) = \beta$, and (b) the remaining length of q is equally distributed among the remaining attributes of q, i.e., $L_k(a) = [(1-\beta)L_q(a)]/(|q|-1)$ for all attribute k in $q - \{f\}$, and the same condition applies to tuple b.

Example 3.3.3. *Consider the two tuples a and b in Figure 3.5.b, where $q = \{u, v, w, f\}$ and $r = \{w, f\}$. We assume that $L_f(a)/L_q(a) = L_f(b)/L_q(b) = \beta$, and $L_u(a)/L_q(a) = L_v(a)/L_q(a)$ and $L_u(b)/L_q(b) = L_v(b)/L_q(b)$.*

Then we can show that (see [90] for a proof sketch):

Theorem 1. *Let $a \in A$ and $b \in B$ be two tuples that satisfy the above assumptions. If*

- (R_1) $sim_q(a, b) \geq [\sqrt{(1+\alpha)^2 + 8} - (1+\alpha)]/4$, *and*
- (R_2) $\beta \geq 1 - \frac{(|q|-1)}{|q \setminus r|} \cdot \frac{\alpha}{(1+\alpha)} \cdot \frac{max\{L_q(a), L_q(b)\}}{L_q(a)+L_q(b)}$,

then pair (a, b) satisfies Condition 1.

Intuitively, this theorem says that if $sim_q(a, b)$ is sufficiently high (Requirement R_1), and attribute f is sufficiently long (Requirement R_2), then $sim_r(a, b)$ will be close to $sim_q(a, b)$. It is not difficult to show that the quantity on the right-hand side of R_1 is upper bounded by 0.5. In practice, we observe that users typically examine only the *top few tens of pairs* in each top-k list (see Section 3.5), and that if these pairs are true matches, their scores often exceed 0.5, making

R_1 true. As a result, if R_2 is also true, then attribute f is long enough to "overwhelm" these pairs. That is, these pairs will change little score-wise when switching from config q to r, thus typically will still show up in the top few tens of pairs of the top-k list for r, an undesirable situation.

To avoid such situations, we will focus on checking R_2. Checking R_2 for many pairs (a, b) is not practical. So we approximate this checking using average lengths, i.e., we (a) replace the quantity β in the left-hand side of R_2 with $min\{AL_f(A)/AL_q(A), AL_f(B)/AL_q(B)\}$, where $AL_f(A)$ for example is the average length of attribute f in Table A, and (b) replace $L_q(a)$ and $L_q(b)$ in the right-hand side of R_2 with $AL_q(A)$ and $AL_q(B)$, respectively.

Procedure FindLongAttr then works as follows. Suppose we have selected config q for expansion (because it does not contain s, the attribute with the least e-score). Then for each attribute f (other than s), we (a) identify $F(T_q)$, the set of configs in T_q that contain f, (b) declare f "too long" if the above approximate checking is true for at least half of the configs $r \in F(T_q)$. It is not difficult to prove that at most one attribute f will be found too long. If so, we do not select q, but select instead the config that does not contain f for expansion. Otherwise, we select q, as usual. This procedure takes less than a second in our experiments.

Discussion: Note that we do not completely remove attributes with many missing values, few unique values, or long values from config generation. Instead, each such attribute f may be removed *only at some point* during the generation process. Configs generated earlier still contain f.

Further, our work here is related to, but very different from work such as [46, 37, 9]. Those works find attributes that are discriminative for classification, often using a labeled sample (as many works in learning do). Here we do not look for discriminative attributes. Instead, we look for attributes such that if two tuples agree on their values, then they are likely to match. For example, suppose all tuples in table A have the same value "US" for "Country", and all tuples in table B have the same value "Canada". Then "Country" is a discriminative attribute because if two tuples disagree on it, they definitely do not match. For our purpose, however, "Country" has little expected benefits, because if two tuples agree on it, it is still not likely that they match (not as much as if they agree on "State" and "City" say).

In fact, the work [143] also treats attributes with missing values and few unique values in a way similar to ours (for blocking and matching). However, it does not handle long attributes, and uses only one config, and thus is significantly outperformed by MatchCatcher (see Section 3.6).

3.4 Top-k String Similarity Joins

So far we have discussed generating a set of configs. We now discuss performing top-k SSJs over these configs (one per config). Previous work has discussed top-k SSJs for a single config [174]. Here we significantly improve that work (and our solution can be applied to top-k SSJ situations beyond this project). We then discuss executing multiple top-k SSJs jointly, by reusing results across the configs, in a parallel fashion.

It is important to note that all SSJ algorithms discussed below (including ours) work with the set-based similarity measures Jaccard, cosine, overlap, and Dice [174]. For ease of exposition, however, we will explain them using the Jaccard measure.

3.4.1 Improving Top-k Join for a Single Configuration

As far as we can tell, the state of the art in top-k SSJs is TopKJoin [174]. Given a set J of strings, TopKJoin finds the k string pairs with the highest similarity scores, for a pre-specified k, in a branch-and-bound fashion. Specifically, it maintains a prefix for each string in J, incrementally extends these prefixes, finds string pairs whose prefixes overlap, computes their similarity scores, use these scores to maintain a top-k list, then extends the prefixes again, and so on.

Example 3.4.1. *Suppose J consists of the four strings w, x, y, z in Figure 3.6.a. We begin by creating a prefix $p(w) =$ "a" for w, then a prefix $p(x) =$ "a" for x. At this point the prefixes of the pair (x, w) overlap. Hence we compute the Jaccard score 0.8 for this pair, then initializes the top-k list K to be containing just this pair. (Here we assume $k = 2$.)*

Next, we create prefix $p(y) =$ "b". This does not produce any new pair whose prefixes overlap. So we continue by creating prefix $p(z) =$ "b". This produces a <u>new</u> pair whose prefixes overlap: (z, y) with score 0.43. Figure 3.6.b shows the updated top-k list K.

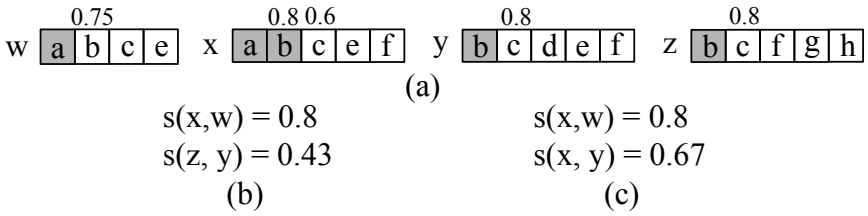

Figure 3.6: An illustration of top-k computation.

Next, we select one prefix to extend (we will discuss shortly how). Suppose we select $p(x)$ *and extend it by one token. Then* $p(x) =$ "ab" *(see Figure 3.6.a). This produces two new pairs whose prefixes overlap:* (x, y) *with score 0.67 and* (x, z) *with score 0.43. Figure 3.6.c shows the updated top-k list* K*. We then select another prefix to extend, and so on. Finding new pairs with overlapping prefix can be done efficiently using an inverted index from token to the prefixes of the strings [174].*

We now discuss how to select a prefix to extend. Suppose we have imposed a global ordering on all tokens, and sorted the tokens in each string w, x, y, z in that order (see Figure 3.6.a). Suppose also that we have created prefixes of size 1, namely $p(w) =$ "a", $p(x) =$ "a", $p(y) =$ "b", $p(z) =$ "b", and are now deciding which prefix to extend. Suppose we select $p(w)$ and extend it by one token, to be "ab". Then it is easy to show that the scores of all *new* pairs generated by this extension are capped by 0.75. Indeed, any *new* pair must involve w. Let such a pair be (w, v). Then the first common token that they share should be "b" (the token just being added to $p(w)$). So they can share at most this token b and the remaining "unseen" tokens of w. Thus $|w \cap v| \leq 3$. Since $|w \cup v| \leq |w| = 4$, it follows that the Jaccard score of (w, v) is capped by 3/4=0.75. We write 0.75 on top of token "b" in w to indicate that when we extend $p(w)$ to include this token, the score of any *new* pair generated by TopKJoin will be *capped* by 0.75. Similarly, we can write 0.8 for the second tokens of x, y, z (see Figure 3.6.a).

We then select the prefix that when extended will include the token with the highest "cap" number (in the hope that it will generate new pairs with the highest possible scores). In this case, we select $p(x)$ (but $p(y)$ and $p(z)$ also work).

We now discuss how to stop. Observe that the "cap" number for "c" in x is 0.6. By the time we have to consider whether to extend $p(x)$ to include "c", the top-k list already has a lower-bound score of 0.67 (see Figure 3.6.c), greater than 0.6. As a result, we do not have to extend $p(x)$ to include "c", and in fact, prefix extension on x can be stopped at this point. TopKJoin terminates when all prefix extensions have stopped, either early (as described above) or because the prefix has covered the entire string. The paper [174] describes TopKJoin in detail, including optimizations to avoid redundant computations.

The QJoin Algorithm: TopKJoin has a major limitation. Every time it generates a new pair (u, v), it immediately computes the similarity score of (u, v) (then updates the top-k list). Computing this score turns out to be very expensive, especially if u and v are long strings. In a sense, it is also "premature", because it can be shown that when we generate (u, v) (as a new pair), we only know that they share a *single* token. There is no evidence yet that they share more tokens and thus are likely to have high similarity score. If they indeed share only one or few tokens, and yet we still compute their score, then that score is likely to be low. So the pair will not make it into the top-k list, yet we have wasted time computing it score.

To address this problem, when generating new pairs, we do not immediately compute their scores. Instead, we keep track of the number of common tokens each pair has, and update this number whenever a prefix is extended. We then compute the score of a pair only if it has q common tokens, and thus is likely to have a high score. It is difficult to select q analytically, so we select it empirically as follows. Assuming at least four CPU cores, we begin by running the modified TopKJoin for $q = 1$, $q = 2$, etc., on all cores, one q value for each core, for $k = 50$. (Note that TopKJoin always does $q = 1$.) Then whichever core finishes first, we keep the process on that core running to produce the rest of the top-k list (effectively selecting the q value associated with that core), and kill the processes on the other cores.

It is straightforward to adapt the above algorithm to work with two tables (instead of just one), and to remove a pair from the top-k list (during the top-k computation) if it happens to be in the candidate set C. Henceforth we refer to this new algorithm as QJoin.

3.4.2 Joint Top-k Joins Across All Configurations

TopKJoin can only be applied to a single config [174]. Our setting however involves multiple related configs. We now describe a solution to find top-k lists *jointly across the configs*. To do so, we use QJoin, but modify it to reuse similarity score computations and top-k lists across the configs, and process the configs in parallel.

Reusing Similarity Score Computations: As discussed in Section 3.4.1, computing the similarity score of a pair (a, b) is very expensive, especially for long strings. Hence, we want to reuse such computations across the configs. To do so, we process the configs in the config tree in a breadth-first order, e.g., processing the root config $f_1f_2f_3$ of the config tree in Figure 3.7 (where the f_i-s are attributes), then the next-level configs, f_2f_3, f_1f_3, f_1f_2, and so on.

When processing a config g (i.e., finding its top-k list), we keep track of certain information, then reuse it when processing configs in the subtree of g. For example, consider again the config tree in Figure 3.7. We start by tokenizing the strings wrt the root config $f_1f_2f_3$ into multisets of word-level tokens. Next, we process the config $f_1f_2f_3$. This process computes the Jaccard score of multiple tuple pairs. When computing the score of such a pair, say (a, b), we compute and store the number of overlapping tokens between any two attributes f_i of a and f_j of b in an in-memory database H. Figure 3.7 illustrates this step. Here, $o(f_1, f_1) = 2$ means attributes f_1 of a and f_1 of b share two tokens. (We only store in H attribute pairs that share tokens.)

Then we can reuse H to drastically speed up processing configs in the subtree rooted at $f_1f_2f_3$. For example, consider processing config f_1f_2. If during this process we need to recompute the score of (a, b) (now with respect to only f_1 and f_2), then we can use H to compute $Overlap_{f_1f_2}(a, b) = o(f_1, f_1) + o(f_1, f_2) + o(f_2, f_1) + o(f_2, f_2)$, then compute the above score as

$$Overlap_{f_1f_2}(a, b)/(L_{f_1f_2}(a) + L_{f_1f_2}(b) - Overlap_{f_1f_2}(a, b)),$$

where $L_{f_1f_2}(a)$ for instance is the length in tokens of the concatenation of f_1 and f_2 for a. Computing the score of (a, b) this way is far faster than computing from scratch.

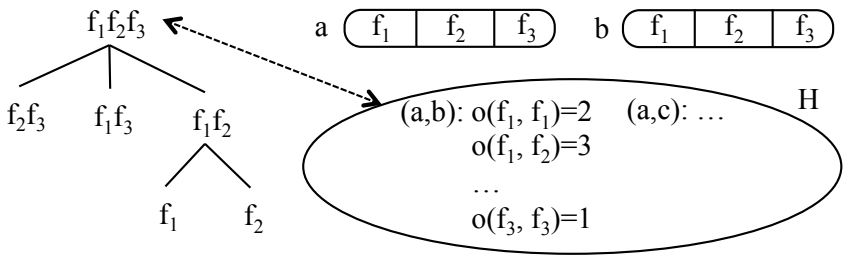

Figure 3.7: Reusing across top-k computations.

Note that while processing config f_1f_2, if we have to compute the score of a new pair (c, d) not yet in H, then we will store similar overlap information for (c, d) in H, to enable reuse when processing configs in the subtree rooted at f_1f_2, and so on.

Reusing Top-k Lists: When applying to a config g, algorithm QJoin starts with an empty top-k list K, then gradually grows K as it iteratively expands the prefixes. In our setting, however, since we process multiple configs, a promising idea is to use the top-k list of a previous config to initialize the top-k list of the current config.

For example, after processing config $g = f_1f_2f_3$ (Figure 3.7), we store its top-k list K_g. Then when processing config $h = f_1f_2$, we use the database H described earlier (which stores overlap information) to re-adjust all scores in K_g. This is necessary because these scores are computed wrt $f_1f_2f_3$, but now we want them to be adjusted to consider only f_1f_2. This re-adjustment is fairly straightforward (and inexpensive) because the overlap information for all pairs in K_g should already be in H. Next, we run the algorithm QJoin as usual to process config $h = f_1f_2$, but using the K_g list with the adjusted scores as the initial top-k list K_h (instead of using an empty list).

Observe that the above procedure enables reusing top-k lists from a parent to a direct child (e.g., from $f_1f_2f_3$ to f_1f_2). Reusing across the "siblings" appears much more difficult. For example, given the top-k list for f_1f_3, there is no obvious way to quickly adjust its scores for f_1f_2, using database H. Hence, currently we do not yet consider such sibling reuse.

Finally, reuse does not come for free. It helps avoid computing certain similarity scores from scratch, but incurs an overhead of storing and looking up the overlap information. If the tuples are short, then the overhead can easily overwhelm the savings. As a result, we trigger reuse only if the average tuple length is at least t tokens (currently set to 20).

Parallel Processing of the Configs: Finally we explore parallel processing on multiple cores. (We consider multicore single machines for now because many data analyst users do not know how to use a machine cluster.) An obvious idea is to process each config across multiple cores. For example, we can split Table A into two halves A_1 and A_2 and Table B into B_1 and B_2, find the top-k list for A_1 and B_1 on the first core, the top-k list for A_1 and B_2 on the second core, etc., then merge the top-k lists. In practice, this approach suffers from severe skew: one core finishes quickly while another runs forever. While it is possible to split the tables intelligently to mitigate skew, this adds considerable overhead and implementation complexity.

As a result, we opted for processing one config per core. Specifically, we traverse the config tree breadth-first, and assign configs to cores in that order (when a core finishes, it gets the next config "in queue"). This solution *continuously utilizes all cores*. But it raises two problems. First, two configs (e.g., $f_1f_2f_3$ and f_1f_2) may concurrently write, or one reads and the other writes, into database H, causing concurrency control issues. To address concurrent writes, observe that only configs with non-empty subtrees (e.g., $f_1f_2f_3$ and f_1f_2 in Figure 3.7) will write. For each such config g, we require it to write into a separate in-memory database H_g.

To address dirty reads (e.g., $f_1f_2f_3$ writes into a database while f_2f_3 reads from it), we note that here each "write" just *inserts* a value; it never modifies or deletes. For such cases there are atomic hashmaps that perform *atomic inserts*, thus avoiding dirty reads. So we implement each database H_g as one such hashmap (using the Atomic Unordered Hashmap in Facebook's C++ Folly package).

Finally, if a parent config, e.g., $g = f_1f_2f_3$, has not yet finished, then a direct-child config, $h = f_1f_2$, cannot reuse g's top-k list. In such situations, we start config h with an initial empty top-k list. When config g finishes, it sends its top-k list to h. Config h merges its current top-k list with the new top-k list from g, to obtain a potentially better top-k list, then continues. The technical report [90] shows the pseudo code of the complete algorithm, and the following theorem shows its correctness (see [90] for a proof sketch):

Theorem 2. *Given two tables A and B, the output C of a blocker on A and B, a set of configs G, a string similarity measure which is Jaccard, Cosine, overlap, or Dice, and a value k, the above*

algorithm returns a set of top-k lists, where each top-k list is the output of applying Algorithm QJoin *to A, B, and C, using a config $g \in G$ and the given similarity measure and k value.*

3.5 Interactive Verification

So far we have discussed processing configs to obtain a set of tuple pairs. We now discuss identifying true matches in this set, via user engagement, rank aggregation, and active/online learning.

Engaging the User: Let E be the union of the top-k lists obtained from processing all configs. Typically E is large (e.g., 3,011-7,089 in our experiments) and the true matches make up just a small portion of E. Thus expecting a user U to be able to examine the entire set E to find true matches is unrealistic.

A reasonable solution is to rank the pairs in E such that the true matches "bubble" to the top, then present the ranked list to U. However, our experiments with a variety of ranking methods (see below) suggest it is very difficult to do so. Typically, the top of the ranked list indeed contains multiple matches. But then the remaining matches tend to be scattered far and wide in the list.

As a result, we decided to engage user U: we rank the pairs in E, present the top-n pairs to U (currently $n = 20$), ask U to identify the true matches, use this feedback to rerank the list, then present the next top-n pairs to U, and so on. As such, we help U iteratively identify true matches, but use this identification to help "bubble" the remaining matches to the top of the ranking.

Using Rank Aggregation: Let m be the number of configs and $L_1, \ldots, L_m$ be the top-k lists obtained from these configs. To engage user U, we first need to aggregate these lists into a single list. Many aggregation methods exist, e.g., [12, 44]. Here we use MedRank [44], a popular method. To use MedRank, we first sort each list L_i in decreasing order of score, then associate each item in the list with a rank, i.e., an integer, such that

L_1	L_2	L_3	L*
a: 1.0 (1)	a: 0.9 (1)	b: 0.8 (1)	a (1)
b: 0.8 (2)	c: 0.7 (2)	a: 0.5 (2)	b (2)
c: 0.8 (2)	d: 0.6 (3)	c: 0.3 (3)	c (2)
d: 0.6 (4)		d: 0.2 (4)	d (4)

Figure 3.8: Combining top-k lists using MedRank.

the higher the score, the lower the rank and items with the same score receive the same rank. Next, we compute for each item a global rank which is the median of its ranks in the lists. Finally, we sort the items in increasing order of global rank, breaking ties randomly, to obtain a list $L*$ which is the aggregation of all top-k lists L_i-s.

Example 3.5.1. *Figure 3.8 shows three top-k lists L_1, L_2, L_3 and the global list $L*$. A line such as "a: 1.0 (1)" under L_1 means that item "a" in list L_1 has score 1.0 and has been assigned rank 1. The ranks for "a" is 1, 1, 2 (see Figure 3.8). So its global rank is 1. The ranks for "b" is 2, 4, 1 (here "b" is missing from L_2, which has ranks 1-3; so we assign to it rank 4). Thus "b"'s global rank is 2. And so on.*

Once we have obtained the global list $L*$, we can present the top-n items of $L*$ to user U. But how do we incorporate the user feedback for the next iteration? A reasonable solution is to use weighted median ranking (WMR): we first assign an equal weight $w_i = 1/m$ to each top-k list L_i ($i \in [1, m]$). At the end of the first iteration, we adjust $w_i = w_i \cdot [1 + log(1 + r_i)]$, where r_i is the number of true matches user U has identified that appear in L_i, then normalize all weights w_i. At the start of the next iteration, we merge the lists $L_1, \ldots, L_m$ again, using WMR to compute the global rank of each item. Next, we present the top-n pairs in this merged list to the user, and so on. Intuitively, the top-k lists in which more true matches appear will become more important, and the weighted global ranking will be "leaning toward" those lists.

Using Learning: WMR does not perform well in our experiments (see Section 3.6). It uses a very limited combination model which fails to fully utilize user feedback. To address this, we explored active learning. Specifically, we iteratively show the next n items of $L*$ to user U, until we have obtained at least one match and one non-match. Suppose we have carried out t iterations, then this produces a set T of nt labeled items. We use T to train a random forest classifier F, use F to find n most informative items in $L*$, show them to the user to label, add the newly labeled items to T, then retrain F, and so on.

Active learning alone however is not quite suited for our purpose. Its goal is to learn a good classifier as soon as possible. Hence it typically shows user U controversial items that it finds

difficult to classify. But many or most of these items can be non-match. User U, however, wants to find many *true matches* as soon as possible (so that U can examine them to quickly understand the problems with the blocker).

The above two goals conflict. To address this problem, we adopt a hybrid solution. After we have obtained the training set T and trained a classifier F, as described above, for the next iteration, we show user U n items where $n/4$ items are the top controversial items chosen by F, as described above. The remaining $3n/4$ items however are those with the *highest positive prediction confidence*, where the confidence is computed as the fraction of decision trees in F that predict the item as a match. Intuitively, the first $n/4$ items are intended to help the active learner, whereas the remaining $3n/4$ items can contain many true matches, and are intended to help the user quickly find many true matches in the first few iterations.

After three such iterations, we stop active learning completely (judging that classifier F has received enough labeled controversial examples in order to do well), but continue the online-learning process with F. Specifically, in each subsequent iteration, we show user U the top n items with the highest positive prediction confidence, produced by F. Once these items have been labeled by U, we add them to the existing training set, retrain F, and so on.

When to Stop? A natural stopping point is when user U finds no new matches in 2 consecutive iterations. Of course, U can stop earlier or continue. If the required blocker recall is very high, U can continue for many iterations. Otherwise, U can stops after the first few iterations (because if these iterations contain many matches, then examining them often already reveals problems with the blocker, which U can then fix).

3.6 Empirical Evaluation

We evaluated MatchCatcher in three ways. First, we asked volunteers to provide blockers for several datasets. These blockers vary in recall, types, and complexity, representing blockers that users may write *at various points* during the blocker development process. We show that MatchCatcher works well with these blockers, thus can effectively support the users in the development process.

Dataset	Tuple type	Table A	Table B	# of matches	# of attrs	Average length
Amazon-Google	software product	1363	3226	1300	5	205, 38
Walmart-Amazon	electronic product	2554	22074	1154	7	76, 179
ACM-DBLP	paper	2294	2616	2224	5	16, 19
Fodors-Zagats	restaurant	533	331	112	7	11, 10
Music_1	song	100000	100000	2978	8	9, 9
Music_2	song	500000	500000	73646	8	9, 9
Papers	paper	455996	628231	unknown	7	17, 18

Table 3.1: Datasets for our experiments.

Second, we performed best-effort blocking on several datasets, by asking volunteers to manually develop the best hash-based blockers, or applying a state-of-the-art solution to learn the best blockers. We show that even in this case MatchCatcher can help uncover problems and improve the blockers.

Finally, we asked real-world users in several data science classes, domain science projects, and at several organizations to use MatchCatcher. We show that MatchCatcher has proven highly effective in helping these users develop blockers.

3.6.1 Supporting Users in Developing Blockers

For this experiment we need "gold" matches, so we use the six datasets shown in the first six rows of Table 3.1. As far as we can tell, these datasets are the largest ones used in previous EM work for which "gold" matches are available. Here we created two versions of the Music dataset, Music1 and Music2, to ensure a diversity of size (from 331 to 100K to 500K of tuples per table). [90] describes these datasets in more details.

For each dataset we asked volunteers to create multiple blockers (see Table 3.2). They are of the types described in Section 3.2: overlap (OL), hash (HASH), similarity-based (SIM), and rule-based (R). For example, the first row of Table 3.2 describes 4 blockers for dataset A-G. These include a hash blocker on attribute "manufacturer" and a rule-based blocker that combines two SIM blockers. See [90] for more details on these blockers. (The next subsection describes experiments with the best hash blockers manually created for these datasets.)

Dataset	Blocker Q
A-G	**(OL)** title_overlap_word<3 **(HASH)** attr_equal_manuf **(SIM)** title_cos_word<0.4 **(R)** title_jac_word<0.2 AND manuf_jac_3gram<0.4
W-A	**(OL)** title_overlap_word<3 **(HASH)** attr_equal_brand **(SIM)** title_cos_word<0.4 **(R)** price_absdiff>20 OR title_jac_word<0.5
A-D	**(OL)** authors_overlap_word<2 **(SIM)** title_jac_3gram<0.7 **(R_1)** title_cos_word<0.8 AND authors_jac_3gram<0.8 **(R_2)** year_abs_diff>0.5 OR title_jac_word<0.7
F-Z	**(OL)** name_overlap_word<2 **(HASH)** attr_equal_city **(SIM)** addr_jac_3gram<0.3 **(R)** (name_cos_word<0.5 AND type_jac_3gram<0.7) OR addr_jac_3gram<0.3
M_1	**(OL)** artist_name_overlap_word<2 **(HASH)** attr_equal_artist_name **(SIM)** title_cos_word<0.5 **(R)** year_absdiff>0.5 OR title_cos_word<0.7
M_2	**($HASH_1$)** attr_equal_artist_name **($HASH_2$)** attr_equal_release_OR_attr_equal_artist_name **(SIM_1)** title_cos_word<0.6 **(SIM_2)** title_cos_word<0.7 **(SIM_3)** title_cos_word<0.8

Table 3.2: Blockers for the first set of experiments.

Developing a blocker is typically *a long process* in which users often start with a simple blocker, then gradually revise it into a more complex one with higher recall. The above blockers differ in type, recall, and complexity, representing blockers that users may write *at various points* during the above process. We now show that MatchCatcher can help debug these blockers, suggesting that it can support the user during the entire development process.

Overall Accuracy: First we examine the top-k SSJs module. The first two columns of Table 3.3 list datasets and blockers. Column C lists the size of C, the output of the blocker on Tables A and B. Column M_D lists the number of true matches in $D = A \times B - C$. This number varies drastically, e.g., 137-1,267 for A-G, 87-566 for W-A, etc., suggesting that blocker recall often varies widely and that it is important to debug to improve recall.

Column E lists the size of E, the union of all top-k lists over the configs (for $k = 1000$). Column M_E lists the number of true matches in E (the numbers outside parentheses), and shows that set E contains a substantial fraction of true matches in D, e.g., 54-65% for A-G, 41-83% for W-A, 96-100% for A-D, etc. (see the numbers in parentheses). This suggests that the top-k module can effectively find the true matches in D.

Next we examine the Match Verifier. We want to know its accuracy if run until its natural stopping point (see Section 3.5). It is difficult to recruit enough real users for this large-scale

	Q	C	M_D	E	M_E	F	I
A-G	OL	8,388	291	4,063	190 (65.3)	166 (87.4)	40
	HASH	1,835	1,267	3,337	820 (64.7)	803 (97.9)	97
	SIM	7,406	192	4,341	104 (54.2)	73 (70.2)	29
	R	27,650	137	4,362	76 (55.5)	65 (85.5)	24
W-A	OL	210,782	87	6,570	48 (55.2)	37 (77.1)	7
	HASH	256,341	201	5,089	168 (83.6)	147 (87.5)	26
	SIM	46,900	135	7,089	56 (41.5)	46 (82.1)	7
	R	4,265	566	5,027	256 (45.2)	233 (91.0)	33
A-D	OL	56,869	41	4,270	41 (100.0)	37 (90.2)	8
	SIM	2,487	61	3,335	59 (96.7)	56 (94.9)	11
	R_1	3,764	41	3,843	41 (100.0)	38 (92.7)	10
	R_2	2,173	107	3,011	104 (97.2)	101 (97.1)	16
F-Z	OL	115	47	5,079	46 (97.9)	46 (100.0)	5
	HASH	10,165	52	4,653	51 (98.1)	51 (100.0)	5
	SIM	2,146	13	5,908	12 (92.3)	12 (100.0)	5
	R	124	33	5,239	32 (97.0)	32 (100.0)	5
M_1	OL	253,286	778	5,045	673 (86.5)	671 (99.7)	38
	HASH	212,296	188	4,948	100 (53.2)	100 (100.0)	13
	SIM	2,601,349	78	5,050	38 (48.7)	36 (94.7)	7
	R	89,344	202	5,213	113 (55.9)	109 (96.5)	11
M_2	$HASH_1$	11,115,136	4,530	5,428	661 (14.6)	648 (98.0)	47
	$HASH_2$	14,632,318	3,844	5,735	450 (11.7)	432 (96.0)	35
	SIM_1	27,461,378	2,220	5,420	1,012 (45.6)	1,012 (100.0)	54
	SIM_2	14,924,148	3,238	5,533	1,087 (33.6)	1,087 (100.0)	58
	SIM_3	8,512,446	4,228	5,587	1,151 (27.2)	1,151 (100.0)	61

Table 3.3: Accuracy in retrieving the killed-off matches.

experiment involving 25 blockers. So we use synthetic users, whom we assume can identify the true matches accurately (we describe multiple experiments with real users below).

Column F of Table 3.3 show that this module can retrieve a large number of matches in E, e.g., 65-803 for A-G (see the numbers outside parentheses), and that the retrieval rate is very high, e.g., 70-98% for A-G, 77-91% for W-A, etc. (see the numbers inside parentheses). Finally, Column I shows that the total number of iterations is 5-13 in 12 cases, 16-40 in 8 cases, 47-61 in 4 cases, and 97 in 1 case. The higher number of iterations is often due to the larger number of matches that have to be retrieved from E, e.g., for blocker HASH of dataset A-G, the module needed 97 iterations to retrieve 803 matches, a reasonable number of iterations given that each iteration shows only 20 tuple pairs to the user.

Thus, if the user runs the Match Verifier until its natural stopping point, he/she can retrieve a large number of matches. This is good news for applications in which blocker recall is critical, thus the user may want to examine *all* matches that the module can retrieve.

Accuracy & Explanations for the First Few Iterations: To examine if users can quickly find many matches and explanations, we asked volunteers to *manually work with the Match Verifier for the first three iterations*. Table 3.4 shows the results (for space reasons we only list five blockers for five datasets, the results for other blockers are similar). The table shows that the user needed only 7-10 mins to examine the first three iterations, was able to identify a large number of matches (28-43), and was able to identify multiple reasons for why they are killed off (a reason such as "large threshold (18)" means that tuple pair #18 was killed off due to the blocker using a large threshold, and this was the first pair where the user observed this problem). Overall, the results suggest that after examining the first few iterations, the user can already identify multiple problems with the blocker (which he/she can then fix).

3.6.2 Debugging State-of-the-Art Blockers

Suppose a user has manually developed a good standard blocker, or has used state-of-the-art techniques to learn a blocker, we want to know if MatchCatcher can still help improve the blocker's accuracy. Toward this goal, we performed two experiments.

Hash Blockers: First, we asked a user well-trained in EM to develop the best possible hash blockers for five datasets (the first five in Table 3.1). For example, for dataset A-G, this user created the blocker Q_1 which keeps a pair of tuples if they agree on "manufacturer" *or* on a hash of "price" *or* on a hash of "title". Thus, Q_1 combines three hash blockers. ([90] describes all five blockers in details.) We selected hash blocking because it is well-known, easy to understand, and fast. Hence it is considered a standard blocking method commonly used in practice. On the five datasets A-G, W-A, A-D, F-Z, and Music1, the best hash blockers achieve 75.6, 95.1, 100, 97.3, and 100% recall, respectively.

We then asked the same user to use MatchCatcher to try improving the above hash blockers. For A-D and Music1, which already have 100% recall, using MatchCatcher the user did not find any killed-off matches (as expected), so debugging terminated early. For A-G, W-A, and F-Z, however, debugging significantly improved recall from 75.6 to 99.7, 95.1 to 99.6, and 97.3 to 100%, respectively. [90] describes one such debugging scenario in details.

Blocker	# iteration	Label time	Blocker problems
OL (A-G)	3 iterations, 31 matches	8 mins	large threshold (18); attribute "manuf" is sprinkled in the attribute "title" (18)
HASH (W-A)	3 iterations, 43 matches	10 mins	different words for the same brand (6); missing values in attribute "brand" (13)
SIM (A-D)	3 iterations, 28 matches	7 mins	large threshold (16); attribute "title" contains subtitle in one table (22)
R (F-Z)	3 iterations, 32 matches	7 mins	different descriptions for attribute "type" (11); unnormalized attribute "address" (33); attribute "city" is sprinkled in "name" (47)
R (M_1)	3 iterations, 41 matches	10 mins	input tables are not lower-cased (5); missing values in attribute "year"

Table 3.4: Accuracy in the first 3 iterations and explanations.

Learned Blockers: From a group of researchers we obtained Papers, the dataset described in the last row of Table 3.1. For this dataset, they have applied the method in [30] to learn blockers using a sample labeled by crowdsourcing, and we were able to obtain three such blockers (learned on three separate samples). [90] describes these blockers, which are the best blockers that the learning method has found in a very large space of blockers, including hash ones. Unfortunately, we do not have the entire set of "gold" matches for Papers (we do have some "gold" matches, but not all of them). Hence, we are unable to report recalls for these blockers.

We then asked a user to apply MatchCatcher to these blockers. After 5 iterations, the user found 76, 61, and 65 matches for the three blockers, respectively. More importantly, the user was able to identify a set of reasons for why these matches were killed off and suggestions for improving the blockers (see [90]). Given the lack of "gold" matches, we were not able to improve the blockers then compare their recalls. Nevertheless, the above experiments suggest that blockers learned using state-of-the-art solutions can still have many problems and MatchCatcher can help pinpoint these, to help the user improve recall.

3.6.3 MatchCatcher "in the Wild"

Over the past two years variations of MatchCatcher have been used by 300+ students in 4 data science classes and 7 EM teams at 6 organizations. The feedback has been overwhelmingly positive. For example, 18 teams used MatchCatcher in a class project, and reported that it helped

(a) discovering data that should be cleaned, (b) finding the correct blocker types and attributes, (c) tuning blocker parameters, and (d) knowing when to stop. We have reported on some of this experience in [79]. Overall, we found that many real-world users have used MatchCatcher as *an integral part of an end-to-end blocker development process:* start with a simple blocker, use MatchCatcher to identify problems, improve the blocker, and so on, until MatchCatcher no longer reports substantial problems with the blocker.

3.6.4 Runtime & Scalability

MatchCatcher was implemented in Cython, and all experiments used a RedHat 7.2 Linux machine with Intel E5-1650 CPU. The top-k module took 6.6-9.4 secs (for dataset A-G), 97-310 (W-A), 2.8-3.2 (A-D), 0.2 (F-Z), 12.1-24.4 (M_1), 57-230 (M_2), and 65-344 (Papers), respectively. For the first five datasets, these times are quite small except 97-310 secs for W-A. On W-A, the k-th pair on the top-k list (recall that $k = 1000$) often has a very low score, e.g., 0.21-0.225. Thus the top-k module took more time. The last two datasets (M_2 and Papers) are much larger (500K tuples per table), and so took longer to run. In all cases, however, the total time is still under 5.8 minutes.

To examine how the top-k module scales, we measure its time as we vary the size of the two largest datasets, M_2 and Papers, at various percentages of the original datasets (which have 500-600K tuples per table). Figure 3.9 shows the results for the first three blockers in Table 3.3 for M_2 and all three blockers for Papers, for $k = 100$ (the left two plots) and $k = 1000$. The results show that the top-k module scales linearly or sublinearly as the table size grows. Finally, on all datasets the Match Verifier took under 0.1 sec to aggregate the top-k lists, and 0.14-0.18 secs to process user feedback in each iteration.

3.6.5 Additional Experiments

The technical report [90] describe extensive experiments on the performance of the MatchCatcher components, sensitivity analysis, and comparison with a recent related work. We briefly summarize those experiments here.

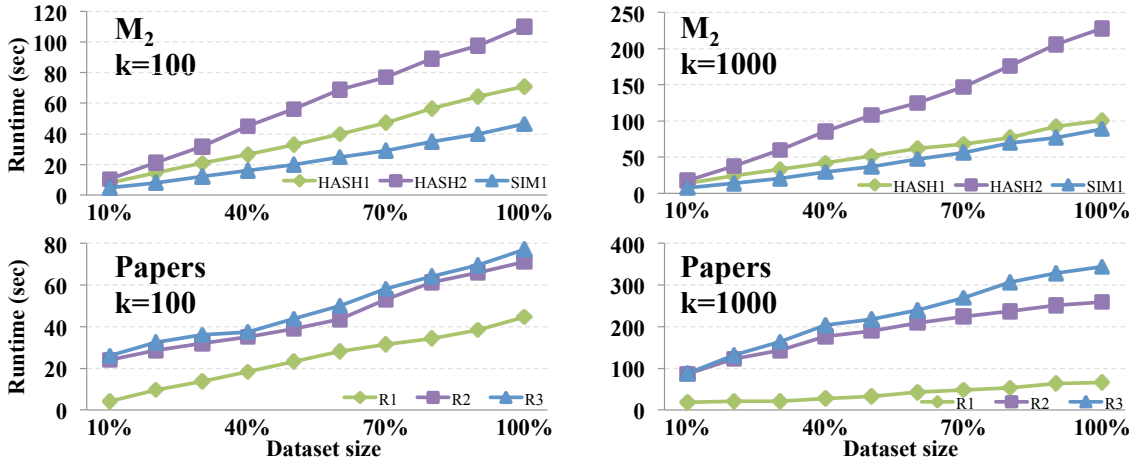

Figure 3.9: Runtime of top-k module for varying table sizes.

Performance of the Components: We show that using multiple configs instead of just one config significantly increases the number of retrieved matches, by 10-74%. Handling long attributes increases the recall of E (the fraction of matches in D that are in E) by up to 11%, compared to not handling them in config generation. Our experiments also show that the joint top-k processing strategy over multiple configs significantly outperforms the baseline of executing each config individually, by as much as 3.5 times. Finally, we found that active/online learning significantly outperforms weighted median ranking in the Match Verifier.

Sensitivity Analysis: We found that increasing k (the number of pairs retrieved per config) does increase the number of true matches retrieved, but only up to a certain k, and comes at the cost of higher runtime, and that using 3 active learning iterations (as we currently do) provides a good balance between increasing the classifier accuracy and increasing recall in the Match Verifier.

Comparison with Recent Work: We found MatchCatcher significantly outperforms the work in [143], which uses a single config, e.g., improving the recall of E by 26-47% on the A-G dataset.

3.7 Additional Related Work

We have discussed related work throughout this chapter. We now discuss additional related work. As far as we can tell, [79] is the first to raise the need for debugging for blocking. But

it does not discuss any debugging solution in depth, as we do here. Other related works include debugging for data cleaning [50], schema mapping [16], and data errors in spreadsheets [6]. They do not address EM and their solutions do not apply to our context. But they do underscore the importance of debugging for data integration and cleaning.

SSJs have received much attention, e.g., [71, 175] (see [177] for a survey). Top-k SSJs are studied in [174, 178]. [178] proposes a B+ tree based index to scale top-k SSJs on edit distance. It does not work well for datasets with large textual difference [177], however, a common occurrence in our case. The work [174], which uses prefix filtering to find the top-k pairs, is better suited to our case. But it does not handle long strings well [177]. Here we have significantly improved this work and extended it to work over multiple configs.

Rank aggregation has been studied extensively in the database/IR communities, e.g., [12, 44, 39]. Active learning (AL) for EM has been studied in [105, 130, 55]. But they perform extensive AL to learn an accurate matcher. In contrast, we use only a few AL iterations to learn a classifier with reasonable accuracy, then use it to surface matches for debugging purposes. The above work also does not combine AL with online learning as we do. Finally, the work [155] uses a learning-based UI model similar to ours, but for IR tasks.

Chapter 4

Deep Learning for Entity Matching: A Design Space Exploration

4.1 Introduction

Entity matching (EM) finds data instances referring to the same real-world entity, such as (Eric Smith, Johns Hopkins) and (E. Smith, JHU). This problem is critical in data cleaning and integration. As a result, it has received significant attention [18]. Tremendous progress has been made. But no satisfactory solution has yet been found.

In the past few years, deep learning (DL) has become a major direction in machine learning [88, 132, 57, 163]. DL yields state-of-the-art results for tasks over data with some hidden structure, e.g., text, image, and speech. On such data, using labeled examples, DL can automatically construct important features, thereby obviating the need for manual feature engineering. This has transformed fields such as image and speech processing, medical diagnosis, autonomous driving, robotics, NLP, and many others [88, 57]. Recently, DL has also gained the attention of the database research community [163, 35].

A natural question then is whether deep learning can help entity matching. Specifically, has DL been applied to EM and other related matching tasks? If so, what are those tasks, and what kinds of solutions have been proposed? How do we categorize those solutions? How would those DL solutions compare to existing (non-DL) EM solutions? On what kinds of EM problems would they help? And on what kinds of problems would they not? What are the opportunities and challenges in applying DL to EM? As far as we know, no published work has studied these questions in depth.

In this project we study the above questions, with the goal of understanding the benefits and limitations of DL when applied to EM problems. Clearly, DL and EM can be studied in many

different settings. In this project, as a first step, we consider the classic setting in which we can automatically train DL and EM solutions on *labeled training data*, then apply them to test data. This setting excludes unsupervised EM approaches such as clustering, and approaches that require substantial human effort such as crowdsourced EM or EM using hand-crafted rules.

Defining a Space of DL Solutions: We begin by defining a space of DL solutions for EM and related matching tasks. As far as we can tell, only one work has proposed a DL solution called DeepER for EM [40]. But numerous DL solutions have been proposed for related matching tasks in the field of natural language processing (NLP), such as entity linking, coreference resolution, textual entailment, etc. [88]. We provide a categorization of these solutions that factors out their commonalities. Building on this categorization, we describe a DL architecture template for EM, and discuss the trade-offs of the design choices in this template. We select four DL solutions as "representative points" in the design space (formed by the combination of the choices). These solutions include relatively simple models such as DeepER, the existing DL solution for EM [40], as well as DL solutions with significantly more representational power. We refer to these four DL solutions as SIF, RNN, Attention, and Hybrid (see Section 4.4).

Defining a Space of EM Problems: Next, we investigate for which types of EM problems DL can be helpful. The most popular type of EM problems has been matching *structured data instances*, e.g., matching tuples where the attribute values are short and atomic, such as name, title, age, city, etc. (see Figure 4.1.a) [18]. Thus, we examine how DL performs on EM setups over such structured data.

In recent years, however, we have also seen an increasing demand for matching *textual data instances*, such as matching descriptions of products that correspond to long spans of text, matching company homepages with Wikipedia pages that describe companies, matching organization descriptions in financial SEC filings, and matching short blurbs describing Twitter users, among others (see Figure 4.1.b). We suspect that traditional learning-based EM solutions (e.g., those that use random forest, SVM, etc.) may have difficulties matching textual instances, because there are few meaningful features that we can create (e.g., computing word-level Jaccard or TF/IDF scores).

On the other hand, we believe that DL can perform well here, due to its ability to learn from raw text, and its current successes in NLP task [23, 156, 179].

Understanding the advantages that DL has to offer for EM over textual data instances raises an intriguing possibility. In our extensive work on EM with many companies, we have seen many cases where the instances to be matched are *structured but dirty*. Specifically, the value for a particular attribute (e.g., brand) is missing from the cell for that attribute, but appears in the cell for another attribute (e.g., name), see for example tuple t_1 in Figure 4.1.c. This commonly arises due to inaccurate extraction of the attributes (e.g., extracting "leather red" as a value for attribute color, even though "leather" is a value for attribute materials). Traditional EM solutions do not work well for such cases. We suspect, however, that DL can be a promising solution to such EM problems, because it can simply ignore the "attribute boundaries" and treat the whole instance as a piece of text, thus in a sense going back to the case of matching textual instances.

Empirical Evaluation: To evaluate the above hypotheses, we assemble a collection of datasets, which includes all publicly available datasets (with labeled data) for EM that we know of, as well as several large datasets from companies. We create 11 EM tasks for structured instances, 6 tasks for textual instances, and 6 tasks for dirty instances, with the number of labeled instances ranging from 450 to 250K. We compare the four DL solutions described earlier (SIF, RNN, Attention, and Hybrid) with Magellan, a state-of-the-art open-source learning-based EM solution [79].

Our results show that DL solutions are competitive with Magellan on structured instances (87.9% vs 88.8% average F_1), but require far longer training time (5.4h vs 1.5m on average). Thus, it is not clear to what extent DL can help structured EM (compared to just using today learning-based EM solutions). On the other hand, DL significantly outperforms Magellan on textual EM, improving accuracy by 3.0-22.0% F_1. Our results also show that DL significantly outperforms Magellan on dirty EM, improving accuracy by 6.2-32.6% F_1. Thus, DL proves highly promising for textual and dirty EM, as it provides new automatic solutions that significantly outperform current best automatic solutions.

In addition to examining the accuracy of the various DL solutions (compared to current learning-based EM solutions) as we vary the type of EM tasks, we also examine how the different design

choices in the space of DL solutions lead to various trade-offs between the accuracy and efficiency of these solutions. We further perform a detailed experimental validation of all identified trade-offs.

Finally, we analyze why DL solutions work better than current EM solutions, and why they do not yet reach 100% accuracy. We discuss the challenges of applying DL to EM (e.g., the effect of domain-specific semantics and training data on the performance of DL solutions, the need for automating the exploration of the accuracy and scalability trade-off for DL solutions for EM), as well as future research directions.

Contributions: To summarize, in this project we make the following contributions:

- We provide a categorization of DL solutions for numerous matching tasks, and define a design space for these solutions, as embodied by four DL solutions SIF, RNN, Attention, and Hybrid. To our knowledge, this is the first work that defines a design space of DL solutions of varying complexity for learning distributed representations that capture the similarity between data instances.

- We provide a categorization of EM problems into structured EM, textual EM, and dirty EM. Structured EM has been studied extensively, but to our knowledge textual EM and dirty EM, while pervasive, have received very little or no attention in the database research community.

- We provide an extensive empirical evaluation that shows that DL does not outperform current EM solutions on structured EM, but it can significantly outperform them on textual and dirty EM. For practitioners, this suggests that they should consider using DL for textual and dirty EM problems.

- We provide an analysis of DL's performance and a discussion of opportunities for future research.

This project is conducted as a part of the larger Magellan project at UW-Madison [80, 79], which builds Magellan, a novel kind of EM system. Magellan provides support for the entire EM pipeline, and is built as a set of interoperable packages in the Python data science ecosystem (rather than as a single monolithic stand-alone system, as is commonly done today).

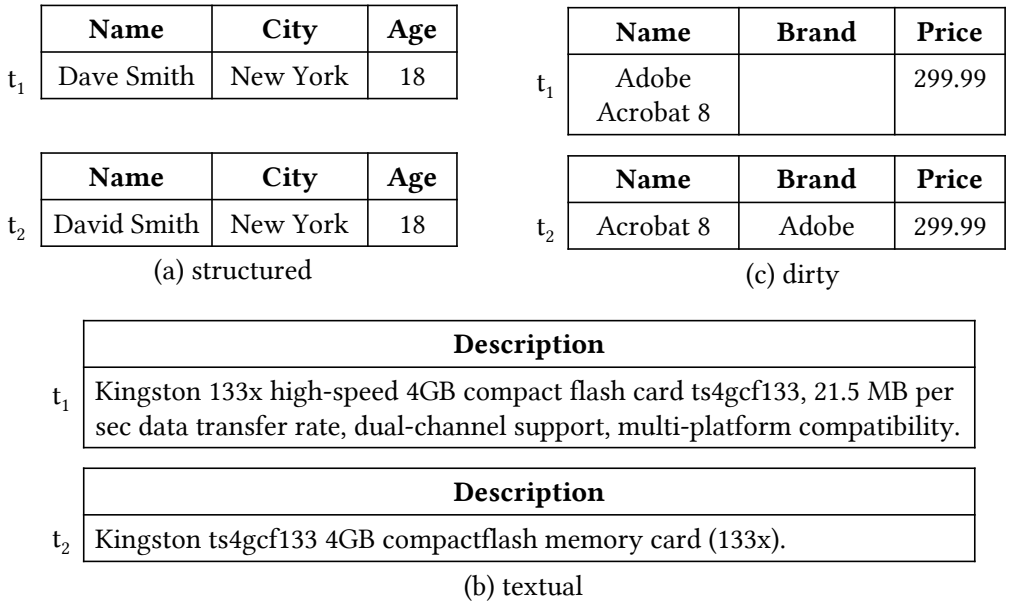

Figure 4.1: Tuple pair examples for the three EM problem types considered in this project.

Magellan has been successfully applied to a range of EM tasks in domain sciences and at companies, and used in many data science classes [80]. We have open-sourced the four DL solutions described here as the deepmatcher Python package, as a part of the open-source code for Magellan.

4.2 Preliminaries and Related Work

4.2.1 Entity Matching

Problem Setting: In Section 2.1 we describe the EM workflow which consists of two major steps: blocking and matching. In this project we focus on the matching step of EM. We assume as input two collections D and D' and a candidate set C containing entity mention pairs $(e_1 \in D, e_2 \in D')$. We further assume access to a set T of tuples $\{(e_1^i, e_2^i, l)\}_{i=1}^{|T|}$ where $\{(e_1^i, e_2^i)\}_{i=1}^{|T|} \subseteq C$, and l is a label taking values in {"match", "no-match"}.

Given the labeled data T our goal is to design a matcher M that can accurately distinguish between "match" and "no-match" pairs (e_1, e_2) in C. We focus on machine learning (ML) based entity matching, because they obtain state-of-the-art results in benchmark EM datasets [53, 81, 79]

and obviate the need for manually designing accurate matching functions [9, 79]. Specifically, we use T as labeled training data to learn a matcher M that classifies pairs of entity mentions in C as "match" or "no-match".

Types of EM Problems: Recall that we want to know for which types of EM problems DL can be helpful. Toward this goal we consider the following three types:

(1) Structured EM: Entity mentions in D and D' are structured records that follow the same schema with attributes $A_1, \ldots, A_N$. We classify a dataset as structured when its entries are relatively clean, i.e., attribute values are properly aligned and cells of each record contain information that is associated only with the attribute describing each cell (see Figure 4.1.a). Further, the data may contain text-based attributes but of restricted length (e.g., product title, address).

(2) Textual EM: All attributes for entity mentions in D and D' correspond to raw text entries (see Figure 4.1.b).

(3) Dirty EM: Entity mentions in D and D' are structured records with the same schema $A_1, \ldots, A_N$. However, attribute values may be "injected" under the wrong attribute (i.e., attribute values are not associated with their appropriate attribute in the schema), see Figure 4.1.c.

For the above three EM problem types, we will experimentally compare state-of-the-art learning-based entity EM solutions that use traditional classifiers (e.g., logistic regression, SVM, and decision trees [79]) with classifiers that use state-of-the-art DL models (described in Section 4.4).

Related EM Work: EM has received much attention [18, 43, 109, 134] (see Chapter 2 for more details). Most EM approaches in the database literature match structured records [53], whereas most related works in NLP and similarity learning match entity mentions that are text spans. We review prior work on NLP tasks related to EM in Section 4.2.2.

In terms of DL for EM, DeepER, a recent pioneering work [40], focuses on designing DL solutions to EM. That work proposes two DL models that build upon neural network (NN) architectures used extensively in the NLP literature. Those two models correspond to instances of the design space introduced in this project and are similar to the two DL models described in Sections 4.4.1 and 4.4.2. Our experimental analysis (see Section 4.5) shows that more complex models

tend to outperform these simpler models. In addition, DeepER [40] discusses blocking and how distributed representations can be used to design efficient blocking mechanisms. Blocking is out of the scope of our project.

Both DeepER and our work here formulate EM as a pairwise binary classification task using logistic loss. But there are other ways to formulate the same problem. For example, triplet learning [67] first learns good embeddings for objects using triplet loss, and then learns an NN classifier. The work in [110] attacks the problem by learning good entity embeddings in vector space using contrastive loss. A distance threshold or an NN classifier is used for classification. Yet another potential approach (used in the Question-Answering (QA) domain) poses matching as a nearest neighbor search problem [176] and can be adapted for EM. Finally, Matching Networks [157], an approach to perform image classification using labels of related images, may also be adapted for EM.

4.2.2 DL Solutions for Matching Tasks in NLP

We now briefly review DL solutions for matching related tasks in NLP (e.g., entity linking, coreference resolution, etc.), then provide a categorization that factors out their commonalities.

Entity Linking: Entity linking aims at linking entity mentions in an input document (usually a small piece of text) to a canonical entity in a knowledge base [137]. For example, given the text span "Apple announced the new generation iPhone" from a document and access to DBpedia, one needs to link entity mention "Apple" to entity "Apple Inc." in DBpedia. The key difference between entity linking and EM is that in entity linking the target knowledge base contains additional information such as the relationships between entities. Most entity linking solutions use this information to collectively reason about multiple entities during linking [53]. DL approaches to entity linking are no exception. For example, recent DL work [150, 48] proposed the use of hierarchical word embeddings to obtain representations that capture entity cooccurrences, Ganea et al. [51] extended attention mechanisms to consider not only the input text span but also surrounding context windows, and Huang et al. [69] rely on the knowledge base structure and develop a deep, semantic entity similarity model using feed-forward NNs.

Coreference Resolution: Coreference resolution takes as input a document (or collection of documents) and aims to identify and group text spans that refer to the same real-world entity [72]. For example, both "N.Y." and "Big Apple" refer to "New York" in an article introducing the city. While related to EM, coreference resolution is significantly different as it operates on entity mentions that correspond to (typically short) text spans that commonly appear in the same document, and thus share similar context. As with most NLP tasks, recent work has proposed DL solutions to coreference resolution. For example, Clark et al. [21] used word embeddings to encode and rank pairs of entity mentions and use a deep neural network to identify clusters of entity mentions that correspond to the same entity, and Wiseman et al. [170] proposed using an RNN to obtain a global representation of entity clusters.

Textual Entailment & Semantic Text Similarity: Textual entailment [28] determines when the meaning of a text excerpt is contained in the meaning of a second piece of text, i.e., if the two meanings are semantically independent, contradictory or in an entailment relationship where one sentence (called the premise) can induce the meaning of the other one (called the hypothesis). For example, the sentence "a cat is chasing a mouse" entails another sentence "a cat is moving", contradicts with "a cat is sleeping", while is neutral with "Tom saw a cat with yellow eyes". A similar task is semantic text similarity, which decides if two given text snippets are semantically similar. Many DL solutions have been proposed for these problems. For example, recent work [96, 15] proposed using Bi-LSTMs—a state-of-the-art RNN—with attention to learn representation vectors for sentence pairs. A different line of work [116, 136] suggested that using only attention mechanisms with simple feed-forward NNs suffice to summarize pairs of sentences, thus avoiding learning complicated representations such as those obtained by RNNs. Neculoiu et al. [110] proposes using Bi-LSTMs with a siamese architecture trained using a contrastive loss function. This is so that matching sentences would be nearby in vector space. More recently, Nicosia et al. [111] builds upon this architecture and proposes training the network by also jointly minimizing a logistic loss apart from the contrastive loss to improve classification accuracy.

Question Answering: In question answering (QA) [61], the task is to answer natural language questions, using either existing text passages or a given knowledge base. Here DL solutions include [154, 56, 176]. Golub et al. [56] build upon RNNs to learn representations that summarize questions and entities. To deal with rare words, [176] adopt a character-level NN to generate the word embeddings.

Categorization of the DL Solutions: While DL models (i.e., solutions) for NLP seems highly specialized at first glance, they do in fact share several commonalities. All models that we have discussed so far take as input a pair of sequences, learn a vectorized representation of the input sequence pair, then perform a comparison between the two sequences. All of these models can be classified along three dimensions: (1) the *language representation* used to encode the input sequences (e.g., use a pre-trained word or character embedding or learn one from scratch), (2) the kind of network used to *summarize* the input, i.e., learn a vector representation of the input sequence pair (e.g., use an RNN or an attention-only mechanism or a combination of the two), and (3) the method used to *compare* the two input sequences (e.g., a neural network).

4.3 A Design Space of DL Solutions

Building on the above categorization of the DL solutions for matching tasks in NLP, we now describe an architecture template for DL solutions for EM. This template consists of three main modules, and for each module we provide a set of choices. The combinations of these choices form a design space of possible DL solutions for EM. The next section selects four DL solutions for EM (SIF, RNN, Attention, and Hybrid) as "representative points" in this design space. Section 4.5 then evaluates these four DL solutions, as well as the trade-offs introduced by the different design choices.

4.3.1 Architecture Template & Design Space

Figure 4.2 shows our architecture template for DL solutions for EM. This template is for the matching phase of EM only (the focus of this project). It uses the categorization of DL models

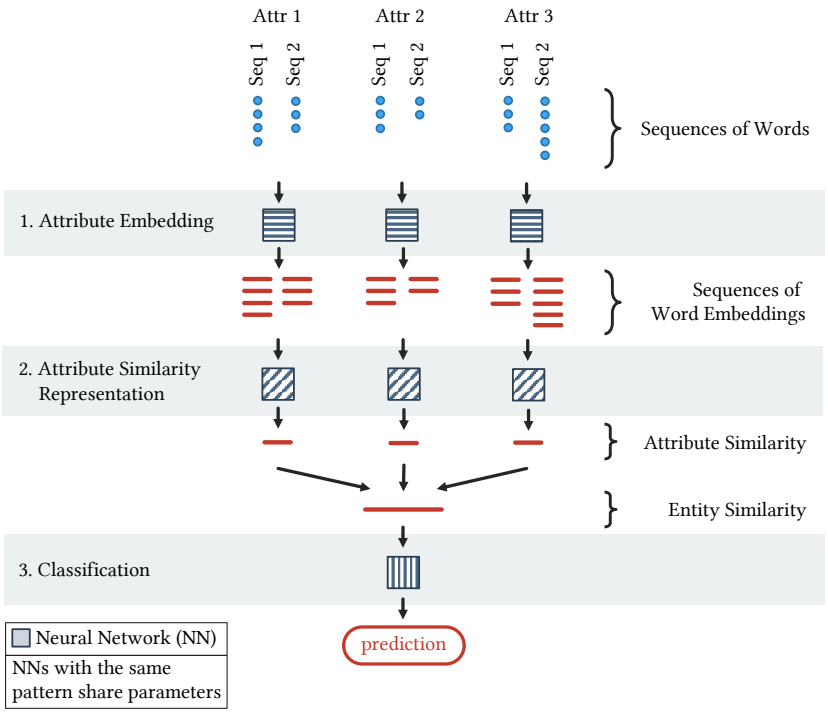

Figure 4.2: Our architecture template for DL solutions for EM.

for matching related tasks discussed in Section 4.2.2, and is built around the same categorization dimensions: (1) the language representation, (2) the summarization technique, and (3) the comparison method used to analyze an input pair of sequences. The template consists of three main modules each of which is associated with one of these dimensions.

Before discussing the modules, we discuss assumptions regarding the input. We assume that each input point corresponds to a pair of entity mentions (e_1, e_2), which follow the same schema with attributes $A_1, \ldots, A_N$. Textual data can be represented using a schema with a single attribute. We further assume that the value of attribute A_j for each entity mention e corresponds to a sequence of words $\mathbf{w}_{e,j}$. We allow the length of these sequences to be different across different entity mentions. Given this setup, each input point corresponds to a vector of N entries (one for each attribute $A_j \in \{A_1, \ldots, A_N\}$) where each entry j corresponds to a pair of word sequences $\mathbf{w}_{e_1,j}$ and $\mathbf{w}_{e_2,j}$.

The Attribute Embedding Module: For all attributes $A_j \in A_1 \cdots A_N$, this module takes sequences of words $\mathbf{w}_{e_1,j}$ and $\mathbf{w}_{e_2,j}$ and converts them to two sequences of word embedding vectors $\mathbf{u}_{e_1,j}$ and $\mathbf{u}_{e_2,j}$ whose elements correspond to d-dimensional embeddings of the corresponding words. More precisely, if for e_1, word sequence $\mathbf{w}_{e_1,j}$ contains m elements then we have $\mathbf{u}_{e_1,j} \in \mathbb{R}^{d \times m}$. The same holds for e_2. The overall output of the attribute embedding module of our template is a pair of embeddings $\mathbf{u}_{e_1,j}$ and $\mathbf{u}_{e_2,j}$ for the values of attribute A_j for entity mentions e_1 and e_2. We denote the final output of this module as $\{(\mathbf{u}_{e_1,j}, \mathbf{u}_{e_2,j})\}_{j=1}^{N}$.

The Attribute Similarity Representation Module: The goal of this module is to automatically learn a representation that captures the similarity of two entity mentions given as input. This module takes as input the attribute value embeddings $\{(\mathbf{u}_{e_1,j}, \mathbf{u}_{e_2,j})\}_{j=1}^{N}$ and encodes this input to a representation that captures the attribute value similarities of e_1 and e_2. For each attribute A_j and pair of attribute embeddings $(\mathbf{u}_{e_1,j}, \mathbf{u}_{e_2,j})$ the operations performed by this module are split into two major parts:

(1) Attribute summarization. This module takes as input the two sequences $(\mathbf{u}_{e_1,j}, \mathbf{u}_{e_2,j})$ and applies an operation H that summarizes the information in the input sequences. More precisely, let sequences $\mathbf{u}_{e_1,j}$ and $\mathbf{u}_{e_2,j}$ contain m and k elements respectively. An h dimensional summarization model H takes as input sequences $\mathbf{u}_{e_1,j} \in \mathbb{R}^{d \times m}$ and $u_{e_2,j} \in \mathbb{R}^{d \times k}$ and outputs two summary vectors $\mathbf{s}_{e_1,j} \in \mathbb{R}^h$ and $\mathbf{s}_{e_2,j} \in \mathbb{R}^h$. The role of attribute summarization is to aggregate information across all tokens in the attribute value sequence of an entity mention. This summarization process may consider the pair of sequences $(\mathbf{u}_{e_1,j}, \mathbf{u}_{e_2,j})$ jointly to perform more sophisticated operations such as *soft alignment* [3].

(2) Attribute comparison. This part takes as input the summary vectors $\mathbf{s}_{e_1,j} \in \mathbb{R}^h$ and $\mathbf{s}_{e_2,j} \in \mathbb{R}^h$ and applies a comparison function D over those summaries to obtain the final similarity representation of the attribute values for e_1 and e_2. We denote that similarity representation by $s_j \in \mathbb{R}^l$ with $s_j = D(\mathbf{s}_{e_1,j}, \mathbf{s}_{e_2,j})$.

Architecture module		Options	
Attribute embedding		*Granularity:* (1) Word-based (2) Character-based	*Training:* (3) Pre-trained (4) Learned
Attribute similarity representation	(1) Attribute summarization	(1) Heuristic-based (2) RNN-based (3) Attention-based (4) Hybrid	
	(2) Attribute comparison	(1) Fixed distance (cosine, Euclidean) (2) Learnable distance (concatenation, element-wise absolute difference, element-wise multiplication)	
Classifier		NN (multi-layer perceptron)	

Figure 4.3: The design space of DL solutions for EM.

The output of the similarity representation module is a collection of similarity representation vectors $\{s_1, \ldots, s_N\}$, one for each attribute $A_1, \ldots, A_N$. There are various design choices for the two parts of this module. We discuss those in detail in Section 4.3.3.

The Classifier Module: This module takes as input the similarity representations $\{s_1, \ldots, s_N\}$ and uses those as features for a classifier M that determines if the input entity mentions e_1 and e_2 refer to the same real-world entity.

A Design Space of DL Solutions for EM: Our architecture template provides a set of choices for each of the above three modules. Figure 4.3 describes these choices (under "Options" on the right side of the figure). Note that we provide only one choice for the classifier module, namely a multi-layer NN, because this is the most common choice today in DL models for classification. For other modules we provide multiple choices. In what follows we discuss the choices for attribute embedding, summarization, and comparison. The numbering of the choices that we will discuss correspond to the numbering used in Figure 4.3.

4.3.2 Attribute Embedding Choices

Possible embedding choices for this module can be characterized along two axes: (1) the *granularity of the embedding* and (2) whether a *pre-trained embedding is used or a domain specific embedding is learned*. We now discuss these two axes.

(1) Word-level vs. (2) Character-level Embeddings: Given a sequence of words, a word-level embedding encodes each word in the sequence as a fixed d-dimensional vector. Procedurally, word level embeddings use a lookup table to convert a word into an embedding [101, 119]. To learn this lookup table word embeddings are trained either on large external corpora, such as Wikipedia, or on the corpus of the task in hand. An important design choice for word-level embeddings is handling out-of-vocabulary (OOV) tokens at test time [34]. A common approach is to replace infrequent words with a special token `UNK`, and use this to model OOV words.

Another option is that of character-level embeddings. This type of embedding takes as input the characters present in a word and use a neural network to produce a d-dimensional vector representation for the word. Unlike world-level embeddings where the end result of training is a lookup table, the end result here is a *trained model* that can produce word embeddings for any word containing characters in its known character vocabulary [10, 75]. The core idea behind these models is that words are made of *morphemes*, or meaningful sequences of characters, of varying lengths. For example, the word "kindness" is made of two morphemes, "kind" and "ness".

Character-level embeddings can offer significant performance improvements in domains with infrequent words (see Section 4.5.4) as they take into account the fact that many words can be morphologically related (e.g., "available", "availability" and "unavailable"). Character-level embeddings are more robust to out-of-vocabulary (OOV) words—OOV words may occur due to misspellings—as they leverage possible substrings of the word to approximate its embedding. This leads to better performance in scenarios such as entity matching where long-tail vocabularies are common and typographical errors are widespread (see Section 4.5.4).

(3) Pre-trained vs. (4) Learned Embeddings: A different choice in our template is to decide between using *pre-trained* word embeddings, such as word-level embeddings (e.g., word2vec [101] and GloVe [119]) or character-level embeddings (e.g., fastText [10]), or train embeddings from scratch. Pre-trained embeddings offer two distinctive advantages: (1) they lead to significantly smaller end-to-end training times, and (2) they have been trained over large corpora, such as Wikipedia, GoogleNews, and Gigaword, and thus, are more robust to linguistic variations. Pre-trained embeddings may not be suitable for domains where the vocabulary contains tokens with

highly specialized semantics (e.g., product barcodes for retail applications). In this case, training a domain-specific embedding can offer improved performance (see Section 4.5.4).

4.3.3 Attribute Summarization Choices

Recall that the role of attribute summarization is to aggregate information across all tokens in the attribute value sequence of an entity mention. Given the attribute embeddings $\mathbf{u}_{e_1,j} \in \mathbb{R}^{d \times m}$ and $u_{e_2,j} \in \mathbb{R}^{d \times k}$ for attribute A_j, a summarization process H outputs two summary vectors $\mathbf{s}_{e_1,j} \in \mathbb{R}^h$ and $\mathbf{s}_{e_2,j} \in \mathbb{R}^h$. We identify four major options for attribute summarization.

(1) Aggregate Function: The summarization process H corresponds to a simple aggregate function over each embedding sequence $\mathbf{u}_{e_1,j}$ and $\mathbf{u}_{e_2,j}$, e.g., average or weighted average. More precisely, function $H : \mathbb{R}^{d \times \cdot} \rightarrow \mathbb{R}^d$ is applied to each input independently and produces a d-dimensional summarization. Here, the output dimension h is equal to d. The biggest advantage of this type of summarization is *training efficiency* since there is usually no learning involved. However, models that rely on this kind of summarization cannot learn complex interactions between words in the input sequence. The performance of this summarization method depends strongly on the quality of the embedding vectors [2].

(2) Sequence-aware Summarization: Here the summarization process H aims to learn complex interactions across the tokens in the input sequences $\mathbf{u}_{e_1,j}$ and $\mathbf{u}_{e_2,j}$. Specifically, function $H : \mathbb{R}^{d \times \cdot} \rightarrow \mathbb{R}^h$ is applied to each input and produces a h-dimensional summarization. To this end, process H can be built around an RNN (see Section 4.2) so that it takes into account the order and the semantics of the tokens in the input sequence. There are many variations of RNN models [88], including long short-term memory (LSTM) networks [66], gated recurrent unit (GRU) [17] networks, and bi-directional networks [60, 96]. Given an RNN we implement process H as follows. We pass an input sequence $\mathbf{u}_{e,j}$ through the RNN to obtain a sequence $\mathbf{h}_{e,j}$ of hidden states. These hidden states are then aggregated into a single h-dimensional vector $\mathbf{s}_{e,j}$. Typical operations for this aggregation correspond to taking the last hidden state of the RNN to be $\mathbf{s}_{e,j}$ or taking an average across all hidden states $\mathbf{s}_{e,j}$ [151]. The basic advantage of this summarization method allows us to reason about the context encoded in the entire input sequence. The limitations of this method

are that (1) it does not learn meaningful representations in the presence of very long sequences (see Section 4.2), and (2) it does not analyze the inputs pairs $\mathbf{u}_{e_1,j}$ and $\mathbf{u}_{e_2,j}$ jointly to identify common contexts across sequences. The latter can lead to significant performance loss when the input sequences vary significantly in length [15].

(3) Sequence Alignment: Here, process H takes as input both sequences $\mathbf{u}_{e_1,j}$ and $\mathbf{u}_{e_2,j}$ and uses one as *context* when summarizing the other. To this end, process H can be built around attention mechanisms (see Section 4.2) that first learn to compute a soft alignment between two given sequences of words and then perform a word by word comparison [52]. Attention mechanisms are also very expressive [156] and have a significant drawback: they only leverage the context given to them as input and ignore any context present in the raw input sequence. For example, given a sequence, attention-based summarization cannot take the position of input tokens into account. As such, attention methods can perform poorly in scenarios where the most informative token for matching two entity mentions is the first one. This problem can be addressed by combining them with sequence-based summarization methods.

(4) Hybrid: These attribute summarization methods are a combination of the sequence-aware and sequence alignment methods described above (see Section 4.4). Using these methods leads to very expressive models that are expensive to train. Section 4.5 empirically shows that DL models for EM that use hybrid attribute summarization methods can be up to $3\times$ slower to train than other DL models. However, hybrid methods obtain more accurate results—up to 4.5% F_1—than other methods.

4.3.4 Attribute Comparison Choices

Recall that attribute comparison identifies the distance between the summary vectors $\mathbf{s}_{e_1,j} \in \mathbb{R}^h$ and $\mathbf{s}_{e_2,j} \in \mathbb{R}^h$ for attribute A_j. We use D to denote this comparison operation. We assume that the output of this operation is a fixed-dimension vector $s_j \in \mathbb{R}^l$. We identify two main options for the comparison operation D: (1) *fixed* and (2) *learnable* distance functions.

(1) Fixed Distance Functions: The first option is to use a pre-defined distance metric such as the cosine or Euclidean distance. Here, the output is a scalar capturing how similar the values of the

two input entity mentions are for the attribute under consideration. Using fixed distance functions leads to lower training times but enforces strong priors over the similarity of attribute values.

(2) Learnable Distance Functions: To allow for more expressivity we can rely on the classification module of our template to learn a similarity function. In this case, the output vector $s_j \in \mathbb{R}^l$ of function D forms the input (features) to the matching classifier. Different operations such as concatenation, element-wise absolute difference, element-wise multiplication or hybrids of these are viable options for D. We experimentally evaluate different operations in Section 4.5. We find that using element-wise comparison to obtain the input for the matching classifier is beneficial if an aggregate function or sequence-aware summarization method was used for attribute summarization. This is because these two methods do not perform any cross sequence comparison during summarization.

4.4 Representative DL Solutions for Entity Matching

The previous section describes a space of DL solutions for EM. We now select four DL solutions as "representative points" in this space. These solutions correspond to DL models of varying representational power and complexity—the more complex a model, the more parameters it has, thus learning requires more resources.

All four solutions use fastText [10]—a pre-trained character-level embedding—to implement the attribute embedding module of our architecture template. (Section 4.5.4 provides a detailed evaluation of the other design choices for this module.) Further, all four solutions use a two layer fully-connected ReLU HighwayNet [146] followed by a softmax layer to implement the classifier module. HighwayNets were used since they sped up convergence and produced better empirical results than traditional fully connected networks across EM tasks, especially in the case of small datasets. The four solutions use different choices for the attribute summarization process, however. They are named SIF, RNN, Attention, and Hybrid, respectively, after the choice for the attribute summarization part of the similarity representation module of our architecture.

4.4.1 SIF: An Aggregate Function Model

We first consider a model (i.e., a DL solution, we use "model" and "solution" interchangeably) that uses an aggregate function, specifically a *weighted average* for attribute summarization and an *element-wise absolute difference* comparison operation to form the input to the classifier module. Specifically, the weights used to compute the average over the word embeddings for an input sequence are as follows: given a word w the corresponding embedding is weighted by a weight $f(w) = a/(a + p(w))$ where a is a hyperparameter and $p(w)$ the normalized unigram frequency of w in the input corpus.

This model was chosen since it is a simple but effective baseline deep learning model. Its performance relies mostly on the expressive power of the attribute embedding and the classifier used. The weighting scheme used during averaging follows the Smooth Inverse Frequency (SIF) sentence embedding model introduced by Arora et al. [2]. This model was shown to perform comparably to complex—and much harder to train—models for text similarity, sentence entailment and other NLP tasks. This model is similar to the Tuple2vec-Averaging model by Ebraheem et al. [40], but is more expressive since it takes word frequencies into account.

4.4.2 RNN: A Sequence-aware Model

This second model uses a *bidirectional RNN* (i.e., a sequence-aware method) for attribute summarization and an *element-wise absolute difference* comparison operation to form the input to the classifier module. This is a medium-complexity model that takes the order of words in the input sequence into account. This model was selected since it is one of the most commonly used DL approaches for computing distributed representations of text snippets. The RNN model we use corresponds to a bidirectional GRU-based RNN model introduced by Cho et al. [17] for machine translation. Bidirectional RNNs are the de-facto deep learning model for NLP tasks [97].

We now provide a high-level description of the model. The model consists of two RNNs: the *forward* RNN that processes the input word embedding sequence $\mathbf{u}$ in its regular order (from element entry $u[1]$ to entry $u[t]$) and produces hidden states $\mathbf{f}_{1:t}$ and the *backwards* network that

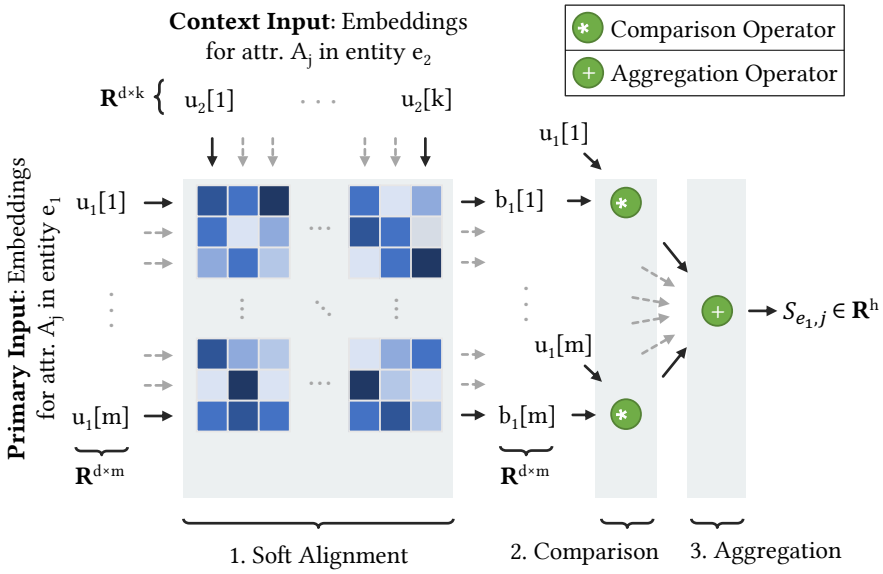

Figure 4.4: Decomposable attention-based attribute summarization module.

processes the input sequence in reverse order to produce hidden states $\mathbf{b}_{t:1}$. The final attribute summarization representation corresponds to the concatenation of the last two outputs of the bidirectional RNN, i.e., to the concatenation of f_t and b_1. In our experiments, we did not use multi-layered RNNs since we did not notice compelling performance gains with them (see Section 4.5.5 under "Performance of DL Model Variations"). This method is similar to the Tuple2Vec-Compositional DL method introduced by Ebraheem et al. [40].

4.4.3 Attention: A Sequence Alignment Model

This model uses *decomposable attention* to implement attribute summarization and *vector concatenation* to perform attribute comparison. This is a medium-complexity model that analyzes both input sequences jointly while learning a similarity representation. The attention model we use is a variation of a model introduced by Parikh et al. [116] for text entailment. This model was selected since it has been shown to achieve results close to the state of the art on NLP tasks related to EM [116]. Intuitively it performs soft alignment and pairwise token comparison across the two input sequences.

Figure 4.4 summarizes the working of Attention. Let $\mathbf{u}_1$ and $\mathbf{u}_2$ be two embedding sequences whose summarized representations we want to compute. To compute the summarized representation for $\mathbf{u}_1$ we give $\mathbf{u}_1$ as *primary input* and $\mathbf{u}_2$ as *context* to the attention model. We proceed in three steps:

(1) Soft Alignment: For each element $u_1[k]$ in the primary input $\mathbf{u}_1$ we compute a *soft-aligned encoding* of $u_1[k]$—denoted by $b_1[k]$—using all elements in the context sequence $\mathbf{u}_2$. To do so we first compute a *soft alignment matrix* $\mathcal{W}$ across all pairs of tokens for $\mathbf{u}_1$ and $\mathbf{u}_2$. Each row in $\mathcal{W}$ corresponds to an entry in $\mathbf{u}_1$ and each column to an entry in $\mathbf{u}_2$. Each entry of this matrix is a weight for a pair of elements $(u_1[k], u_2[m])$. This weight corresponds to a log-linear transformation of the dot product over the hidden representations of $u_1[k]$ and $u_2[m]$ obtained by a two layer HighwayNet. We compute the encoding $b_1[k]$ for each $u_1[k] \in \mathbf{u}_1$ by taking a weighted average over all elements $u_2[m] \in \mathbf{u}_2$ with the weights being the entries of the k-th row of $\mathcal{W}$.

(2) Comparison: We compare each embedding $u_1[k] \in \mathbf{u}_1$ with its soft-aligned encoding $b_1[k]$ using a two layer HighwayNet with ReLU non-linearities. We denote as $x_1[k]$ the comparison representation for each $u_1[k] \in \mathbf{u}_1$.

(3) Aggregation: Here, we sum the comparison representation of all elements in $\mathbf{u}_1$ and normalize the output by dividing with $\sqrt{|\mathbf{u}_1|}$. This extension over the original model ensures that the variance of the final attribute summarization does not change as a function of the number of words in the attribute. *This ensures that the gradient magnitude of the parameters in the comparison module do not depend on the length of the attribute and thus promotes robustness.*

These steps are repeated for $\mathbf{u}_2$ as primary input and $\mathbf{u}_1$ as context.

4.4.4 Hybrid: Sequence-aware with Attention

This model uses *a bidirectional RNN with decomposable attention* to implement attribute summarization and a *vector concatenation augmented with element-wise absolute difference* during attribute comparison to form the input to the classifier module. This is the model with the highest

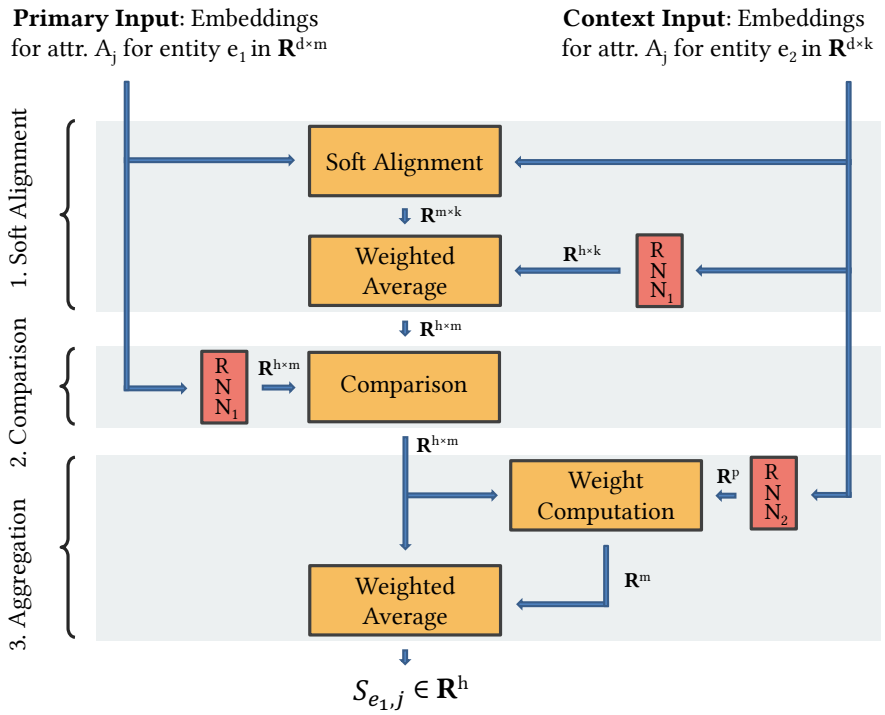

Figure 4.5: The Hybrid attribute summarization module.

representational power we consider in this project. To our knowledge we are the first to consider such a model for entity matching. Our model is inspired by other hybrid models proposed in the NLP literature [179, 161]. However, those models either build upon convolutional neural networks [161] or use different attention mechanisms [179].

We now describe the internals of this model. Again, let $\mathbf{u}_1$ and $\mathbf{u}_2$ be two embedding sequences whose summarized representations we want to compute. Our hybrid model follows steps that are similar to those of the Attention model of Section 4.4.3. But in contrast to Attention, it also utilizes sequence-aware encodings of $\mathbf{u}_1$ and $\mathbf{u}_2$ obtained by a Bi-RNN. Figure 4.5 provides an overview. We have:

(1) Soft Alignment: First, Hybrid constructs a soft alignment matrix $\mathcal{W}$ between the primary input sequence $\mathbf{u}_1$ and the context sequence $\mathbf{u}_2$. The construction is the same as that described in Section 4.4.3. Then, Hybrid obtains a soft-aligned encoding $b_1[k]$ for each element $u_1[k] \in \mathbf{u}_1$. In contrast to Attention $b_1[k]$ is constructed by taking a weighted average over an encoding of $\mathbf{u}_2$. The weights are obtained by the soft-alignment matrix. The encoding of $\mathbf{u}_2$ is obtained by passing

$\mathbf{u}_2$ through a Bi-RNN and concatenating all the hidden states of the Bi-RNN. Let RNN_1 denote this RNN. This process generates a vector $\mathbf{b}_1$ of soft-aligned encodings.

(2) Comparison: To obtain the comparison between $\mathbf{b}_1$ and the primary input $\mathbf{u}_1$ we: (i) obtain an encoding of $\mathbf{u}_1$—denoted by $\mathbf{u}'_1$ by passing it via the same RNN_1, and concatenating all of its hidden states; (ii) perform an element-wise comparison between $\mathbf{u}'_1$ and $\mathbf{b}_1$. Similar to Attention we use a two layer HighwayNet with ReLU to perform this element-wise comparison. Let $x_1[k]$ be the comparison representation for each $u_1[k] \in \mathbf{u}_1$.

(3) Aggregation: Finally, Hybrid aggregates the elements of the comparison vector $\mathbf{x}_1$ produced by the previous step using a *weighted average scheme*. The weight for each element $x_1[k]$ is obtained as follows: (i) we first obtain an encoding of $\mathbf{u}_2$—denoted g_2—by using a Bi-RNN (RNN_2) and taking its last hidden state to be the encoding of $\mathbf{u}_2$; (ii) we compute a weight for each element $x_1[k] \in \mathbf{x}_1$ by concatenating $x_1[k]$ with g_2 and passing it via a two layer ReLU HighwayNet followed by a soft-max layer. Intuitively, this corresponds to a simple attention mechanism that identifies the importance of each element $x_1[k]$ given $\mathbf{u}_2$ as context; (iii) we take a weighted average over all elements of $\mathbf{x}_1$ using these weights.

The same steps are repeated for $\mathbf{u}_2$ as primary input and $\mathbf{u}_1$ as context. Steps 1-2 in this model are different from typical hybrid architectures such as [161, 179]. The modifications make the alignment computation between the input word embedding sequences to not rely on the RNN encoding of the input. This enables the model to converge faster since the alignment network can receive useful gradients right from the start of training when the RNN weights are random.

4.5 Empirical Evaluation

Goals and Takeaways: Our first goal is to understand where DL outperforms current EM solutions. Toward this goal, we experimentally compare the four DL models from Section 4.4 (i.e., SIF, RNN, Attention, and Hybrid) with Magellan, a state-of-the-art learning-based EM solution [79]. We used 23 EM tasks that cover three types of EM problems: structured, textual, and dirty.

Problem Type	Average F_1			Average Train Time	
	DL	Magellan	ΔF_1	DL	Magellan
Structured	87.9	88.8	-0.9	5.4h	1.5m
Textual	88.0	83.4	4.6	4.4h	6s
Textual w/o info. attr.	88.3	78.7	9.6	4.0h	5.5s
Dirty	87.9	68.5	19.4	0.7h	1.5m

Table 4.1: Comparison of Magellan (a current ML-based solution) vs. the best-performing DL solution.

The main takeaways are as follows. (1) On structured EM tasks, DL solutions are competitive with Magellan but takes far more training time. Thus, it is not yet clear to what extent DL can help structured EM (see Table 4.1). (2) On textual EM tasks (i.e., instances having a few attributes all of which are textual blobs), DL outperforms Magellan. The gain may not be large if there are "informative" attributes (e.g., titles full of discriminative information), otherwise it can be significant. (3) On dirty EM tasks, DL significantly outperforms Magellan. Thus, we find that in the absence of labor-intensive data extraction/cleaning DL is highly promising, outperforming current automatic EM solutions for textual and dirty data.

Our second goal is to understand the impact of performance factors, such as model complexity (e.g., do we need complex DL models?) and amount of labeled data. The main takeaways are as follows. (1) When a limited amount of training data is available, models that use soft alignment (see Section 4.3.3) during attribute summarization should be preferred as they yield up to 23% higher F_1 over simpler DL models (see Section 4.5.4.2). (2) When a lot of training data is available, the accuracy difference between complex and simpler DL models is smaller. Thus, one can use simpler DL models that are faster to train (see Section 4.5.4).

Datasets: We use datasets from a diverse array of domains and different sizes (see Table 4.2). Dataset details are deferred to the technical report [106]. For structured EM, we use 11 datasets. The first seven datasets are publicly available and have been used for EM (e.g., [29, 81]). The last four datasets (Clothing$_1$, etc.) describe products in various categories and come from a major retailer. Column "Size" lists the number of labeled examples for each dataset. Each example has

Type	Dataset	Domain	Size	# Pos.	# Attr.
Structured	BeerAdvo-RateBeer	beer	450	68	4
	iTunes-Amazon$_1$	music	539	132	8
	Fodors-Zagats	restaurant	946	110	6
	DBLP-ACM$_1$	citation	12,363	2,220	4
	DBLP-Scholar$_1$	citation	28,707	5,347	4
	Amazon-Google	software	11,460	1,167	3
	Walmart-Amazon$_1$	electronics	10,242	962	5
	Clothing$_1$	clothing	247,627	105,608	28
	Electronics$_1$	electronics	249,904	98,401	28
	Home$_1$	home	249,513	111,714	28
	Tools$_1$	tools	249,317	96,836	28
Textual	Abt-Buy	product	9,575	1,028	3
	Company	company	112,632	28,200	1
	Clothing$_2$	clothing	247,627	105,608	3
	Electronics$_2$	electronics	249,904	98,401	3
	Home$_2$	home	249,513	111,714	3
	Tools$_2$	tools	249,317	96,836	3
Dirty	iTunes-Amazon$_2$	music	539	132	8
	DBLP-ACM$_2$	citation	12,363	2,220	4
	DBLP-Scholar$_2$	citation	28,707	5,347	4
	Walmart-Amazon$_2$	electronics	10,242	962	5
	Home$_3$	home	249,513	111,714	28
	Tools$_3$	tools	249,317	96,836	28

Table 4.2: Datasets for our experiments.

two tuples to be matched. The tuples are *structured*, i.e., attribute values are atomic, i.e., short and pure, and not a composition of multiple values that should appear separately.

For textual EM, we use six datasets. Abt-Buy describes products [81]. Company is a dataset that we created. It tries to match company homepages and Wikipedia pages describing companies. The last four datasets (Clothing$_2$, etc.) describe products in different categories and come from a major retailer. In these datasets, each tuple has 1-3 attributes, all of which are *long textual blobs* (e.g., long title, short description, long description).

For dirty EM, we use six datasets. As discussed earlier, we focus on dirty data where attribute values are "injected" into other attributes, e.g., the value of brand is embedded in title while leaving the correct value cell empty. All dirty datasets are derived from the corresponding structured datasets described above. To generate them, for each attribute we randomly move its value to attribute title in the same tuple with 50% probability. This simulates a common dirty-data problem in real-world scenarios (e.g., information extraction) while keeping the modifications simple.

Dataset	Model F_1 Score					ΔF_1
	SIF	RNN	Attention	Hybrid	Magellan	
BeerAdvo-RateBeer	58.1	72.2	64.0	72.7	78.8	-6.1
iTunes-Amazon$_1$	81.4	88.5	80.8	88.0	91.2	-2.7
Fodors-Zagats	100	100	82.1	100.0	100	0.0
DBLP-ACM$_1$	97.5	98.3	98.4	98.4	98.4	0.0
DBLP-Scholar$_1$	90.9	93.0	93.3	94.7	92.3	2.4
Amazon-Google	60.6	59.9	61.1	69.3	49.1	20.2
Walmart-Amazon$_1$	65.1	67.6	50.0	66.9	71.9	-4.3
Clothing$_1$	96.6	96.8	96.6	96.6	96.3	0.5
Electronics$_1$	90.2	90.6	90.5	90.2	90.1	0.5
Home$_1$	87.7	88.4	88.7	88.3	88.0	0.7
Tools$_1$	91.8	93.1	93.2	92.9	92.6	0.6

Table 4.3: Results for structured data.

Methods: We evaluate the four DL models described in Section 4.4: SIF (aggregation-based), RNN (RNN-based), Attention (attention-based), and Hybrid. They are implemented using Torch [24] (a DL framework with extensive support for accelerating training using GPUs), and trained and evaluated on AWS p2.xlarge instances with Intel Xeon E5-2686 CPU, 61 GB memory, and Nvidia K80 GPU. We compare the DL models with Magellan, a state-of-the-art machine-learning based EM solution [79].

To measure accuracy, we use precision (P), the fraction of match predictions that are correct, recall (R), the fraction of correct matches being predicted as matches, and F_1, defined as $2PR/(P+R)$.

To apply the DL models to a dataset, we split all pairs in the dataset (recall that each example is a pair of tuples) into three parts with ratio of 3:1:1, for training, validation, and evaluation respectively. We use Adam [76] as the optimization algorithm for all DL models for 15 training epochs. The validation set is used to select the best model snapshot for evaluation after each epoch to avoid over-fitting (see the technical report [106] for more details on training DL models). To apply Magellan to a dataset, we use the same 3:1:1 data split. We train five classifiers (decision tree, random forest, Naive Bayes, SVM and logistic regression), use the validation set to select the best classifier, and then evaluate on the evaluation set. It is important to note that Magellan uses the tuple attributes to automatically generate a large set of features used during training.

4.5.1 Experiments with Structured Data

Table 4.3 shows the accuracy of the four DL models and Magellan on the 11 structured datasets. We use red font to highlight the highest score in each row. The last column shows the relative increase in F_1 for the best DL model vs Magellan (see detailed in the technical report [106]).

The results show that DL models perform comparably with Magellan. The best DL model outperforms Magellan in 8 out of 11 cases. But the gain is usually small (less than 0.7%, see the last column), except 2.4% for DBLP-Scholar$_1$ and 20.2% for Amazon-Google (AG). This is because the product titles across matching pairs from the source datasets (Amazon and Google) correspond to synonyms of one another. That is, they are semantically similar but have large string similarity distances. This makes it difficult for Magellan to capture the similarity between mentions as it mainly relies on string similarity based features. However, DL can identify the semantic similarities across data instances to achieve a 20% F_1 gain. In the case of Walmart-Amazon$_1$, however, this ability to learn rich information from data hurts DL, because the model overfits due to the quirks of the training set, thus reaching 4.3% less F_1 compared to Magellan. This is less of an issue for Magellan which has a more restricted search space (as it uses string similarity-based features).

Hybrid performs the best among the four DL models. As discussed previously, DeepER, the current DL solution for EM [40], is comparable to SIF and RNN. Our results suggest that these models are not adequate, and that we should explore more sophisticated models, such as Hybrid, to obtain the highest possible EM accuracy.

Moreover, DL does appear to need a large amount of labeled data to outshine Magellan. The first three datasets (in Table 4.3) have only 450-946 labeled examples. Here DL performs worse than Magellan, except on Fodors-Zagats, which is easy to match. The next four datasets have 12.3K-28.7K labeled examples. Here DL outperforms Magellan in two cases. It performs reliably better than Magellan in the last four datasets, which have about 249K labeled examples.

Dataset	Model F_1 Score					ΔF_1
	SIF	RNN	Attention	Hybrid	Magellan	
Abt-Buy	35.1	39.4	56.8	62.8	43.6	19.2
Clothing_2	84.7	85.3	85.0	85.5	82.5	3.0
Electronics_2	90.4	92.2	91.5	92.1	85.3	6.9
Home_2	84.5	85.5	86.1	86.6	82.3	4.3
Tools_2	92.9	94.5	93.8	94.3	90.2	4.3

Table 4.4: Results for textual data (w. informative attributes).

Dataset	Model F_1 Score					ΔF_1
	SIF	RNN	Attention	Hybrid	Magellan	
Abt-Buy	32.0	38.5	55.0	47.7	33.0	22.0
Company	71.2	85.6	89.8	92.7	79.8	12.9
Clothing_2	84.6	84.4	84.6	84.3	78.8	5.8
Electronics_2	89.6	90.4	90.8	91.1	82.0	9.1
Home_2	84.0	84.8	83.7	85.4	74.1	11.3
Tools_2	91.6	92.5	92.6	93.0	84.4	8.6

Table 4.5: Results for textual data (w.o. informative attributes).

4.5.2 Experiments with Textual Data

Textual datasets have few attributes, all of which are text blobs. For 5 out of 6 textual datasets listed in Table 4.2, we observe that they contain an "informative" attribute that packs a lot of information (i.e., having a high "signal-to-noise" ratio), which is title (containing brand name, product model, etc.). For these datasets, we expect Magellan to do well, as it can use similarity measures such as Jaccard to compare this attribute. Table 4.4 shows that this is indeed the case. Yet DL's F_1-score is 3.0-19.2% higher than Magellan's.

If we remove this "informative" attribute, the gain increases from 5.8 to 22.0% relative F_1, as shown in Table 4.5. One textual dataset, Company, has no such informative attribute. For this dataset, DL outperforms Magellan, achieving 92.7% F_1 vs 79.8% F_1.

The results suggest that DL outperforms Magellan for textual EM. They suggest that Hybrid is still the best among the DL models. We also find that the F_1-score difference between the best performing attention-based model (Attention or Hybrid) and the best performing model that does not use soft alignment (SIF and RNN) is around 4.5 points on average but goes up to 23.5 points (i.e., for the Abt-Buy dataset in Table 4.4). To understand this better, we investigated examples

Dataset	Model F_1 Score					ΔF_1
	SIF	RNN	Attention	Hybrid	Magellan	
iTunes-Amazon$_2$	66.7	79.4	63.6	74.5	46.8	32.6
DBLP-ACM$_2$	93.7	97.5	97.4	98.1	91.9	6.2
DBLP-Scholar$_2$	87.0	93.0	92.7	93.8	82.5	11.3
Walmart-Amazon$_2$	43.2	39.6	53.8	46.0	37.4	16.4
Home$_3$	82.8	86.4	88.0	87.2	68.6	19.4
Tools$_3$	88.5	92.8	92.6	92.8	76.1	16.7

Table 4.6: Results for dirty data.

from datasets where the F_1 between the two types of methods is large. We found that in all cases the two data instances corresponded to misaligned word sequences similar to those found in the problem of text entailment. An example is matching sequences "samsung 52 ' series 6 lcd black flat panel hdtv ln52a650" and "samsung ln52a650 52 ' lcd tv".

4.5.3 Experiments with Dirty Data

Recall that dirty datasets are those where some attribute values are not in their correct cells but in the cells of some other attributes (e.g., due to inaccurate extraction). In such cases, we can ignore the "attribute boundaries" and treat the entire tuple as a single text blob, then apply DL, as in the textual case. The results in Table 4.6 show that DL significantly outperforms Magellan, by 6.2-32.6% relative F_1. Interestingly, even in the presence of extensive noise, for four out of six dirty datasets, Hybrid and Attention still perform only at most 1.1% lower in F_1 compared to their scores for the corresponding structured datasets. This suggests that DL is quite robust to certain kinds of noise (e.g., moving attribute values around).

4.5.4 Trade-offs for Deep Entity Matching

We now validate that the different design choices described in Section 4.3.1 have an impact on the performance of deep learning for entity matching. We report on trade-offs related to all design choices introduced by our architecture template.

Attribute Embedding		Structured		Textual		Dirty	
		Home_1	Tools_1	Company	Home_2	Home_1	Tools_1
Pre-trained	Glove	86.5	86.5	93.5	86.6	87.1	88.3
	fastText	88.3	92.9	92.7	86.6	87.2	92.8
Char-based learned		88.2	92.8	87.7	86.9	87.5	93.8

Table 4.7: F_1-score for Hybrid with different language representations.

4.5.4.1 Language Representation Selection

We validate that different types of language representations lead to DL models with different performance and that no single option dominates the others. We run our Hybrid model over structured, textual, and dirty data and compare different language representations with respect to (1) the granularity of the embeddings (i.e., word vs. character) and (2) pre-trained vs. learned. The results are shown in Table 4.7.

Word vs. Character: We compare GloVe [119] (a word-level embedding) with fastText [10] (a character-level embedding). Table 4.7 shows that the F_1-scores are similar but there are two exceptions. (1) On Tools fastText achieves 6% higher F_1-scores. This is because Tools' vocabulary include domain-specific words not present in the vocabulary of GloVe. Recall that GloVe maps all OOV words to the same embedding. FastText can approximate the embedding of those words by using a character-level strategy. (2) On Company GloVe outperforms fastText, though the by less than one point F_1. Recall that the entries in Company are obtained from Wikipedia which corresponds to the corpus used to train GloVe. In general, we find that character-level embeddings often obtain higher F_1-scores than word-level embeddings.

Pre-trained vs. Learned: We compare fastText with a character-level embedding trained from scratch. Table 4.7 shows that the two obtain similar F_1-scores with two exceptions. (1) The learned-from-scratch model is better for Tools, suggesting that learning an embedding from scratch is beneficial for highly-specialized datasets. (2) fastText achieves 5% higher F_1 on Company, demonstrating the effect of limited training data when learning an embedding from scratch. Overall, we find that training an embedding from scratch does not lead to performance improvements unless we perform EM over domains with highly specialized vocabularies.

Dataset		SIF	RNN	Attention	Hybrid	Magellan
Structured	small	3-70s	5-15m	7-25m	10-45m	1s
	large	25m	6.5-7h	7-7.5h	9.5-11h	2-4m
Textual	small	15s	5m	7.5m	15m	1s
	large	8-16m	3-6h	3-6h	7-10h	9-12s
Textual w/o info. attr.	small	13s	4m	5m	10m	1s
	large	6-9m	3-3.5h	3-3.5h	6.5-9h	8-12s
Dirty	small	3-30s	2.5-7m	3-10m	5-20m	1s
	large	5m	25-35m	40-55m	1-1.5h	2-4m

Table 4.8: Train time comparison for different deep learning solutions and Magellan for different dataset types and sizes.

4.5.4.2 Attribute Summarization Selection

So far we have shown that attribute summarization methods with cross-sequence alignment (Attention and Hybrid) outperform simpler methods (SIF and RNN). We now validate the *accuracy vs. training time* trade-off with respect to the attribute summarization used by different DL solutions. The factors that affect this trade-off are the complexity of the model and the size of the input dataset.

The complexity of a DL solution depends on the attribute summarization choice in our DL architecture template. In general, the more expressive (complex) a model is, the higher its accuracy will be, but with longer training time. To validate the accuracy vs. training time trade-off we ran the four DL methods and Magellan while varying the size of the input dataset from "small" to "large". The results are shown in Tables 4.8 and 4.9.

We observe that the F_1-score gap between attribute summarization methods that perform cross-sequence alignment (Attention and Hybrid) and those that do not (SIF and RNN) decreases as the size of the input datasets increases. The F_1-score difference is up to $8\times$ larger for small datasets than for large datasets—23.5 vs. 2.8 F_1 point difference. On the other hand, the training time for cross-sequence alignment models increases dramatically for large datasets (it sometimes exceeds 10 hours).

We attribute the above results to the soft alignment performed by attention mechanisms. Word embeddings coupled with a soft-alignment mechanism already capture similarity and comparison

Dataset		SIF	RNN	Attention	Hybrid	Magellan
Structured	small	79.1	83.9	76.2	84.6	86.5
	large	91.0	91.8	91.8	91.5	91.3
Textual	small	35.1	39.4	56.8	62.8	43.6
	large	87.5	88.9	88.8	89.3	83.9
Textual w/o info. attr.	small	32.0	38.5	55.0	47.7	33.0
	large	86.1	86.9	88.4	89.7	80.2
Dirty	small	76.8	86.2	78.2	84.2	64.7
	large	85.6	89.6	90.3	90.0	72.3

Table 4.9: F_1-score comparison for different deep learning solutions and Magellan for different dataset types and sizes.

semantics. On the other hand, mechanisms that encode each input entity mention in isolation (SIF and RNN) rely only on the final classifier module to capture similarity and comparison semantics. We conjecture this to be the reason why methods that perform soft alignment are superior in the presence of little training data (i.e., small datasets). However, with more training data SIF and RNN are more attractive as they take far less time to train.

4.5.4.3 Attribute Comparison Selection

We validate the effect of different attribute comparison functions (see Section 4.3.4) on the performance of DL models. We fix the attribute summarization strategy for the four solutions discussed in Section 4.4 and for each solution we vary the attribute comparison function used. We find that fixed distance functions perform poorly overall. Thus, we focus on the results for learnable distance functions. Specifically, we evaluate the performance of (1) concatenation, where the final classifier is responsible for learning the semantics of a distance between the entity mention encodings, and (2) element-wise absolute distance, where the features to the final classifier already capture the semantics of distance. Table 4.10 shows the results. We observe that for methods without cross-sequence alignment, using an element-wise comparison leads to F_1 improvements of up to 40% F_1 (see the results for SIF and RNN). For methods with cross-sequence alignment, however, there is no dominating option.

Model	Comparison	Abt-Buy	W-A$_1$	Home$_1$
SIF	Concatenation	22.6	34.7	83.8
	Element-wise Abs. Diff.	35.1	60.6	87.7
RNN	Concatenation	25.9	27.0	86.8
	Element-wise Abs. Diff.	38.5	67.6	88.4
Attention	Concatenation	54.9	50.0	88.7
	Element-wise Abs. Diff.	36.0	65.9	87.6
Hybrid	Concatenation	64.7	60.0	86.0
	Element-wise Abs. Diff.	39.3	67.1	86.7

Table 4.10: F_1-score as we vary attribute comparison choices.

4.5.5 Micro-benchmarks

We perform micro-benchmark experiments to evaluate: (1) the effect of training data on the accuracy of models, (2) the sensitivity of DL to noisy labels, (3) how DL models compare to domain-specific approaches to EM, and (4) how different variations in the DL architecture, such as different dropout levels, using multiple layers, etc. affect the performance of DL. We find that Hybrid—the most expressive out of all DL models—is more effective at exploiting the information encoded in training data, DL is significantly more robust to noise than traditional learning techniques (e.g., used in Magellan), DL methods are competitive to domain-specific methods when we have access to 10K training examples or more, the performance of DL is robust to variations in the type of RNN network used (e.g., LSTM vs. GRU), the dropout levels, and the number of layers in the recurrent part of the architecture. We discuss these experiments in more detail below.

Varying the Size of Training Set: We analyze the sensitivity of different entity matching methods to changes in the amount of available training data. For each of the three types of datasets considered, we pick two large representative datasets and analyze how the performance of two DL models and Magellan varies as we change the size of training data. The results are shown in Figure 4.6. For each dataset, we keep the ratio of training set size to validation set size constant (used 3:1 as we discussed in Section 4.5), and vary the total number of entity pairs in these two sets (called dataset size from here on) by sampling from the original large datasets. We pick the best DL model considered, i.e., the hybrid model and the simplest and fastest DL model, i.e., the SIF model, for this analysis.

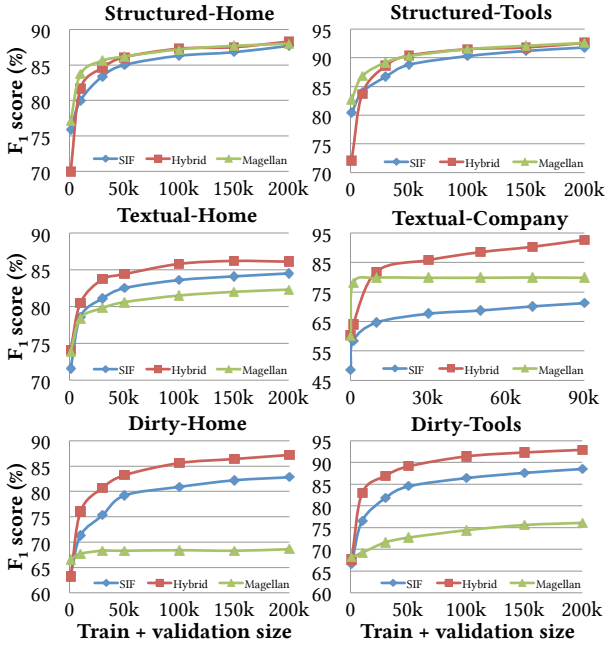

Figure 4.6: Varying the training size.

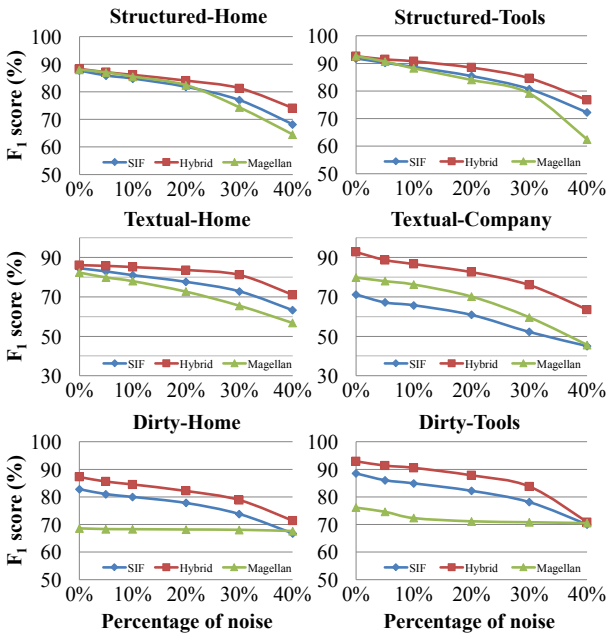

Figure 4.7: Varying label noise.

In the case of structured data, we see that Magellan outperformed the hybrid DL model when the dataset size is less than 50K. With more data, the hybrid model starts becoming comparable to Magellan and largely stays this way, until the dataset size reaches 200K at which point it slightly outperforms Magellan.

For textual data we picked dataset "Home-Textual" with an atomic informative attribute ("Title") and the purely textual dataset "Company". In the first case, we see that DL starts outperforming Magellan even with a dataset size of 1K but the difference is not very significant. With a few thousand instances, the difference becomes more significant. In the purely textual case, we see that Magellan quickly attains a relatively high F_1 score with a dataset size of just 1K, due to its heuristic-based string similarity features, while the hybrid DL model lags behind. It takes nearly 10K data instances before the hybrid model starts outperforming Magellan. With more data its performance steadily continues to increase until we finally exhaust our set of labeled data instances.

For dirty data, Magellan initially outperforms DL when the dataset size is only 1K, but DL starts outperforming Magellan when a few thousand training instances are available.

Type	Dataset	DL-Hybrid	DS-Magellan	ΔF_1	DS Approach
Structured	Clothing	96.6	96.5	-0.1	(1) Create domain specific features. (2) Train classifiers using Magellan.
	Electronics	90.2	91.3	1.1	
	Home	88.3	89.3	1.0	
	Tools	92.9	94.0	1.1	
Textual	Clothing	85.5	89.2	3.7	(1) Perform IE to extract attributes from text. (2) Create domain specific features. (3) Train classifiers using Magellan.
	Electronics	92.1	90.9	-1.2	
	Home	86.6	89.0	2.4	
	Tools	94.3	93.9	-0.4	
Dirty	Clothing	96.3	96.5	0.2	(1) Clean all dirty attributes. (2) Create domain specific features. (3) Train classifiers using Magellan.
	Electronics	89.0	91.3	2.3	
	Home	87.2	89.3	2.1	
	Tools	92.8	94.0	1.2	

Table 4.11: Comparison to domain specific approaches.

Robustness to Label Noise: We introduce noise in the match / non-match labels assigned to entity pairs in six EM datasets, two from each EM category considered. We picked the same six datasets as in the previous experiment varying training set size. The results are shown in Figure 4.7. For each dataset, we introduce label noise in the range of 0 - 40%. For example, for the case with 20% noise, we flip the labels of a randomly selected 20% subset of the entity pairs in the dataset.

In all cases we see that the hybrid model is fairly robust to noise especially until 30% noise, after which we see a steeper drop. On structured and textual datasets, we note that the performance gap between the hybrid model and Magellan increases as the noise increases indicating that the hybrid model is more robust to noise.

Comparison to Domain-Specific EM Approaches: We would like to understand how DL models compare to domain specific (DS) EM approaches involving manual information extraction and feature engineering. To do so, we take four product datasets from each of the three EM categories described in Section 4.2.1. For each dataset, we compare the performance of our best DL model considered with rigorous DS approach based EM.

We perform several experiments, the results of which are shown in Table 4.11. In the "Approach" column we describe how the domain specific EM was performed. In general, for each

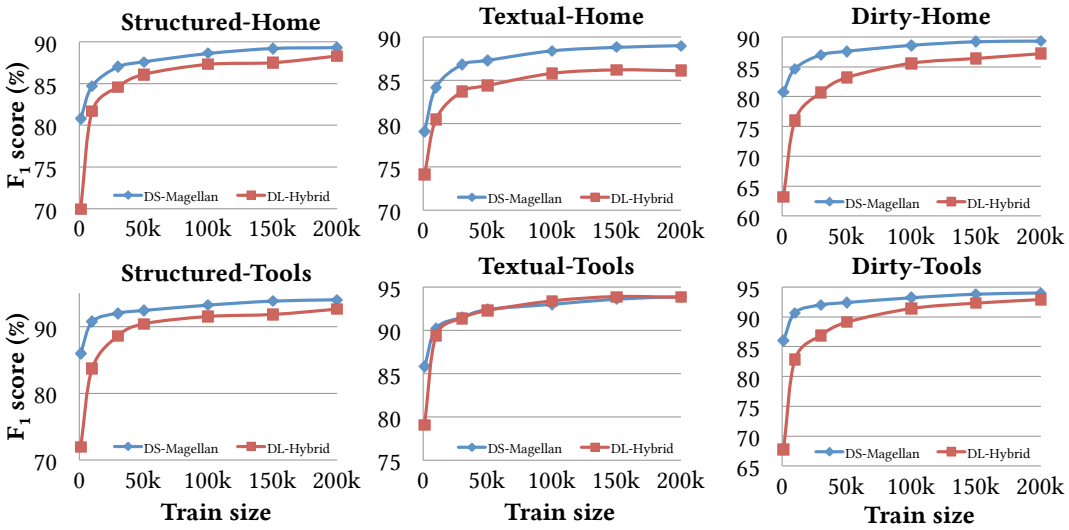

Figure 4.8: Varying the training size (domain specific).

dataset we first perform IE and feature engineering, then train a machine learning based system (Magellan) using these features.

Note that in the case of textual datasets we make use of structured information extracted from the textual attributes by data analysts over *several months*. We also make use of relevant ideas discussed in [82] to perform our own extraction and feature engineering which took multiple days.

We see that domain specific extraction does help as compared to the DL model without domain specific features. However, the average improvement is only 0.8%, 1.1% and 1.4% respectively for structured, textual and dirty datasets. The DL model was able to approach quite close to the performance of months of intensive domain specific EM effort with less than half a day of training.

We also perform additional experiments to investigate how DL models compare to domain specific approaches in the presence of limited training data. We vary the training data size in the same way as described in the paragraph "Varying the Size of Training Set" above. The results in Figure 4.8 indicate that with very limited training data, DS EM approaches are very helpful. However, with a few 10s of thousands of labeled data instances the DL model catches up — with 10K labeled instances, the hybrid DL model is within 5.1 percent points of the DS approach on average, and within 2.9 percent points on average with 50K labeled data.

RNN Unit & Layers	Layer Stacking	Dropout		Attention	Structured		Textual		Dirty	
		Probability	Location		Home	Tools	Company	Home	Home	Tools
1L GRU	Standard	0	None	Standard	88.3	92.9	92.7	86.6	87.2	92.8
2L GRU	Standard	0	None	Standard	87.6	92.5	92.8	86.4	87.1	92.5
1L LSTM	Standard	0	None	Standard	88.1	92.7	93.3	86.4	87.8	93.5
2L LSTM	Standard	0	None	Standard	87.5	92.0	93.7	85.8	88.3	93.0
2L GRU	Residual	0	None	Standard	88.0	93.0	92.5	86.7	87.3	93.2
2L GRU	Highway	0	None	Standard	88.0	93.4	93.0	86.6	87.6	93.3
1L GRU	Standard	0.05	Before RNN	Standard	88.0	93.7	92.4	85.7	87.6	93.0
1L GRU	Standard	0.2	Before RNN	Standard	87.5	93.4	92.7	84.8	88.5	93.8
1L GRU	Standard	0.05	After RNN	Standard	88.1	92.8	92.4	86.4	87.3	92.9
1L GRU	Standard	0.2	After RNN	Standard	87.7	92.6	91.8	86.0	87.1	92.9
2L GRU	Standard	0.05	Between RNN layers	Standard	87.7	92.5	92.4	86.2	86.8	92.5
2L GRU	Standard	0.2	Between RNN layers	Standard	87.3	92.3	92.5	85.6	87.9	92.5
1L GRU	Standard	0	None	2 head attention	88.3	92.8	92.5	86.6	87.2	92.8
1L GRU	Standard	0	None	Scaled dot product	88.3	92.8	87.0	86.5	86.7	92.7
2L LSTM	Standard	0.2	Between RNN layers	Standard	87.3	92.1	93.4	85.2	88.6	93.2
2L LSTM	Highway	0	None	Standard	88.2	92.7	93.5	86.5	88.5	94.0
1L LSTM	Standard	0.2	Before RNN	Standard	87.2	93.0	93.3	84.2	88.8	94.3

Table 4.12: Model variations.

Performance of DL Model Variations: As with most DL models, the models we presented in Section 4.4 can be altered along several dimensions to obtain variants. We analyzed several variants of the best model we considered, i.e., the hybrid model in order to determine its optimal configuration. To do so we altered the model along 4 primary dimensions and evaluated 17 variants of the hybrid model on six datasets, two from each EM category discussed in Section 4.2.1. We picked the same six datasets as in the experiment varying training set size. We present the F_1 scores corresponding to each variant for each dataset considered in Table 4.12. In order to efficiently determine the optimal setting from the large space of configurations formed by the 4 primary dimensions, we initially make the assumption that all dimensions are independent of each other, and vary each dimension one by one to form the first 14 variants listed in Table 4.12. After this set of experiments, we altered multiple dimensions concurrently, based on the best configurations we observed for each dimension independently. The last 3 rows in Table 4.12 show the results for these.

We note that no single configuration is universally much better than the most basic variant of the hybrid model (standard 1 layer GRU, highlighted in blue in Table 4.12). The best setting compared to this variant, the 2 layer LSTM with highway stacking (second last row in Table 4.12), is only 0.5% better on average across the six datasets. Moreover, the maximum improvement in F_1

Model	Structured		Textual		Dirty	
	Home	Tools	Company	Home	Home	Tools
Hybrid	88.3	92.9	92.7	86.6	87.2	92.8
RNN	88.4	93.1	85.6	85.5	86.4	92.8
Neculoiu et al.	84.2	86.0	83.1	81.7	78.3	86.1

Table 4.13: Comparing the F_1 accuracy of the Hybrid and RNN models in our submission with the model proposed in [110] for an NLP task.

score by any variant compared to the most basic setting across the six datasets is only 1.6%. Hence for our analysis we only considered the simplest setting of the hybrid model, with no bells and whistles, to keep the exposition straightforward and to avoid unnecessary complexity. However, in practical application scenarios, the model variation dimensions can be treated as hyperparameters and the best configuration can be automatically discovered using hyperparameter optimization tools.

Comparing with Other Ways of Formulating EM: As discussed in Section 4.2.1, the triplet framework [67] can be adapted for EM. We have done so and compared it with the hybrid model, on two structured datasets (Home and Tools), two textual datasets (Company and Home), and two dirty datasets (Home and Tools). The triplet solution performs significantly worse than Hybrid in all six cases, with a F_1 score difference ranging from 2.5% to 21.1%, or 7.6% lower on average. This is likely due to the fact that our DL model directly optimizes the DL model to maximize the classification accuracy, whereas the triplet approach focuses on obtaining better hidden representations of entities, even at the cost of classification performance. We have also empirically found that the solution in [110] performs worse than Hybrid and RNN (e.g., by 4.1-9.6% compared to Hybrid). We present the results for the comparison with [110] in Table 4.13.

4.6 Discussion

4.6.1 Understanding What DL Learns

To gain insights into why DL outperforms Magellan on textual and dirty data, we focus on Hybrid and use first derivative saliency [73, 91, 92, 148] to analyze what it learns. Saliency indicates how sensitive the model is to each word in the string sequence of an entity mention. We

expect words with higher saliency to influence the model output more. This is an indirect measure of how important a word is for Hybrid's final prediction. We consider "Home" and "Company" and compute the importance of each word in one attribute. Overall, we find that Hybrid was able to assign high weights to tokens that carry important semantic information, e.g., serial numbers of products, names of locations or people associated with an entity and special entity-specific attributes such as patterns or product color (see [106]). We obtained similar results for all datasets. Finally, we find that the errors made by DL models are mostly due to (1) linguistic variations of domain-specific terms, (2) missing highly-informative tokens, and (3) tokens that are similar but semantically different. A detailed discussion can be found in our technical report [106].

4.6.2 Challenges and Opportunities

We discuss challenges (C) and opportunities (O) for DL for EM.

(C1) DL for Structured Data: Overall, we find the advantages of sophisticated DL methods to be limited when operating over clean, structured data, in that simpler learning models (e.g., logistic regression, random forests) often offer competitive results. This topic will need more empirical evaluation before we can reach a solid conclusion.

(C2) Scalability vs. Accuracy: For textual or dirty data, we find that complex DL offer significant accuracy improvements over existing state-of-the-art EM methods, but often require far longer training time. Their poor scalability will need to be addressed. (In addition, we find that given a large amounts of training data one can leverage simpler and faster DL solution without significant losses in accuracy.)

(C3) The Value of Training Data: As expected, DL models often require large amounts of training data to achieve good performance. Obtaining large amounts of training examples can be resource-intensive in many practical scenarios. Recent work on weak supervision [123] focuses on obviating the need for tedious and manual annotation of training examples. What makes this challenge unique to EM is that existing weak supervision approaches have focused primarily on textual data for tasks such as information extraction [123, 171], or visual data for tasks such as

image classification [104]. As such, a fundamental challenge for DL-based EM is to devise new weak supervision methods for structured data as well as methods that are more robust to the class imbalance (between positive and negative examples) in EM.

(O1) DL and Data Integration: Our DL results suggest that DL can be promising for many problems in the broader field of data integration (DI), e.g., data cleaning, automated data extraction, data reformatting, and value canonicalization. Preliminary successes have already been reported in recent work [171, 135] but more effort to examine how DL can help DI is necessary.

(O2) Optimizers for DL Models: As showed in Section 4.5 there are several design choices with trade-offs when constructing DL models for EM, e.g., the choice of attribute summarization network, the kind of word embeddings, etc. An exciting future direction is to design simple rule-based optimizers that would analyze the EM task at hand and automate the deployment of such DL models. We can also explore how to use database-inspired optimization techniques to scale up DL models [163].

(O3) Semantic-aware DL: Our study revealed that DL has limited capability of capturing domain-specific semantics. A promising research direction is to explore mechanisms for introducing domain-specific knowledge to DL models. We envision this to be possible either via new weak-supervision methods [171] or by integrating domain knowledge in the architecture of DL models itself [22]. We also envision the design of new domain-specific representation learning models, such as domain-specific word embeddings for EM.

Chapter 5

Deep Learning for Blocking: A Design Space Exploration

5.1 Introduction

In Chapter 4, we showed how DL can be applied to matching. The results are promising: with little manual feature engineering, DL produces competitive matching accuracy against state-of-the-art non-DL work on structured EM problems, and much better accuracy on textual and dirty EM problems. Therefore a natural follow-up question to ask is that whether we can use DL for blocking. Specifically, what kind of techniques can be applied to blocking? Are there any tasks related to blocking such that we can adapt the DL approaches? Suppose there exist such solutions, will they work on different EM application settings as described in Section 4.2.1? How would the DL solutions compare to the existing non-DL ones? Would DL for blocking scale to large datasets?

To answer these questions, following the style of Chapter 4, we explore a design space using DL for blocking. As described in Section 4.2.1, there are different EM problem settings. In this work we again focus on all of them: *structured*, *textual* and *dirty* data. For all settings, we receive two tables as the input to the blocker and produce a candidate set which later will be used in the matching step.

To explore DL for blocking, we first define a DL architecture template, by summarizing existing DL work on blocking as well as distilling and adapting non-DL work. This template consists of three modules: *a word embedding module* to convert each of the given input tuples to a sequence of word embeddings, *a tuple embedding module* to summarize each embedding sequence into an embedding vector, and *a vector-based pairing module* to pair embedding vectors with high similarity scores across two tables, to produce a candidate set. Then we explore a space of DL

solutions with multiple choices for each module. Realizing the unsupervised nature of blocking which is different from matching, the DL techniques proposed in Section 4.4 cannot be used for tuple embedding summarization. Instead, we choose a set of self-training based DL approaches for the tuple embedding module realization.

To see if DL can help blocking, we select five solutions as representatives in the proposed design space, and compare them with RBB (Rule-Based Blocking) [55], a state-of-the-art non-DL rule-based blocking solution using active learning. The results show that it is not clear whether DL can help blocking on structured data, but it provides better blocking results on textual and dirty data. Also, with GPU acceleration, the proposed DL solutions can be executed efficiently.

5.2 Preliminaries and Related Work

5.2.1 Entity Matching

Problem Setting: In Section 2.1 we describe the EM workflow which consists of two major steps: blocking and matching. In this project we focus on the blocking step of EM. Let A and B be two input tables of tuples. We assume that tuples in A and B follow the same representation format (e.g., the same schema in the case of structured data). *Given A and B, our goal is to design a blocker Q: $A \times B \mapsto C$ such that the candidate set C includes as many matches as possible, while keeping the size minimal.*

Types of EM Problems: In Section 4.2.1, we identify an EM problem space with three different EM settings, which are (1) structured EM, (2) textual EM, and (3) dirty EM. In this work, we also want to know if DL can be helpful for blocking on all three EM settings. For the three EM problem types, we will experimentally compare different DL-based blocking solutions, and also compare those with state-of-the-art non-DL solutions.

Related EM Work Using DL: Recent work [40, 107] focuses on applying DL to EM. [40] proposes a DL solution considering both blocking and matching. For blocking, it devises an LSH-based method that applies to the embedding of a tuple. For matching, it proposes two DL models that build upon neural network (NN) architectures used extensively in the NLP literature. Different

from [40], [107], which is the project described in Chapter 4, provides a much deeper analysis of the matching step by exploring a DL design space, and proposes four DL models for matching with different complexity and representation power. For both works, their main focus is on the matching step (while a blocking method is proposed in [40] , the usage of DL is very limited). In this work, however, we focus on exploring DL for blocking.

5.2.2 Deep Learning

In Section 2.4 we reviewed the basic DL concepts, with some that are necessary for DL-based blocking including neural networks, recurrent neural networks, word embeddings, etc. In what follows, we briefly introduce the following learning methods and models that are important for the DL solutions described in Section 5.4.

Self-supervised Learning: Self-supervised learning, or self training, unsupervised representation learning, is the task of learning feature representations from unlabeled data [4, 33]. A general paradigm for this task is that given a set of items without human annotations, we need to devise an unsupervised learning algorithm to generate a good representation for each item (e.g., a vector) with respect to some objective. Such objectives can be generating universal representations capturing syntactic and semantic information (e.g., word embeddings if each item is a word), or generating useful representations for downstream tasks (e.g., summarizing a tuple vector for blocking if each input item is a tuple in our case), etc. In practice, self-supervised learning will be applied often under two scenarios. First, only unlabeled data is available at hand and it's hard to get labeled data that will be sufficient for a supervised learning algorithm (especially for DL). Blocking is covered by this case, where we are given only two tables. The procedure of generating a labeled pair set with enough matches will be like solving the EM problem itself. The second scenario is that we have a small labeled dataset for a target task, and a huge amount of unlabeled data (which may or may not be directly related to the target task) is available. In this case taking advantage of the unsupervised data to learn features and later combing with labeled data can be beneficial for the target task learning.

Different types of self-supervised learning methods have been proposed. Traditional (non-DL) methods like PCA [87] have been used for dimensionality reduction and feature representation. For deep learning methods, autoencoders and pre-training based models [120, 121, 33] have been developed and widely used.

Antoencoders: A family of DL-based methods called *autoencoders* [4, 57] is well studied by the machine learning community, and used for feature generation. A general framework for autoencoders consists of two parts: an *encoder* and a *decoder*. Given an input item, the encoder will first encode the item to generate a hidden vector, and then the decoder will take the hidden vector to produce an output. The training goal of the autoencoders is to learn the encoder and decoder such that the output is an (approximate) reconstruction of the input (i.e., minimizing the distance of the output and input w.r.t. some measure). To generate the feature representation for an item, only the encoder will be used and the generated hidden vector will be the final feature representation. Many different model variants have been developed, such as denoising autoencoder, variational autoencoder, sparse encoding, etc. (see [57] for a book chapter on autoencoders).

Autoencoders are related to the traditional (non-DL) method PCA. PCA performs a linear orthogonal transformation over a set of data instances to generate a set of linearly uncorrelated vectors. It has been shown that when the decoder is linear and the training loss is the mean squared error, an undercomplete autoencoder (with the size of generated hidden vector smaller than the size of the input vector) learns to span the same linear subspace as PCA [57]. When encoders and decoders are nonlinear, autoencoders learn a more powerful nonlinear generalization of PCA. In Section 5.6.5, we compare autoencoders with PCA to see how these two methods perform on blocking.

Model Pre-training in NLP: Recently, the idea of model pre-training becomes popular and has gained much attention in NLP. Given a large amount of unlabeled data, model pre-training aims at discovering useful knowledge and learning to generate good representations that will be beneficial for a target task. It basically follows a two step learning scheme. Suppose a target downstream task is known with a limited set of labeled data. First, a surrogate learning task will be

defined on the unlabeled data to get a pre-trained model. Then in the second step, the pre-trained model will bootstrap the supervised learning on the target task with labeled data. The core part of pre-training lies in the design of the surrogate learning task. Typically it focuses on exploring the internal structures of the unlabeled data, and thus may not be directly related to the target task. However, it has been shown that the knowledge learned via surrogate training followed by a supervised fine-tune learning can benefit various tasks in NLP. There are many different model pre-training approaches, such as ELMo [120], OpenAI GPT [121], BERT [33], etc. Specifically, BERT achieves the state-of-the-art performance on a series of NLP tasks. The model pre-training paradigm does not fit blocking in this project, as we do not have the supervised data for the model fine-tuning. However, it inspires the auxiliary training method proposed in Section 5.3.

5.3 A Design Space of DL Solutions for Blocking

In this section we describe a design space using deep learning for blocking. We begin by devising a DL solution template. This template consists of three modules: word embedding, tuple embedding, and vector-based pairing. Then for each of them we explore a set of choices. Different combinations of the module choices form a design space for blocking. As there can be many different module combinations, it's very expensive to examine all. In this work, we select five representative solutions from the design space to focus on: SIF, Autoencoder, Seq2seq, CTT, and Hybrid. All five solutions vary only in the tuple embedding module implementation, as it performs information summarization and is the most crucial module in the template. We propose five different tuple embedding models, with one for each solution. These models will be discussed in the next section. As for the other two modules, for the word embedding module, we use fastText as the implementation which has been discussed and used in Chapter 4. For the vector-based pairing module, we use top-k based Cosine similarity, which will be discussed in Section 5.5.

5.3.1 A DL Solution Template for Blocking

We first describe the DL model template for blocking, which is shown in Figure 5.1. As there is very little existing work using DL for blocking, we summarize the model template mainly

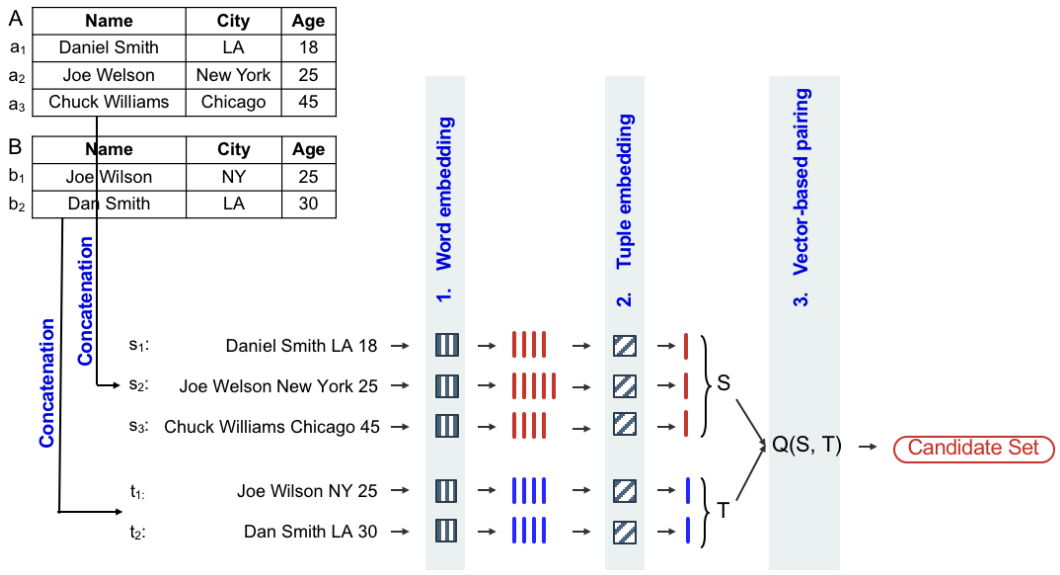

Figure 5.1: The DL model template for blocking.

by abstracting the blocker proposed in [40]. The proposed template consists of a concatenation operation followed by three main modules: word embedding, tuple embedding, and vector-based pairing.

The Concatenation Operation: As the first step of the solution template, we apply a concatenation operation to each tuple in the input two tables A and B, to convert the tuple to a string. Assume there are k attributes in the schema, we take a tuple a_i in A as an example to show how the concatenation operation works. Given the tuple a_i with the k attribute values $a_{i,1}, a_{i,2}, \ldots, a_{i,k}$, in the first step, we concatenate the attribute values of a_i to get a string $\hat{w}_{a_i} = a_{i,1} + a_{i,2} + \cdots + a_{i,k}$, where "+" indicates string concatenation with a white space delimiter. Then in the next step, we preprocess the string $\hat{w}_{a_i}$ by lower-casing each character, which gives us the final string w_{a_i} that will be used as the input to the word embedding module in the template. Denote A' contain the concatenated strings for tuples in A. We repeat the above procedure for every tuple b_j in B to get the string w_{b_j}, and denote B' contain the concatenated strings for B. In this project, we do not consider any weighting over different attributes in the schema for the string concatenation, and we leave it as future work.

There are reasons for us to perform this concatenation. First, if we handle each attribute separately, we have to increase the DL model capacity as well. And if there are many attributes in the schema (e.g., tens or even hundreds), it would be time consuming to train the model. More importantly, as there is no supervision for blocking, it's not clear how to aggregate the information from each attribute even if we process each independently. Instead, we can merge them together and let the model learn to identify the most important features.

Now we discuss the three main modules of the template.

The Word Embedding Module: Given a word sequence w_{a_i} in A' (or w_{b_j} in B'), this module takes w_{a_i} (or w_{b_j}) and produces a sequence of word embedding vectors $\mathbf{e}_{a_i}$ (or $\mathbf{e}_{b_j}$). Each element in $\mathbf{e}_{a_i}$ (or $\mathbf{e}_{b_j}$) is a d_e-dimensional embedding vector one-on-one mapping to the corresponding word in the sequence w_{a_i} (or w_{b_j}). Specifically, if the sequence w_{a_i} contains m words, the output $\mathbf{e}_{a_i}$ will be a matrix of size $d_e \times m$, and similar for w_{b_j}.

The Tuple Embedding Module: Given the word embedding sequences $\mathbf{e}_{a_i}$ for each w_{a_i} in A and $\mathbf{e}_{b_j}$ for each w_{b_j} in B, this module will take as input a word embedding sequence $\mathbf{e}_{a_i}$ or $\mathbf{e}_{b_j}$ to generate a d_u-dimensional summarization vector $\mathbf{u}_{a_i}$ or $\mathbf{u}_{b_j}$. The generated summarization vector $\mathbf{u}_{a_i}$ is called the *tuple embedding* of the corresponding tuple a_i, and similar for $\mathbf{u}_{b_j}$ and b_j. All tuple embeddings for tuples in A form a new table, denoted as S. Similarly, tuple embeddings for B are denoted as T.

The Vector-based Pairing Module: After the previous module, each tuple in A and B is associated with a tuple embedding vector. This module will perform pairing over the tuple embeddings to rule out all pairs not similar with respect to their embeddings to reduce the number of pairs considered for matching. Specifically, given the tuple embedding tables S and T from the previous module, this module takes as input S and T, and perform vector-based pairing with a function $Q(S, T)$, to produce a candidate set C.

A Design Space of DL Solutions for Blocking: For each module in the template there is a set of options. Figure 5.2 shows different options for each module, which form a design space on DL for blocking. Note that the options for the word embedding module is identical to what have been

Template module	**Options**	
Word embedding	*Granularity:* (1) Word-based (2) Character-based	*Training:* (3) Pre-trained (4) Learned
Tuple embedding	(1) Aggregation-based a) Simple Average b) Weighted Average • SIF (2) Self-supervised a) Self-reproduce • Autoencoder • Seq2seq b) Cross-tuple training c) Hybrid	
Vector-based pairing	(1) Hash • LSH (2) Similarity-based a) Similarity measure • Cosine • Euclidean b) Criteria • Threshold • kNN (3) Composite	

Figure 5.2: The design space of DL solutions for blocking.

used in Section 4.3.2, so we simply skip the description on the word embedding module here and refer to Section 4.3.2 for more details. We discuss the choices for the tuple embedding in Section 5.3.2 and vector-based pairing modules in Section 5.3.3 respectively.

From the proposed template, we formalize DL blocking as a similarity search problem. Specifically, we build DL models to learn to make the tuple embeddings of matching tuples close to one another in the embedding space. Therefore, the blocking procedure can be viewed as a coarse matching step without supervision. There are other approaches to formalize the blocking step. For example, [55] developed a rule-based blocker with active learning, which focuses on removing non-matching tuple pairs. We consider exploring different problem formalizations for DL blocking as future work.

5.3.2 Choices for Tuple Embedding

Given the embedding sequence $\mathbf{e}_t \in \mathbb{R}^{d_e \times m}$ for a tuple t (in A or B) with m words in the concatenated string w_t, this module generates a tuple embedding vector $\mathbf{u}_t \in \mathbb{R}^{d_u}$ that summarizes

the semantic information in t. In this project we select two major categories of options for tuple embedding.

(1) Aggregation-based Methods: Similar to the one used for attribute summarization in Section 4.3.3, in aggregation-based methods, tuple embeddings are generated by an aggregation function F over an input embedding sequence. Concretely, given an input embedding sequence, we apply the aggregation function $F : \mathbb{R}^{d_e \times \cdot} \mapsto \mathbb{R}^{d_e}$ to get a d_e-dimensional embedding vector, with an aggregation operation like averaging or weighted averaging. Since there is no learning engaged in aggregation-based methods, the quality of the generated tuple embeddings are largely affected by the quality of the word embeddings used in the previous module [2]. Conversely, the advantage of this type of methods is efficiency as there is no training occurred.

(2) Self-supervised Methods: The self-supervised methods aim at generating embedding vectors for tuples with help of a self-supervised learning procedure on the input tables themselves. And the basic procedure consists of two major stages: *model implementation* and *tuple embedding generation*.

The model implementation stage can be decomposed into three steps. First, a self-supervised learning task is defined on the data we have, which are the two tables for blocking in our case. This task may not be directly related to our goal which is to generate tuple embeddings, but the learning on the task ideally should discover useful knowledge of the input tables that can be beneficial for the goal. Second, a set of training data is generated according to the definition of the self-supervised learning task, only from the two tables A and B. This step is optional with different task definitions. Third, a deep learning based model will be devised to train on the training data. Remember our goal is to generate an embedding vector for a tuple. In order to achieve this, we force the proposed DL model to contain a "summarization module" which outputs a vector that can be used as the tuple embedding.

In the tuple embedding generation stage, we simply take the summarization module of the trained DL model, and use it for generating tuple embeddings.

From this procedure, we can see that the tuple embeddings are the secondary products of the self-supervised learning. Typically, good tuple embeddings can be learned if the self-supervised learning task is well defined, but in practice it is often hard to know and usually they are evaluated empirically. In this work we provide three subcategories of options for self-supervised methods.

- **Self-reproduce based methods**. Given the embedding sequence $\mathbf{e}_t$ for a tuple t in A or B, the self-reproduce based methods generate the tuple embedding with help of a self-reproduce learning task.

  In the model implementation stage, given the embedding sequence $\mathbf{e}_t$ and the corresponding word sequence w_t as input for a tuple t, this type of methods first define a self-supervised learning task that aims at generating an output which will reproduce the information in the input. With different task definitions, the actual format of the input and the information we want to recover may vary. See Section 5.4.2 and 5.4.3 for two self-reproduce variants. Since the task is to perform tuple information recovery, and we already have two tables of tuples A and B at hand, there is no data generation step in the training stage. Next, a DL model is devised for this type of methods, which consists of two major modules: an encoder and a decoder. Given an input ($\mathbf{e}_t$, w_t), the encoder summarizes the information in $\mathbf{e}_t$ to generate a hidden vector $\mathbf{u}_t$. Then the decoder receives $\mathbf{u}_t$ as input and try to recover the information in either $\mathbf{e}_t$ (see Section 5.4.2) or w_t (see Section 5.4.3) depending on the actual self-reproduce task definition.

  In the tuple embedding generation stage, given the embedding sequence $\mathbf{e}_t$ for a tuple t, we pass $\mathbf{e}_t$ to the encoder. The generated hidden vector $\mathbf{u}_t$ will be used as the tuple embedding. We can see that the inputs in this stage are only the embedding sequences from the word embedding module, and this is different from the model implementation stage where extra information (e.g., the word sequences) is needed to be able to conduct the self-supervised learning.

  With different task definition, various types of models have been developed, such as autoencoders [4, 57], sequence-to-sequence learning [145], etc.. The basic advantage of this type

of models is that the training data is easy to get, and the model is pretty straight-forward to learn with a clear goal. However, as it generates tuple embedding by only exploring information within the tuple, the limitation is that it may not summarize the most useful information without comparing against other tuples.

- **Auxiliary training based methods**. Given the embedding sequence $\mathbf{e}_t$ for a tuple t in A or B, this type of methods generate the tuple embedding with the help of an auxiliary learning task. Similar to the self-reproduce based methods, the tuple embedding for each embedding sequence will be generated as the side product of the auxiliary learning. However, different from the self-produce based ones which focus on recovering the input, there is no fixed format on how the auxiliary training task should be defined, and it is relied on the domain expertise of the task designer. Practically we want to design a task that: (i) It's easy and natural to understand the task goal; (ii) It learns useful knowledge of the tuples in A and B, and this knowledge should be beneficial for tuple embedding generation, which is our real target; (iii) It should be easy to generate the training data for the task with no or little feature engineering; (iv) A DL model can be designed for the task and learned efficiently. And the model can be used to generate tuple embeddings (as the side products).

  In this work, we focus on one type of auxiliary training tasks, *cross-tuple training*, to capture cross-tuple information that cannot be learned with the self-reproduce based methods. In Section 5.4.4 we describe one task realization. Typically, compared to the self-reproduce methods, auxiliary training is more flexible as there is no hard constraint on the task format. Conversely, the down side is that it relies on domain expertise to design a good task that can be helpful for tuple embedding generation.

- **Hybrid methods**. The Hybrid methods are basically a combination of the self-reproduce and cross-tuple training methods, to generate tuple embeddings that capture both in-tuple and cross-tuple information. This type of methods is the most powerful one in terms of the model capacity and expressiveness power. However, it will take longer time to train.

5.3.3 Choices for Vector-based Pairing

Given two tuple embedding tables S and T for A and B respectively, the goal of this module is to apply a vector-based pairing function Q to generate a candidate set $C \subseteq A \times B$. For this module, we identify three major options by adapting all possible existing non-DL blocking types summarized in Section 2.1. Based on the description in Section 2.1, we can categorize non-DL blocking methods in to five different types: hashing-based (e.g., AE, hash), sorting-based (e.g.,sorted neighborhood), similarity-based, composite (e.g. canopy clustering, rule-based), and schema-agnostic blocking. Among these five different categories, we provide no option for sorting-based and schema-agnostic blocking as they are not applicable in our setting, which is to pair high-dimensional vectors. We summarize the options for the rest three types below.

(1) Hashing-based Pairing: Hashing-based pairing generates candidate tuple pairs with a hashing procedure. It typically consists of the following steps: First, a hash function is devised. Second, for each tuple embedding vector, we apply the hash function to generate a hash key. Each unique hash key will be associated with a hash bucket, and all tuples with the same key will be placed into the same bucket. Finally, we go over each bucket and add tuple pairs belonging to $A \times B$ to the candidate set. As the objects for hashing are vectors (and often in high dimensional space) in our case, a good design of hashing function should make similar vectors with respect to some measure placed into the same bucket. A well-known family of hashing functions over vectors is Locality Sensitive Hashing (LSH), which hashes similar items into the same bucket with high probability. Many different LSH variants have been developed (see [160] for a survey). Specifically, [40] uses cosine-based LSH as the hashing realization for blocking.

(2) Similarity-based Pairing: Similarity-based pairing generates candidate tuple pairs by directly comparing the tuple embedding similarity with respect to some measure. Typically a similarity-based pairing function consists of two parts: a similarity measure and a filtering condition. A similarity measure takes as input a pair of tuple embedding vectors and generates a similarity score. Commonly used measures include cosine similarity, Euclidean distance, etc.. Once we have the similarity scores for all tuple pairs, a filtering condition will be applied to filter out pairs

that are unlikely to match. All pairs surviving the condition form the candidates set. For the filtering condition, the simplest choice is to set a threshold. For example, suppose we have a pairing function using the cosine similarity with a threshold of 0.5, then all pairs with a cosine score larger than 0.5 will be included in the candidate set. Another filtering strategy is k-Nearest-Neighbor (kNN) selection. For example, suppose the cosine similarity is used, for each tuple $a_i \in A$ with the tuple embedding $\mathbf{u}_{a_i}$, we first calculate the cosine scores between $\mathbf{u}_{a_i}$ and every $\mathbf{u} \in T$. Then, we pick k tuples $B_{a_i} \subseteq B$ whose corresponding embeddings vectors $T_{a_i} \subseteq T$ have the highest k cosine similarity scores. Those k pairs (a_i, b_j) with $b_j \in B_{a_i}$ will be included in the candidates set.

Theoretically, similarity-based pairing requires a cross-product comparison between A and B, because for each tuple in A we need to compare against all tuples in B in order to get the candidate set. This can be expensive if the tables are large in size. However, as we are comparing vectors all in the same dimensional space, we can parallelize the procedure and speed up the computation significantly with matrix operations and GPU acceleration.

(3) Composite Pairing: Composite pairing generates candidate tuple pairs by applying a series of blocking steps. For example, a candidate set can be obtained by first applying LSH to get a set of hash buckets, then selecting k pairs with the highest cosine similarity scores in each bucket.

5.4 Implementations for Tuple Embedding

The previous section describes a DL solution space for blocking, which consists three modules: word embedding, tuple embedding, and vector-based pairing. As the design space provides many different module combinations and it is unrealistic for us to examine all, we choose five representative solutions out of the space by only varying the tuple embedding module. In this section we focus on the tuple embedding module implementations, where we build five tuple embedding models with one for a different solution. As all solutions only vary in the tuple embedding implementation, we interchangeably use the name of each solution to represent the corresponding tuple embedding model. Hence, the five tuple embedding models are SIF, Autoencoder, Seq2seq,

CTT, and Hybrid. Among them, SIF is aggregation-based, and the rest four are self-supervised models. Specifically, Autoencoder and Seq2seq belong to the category of self-reproduce, CTT is a cross-tuple training model, and Hybrid is a combination of Autoencoder and CTT. These methods vary in complexity and representational power. Below we describe each of them.

5.4.1 SIF: An Aggregation-based Model

Given the word embedding sequence of a tuple, SIF generates a tuple embedding by a weighted average aggregation procedure using the SIF model [2], which consists of three steps. First, for each tuple, we calculate a weighted average over the word embeddings. Specifically, the weights used to calculate the average over the word embeddings are as follows: given a word w in the tuple string, the corresponding word embedding is weighted by a weight $f(w) = a/(a + p(w))$ where a is a hyperparameter and $p(w)$ the normalized unigram frequency of w in the input two tables. Once we get the weighted average aggregation vectors for all tuples, in the second step we calculate the first principal component of the aggregation vectors using PCA. Finally, we calculate the final tuple embedding for each tuple by subtracting the projection of its aggregation vector over the first principal component. Specifically, denote $\mathbf{a}_t$ be the aggregation vector for the tuple t, and $\mathbf{u}$ be the first principal component, the tuple embedding $\mathbf{u}_t = \mathbf{a}_t - \mathbf{u}\mathbf{u}^\top \mathbf{a}_t$.

This model was used in Chapter 4 for DL exploration on matching. In this project we use it again, as it is a simple (i.e., without learning incurred) but effective baseline (e.g., it was shown to perform comparably to complex models in some NLP tasks such as text similarity) deep learning model. Its performance relies mostly on the expressiveness of the word embeddings used.

5.4.2 Autoencoder: A Self-reproduce Approach

In this part, we describe Autoencoder, a self-reproduce approach, to generate tuple embeddings. Below we first describe the key idea, then talk about model implementation and tuple generation.

Key Idea: Given the word embedding sequence $\mathbf{e}_t$ for a tuple t, we want to generate the tuple embedding $\mathbf{u}_t$ such that $\mathbf{u}_t$ summarizes the key information in $\mathbf{e}_t$. This is analogous to a similar

problem that given a paper, we want to write an abstract such that the abstract summarizes the paper. Conceptually, we prefer to have an abstract which contains the most essential information of the paper such that by only taking the abstract, we can use it to roughly recreate the whole paper. Adapting this idea to tuple embedding generation, similarly we can say that $\mathbf{u}_t$ is good if it summarizes the most important information in $\mathbf{e}_t$ such that we can use $\mathbf{u}_t$ to recreate the information in $\mathbf{e}_t$. Autoencoders aims at implementing this idea. Specifically, we want to find two neural networks NN_1 and NN_2 such that we use NN_1 to summarize $\mathbf{e}_t$ to generate the tuple embedding $\mathbf{u}_t$, and use NN_2 to recover the information in $\mathbf{e}_t$ from $\mathbf{u}_t$. Denote NN_1 as an encoder and NN_2 as a decoder, combining the two we have the basic format of an autoencoder model.

The idea of using autoencoders for summarization generation is not new. For example, it has been used in dimensionality reduction for data visualization, image denoising, etc. [57]. However, to our knowledge, no one has ever applied autoencoders for blocking.

Model Implementation: In this project, we choose to use a two-layer feed-forward NN for both the encoder and the decoder implementation. Below we give a formal description of the model implementation.

(1) Task Definition: Given the word embedding sequence $\mathbf{e}_t$ for every tuple t in A or B, we want to build an autoencoder model, which uses $\mathbf{e}_t$ as the model input, and generates an output vector $\mathbf{o}_t$ that recovers the information in $\mathbf{e}_t$. Ideally, we want the model to reproduce $\mathbf{e}_t$ itself which is the input to the model. However, as we use a feed-forward NN for the encoder and it cannot take word embedding sequences of variable lengths as input, we modify the classic autoencoder model by performing an aggregation operation $f(\cdot)$ in the first step to convert $\mathbf{e}_t$ into a fixed-size vector $\mathbf{v}_t = f(\mathbf{e}_t)$. Then we use $\mathbf{v}_t$ as the input to the encoder, and the decoder will reproduce $\mathbf{v}_t$ as the output vector.

(2) Data Generation: The task goal is to build an autoencoder to recover the information of an input tuple, and we already have the tuples in two tables A and B, so there is no need to do extra data generation for the training task.

(3) Model Building: In this project, we use two-layer feed-forward NNs with the Tanh activation function for both the encoder and decoder. The autoencoder model built in the project consists

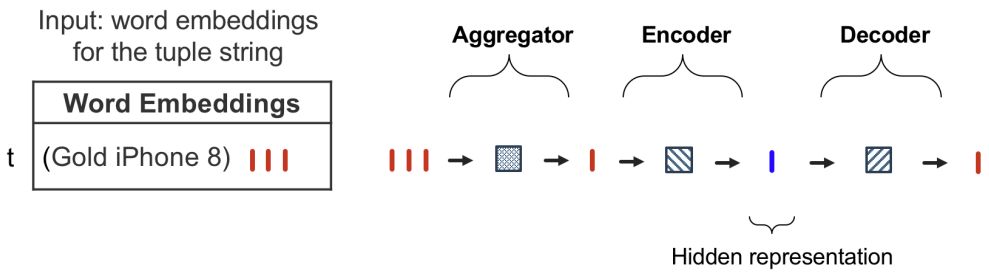

Figure 5.3: The autoencoder model architecture.

of three parts: an aggregator, an encoder and a decoder. Figure 5.3 shows the model architecture. Given a word embedding sequence $\mathbf{e}_t$ as the input, we first apply $\mathbf{e}_t \in \mathbb{R}^{d_e \times \cdot}$ to the aggregator to generate the aggregation vector $\mathbf{v}_t \in \mathbb{R}^{d_e}$. For the aggregation implementation, we use the SIF model (see Section 4.4.1) which calculates a weighted average over the word embeddings in $\mathbf{e}_t$. Next, the encoder receives $\mathbf{v}_t$ as the input and generates a hidden representation vector $\mathbf{u}_t \in \mathbb{R}^{d_u}$. Then the decoder uses $\mathbf{u}_t$ to produce the output $\mathbf{o}_t \in \mathbb{R}^{d_e}$ which tries to copy $\mathbf{v}_t$.

For the model training objective, we define the training loss on the tuple t as $l_t = \|\mathbf{v}_t - \mathbf{o}_t\|_2^2$, which is the squared L_2 distance between the aggregation vector and the output vector. The training goal is to update the parameters in the encoder and the decoder such that the training loss is minimized.

Tuple Embedding Generation: Once the training is done, to generate the tuple embedding for a given tuple t, we feed the word embedding sequence $\mathbf{e}_t$ of t to the aggregator followed by the encoder, and use the generated hidden representation vector $\mathbf{u}_t$ as the tuple embedding vector for t.

5.4.3 Seq2seq: A Sequence-aware Self-reproduce Approach

Seq2seq, or the sequence-to-sequence model, can be viewed as a sequence-based autoencoder model. Autoencoder described in Section 5.4.2 tries to recover an aggregation of the input word embedding sequence, which does not explore the sequence information. So in this model we aim at generating tuple embeddings that take into account the sequence information.

Model Implementation: We first describe the model implementation.

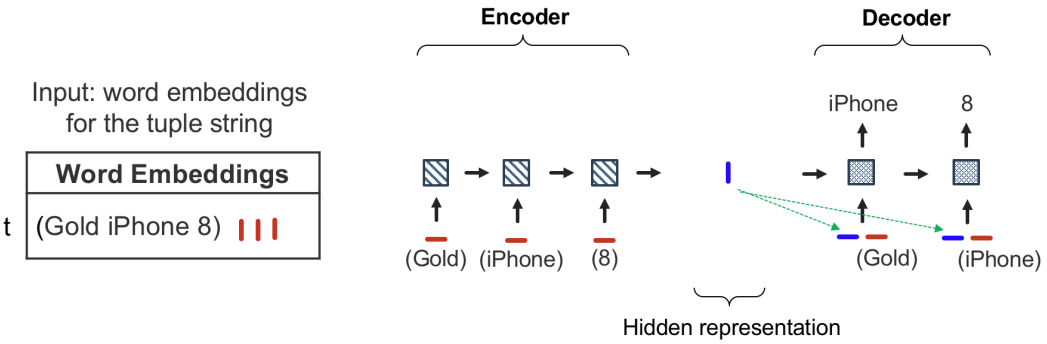

Figure 5.4: The seq2seq model architecture.

(1) Task Definition: Given an embedding sequence $\mathbf{e}_t$ and the corresponding word sequence w_t for a tuple t in A or B, the task here is to build a model that receives $\mathbf{e}_t$ as the input to reproduce the word sequence w_t.

(2) Data Generation: Similar to Autoencoder, as the task goal is to recover the word sequence of a tuple, there is no need to do extra data generation.

(3) Model Building: Seq2seq consists of two parts: the encoder and the decoder. To be able to handle sequences with variable length, we choose to use LSTM-RNNs [66] for both the encoder and decoder. LSTMs are one of the most popular RNNs used in the NLP community.

Figure 5.4 shows the model architecture. For the encoder, an LSTM is used to take each embedding vector in $\mathbf{e}_t$ one-by-one to get a hidden representation vector $\mathbf{u}_t$. Formally, given the i-th embedding $\mathbf{e}_t[i]$ in $\mathbf{e}_t$, the LSTM unit $LSTM_{enc}$ takes $\mathbf{e}_t[i]$ and the hidden state $\mathbf{h}_{enc}[i-1]$ from the previous step as input to produce a new hidden state $\mathbf{h}_{enc}[i] = LSTM_{enc}(\mathbf{e}_t[i], \mathbf{h}_{enc}[i-1])$. We denote the last hidden state output as $\mathbf{u}_t$, which will be used as the context for the decoder to recover the word sequence. For the first hidden state input $\mathbf{h}_{enc}[0]$ we randomly initialize the vector.

For the decoder, we use a combination of an LSTM $LSTM_{dec}$ and a one-layer feed-forward NN Out to recover the word sequence w_t. It works as follows. First, the hidden state $\mathbf{u}_t$ will be used as the initial state $\mathbf{h}_{dec}[0]$ for $LSTM_{dec}$, which serves as the context for the sequence decoding. Second, we pass the i-th word embedding $\mathbf{e}_t[i]$, the context $\mathbf{u}_t$, and the previous hidden state $\mathbf{h}_{dec}[i-1]$ to the decoder and try to predict the next word in the sequence $w_t[i+1]$. This can be further decomposed into two substeps. First, we pass the three vectors $\mathbf{e}_t[i]$, $\mathbf{u}_t$, and $\mathbf{h}_{dec}[i-1]$

to the decoder LSTM to get the hidden state $\mathbf{h}_{dec}[i] = LSTM_{dec}(\mathbf{e}_t[i], \mathbf{u}_t, \mathbf{h}_{dec}[i-1])$. Once we have the hidden state $\mathbf{h}_{dec}[i]$, then we send it through the feed-forward NN Out, to output a score vector $\mathbf{o}_i = Out(\mathbf{h}_{dec}[i]) \in \mathbb{R}^{|V|}$ over entire the word vocabulary V. The j-th index $\mathbf{o}_i[j]$ is a score indicating how likely the next word in the sequence will be j-th word in the vocabulary V. As we want to predict the next word $w_t[i+1]$ in the sequence, suppose $w_t[i+1]$ is the k-th word in the vocabulary, we want to have $\mathbf{o}_i[k]$ to be the largest value in $\mathbf{o}_i$.

For the model training objective, we use the cross-entropy loss for each position i, to maximize the value in $\mathbf{o}_i$ whose index corresponds to the index of the word $\mathbf{w}_t[i+1]$ in the vocabulary.

Note that in the decoding procedure, besides using $\mathbf{u}_t$ as the context to the LSTM, we also use it as a part of the input for each decoding step, as shown in Figure 5.4. The reason is that we want to learn $\mathbf{u}_t$ such that it will affect the recovery of each word in the sequence. This will reduce the importance of the LSTM in decoding and force $\mathbf{u}_t$ to summarize the information in $\mathbf{e}_t$ well to be able to recover the word sequence. And this is exactly what we want because we use $\mathbf{u}_t$ to be the tuple embedding for t shown below.

Tuple Embedding Generation: Given the embedding sequence $\mathbf{e}_t$ for a tuple t, we apply $\mathbf{e}_t$ to the LSTM encoder $LSTM_{enc}$, and the last hidden state output $\mathbf{u}_t$ will be used as the tuple embedding.

5.4.4 CTT: A Cross-tuple Training Approach

In Section 5.4.2 and 5.4.3 we show two approaches for tuple embedding generation with self-reproduce training. A problem is that they generate the tuple embedding by only looking into the tuple itself. This way of tuple embedding generation may not work optimally without comparing against other tuples, as it is hard to know which part in a tuple would be most important. So in this section we define a cross-tuple training task to compare tuple pairs, and generate tuple embeddings that capture cross-tuple information. We first describe the key idea of this approach.

Key Idea: Given two tables A and B, we want to generate a tuple embedding vector for every tuple in A and B. Intuitively, if a tuple $a \in A$ and a tuple $x \in B$ is a match (i.e., referring to the same real-world entity), we want the generated tuple embeddings $\mathbf{u}_a$ and $\mathbf{u}_x$ to be close in the

vector space. On the other hand, if $a \in A$ does not match a tuple $y \in B$, we want the generated tuple embeddings $\mathbf{u}_a$ and $\mathbf{u}_y$ to be far. Given this, we can propose the following ideal model implementation procedure.

Ideal Model Implementation: The ideal model implementation consists of three parts.

(1) Task Definition: For this cross-tuple training approach, we define a surrogate matching task. Specifically, given a set of tuple pairs $S \subseteq A \times B$ where each pair in S has been labeled as a match or non-match, we want to build CTT (representing cross-tuple training) training on S, that can successfully learn to predict each pair in S to the correct label. During this procedure, knowledge that is important for correct matching predictions from tuple comparisons will be embedded into CTT, such that the tuple embeddings generated by (part of) the model will capture cross-tuple information.

According to the task definition, we need to get a labeled set S in order to train CTT. Below we describe an ideal data generation procedure to get a set of ideal labeled pairs. This dataset contains all matching tuple pairs across A and B as well as a set of non-matching pairs, to allow the model to learn good cross-tuple information. However. this ideal dataset is very difficult to get in practice (as it requires to know all matching pairs in advance). Later in this section we propose a procedure to approximate this ideal training dataset.

(2) Ideal Data Generation: The ideal training data consists of two parts: positive training instances which are a set of matching pairs, and negative training instances which are non-matches.

To generate the positive training data, suppose the set $M \subseteq A \times B$ contain all matching pairs, we simply take all pairs in M as the positive training instances.

To generate the negative training data, we randomly select a set of pairs from $A \times B$ that are not in M as the negative training instances. Specifically, we first union the two tables A and B to get a table E. Then, for each tuple $t \in E$, we randomly select a set $S_t \subseteq E$ of p tuples (where p is a hyperparameter), to form a set of non-matching pairs $N_t = \{(t, s) \mid s \in S_t\}$ satisfying each pair $(t, s) \notin M$. We repeat this procedure for every tuple $t \in E$, and finally we take the union $\cup_{t \in E} N_t$ as the negative training data.

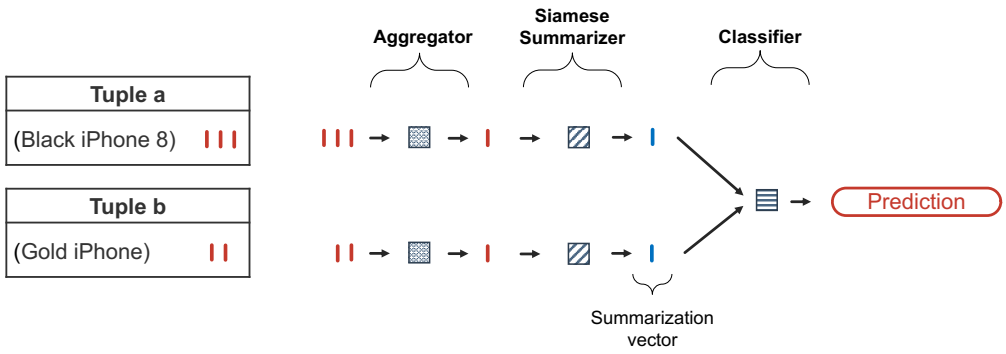

Figure 5.5: The cross-tuple training model architecture.

(3) Model Building: We use a siamese feed-forward neural network as the model implementation for CTT. Figure 5.5 shows the model architecture. This model consists of three modules: an aggregator, a siamese summarizer, and a classifier. Given a pair of word embedding sequences $\mathbf{e}_1$ and $\mathbf{e}_2$ from the training data as input, we first apply the aggregator to convert each embedding sequence into a fix-size vector respectively, denoted as $\mathbf{v}_1 \in \mathbb{R}^{d_e}$ and $\mathbf{v}_2 \in \mathbb{R}^{d_e}$. Here again we use the SIF model for the embedding sequence aggregation. Next, for each of $\mathbf{v}_1$ and $\mathbf{v}_2$, a siamese summarizer, which is a two-layer feed-forward neural network, is used to generate a summarized vector $\mathbf{u}_1 \in \mathbb{R}^{d_u}$ and $\mathbf{u}_2 \in \mathbb{R}^{d_u}$, respectively. Then, we take an element-wise absolute difference between $\mathbf{u}_1$ and $\mathbf{u}_2$, and send it to the classifier, which is a two-layer feed-forward neural network, to predict a label indicating the input pair a match or non-match. The training goal is to learn the model parameters in the siamese summarizer and the classifier such that the predictions are the same as the gold labels in the training data.

nNotice that we use a siamese network for summarizing the two input tuples. By doing this, we reduce the capacity of the model (i.e., without using two separate summarizers for each of the tuple in a input pair) such that the learned knowledge does not diffuse too much into the model parameters. This makes the summarizer generate better summarization vectors.

Tuple Embedding Generation: Given the embedding sequence $\mathbf{e}_t$ for tuple t in A or B, we apply the aggregator followed by the siamese summarizer in the trained CTT model. The generated summarization vector $\mathbf{u}_t$ is used as the tuple embedding.

As the training data already contains the perfect information on which pairs are match and which pairs are not, the generated tuple embeddings are expected to capture cross-tuple information such that the embeddings for matching tuples are close while the embeddings for non-matching tuples are far in the vector space.

Approximating the Ideal Training Data: There is one problem with the model implementation described above. From the data generation procedure, in order to get the positive training instances, we need to know all matching pairs in $A \times B$ in advance. By doing itself we already solve the EM problem! This means in practice it is very difficult to get the ideal training data. To address this, we propose a data generation procedure to approximate the ideal training data as follows.

First, we union the two tables A and B to get a table E. Then for each tuple $t \in E$, we generate one positive training instance and p negative instances. Each training instance here contains three parts: a pair of tuple strings, and a label indicating if this pair is a match or not.

We first generate a positive instance for t, which is a matching pair containing t as one tuple. In the first step, we concatenate the attribute values of t to get the word sequence w_t, which will be used as the first tuple. Then, we need to find the second tuple in E which will be a match to t. However, at this point we do not know which tuple matches t. To solve this, we approximate a matching tuple string by randomly selecting 60% of words in w_t, denoted as w'_t. As we know w'_t is selected from w_t itself, we associate a label "1" for this pair (w_t, w'_t) indicating it is a match.

Then we generate the negative instances for t. To generate a negative instance, similarly in the first step, we concatenate the attribute values of t to get the string w_t. Then we need to find a non-match in E to t. To get this, we randomly select a tuple s in E (excluding t), concatenate the attribute values of s to get w_s, and use w_s as the second tuple. As the tuple s is randomly selected and very likely it will be a non-match to t (assuming matches are rare compared to non-matches), we associate a label "0" indicating a non-match for the pair (w_t, w_s). We repeat this procedure p times with p random tuple selections in E, which gives us p negative training instances.

Above is the procedure to generate training instances for one tuple. We repeat the procedure for every tuple in the tables A and B, and finally take the union of the training instances for every tuple as the approximated training data.

Once we have the approximated training data, we train a CTT model, and use the aggregator followed by the siamese summarizer to generate tuple embeddings. These steps are the same as what we described above.

5.4.5 Hybrid: Cross-tuple Training with Autoencoders

The last approach is Hybrid where we combine Autoencoder (Section 5.4.2) and CTT (Section 5.4.4) to generate tuple embeddings considering both in-tuple and cross-tuple information. Here we do not combine Seq2seq (Section 5.4.3) and CTT as empirically Seq2seq does not outperform Autoencoder (see the comparison in Section 5.6.2) while taking much longer time to train (see the comparison in Section 5.6.3). We discuss the model implementation and tuple embedding generation steps below.

Model Implementation: We first describe the model implementation.
(1) Task Definition: The task for building Hybrid consists of two sub-training tasks. In the first sub-task, we will train an autoencoder model on the two tables A and B according to the method in Section 5.4.2. Once the autoencoder training is done, in the second sub-task, we will first generate the cross-tuple training data according to the approximated data generation procedure in Section 5.4.4, and then we train Hybrid combining the trained autoencoder model and an untrained CTT model on the data.
(2) Data Generation: The first sub-task is to train the autoencoder model on the two tables A and B, therefore no need for extra data generation. To train the CTT model in the second sub-task, we need to perform the same approximated data generation procedure as described in Section 5.4.4.
(3) Model Building: We use a stacked training procedure to train Hybrid, which consists of two steps. In the first step, give the word embedding sequences of tuples in A and B, we train an autoencoder model M_1 end-to-end. Once the training on M_1 is done, we train a CTT model M_2 next. The training step for M_2 is the same as what we described in Section 5.4.4, except that we use a modified version of the original CTT model. The modification we made is that instead of using the SIF model, we take the (trained) encoder of M_1 as the aggregator for M_2.

Note that here M_1 and M_2 are stacked by training M_1 first then M_2, as opposed to training M_1 and M_2 jointly. The reason is that we want to keep the two models M_1 and M_2 separate to avoid cross-tuple information diffusing to the model parameters of M_1 (if we do the joint training), such that M_1 does not summarize the in-tuple information well.

Tuple Embedding Generation: Given the embedding sequence $\mathbf{e}_t$ for tuple t in A or B, we feed $\mathbf{e}_t$ to the aggregator (which is the encoder in M_1) followed by the siamese summarizer in M_2, and use the output as the tuple embedding vector.

5.5 Implementation for Vector-based Pairing

The previous section describes five implementations for the tuple embedding module. In this section we discuss the vector-based pairing implementation to generate the candidate set. In this project, we use the topk-based cosine similarity, a similarity-based pairing function, as the implementation for all five selected representative DL solutions. This pairing function is selected for two reasons. First, it has only one hyperparameter, which is k for the topk pair selection. This makes it easily tuned in practice, as the user does not need to enumerate different combinations of many hyperparameters. Second, it is easy for us to control the size of the generated candidate set by varying the value for k (where as for other options like threshold-based cosine similarity or LSH, it is very difficult for us to know the size of the generated candidate set without executing the function). For the rest of this section, we will first describe the way this pairing function generating the candidate set in Section 5.5.1, then we describe how to run the function efficiently in Section 5.5.2.

5.5.1 Topk-based Cosine Similarity

Given two tuple embedding tables S and T for tuples in A and B respectively, the topk-based cosine similarity function performs pairing as follows. First, for a tuple embedding $\mathbf{u}_{a_i}$ with the corresponding tuple a_i in A, we calculate the cosine similarity between $\mathbf{u}_{a_i}$ and every tuple embedding $\mathbf{u} \in T$. Then, we select a subset $T_{a_i} \in T$ which contains the k tuple embeddings with the

highest k cosine scores comparing against $\mathbf{u}_{a_i}$. That is, for any $\mathbf{u} \in T_{a_i}$ and $\mathbf{v} \in T \setminus T_{a_i}$, we have $cosine(\mathbf{u}_{a_i}, \mathbf{u}) \geq cosine(\mathbf{u}_{a_i}, \mathbf{v})$. Last, for each $\mathbf{u} \in T_{a_i}$, we find the corresponding tuple $b \in B$, and add the tuple pair (a_i, b) into the candidate set C.

There is one hyperparameter in this pairing function, which is k for the topk selection. With a different k value we can get a candidate set C of different size. In practice, a user has to pick a value for k in order to get a fixed C. Ideally, we prefer a k value such that the pairing function achieves a good balance of having high recall while keeping C small. However, at this blocking stage we do not know the true matches yet, therefore the recall cannot be evaluated, which makes it hard to know whether a selected value k is good or not. To address this, in practice a user can run MatchCatcher described in Chapter 3 to debug blocking. It will help the user understand the qualify of the candidate set given a value k and select a better k.

5.5.2 Efficient Execution with Matrix Operations and GPU acceleration

From the definition of the pairing function we can see that in order to get a candidate set, it requires a cross-product comparison between A and B. Theoretically this can be very expensive when A and B are large in size. In practice, as all generated tuple embedding vectors are in the same vector space, we can speed up the computation with matrix operations accelerated by GPUs. Below we describe the procedure, which consists of three steps: *matrix construction*, *cosine score calculation*, and *candidate set generation*.

Matrix Construction: Given the two sets of tuple embeddings S and T for tables A and B respectively, in this step we convert each of S and T to a matrix. Assume there are m tuples $a_1, a_2, \ldots, a_m$ in A and n tuples $b_1, b_2, \ldots, b_n$ in B. First, we construct the matrix $M_A \in \mathbb{R}^{m \times d_u}$, where the i-th row of M_A is the tuple embedding $\mathbf{u}_{a_i}$ in S for the tuple a_i. Then similarly we construct the matrix $M_B \in \mathbb{R}^{n \times d_u}$, where the j-th row of M_B is the tuple embedding $\mathbf{u}_{b_j}$ in T for

the tuple b_j. The generated the matrices are represented as follows:

$$M_A = \begin{bmatrix} \mathbf{u}_{a_1} \\ \mathbf{u}_{a_2} \\ \vdots \\ \mathbf{u}_{a_m} \end{bmatrix} \qquad M_B = \begin{bmatrix} \mathbf{u}_{b_1} \\ \mathbf{u}_{b_2} \\ \vdots \\ \mathbf{u}_{b_n} \end{bmatrix}$$

Cosine Score Calculation: Based on the two matrices M_A and M_B, we can compute the cosine similarity scores for all tuple pairs across A and B with matrix operations, which consists of the following steps.

First, we compute the matrix P, which is the product of M_A and the transpose of M_B:

$$P = M_A M_B^\top = \begin{bmatrix} (\mathbf{u}_{a_1} \cdot \mathbf{u}_{b_1}) & (\mathbf{u}_{a_1} \cdot \mathbf{u}_{b_2}) & \cdots & (\mathbf{u}_{a_1} \cdot \mathbf{u}_{b_n}) \\ (\mathbf{u}_{a_2} \cdot \mathbf{u}_{b_1}) & (\mathbf{u}_{a_2} \cdot \mathbf{u}_{b_2}) & \cdots & (\mathbf{u}_{a_2} \cdot \mathbf{u}_{b_n}) \\ \vdots & \vdots & \ddots & \vdots \\ (\mathbf{u}_{a_m} \cdot \mathbf{u}_{b_1}) & (\mathbf{u}_{a_m} \cdot \mathbf{u}_{b_2}) & \cdots & (\mathbf{u}_{a_m} \cdot \mathbf{u}_{b_n}) \end{bmatrix}$$

Each element P_{ij} in the matrix is the dot product of two tuple embedding vectors $\mathbf{u}_{a_i}$ and $\mathbf{u}_{b_j}$.

Next, we normalize the matrix P row-wise. Specifically, for the i-th row $P_{i\cdot}$, we normalize each element P_{ij} by the norm of $\mathbf{u}_{a_i}$, with j in the range of 1 to n. This gives us a matrix Q:

$$Q = \begin{bmatrix} \frac{P_{11}}{\|\mathbf{u}_{a_1}\|} & \frac{P_{12}}{\|\mathbf{u}_{a_1}\|} & \cdots & \frac{P_{1n}}{\|\mathbf{u}_{a_1}\|} \\ \frac{P_{21}}{\|\mathbf{u}_{a_2}\|} & \frac{P_{22}}{\|\mathbf{u}_{a_2}\|} & \cdots & \frac{P_{2n}}{\|\mathbf{u}_{a_2}\|} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{P_{m1}}{\|\mathbf{u}_{a_m}\|} & \frac{P_{m2}}{\|\mathbf{u}_{a_m}\|} & \cdots & \frac{P_{mn}}{\|\mathbf{u}_{a_m}\|} \end{bmatrix} = \begin{bmatrix} \frac{\mathbf{u}_{a_1} \cdot \mathbf{u}_{b_1}}{\|\mathbf{u}_{a_1}\|} & \frac{\mathbf{u}_{a_1} \cdot \mathbf{u}_{b_2}}{\|\mathbf{u}_{a_1}\|} & \cdots & \frac{\mathbf{u}_{a_1} \cdot \mathbf{u}_{b_n}}{\|\mathbf{u}_{a_1}\|} \\ \frac{\mathbf{u}_{a_2} \cdot \mathbf{u}_{b_1}}{\|\mathbf{u}_{a_2}\|} & \frac{\mathbf{u}_{a_2} \cdot \mathbf{u}_{b_2}}{\|\mathbf{u}_{a_2}\|} & \cdots & \frac{\mathbf{u}_{a_2} \cdot \mathbf{u}_{b_n}}{\|\mathbf{u}_{a_2}\|} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\mathbf{u}_{a_m} \cdot \mathbf{u}_{b_1}}{\|\mathbf{u}_{a_m}\|} & \frac{\mathbf{u}_{a_m} \cdot \mathbf{u}_{b_2}}{\|\mathbf{u}_{a_m}\|} & \cdots & \frac{\mathbf{u}_{a_m} \cdot \mathbf{u}_{b_n}}{\|\mathbf{u}_{a_m}\|} \end{bmatrix}$$

Finally, we normalize the matrix Q column-wise. Specifically, for the j-th row $Q_{\cdot j}$, we normalize each element Q_{ij} by the norm of $\mathbf{u}_{b_j}$, with i in the range of 1 to m. This gives us a matrix W:

$$W = \begin{bmatrix} \frac{Q_{11}}{\|\mathbf{u}_{b_1}\|} & \frac{Q_{12}}{\|\mathbf{u}_{b_2}\|} & \cdots & \frac{Q_{1n}}{\|\mathbf{u}_{b_n}\|} \\ \frac{Q_{21}}{\|\mathbf{u}_{b_1}\|} & \frac{Q_{22}}{\|\mathbf{u}_{b_2}\|} & \cdots & \frac{Q_{2n}}{\|\mathbf{u}_{b_n}\|} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{Q_{m1}}{\|\mathbf{u}_{b_1}\|} & \frac{Q_{m2}}{\|\mathbf{u}_{b_2}\|} & \cdots & \frac{Q_{mn}}{\|\mathbf{u}_{b_n}\|} \end{bmatrix} = \begin{bmatrix} \frac{\mathbf{u}_{a_1} \cdot \mathbf{u}_{b_1}}{\|\mathbf{u}_{a_1}\| \cdot \|\mathbf{u}_{b_1}\|} & \frac{\mathbf{u}_{a_1} \cdot \mathbf{u}_{b_2}}{\|\mathbf{u}_{a_1}\| \cdot \|\mathbf{u}_{b_2}\|} & \cdots & \frac{\mathbf{u}_{a_1} \cdot \mathbf{u}_{b_n}}{\|\mathbf{u}_{a_1}\| \cdot \|\mathbf{u}_{b_n}\|} \\ \frac{\mathbf{u}_{a_2} \cdot \mathbf{u}_{b_1}}{\|\mathbf{u}_{a_2}\| \cdot \|\mathbf{u}_{b_1}\|} & \frac{\mathbf{u}_{a_2} \cdot \mathbf{u}_{b_2}}{\|\mathbf{u}_{a_2}\| \cdot \|\mathbf{u}_{b_2}\|} & \cdots & \frac{\mathbf{u}_{a_2} \cdot \mathbf{u}_{b_n}}{\|\mathbf{u}_{a_2}\| \cdot \|\mathbf{u}_{b_n}\|} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\mathbf{u}_{a_m} \cdot \mathbf{u}_{b_1}}{\|\mathbf{u}_{a_m}\| \cdot \|\mathbf{u}_{b_1}\|} & \frac{\mathbf{u}_{a_m} \cdot \mathbf{u}_{b_2}}{\|\mathbf{u}_{a_m}\| \cdot \|\mathbf{u}_{b_2}\|} & \cdots & \frac{\mathbf{u}_{a_m} \cdot \mathbf{u}_{b_n}}{\|\mathbf{u}_{a_m}\| \cdot \|\mathbf{u}_{b_n}\|} \end{bmatrix}$$

Type	Dataset	Domain	Table A	Table B	# Matches	# Attr.
Structured	Amazon-Google$_1$	software	1,363	3,226	1,300	4
	Walmart-Amazon$_1$	electronics	2,554	22,074	1,154	6
	DBLP-Google$_1$	citation	2,616	64,263	5,347	4
	DBLP-ACM$_1$	citation	2,616	2,294	2,224	4
	Fodors-Zagats$_1$	restaurant	533	331	112	7
	Hospital$_1$	person	1,786	1,786	3,949	7
	Songs-Songs$_1$	music	1,000,000	1,000,000	146,011	5
Textual	Amazon-Google$_2$	software	1,363	3,226	1,300	2
	Walmart-Amazon$_2$	electronics	2,554	22,074	1,154	2
	Abt-Buy	product	1,081	1,092	1,097	3
Dirty	Amazon-Google$_3$	software	1,363	3,226	1,300	4
	Walmart-Amazon$_3$	electronics	2,554	22,074	1,154	6
	DBLP-Google$_2$	citation	2,616	64,263	5,347	4
	DBLP-ACM$_2$	citation	2,616	2,294	2,224	4
	Fodors-Zagats$_2$	restaurant	533	331	112	7
	Hospital$_2$	person	1,786	1,786	3,949	7
	Songs-Songs$_2$	music	1,000,000	1,000,000	146,011	5

Figure 5.6: Datasets for our experiments

According to the definition of the cosine similarity of two vectors: $\cos(\mathbf{u}, \mathbf{v}) = (\mathbf{u} \cdot \mathbf{v})/(\|\mathbf{u}\| \cdot \|\mathbf{v}\|)$, the matrix W contains the cosine similarity scores for all tuple pairs across A and B:

$$W = \begin{bmatrix} \cos(\mathbf{u}_{a_1}, \mathbf{u}_{b_1}) & \cos(\mathbf{u}_{a_1}, \mathbf{u}_{b_2}) & \cdots & \cos(\mathbf{u}_{a_1}, \mathbf{u}_{b_n}) \\ \cos(\mathbf{u}_{a_2}, \mathbf{u}_{b_1}) & \cos(\mathbf{u}_{a_2}, \mathbf{u}_{b_2}) & \cdots & \cos(\mathbf{u}_{a_2}, \mathbf{u}_{b_n}) \\ \vdots & \vdots & \ddots & \vdots \\ \cos(\mathbf{u}_{a_m}, \mathbf{u}_{b_1}) & \cos(\mathbf{u}_{a_m}, \mathbf{u}_{b_2}) & \cdots & \cos(\mathbf{u}_{a_m}, \mathbf{u}_{b_n}) \end{bmatrix}$$

The above procedure shows how we calculate the cosine similarity scores for all tuple pairs with matrix operations. A nice property of this procedure is that we can speed up the matrix operations with GPUs, which allows us to get the similarity scores efficiently in practice. In this project, we implement this procedure in PyTorch [118], a popular DL framework with GPU acceleration support. It provides an easy way to speed up the computation using GPUs without knowing the details underneath.

Candidate Set Generation: Once we have the matrix W, in this step we generate a candidate set by selecting the pairs with the k highest cosine scores in each row of W. PyTorch also provides the row-wise topk selection function to make sure it runs efficiently.

5.6 Empirical Evaluation

5.6.1 Evaluation Setting

Datasets: Figure 5.6 shows the datasets used for our experiments. These datasets are of different domains, sizes, and number of matches. All of them have been used for EM evaluation (e.g., [107, 30, 58]). For structured EM problem setting, we use 7 datasets. Six of them are publicly available, and the rest one, Hospital_1, describes the staff information from a local hospital. For this type of EM problem, the tuples in each dataset are *structured* with short and atom attribute values (i.e., not a composition of multiple values that should appear separately).

For textual EM, we use 3 datasets. In these datasets, each tuple has 2-3 attributes, all of which are *textual blobs* (e.g., titles, descriptions, etc.).

For dirty EM, we use 7 datasets. As described in Section 4.2, we focus on one type of dirtiness where attribute values are "injected" into other attributes, e.g., the value of the attribute "brand" is missing from the cell and it appears in the cell for attribute "title". As we do not have real-world dirty data available for evaluation, in this project we use the same way to generate the dirty data as described in Section 4.5. Specifically, all dirty datasets are derived from the corresponding structured datasets described above (e.g., Amazon-Google_3 is generated based on Amazon-Google_1). To generate the datasets, for each attribute we randomly move its value to the attribute "title" with 50% probability.

Methods: We evaluate six DL solutions in this project. Five of them are the representative solutions we propose: SIF, Autoencoder, Seq2seq, CTT, and Hybrid. The rest one is the existing solution DeepER [40]. Our proposed solutions are implemented in PyTorch [118]. For DeepER we use the code[1] provided by [40]. All solutions are trained and evaluated on a workstation with one Intel Xeon Gold 6140 CPU, 188 GB memory, and one NVIDIA 1080 TI GPU.

[1] https://github.com/daqcri/deeper-lite

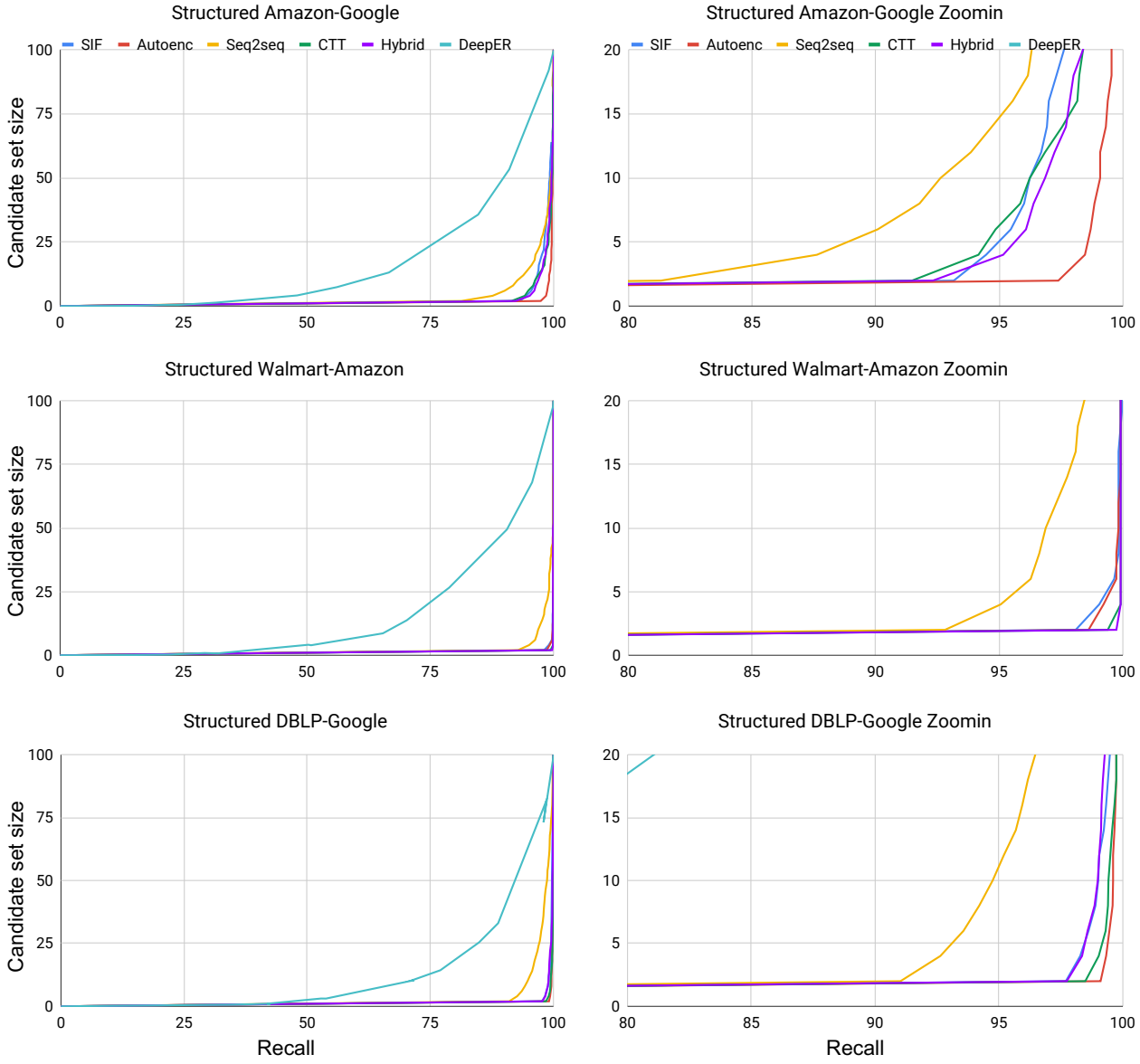

Figure 5.7: R-C curve comparison of different DL solutions for the first three structured datasets.

We compare the DL models with RBB [55], a state-of-the-art learning-based EM solution. We train the DL models with mini-batch gradient descent, and use Adam [76] as the optimization algorithm. We train each DL model for 50 epochs, and select the best model snapshot with the minimal training loss. For the approximated data generation procedure for CTT and Hybrid solutions, we generate 5 negative samples for each tuple in Song-Song$_1$ and Song-Song$_2$ (as this dataset is large in size), and 20 negative samples for all the other datasets.

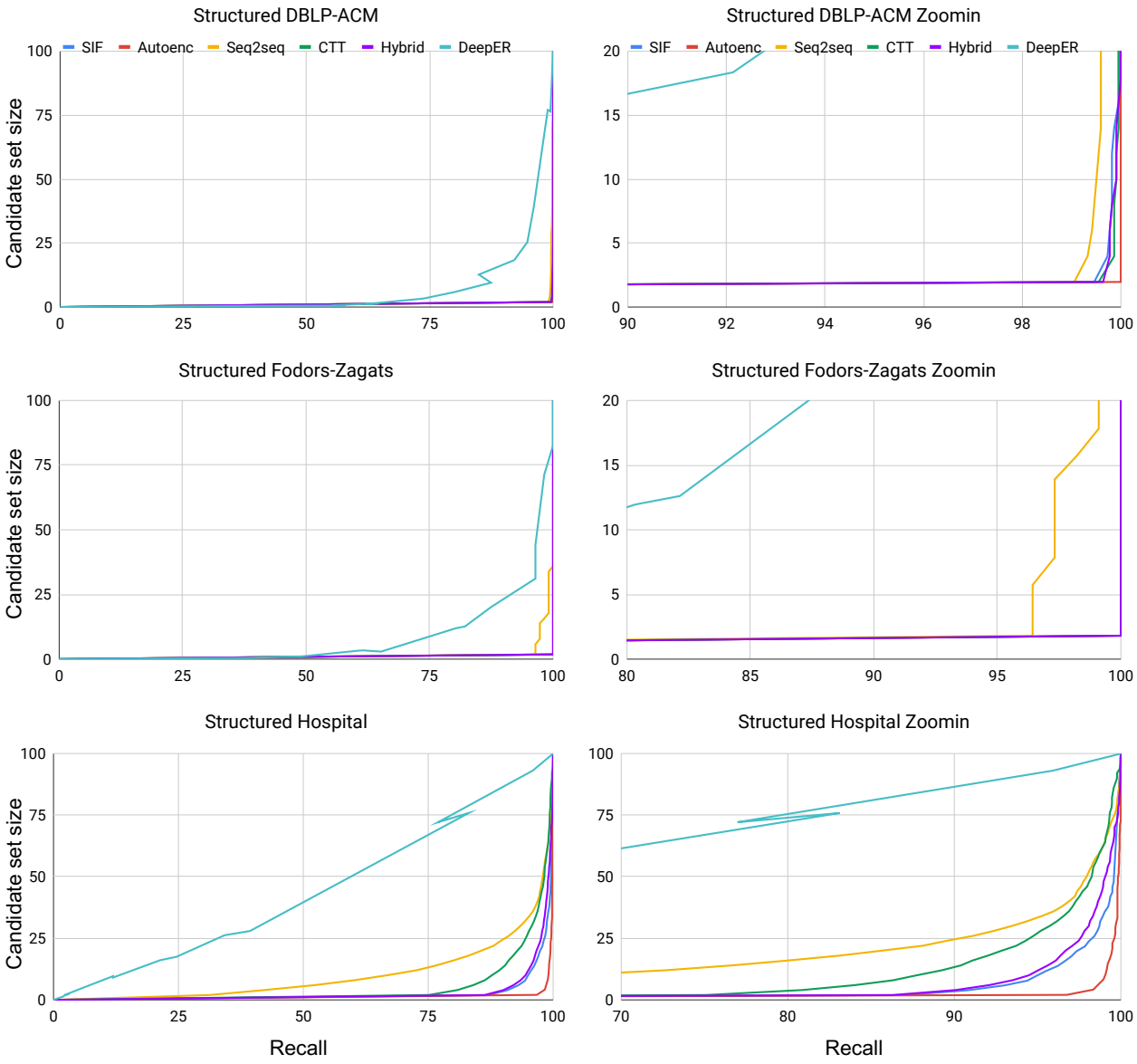

Figure 5.8: R-C curve comparison of different DL solutions for the last three structured datasets.

To quantify the blocking performance, we use three measures: recall, candidate set size, and blocking time. These measures are defined in Section 2.2.

5.6.2 Comparing Different DL Solutions

We first compare the performance of our proposed five representative DL solutions as well as the existing method DeepER, under three different EM problem settings: structured, textual and dirty. For evaluation methods, we choose to plot the R-C (representing Recall-CandidateSetSize)

curve for each DL solution, to show the trend on how the recall and candidate set size will change by varying the hyperparameters in the vector-based pairing module. As we do not fix the hyperparameter values rather compare the R-C curves of different solutions, this evaluation compare the potentials of different DL solutions.

5.6.2.1 Experiments with Structured Data

Figure 5.7 and 5.8 show the R-C curve plots on structured datasets for the six DL solutions. All plots on the left side contain the complete curves, where as the plots on the right side zoom into the bottom right of the corresponding plots on the left side to emphasize on the difference among the solutions. For our proposed five solutions, we get each curve by increasing the hyperparameter K from 0 until we cannot increase the candidate set size (i.e., the candidate set already includes all pairs across the two tables of a dataset). For DeepER, as there are two hyperparameters M (the number of bits for hashcodes) and N (the number of independent hashing rounds), we fix $N = 10$ (which has been used in the evaluation in [40]) and vary M from 0 to 30. The horizontal axis of each plot shows the recall value, and the vertical axis shows the candidate set size ratio which equals the size of the candidate set over the size of the Cartesian product of the two input tables in each dataset. Note that due to the large size of the dataset Song-Song$_1$, it is very expensive to vary K to be large (because the candidate set is very large and time-consuming to get), therefore we skip the experiment on Song-Song$_1$ as we are not able to get the complete R-C curve.

By observing the complete R-C curves of each left-side plot in Figure 5.7 and 5.8, we can see that by varying K, a general trend for all R-C curves is that when recall increases, the size of the candidate set also increases. However, the increase patterns for different DL solutions are different. Technically, we prefer a DL solution whose R-C curve is closer to the bottom-right corner of the plot (e.g., the R-C curve for Autoencoder is closer to the bottom-right corner than the curve for DeepER for Amazon-Google$_1$), as under the best-case scenario, we can get a candidate set with higher recall and smaller size given a proper K value.

From the complete R-C curves on the left side, we summarize the following messages. First, for all datasets, all of our five DL solutions outperform DeepER by a large margin. We suspect

the reason is DeepER uses simple average over the word embeddings of each tuple to get the tuple embeddings, which is very crude such that the tuple vectors are not close in the vector space. This does not satisfy the condition to make LSH work where tuples will be placed into the same bucket with high probability if they are close in the vector space. Second, the R-C difference between our proposed solutions are not significant compared to DeepER, especially on datasets DBLP-ACM$_1$ and Fodors-Zagats$_1$.

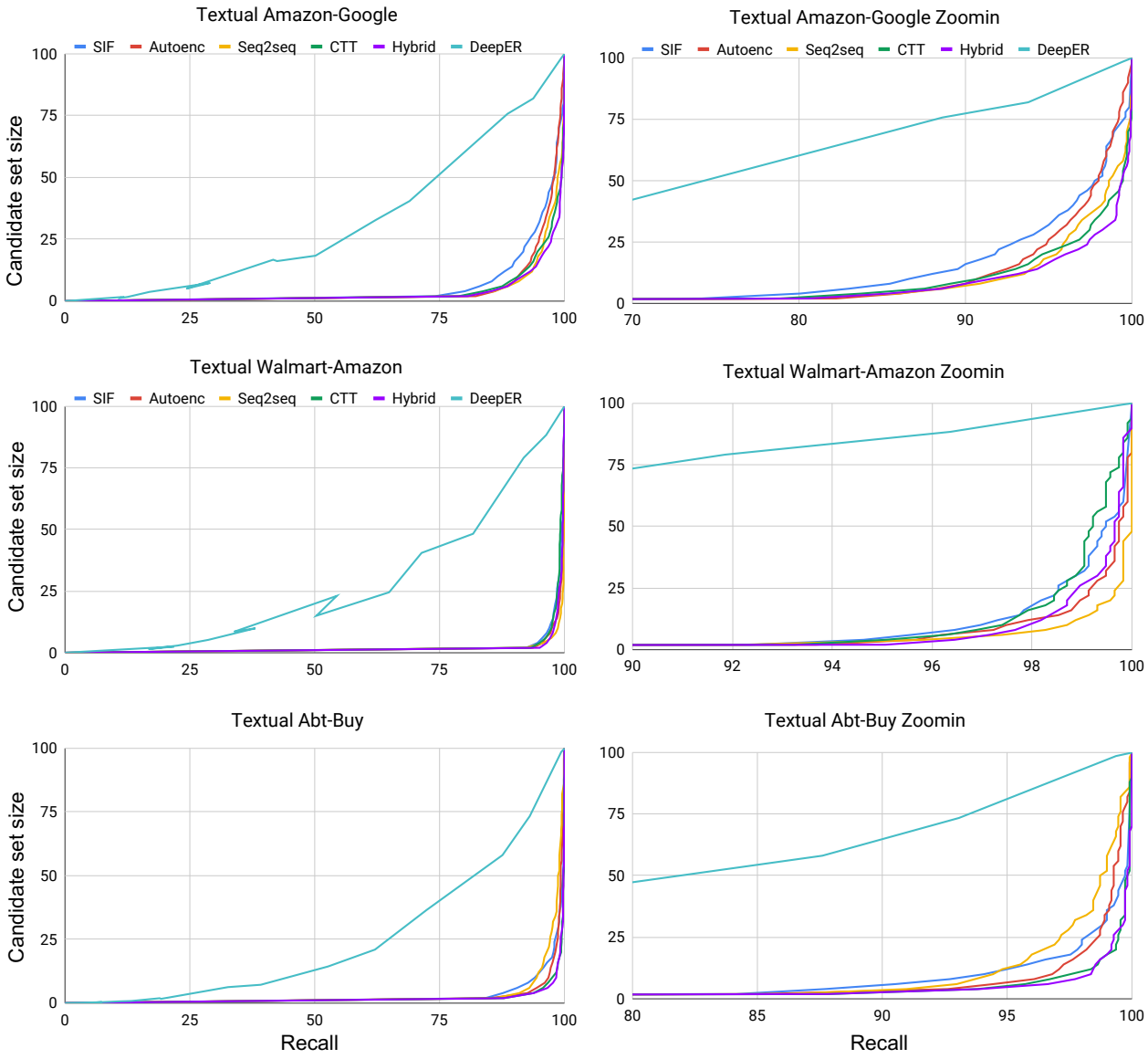

Figure 5.9: R-C curve comparison of different DL solutions for the textual datasets.

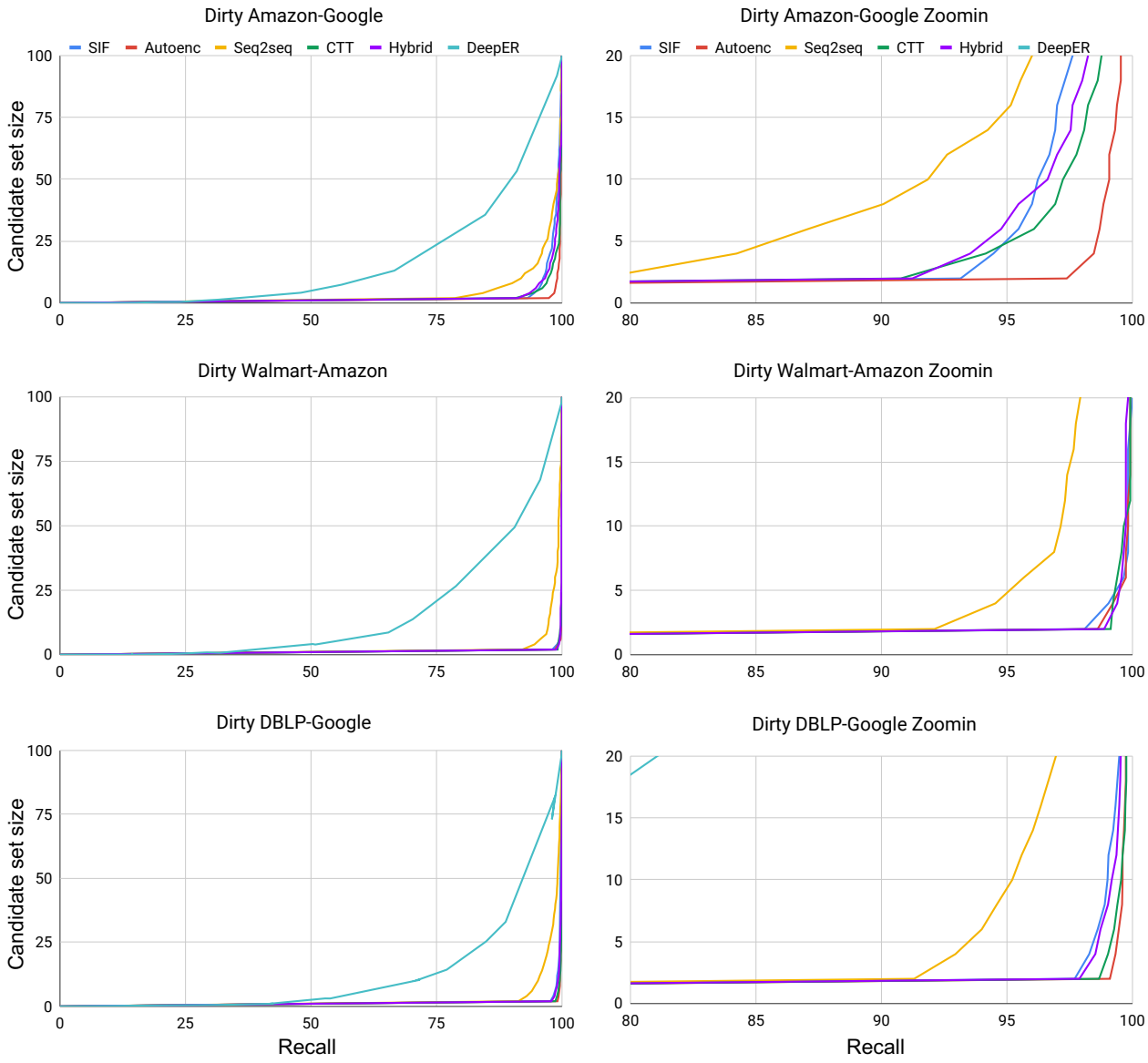

Figure 5.10: R-C curve comparison of different DL solutions for the first three dirty datasets.

If we zoom into the bottom-right of each complete R-C curve plot, we can still see some difference of the proposed solutions, which is shown on the right side in Figure 5.7 and 5.8. Comparing the R-C curves we summarize the following messages. First, Autoencoder achieves the best performance potential on average. Specifically, the R-C curves of Autoencoder are closest to the bottom-right corner on all datasets except for Walmart-Amazon$_1$. This suggests that given structured data where each attribute value is short and atomic, the idea of self-reproduce is very effective

to summarize essential information of a tuple to generate tuple embeddings. Second, Hybrid only achieves the best potential on one dataset Walmart-Amazon$_1$, even if the model has the best representation power of our five proposed solutions. We suspect that the data approximation procedure in Section 5.4.4 restrains the model from learning good cross-tuple information. Third, Seq2seq gives the worst potential on average, even outperformed by SIF. We suspect the reason is that this model aims at recovering the whole tuple string sequence by treating each word equally (in all other four models we incorporate some form of weighting over the words such as SIF aggregation), and in this way it may not capture the most important information of a tuple.

5.6.2.2 Experiments with Textual Data

Figure 5.9 shows the R-C curve plots on textual datasets for the six DL solutions. All plots on the left side contain the complete curves, where as the plots on the right side zoom into the bottom right of the corresponding plots on the left side to emphasize on the difference among the solutions.

By observing the complete R-C curves on the left side we learn the following facts. First, similar to what we have seen on the plots for structured data, all of our proposed solutions outperform DeepER by a large margin. Second, the potential of our proposed solutions is similar, with the corresponding R-C curves close to each other. Third, compared to the R-C curves of the structured data above, here we can see that the R-C curves are further from the bottom-right corner of each plot. This means given textual attributes where the values are long and uncleaned, it is more challenging for DL methods to extract useful information of each tuple and generate good blocking results.

Observing the R-C curves on the right side in Figure 5.9 which give a more detailed comparison on different DL solutions, we summarize the following messages. First, obviously DeepER is outperformed by the others by a large margin. Second, while the procedure is simple without any learning, the performance of SIF is good on textual datasets, which is comparable to or even outperform other methods with more complex network structures. Third, Hybrid achieves the

best potential on average (with the best potential on two datasets Amazon-Google$_2$ and Walmart-Amazon$_2$). This suggests that when we have textual data, it is beneficial to capture cross-tuple information in order to generate better tuple embeddings. On the other hand, this means it is more difficult for the self-reproduce based solutions to summarize the essential information of tuples for textual data.

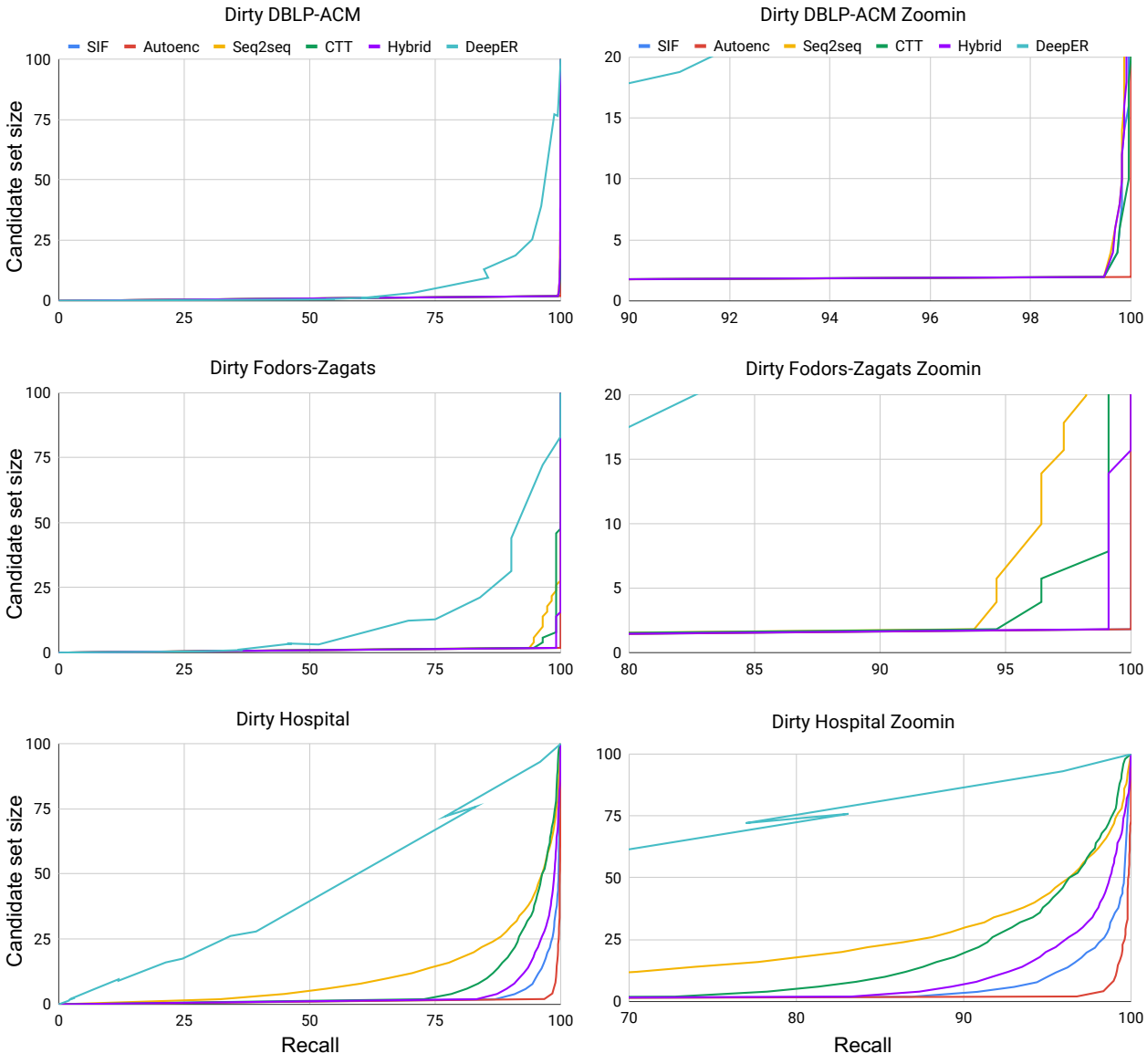

Figure 5.11: R-C curve comparison of different DL solutions for the last three dirty datasets.

5.6.2.3 Experiments with Dirty Data

Figure 5.10 and 5.11 show the R-C curve plots on dirty datasets for the six DL solutions. Again, all plots on the left side contain the complete curves, where as the plots on the right side zoom into the bottom right of the corresponding plots on the left side to emphasize on the difference among the solutions.

Here the plots for each dataset is similar to the corresponding structured ones in the curve patterns, and we get the following observations. First, all of our proposed solutions outperform DeepER by a large margin. Second, though the R-C curves are close in most cases, Autoencoder still outperforms the others slightly with the R-C curves closest to the bottom-right corner in all datasets except Walmart-Amazon$_2$.

5.6.3 Runtime of Different DL Solutions

We compare the runtime of our proposed five DL solutions in this section. The runtime consists of two parts: the training time of the tuple embedding module, and the runtime of the vector-based pairing module.

5.6.3.1 Training Time of the Tuple Embedding Module

We first compare the training of different tuple embedding implementations described in Section 5.4 for the five representative solutions. Table 5.1 shows the training time for each dataset under three EM settings, and Table 5.2 summarizes the training time for each tuple embedding model by taking the average over each type of EM datasets. From the tables we summarize the following observations. First, as there is no learning occurred for SIF, this model is the most efficient one among the five (with training time 0). Second, the training time of Seq2seq is much longer than the others. Especially for the large datasets Song-Song$_1$ and Song-Song$_2$, the training time is 34.5 and 34.1 hours (shown in Table 5.1) for each, which takes more than 30x longer time to train than all of the rest models. This is because the sequential nature of LSTMs which cannot be parallelized. Third, observing Table 5.2, we see that apart from Seq2seq, a general trend is that with a more complex model, it takes longer time to train. However, the training time is still reasonable.

Type	Datasets	SIF	Autoencoder	Seq2seq	CTT	Hybrid
Structured	Amazon-Google$_1$	0	39s	3m	2.5m	3.2m
	Walmart-Amazon$_1$	0	49s	51.2m	8m	9.5m
	DBLP-Google$_1$	0	1.8m	4.7h	12m	15.5m
	DBLP-ACM$_1$	0	41s	8.7m	2.7m	3.5m
	Fodors-Zagats$_1$	0	30s	1.3m	1.5m	2.2m
	Hospital$_1$	0	31s	3.3m	1.7m	2.3m
	Song-Song$_1$	0	12m	34.5h	45.3m	57.8m
Textual	Amazon-Google$_2$	0	56s	10m	2.9m	3.9m
	Walmart-Amazon$_2$	0	1.1m	1.2h	12.6m	14m
	Abt-Buy	0	30s	4.3m	2.1m	2.7m
Dirty	Amazon-Google$_3$	0	37s	3m	3m	3.2m
	Walmart-Amazon$_3$	0	46s	52.5m	8.3m	9.6m
	DBLP-Google$_2$	0	1.8m	4.8h	12m	15.4m
	DBLP-ACM$_2$	0	44s	9.2m	2.7m	3.5m
	Fodors-Zagats$_2$	0	32s	1.3m	1.5m	2.2m
	Hospital$_2$	0	31s	3.1m	1.5m	2.2m
	Song-Song$_2$	0	12m	34.1h	43.3m	1.0h

Table 5.1: Training time comparison of different tuple embedding implementations.

For example, for the most complex model Hybrid, the average training time is 13.4m, 6.9m, and 13.7m for structured, textual, and dirty data respectively. Even for the large datasets Song-Song$_1$ and Song-Song$_2$ with 1 million tuples for each table, it takes 57.8m and 1h to train respectively. This indicates that our proposed solutions (except for Seq2seq) scale well on the tuple embedding module training given large datasets. Last, for Autoencoder which achieves the best potential for structured and dirty data (shown in Section 5.6.2), the average training time is only 2.4m, 51s, and 2.4m for structured, textual, and dirty data respectively. Specifically, the training time for large datasets Song-Song$_1$ and Song-Song$_2$ is only 12m each. This suggests that Autoencoder is also a very efficient model to train and use in practice.

5.6.3.2 Runtime of Vector-based Pairing

In this section we compare the runtime of the pairing function for our proposed five DL solutions. As we use topk-based cosine similarity for the vector-based pairing implementation for all

Type	SIF	Autoencoder	Seq2seq	CTT	Hybrid
Structured	0	2.4m	5.8h	10.5m	13.4m
Textual	0	51s	28.8m	5.9m	6.9m
Dirty	0	2.4m	5.7h	10.3m	13.7m

Table 5.2: Average training time comparison of different tuple embedding implementations.

solutions, there is one hyperparameter K to tune. Therefore we compare the runtime curves of DL solutions by varying the hyperparameter K.

Structured Data: Figure 5.12 shows the runtime curves for the structured datasets shown in Table 5.6. For five datasets we vary the value K from 1 to 1000, and for the rest two datasets we vary K from 1 to 100 (as Fodors-Zagats$_1$ is too small and Song-Song$_1$ is too large to select a large K value).

The first six datasets are relatively small in table sizes where as the last one Song-Song$_1$ is large and we want to see if the topk-based cosine similarity pairing function scales well with GPU acceleration on large datasets. From the plots we observe the following facts.

First, as K increases, the runtime increases linearly for each DL solution. This is aligned with our expectation because when K increases, the candidate set size also increases linearly.

Second, when the tables in a dataset are small, the runtime is very short. For the first six datasets, even if we select the largest K in each plot (i.e., K=100 for Fodors-Zagats$_1$, and K=1000 for the rest five datasets), the runtime is in the range of 2-17 seconds for all solutions. Also, the runtime difference is very small among the solutions (and the difference is largely from data preprocessing and model loading). This shows the topk cosine similarity function with GPU acceleration runs very fast given relatively small datasets.

Third, when the tables of a dataset are large in size, the runtime is longer. As shown in Figure 5.12, when we select K=100, the runtime on Song-Song$_1$ is in the range of 645-1588 seconds for different DL solutions. But still, this is pretty fast because it takes only up to about 26 minutes for the pairing function to perform a cross-product comparison over two tables with 1 million tuples each, and generate a candidate set of 100 million pairs. This suggests with GPU acceleration, the pairing function scales well given large datasets.

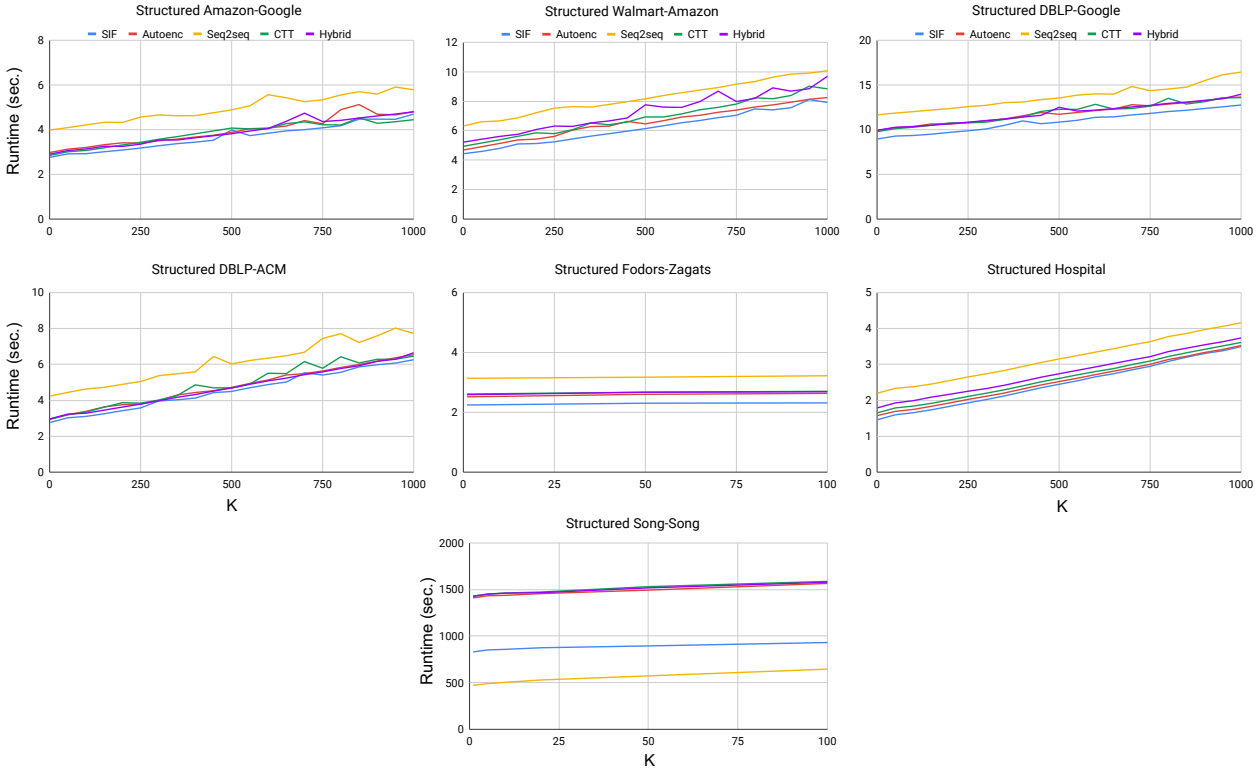

Figure 5.12: Vector-based pairing time comparison of different DL solutions for structured datasets.

Last, we can see that for the large dataset Song-Song$_1$, the runtime difference among different DL solutions is significant. This is because the generated tuple embeddings of different solutions are of different dimensions, and it takes longer time to compute the cosine similarity scores for the tuple embedding pairs of higher dimensions. This may not be obvious when the tables are small as the cosine similarity computation across the two tables can be finished quickly (see the runtime plots in Figure 5.12 for the first six datasets). However, when the tables are large and the cosine similarity computation takes the major part of runtime, the effect of using different tuple embedding dimensions will be enlarged. In this project, we use 150 for the generated tuple embedding dimension for Seq2seq, 300 for SIF, and 600 for the rest three solutions (the tuple embedding dimension for each solution is tuned on Amazon-Google$_1$ and used for all datasets in Table 5.6). In the runtime plot for Song-Song$_1$, we can see that Seq2seq has the shortest

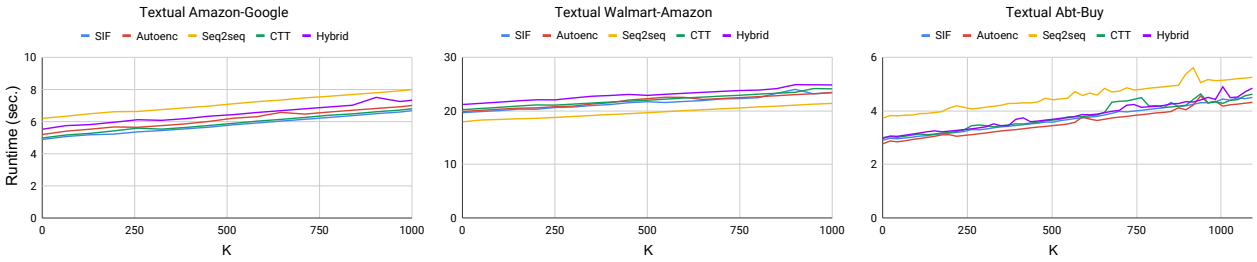

Figure 5.13: Vector-based pairing time comparison of different DL solutions for textual datasets.

pairing runtime, then SIF, and finally the rest three solutions. Also, the runtime difference among different solutions is roughly proportional to the number of tuple embedding dimensions used, which is aligned with our expectation.

Textual Data: Figure 5.13 shows the runtime curves of different DL solutions for the textual datasets shown in Table 5.6. From the figure we observe the following facts.

First, again we can see the runtime increases linearly for each solution as we increase the value for K, which is aligned with our expectation.

Second, the runtime is very short, with less than 25 seconds for all datasets, and the runtime of different DL solutions is very close. This means the topk-based cosine similarity function scales well for the textual datasets we have. In this project we do not have large textual datasets (e.g., with 1 million tuples in each table like Song-Song$_1$) for evaluation, and we plan to evaluate in future if there are large textual datasets available.

Dirty Data: Figure 5.14 shows the runtime curves of different DL solutions for the dirty datasets shown in Table 5.6. We omit the discussion here as the runtime patterns for dirty datasets are very similar to the structured ones as shown in Figure 5.12.

5.6.4 Comparing DL with Non-DL Solutions

In this section, we compare DL solutions with the non-DL ones in real blocking settings (where we need to select values for hyperparameters in the DL solutions to fix the candidate set), to see whether DL can help blocking or not. As for the non-DL baseline, we use a state-of-the-art

rule-based blocker RBB, which learns a set of rules with active learning, following the blocking procedure in [55].

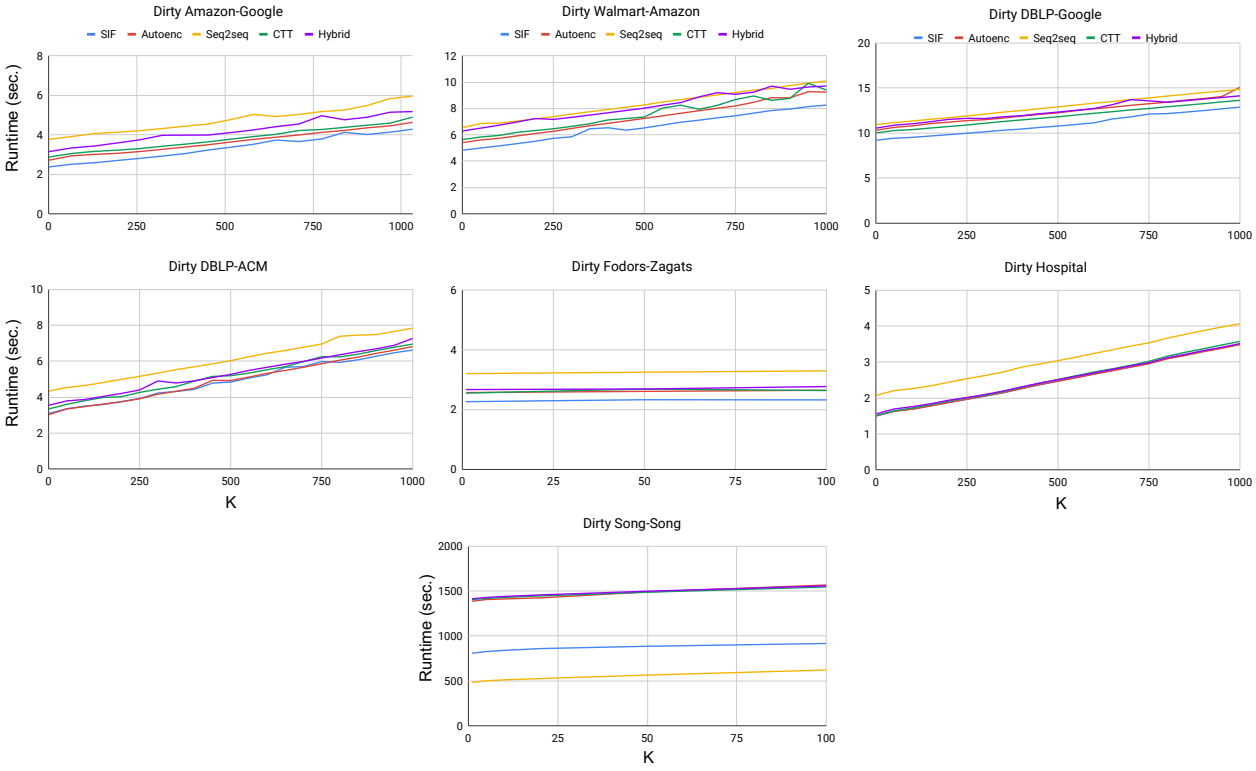

Figure 5.14: Vector-based pairing time comparison of different DL solutions for dirty datasets.

Evaluation Method: The evaluation method consists of four steps. First, we pick the best-performed DL solution on average for each dataset. That is, we choose to use Autoencoder if the dataset is structured or dirty, and Hybrid if the dataset is textual.

Then we run a complete DL blocking procedure by choosing a value for the hyperparameter K. In order to select a good value for K, we run DL blocking with debugging. Specifically, in this project we debug blocking using MatchCatcher described in Chapter 3. The DL blocking with debugging works in the following procedure. First, as we cannot try all possible values for K, we define a value sequence Q_K for K: 1, 5, 10, 20, 50, 100, etc. (i.e., with 50 as the increment step after 50). Next, we start with $K = 1$, and run DL blocking with the current K value to generate the candidate set C_K. Then we run MatchCatcher to debug the candidate set C_K which returns

the $m(= 200)$ most similar pairs that are not in C_K. We verify the m pairs to see if there are true matches killed off by the blocker. If we find more than $n(= 10)$ killed-off true matches, we say the current K value is too small to achieve good recall, and increase K to be the next value in Q_K and repeat the previous steps. Otherwise, we stop the blocking procedure with debugging, and use C_K as the final generated candidate set by DL.

Next, we run RBB to generate the candidate set. Specifically, RBB uses active learning to learn a set of blocking rules, which will ask one to iteratively label at most 30 rounds of tuple pairs selected by the active learner, with 20 pairs a round. Then we apply the learned blocking rules to get the candidate set.

Finally, we compare the candidate sets of DL and RBB to see if one will outperform the other in terms of recall and candidate set size ratio. As mentioned in Section 2.2, there is a trade-off between getting high recall and a small candidate set ratio. To claim one blocking method m_1 outperforms another method m_2, we use the following rules. Denote r_1 and r_2 be the recall of the candidate set of m_1 and m_2 respectively, and c_1 and c_2 be the size of the candidate set of m_1 and m_2 respectively. First, if $r_1 \geq r_2$ and $c_1 \leq c_2$, m_1 outperforms m_2 as m_1 generates a smaller candidate set with higher recall. Second, if r_1 is comparable to r_2 (with less than 1% difference in this project) and $c_1 < c_2$ (with at least 20% smaller in size in this project), m_1 outperforms m_2. Third, if $r_1 > r_2$ (at least 1% higher in this project) and c_1 is reasonable (e.g., less than 10% of the size of the Cartesian product of the two tables), m_1 outperforms m_2. The third rule implies that in our evaluation we prioritize achieving high recall over a small candidate set size, which has been used in the evaluation of many real-world EM tasks. Of course there can be many other different ways to compare the performance of different blocking methods. For example, we can compare the size of the candidate sets when we make different methods achieve roughly the same recall, or we can compare the recall when we fix the size of the candidate set of different methods. We consider applying these different comparison methods as future work.

Comparing DL with RBB: Table 5.3 summarizes the blocking result comparison of DL and RBB on each dataset. The "K" column under the "DL" section shows the final value selected for the hyperparameter K for each dataset, using MatchCatcher debugging. The "|C|" columns

Type	Dataset	DL			RBB	
		K	|C|	Recall	|C|	Recall
Structured	Amazon-Google$_1$	50	68.2k	97.1	16.3k	88.2
	Walmart-Amazon$_1$	20	51.1k	92.2	2.1m	92.0
	DBLP-Google$_1$	150	392.4k	98.1	7.3m	96.9
	DBLP-ACM$_1$	5	13.1k	99.6	189.7k	95.5
	Fodors-Zagats$_1$	1	533	100	-	-
	Hospital$_1$	150	140k	99.0	90k	99.3
	Song-Song$_1$	50	50m	95.0	486.1k	94.7
Textual	Amazon-Google$_2$	20	27.3k	70.5	8.4k	60.2
	Walmart-Amazon$_2$	5	12.8k	68.7	7.9m	64.2
	Abt-Buy	20	21.6k	87.2	28.3k	82.9
Dirty	Amazon-Google$_3$	50	68.2k	97.1	320.5k	87.3
	Walmart-Amazon$_3$	10	25.5k	88.0	924.3k	87.0
	DBLP-Google$_2$	150	392.4k	98.1	47.5m	97.2
	DBLP-ACM$_2$	5	13.1k	99.6	991.7k	98.7
	Fodors-Zagats$_2$	1	533	100	-	-
	Hospital$_2$	10	8.3k	89.0	133.9k	88.4
	Song-Song$_2$	50	50m	95.0	51k	8.9

Table 5.3: Comparison of DL and RBB .

shows the size of the DL candidate set, and the "Recall" column shows the recall value of the candidate set for each dataset. Similarly, the "|C|" and "Recall" columns under the "RBB" section show the size of the candidate set and the recall for each dataset using RBB. In this evaluation, we have no results for datasets Fodors-Zagats$_1$ and Fodors-Zagats$_2$ using RBB, because these two datasets are too small to be applicable.

From the table, we summarize the following key messages. First, on structured datasets, it is not clear whether DL will outperform RBB, as we observe a mixed pattern of the comparison results of DL and RBB. Specifically, on the first four structured datasets (i.e., from Amazon-Google$_1$ to DBLP-ACM$_1$), DL consistently achieves higher recall than RBB, and generates a smaller candidate set on all datasets except for Amazon-Google$_1$ (in this case, the recall of DL is 8.9% higher than RBB). Therefore DL outperforms RBB on the first four structured datasets. However, for the last two structured datasets Hospital$_1$ and Song-Song$_1$, the recall of DL and RBB is comparable (i.e., 99.0% vs. 99.3% on Hospital$_1$ and 95.0% vs. 94.7% on Song-Song$_1$), but the size of the candidate set generated by DL is much larger than the one generated by RBB (e.g., 50m vs. 486.1k

Type	Datasets	DL		RBB		Union	
		|C|	Recall	|C|	Recall	|C|	Recall
Structured	Amazon-Google$_1$	68.2k	97.1	16.3k	88.2	77.7k	98.8
	Walmart-Amazon$_1$	51.1k	92.2	2.1m	92.0	2.1m	98.9
	DBLP-Google$_1$	392.4k	98.1	7.3m	96.9	7.6m	99.6
	DBLP-ACM$_1$	13.1k	99.6	189.7k	95.5	198.4k	99.9
	Fodors-Zagats$_1$	533	100.0	-	-	-	-
	Hospital$_1$	140k	99.0	90k	99.3	209.8k	99.9
	Song-Song$_1$	50m	95.0	486.1k	94.7	50m	98.7
Textual	Amazon-Google$_2$	27.3k	70.5	8.4k	60.2	33.6k	85.0
	Walmart-Amazon$_2$	12.8k	68.7	7.9m	64.2	7.9m	83.0
	Abt-Buy	21.6k	87.2	28.3k	82.9	44.6k	95.7
Dirty	Amazon-Google$_3$	68.2k	97.1	320.5k	87.3	360.0k	99.3
	Walmart-Amazon$_3$	25.5k	88.0	924.3k	87.0	935.9k	97.9
	DBLP-Google$_2$	392.4k	98.1	47.5m	97.2	47.6m	99.8
	DBLP-ACM$_2$	13.1k	99.6	991.7k	98.7	1.0m	99.8
	Fodors-Zagats$_2$	533	100.0	-	-	-	-
	Hospital$_2$	8.3k	89.0	133.9k	88.4	136.8k	98.5
	Song-Song$_2$	50m	95.0	51k	8.9	50m	95.2

Table 5.4: Combining the candidate sets of DL and RBB.

on Song-Song$_1$). Therefore RBB outperforms DL on the last two structured datasets. Given this, it is not clear whether DL will outperform RBB on structured datasets, and we plan to find more datasets to verify this in future.

Second, on textual data, DL consistently achieves significant recall improvement over RBB (4.3%-10.3%), and in two cases DL generates the candidate set smaller in size. Therefore, we conclude that on textual datasets DL outperforms RBB.

Third, on dirty data, we can see that again DL consistently achieves higher recall than RBB. In all cases except for Song-Song$_2$, DL generates a smaller candidate set in size. Therefore DL also outperforms RBB on dirty data.

Combining the Candidate Sets of DL and RBB: From Table 5.3 we see that in most cases DL achieves higher recall than RBB. One may ask the question that, if it is the case that DL will learn a more complex model using neural networks than RBB, such that all matches in the candidate set of RBB are included in the candidate set of DL. To answer this, in this experiment we combine the candidate sets of DL and RBB to see if we can get recall improvement.

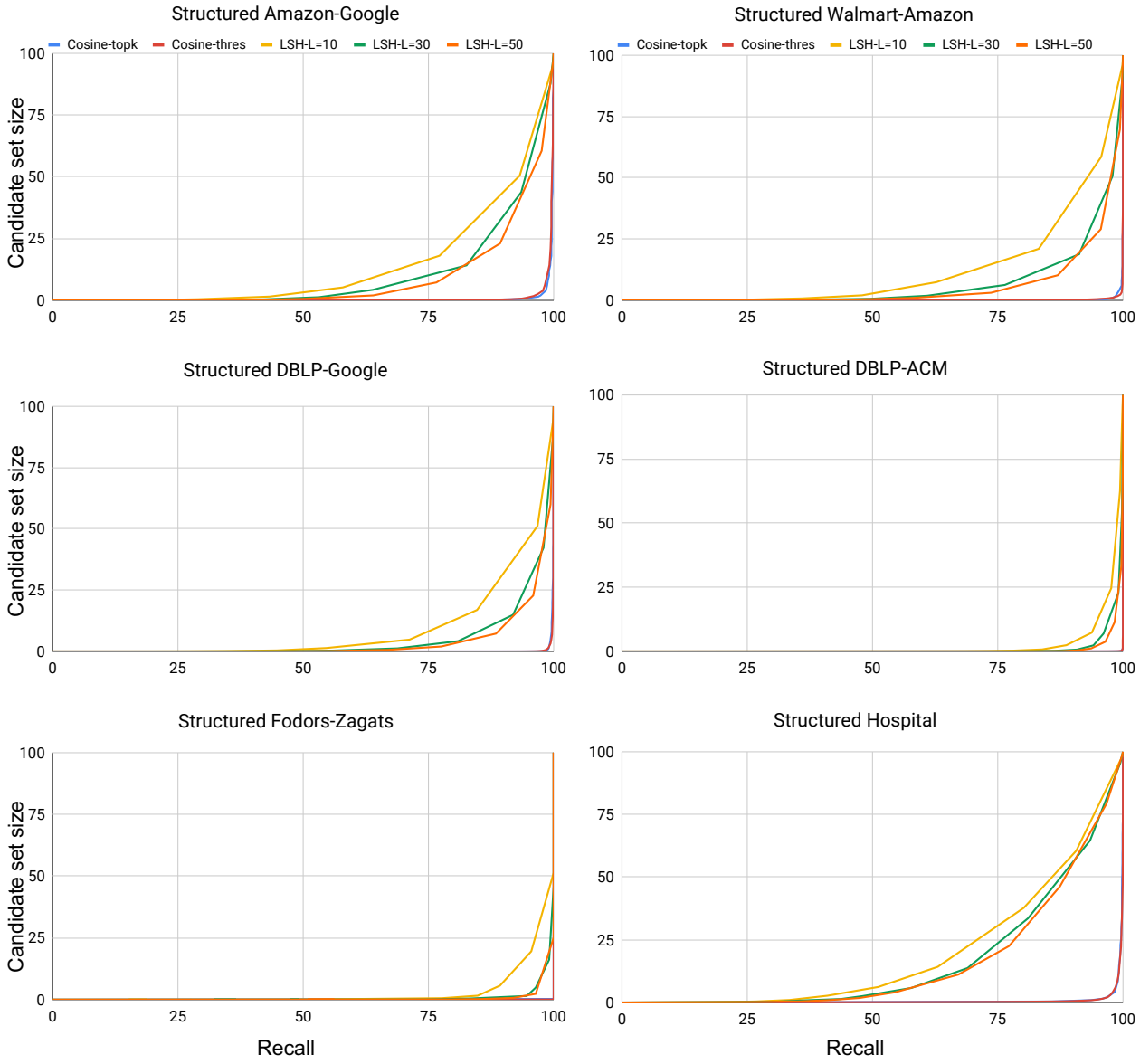

Figure 5.15: Comparison of the R-C curves of the three different pairing functions on structured datasets.

Table 5.4 summarizes the results. The columns "$|C|$" and "Recall" under the "Union" section shows the size of the union of the candidate sets of DL and RBB, and its recall respectively.

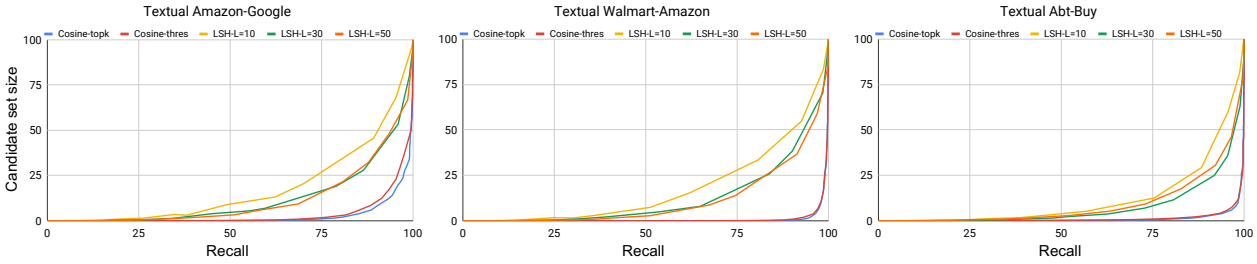

Figure 5.16: Comparison of the three different pairing functions on textual datasets.

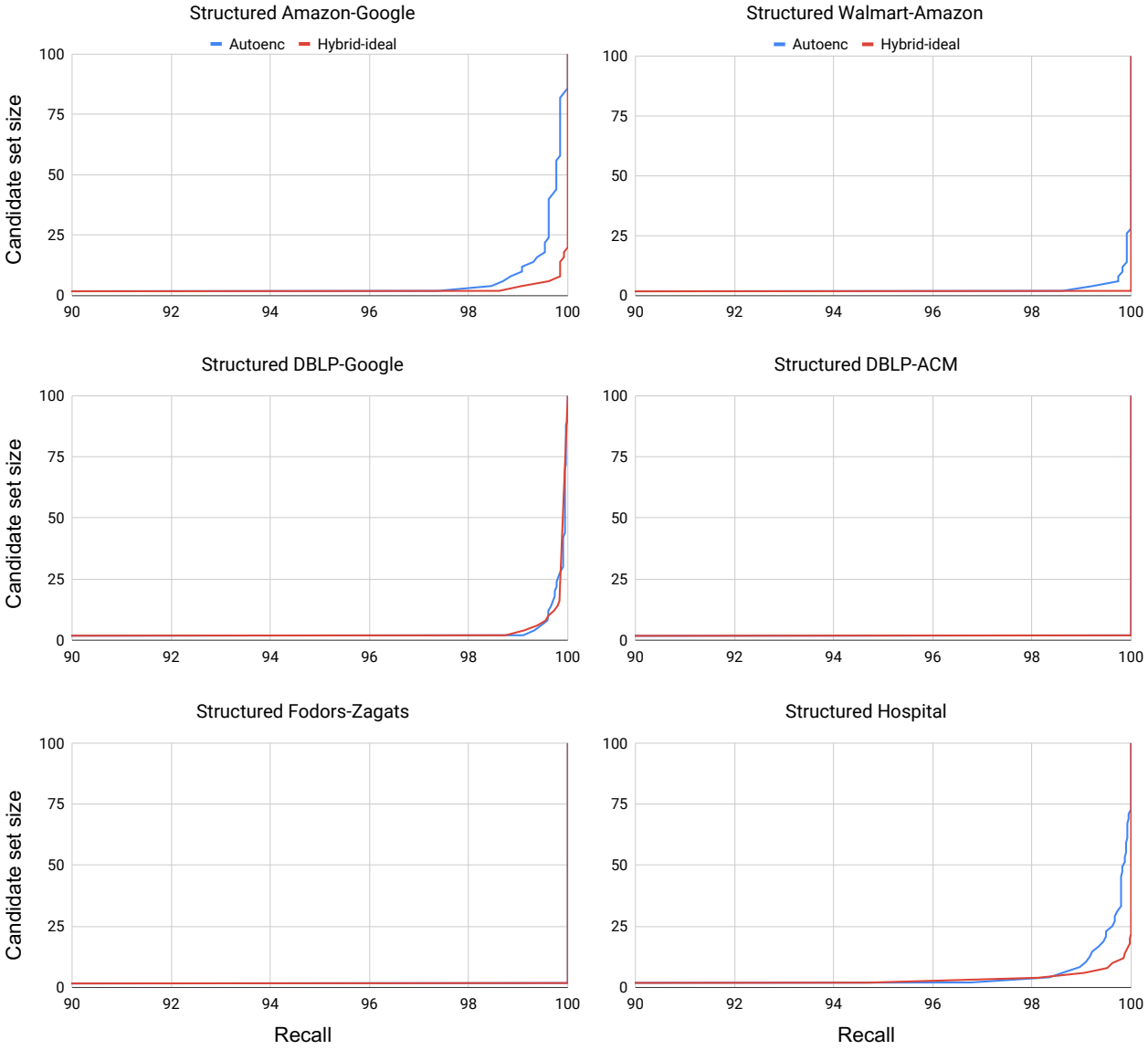

Figure 5.17: Comparison of the R-C curves for Autoencoder and Hybrid with ideal training data on structured datasets.

Figure 5.18: Comparison of the R-C curves for Autoencoder and Hybrid with ideal training data on textual datasets.

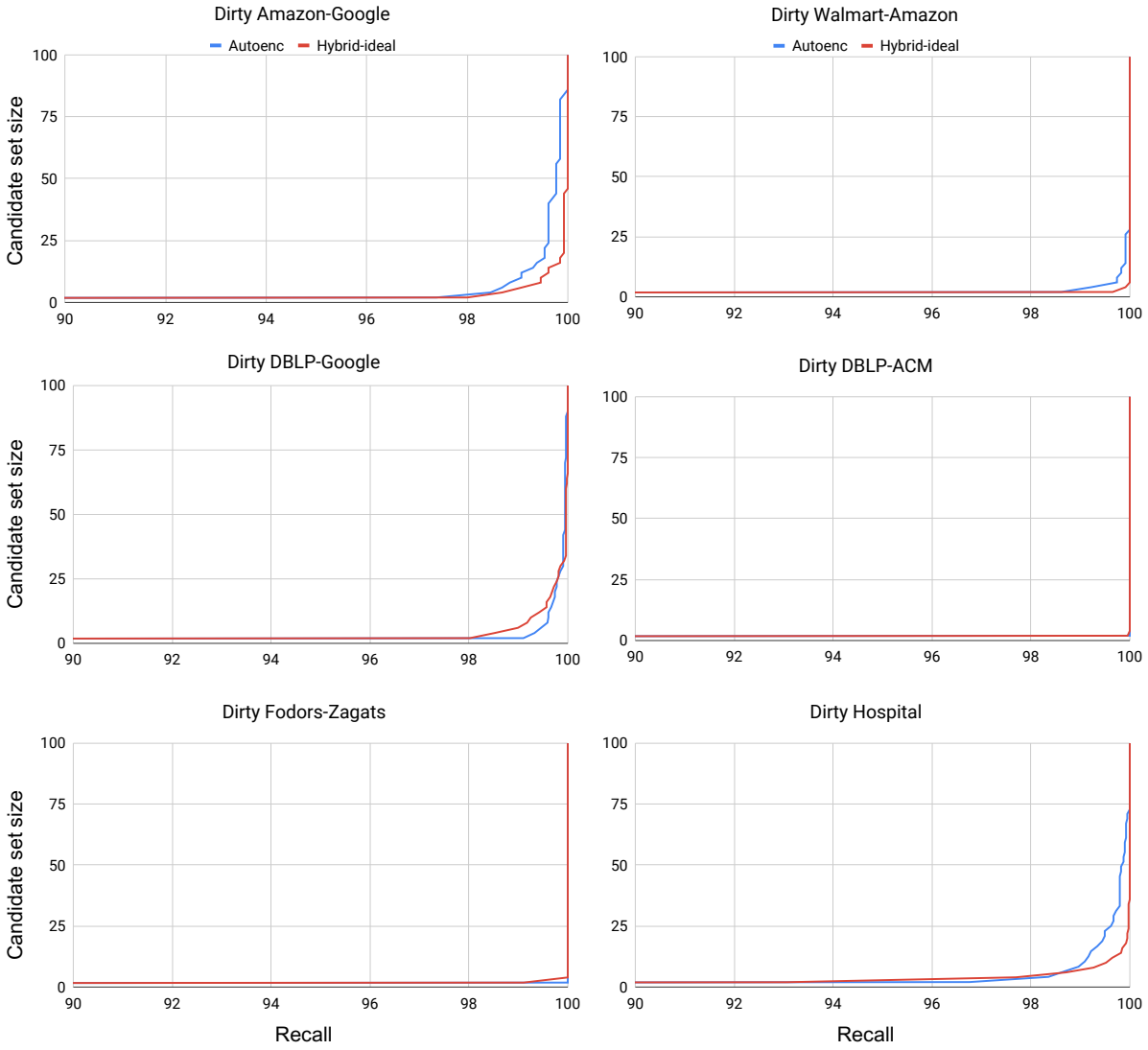

Figure 5.19: Comparison of the R-C curves for Autoencoder and Hybrid with ideal training data on dirty datasets.

From the table, we can see that we do get recall improvement, and in many cases the improvement is significant (e.g., numbers highlighted in red). Specifically, by taking the union we get 0.3-6.7% absolute recall improvement compared to the larger one of DL and RBB on structured datasets, 8.5-14.5% improvement on textual datasets, and 0.2-9.9% improvement on dirty datasets. Also, the candidate set size increase is not large. Specifically, we get up to 49.9% relative size increase compared to the larger value of DL and RBB on structured datasets, up to 57.6% on textual datasets, and up to 12.3% on dirty datasets. This suggests that DL and RBB learn different concepts that are important for blocking, and they are complementary to each other. Therefore, instead of picking one method out of DL and RBB, a better idea to perform blocking is that we first generate the candidate sets using DL and RBB, and then combine the two as the final blocking candidate set.

5.6.5 Micro-benchmarks

In this section, we perform a set of micro-benchmarks.

Comparing Different Pairing Functions: As mentioned in Section 5.5, all five representative solutions use the same pairing function, the topk-based cosine similarity. To better understand its performance, in this evaluation we fix the tuple embedding module for each dataset, and compare it with a few other popular pairing methods. Specifically, we consider three different pairing functions: topk-based cosine similarity, threshold-based cosine similarity, and LSH used in DeepER [40].

Figure 5.15 shows the R-C curves using the autoencoder as the tuple embedding generation implementation and different pairing functions for each structure dataset (except for Song-Song$_1$ where it is very expensive to get the complete R-C curve). For topk-based cosine and threshold-based cosine, since there is only one hyper-parameter, we generate one R-C curve for each function by varying the value of the hyper-parameter. For LSH, as there are two hyper-parameters K (the number of bits for the hashcode) and L (the number of repeated hash runs), we generate three R-C curves, by first selecting three values 10, 30, 50 for L, and then varying the value of K from 0 to 100 for each L. From the figure we summarize the following messages. First, by

increasing the value for L, LSH gets better R-C curves (at the cost of longer blocking time), and the improvement converges with L larger than 30. Second, even with the best R-C curve for LSH where $L = 50$, it is outperformed by threshold-based and topk-based cosine pairing functions by a large margin. This suggests that the LSH function used in DeepER is not very effective for pairing high dimensional vectors. Third, the R-C curves for topk-based and threshold-based cosine functions are very close on all structured datasets. This means that even if they generate the candidate sets based on different ideas, the effect of the two functions is almost the same on structured data.

Figure 5.16 shows the R-C curves of different pairing functions for each textual dataset, where we use the hybrid model for generating tuple embeddings. From the figure we summarize the following observations. First, we can see that again LSH is outperformed by the two cosine-based pairing functions by a large margin. This suggests that the LSH function used in DeepER does not work well on textual data either. Second, the topk-based cosine function outperforms the threshold-based one by a small margin. This means given textual data, the topk-based cosine function is more effective.

As the autoencoder model for tuple embedding generation is insensitive to the word order in a tuple string, it is robust to the type of dirtiness we consider in this project. Therefore the R-C curve for each dirty dataset is identical to the corresponding structured one (e.g., the R-C curve for Amazon-Google$_3$ is identical to the one for Amazon-Google$_1$ shown in Figure 5.15). Hence we omit the discuss on dirty datasets.

To summarize, by comparing three different pairing functions, we get the following conclusions. First, the topk-based cosine function achieves roughly the same performance to the threshold-based cosine on structured and dirty datasets, and outperforms the latter on textual datasets by a small margin. Second, both cosine-based variants outperform LSH used in DeepER by a large margin on all datasets. Hence, the topk-based based cosine function achieves the best performance on average out of the three.

Using Ideal Training Data for Hybrid: We see that Hybrid is outperformed by Autoencoder for structured and dirty data in Section 5.6.2. One possible reason is that we use the approximated

training data such that Hybrid cannot learn good cross-tuple information to generate good tuple embeddings. So in this evaluation we train Hybrid using the ideal training data (described in Section 5.4.4) by assuming we know all true matches across A and B. This will give us the performance upperbound of Hybrid. We want to see under the best-case scenario whether Hybrid will outperform Autoencoder or not.

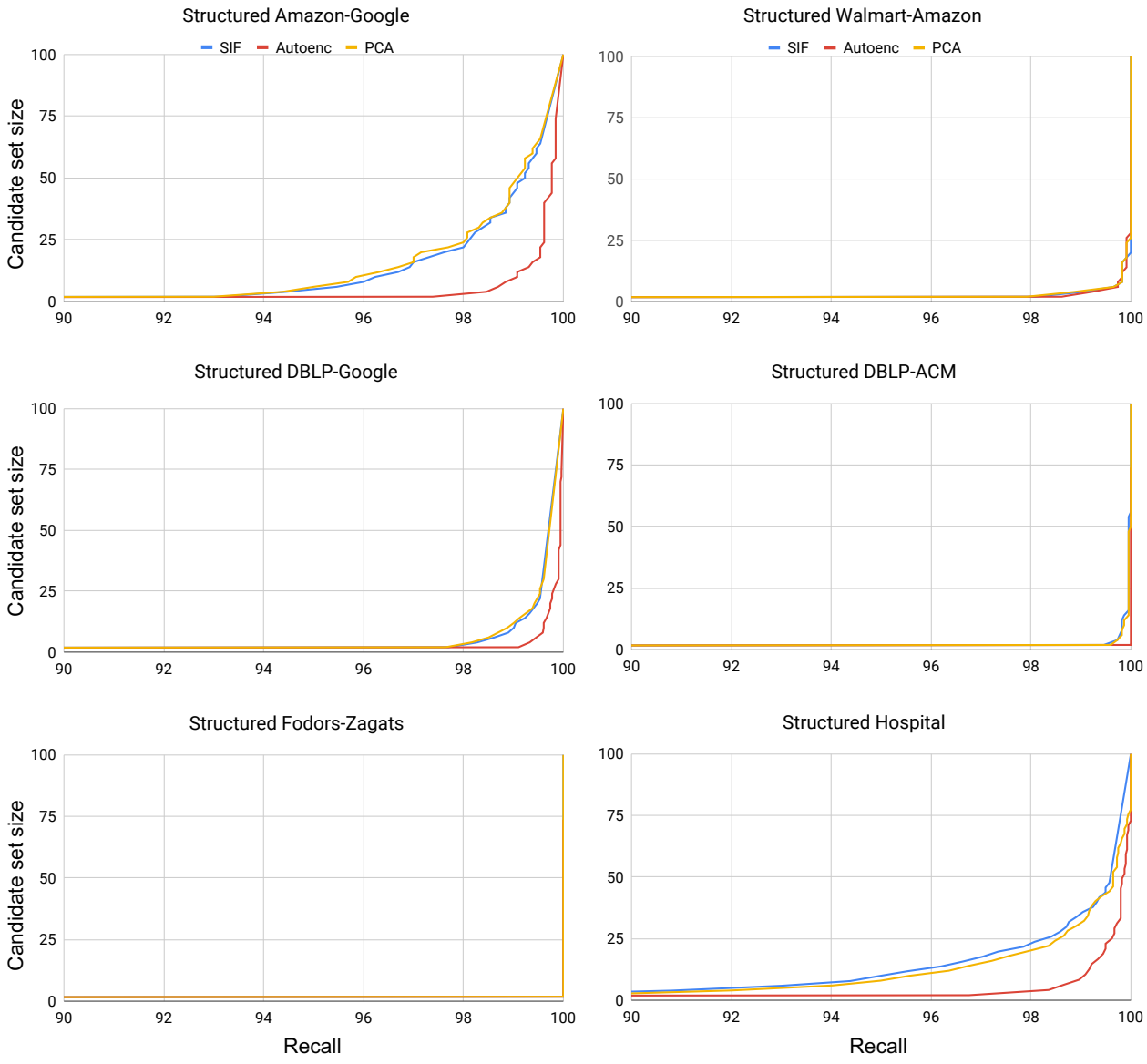

Figure 5.20: Comparison of the R-C curves for SIF, Autoencoder, and PCA on structured datasets.

Figure 5.21: Comparison of the R-C curves for SIF, Autoencoder, and PCA on textual datasets.

Figure 5.17-5.19 show the R-C curve comparison of Autoencoder and Hybrid with ideal training data on structured, textual, and dirty data respectively. From these figures, we can see that given the ideal training data, Hybrid outperforms Autoencoder on all three types of data. Specifically, on structured datasets, Hybrid achieves a better R-C curve on three datasets (i.e., Amazon-Google$_1$, Walmart-Amazon$_1$, and Hospital$_1$), and is comparable to Autoencoder on the rest three. On textual datasets, Hybrid consistently outperforms Autoencoder with a better R-C curve. On dirty datasets, Hybrid achieves a better R-C curve on three datasets (i.e., Amazon-Google$_3$, Walmart-Amazon$_3$, and Hospital$_2$), is comparable to Autoencoder on two datasets (i.e., DBLP-ACM$_2$, Fodors-Zagats$_2$), and is outperformed by Autoencoder on one dataset (i.e., DBLP-Google$_2$). This suggests that given the training data of better quality, Hybrid can learn better cross-tuple information and outperform Autoencoder. Based on this, a promising future direction is to find a better way to approximate the ideal training data to improve the performance of Hybrid.

Using PCA for Blocking: PCA is a traditional non-DL method for dimensionality reduction and information summarization. It is closely related to autoencoders [57], as mentioned in Section 5.2.2. In this evaluation, we build a model for blocking using PCA, and compare it against SIF and Autoencoder. Specifically, the model first applies SIF-based aggregation to generate an aggregation vector for each tuple in the two input tables A and B. Then it performs a PCA decomposition on all aggregation vectors of tuples in A and B to generate a tuple embedding vector for each tuple. The dimension of the tuple embeddings generated by PCA decomposition is selected to be 250 (We tune the value on Amazon-Google$_1$ and use it for all datasets.). Finally, we run vector-based

pairing on the generated tuple embeddings using the topk-based cosine similarity function, which is the same as in SIF and Autoencoder.

Figure 5.20-5.21 show the R-C curve comparison of the three methods on structured and textual datasets respectively. From the figures, we summarize the following observations. First, the R-C curves for SIF and PCA are very close on each dataset shown in the figures. This means that applying PCA decomposition upon SIF to select the most prominent features achieves almost identical effect to SIF. Second, both SIF and PCA are outperformed by Autoencoder. Specifically, on structured datasets, Autoencoder achieves a better R-C curve on four datasets and is comparable to SIF and PCA on the rest two (i.e., Walmart-Amazon$_1$ and Fodors-Zagats$_1$). On textual datasets, Autoencoder achieves a better R-C curve on two datasets, and is comparable to SIF and PCA on the rest one (i.e., Abt-Buy). This suggests that using neural networks with nonlinear transformation, Autoencoder learns a more powerful generalization of PCA.

For all three methods SIF, Autoencoder, and PCA, they are insensitive to the type of dirtiness we consider in this project. Hence for each dirty dataset, we get R-C curves identical to the corresponding structured ones for the three methods. We omit the discussion on dirty data here.

Chapter 6

Conclusions

Entity matching (EM) is the task of finding data records that refer to the same real-world entity. This problem is critical for many Big Data and data science applications. It has received much attention and is becoming more and more important in the data science era.

There have been many EM solutions proposed over the decades. In real applications, oftentimes the developed solutions provide reasonable results. However, there are still many limitations that prevent these solutions from delivering good performance for certain application scenarios. In this dissertation I have focused on three limitations: limited support for debugging blocking, difficulties in handling structured but dirty data, and difficulties in handling textual data. Toward robust EM, I developed solutions to address the above three limitations. Below are the key contributions in this dissertation.

First, I developed MatchCatcher for debugging blocking, which provides a practical way to find likely matches killed off by the blocker, so that the user can examine these matches to understand how well the blocker does recall-wise and what can be done to improve its recall. So far, no existing work has addressed this problem for EM. I also implemented the MatchCatcher package in Python. It has been open sourced and used by 300+ students in data science class projects and 7 teams at 6 organizations, and has received overwhelmingly positive feedback.

Second, in collaboration with Sidharth Mudgal, I explored using DL for matching to understand the benefits and limitations of DL. Specifically, I defined a space of DL solutions for matching, as embodied by four representative solutions, and compared them extensively with Magellan, a non-DL state-of-the-art, under three types of EM problems: structured, textual and dirty EM. The results show that DL produces competitive performance on structured EM problems, but can

significantly outperform Magellan on textual and dirty ones. For practitioners, this suggests that they should seriously consider using DL for textual and dirty EM problems.

Finally, I explored DL for blocking. By summarizing existing DL work on blocking as well as distilling and adapting non-DL work, I defined a space of DL solutions for blocking, as embodied by five representative solutions. Comparing against RBB, a state-of-the-art non-DL solution, the results show that it is not clear whether DL will help blocking on structured data, but it provides better blocking results on textual and dirty data. I also showed that with GPU acceleration, the proposed DL solutions can be executed efficiently.

Bibliography

[1] D. H. Ackley, G. E. Hinton, and T. J. Sejnowski. A learning algorithm for boltzmann machines. *Cognitive science*, 9(1):147–169, 1985.

[2] S. Arora, Y. Liang, and T. Ma. A simple but tough-to-beat baseline for sentence embeddings. ICLR, 2017.

[3] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. ICLR, 2015.

[4] P. Baldi. Autoencoders, unsupervised learning, and deep architectures. In *Proceedings of ICML workshop on unsupervised and transfer learning*, pages 37–49, 2012.

[5] T. Bansal, J. Pachocki, S. Sidor, I. Sutskever, and I. Mordatch. Emergent complexity via multi-agent competition. *arXiv preprint arXiv:1710.03748*, 2017.

[6] D. Barowy, D. Gochev, and E. Berger. Checkcell: data debugging for spreadsheets. OOPSLA, 2014.

[7] Y. Bengio, R. Ducharme, P. Vincent, and C. Jauvin. A neural probabilistic language model. *Journal of machine learning research*, 3(Feb):1137–1155, 2003.

[8] M. Bilenko, B. Kamath, and R. J. Mooney. Adaptive blocking: Learning to scale up record linkage. In *Proceedings of the 6th IEEE International Conference on Data Mining (ICDM)*, 2006.

[9] M. Bilenko and R. J. Mooney. Adaptive duplicate detection using learnable string similarity measures. SIGKDD, 2003.

[10] P. Bojanowski, E. Grave, A. Joulin, et al. Enriching word vectors with subword information. *CoRR*, abs/1607.04606, 2016.

[11] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.

[12] B. Brancotte, B. Yang, G. Blin, S. Cohen-Boulakia, A. Denise, and S. Hamel. Rank aggregation with ties: Experiments and analysis. VLDB, 2015.

[13] J. Camacho-Collados and T. Pilehvar. From word to sense embeddings: A survey on vector representations of meaning. *arXiv preprint arXiv:1805.04032*, 2018.

[14] C. Chen, A. Seff, A. Kornhauser, and J. Xiao. Deepdriving: Learning affordance for direct perception in autonomous driving. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2722–2730, 2015.

[15] Q. Chen, X. Zhu, Z.-H. Ling, et al. Recurrent neural network-based sentence encoder with gated attention for natural language inference. *CoRR*, abs/1708.01353, 2017.

[16] L. Chiticariu and W.-C. Tan. Debugging schema mappings with routes. VLDB, 2006.

[17] K. Cho et al. Learning phrase representations using rnn encoder-decoder for statistical machine translation. EMNLP, 2014.

[18] P. Christen. *Data Matching*. Springer, 2012.

[19] P. Christen. A survey of indexing techniques for scalable record linkage and deduplication. *IEEE TKDE*, 24(9):1537–1555, 2012.

[20] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

[21] K. Clark et al. Improving coreference resolution by learning entity-level distributed representations. *CoRR*, abs/1606.01323, 2016.

[22] W. W. Cohen. Tensorlog: A differentiable deductive database. *CoRR*, abs/1605.06523, 2016.

[23] R. Collobert et al. Natural language processing (almost) from scratch. JMLR, 2011.

[24] R. Collobert, K. Kavukcuoglu, and C. Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, 2011.

[25] A. Conneau, D. Kiela, H. Schwenk, L. Barrault, and A. Bordes. Supervised learning of universal sentence representations from natural language inference data. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 670–680, 2017.

[26] C. Cortes and V. Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[27] G. Cybenko. Approximation by superpositions of a sigmoidal function. *MCSS*, 2:303–314, 1989.

[28] I. Dagan, D. Roth, F. Zanzotto, and G. Hirst. *Recognizing Textual Entailment*. Morgan & Claypool Publishers, 2012.

[29] S. Das, A. Doan, P. S. G. C., C. Gokhale, and P. Konda. The magellan data repository. `https://sites.google.com/site/anhaidgroup/useful-stuff/data`.

[30] S. Das, P. S. GC, A. Doan, J. F. Naughton, et al. Falcon: Scaling up hands-off crowdsourced entity matching to build cloud services. SIGMOD, 2017.

[31] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255. IEEE, 2009.

[32] L. Deng, D. Yu, et al. Deep learning: methods and applications. *Foundations and Trends® in Signal Processing*, 7(3–4):197–387, 2014.

[33] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[34] B. Dhingra, H. Liu, et al. A comparative study of word embeddings for reading comprehension. *CoRR*, abs/1703.00993, 2017.

[35] J. Dittrich. Deep learning (m)eats databases. VLDB Keynote, 2017.

[36] A. Doan, A. Halevy, and Z. Ives. *Principles of data integration*. Elsevier, 2012.

[37] X. Dong, A. Halevy, and J. Madhavan. Reference reconciliation in complex information spaces. SIGMOD, 2005.

[38] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

[39] C. Dwork, R. Kumar, M. Naor, and D. Sivakumar. Rank aggregation methods for the web. WWW, 2001.

[40] M. Ebraheem, S. Thirumuruganathan, S. Joty, M. Ouzzani, and N. Tang. Distributed representations of tuples for entity resolution. *Proceedings of the VLDB Endowment*, 11(11):1454–1467, 2018.

[41] V. Efthymiou, G. Papadakis, G. Papastefanatos, et al. Parallel meta-blocking: Realizing scalable entity resolution over large, heterogeneous data. In *IEEE Big Data*. IEEE, 2015.

[42] V. Efthymiou, G. Papadakis, G. Papastefanatos, K. Stefanidis, and T. Palpanas. Parallel meta-blocking for scaling entity resolution over big heterogeneous data. *Inf. Syst.*, 65:137–157, 2017.

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

[44] R. Fagin, R. Kumar, and D. Sivakumar. Efficient similarity search and classification via rank aggregation. SIGMOD, 2003.

[45] W. Fan, X. Jia, J. Li, and S. Ma. Reasoning about record matching rules. VLDB, 2009.

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

[47] L. Ferrone and F. M. Zanzotto. Symbolic, distributed and distributional representations for natural language processing in the era of deep learning: a survey. *arXiv preprint arXiv:1702.00764*, 2017.

[48] M. Francis-Landau et al. Capturing semantic similarity for entity linking with convolutional neural networks. *CoRR*, abs/1604.00734, 2016.

[49] K. Fukushima and S. Miyake. Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition. In *Competition and cooperation in neural nets*, pages 267–285. Springer, 1982.

[50] H. Galhardas, D. Florescu, D. Shasha, et al. Ajax: an extensible data cleaning tool. SIGMOD, 2000.

[51] O.-E. Ganea and T. Hofmann. Deep joint entity disambiguation with local neural attention. *CoRR*, abs/1704.04920, 2017.

[52] J. Gehring, M. Auli, D. Grangier, D. Yarats, and Y. N. Dauphin. Convolutional sequence to sequence learning. ICML, 2017.

[53] L. Getoor and A. Machanavajjhala. Entity resolution: Theory, practice & open challenges. VLDB, 2012.

[54] G. B. Goh, N. O. Hodas, and A. Vishnu. Deep learning for computational chemistry. *Journal of computational chemistry*, 38(16):1291–1307, 2017.

[55] C. Gokhale, S. Das, A. Doan, J. F. Naughton, N. Rampalli, J. Shavlik, and X. Zhu. Corleone: hands-off crowdsourcing for entity matching. SIGMOD, 2014.

[56] D. Golub and X. He. Character-level question answering with attention. *CoRR*, abs/1604.00727, 2016.

[57] I. Goodfellow et al. *Deep Learning*. MIT Press, 2016.

[58] Y. Govind, P. Konda, P. S. GC, P. Martinkus, P. Nagarajan, H. Li, A. Soundararajan, S. Mudgal, J. R. Ballard, H. Zhang, et al. Entity matching meets data science: A progress report from the magellan project. *integration*, 1:3, 2019.

[59] Y. Govind, E. Paulson, M. Ashok, P. S. GC, A. Hitawala, A. Doan, Y. Park, P. L. Peissig, E. LaRose, and J. C. Badger. Cloudmatcher: A cloud/crowd service for entity matching.

[60] A. Graves, S. Fernández, et al. Bidirectional lstm networks for improved phoneme classification and recognition. ICANN'05, 2005.

[61] B. F. Green, Jr., A. K. Wolf, C. Chomsky, and K. Laughery. Baseball: An automatic question-answerer. IRE-AIEE-ACM '61 (Western), 1961.

[62] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[63] M. A. Hernández and S. J. Stolfo. Real-world data is dirty: Data cleansing and the merge/purge problem. *Data Min. Knowl. Discov.*, 2(1):9–37, 1998.

[64] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal processing magazine*, 29(6):82–97, 2012.

[65] G. E. Hinton, S. Osindero, and Y.-W. Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006.

[66] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[67] E. Hoffer and N. Ailon. Deep metric learning using triplet network. In *International Workshop on Similarity-Based Pattern Recognition*. Springer, 2015.

[68] K. Hornik. Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2):251–257, 1991.

[69] H. Huang et al. Leveraging deep neural networks and knowledge graphs for entity disambiguation. *CoRR*, abs/1504.07678, 2015.

[70] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning*, pages 448–456, 2015.

[71] Y. Jiang, G. Li, J. Feng, et al. String similarity joins: An experimental evaluation. VLDB, 2014.

[72] D. Jurafsky and J. H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Prentice Hall PTR, 1st edition, 2000.

[73] A. Karpathy, J. Johnson, and L. Fei-Fei. Visualizing and understanding recurrent networks. ICLR Workshop, 2015.

[74] Y. Kim. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*, 2014.

[75] Y. Kim, Y. Jernite, D. Sontag, and A. M. Rush. Character-aware neural language models. AAAI, 2016.

[76] D. Kingma and J. Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[77] L. Kolb, A. Thor, and E. Rahm. Parallel sorted neighborhood blocking with mapreduce. BTW, 2011.

[78] D. Koller, N. Friedman, and F. Bach. *Probabilistic graphical models: principles and techniques*. MIT press, 2009.

[79] P. Konda, S. Das, P. Suganthan GC, A. Doan, A. Ardalan, J. R. Ballard, H. Li, F. Panahi, H. Zhang, J. Naughton, et al. Magellan: Toward building entity matching management systems. VLDB, 2016.

[80] P. Konda et al. Magellan: Toward building entity matching management systems (SIGMOD Research Highlight). *SIGMOD Record*, 2018.

[81] H. Köpcke, A. Thor, and E. Rahm. Evaluation of entity resolution approaches on real-world match problems. VLDB, 2010.

[82] H. Köpcke, A. Thor, S. Thomas, and E. Rahm. Tailoring entity resolution for matching product offers. EDBT, 2012.

[83] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*, pages 489–504. ACM, 2018.

[84] A. Krizhevsky and G. Hinton. Learning multiple layers of features from tiny images. Technical report, Citeseer, 2009.

[85] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[86] Q. V. Le. Building high-level features using large scale unsupervised learning. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 8595–8598. IEEE, 2013.

[87] R. Lebret et al. Word embeddings through hellinger pca. EACL, 2014.

[88] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[89] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[90] H. Li, P. Konda, A. Doan, B. Snyder, Y. Park, G. Krishnan, R. Deep, and V. Raghavendra. MatchCatcher: A debugger for blocking in entity matching. Technical report, 2017. `http://pages.cs.wisc.edu/~anhai/papers1/matchcatcher-tr.pdf`.

[91] J. Li et al. Visualizing and understanding neural models in nlp. NAACL, 2016.

[92] J. Li, W. Monroe, and D. Jurafsky. Understanding neural networks through representation erasure. *CoRR abs/1612.08220*, 2016.

[93] V. C. Liang, R. T. Ma, W. S. Ng, L. Wang, M. Winslett, H. Wu, S. Ying, and Z. Zhang. Mercury: Metro density prediction with recurrent neural network on streaming cdr data. In *Data Engineering (ICDE), 2016 IEEE 32nd International Conference on*, pages 1374–1377. IEEE, 2016.

[94] Z. C. Lipton, J. Berkowitz, and C. Elkan. A critical review of recurrent neural networks for sequence learning. *arXiv preprint arXiv:1506.00019*, 2015.

[95] J. Liu, K. Zhao, B. Kusy, J.-r. Wen, K. Zheng, and R. Jurdak. Learning abstract snippet detectors with temporal embedding in convolutional neural networks. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, pages 895–905. IEEE, 2016.

[96] Y. Liu et al. Learning natural language inference using bidirectional lstm model and inner-attention. *CoRR*, abs/1605.09090, 2016.

[97] C. Manning. Representations for language: From word embeddings to sentence meanings. `https://simons.berkeley.edu/talks/christopher-manning-2017-3-27`, 2017.

[98] W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[99] M. Michelson and C. A. Knoblock. Learning blocking schemes for record linkage. In *Proceedings, The 21st National Conference on Artificial Intelligence*, 2006.

[100] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[101] T. Mikolov, I. Sutskever, K. Chen, et al. Distributed representations of words and phrases and their compositionality. NIPS, 2013.

[102] T. Mikolov, W.-t. Yih, and G. Zweig. Linguistic regularities in continuous space word representations. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 746–751, 2013.

[103] M. Minsky and S. Papert. Perceptrons. 1969.

[104] V. Mnih and G. E. Hinton. Learning to label aerial images from noisy data. ICML, 2012.

[105] B. Mozafari, P. Sarkar, M. Franklin, M. Jordan, and S. Madden. Scaling up crowd-sourcing to very large datasets: a case for active learning. VLDB, 2014.

[106] S. Mudgal et al. Deep learning for entity matching: A design space exploration. Technical report, 2018. `http://pages.cs.wisc.edu/~anhai/papers/deepmatcher-tr.pdf`.

[107] S. Mudgal, H. Li, T. Rekatsinas, A. Doan, Y. Park, G. Krishnan, R. Deep, E. Arcaute, and V. Raghavendra. Deep learning for entity matching: A design space exploration. In *Proceedings of the 2018 International Conference on Management of Data*, pages 19–34. ACM, 2018.

[108] V. Nair and G. E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[109] F. Naumann and M. Herschel. *An Introduction to Duplicate Detection*. Morgan and Claypool Publishers, 2010.

[110] P. Neculoiu, M. Versteegh, and M. Rotaru. Learning text similarity with siamese recurrent networks. ACL, 2016.

[111] M. Nicosia and A. Moschitti. Accurate sentence matching with hybrid siamese networks. CIKM, 2017.

[112] B. Oancea, T. Andrei, and R. M. Dragoescu. Gpgpu computing. *arXiv preprint arXiv:1408.6923*, 2014.

[113] G. Papadakis, G. Alexiou, G. Papastefanatos, and G. Koutrika. Schema-agnostic vs schema-based configurations for blocking methods on homogeneous data. VLDB, 2015.

[114] G. Papadakis, E. Ioannou, T. Palpanas, C. Niederee, and W. Nejdl. A blocking framework for entity resolution in highly heterogeneous information spaces. *IEEE TKDE*, 25(12):2665–2682, 2013.

[115] G. Papadakis, G. Koutrika, T. Palpanas, and W. Nejdl. Meta-blocking: Taking entity resolutionto the next level. *IEEE TKDE*, 26(8):1946–1960, 2014.

[116] A. P. Parikh, O. Täckström, D. Das, and J. Uszkoreit. A decomposable attention model for natural language inference. EMNLP, 2016.

[117] R. Pascanu, T. Mikolov, and Y. Bengio. On the difficulty of training recurrent neural networks. In *International Conference on Machine Learning*, pages 1310–1318, 2013.

[118] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in PyTorch. In *NIPS Autodiff Workshop*, 2017.

[119] J. Pennington, R. Socher, and C. Manning. Glove: Global vectors for word representation. EMNLP, 2014.

[120] M. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer. Deep contextualized word representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 2227–2237, 2018.

[121] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever. Improving language understanding by generative pre-training. *Technical Report*, 2018.

[122] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang. Squad: 100,000+ questions for machine comprehension of text. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 2383–2392, 2016.

[123] A. Ratner, S. H. Bach, H. Ehrenberg, et al. Snorkel: Rapid training data creation with weak supervision. VLDB, 2017.

[124] F. Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[125] S. Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[126] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533, 1986.

[127] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, et al. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015.

[128] T. N. Sainath, A.-r. Mohamed, B. Kingsbury, and B. Ramabhadran. Deep convolutional neural networks for lvcsr. In *Acoustics, speech and signal processing (ICASSP), 2013 IEEE international conference on*, pages 8614–8618. IEEE, 2013.

[129] A. E. Sallab, M. Abdou, E. Perot, and S. Yogamani. Deep reinforcement learning framework for autonomous driving. *Electronic Imaging*, 2017(19):70–76, 2017.

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

[131] K. Schawinski, C. Zhang, H. Zhang, L. Fowler, and G. K. Santhanam. Generative adversarial networks recover features in astrophysical images of galaxies beyond the deconvolution limit. *Monthly Notices of the Royal Astronomical Society: Letters*, 467(1):L110–L114, 2017.

[132] J. Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[133] M. Schuster and K. K. Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):2673–2681, 1997.

[134] Z. Sehili, L. Kolb, C. Borgs, R. Schnell, and E. Rahm. Privacy preserving record linkage with ppjoin. BTW, 2015.

[135] U. Shaham, X. Cheng, O. Dror, et al. A deep learning approach to unsupervised ensemble learning. ICML, 2016.

[136] T. Shen et al. Disan: Directional self-attention network for rnn/cnn-free language understanding. *CoRR*, abs/1709.04696, 2017.

[137] W. Shen, J. Wang, and J. Han. Entity linking with a knowledge base: Issues, techniques, and solutions. *TKDE*, 27(2):443–460, 2015.

[138] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484, 2016.

[139] R. Singh, V. Meduri, A. Elmagarmid, et al. Generating concise entity matching rules. SIGMOD, 2017.

[140] P. Singla et al. Entity resolution with markov logic. ICDM, 2006.

[141] P. Smolensky. Information processing in dynamical systems: Foundations of harmony theory. Technical report, COLORADO UNIV AT BOULDER DEPT OF COMPUTER SCIENCE, 1986.

[142] R. Socher, A. Perelygin, J. Wu, J. Chuang, C. D. Manning, A. Ng, and C. Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 conference on empirical methods in natural language processing*, pages 1631–1642, 2013.

[143] D. Song and J. Heflin. Automatically generating data linkages using a domain-independent candidate selection approach. In *The 10th International Semantic Web Conference*, 2011.

[144] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[145] N. Srivastava, E. Mansimov, and R. Salakhudinov. Unsupervised learning of video representations using lstms. In *International conference on machine learning*, pages 843–852, 2015.

[146] R. K. Srivastava et al. Highway networks. ICML, 2015.

[147] M. Stonebraker, D. Bruckner, I. F. Ilyas, et al. Data curation at scale: The data tamer system. CIDR, 2013.

[148] H. Strobelt et al. Visual analysis of hidden state dynamics in recurrent neural networks. *CoRR abs/1606.07461*, 2016.

[149] C. Sun, A. Shrivastava, S. Singh, and A. Gupta. Revisiting unreasonable effectiveness of data in deep learning era. In *Computer Vision (ICCV), 2017 IEEE International Conference on*, pages 843–852. IEEE, 2017.

[150] Y. Sun, L. Lin, D. Tang, et al. Modeling mention, context and entity with neural networks for entity disambiguation. IJCAI, 2015.

[151] I. Sutskever. *Training recurrent neural networks*. PhD thesis, University of Toronto, 2013.

[152] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[153] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[154] M. Tan et al. Improved representation learning for question answer matching. ACL, 2016.

[155] A. Tian and M. Lease. Active learning to maximize accuracy vs. effort in interactive information retrieval. SIGIR, 2011.

[156] A. Vaswani et al. Attention is all you need. NIPS, 2017.

[157] O. Vinyals, C. Blundell, T. Lillicrap, D. Wierstra, et al. Matching networks for one shot learning. ACL, 2016.

[158] H. Wang and B. Raj. On the origin of deep learning. *arXiv preprint arXiv:1702.07800*, 2017.

[159] J. Wang, T. Kraska, M. J. Franklin, and J. Feng. Crowder: Crowdsourcing entity resolution. VLDB, 2012.

[160] J. Wang, H. T. Shen, J. Song, and J. Ji. Hashing for similarity search: A survey. *arXiv preprint arXiv:1408.2927*, 2014.

[161] S. Wang and J. Jiang. A compare-aggregate model for matching text sequences. ICLR, 2017.

[162] W. Wang et al. Effective multi-modal retrieval based on stacked auto-encoders. 2014.

[163] W. Wang et al. Database meets deep learning: Challenges and opportunities. *ACM SIGMOD Record*, 45(2):17–22, 2016.

[164] W. Wang, N. Yang, F. Wei, B. Chang, and M. Zhou. Gated self-matching networks for reading comprehension and question answering. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pages 189–198, 2017.

[165] W. Wang, M. Zhang, G. Chen, H. Jagadish, B. C. Ooi, and K.-L. Tan. Database meets deep learning: Challenges and opportunities. *ACM SIGMOD Record*, 45(2):17–22, 2016.

[166] Z. Wang, W. Hamza, and R. Florian. Bilateral multi-perspective matching for natural language sentences. *arXiv preprint arXiv:1702.03814*, 2017.

[167] P. Werbos. Beyond regression:" new tools for prediction and analysis in the behavioral sciences. *Ph. D. dissertation, Harvard University*, 1974.

[168] P. J. Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

[169] S. E. Whang, D. Menestrina, G. Koutrika, M. Theobald, and H. Garcia-Molina. Entity resolution with iterative blocking. SIGMOD, 2009.

[170] S. Wiseman, A. M. Rush, and S. M. Shieber. Learning global features for coreference resolution. NAACL, 2016.

[171] S. Wu, L. Hsiao, X. Cheng, et al. Fonduer: Knowledge base construction from richly formatted data. *CoRR*, abs/1703.05028, 2017.

[172] S. Wu, W. Ren, C. Yu, G. Chen, D. Zhang, and J. Zhu. Personal recommendation using deep recurrent neural networks in netease. In *Data Engineering (ICDE), 2016 IEEE 32nd International Conference on*, pages 1218–1229. IEEE, 2016.

[173] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[174] C. Xiao, W. Wang, X. Lin, and H. Shang. Top-k set similarity joins. ICDE, 2009.

[175] C. Xiao, W. Wang, X. Lin, J. X. Yu, and G. Wang. Efficient similarity joins for near-duplicate detection. *TODS*, 36(3):15, 2011.

[176] W. Yin et al. Simple question answering by attentive convolutional neural network. COLING, 2016.

[177] M. Yu, G. Li, D. Deng, and J. Feng. String similarity search and join: a survey. *Frontiers of Computer Science*, 10(3):399–417, 2016.

[178] Z. Zhang, M. Hadjieleftheriou, B. C. Ooi, and D. Srivastava. Bed-tree: an all-purpose index structure for string similarity search based on edit distance. SIGMOD, 2010.

[179] R. F. Zhiguo Wang, Wael Hamza. Bilateral multi-perspective matching for natural language sentences. IJCAI, 2017.

[180] Y. Zou, X. Jin, Y. Li, Z. Guo, E. Wang, and B. Xiao. Mariana: Tencent deep learning platform and its applications. *Proceedings of the VLDB Endowment*, 7(13):1772–1777, 2014.