Synthesizing Entity Matching Rules by Examples

Rohit Singh† Vamsi Meduri‡ Ahmed Elmagarmid⋆ Samuel Madden†
Paolo Papotti‡ Jorge-Arnulfo Quiané-Ruiz⋆ Armando Solar-Lezama† Nan Tang⋆

†CSAIL, MIT, USA ‡Arizona State University, USA
⋆Qatar Computing Research Institute, HBKU, Doha, Qatar

{rohitsingh, madden, asolar}@csail.mit.edu, {vmeduri, ppapotti}@asu.edu
{aelmagarmid, jquianeruiz, ntang}@hbku.edu.qa

ABSTRACT

Entity matching (EM) – the task of determining if two objects in a database are the same – is a critical part of data integration and cleaning. Typically, entity matching is done via rules which compare two entities and indicate whether they are the same or not; these rules may be provided by an expert, or learned from training data.

In this work, we study the problem of *synthesizing entity matching rules from examples*. Given *only* positive and negative matching examples from the user, our method generates EM rules. The core of our solution is *program synthesis*, a powerful tool to automatically generate rules (or programs) that provably satisfy a given high-level specification, defined using a predefined grammar. This grammar is expressive enough to model entity matching problems, from capturing arbitrary attribute combinations to handling missing attribute values. The result is a *General Boolean Formula* (**GBF**) which can include arbitrary attribute matching predicates combined by conjunctions ($\bigwedge$), disjunctions ($\bigvee$) and negations ($\neg$). Our synthesis engine generates rules that are more concise than traditional EM rules represented in Disjunctive Normal Form, and consequently more interpretable than decision trees and other machine learning algorithms that output deep trees with many branches. Existing synthesis techniques by themselves are insufficient for the EM-problem because they cannot cope with large numbers of examples. We address this issue with a new algorithm that smartly builds small sets of training examples covering various corner cases and from them synthesizes EM rules that are effective over the entire dataset.

Extensive experiments show that we outperform other interpretable rules (e.g., decision trees with low depth) in terms of effectiveness, and are comparable with other unexplainable tools, e.g., decision trees with high depth and SVM.

1. INTRODUCTION

Entity matching (EM), where a system or user finds records that refer to the same real-world object, is a fundamental part of data integration and data cleaning. For example, when integrating two datasets, duplicate entities are very common, and merging or eliminating those duplicates is necessary before the integrated data can be useful.

There is a key tension in EM algorithms: on one hand, algorithms that properly match records are clearly preferred. On the other hand, *interpretable* EM rules – that a human can easily understand because they consist of a logical structure (as opposed to statistical models that are composed of weights and functional parameters) – are desirable for two reasons. First, there are a number of critical applications, such as healthcare [7] and other domains [6, 26, 27], where users need to understand *why* two entities are considered a match. Second, *interpretable* EM rules can be easily optimized at execution time by using blocking-based techniques [18], to alleviate the quadratic complexity of pairwise comparisons; such techniques do not apply to uninterpretable models. Systems that use probabilistic models – such as machine learning methods based on SVMs [5], or fuzzy matching [19] – are much harder to interpret and hence are often not preferred. In contrast, systems that are rule-based ("deterministic") [18] offer better interpretability, particularly when the rules can be constrained to be simple (consisting of relatively few clauses). A key question, however, is whether such simple rules can match the effectiveness of probabilistic approaches while preserving interpretability.

Although there are many possible factors that can contribute to whether a model – which is a series of clauses or predicates – is interpretable, we use a simple and intuitive metric of interpretability in this paper: we consider a model as more interpretable than another if it consists of fewer logical predicates or atoms.

Of course, hand-writing EM rules may be practical in some limited domains, doing so is extremely time consuming and error-prone. Hence, a promising direction is to use automatic methods to generate deterministic EM rules, e.g., by learning rules from training examples. This learning should be done with as few examples as possible – since generating the training examples is itself laborious.

We describe an automated entity matching system that can learn EM rules that 1) match the performance of probabilistic methods, 2) produce simple, interpretable rules, and 3) learn effective rules from limited training examples. Our approach is to use *program synthesis* [38] (PS), in which a program (set of rules) is generated by using positive and negative examples as constraints that guide the synthesizer towards rules that match the examples. We show that this

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

approach generates rules that are both concise and effective.

1.1 Motivating Examples

To further explain the problem and challenges, we consider two motivating examples.

Example 1: Two tables of famous people are shown in Figure 1. Dataset D_1 is an instance of schema R (name, address, email, nation, gender) and D_2 is of schema S (name, apt, email, country, sex). The EM problem is to find tuples in D_1 and D_2 that refer to the same person.

Off-the-shelf schema matching tools [16, 32] may decide that the attributes name, address, email, nation, gender in table R map to attributes name, apt, email, country, sex in table S, respectively.

Given a tuple $r \in D_1$ and a tuple $s \in D_2$, a straightforward EM rule is that *all aligned attributes should match* such as:

φ_1: $r[\text{name}] \approx_1 s[\text{name}] \bigwedge r[\text{address}] \approx_2 s[\text{apt}]$
$\bigwedge r[\text{email}] = s[\text{email}] \bigwedge r[\text{nation}] = s[\text{country}]$
$\bigwedge r[\text{gender}] = s[\text{sex}]$

Here, $\approx_1$ and $\approx_2$ are different domain-specific similarity functions. Rule φ_1 says that a tuple $r \in D_1$ and a tuple $s \in D_2$ refer to the same person (i.e., a match), if they have *similar* or *equivalent* values on all aligned attributes. □

However, in practice, the rule φ_1 above may result in very low recall, since real-world data may contain multiple issues such as misspellings (e.g., s_3[name]), different formats (e.g., r_2[name] and s_1[name]), and missing values (e.g., r_3[email]). Naturally, in practice, a robust solution is to have a set of rules that collectively cover different cases.

Example 2: Alternatively, we may have two rules as below.

φ_2: $r[\text{name}] \approx_1 s[\text{name}] \bigwedge r[\text{address}] \approx_2 s[\text{apt}]$
$\bigwedge r[\text{nation}] = s[\text{country}] \bigwedge r[\text{gender}] = s[\text{sex}]$;
φ_3: $r[\text{name}] \approx_3 s[\text{name}] \bigwedge r[\text{email}] = s[\text{email}]$

Typically, these kinds of rules are specified as disjuncts, e.g., $\varphi_2 \bigvee \varphi_3$, which indicates that a tuple $r \in D_1$ and a tuple $s \in D_2$ match, if either φ_2 or φ_3 holds. However, a more natural way, from a user perspective, is to specify the rule in a logical flow. For instance, when handling Null values, the following representation may be more user-friendly:

φ_4: **if** $(r[\text{email}] \neq \text{Null} \bigwedge s[\text{email}] \neq \text{Null})$
then $r[\text{name}] \approx_1 s[\text{name}] \bigwedge r[\text{email}] = s[\text{email}]$
else $r[\text{name}] \approx_3 s[\text{name}] \bigwedge r[\text{address}] \approx_2 s[\text{apt}] \bigwedge$
$r[\text{nation}] = s[\text{country}] \bigwedge r[\text{gender}] = s[\text{sex}]$

These if-then-else rules provide a more flexible way to model matching rules and have more expressive power than simple disjunctions. □

1.2 Challenges

There are three key challenges in automatically discovering good EM rules from examples.

Challenge 1: Huge Search Space. Consider two relations with n aligned attributes. There are $m = 2^n$ possible combinations of attributes. If we constrain ourselves to EM rules that consist of arbitrary selections of these attribute combinations represented in **DNF** (disjunctive normal form), this results in a search space of $\Sigma_{i=1}^{m}\binom{m}{i} = 2^m - 1 = 2^{2^n} - 1$.

In Example 2, the search space is $2^{2^5} - 1$ (4 billion combinations) for only 5 attributes! Adding to the above complexity is that for each attribute, there is a large set of possible similarity functions, e.g., Levenshtein distance or Jaccard similarity, that could be used to determine if the attributes match. Each function has a similarity threshold as well.

Challenge 2: Limited Training Data. Good training data is hard to obtain [31], requiring significant human effort. Thus, a solution to the EM rule mining problem should try to leverage limited training data as much as possible.

Challenge 3: Missing Values. In the real world, there are many missing values across multiple attributes, e.g., the two missing email values in Figure 1(a) are unexplainable by any candidate similarity function – i.e., the missing value may or may not be similar to one of the known values.

For instance, the fact that $r_4 \in D_1$ and $s_2 \in D_2$ refer to the same person is correct, but this example will mislead the training process when trying to find a similarity function on the email attribute that matches a Null value to "bob.dylan@gmail.com". The same issue happens with incorrect values in the examples.

1.3 Contributions

Our major contribution in this paper is a new EM rule synthesis engine that generates rules that are both concise and effective. These compact rules are easy for the end user to interpret, but, because they are expressed in a rich grammar that includes negations and if/then/else clauses, they perform as well as much more complicated probabilistic rules or rules written purely as DNFs. Similar to other database applications that rely on synthesis [34, 35], our system uses a predefined grammar fixed across all the datasets – users do not need to know it.

The core of our approach is an algorithm based on *Program Synthesis* [38] (PS), in the *Syntax-Guided Synthesis* (SyGuS) framework [2]. Given a predefined grammar for EM rules, PS is optimized to explore the (massive) space of possible rules and find rules that satisfy the provided examples – i.e., that output "True" for positive examples and "False" for negative ones (Challenge 1). Unfortunately, existing SyGuS solvers are designed to find solutions that satisfy all, not partial, examples. To cope with these challenges, we devise a new algorithm, namely RuleSynth, which adopts the idea of Counter-Example Guided Inductive Synthesis (CEGIS) [39] to perform synthesis from small sets of smartly chosen examples, and is inspired by Random Sample Consensus (RANSAC) [21] to avoid examples that may make the algorithm under-perform. We show experimentally that our approach deals effectively with limited training data (Challenge 2), and we use a rich SyGuS grammar to properly handle missing (null) values (Challenge 3).

We summarize our contributions as follows.

1. We formally define the problem of generating EM rules from positive/negative examples. In particular, we use *General Boolean Formulas* (**GBF**) to represent EM rules, and define an optimization problem to find a *good* EM rule (Section 2).
2. We show how to solve this optimization problem using the SyGuS framework (Section 3). We also develop a new algorithm, built on an open source SyGuS engine named Sketch [38], to synthesize EM rules from positive/negative examples (Section 4).
3. We describe new optimizations in our algorithm, such as techniques to avoid over-fitting, to eliminate biased samples, and to compute composition of multiple rules

	name	address	email	nation	gender
r_1	Catherine Zeta-Jones	9601 Wilshire Blvd., Beverly Hills, CA 90210-5213	c.jones@gmail.com	Wales	F
r_2	C. Zeta-Jones	3rd Floor, Beverly Hills, CA 90210	c.jones@gmail.com	US	F
r_3	Michael Jordan	676 North Michigan Avenue, Suite 293, Chicago		US	M
r_4	Bob Dylan	1230 Avenue of the Americas, NY 10020		US	M

(a) D_1: an instance of schema R

	name	apt	email	country	sex
s_1	Catherine Zeta-Jones	9601 Wilshire, 3rd Floor, Beverly Hills, CA 90210	c.jones@gmail.com	Wales	F
s_2	B. Dylan	1230 Avenue of the Americas, NY 10020	bob.dylan@gmail.com	US	M
s_3	Michael Jordan	427 Evans Hall #3860, Berkeley, CA 94720	jordan@cs.berkeley.edu	US	M

(b) D_2: An instance of the schema S

Figure 1: Sample tables for persons

(Section 5).

4. We experimentally verify that our approach can produce very concise EM rules that perform well both with small and large amounts of training data. In particular, we outperform other interpretable models (i.e., decision trees with low depth, SIFI [41]) in terms of matching accuracy, even when our rules have substantially fewer clauses, and are comparable with other uninterpretable models, e.g., decision trees with large depth and SVM, on accuracy (Section 6).

2. PROBLEM OVERVIEW

In this section, we introduce the notation used in the paper (Section 2.1) and define the problem (Section 2.2).

2.1 Notation

Let $R[\mathsf{A}_1, \mathsf{A}_2, \ldots, \mathsf{A}_n]$ and $S[\mathsf{A}'_1, \mathsf{A}'_2, \ldots, \mathsf{A}'_n]$ be two relations with corresponding sets of n aligned attributes A_i and A'_i ($i \in [1, n]$). As in previous work on entity matching [3, 10, 15, 20, 23, 41], we assume that the attributes between two relations have been aligned and provided as an input; this can be either manually specified by the users, or done automatically using *off-the-shelf* schema matching tools [4, 16, 32]. Note that our approach naturally applies to one relation, i.e., $R = S$ where no schema alignment is needed ($\mathsf{A}_i = \mathsf{A}'_i, \forall i \in [1, n]$). In the rest of the paper, we focus on the discussion using two relations for ease of explanation, but we also run several experiments on entity matching within one relation.

Let r, s be records in R, S and $r[\mathsf{A}_i], s[\mathsf{A}'_i]$ be the values of the attribute $\mathsf{A}_i, \mathsf{A}'_i$ in records r, s, respectively.

A *similarity function* $f(r[\mathsf{A}_i], s[\mathsf{A}'_i])$ computes a similarity score in the real interval $[0, 1]$. A bigger score means that $r[\mathsf{A}_i]$ and $s[\mathsf{A}'_i]$ have a higher similarity. Examples of similarity functions are cosine similarity, edit distance, and Jaccard similarity. A library of similarity functions $\mathcal{F}$ is a set of such general purpose similarity functions, for example the Simmetrics[1] Java package.

Now we are ready to define rules for *matching* the records $r \in R$ and $s \in S$.

Attribute-Matching Rules. An *attribute-matching rule* is a triple $\approx(i, f, \theta)$ representing a Boolean function with value $f(r[\mathsf{A}_i], s[\mathsf{A}'_i]) \geqslant \theta$, where $i \in [1, n]$ is an index, f is a similarity function and $\theta \in [0, 1]$ is a threshold value. Attribute-matching rule $\approx(i, f, \theta)$ evaluating to *true* means that $r[\mathsf{A}_i]$ *matches* $s[\mathsf{A}'_i]$ relative to the specific similarity function f and threshold θ.

[1]https://github.com/Simmetrics/simmetrics

We use the notation $r[\mathsf{A}_i] \approx_{(f,\theta)} s[\mathsf{A}'_i]$ to denote an attribute-matching rule for some underlying similarity function f and threshold θ. We will simply write $r[\mathsf{A}_i] \approx s[\mathsf{A}'_i]$ when it is clear from the context.

Record-Matching Rules. A *record-matching rule* is a conjunction of a set of attribute-matching rules on different attributes. Intuitively, two records r and s *match* iff all attribute-matching rules in the set evaluate to *true*.

Disjunctive Matching Rule. A *disjunctive matching rule* is a disjunction of a set of record-matching rules. Records r and s are matched by this rule iff they are matched by at least one of this rule's record-matching rules.

Indeed, a *disjunctive matching rule* can be seen as a formula in Disjunctive Normal Form (**DNF**) over *attribute-matching rules* as: $\bigvee_{p=1}^{P} \left(\bigwedge_{q=1}^{Q_p} \approx(i_{(p,q)}, f_{(p,q)}, \theta_{(p,q)}) \right)$

There are two main shortcomings of using **DNF** rules:

(1) [Not Concise.] A **DNF** $(u_1 \bigwedge v_1) \bigvee (u_1 \bigwedge v_2) \bigvee (u_2 \bigwedge v_1) \bigvee (u_2 \bigwedge v_2)$ is equivalent to a much more concise formula $(u_1 \bigvee u_2) \bigwedge (v_1 \bigvee v_2)$.

(2) [Expressive Power.] A **DNF** rule without negations cannot express the logic "if (u) then (v) else (w)", which can be modeled using a formula such as $(u \bigwedge v) \bigvee (\neg u \bigwedge w)$. Traditionally, negations are not used in positive EM rules.

Hence, a more natural way than **DNF** to define ER rules is to use general boolean formulas, as defined below.

Boolean Formula Matching Rule. A *Boolean formula matching rule* is an arbitrary *Boolean formula* over attribute-matching rules as its variables and conjunction ($\bigwedge$), disjunction ($\bigvee$) and negation ($\neg$) as allowed operations.

Hence, a *Boolean formula matching rule* is formulated as a *General Boolean Formula* (**GBF**).

Example 3: Consider Example 2. Let the similarity function for matching attributes name in R and name in S (resp. address in R and apt in S) be Levenshtein (resp. Jaccard), with threshold 0.8 (resp. 0.7).

[Attribute-matching rule.] $r[\mathsf{name}] \approx s[\mathsf{name}]$ can be formally represented as $\approx(1, \texttt{Levenshtein}, 0.8)$, where the number 1 is the positional index for the 1^{st} pair of aligned attributes, i.e., name in R and name in S.

[Record-matching rule.] φ_2 is a record-matching rule that can be formalized as:

$$\varphi_2 : \approx(1, \texttt{Levenshtein}, 0.8) \bigwedge \approx(2, \texttt{Jaccard}, 0.7) \bigwedge =(4, \texttt{Equal}, 1.0) \bigwedge =(5, \texttt{Equal}, 1.0)$$

Similarly, φ_3 can be formalized as:

$$\varphi_3 : \approx(1, \texttt{Levenshtein}, 0.8) \bigwedge =(3, \texttt{Equal}, 1.0)$$

[Disjunctive matching rule.] A disjunctive matching rule for φ_2 and φ_3 is the disjunction of the above two record-matching rules, $\varphi_2 \bigvee \varphi_3$.

[Boolean formula matching rule.] Consider a custom *similarity* function `noNulls` that returns 1.0 when the values of the corresponding attributes are both not null and 0.0 otherwise. Using this function, we can formalize φ_4 as:

φ_4: **if** $(\approx(1, \texttt{noNulls}, 1.0))$ **then** φ_2 **else** φ_3 □

Traditionally, a **DNF** captures the language used by experts [11, 41] for quantifying the similarity of records. Each record-matching rule can independently deal with different corner-cases to cover all examples. As explained above, **GBF** can concisely represent a **DNF** and increase its expressibility thereby enhancing the readability. Moreover, another difference with the traditional tools [11, 41] that generate rules as **DNF**s for entity resolution is that we do not require each attribute to show up with the same similarity function and threshold everywhere in the $Boolean formula$. Consider for instance the attribute `name`. In rule φ_2, the similarity function used is Levenshtein with threshold 0.8. A variant φ_3' of φ_3 could use Jaccard similarity with threshold 0.6 for `name`. The reason for this flexibility is evident: values in the same column are not always homogeneous, and we need different similarity functions to capture different matching cases. **DNF** rules, in general, can also use different thresholds for the same attribute, but existing methods [41] typically limit their solution to find only one similarity function for each attribute to reduce the search space.

2.2 Problem Statement

We want to generate an *optimal* general Boolean formula (**GBF**) without user involvement in providing structure for the **GBF**. To evaluate the quality of a **GBF**, we assume that the user provides a set of examples, denoted by $\mathbf{E} = \mathbf{M} \cup \mathbf{D}$, where $\mathbf{M}$ are positive examples, i.e., pairs of records that represent the same entity, and $\mathbf{D}$ are negative examples, i.e., pairs of records that represent different entities.

Before formally defining the studied problem, we discuss the optimality metric to quantify the quality of a **GBF**.

Optimality Metric. Consider a **GBF** Φ and the sets of positive and negative examples $\mathbf{M}$ and $\mathbf{D}$. We define a metric $\mu(\Phi, \mathbf{M}, \mathbf{D})$ returning a real number in $[0, 1]$ that quantifies the *goodness* of Φ. The larger the value of μ, the better is the **GBF**.

Let $\mathbf{M}_\Phi \subset \mathbf{E}$ be the set of all examples (r', s') such that r' and s' are *matched* by Φ. Alternative candidates for optimality metric μ are:

$$\mu_{\text{precision}} = \frac{|\mathbf{M}_\Phi \cap \mathbf{M}|}{|\mathbf{M}_\Phi \cap \mathbf{M}| + |\mathbf{M}_\Phi \cap \mathbf{D}|}$$

$$\mu_{\text{recall}} = \frac{|\mathbf{M}_\Phi \cap \mathbf{M}|}{|\mathbf{M}|}$$

$$\mu_{\text{F-measure}} = \frac{2 \cdot \mu_{\text{precision}} \cdot \mu_{\text{recall}}}{\mu_{\text{precision}} + \mu_{\text{recall}}}$$

Problem Statement (ER-GBF). Given two relations R and S, the aligned attributes between R and S, sets $\mathbf{M}$ and $\mathbf{D}$ of positive and negative examples, a library of similarity functions $\mathcal{F}$, and an optimality metric μ, the **ER-GBF** problem is to discover a **GBF** Φ that maximizes μ.

3. SYNTHESIS OVERVIEW

In this section, we provide an overview of program synthesis (PS), and describe how to formulate the **ER-GBF** problem as a PS problem. The basic idea of program synthesis is to search for a complete program that satisfies a set of input-output examples, given a grammar that specifies what is valid in the programming language of choice. In the context of entity resolution, the grammar represents the space of **GBF**s, and the problem is to search for a **GBF** that satisfies as many of the user-supplied examples as possible.

Our system assumes that the two input relations already have attributes that have been aligned (using an *off-the-shelf* schema matching tool like [32]). Afterwards, to solve the synthesis problem our system must find the best structure of the **GBF** along with the best combinations of attributes, similarity functions, and thresholds for each attribute matching rule in the **GBF**. We use a default library of standard similarity functions; users can easily extend this library by providing domain-specific similarity functions.

Internally, our system uses the formalism of partial programs to frame the synthesis problem. In these partial programs, *holes* are used to represent the unknown aspects of the program, and the space of all possible code fragments that can fill those holes is given by a *grammar*. The partial program also includes *behavioral constraints*, which in the case of our problem require the synthesized program to match the provided examples (Section 3.1). This style of synthesis based on partial programs, behavioral constraints, and grammars of possible code fragments is known as Syntax Guided Synthesis (SyGuS) and it has recently received significant attention by the formal methods community [2]. We give the modeling of the **ER-GBF** problem as a SyGuS problem and show how the appropriate grammar can be represented as a partial program in an open source SyGuS solver, namely SKETCH [38] (Section 3.2). Since, in practice, not all examples may be satisfied, we also define a version of the SyGuS problem as an optimization problem (Section 3.3).

3.1 Partial Programs and Grammars

To better explain how synthesis works, we start with an example of partial programs and grammars.

Partial Programs. A *partial program* represents a space of possible programs by giving the synthesizer explicit choices about what code fragments to use in different places. A simple partial program in SKETCH is shown below:

```
void tester(bit x, bit y) {
    bit t = boolExp(x, y);
    if(x) assert t ==~y;
    if(y) assert t == ~x;
}
bit boolExp(bit x, bit y){
     return {| ((x | ~x) || (y |~y)) |}; }
```

The partial program (called a sketch) gives the synthesizer choices about whether to negate x and y before or-ing them together in order to satisfy the assertions in the tester. The language also supports *unknown constants* indicated with "??" which the solver replaces with concrete values.

Grammar. SyGuS problems are also represented abstractly as a *grammar* G representing a set of expressions, together with a constraint C on the behavior of the desired expression. A *grammar* G is a set of recursive rewriting rules (or *productions*) used to generate expressions over a set of

terminal symbols and *non-terminal (recursive) symbols*. The productions provide a way to *derive* expressions from a designated *initial symbol* by applying the productions one after another. An example SyGuS problem i.e., a grammar and an associated constraint is given below:

$$\begin{array}{ll} \textsf{grammar} & expr \rightarrow expr \vee expr \ (\textbf{bound} : B) \\ & expr \rightarrow x \mid y \mid \neg x \mid \neg y \\ \textsf{constraint} & (x \Rightarrow \neg y = expr) \wedge (y \Rightarrow \neg x = expr) \end{array}$$

The above grammar has a non-terminal symbol (also the initial symbol) *expr* that represents a disjunction of variables x, y or their negations. Unlike the sketch above, the space of expressions is unbounded, except for a parameter B that bounds the number of times a rule can be used. SKETCH also supports recursive definitions of program spaces that are equivalent to the grammar above; for the rest of the paper we alternate between showing more abstract descriptions of a space of expressions as a grammar, and showing more concrete SKETCH syntax when necessary.

3.2 SyGuS Components for ER-GBF

We are ready to give the grammar and constraints to formulate the **ER-GBF** problem for synthesis and detailed examples to illustrate the SKETCH partial programs, grammars, and constraints involved. It is important to emphasize that these partial programs are built into the tool; the encoding of these grammars as sketches is invisible to the user.

Grammar for ER-GBF. In order to formulate the **ER-GBF** problem in the SyGuS framework, we use a generic Boolean formula grammar ($G_{\mathbf{GBF}}$) defined below:

$$\begin{array}{ll} \textsf{grammar} & G_{\textsf{attribute}} \rightarrow r[\mathsf{A_i}] \approx_{(f,\theta)} s[\mathsf{A_i'}] \\ & i \in [1,n]; f \in \mathcal{F}; \theta \in [0,1] \\ \textsf{grammar} & G_{\mathbf{GBF}} \rightarrow G_{\textsf{attribute}} \ (\textbf{bound} : N_a) \\ & \left.\begin{array}{l} G_{\mathbf{GBF}} \rightarrow \neg G_{\mathbf{GBF}} \\ G_{\mathbf{GBF}} \rightarrow G_{\mathbf{GBF}} \wedge G_{\mathbf{GBF}} \\ G_{\mathbf{GBF}} \rightarrow G_{\mathbf{GBF}} \vee G_{\mathbf{GBF}} \end{array}\right\} (\textbf{bound} : N_d) \end{array}$$

The grammars $G_{\texttt{attribute}}$ and $G_{\mathbf{GBF}}$ represent an attribute-matching rule and a Boolean formula matching rule (**GBF**), respectively. Note that the search space represented by the above grammars is infinite because there are infinitely many real values for $\theta \in [0,1]$. We tackle this by introducing a custom synthesis procedure (Section 4.2). The bounds N_a and N_d make the search space for the Boolean formula finite by bounding the number of attribute-matching rules ($G_{attribute}$) in $G_{\mathbf{GBF}}$ and the number of recursive productions being used, i.e., depth of the expansion of the grammar, respectively.

Constraints for ER-GBF. A candidate selected from the grammar $G_{\mathbf{GBF}}$ can be interpreted as a Boolean formula. Given both positive (**M**) and negative (**D**) examples, the SyGuS constraints are specified as the evaluation of this **GBF** on the provided examples being consistent:

$$\begin{array}{ll} \textsf{constraint} & G_{\mathbf{GBF}}(r_m, s_m) = \mathit{true} \ \forall \ (r_m, s_m) \in \mathbf{M} \\ \textsf{constraint} & G_{\mathbf{GBF}}(r_d, s_d) = \mathit{false} \ \forall \ (r_d, s_d) \in \mathbf{D} \end{array}$$

Partial Programs for ER-GBF. Now let's showcase the partial programs used for the **ER-GBF** problem.

Example 4: Consider the two tables discussed in Example 1. The partial program that represents a Boolean formula matching rule (**GBF**) with N_a attribute-matching rules and N_d depth of grammar expansion is listed below.

```
// e = Example Id
grammar bool attributeRule(int e){
    int i = ??; // Attribute Id
    assert (1 <= i && i <= 5);
    int f = ??; // Similarity Fn Id
    assert (1 <= f && f <= 29);
    double θ = customSynth(i,f);
    return (evalSimFn(e,i,f) >= θ);
}

@depth(N_d)
grammar bool gbfRule(int e, int &A){
    if (??){ A++; return attributeRule(e); }
    else if (??) return ! (gbfRule(e,A));
    else if (??) return gbfRule(e,A) && gbfRule(e,A);
    else return gbfRule(e,A) || gbfRule(e,A);
}

bool matchingRule(int e, int N_a){
  int A=0;
  bool b = gbfRule(e,A);
  assert (A<=N_a);
  return b;
}

constraint void examples(int N_a){
  //Example Id 1 is a positive example
  assert(macthingRule(1,N_a) == true);
  //Example Id 2 is a negative example
  assert(macthingRule(2,N_a) == false);
}
```

In the code above, some functions are annotated with being a `grammar` or a `constraint`. For example, `attributeRule` is a `grammar` function. Since there are 5 aligned attributes, we assert that the values taken by i lie between 1 and 5. Similarly, the candidate space of 29 similarity functions is asserted accordingly. The values for threshold θ are chosen using a custom synthesis procedure. The function `evalSimFn` symbolically represents the evaluation of function f on attribute i of the records from example e (see more details in Section 4.2). Also, `gbfRule` is a `grammar` function with function `attributeRule` being inlined at most N_a times (enforced by a variable A passed by reference) and multiple recursive calls to itself to specify the possible expansion of the grammar. The expansion is bounded by a depth N_d passed as a parameter in the "`@`" annotation. Note that, in SKETCH, each `grammar` function is completely inlined up to the specified depth as a parameter. This results into the *holes* ("??"s) occurring multiple times as well. Each hole inside the `if`'s represents a possible *true* or *false* value.

The `examples` function is a `constraint` that represents the requirement that the resulting rule should work for the positive and negative example.

The SKETCH synthesizer will fill all the holes in the above partial program to synthesize a *complete program*, with a function `matchingRule` that represents a Boolean formula (**GBF**) for entity matching. □

Wrap Up. Next we put together the sample grammars and constraints to show how to obtain a **GBF**.

Example 5: Consider the example in Figure 1. A specific grammar $G^5_{\mathbf{GBF}}$ for representing a Boolean formula matching rule (**GBF**) in this scenario is obtained by using the following in the above definition of the grammar $G_{\mathbf{GBF}}$:

– let $n = 5$ (number of aligned attributes),

– let $\mathcal{F} = \{\texttt{Equal}, \texttt{Levenshtein}, \texttt{Jaccard}\}$,

– let examples be: matching $\mathbf{M} = \{(r_1, s_1), (r_2, s_1)\}$ and non-matching $\mathbf{D} = \{(r_1, s_2)\}$

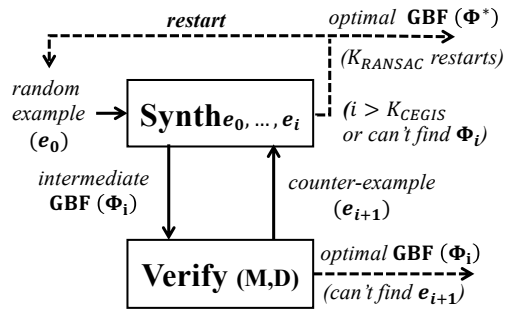


Figure 2: RuleSynth Overview for ER-Synth

Our system gives the synthesizer a table representing the evaluations of each similarity function $f \in \mathcal{F}$ on each attribute $i \in [1, n]$ of every provided example $(r, s) \in \mathbf{E}$ (the function `evalSimFn` in the sketch). The **GBF** $\varphi_2 \vee \varphi_3$ from Example 2 can now be obtained as candidate **GBF** from this grammar $G^5_{\mathbf{GBF}}$. □

3.3 Optimization SyGuS Problem

We are ready to model the **ER-GBF** problem by extending the *Syntax-Guided Synthesis (SyGuS) framework* [2]. We consider two versions of the problem: the *exact problem* and the *optimization problem*. The former corresponds to finding a **GBF** that satisfies all constraints from examples. Unfortunately, such a perfect **GBF** oftentimes does not exist in practice because examples may have errors or the grammar may not be expressive enough to correctly classify all examples. The latter relaxes the condition by discovering a **GBF** of partial satisfaction of constraints based on an optimality metric μ. The **ER-GBF** problem is equivalent to the optimization version of the SyGuS problem defined below.

Optimization SyGuS Problem (ER-Synth). Given grammars and constraints from positive/negative examples, the *optimization SyGuS problem* is to find a candidate **GBF** in the grammar that satisfies a subset of the constraints that maximizes the given optimality metric μ.

As will be seen shortly, although exact SyGuS cannot solve the studied **ER-Synth** problem, it can still be used as a building block in our algorithm (Section 4).

4. SYNTHESIS ALGORITHMS

Existing SyGuS solvers are designed to solve the *exact SyGuS problem*, not the *optimization SyGuS problem* (**ER-Synth**) that would discover a **GBF** that maximizes a given optimization metric. In this section, we describe our approach to solve this problem. We start by giving a naïve solution (Section 4.1) to solve **ER-Synth**. We then incrementally develop our RuleSynth algorithm (Section 4.2).

4.1 A Naïve Solution & Its Limitations

We start by discussing why the *exact SyGuS problem* will fail for the studied **ER-GBF** problem. Given a set **E** of examples, grammars, and constraints, the **GBF** that satisfies all constraints from all examples may not exist. Hence, we shift our goal to finding a **GBF** that satisfies all constraints from a *subset* of the user-provided examples.

A Naïve Solution. Informally, a simple approach would be to choose multiple, random, subsets **S** from all examples **E**, and invoke the Sketch SyGuS solver on each subset in **S**. The solver will only succeed on some of these subsets, but for those that it succeeds on, we can take the best performing **GBF** based on the optimality metric μ evaluated on all examples in **E**.

Algorithm 1: Synthesis Algorithm for **ER-Synth**

input : $\mathbf{E} = \mathbf{M} \cup \mathbf{D}$: Set of examples
$G_{\mathbf{GBF}}(N_a, N_d)$: Bounded **GBF** grammar
$\mathcal{F}$: Library of Similarity Functions
μ : Optimality metric
K_{RANSAC} : Bound on RANSAC restarts
K_{CEGIS} : Bound on CEGIS iterations
output: Φ^* : A **GBF** from $G_{\mathbf{GBF}}(N_a, N_d)$ maximizing μ

```
1  r ← 0
2  Φ* ← true
3  while r < K_RANSAC do                                  // RANSAC loop
4      i ← 0
5      e_0 ← sample (E)
6      E_SYN ← List(e_0)
7      while i < K_CEGIS do                               // CEGIS loop
8          Φ_i ← Synth (G_GBF(N_a, N_d), E_SYN, F)
9          if Φ_i = null then                             // Unsatisfiable Synth
10             break                                      // restart CEGIS
11         Ē_{Φ_i} ← Verify (Φ_i, E)                      // Counter-examples
12         if Ē_{Φ_i} = ∅ then
13             return Φ_i
14         else
15             e_{i+1} ← sample (Ē_{Φ_i})
16             E_SYN ← E_SYN.append(e_{i+1})
17         Φ* = arg max_{Φ∈{Φ*,Φ_i}} μ(Φ, M, D)
18         i ← i + 1
19     r ← r + 1
20 return Φ*
```

Limitations. The naïve solution faces three challenges.

(i) We must choose subsets of **E** in a way that allows us to synthesize a **GBF** with good coverage of the example sets.

(ii) We have to avoid examples that do not lead to a *good* solution, i.e., *sub-optimal* examples that are not matched correctly by any matching rule with high μ value.

(iii) We have to reason about numerical similarity functions and thresholds in a symbolic solver like Sketch, but such reasoning is not supported by existing solvers.

In practice, the above solution will only be able to solve the **ER-Synth** problem with small numbers of examples, since it would not scale due to the huge number of constraints (from examples) imposed by real workloads.

4.2 A Novel Solution (RuleSynth)

We are ready to introduce our new algorithm, designed to overcome the above three limitations. For Limitation (i), we use ideas from the Counter-Example Guided Inductive Synthesis (CEGIS) [39] to perform synthesis from a few examples. For Limitation (ii), we are inspired by the Random Sample Consensus (RANSAC) [21] to avoid sub-optimal examples. For Limitation (iii), we add a custom synthesizer for finding a numerical threshold within the symbolic solver.

Algorithm. The algorithm, referred to as RuleSynth, is presented in Algorithm 1 with an overview in Figure 2. It has two loops. The outer (RANSAC) loop (lines 3-19) picks

random samples to bootstrap the synthesis algorithm. In each iteration, given a sample (line 5), it starts with the **Synth** routine (line 6). It then invokes the inner (CEGIS) loop (lines 7-18). In each iteration, it first synthesizes a **GBF** (line 8). If it cannot find a satisfiable **GBF**, it will restart (lines 9-10); otherwise, it will **Verify** to find counter-examples (line 11). Either there is no counter-example so the process will terminate (lines 12-13), or a randomly selected counter-example will be added to be considered in the next CEGIS iteration (lines 14-16). The current best **GBF** will be re-calculated (line 17). Finally, the algorithm will return a **GBF** (line 20).

We explain different parts of the algorithm below.

Customized Synth Routine: We begin with the core **Synth** routine (line 8) that solves the exact SyGuS problem, i.e., it finds a candidate **GBF** from the bounded grammar $G_{\mathbf{GBF}}(N_a, N_d)$ that satisfies all the constraints arising from examples in $\mathbf{E}_{\text{SYN}}$. Note that the grammar $G_{\mathbf{GBF}}$ is built on top of attribute indices $i \in [1, n]$, similarity functions $f \in \mathcal{F}$ and thresholds $\theta \in [0, 1]$. To reason about similarity functions and numerical thresholds symbolically, as a first step, we round the similarity function outputs to a fixed decimal precision ($\mathbf{T}$) and make the space of thresholds discrete. In our experiments, we use $\mathbf{T} = 3$. Secondly, to reason about numerical functions in SKETCH, we enumerate the function evaluations on all possible values that can be obtained from aligned attributes in the examples. For example, if we have just one example $e_1 = (r, s)$ with

$$r \equiv \{\mathsf{name}\text{`=C. Zeta-Jones'}, \mathsf{gender}\text{='F'}\}$$
$$s \equiv \{\mathsf{name}\text{='Catherine Zeta-Jones'}, \mathsf{sex}\text{='F'}\}$$

then we evaluate the `Jaccard` similarity function on aligned attributes and provide the following table `evalSimFn` to SKETCH solver:

example id	function	matched attribute	evaluation
e_1	Jaccard	name|name	0.5
e_1	Jaccard	gender|sex	1.0

This will help SKETCH evaluate the candidate **GBF**s as required in the next step. Finally, we add a custom procedure inside the SKETCH solver that, given the symbolic decisions made for values of attributes and the similarity function, will find a numerical threshold that separates the chosen positive ($\mathbf{E}_+$) and negative examples ($\mathbf{E}_-$), if one exists. This procedure will be called multiple times while the symbolic SKETCH synthesizer finds the overall **GBF** structure. Its pseudocode is given in Algorithm 2.

Synthesis from a few Examples (CEGIS). We use ideas from the Counter-Example Guided Inductive Synthesis (CEGIS) [39] approach to build an iterative algorithm that has two phases: **Synth** (line 8) and **Verify** (line 11). The idea is to iteratively synthesize a **GBF** that works for a small set of examples and expand this set in a smart manner by adding only those examples that are currently not being handled correctly by the synthesized **GBF**.

For example, consider Figure 1 with matching examples $\mathbf{M} = \{(r_1, s_1), (r_4, s_2), (r_2, s_1)\}$ and non-matching examples $\mathbf{D} = \{(r_1, s_2), (r_4, s_1)\}$. Suppose the algorithm picks (r_1, s_1) as the first example and **Synth** returns the function $\Phi_0 =$ `Equal[name]` $\geqslant 1.0$. **Verify** tries this function on all examples in $\mathbf{M} \cup \mathbf{D}$ and randomly picks (r_4, s_1) as the counter-example, i.e., an example which is not correctly matched by the function Φ_0 since the names are not equal for (r_4, s_1). It would then add this counter-example to the set $\mathbf{E}_{\text{SYN}}$ and start the next CEGIS iteration. In this iteration **Synth** may now return the function $\Phi_1 =$ `Jaccard[name]` $\geqslant 0.4$, which matches all examples correctly.

Algorithm 2: Custom Synthesis inside SKETCH

```
input : f : Chosen similarity function
        a : Matched attribute Id
        E+ : Examples chosen to be positive
        E− : Examples chosen to be negative
output: exists : Does a valid threshold exist
        θ : A valid threshold separating E+ & E−
1  θ_atmost ← 1.0
2  for e ∈ E+ do
3      θ_atmost = min(θ_atmost, evalSimFn(e, a, f))
4  θ_atleast ← 0.0
5  for e ∈ E− do
6      θ_atleast = max(θ_atleast, evalSimFn(e, a, f))
7  if θ_atleast < θ_atmost then
8      exists ← true
9      θ ← (θ_atleast + θ_atmost) / 2
10 else
11     exists ← false
```

At iteration i, **Synth** uses the currently available examples $\mathbf{E}_{\text{SYN}} = \{e_0, e_1, \ldots, e_i\}$ and solves the Exact SyGuS problem with SKETCH to find a **GBF** Φ_i from the bounded grammar $G_{\mathbf{GBF}}(N_a, N_d)$ that correctly handles all the examples in $\mathbf{E}_{\text{SYN}}$. **Verify**, on the other hand, considers the full set of examples $\mathbf{E} = \mathbf{M} \cup \mathbf{D}$ and finds the *counter-example* subset $\overline{\mathbf{E}}_{\Phi_i} \subset \mathbf{E}$, which contains examples $e \in \mathbf{E}$ such that $\Phi_i(e) = \mathit{false}$ if $e \in \mathbf{M}$ and $\Phi_i(e) = \mathit{true}$ if $e \in \mathbf{D}$. In other words, it identifies examples that are incorrectly handled by Φ_i. A counter-example e_{i+1} chosen randomly from $\overline{\mathbf{E}}_{\Phi_i}$ is added to the set $\mathbf{E}_{\text{SYN}}$ to be considered in the next **Synth** phase. The process continues until either **Synth** is unable to find a **GBF** for the current set of examples or until it has performed K_{CEGIS} (*CEGIS cutoff*) iterations. If **Verify** cannot find any counter-example (i.e., $\overline{\mathbf{E}}_{\Phi_i} = \varnothing$) then the algorithm terminates and outputs Φ_i as the optimal **GBF** since it correctly handles all examples in $\mathbf{E}$.

Synthesis with Sub-optimal Examples (RANSAC). We use ideas from the Random Sample Consensus (RANSAC) [21] approach and build a loop on top of the CEGIS loop to restart it multiple times with different initial random examples (e_0). The idea is that if the provided example set contains a small number of sub-optimal examples, then multiple runs are more likely to avoid them. Note that some examples individually may not be sub-optimal, i.e., the algorithm may still find a good **GBF** after choosing them in the CEGIS loop. Instead, certain larger subsets of examples may correspond to conflicting constraints on the **GBF** grammar and constitute sub-optimality only when all examples in that subset are chosen together. Both the randomness in **sample** routine and the RANSAC restarts help avoid choosing all such points together. Before restarting CEGIS, if the number of restarts reaches K_{RANSAC} (the RANSAC cutoff) then the algorithm terminates and outputs the *best* **GBF** Φ^* seen across all CEGIS and RANSAC iterations w.r.t. the optimality metric μ.

5. SYNTHESIS OPTIMIZATIONS

In this section, we describe optimizations to RuleSynth.

5.1 Grammar: Conciseness and Null Values

We use the power of synthesis to control the structure of the **GBF** and provide a concise formula as the output. Note that these techniques also help us avoid over-fitting to the provided examples since our **GBF**s are as small as possible.

Handling Null values in $G_{\mathbf{GBF}}$: Null (missing) values are problematic because we cannot know whether two records match on some attribute A if one record has a Null value for A. Rather than assuming that such records do not match (as was done in previous work), we learn different rules for the Null and noNull case. We specify a new grammar production in $G_{\mathbf{GBF}}$ for deriving **GBF**s that capture this intuition:

$$\begin{array}{lll} \textsf{grammar} & G_{\mathbf{GBF}} \rightarrow & \textbf{if } (\approx(i, \texttt{noNulls}, 1.0)) \\ & & \textbf{then } (G_{\mathbf{GBF}}) \textbf{ else } (G_{\mathbf{GBF}}) \\ & & i \in [1, n] \end{array}$$

This production says that if there are no nulls in the matching attributes in a pair of records, then we should use one **GBF**; otherwise we should use a different **GBF**. This makes it possible for the synthesizer to quickly find rules similar to example φ_4 (Section 1). Note that this addition does not affect the expressibility of the grammar and is purely for making the grammar $G_{\mathbf{GBF}}$ and the synthesis process more targeted towards databases with large numbers of nulls.

Incremental Grammar Bounds To make sure that the generated rules are small and concise, RuleSynth iteratively adjusts the grammar bound on number of attribute-matching rules (N_a) as it runs, starting with rules of size 1 and growing up to N_a, so that it prefers smaller rules when they can be found. To be more precise, we introduce the following loop in Algorithm 1 replacing line 8:

Procedure Incremental Grammar Bounds

```
1 n_a ← 1                          // attribute-matching rules bound n_a
2 while n_a ⩽ N_a do
3 |  Φ_i ← Synth(G_DNF(n_a, N_d), E_SYN, F)
4 |  if Φ_i = null then                    // Unsatisfiable Synth
5 |  |  n_a ← n_a + 1                        // try larger n_a
6 |  else
7 |  |  break
```

RuleSynth uses an optimized version of this loop where in CEGIS iteration $i \geqslant 1$, the initial value of n_a is set to the value of n_a used to synthesize Φ_{i-1} in the previous CEGIS iteration (instead of starting with $n_a = 1$). Since the set of examples being considered in iteration i is a superset of examples considered in iteration $i-1$, if for any n_a **Synth** could not find a **GBF** in iteration $i-1$ then for the same n_a it will not be able to find a **GBF** that matches all the examples in iteration i.

5.2 Sampling: Bias in Picking Examples

In CEGIS iteration i, the RuleSynth algorithm tries to primarily choose an example that is currently not being matched correctly. This guides the resulting **GBF** towards higher accuracy on the example set by making more and more examples match correctly. On top of this, RuleSynth picks the best **GBF** that maximizes μ across all CEGIS and RANSAC iterations. For optimality metrics like $\mu_{\text{F-measure}}, \mu_{\text{precision}}, \mu_{\text{recall}}$ it is important to focus on finding **GBF**s that maximize the number of positive examples being matched correctly. Note that if the set of examples is largely only negative examples then the likelihood of most of the chosen examples being negative is high. This may result in the algorithm missing certain positive examples for smaller CEGIS cutoffs (K_{CEGIS}) and thereby finding a solution with possibly lower μ even when the accuracy is high. Hence, in RuleSynth we eliminate this bias based on the actual distribution of positive and negative examples and replace it with a 50-50 chance of choosing a positive or negative example, i.e., the **sample** routine (line 5 in Algorithm 1) is modified as shown in Appendix B.

5.3 Algorithm Optimizations

We describe some additional optimizations that make the synthesis process more efficient. In our experiments, we run RuleSynth with and without these optimizations.

BestThresholds: Pre-determining *Good* Thresholds. BestThresholds restricts the space for thresholds in attribute-matching rules in the $G_{\mathsf{attribute}}$ grammar to just one pre-determined value instead of the whole interval $[0, 1]$. The idea is to automatically choose the threshold for a given attribute independently of the other attributes included in a **GBF**. Note that this optimization substantially reduces the search space, but also can produce inferior **GBF**s, so it presents a tradeoff. To apply the optimization, we modify the grammar as shown below:

$$\begin{array}{ll} \textsf{grammar} & G_{\mathsf{attribute}} \rightarrow r[\mathsf{A_i}] \approx_{(f,\theta)} s[\mathsf{A'_i}] \\ & i \in [1, n]; f \in \mathcal{F}; \theta = \mathsf{bestTh}(i, f) \end{array}$$

For each attribute $i \in [1, n]$ and similarity function $f \in \mathcal{F}$, BestThresholds chooses the value of the threshold θ such that when using the attribute-matching rule $\varphi' = \approx(i, f, \theta)$ the metric $\mu\left(\varphi', \mathbf{M}, \mathbf{D}\right)$ attains the maximum possible value:

$$\mathsf{bestTh}(i, f) = \underset{\theta \in [0,1]}{\arg\max} \left\{ \mu\left(\varphi', \mathbf{M}, \mathbf{D}\right) \right\}$$

Intuitively, for each attribute and matching function BestThresholds finds the threshold θ that maximizes its accuracy on the positive and negative examples when considering this attribute, function and θ alone as a matching rule. To enable BestThresholds to evaluate $\mathsf{bestTh}(i, f)$, we make the space for θ finite from $[0, 1]$ by using the theorem from [41] that generates these candidate thresholds from the set of positive examples $\mathbf{M}$. Specifically, the new space is $\Theta(i, f) = \left\{ f(r[\mathsf{A_i}], s[\mathsf{A'_i}]) \mid (r, s) \in \mathbf{M} \right\}$. BestThresholds finds such a threshold θ by iterating over all values in $\Theta(i, f)$ and picking the threshold that maximizes μ. BestThresholds uses an optimized version of this search that iterates over sorted evaluations of the function f on attribute i of examples from $\mathbf{M} \cup \mathbf{D}$ and cumulatively computes μ by maintaining the true/false positives, and true/false negatives across iterations.

As an example, consider the function `Jaccard[name]` applied to the data in Figure 1; suppose a threshold $\theta = 0.5$ correctly matches all positive examples, but also matches one negative example (r_3, s_3), whereas a threshold $\theta = 1.0$ correctly matches only (r_1, s_1) as well as (r_3, s_3). If our metric is F-measure, then we would prefer the threshold 0.5, and would use this for all rules involving `Jaccard[name]`.

SynthComp: Composition of Discovered Rules. The SynthComp optimization efficiently produces larger **GBF**s

from smaller **GBF**s. SynthComp supplements RuleSynth with an additional step at the end, where it synthesizes a general boolean formula (**GBF**) using $B_{\text{SynthComp}}$ **GBF**s from a set of top $K_{\text{SynthComp}}$ **GBF**s collected across all CEGIS and RANSAC iterations while maximizing the metric μ. That is, if RuleSynth found 3 functions $\varphi_1, \varphi_2, \varphi_3$ with metric μ being $0.82, 0.77, 0.64$, respectively, then it will look at all boolean combinations of these **GBF**s and come up with the best one, say, $(\varphi_1 \wedge \varphi_3) \vee \varphi_2$ with metric $\mu = 0.87$ that is better than all three functions individually. Our implementation of SynthComp uses ideas from truth-table based synthesis of Boolean circuits [17] and takes less than 5 minutes for $B_{\text{SynthComp}} \leqslant 4$ and $K_{\text{SynthComp}} \leqslant 10$. Note that a larger value of $B_{\text{SynthComp}}$ would make this time grow substantially, but it will also make the final rules large and uninterpretable. As will be seen in the experiments (Section 6), these values ($K \leqslant 10, B \leqslant 4$) work well in RS-SynthComp method for all datasets and lead to effective and interpretable rules, which are generated in reasonable time. Using SynthComp also involves a tradeoff between conciseness (number of attribute-matching rules in **GBF**) and performance (metric μ).

6. EXPERIMENTS

We now study the quality of the **GBF**s produced by RuleSynth in terms of both interpretability and effectiveness. The key questions we answer with our evaluation are:

1. How do our rules compare in interpretability and accuracy to other interpretable models? (Exp-1, Exp-2)
2. How do they compare in accuracy to expert-provided rules? (Exp-3)
3. How do they compare in accuracy to non-interpretable models, such as SVMs? (Exp-4)
4. How do we perform when using limited training data? (Exp-5)
5. Can RuleSynth discover rules in reasonable amounts of time? (Exp-6)

6.1 Experimental Setup

Datasets. Table 1 shows the datasets used in our evaluation. Four datasets are real-world data and Febrl is synthetic. Datasets Cora and Febrl have one relation, while the others have two relations with aligned schemas. Positive examples for every dataset are also given. Because there are many more non-matching values than matching values, we needed a reasonable subset of non-matching pairs (negative examples). To ensure that negative examples are quite different from each other, we took the Cartesian product of the relations and pruned pairs with high Jaccard tri-gram similarity values [24, 25]. We varied the similarity threshold across datasets to control the number of negative examples.

Table 1 also shows the average number of record pairs with at least one null value. These numbers show the importance of using the custom noNulls function in a formula because noNulls in the **if** condition enables the synthesizer to find smaller rules for the noNulls (then) vs nulls (else) cases. Some datasets have a skewed distribution of nulls across attributes, e.g., for **DBLP-Scholar**, the attribute year has around $40K$ nulls, whereas title and authors have 0.

Inputs for ER-GBF problem (Section 2.2). In the following, we use F-measure as the metric to be optimized. We present results for optimizing Precision and Recall in Appendix C. We use a set of 28 similarity functions that were also used in the SIFI project [41]. This set includes functions from the Simmetrics library [1] and functions implemented by authors of SIFI. We also treat Equal and `noNulls` as two similarity functions that evaluate to 0 or 1.

	#Matching Pairs	#Record Pairs	#Attr	Avg #nulls per Attr
D_C	14,280	184,659	9	92,955(50%)
D_{AG}	1,300	97,007	4	22,583(23%)
D_{LF}	6,048	341,244	10	99,629(29%)
D_{DS}	5,347	112,839	4	12,685(11%)
D_F	400	239,999	13	41,704(17%)
D_C =**Cora**, D_{DS} =**DBLP-Scholar**, D_F =**Febrl** D_{AG} =**Amazon-GoogleProducts**, D_{LF} =**Locu-FourSquare**				

Table 1: Dataset statistics

Comparisons with State-of-the-Art ML approaches. We compared the basic and the optimized variants of RuleSynth against existing machine learning (ML) approaches, such as support vector machines (SVM) [41] and Random Forests [22]. Both ML methods convert entity matching into a binary classification problem to discover matching and non-matching record pairs.

While the output from SVM lacks logical interpretability, a tree traversal of decision trees can be interpreted as a boolean formula with multiple **DNF** clauses arising from traversal of paths that lead to positive classification. However, the output of Random Forests are tedious to interpret because: (1) the output has tens to hundreds of trees that are aggregated to make the final decision (2) each decision tree has a large depth resulting into thousands of nodes, making them hard to interpret individually. Since a similar aggregation mechanism like bagging can be used over RuleSynth as well, we focus on (2) and compare our results with a single decision tree from [22], looking at trees with variable depths. We show that the logical expressions from even a deep decision tree result in many atoms when represented as a boolean formula, while shallow decision trees are easier to interpret but perform poorly on F-measure. In contrast, our methods result in compact, interpretable rules that also provide good F-measure scores.

Comparisons with Rule-based learning approaches. We evaluate RuleSynth against a heuristic-based approach, SIFI [41], which searches for optimal similarity functions and thresholds for the attribute comparison given a DNF grammar provided by a human expert. In contrast, the **GBF**s are automatically discovered by RuleSynth without any expert-provided grammar.

Implementation. All experiments were run on a machine with Ubuntu 14.04 OS, 32 GB RAM and 16-core 2.3 GHz CPU. We implemented RuleSynth in Python 2.7.6 as scripts that interacted with the Sketch synthesis tool (written in Java and C++). We implemented SVM in Python using the scikit library and libsvm [8]. The two baseline approaches, i.e., SIFI and decision trees, were obtained from the authors of [41] and [22], respectively. SIFI was implemented in C++, and the random forest entity matcher was coded in Java using the Weka library [37].

Techniques and parameters. For Decision Trees, we use depths 3, 4 and 10 (the default configuration in Weka). We use an SVM classifier with linear kernel (*linearSVC* implementation from scikit) with balanced class-weights as a low-

effort configuration for optimizing F-measure [29]. We ran SIFI with default configurations and grammars generated by database experts as input.

We use three variants of our algorithm: (1) the basic CEGIS+RANSAC based RuleSynth (Section 4.2), (2) RS-BestTh that uses the BestThresholds optimization (Section 5.3), and (3) RS-SynthComp that uses the SynthComp optimization (Section 5.3). For all variants, we have the following parameters with their respective default values: (i) The depth of the grammar $N_d = 4$, which is enough to represent formulas with at most 15 atoms; (ii) A high $K_{\text{CEGIS}} = 1000$ with a timeout of 15 minutes per CEGIS iteration so that the CEGIS loop runs until it finds a set of examples for which Sketch cannot synthesize a valid rule or it times out and picks the best rule obtained till then; (iii) The bound $K_{\text{RANSAC}} = 5$ to restart CEGIS 5 times and explore different underlying sets of examples. (iv) The number of attribute-matching rules N_a: for RuleSynth and RS-BestTh, we set $N_a = 8$ so that it is comparable with the number of atoms in a decision tree of depth 3; For RS-SynthComp we use $N_a = 5$ and combine $B_{\text{SynthComp}} = 3$ rules out of $K_{\text{SynthComp}} = 10$ rules (Section 5.3) to generate a composite **GBF** so that in total #-atoms is bounded by 15 and is comparable with #-atoms in a decision tree of depth 4. The user can modify these parameters to tradeoff quality for training time, but for all our experiments, we used only the default values. In general, adding more time should not affect the results drastically but may improve them slightly.

Performance Evaluation. We performed K-fold cross-validation (for K=5) on each of the datasets used, where we divided the data into K equal fractions (folds) randomly and performed K experiments. In each experiment one of the K folds was the test set while the remaining K-1 folds were training. We report the average F-measure obtained across all folds on the test sets as the performance metric (Figure 4). We also present the corresponding average precision and recall values (Figures 8 and 9 in the Appendix). Note that we use the same folds for each technique we compare. For limited training data experiments (*Exp-5*), we randomly sample a fraction $\frac{1}{K}$ of examples (for multiple values of $K = 100, 40, 20, 10, 7, 5$) and use it for training. Each fraction $\frac{1}{K}$ corresponds to a different percentage $P\%$ of examples (i.e., $P = 1, 2.5, 5, 10, 14.3, 20$). We use the rest $(100 - P)\%$ of the examples for testing, and, we train and test on 100 such randomly selected sets for each percentage P. We report the average test-set F-measure and size of matching rules obtained across all 100 runs (with 99% confidence intervals) for each dataset.

6.2 Experimental Results

Exp-1: Interpretability. By interpretability, we mean how readable and understandable the discovered rules are. Thus, we measure it as being inversely proportional to the number of *attribute-matching rules* (or atoms) present in the rule. In other words, interpretability is defined as the number of atomic similarity function comparisons with a threshold $\approx(i, f, \theta)$ in the formula representing the rule. For clarity, we represent atoms or attribute-matching rules as $\big(fn[attr] \geqslant \theta\big)$, where fn is the name of the applied similarity function, $attr$ is the name of the matched attribute, and θ is the corresponding threshold, e.g., EditDistance[title] $\geqslant$ 0.73 is a valid atom. The rationale behind this interpretability definition is that fewer atoms make a rule easy to read and the presence of logical operators, such as negation and conditionals such as "if", also contributes to better understand a rule. This supports the idea that a complex **DNF** is less interpretable than a semantically equivalent but concise **GBF**. In short, statistical ML methods with weights and function parameters (e.g., SVM), and logical structures with hundreds or thousands of atoms (e.g., decision tree with depth 10) are not human interpretable. Methods with clear logical structures, such as **GBF**s, **DNF**s, and decision trees with depths 3 and 4, are human interpretable.

Below, we present two **GBF**s, φ_{synth} and φ_{tree}, obtained by using RuleSynth and decision trees of depth 3, respectively, on record pairs from the Cora dataset. We obtained both **GBF**s on the same training set as the best rules. These rules result in average F-measures of 0.83 (φ_{synth}) and 0.77 (φ_{tree}) on test data. The **GBF** φ_{synth} demonstrates the conciseness of formulas generated by RuleSynth as compared to φ_{tree}, as φ_{synth} has only 6 atoms whereas φ_{tree} has 12 atoms. Also note that the RuleSynth rules include if/then/else clauses that allow them to be more compact that the DNF-based rules the decision tree produces. The rules for other datasets are presented in Appendix B.

φ_{synth} :
(ChapmanMatchingSoundex[author] $\geqslant 0.937$
$\bigwedge$ **if** noNulls[date] $\geqslant 1$
then CosineGram$_2$[date] $\geqslant 0.681$
else NeedlemanWunch[title] $\geqslant 0.733$) $\bigvee$
(EditDistance[title] $\geqslant 0.73$
$\bigwedge$ OverlapToken[venue] $\geqslant 0.268$)

φ_{tree} :
(OverlapGram$_3$[title] $\geqslant 0.484$
$\bigwedge$ MongeElkan[volume] $\geqslant 0.429$
$\bigwedge$ Soundex[title] $\geqslant 0.939$) $\bigvee$
(OverlapGram$_2$[pages] $\geqslant 0.626$
$\bigwedge$ MongeElkan[volume] $\geqslant 0.429$
$\bigwedge \neg$ (Soundex[title] $\geqslant 0.939$)) $\bigvee$
(ChapmanMeanLength[title] $\geqslant 0.978$
$\bigwedge \neg$ (OverlapGram$_3$[author] $\geqslant 0.411$)
$\bigwedge \neg$ (MongeElkan[volume] $\geqslant 0.429$)) $\bigvee$
(CosineGram$_2$[title] $\geqslant 0.730$
$\bigwedge$ OverlapGram$_3$[author] $\geqslant 0.411$
$\bigwedge \neg$ (MongeElkan[volume] $\geqslant 0.429$))

Figure 3 shows the interpretability results for all datasets. We observe that our algorithm produces more interpretable rules, i.e., with fewer atoms, than decision trees with depths 3 and 4 for all datasets. In particular, RS-BestTh produces more interpretable rules than decision trees with depth 4 for all datasets and decision trees with depth 3 for 3 out of 5 datasets. We also see that RuleSynth produces rules up to (i) three times more interpretable than decision trees with depth 3 (see dataset `Amazon-GoogleProducts` and `Febrl`) and (ii) eight times more interpretable than decision trees with depth 4 (see dataset `Amazon-GoogleProducts`). The third variant RS-SynthComp produces rules with more atoms but still has better interpretability than decision trees with depth 4. Moreover, as we will see in Exp-2, the rules produced by RS-SynthComp are more effective than decision trees with both depth 3 and 4.

We also observe that the number of atoms increases exponentially with the depth of the decision trees i.e., the deeper is the tree, the less interpretable the corresponding rules are. For example, it is nearly impossible to interpret decision trees of depth 10 with thousands of atoms. These

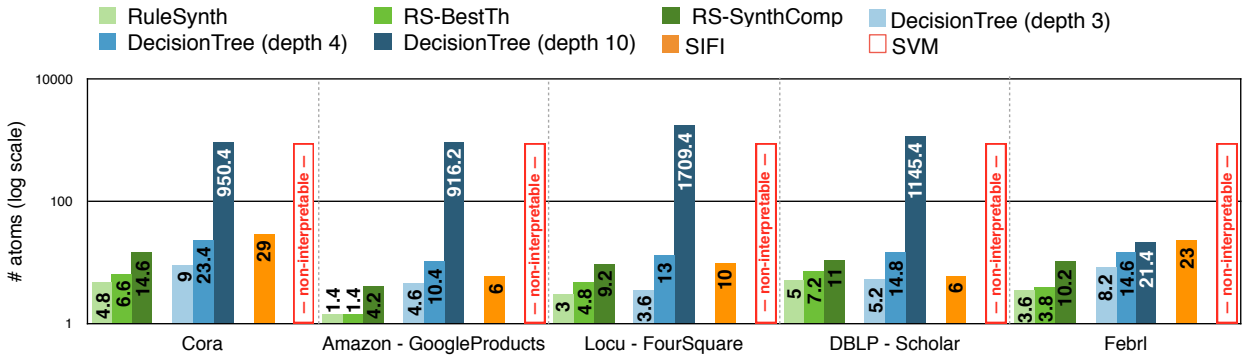


Figure 3: Interpretability results for 5-folds experiment (80% Training and 20% Testing Data)

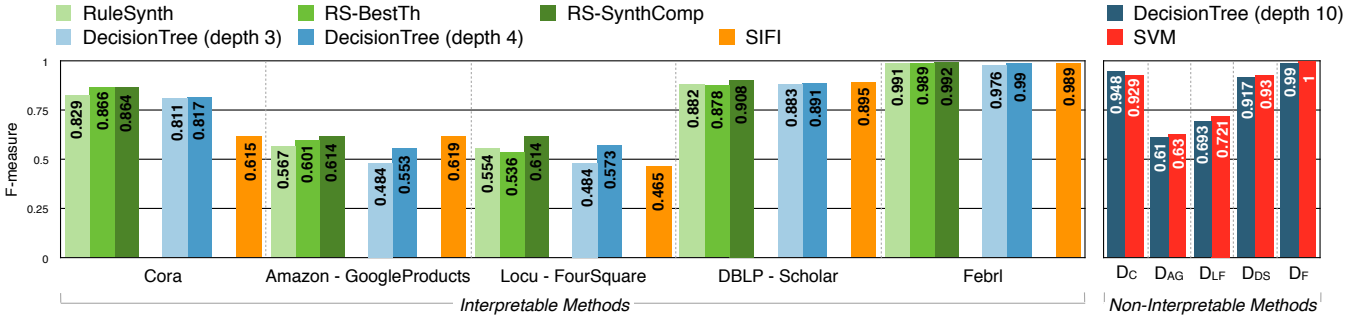


Figure 4: Effectiveness results for 5-folds experiment (80% Training and 20% Testing Data)

results showcase the power of program synthesis to generate concise formulas from a rich grammar, yielding compact and more interpretable rules.

Exp-2: Effectiveness vs. Interpretable Decision Trees. We now evaluate the effectiveness of rules generated by our algorithms against the ones found by decision trees. As mentioned before, we use the average F-measure across 5 folds as the effectiveness metric.

Figure 4 shows the average F-measures for different techniques. We observe that RuleSynth and RS-BestTh achieve a higher F-measure than decision trees with depth 3 for all datasets, except for `DBLP-Scholar` where the F-measures are comparable. Decision trees achieve higher F-measures when increasing their depth from 3 to 4 for all datasets. However, applying SynthComp optimization on RuleSynth still results into higher F-measures than decision trees with depth 4 on all data sets.

Moreover, as we saw in Figure 3, each of RuleSynth, RS-BestTh, and RS-SynthComp produces more interpretable rules than decision trees with depth 4 for all datasets. From figures 3 and 4, we conclude that decision trees can get better F-measures by increasing their depth but this comes at a significant sacrifice to their interpretability. In contrast, RuleSynth can get better F-measures by applying BestThresholds and SynthComp optimizations while not sacrificing on interpretability as much. For example, for the `Amazon-GoogleProducts` dataset, increasing the depth of decision tree from 3 to 4 increases the F-measure from 0.484 to 0.553 while the average number of atoms increases from 4.6 to 10.2. In contrast, the SynthComp optimization increases the F-measure from 0.567 to 0.614 while increasing the average number of atoms from 1.4 to 4.2.

Exp-3: Effectiveness vs. Expert-Provided Rules. To further demonstrate the effectiveness of **GBF**s produced by RuleSynth and its variants, we compare RuleSynth with SIFI [41]. SIFI requires experts to provide a **DNF** template from experts as an input and completes it to generate a rule. In contrast, RuleSynth discovers rules automatically, reducing the effort needed from an expert.

Figure 4 shows that RuleSynth and its variants perform better than SIFI for all datasets except `Amazon-GoogleProducts` where the F-measures are comparable (0.614 and 0.619). In contrast with SIFI which employs a heuristic to search through a smaller space of rules, RuleSynth searches through a huge space of generic **GBF**s. This allows us to discover various corner cases that can be sometimes missed by an expert-provided expression. In addition, as shown in Figure 3, RuleSynth generates **GBF**s that are more concise (and thus interpretable) than the **DNF**s produced by SIFI for all datasets. Please refer to Appendix D for more results.

Exp-4: Effectiveness vs. Non-interpretable Methods. We now compare RuleSynth and its variants with two ML algorithms: (1) Decision trees with depth 10 (the default in Weka library) and (2) SVM. In Figure 4, we observe that all the three variants of RuleSynth achieve smaller F-measure values than both the ML algorithms on an average. Still, we observe that SynthComp achieves quite comparable F-measures, with the F-measure difference between the ML best algorithm and SynthComp being 0.09, 0.02, 0.11,

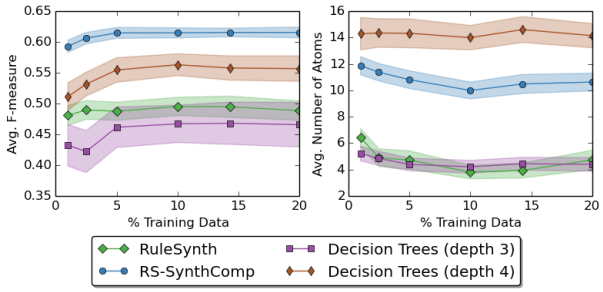


Figure 5: Locu-Foursquare (100 runs with 99% confidence intervals on the means in the shaded regions)

0.02, and 0 for each of the five data sets. However, the efficiency of these ML algorithms comes at a high price. We see in Figure 3 that these two ML algorithms are not interpretable: (i) SVM does not produce rules, and (ii) decision trees with depth 10 yield rules with around $1K$ atoms for all datasets, which are also impossible to interpret for a human.

Exp-5: Limited Training Data. Figure 5 shows the comparison between interpretable decision trees and RuleSynth variants on Locu-Foursquare dataset with different percentages (1% to 20%) of training data. Figure 11 in Appendix E provides complete results comparing RuleSynth and its variants to Decision Tree methods on other datasets. We compare Decision Trees (depth 3) with RuleSynth since they both produce rules with smaller sizes, and, Decision Trees (depth 4) with RS-SynthComp since they both produce interpretable rules with larger sizes. Both RuleSynth and RS-SynthComp outperform Decision Trees of depth 3 and 4, respectively, in effectiveness (higher F-measure) on all datasets. At the same time, RS-SynthComp generates more interpretable (lower number of atoms) rules than Decision Trees (depth 4). RuleSynth and Decision Trees (depth 3) both generate small and interpretable rules (2-7 atoms on average). RuleSynth generates smaller rules for 3 out of 5 datasets, has similar interpretability for `Locu-Foursquare`, and, generates slightly larger rules for `DBLP-Scholar`. RS-SynthComp is the most effective method for generating interpretable rules (2-14 atoms on average) with limited training data on all datasets.

Exp-6: Efficiency. RuleSynth and its variants provide the flexibility for users to control how much the algorithm should explore in the CEGIS loop (bound K_{CEGIS}, time-limit, grammar bounds) and how many times it should restart (bound K_{RANSAC}). More exploration and restarts result in better rules but also take more time. Please refer to Figure 12 in Appendix F and Appendix E for details.

7. RELATED WORK

Machine Learning-Based Entity Matching. Most current solutions are variants of the Fellegi-Sunter model [19], where entity matching is treated as a *classification* problem. Such approaches include SVM based methods [5], decision tree based solutions [9, 22], clustering based techniques [14, 31], and Markov logic based models [36].

However, as pointed out in recent work in the ML area [26], one of the most important obstacles to deploying predictive models is the fact that humans do not understand and trust them because such models cannot justify or explain their choices [27]. Even when ML classifiers produce results with good accuracy, industrial systems rely on rules to be able to trace errors and adjust the system quickly [6].

Rule-Based Entity Matching. Declarative entity matching rules are normally desirable to end users. Such rules are also popular in the database community since they provide great opportunities for improving the performance at execution time, such as those studied in [3, 15, 23]. However, these approaches typically assume that EM rules are given by domain experts, which, in practice, it is a hard problem.

Closer to our work is [41], which automatically discovers similarity functions and their associated thresholds by assuming that the rule is given as a **DNF**. In contrast to it, our approach can automatically discover the optimal and more expressive **GBF** based rules with corresponding similarity functions and thresholds, which provides an *end-to-end* solution for generating entity matching rules. An unsupervised learning method for link discovery configuration for the Web of Data has also been proposed [30]. This is based on three assumptions: (1) no duplicate URIs exist in a dataset; (2) two datasets have a strong degree of overlap; and (3) a meaningful similarity function returns values close to 1.0 for matching pairs. They rely on the above three assumptions to compute indicators of "good characteristics", i.e., to simulate user examples. Although the solution works well for the Web of Data, the above three assumptions do not hold in the general entity matching settings that we study. Moreover, they produce a weighted combination of attribute similarities using an average or a maximum function that is arguably harder to interpret than a boolean expression with if-then-else conditions.

Active Learning and Crowdsourcing. Since good and sufficient training dataset is always hard to get in practice, a natural line of study is how to actively involve users in verifying ambiguous tuple pairs, *a.k.a.* active learning in entity matching [22, 28, 31]. Also, due to the popularity of crowdsourcing platforms, there have also been efforts of leveraging crowd workers for entity matching problems [20, 22, 40].

The above works are essentially orthogonal, but complementary, to the rule generation problem studied in this work.

Program Synthesis. Program synthesis, or programming by example, has shown great promise for database related problems, such as data transformation [33], social recommendations [13], and translating imperative code into relational queries [12]. In this paper, we explore the potential of program synthesis in supporting the generation of EM rules.

8. CONCLUSION

We presented how to synthesize EM rules from examples. Given a high level specification of rules and some examples, our solution uses program synthesis to automatically synthesize the **GBF** rules for EM. We presented optimizations based on RANSAC and CEGIS to improve the effectiveness of the optimizer, and showed that our solution produces rules which are both concise and easy to interpret for end-users, while matching test examples with accuracies that are comparable with the state-of-the-art solutions, despite the fact that those solutions produce non-interpretable results.

9. REFERENCES

[1] Simmetrics: A java library of similarity and distance metrics. https://github.com/Simmetrics/simmetrics.

[2] R. Alur, R. Bodík, E. Dallal, D. Fisman, P. Garg, G. Juniwal, H. Kress-Gazit, P. Madhusudan, M. M. K. Martin, M. Raghothaman, S. Saha, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *Dependable Software Systems Engineering*, pages 1–25. 2015.

[3] O. Benjelloun, H. Garcia-Molina, D. Menestrina, Q. Su, S. E. Whang, and J. Widom. Swoosh: a generic approach to entity resolution. *VLDB J.*, 18(1), 2009.

[4] P. A. Bernstein, J. Madhavan, and E. Rahm. Generic schema matching, ten years later. *PVLDB*, 4(11):695–701, 2011.

[5] M. Bilenko and R. J. Mooney. Adaptive duplicate detection using learnable string similarity measures. In *SIGKDD*, pages 39–48, 2003.

[6] P. S. G. C., C. Sun, K. G. K., H. Zhang, F. Yang, N. Rampalli, S. Prasad, E. Arcaute, G. Krishnan, R. Deep, V. Raghavendra, and A. Doan. Why big data industrial systems need rules and what we can do about it. In *SIGMOD*, pages 265–276, 2015.

[7] L. F. M. Carvalho, A. H. F. Laender, and W. M. Jr. Entity matching: A case study in the medical domain. In *Proceedings of the 9th Alberto Mendelzon International Workshop on Foundations of Data Management*, 2015.

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

[9] S. Chaudhuri, B. Chen, V. Ganti, and R. Kaushik. Example-driven design of efficient record matching queries. In *VLDB*, pages 327–338, 2007.

[10] S. Chaudhuri, B.-C. Chen, V. Ganti, and R. Kaushik. Example-driven design of efficient record matching queries. In *VLDB*, 2007.

[11] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *ICDE*, page 5, 2006.

[12] A. Cheung, O. Arden, S. Madden, A. Solar-Lezama, and A. C. Myers. Statusquo: Making familiar abstractions perform using program analysis. In *CIDR*, 2013.

[13] A. Cheung, A. Solar-Lezama, and S. Madden. Using program synthesis for social recommendations. In *CIKM*, pages 1732–1736, 2012.

[14] W. W. Cohen and J. Richman. Learning to match and cluster large high-dimensional data sets for data integration. In *SIGKDD*, pages 475–480, 2002.

[15] N. Dalvi, V. Rastogi, A. Dasgupta, A. Das Sarma, and T. Sarlós. Optimal hashing schemes for entity matching. In *WWW*, pages 295–306, 2013.

[16] H. H. Do and E. Rahm. COMA - A system for flexible combination of schema matching approaches. In *VLDB*, 2002.

[17] R. Drechsler and R. Wille. From truth tables to programming languages: Progress in the design of reversible circuits. In *2011 41st IEEE International Symposium on Multiple-Valued Logic*, pages 78–85. IEEE, 2011.

[18] 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.

[19] I. Fellegi and A. Sunter. A theory for record linkage. *Journal of the American Statistical Association*, 64 (328), 1969.

[20] D. Firmani, B. Saha, and D. Srivastava. Online entity resolution using an oracle. *PVLDB*, 9(5):384–395, 2016.

[21] M. A. Fischler and R. C. Bolles. Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography. *Commun. ACM*, 24(6):381–395, June 1981.

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

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

[24] H. Köpcke and E. Rahm. Training selection for tuning entity matching. In *International Workshop on Quality in Databases and Management of Uncertain Data*, pages 3–12, 2008.

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

[26] H. Lakkaraju, S. H. Bach, and J. Leskovec. Interpretable decision sets: A joint framework for description and prediction. In *SIGKDD*, pages 1675–1684, 2016.

[27] T. Lei, R. Barzilay, and T. S. Jaakkola. Rationalizing neural predictions. In *EMNLP*, pages 107–117, 2016.

[28] B. Mozafari, P. Sarkar, M. J. Franklin, M. I. Jordan, and S. Madden. Scaling up crowd-sourcing to very large datasets: A case for active learning. *PVLDB*, 8(2):125–136, 2014.

[29] D. R. Musicant, V. Kumar, and A. Ozgur. Optimizing f-measure with support vector machines. In *Proc. of the 16^{th} International Florida Artificial Intelligence Research Society Conference*, pages 356–360, 2003.

[30] A. Nikolov, M. d'Aquin, and E. Motta. Unsupervised learning of link discovery configuration. In *ESWC*, pages 119–133, 2012.

[31] S. Sarawagi and A. Bhamidipaty. Interactive deduplication using active learning. In *KDD*, pages 269–278, 2002.

[32] L. Seligman, P. Mork, A. Y. Halevy, K. P. Smith, M. J. Carey, K. Chen, C. Wolf, J. Madhavan, A. Kannan, and D. Burdick. Openii: an open source information integration toolkit. In *SIGMOD*, 2010.

[33] R. Singh. Blinkfill: Semi-supervised programming by example for syntactic string transformations. *PVLDB*, 9(10):816–827, 2016.

[34] R. Singh and S. Gulwani. Synthesizing number transformations from input-output examples. In *CAV*, pages 634–651, 2012.

[35] R. Singh and S. Gulwani. Transforming spreadsheet data types using examples. In *POPL*, pages 343–356, 2016.

[36] P. Singla and P. Domingos. Entity resolution with markov logic. In *ICDM*, pages 572–582, 2006.

[37] T. C. Smith and E. Frank. *Statistical Genomics: Methods and Protocols*, chapter Introducing Machine Learning Concepts with WEKA, pages 353–378. Springer, New York, NY, 2016.

[38] A. Solar-Lezama. The sketching approach to program synthesis. In *APLAS*, pages 4–13, 2009.

[39] A. Solar-Lezama. Program sketching. *STTT*, 15(5-6):475–495, 2013.

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

[41] J. Wang, G. Li, J. X. Yu, and J. Feng. Entity matching: How similar is similar. *PVLDB*, 4(10), 2011.

APPENDIX

A. IMPLEMENTATION DETAILS

The pseudocode of the **sample** routine discussed in Section 5.2 is given below.

B. SAMPLE GENERATED GBF RULES

In Section 6.2, we listed a **GBF** rule generated by RuleSynth for Cora and compared it with the rule by traversing the decision tree of depth 3. Here, we give one

Effectiveness (Avg F-measure)	RuleSynth	RS-BestTh	RS-SynthComp	Decision Tree			SVM	SIFI
				3	4	10		
Cora	0.829	0.866	0.864	0.811	0.817	0.948	0.929	0.615
Amazon-GoogleProducts	0.567	0.601	0.614	0.484	0.553	0.61	0.63	0.619
Locu-Foursquare	0.554	0.536	0.614	0.484	0.573	0.693	0.721	0.465
DBLP-Scholar	0.882	0.878	0.908	0.883	0.891	0.917	0.93	0.895
Febrl	0.991	0.989	0.992	0.976	0.991	0.99	1	0.989

Interpretability (Avg #atoms)	RuleSynth	RS-BestTh	RS-SynthComp	Decision Tree			SVM	SIFI
				3	4	10		
Cora	4.8	6.6	14.6	9	23.4	950.4	-	29
Amazon-GoogleProducts	1.4	1.4	4.2	4.6	10.4	916.2	-	6
Locu-Foursquare	3	4.8	9.2	3.6	13	1709.4	-	10
DBLP-Scholar	5	7.2	11	5.2	14.8	1145.4	-	6
Febrl	3.6	3.8	10.2	8.2	14.6	21.4	-	23

Efficiency Avg time per fold (in s)	RuleSynth	RS-BestTh	RS-SynthComp	Decision Tree			SVM	SIFI
				3	4	10		
Cora	2012	2779	1678	30	22	151	47	2503
Amazon-GoogleProducts	838	387	553	20	18	84	35	206
Locu-Foursquare	3687	1194	2114	51	54	433	204	396
DBLP-Scholar	2305	798	1092	18	19	97	19	464
Febrl	2756	2907	2954	49	60	53	31	284

Figure 6: Tables for Effectiveness, Interpretability and Efficiency when optimizing for F-measure

Procedure Modified Sample Routine

input : $\mathbf{E}_{inp}$: Set of examples to be sampled from
$\mathbf{M}, \mathbf{D}$: Sets of positive and negative examples
output: e : An example sampled from $\mathbf{E}_{inp}$ with equal chances of it being positive or negative

$choice \leftarrow$ **random** $(0, 1)$ `// uniform choice: 0 or 1`
if $choice = 1$ **then** `// Choosing Positive Example`
  $\mathbf{E}_{sample} \leftarrow \mathbf{E}_{inp} \cap \mathbf{M}$
else `// Choosing Negative Example`
  $\mathbf{E}_{sample} \leftarrow \mathbf{E}_{inp} \cap \mathbf{D}$
if $\mathbf{E}_{sample} = \varnothing$ **then**
  $\mathbf{E}_{sample} \leftarrow \mathbf{E}_{inp}$
return random $(\mathbf{E}_{sample})$

representative **GBF** detected by RuleSynth (or its variants) for each of the remaining datasets. φ_{synth} indicates a **GBF** generated using RuleSynth whereas, φ_{BestTh} is found by RS-BestTh.

We give the rules generated by RuleSynth for Amazon-GoogleProducts and Locu-Foursquare – the rules for the other datasets are ignored for space constraints.

- Amazon-GoogleProducts:

$$\varphi_{synth}: \ \mathsf{CosineToken[title]} \geqslant 0.571$$

- Locu-Foursquare:

$$\varphi_{synth}: \ \textbf{if} \ \ \mathsf{noNulls[region]} \geqslant 1$$
$$\textbf{then} \ \ \big(\big(\mathsf{MongeElkan[street\_address]} \geqslant 0.861 \bigvee \big(\mathsf{ChapmanMeanLength[locality]} \geqslant 0.161 \bigwedge \mathsf{Jaro[name]} \geqslant 0.753\big)\big)\big)$$
$$\textbf{else} \ \ \mathsf{EuclideanDistance[name]} \geqslant 0.678$$

As descried in Section 5.3, RS-SynthComp produces larger **GBF**s by composing smaller **GBF**s. $\varphi_{SynthComp}$ is one such composite **GBF** generated by RS-SynthComp for Amazon-GoogleProducts.

$$\varphi_{SynthComp}: \ \big(\big(\mathsf{a0} \bigvee \mathsf{a1}\big) \bigwedge \mathsf{a2}\big)$$
$$\mathsf{a0}: \ \mathsf{CosineToken[title]} \geqslant 0.55$$
$$\mathsf{a1}: \ \mathsf{OverlapGram_3[title]} \geqslant 0.609$$
$$\mathsf{a2}: \ \mathsf{CosineToken[title]} \geqslant 0.489$$

φ_{SIFI} shows an expert-provided **DNF** expression with the chosen similarity functions and thresholds from one of the 5 folds for the Amazon-GoogleProducts dataset.

$$\varphi_{SIFI}: \ \big(\big(\mathsf{OverlapGram_3[title]} \geqslant 0.78 \bigwedge \mathsf{ChapmanLengthDeviation[price]} \geqslant 0.78\big) \bigvee \big(\big(\mathsf{OverlapGram_3[title]} \geqslant 0.78 \bigwedge \mathsf{OverlapToken[description]} \geqslant 0.536\big) \bigwedge \mathsf{ChapmanLengthDeviation[price]} \geqslant 0.125\big)\big)$$

C. OPTIMIZING PRECISION AND RECALL METRICS

The results presented in this paper correspond to optimizing F-measure. In this section, we present results of optimizing for Precision and Recall using RuleSynth on all 5 datasets used in this paper. In Figure 10, we show the average precision and recall across 5 folds when optimizing for precision and recall respectively. The datasets are represent by shorthand notations: $\mathsf{D_C}$ =**Cora**, $\mathsf{D_{DS}}$ =**DBLP-Scholar**, $\mathsf{D_F}$ =**Febrl**, $\mathsf{D_{AG}}$ =**Amazon-GoogleProducts**, $\mathsf{D_{LF}}$ =**Locu-FourSquare**. We can see from the figure that average precision (recall) values when optimizing precision (recall) are higher than the corresponding values when optimizing F-measure(Figures 8 & 9). But, average precision (recall) when optimizing recall (precision) decreases by a lot as compared to when optimizing for F-measure. It can also be seen that the average F-measure has consistently dropped while optimizing for only precision or only recall that justifies our choice of F-measure as the optimization metric. Note that this change in metric is not easily adaptable to decision trees or SVM as they have internal heuristic in the algorithm to be optimized such as entropy or maximum margin.

Cora	pairs with nulls (out of 184660)	
Attribute	**#**	**%**
author	774	0.4
title	5657	3.1
venue	46924	25.4
address	127581	69.1
publisher	130428	70.6
editor	164078	88.9
date	93476	50.6
volume	157863	85.5
pages	109812	59.5

Amazon-GoogleProducts	pairs with null(s) (out of 97007)	
attribute	**#**	**%**
title	0	0.0
description	739	0.8
manufacturer	89593	92.4
price	0	0.0

Locu-Foursquare	pairs with nulls (out of 341245)	
Attribute	**#**	**%**
website	318588	93.4
name	367	0.1
locality	125012	36.6
country	3	0.0
region	132021	38.7
longitude	0	0.0
phone	17372	5.1
postal_code	206633	60.6
latitude	0	0.0
street_address	196293	57.5

DBLP-Scholar	pairs with nulls (out of 112840)	
Attribute	**#**	**%**
title	0	0.0
authors	0	0.0
venue	7645	6.8
year	43093	38.2

Febrl	pairs with nulls (out of 239999)	
Attribute	**#**	**%**
given_name	9896	4.1
surname	21795	9.1
street_number	8528	3.5
address_1	26880	11.2
address_2	212630	88.5
suburb	5570	2.3
postcode	5370	2.2
state	57714	24.0
date_of_birth	61310	25.5
age	109986	45.8
phone_number	22471	9.4
soc_sec_id	0	0.0
blocking_number	0	0.0

Figure 7: Number of record pairs having at least one null value of an attribute for each dataset

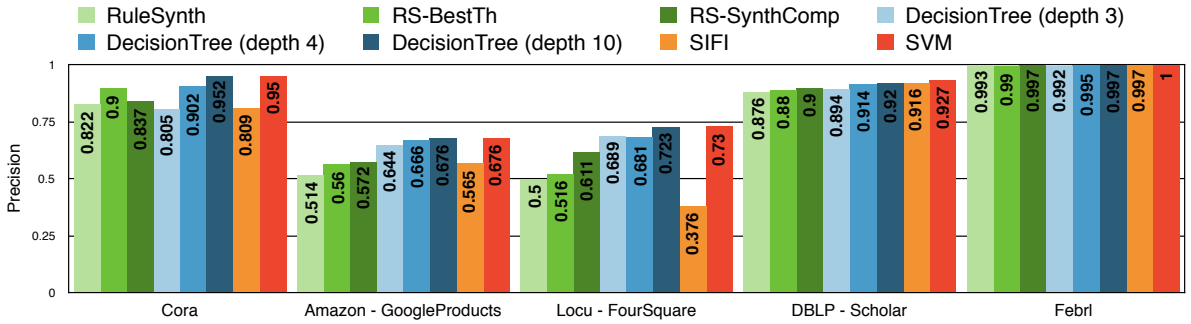


Figure 8: Average Precision across 5 folds (When optimizing for F-measure)

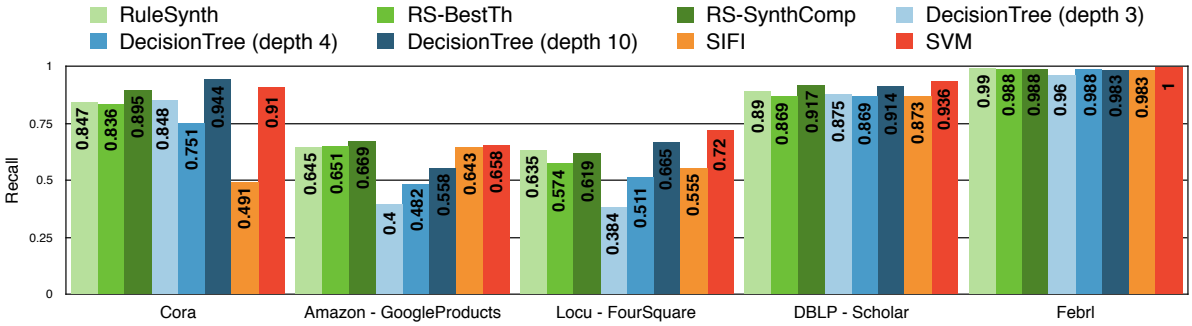


Figure 9: Average Recall across 5 folds (When optimizing for F-measure)

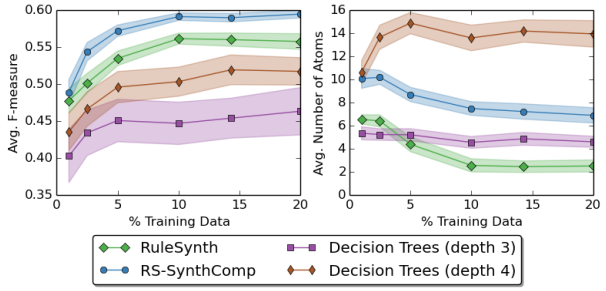
Optimizing Precision	RuleSynth			
Dataset	**Avg Precision**	**#atoms**	**Avg Recall**	**Avg F − measure**
D_C	0.926	3.2	0.541	0.647
D_{AG}	0.694	3.4	0.318	0.388
D_{LF}	0.596	3.4	0.483	0.498
D_{DS}	0.94	1.8	0.626	0.738
D_F	1	2.8	0.965	0.982

Optimizing Recall	RuleSynth			
Dataset	**Avg Precision**	**#atoms**	**Avg Recall**	**Avg F − measure**
D_C	0.077	1.0	1.0	0.144
D_{AG}	0.013	1.0	1.0	0.026
D_{LF}	0.018	1.0	1.0	0.035
D_{DS}	0.047	1.0	1.0	0.09
D_F	0.05	1.6	0.997	0.084

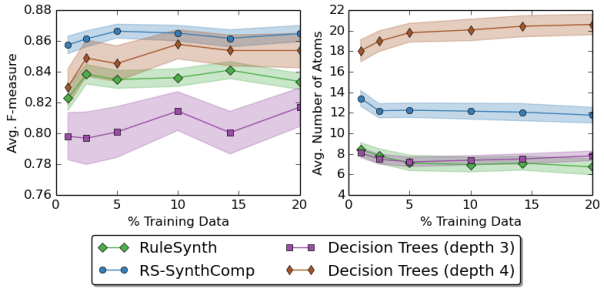
Figure 10: Average Precision, Recall & F-measure for 5 folds, optimizing for Precision and Recall, resp.

D. EXPERT RULE COMPARISON ON SIFI

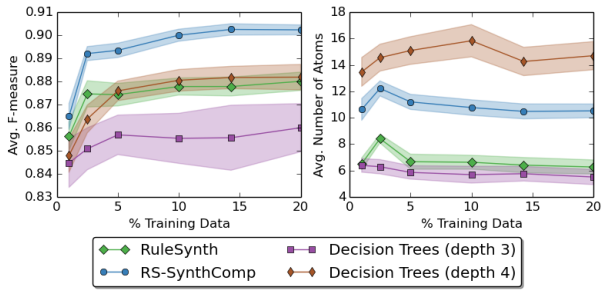
We have obtained the input **DNF** rules for SIFI experiments from two database experts. While we discussed the results obtained using one set of **DNF** rules in Section 6, we present the other set of results in the following figure in which **F** is F-measure, **#A** is the number of atoms and **T** is the training time in seconds. The expert-provided rules containing more atoms typically produce higher F-measures but the search time involved in those cases also increases significantly. For instance, the average runtime per fold of SIFI using Expert-2 rules upon datasets Cora and Locu-Foursquare are ∼2 hours and ∼8 hours, respectively. In the



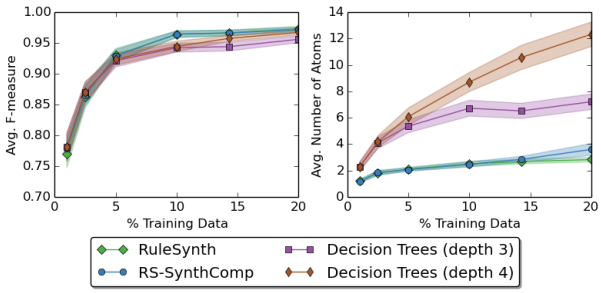
(a) Amazon-GoogleProducts



(b) Cora



(c) DBLP-Scholar



(d) Febrl

Figure 11: Limited Training Experiment (Averages for 100 runs with 99% Confidence Intervals)

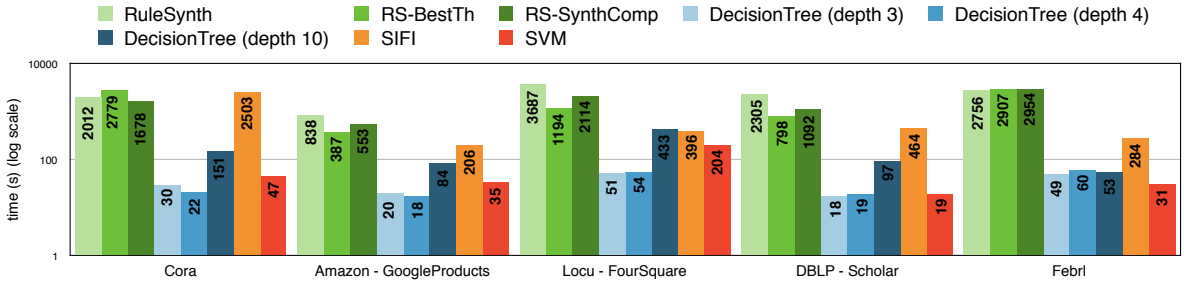


Figure 12: Efficiency (Average time for training per fold) for 5-folds experiment (80% Training/ 20% Testing)

case of Amazon-GoogleProducts, DBLP-Scholar and Febrl, the run-times using Expert-2 rules are relatively lesser because of fewer rules. These run-times can be compared with the variants of RuleSynth which ran at most for an hour to produce competitive F-measures as reported in Section 6.

	SIFI Expert 1			SIFI Expert 2		
	F	#A	T	F	#A	T
D_C	0.615	29	2503	0.822	35	7611.5
D_{AG}	0.619	6	206	0.601	6	199.8
D_{LF}	0.465	10	396	0.641	36	28239.5
D_{DS}	0.895	6	464	0.668	11	704.6
D_F	0.989	23	284	0.953	12	224.2

E. LIMITED TRAINING DATA EXPERIMENTS: ALL DATASETS

We present the results for the other 4 datasets in Figure 5. We used a cluster with 70 parallelism to run these experiments. For each of the 5 datatsets, 4 techniques and 6 training data percentages - we run the experiment 100 times on different random training sets (note that the training sets were kept the same across each technique). These $5 \times 4 \times 6 \times 100 = 12000$ runs with 70 parallelism took 36 hours in total with various techniques taking different times on average across their 3000 runs each: (1) RuleSynth: 2436 s (2) RS-SynthComp: 663 s (3) Decision Trees (depth 3): 79 s (4) Decision Trees (depth 4): 86 s.

F. EFFICIENCY STUDY

From Figure 12, we observe that RuleSynth and its variants take at most an hour to search through the huge space of rules in order to produce an effective and concise rule as output for all datasets in Table 1. This is a reasonable amount of time as compared to what it takes experts to examine the dataset and write their own rule expressions, especially given the low-cost of computation relative to human time. For example, our experts took around 2 hours on average to write a **DNF** expression for SIFI per dataset.

Figure 12 also shows that SIFI searches through a smaller constrained space in at most 40 minutes to produce a rule. Decision trees with depth 3 and 4 produce a rule in less than a minute but the produced rules are neither as concise nor as effective as rules produced by RuleSynth and its variants (Exp-2). Both decision trees with depth 10 and SVM take at most 8 minutes to produce a rule but they lack any kind of interpretability (Exp-4).