Auto-FuzzyJoin:
Auto-Program Fuzzy Similarity Joins Without Labeled Examples

Peng Li∗
Georgia Institute of Technology
Atlanta, USA
pengli@gatech.edu

Xiang Cheng∗
Georgia Institute of Technology
Atlanta, USA
cxworks@gatech.edu

Xu Chu
Georgia Institute of Technology
Atlanta, USA
xu.chu@cc.gatech.edu

Yeye He
Microsoft Research
Redmond, USA
yeyehe@microsoft.com

Surajit Chaudhuri
Microsoft Research
Redmond, USA
surajitc@microsoft.com

ABSTRACT
Fuzzy similarity join is an important database operator widely used in practice. So far the research community has focused exclusively on optimizing fuzzy join scalability. However, practitioners today also struggle to optimize fuzzy-join quality, because they face a daunting space of parameters (e.g., distance-functions, distance-thresholds, tokenization-options, etc.), and often have to resort to a manual trial-and-error approach to program these parameters in order to optimize fuzzy-join quality. This key challenge of automatically generating high-quality fuzzy-join programs has received surprisingly little attention thus far.

In this work, we study the problem of “auto-program” fuzzy-joins. Leveraging a geometric interpretation of distance-functions, we develop an unsupervised Auto-FuzzyJoin framework that can infer suitable fuzzy-join programs on given input tables, without requiring explicit human input such as labelled training data. Using Auto-FuzzyJoin, users only need to provide two input tables $L$ and $R$, and a desired precision target $\tau$ (say 0.9). Auto-FuzzyJoin leverages the fact that one of the input is a reference table to automatically program fuzzy-joins that meet the precision target $\tau$ in expectation, while maximizing fuzzy-join recall (defined as the number of correctly joined records).

Experiments on both existing benchmarks and a new benchmark with 50 fuzzy-join tasks created from Wikipedia data suggest that the proposed Auto-FuzzyJoin significantly outperforms existing unsupervised approaches, and is surprisingly competitive even against supervised approaches (e.g., Magellan and DeepMatcher) when 50% of ground-truth labels are used as training data. We have released our code and benchmark on GitHub† to facilitate future research.

†Both authors contributed equally.

†https://github.com/chu-data-lab/AutomaticFuzzyJoin

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SIGMOD ’21, June 20–25, 2021, Virtual Event, China
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8345-1/21/06...$15.00
https://doi.org/10.1145/3448016.3452824

CCS CONCEPTS
• Information systems → Entity resolution; • Computing methodologies → Unsupervised learning.

KEYWORDS
fuzzy join; similarity join; entity resolution; unsupervised learning

ACM Reference Format:

1 INTRODUCTION
Fuzzy-join, also known as similarity-join, fuzzy-match, and entity resolution, is an important operator that takes two tables $L$ and $R$ as input, and produces record pairs that approximately match from the two tables. Since naive implementations of fuzzy-joins require a quadratic number of comparisons that is prohibitively expensive on large tables, extensive research has been devoted to optimizing the scalability of fuzzy-joins (e.g., [10, 11, 13, 19, 22, 25, 35, 39]).

We have witnessed a fruitful line of research producing significant progress in this area, leading to wide adoption of fuzzy-join as features in commercial systems that can successfully scale to large tables (e.g., Microsoft Excel [3], Power Query [8], and Alteryx [1]).

The need to parameterize fuzzy-joins. With the scalability of fuzzy-join being greatly improved, we argue that the usability of fuzzy-join has now become a main pain-point. Specifically, given the need to optimize join quality for different input tables, today’s fuzzy-join implementations offer rich configurations and a puzzling number of parameters, many of which need to be carefully tuned before high-quality fuzzy-joins can be produced.

Microsoft Excel, for instance, has a popular fuzzy-join feature available as an add-in [3]. It exposes a rich configuration space, with a total of 19 configurable options across 3 dialogs, as shown in Figure 1. Out of the 19 options, 11 are binary (can be either true or false), which already correspond to $2^{11} = 2048$ discrete configurations, which are clearly difficult to program manually. Similarly, py_stringmatching [6], a popular open-source fuzzy-join package, boasts 92 options. Note that we have not yet included parameters from numeric continuous domains, e.g., “similarity threshold” and “containment bias” that can take any value in ranges like $[0, 1]$. 

Because different types of string variations—when doing configurations—one-matchup.html

3. In Excel or Tableau) struggle as they either have to laboriously try ten still requiring trials-and-errors); less-technical users (e.g., those using their experience to make educated guess of suitable parameters (of-

5. Unions Of Configurations. Because different types of string variations are often present at the same time (e.g., typos vs. extraneous tokens), ideally a fuzzy-join program should contain a union of fuzzy-join configurations to optimize recall – in the example above, Edit-distance (l, r) < 3 \lor Jaccard-distance (l, r) < 0.2, to correctly join these records. Note that for humans, programming such a union of configurations manually is even more challenging than tuning for a single configuration. Our approach can automatically search disjunctive join programs suitable for two given input tables, which can achieve optimized join quality (Section 2.2).

6. Figure 1: A total of 19 parameters exposed to users in Fuzzy-join for Microsoft Excel (across 3 dialog windows).

7. Figure 2: A sample of fuzzy-join parameters.

8. Sub-optimal default parameters. We argue that this is a significant pain point, and a major roadblock to wider adoption of fuzzy-join.

9. In this paper, we explore the possibility of automatically programming fuzzy-joins, using suitable parameters tailored to given input tables. Our approach is designed to be unsupervised, requiring no inputs from human users (e.g., labeled training examples for matches and non-matches). It exploits a key property of fuzzy-join tasks, which is that one of the input tables is often a “reference table”, or a curated master table that contains few or no duplicates. We note that the notion of reference tables is widely used in the literature (e.g., [17, 18, 44]), and adopted by commercial systems (e.g., SQL Server [4], OpenRefine [5], Excel [3], etc.). As we will see, leveraging this key property of reference tables allows us to infer high-quality fuzzy-join programs without using labeled data.

10. An intuitive example. We illustrate a few key ideas we leverage to auto-program fuzzy-joins using an intuitive example. On the left of Figure 3(a) is a reference table L with NCAA team names, and on the right is a table R with team names that need to be matched against L. As can be seen, (l₁, r₁) and (l₂, r₂) share a large set of common tokens, so intuitively we should token-ize them by word boundaries and join them using set-based metrics like Jaccard distance or Jaccard-Containment distance. On the other hand, (l₃, r₃) is not intuitively also join, but their token overlap is not high (Jaccard distance can be computed as 0.5), because of misspelled “Mississippi” and “Bulldog” in R. Such pairs are best joined by viewing input strings as sequences of characters, and compared using Edit-distance (e.g., Edit-distance(l, r) < 3).

11. Figure 3: Examples of Fuzzy Join Cases.
Learn-safe-join-boundaries. In order to determine suitable parameters for fuzzy-joins on given input tables, we leverage reference-tables $L$ to infer what fuzzy-join boundaries are “safe” (generating few false-positives). In Figure 3(b) for instance, unlike Figure 3(a), even a seemingly small Edit-distance $(l, r) \leq 1$ is not “safe” on this data set, and would produce many false-positives like $(l_1, r_1), (l_2, r_2)$, etc., none of which are correct joins. While it may be obvious to humans recognizing roman numerals in the data, it is hard for algorithms to know without labeled examples. Here we leverage an implicit property of the reference table $L$ that it has few or no duplicates, to perform automated inference. Assume for a moment that a fuzzy-join with Edit-distance $(l, r) \leq 1$ on this pair of $L$ and $R$ is a “safe” distance to use. Because $L$ and $R$ are similar in nature, it then follows that this fuzzy-join on $L$ and $L$ is also “safe”. However, applying this self-fuzzy-join on $L$ leads to many joined pairs like $(l_1, l_2), (l_2, l_3)$, etc., contradicting with the belief that $L$ has few fuzzy-duplicates, and suggesting that Edit-distance $(l, r) \leq 1$ likely joins overly aggressively and is actually not “safe”. We generalize this idea using a geometric interpretation of distance functions in a fine-grained (per-record) manner, in order to learn “safe” fuzzy-join programs that can maximize recall while ensuring high precision (Section 3.1). This is a key idea behind Auto-FuzzyJoin.

Negative-rule-learning. We observe that similarity functions and scores alone are sometimes still not sufficient for high-quality fuzzy joins. For instance, existing solutions would join $(l_6, r_6)$ and $(l_7, r_7)$ in Figure 3(a) because of their high similarity scores, which as humans we know are false-positive results. Our approach can automatically learn what we call “negative rules”, by analyzing the reference table $L$. Specifically, we will find many pairs of records like (“2008 LSU Tigers baseball team”, “2008 LST Tigers football team”) that coexist in the reference table $L$, and because these are from the reference table and thus unlikely to be duplicates, we can infer a negative rule of the form “baseball” ≠ “football”, which would prevent $(l_6, r_6)$ from being joined. Similarly, we can learn a negative rule “2007” ≠ “2008”, so that $(l_7, r_7)$ is not joined (Section 3.3).

Key features of Auto-FuzzyJoin. Auto-FuzzyJoin has the following key features that we would like to highlight:

- Unsupervised. Unlike most existing methods, Auto-FuzzyJoin does not require labeled examples of matches/non-matches.
- High-Quality. Despite not using labeled examples, it outperforms strong supervised baselines (e.g., Magellan and DeepMatcher) even when 50% of ground-truth joins are used as training data.
- Robust. Our approach is robust to tables with varying characteristics, including challenging test cases adversarially-constructed.
- Explainable. Compared to black-box methods (e.g., deep models), our approach produces fuzzy-join programs in a disjunctive form, which is easy for practitioners to understand and verify.
- Extensible. Parameter options listed in Figure 2 are not meant to be exhaustive, and can be easily extended (e.g., new distance functions) in our framework in a manner transparent to users.

2 PRELIMINARIES

2.1 Many-to-one Fuzzy Joins

Definition 2.1. Let $L$ and $R$ be two input tables, where $L$ is the reference table. A fuzzy join $J$ between $L$ and $R$ is a many-to-one join, defined as $J : R \rightarrow L \cup \perp$.

The fuzzy join $J$ defines a mapping from each tuple $r_j \in R$ to either one tuple $l_i \in L$, or an empty symbol $\perp$, indicating that no matching record exists in $L$ for $r_j$, as $L$ may be incomplete. Note that because $L$ is a reference table, each $r_j$ can join with at most one tuple in $L$. However, in the other direction, each tuple in $L$ can join with multiple tuples in $R$, hence a many-to-one join.

The notion of reference tables is widely used both in the fuzzy-join/entity-matching literature (e.g., [17, 18, 44]) and commercial systems (e.g., Excel [3], SQL Server [4], OpenRefine/GoogleRefine [5], etc.). In practice, we find that most benchmark datasets used for entity-resolution in the literature indeed have a reference table, for which our approach is applicable. For example, in the well-known Magellan data repository of ER6, we find 19/29 datasets to have a reference table that is completely duplicate-free, and 26/29 datasets to have a reference table that has less than 5% duplicates, confirming the prevalence of reference tables in practice.7

The reference table property essentially serves as a structural constraint to prevent our algorithm from using fuzzy-join configurations that are too “loose” (or join more than what is correct). We refer readers to a full version of the paper [36] for an additional analysis showing that some form of constraints are necessary before good fuzzy-joins can be inferred.

2.2 The Space of Join Configurations

A standard way to perform fuzzy-join is to compute a distance score between $r$ and $l$. There is a rich space of parameters that determine how distance scores are computed. Figure 2 gives a sample such space. There are four broad classes of parameters: pre-processing (P), tokenization (T), token-weights (W), and distance-functions (D). The second column of the figure gives example parameter options commonly used in practice [10, 11, 13, 22, 25, 35, 39]. Combination of these parameters (P, T, W, D) uniquely determines a distance score for two input strings $(l, r)$, which we term as a join function $f \in \mathcal{F}$, where $\mathcal{F}$ denotes the space of join functions.

Example 2.1. Consider join function $f = (L, SP, EW, JD)$, which uses lower-casing (L), space-tokenization (SP), equal-weights (EW), and Jaccard-distance (JD) from Figure 2. Applying this $f$ to $l_1$, $r_1$ in Figure 3(a), we can compute $f(l_1, r_1) = 0.2$. Additional examples of score computation are shown in the last column of Figure 2.

In our experiments we consider a rich space with hundreds of join functions. Our Auto-FuzzyJoin approach treats these parameters as black-boxes, and as such can be easily extended to additional parameters not listed in Figure 2.

Given distance $f(l, r)$ computed using $f$, the standard approach is to compare it with a threshold $\theta$ to decide whether $l$ and $r$ can be joined. Together $\theta$ and $f$ define a join configuration $C$.

Definition 2.2. A join configuration $C$ is a 2-tuple $C = (f, \theta)$, where $f \in \mathcal{F}$ is a join function, while $\theta$ is a threshold. We use $\mathcal{S} = \{(f, \theta) \mid f \in \mathcal{F}, \theta \in \mathbb{R}\}$ to denote the space of join configurations.

Given two tables $L$ and $R$, a join configuration $C \in \mathcal{S}$ induces a fuzzy join mapping $J_C$, defined as:

$$J_C(r) = \arg \min_{l \in L} \{ f(l, r) : \forall r \in R \}$$

However, for reduced recall.

---

6https://sites.google.com/site/anhaidgroup/useful-stuff/data
7As we will see, for cases where reference tables are absent, our approach will still work but may generate overly conservative fuzzy-join programs, which is still of high precision but may have reduced recall.
The fuzzy join $J_C$ defined in Equation (1) ensures that each $r$ record joins with $l \in L$ with the smallest distance. Note that this can also be empty if $f(l, r) > 0$ for all $l, r \in L$.

We observe that real data often have different types of variations simultaneously (e.g., typos vs. missing tokens vs. extraneous information), one join configuration alone is often not enough to ensure high recall. For example, in Figure 3(a), Jaccard distance with threshold 0.2 may be suitable for joining pairs like $(l_1, r_1)$ as these pairs differ by one or two tokens. However, for pairs like $(l_2, r_3)$ that have spelling variations, Jaccard-distance (0.5) is high, and Edit-distance is required to join $(l_2, r_3)$.

To handle different types of variations, in this work our algorithm will use a set of configurations $U = \{C_1, C_2, \ldots, C_K\}$ (as opposed to a single configuration), where the join result of $U$ is defined as the union of the result from each configuration $C_i$.

Definition 2.3. Given $L$ and $R$, a set of join configurations $U = \{C_1, C_2, \ldots, C_K\}$ induces a fuzzy join mapping $J_U$, defined as:

$$J_U(r) = \bigcup_{C_i \in U} J_{C_i}(r), \forall r \in R$$

(2)

Intuitively, each $C_i$ produces high-quality joins capturing a specific type of string variations, and two records are joined in $U$ if and only if they are joined by one configuration $C_i \in U$ (we discuss scenarios with conflicts in Section 3).

2.3 Auto-FuzzyJoin: Problem Statement

Given $R$ and $L$, and a space of join configurations $S$, the problem is to find a set of join configurations $U$ that produces “good” fuzzy-join results. Let $J_U$ denote the fuzzy join mapping induced by $U$ and let $J_C$ denote the ground truth fuzzy join mapping. The “goodness” of a solution $U$ can be measured using precision and recall:

$$precision(U) = \frac{|\{r \mid r \in R, J_U(r) \neq \emptyset, J_C(r) = J_U(r)\}|}{|\{r \mid r \in R, J_C(r) \neq \emptyset\}|}$$

(3)

$$recall(U) = \frac{|\{r \mid r \in R, J_C(r) \neq \emptyset, J_U(r) = J_C(r)\}|}{|\{r \mid r \in R, J_C(r) \neq \emptyset\}|}$$

(4)

The precision of $U$ is the fraction of predicted joins that are correct according to the ground-truth; and the recall of $U$ is defined as the number of correct matches (a variant widely-used in the IR literature [38]). We note that this definition of recall in absolute terms simplifies our analysis, which will be relaxed later.

Problem Statement. Given $L$ and $R$, and a target precision $r$. Let $S = \{(f, \theta) \mid f \in F, \theta \in \mathbb{R}\}$ be the space of fuzzy-join configurations. We would like to find a set of configurations $U = \{C_1, C_2, \ldots, C_K\}$ with $C_i \in S$, that maximizes recall($U$), while observing the required precision $r$. This recall-maximizing fuzzy-join problem (RM-FJ) formulation can be written as an optimization problem:

$$\text{(RM-FJ)} \quad \max \text{recall}(U)$$

s.t. $\text{precision}(U) \geq r$

$U \in 2^S$

(5)

(6)

(7)

Theorem 2.1. The decision version of the RM-FJ problem is NP-hard.

The hardness result can be obtained using a reduction from densest-$k$-subhypergraph [29]. A proof of the result can be found at the full version of this paper [36].

3 SINGLE-COLUMN AUTO-FUZZYJOIN

We now discuss Auto-FuzzyJoin when the join key is a pair of single column (we will extend it to multi-columns in Section 4).

3.1 Estimate Precision and Recall

In the RM-FJ formulation above, the hardness result assumes that we can compute the precision and recall of any configuration $U$. In reality, however, we are only given $L$ and $R$, with no ground truth. To solve RM-FJ, we first need a way to estimate precision/recall of a fuzzy-join without using ground-truth labels, which is a key challenge we need to address in Auto-FuzzyJoin.

In the following, we show how precision/recall can be estimated in our specific context of fuzzy-joins, by leveraging a geometric interpretation of distances, and unique properties of the reference table $L$. For ease of exposition, we will start our discussion with a single join configuration $C$, before extending it to a set of configurations $U = \{C_1, C_2, \ldots, C_K\}$.

Estimate for a single-configuration $C$. Given a configuration $C$, and two tables $L$ and $R$, we have the following estimation of precision/recall of $C$. Recall that a configuration $C = (f, \theta)$ consists of a join function $f$ that computes distance between two records, and a threshold $\theta$.

Assuming a “complete” $L$. We will start by analyzing a simplified scenario where the reference table $L$ is assumed to be complete (with no missing records). This is not required in our approach, but used only to simplify our analysis (which will be relaxed later).

Using a geometric interpretation, given some distance function $f$, records in a table can intuitively be viewed as points embedded in a multi-dimensional space (e.g., with metric embedding [9]), as visualized in Figure 4.

When $L$ is complete (containing all possible $l$ records in the same domain), for each $l \in L$, the closest neighbors of each $l$ tend to differ in some standardized/structured manner, making these closest neighbors to have similar distances to $l$. For instance, for most records $l$ in Figure 3(a), their closest neighbors differ from $l$ by only one token (either year or sport-name), which translates to a Jaccard-distance of around 0.2. In Figure 3(b), for most records $l$, their closest neighbors differ from $l$ by one character (in roman numerals), or an Edit-distance of 1. The same extends to many other domains – e.g., in a reference table with addresses, closest neighbors to each $l$ will likely differ from $l$ by only house-numbers (one token); in a reference table with person-names, closest neighbors to each $l$ will likely differ by only last-names (one token), etc.

Because the closest neighbors of $l$ tend to have similar distances to $l$, we can intuitively visualize $L$ points in the local neighborhoods of each $l$ as points on unit-grids in a multi-dimensional space – Figure 4 visualizes reference records $l$ as blue points on the grid, with similar distances between close neighbors (this is shown in 2D but can generalize to higher dimensions too).

Using an analogy from astronomy, we can intuitively think of reference records in $L$ as “stars” on unit-grids, whereas the query records $r \in R$ (having different string variations from their corresponding $l$) can be thought of as “planets” orbiting around “stars” (with different distances to their corresponding $l$). Determining which $l$ should each $r$ join amounts to finding the closest $l$ (star), when using a suitable distance function $f$.

In an idealized setting where $L$ is complete, conceptually finding the correct left record to join for a given $r$ is straightforward, as
In practice, however, "bad" join (false-positive).

In the case when the record in the ball of distance 2$d$ becomes

Dealing with an "incomplete" $L$. In practice, however, $L$ can be incomplete or have many missing records – in our visual model, this would lead to missing blue points on the grid. Given some $r \in R$ whose corresponding $l$ record is missing in $L$, the simplistic approach of joining this $r$ with its closest $l \in L$ leads to a false-positive and lower precision.

For example, in Figure 4(b), $r_2$ is a record that should join with the reference record $l_2$, which however is currently missing in $L$. This makes correct fuzzy-joins challenging, because given $l_2$ is absent in $L$, the closest $L$ record to $r_2$ becomes $l_1$, and a naive approach would attempt to fuzzy-join $(r_2, l_1)$ using a distance of $d = f(r_2, l_1)$, which creates a false-positive join. The key question we try to address, is how to infer (without using ground-truth) that this particular $d$ is too lax of a distance to use, and the resulting $(r_2, l_1)$ join is likely a "bad" join (false-positive).

Our key idea here, is to infer the distances between $L$ records, and use that to determine "safe" fuzzy-join distances to use. Specifically, recall that when $L$ is complete and $L$ records are visualized on unit-grids, like shown in Figure 4(a), closest neighbors of an $l$ tend to have similar distances to $l$, which we refer to as $w$, or the "width" of the grid (shown in the figure). A hypothetical record $r_1$ in Figure 4(a) that lies right in between of $l_1$ and $l_3$ is then of distance $\frac{w}{2}$ to both the two $L$ records, which intuitively cannot be reliably joined with either $l_1$ or $l_3$. Analogously, a "planet" lying right in between two "stars" cannot be "claimed" by either. Intuitively, we can see that in this case, the "safe" distance to join a pair of $(l, r)$ is when $d = f(l, r) < \frac{w}{2}$, which is when this $r$ would clearly lie on one side and be closest to one $L$ record (this can be shown formally via triangle-inequality).

In the case where $L$ is incomplete with missing records, like shown in Figure 4(b), estimating this grid-width $w$ may not be as straightforward. As a result, we perform this analysis in the other direction – given a pair $(l, r)$ that we want to join, we compute their distance $d = f(l, r)$. We then draw a ball centered around $l$ with a radius of $2d$, and test how many additional $L$ records would fall within this $2d$-ball. Because if $d < \frac{w}{2}$, which is a "safe" distance to join based on our analysis above, then it follows that $2d < w$, meaning that in this $2d$-ball centered around $l$ we should expect to see no other $L$ records (except $l$). If we indeed see no $L$ record in the $2d$-ball, we can be confident that this $d$ used to join $(l, r)$ is small enough and the join is "safe". Alternatively, if we observe many $L$ records in the $2d$-ball, this likely indicates that $2d \geq w$, or $d \geq \frac{w}{2}$, which based on our analysis above is too "lax" to be "safe".

Example 3.1. In Figure 4(a), to join $r_1$, we first find its closest $L$, which is $l_1$, and compute $d = f(l_1, r_1)$. We then draw a $2d$-ball around $l_1$, and find no other $L$ records, indicating that this $d$ is small enough and a "safe" distance to use based on $l_1$'s local neighborhood.

In Figure 4(b), $r_2$ should join $l_2$, which however is missing in $L$. In this case, we would find $l_1$ to be closest to $r_2$ in the absence of $l_2$, with a distance $d' = f(l_1, r_2)$. When we draw a $2d'$ ball around $l_1$, we find many additional $L$ records, which based on our analysis above indicates that it is likely that this $d' \geq \frac{w}{2}$, which is too "lax" to use in this local neighborhood, and we should not join $r_2$ with $l_1$.

Note that in Figure 4(b), we have 4 missing $L$ records (marked by dotted rectangles). This incomplete $L$, however, still allows us to conclude that joining $(l_1, r_2)$ is not "safe". In fact, in this 2-D example, we can "tolerate" up to 7 missing $L$ records in the neighborhood while still correctly deciding that $(l_1, r_2)$ is likely not "safe" to join. We should note that this tolerance level goes up exponentially when records are embedded in a higher-dimensional space (e.g., in a 3-D unit-cube, we can tolerate up to 25 missing $l$ out of 27 positions).

Estimating join precision. Given $r \in R$, let $l \in L$ be the closest to $r$ with distance $d = f(l, r)$, we can estimate the precision of this join pair $(l, r)$ (the likelihood of it being correct), to be the inverse of the number of $L$ records within the $2d$-ball. We write this as precision$(l, r)$, shown in Equation (8). We use the reciprocal to estimate precision, because all $l$ within the $2d$-ball are reasonably close to $r$, and are thus plausible counterparts to join with this $l$.

$$\text{precision}(l, r) = \frac{1}{|\{l' \in L : f(l, l') \leq 2f(l, r)\}|}$$

(8)

Example 3.2. The precision of $(l_1, r_1)$ in Figure 4(a) can be estimated as $\frac{1}{5}$ per Equation (8), because $l_1$ is closest to $r_1$, and the $2d$ ball around $l_1$ has only one $L$ record (itself).

The precision of $(l_1, r_2)$ in Figure 4(b) can be estimated as $\frac{1}{5}$, since the $2d'$ ball has 5 $L$ records (note that 4 $L$ records are missing).

For an example from tables, we revisit Figure 3(b). Here for $r_1$, the closest in $L$ by Edit-distance is $l_2$ with $d = 1$. While the pair is as close as it gets for Edit-distance, the $2d$-ball around $l_2$ (with a radius of Edit-distance=2) has many $L$ records (e.g., $l_2, l_3$, etc.), indicating that the join $(r_1, l_1)$ is of low precision.

We would like to note that this estimated precision$(l, r)$ is not intended to be exact when $L$ is incomplete. Because in our application users typically want high-precision fuzzy-joins (e.g., target precision of 0.9 or 0.8), our precision estimate only needs to be informative to qualitatively differentiate between high-confidence joins (clean balls), and low-confidence joins (balls with more than one $L$ record). As soon as the balls contain more than one $L$ record, the estimated precision drops quickly to below 0.5, at which point our algorithm would try to avoid given a high precision target (i.e., it does not really matter if the estimate should really be $\frac{1}{2}$ or $\frac{3}{4}$).

Using the precision estimate for a single $(l, r)$ pair in Equation (8), we can now estimate precision for a given configuration $C = (f, \theta)$.
Recall that given $C$, each $r \in R$ is joined with $J_C(r)$ (defined in Equation (1)), which can be an $l \in L$ or empty (no suitable $l$ to join with). The estimated precision of $r$ joined using $C$ if $J_C(r) \neq \emptyset$ is:

$$\text{precision}(r, C) = \frac{1}{|\{l' | l' \in L, l = J_C(r), f(l, l') \leq 2\theta\}|}$$ (9)

The expected number of true-positives $TP(C)$ is the sum of expected precision of each $r$ that $C$ can join:

$$TP(C) = \sum_{r \in J_C(r) \neq \emptyset} \text{precision}(r, C)$$ (10)

And the expected number of false-positives $FP(C)$ is:

$$FP(C) = \sum_{r \in R \setminus J_C(r) \neq \emptyset} (1 - \text{precision}(r, C))$$ (11)

Thus, the estimated precision and recall of a given $C$ is:

$$\text{precision}(C) = \frac{TP(C)}{TP(C) + FP(C)}; \text{recall}(C) = \frac{TP(C)}{TP(C)}$$ (12)

**Estimate for a set of configurations $U$.** We now discuss how to estimate the quality for a set of configurations $U$.

In the simple (and most common) scenario, the join assignment of each $r \in R$ has no conflicts within $U$. This can be equivalently written as $\forall r \in R, |J_U(r)| \leq 1$ (recall $J_U(r)$ is the result induced by $U$ defined in Equation (2)). In such scenarios, estimating for $U$ is straightforward. $TP(U)$ can be simply estimated as $TP(U) = \sum_{C \in U} TP(C)$, and $FP(U)$ as $FP(U) = \sum_{C \in U} FP(C)$.

It is more complex when some $r$ has conflicting join assigning in $U$, with say $J_U(r) = I$ and $J_U(r') = I'$, where $l \neq l'$. Because we know each $r$ should only join with at most one $l \in L$ (as $L$ is the reference table), we use our precision estimate in Equation (9) to compare $\text{precision}(r, C_I)$ and $\text{precision}(r, C_J)$, and pick the more confident join as our final assignment. Other derived estimates like $\text{TP}(U)$ and $\text{FP}(U)$ can be updated accordingly.

Given $TP(U)$ and $FP(U)$, the estimated precision/recall of $U$ is:

$$\text{precision}(U) = \frac{TP(U)}{TP(U) + FP(U)}; \text{recall}(C) = \frac{TP(U)}{TP(U)}$$ (13)

### 3.2 AutoFJ Algorithm

Given the hardness result, we propose an intuitive and efficient greedy approach AutoFJ to solve the RM-FJ problem. Recall that our goal is to maximize recall while keeping precision above a certain threshold $\tau$, where precision and recall can be estimated according to Equation (13). A greedy strategy is then to prefer configurations that can produce the most number of true-positives (TP), i.e., maximal recall, at the “cost” of introducing as few false-positives as possible (FP), i.e., minimal precision loss. We call this ratio of TP to FP “profit” to quantify how desirable a solution is:

$$\text{profit}(U) = \frac{TP(U)}{FP(U)}$$ (14)

Given a space of possible configurations $S$, our greedy algorithm in Algorithm 1 starts with an empty solution $U$ (Line 5). It iteratively finds the configuration from the remaining candidates in $S \setminus U$, whose addition into the current $U$ leads to the highest profit (Line 9). The entire greedy algorithm terminates when the estimated precision of $U$ falls below the threshold $\tau$ (Line 14) or there is no remaining candidate configurations (Line 6).

### Efficiency Optimizations

We perform two main optimizations to improve efficiency of the greedy algorithm. First, we pre-compute $\text{profit}(r, C) \forall r \in R, C \in S$ based on given $L$ and $R$, as opposed to computing these measures repeatedly in each iteration.

Second, we apply blocking [14, 19, 41] to avoid comparing all record pairs. However, unlike standard blocking, we could not expect users to tune parameters in the blocking component (e.g., tokenization schemes, what fraction of tokens to keep, etc.) based on input data, precisely because our goal is to have end-to-end hands-off Auto-FJ JOIN. Instead of performing automated parameter-tuning for blocking, we use a default blocking that is empirically effective: we use 3-gram tokenization to tokenize each record and use TF-IDF weighting schema to weight each token; we measure the similarity between each $l$ and $r$ by summing the weights of their common tokens; for each $r$, we keep the top $|L|$ number of candidates from $L$ with the largest similarity scores and block others. As we will show in experiments, our default blocking strategy significantly reduces running time with almost zero loss in recall.

### Complexity of Algorithm 1

Since the number of $L$ and $L$-records after blocking is $|L|\sqrt{|L|} + |R|\sqrt{|L|}$, it takes $O(|S||L|\sqrt{|L|} + |R|\sqrt{|L|})$ to compute the distance (Line 3). To compute $\text{precision}(r, C)$, we need to first find $l \in L$ closest to $r$, then we need to find $l' \in L$ that have distance smaller than 20 with $l$. Since after blocking, for each $r \in R$ or $l \in L$, we have $|L|$ records in the candidate set. Hence, the time complexity for computing the $\text{precision}(r, C)$ is $O(\sqrt{|L|})$ and the complexity for the pre-computing step (Line 4) is $O(|S||R|\sqrt{|L|})$. At each iteration, with our pre-computation, it takes $O(1)$ time to compute the profit for each configuration (Line 9). Therefore, the time complexity of greedy steps (Lines 6 and 14) is $O(\sqrt{|L|}|S||R|\sqrt{|L|})$. The space complexity is dominated by computing distance between tuple pairs, which is in $O(|S||L|\sqrt{|L|} + |S||R|\sqrt{|L|})$.

### 3.3 Learning of Negative-Rules

While tuning fuzzy-join parameters is clearly important and useful, we observe that there is an additional opportunity to improve join quality not currently explored in the literature.

Specifically, in many real datasets there are record pairs that are syntactically similar but should not join. For example, in Figure 3(a), $(l_8, r_5)$ with “2007 LSU Tigers football team” and “2007 LSU Tigers baseball team” should not join despite their high similarity, because as human we know that “football” ≠ “baseball”. Similarly $(l_7, r_7)$

---

If there are multiple configurations with the same profit at an iteration, which rarely happens on large datasets, we break ties randomly.
Algorithm 2: Learning and Applying Negative Rules

Require: Tables $L$ and $R$, $LL$ and $LR$

1: Apply lowercasing, stemming, and removing punctuation for all $L$ and $R$
2: $NR ←∅$
3: for $l, r ∈ LL$ do
4:     $W_l ←$ set of words of $l$, $W_r ←$ set of words of $r$
5:     $Δ_l = |W_l \setminus W_r|$, $Δ_r = |W_r \setminus W_l|$
6:     if $|Δ_l| = 1$ and $|Δ_r| = 1$ then
7:         $NR ← NR ∪ (Δ_l, Δ_r)$
8:     if $l, r ∈ LR$ do
9:         $W_l ←$ set of words of $l$, $W_r ←$ set of words of $r$
10:        $Δ_l = |W_l \setminus W_r|$, $Δ_r = |W_r \setminus W_l|$
11:        if $|Δ_l| = 1$ and $|Δ_r| = 1$ and $(Δ_l, Δ_r) ∈ NR$ then
12:            Remove $(l, r)$ from $LR$
13:     return $LR$

with "2007 Wisconsin Badgers football team" and "2008 Wisconsin Badgers football team" should not join, since "2007 ≠ 2008".

Such negative rules are often dataset-specific with no good "global" rules to cover diverse data. Our observation is that we can leverage reference table $L$ to "learn" such negative rules — if a pair of records in the $L$ table only differ by one pair of words, then we learn a negative rule from that pair. The learned negative rules can then be used to prevent false positives in joining $L$ and $R$.

**Definition 3.1.** Let $l_1, l_2 ∈ L$ be two reference records, $W(l_1)$ and $W(l_2)$ be the set of words in the two records, respectively. Denote by $Δ_{12} = W(l_1) \setminus W(l_2)$, and $Δ_{21} = W(l_2) \setminus W(l_1)$. We learn a negative rule $NR(Δ_{12}, Δ_{21})$, if $|Δ_{12}| = 1$ and $|Δ_{21}| = 1$.

Note that since $L$ is a reference table with little or no duplicates, the negative rules we learned intuitively capture different "identifiers" for different entities of the same entity type.

We summarize the algorithm for learning and applying negative rules in Algorithm 2. The inputs are the $L$-$L$ and $L$-$R$ tuple pairs that survive in the blocking step. The tuples will be first preprocessed by lowercasing, stemming and removing punctuation (Line 1). The algorithm will then learn negative rules from $L$-$L$ tuple pairs (Line 2 to Line 6). Then it applies the learned negative rules on $L$-$R$ tuple pairs (Line 7 to Line 11), where the tuple pairs that meet the negative rules will be discarded and will not be joined. While negative-rule learning can be applied broadly regardless of whether fuzzy-joins are auto-tuned or not, in the context of Auto-FuzzyJoin our experiments show that it provides an automated way to improve join quality on top of automated parameter tuning.

**4 MULTI-COLUMN AUTO-FUZZYJOIN**

We now consider the more general case, where the join key is given as multiple columns, or when the join key is not explicitly given, in which case our algorithm has to consider all columns.

Figure 5 shows an example of two movie tables with attribute like names, directors, etc. Intuitively, we can see that names and directors are important for fuzzy-join, but not descriptions. Users may either select name and director as key columns for Auto-FuzzyJoin, or may provide no input to the algorithm. In either case, the algorithm has to figure out what columns to use and their relative "importance" in making overall fuzzy-join decisions.

**4.1 Multi-Column Join Configuration**

Given that multiple columns may have different relative "importance", we extend single-column configuration $C = (f, θ)$ as follows. We define a *join function vector* as $F = (f^1, f^2, ..., f^m)$, where

$f^j ∈ F$ is the join function used for the $j$th column pair. In addition, we define a *column-weight vector* as $w = (w^1, w^2, ..., w^m)$, where $w^j ∈ [0, 1]$ is the weight associated with $j$th column pair.

Let $l[j]$ and $r[j]$ be the value in $j$th column of record $l$ and $r$, respectively. Given $F$ and $w$, the distance between $l$ and $r$ is computed as the sum of weighted distances from all columns:

$$F_w(l, r) = \sum\limits_{j=1}^m w^j f^j(l[j], r[j])$$

**Definition 4.1.** A multi-column join configuration is a 3-tuple $(F, w, θ)$, where $F ∈ F^m$ is a join function vector, $w ∈ R^m$ is a column-weight vector, and $θ ∈ R$ is a threshold.

Let $S = \{(f, w, θ) | F \in F^m, w \in R^m, θ ∈ R\}$ be the space of possible multi-column join configurations. A multi-column join configuration $C ∈ S$ induces a fuzzy join mapping $J^C(r)$ for each $r ∈ R$, defined as:

$$J^C(r) = \arg\min_{l ∈ L} F_w(l, r), \forall r ∈ R$$

(15)

**4.2 Multi-Column AutoFJ**

Given the space of multi-column configurations $S$, the AutoFuzzyJoin problem is essentially the same as RM-FJ in the single-column setting: we want to find a set of configuration $U ∈ 2^S$ that maximizes the recall($U$), while having precision($U$) ≥ $τ$.

A naive approach is to invoke the single-column fuzzy join solution in Algorithm 1 with the multi-column join configuration space $S$. However, such a simple adaptation is not practical, because the new multi-column search space is exponential in the number columns $m$ (each column has its own space of fuzzy-join configurations, which can combine freely with configurations from other columns). Exploring this space naively would be too slow.

Our key observations here is that (1) given a wide table, there are often only a few columns that contribute positively to the overall fuzzy-join decisions; (2) the relative "importance" of these useful columns is often a static property, which depends only on the data and task at hand, and is independent of the search algorithm used. For example, in Figure 5, the fact that column "names" is the most important, "directors" is less important, and "description" is not important, "directors" is less important, and "description" is not important, would hold true irrespective of the distance-functions used and/or the set of input columns considered.

We therefore propose a multi-column AutoFJ algorithm shown in Algorithm 3, which is inspired by the forward selection approach to feature selection in machine learning [16]. At a high-level, our algorithm starts from an empty set of join column (Line 1), and iteratively expands this set by adding the most important column from the remaining columns (Line 4 to Line 9). The importance of a candidate column is determined by the resulting join quality after adding it, which can be estimated using techniques from Section 3.1 (Line 7 to Line 9). The algorithm terminates when the join quality cannot be improved by adding an extra column (Line 13) or there is no remaining columns (Line 3). This adding one-column-at-a-time approach is reminiscent of forward selection [16].

In addition, as the set of candidate columns expands, instead of searching for the column-weight vector $w$ blindly (which would...
The effectiveness, efficiency, and robustness of fuzzy-Join algorithms, using the standard precision (P) and recall (R) metrics, defined in Equation (3) and (4) of Section 2. Recall that AUTOFJ automatically produces a solution that maximizes recall while meeting a certain precision target. In comparison, existing fuzzy-join approaches usually output (their own version of) similarity/probability scores for each tuple pair, and ask users to pick the right threshold. In order to compare, for each existing method, we search for the similarity (probability) threshold that would produce a precision score that is “closest to but not greater than” AUTOFJ, and report the corresponding recall score (which favors baselines). We call this recall score adjusted recall (AR).

For example, suppose AUTOFJ produces results with precision 0.91, recall 0.72. Suppose an existing baseline produces the following (P, R) values at different threshold-levels: (0.8, 0.8), (0.9, 0.7), (0.92, 0.6), (0.95, 0.5)). The adjusted recall (AR) for this baseline will be reported as 0.7, for its corresponding precision (0.9) is “closest to but not greater than” the 0.91 precision produced by AUTOFJ. We can see that this reported AR clearly favors the baseline, but allows us to compare recall at a fixed precision target.

In addition to the AR, we also measure the quality of fuzzy-joins using Precision-Recall AUC score (PR-AUC), defined as the entire area under the Precision-Recall curves. This is a standard metric that does not require the thresholding procedure above.

### Single-Column Fuzzy Join Algorithms

- **AUTOFJ.** This is our method, and we use target precision \( \tau = 0.9 \), the step size for discretizing numeric parameters \( s = 50 \). Table 1 lists the parameter values used in experiments (c.f. Figure 2). In total, we consider 4 options for preprocessing, 2 for tokenization and 2 for token weights. For distance function, we consider 2 character-based distance, 8 set-based distance and 1 embedding distance. Among the 8 set-based functions, the first 5 are standard functions; while the last 3 are hybrid ones we added. In total we have \( 4 \times 2 + 4 \times 2 \times 2 \times 5 \).

### Table 1: Parameter Options Considered in the Experiments

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preprocessing</td>
<td>L, L<em>5, L</em>50, L*500</td>
</tr>
<tr>
<td>Tokenization</td>
<td>3G, SP</td>
</tr>
<tr>
<td>Token Weights</td>
<td>DW, RSPW</td>
</tr>
<tr>
<td>Distance Function</td>
<td>Character-based, Set-based, Contain-Jaccard, Contain-Cosine, Contain-Dice Distance</td>
</tr>
<tr>
<td>Embedding</td>
<td>TEDD</td>
</tr>
</tbody>
</table>

We design three hybrid distance functions named Contain-Jaccard, Contain-Cosine and Contain-Dice. If two records have containment relationship (i.e. \( r \subseteq h \), they are equivalent to the standard distance functions. Otherwise: output 1.}

---

**Algorithm 3 AutoFJ for multiple columns**

<table>
<thead>
<tr>
<th>Require:</th>
<th>Tables L and R, precision target ( \tau ), and search space ( S )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. ( U \leftarrow \emptyset ), ( U^* \leftarrow \emptyset ), ( w \leftarrow (0, 0, \ldots 0) ).</td>
<td></td>
</tr>
<tr>
<td>2. ( E \leftarrow {e_1, e_2, \ldots e_m } ), where ( e_j ) is a ( m )-dimensional vector with the ( j )-th position set to 1 and the rest to 0.</td>
<td></td>
</tr>
<tr>
<td>3. while ( E \neq \emptyset ) do</td>
<td></td>
</tr>
<tr>
<td>4. for all ( e_j \in E ) do</td>
<td></td>
</tr>
<tr>
<td>5. for all ( a \in {\frac{1}{p}, \frac{2}{p}, \ldots \frac{p-1}{p} } ) do</td>
<td></td>
</tr>
<tr>
<td>6. ( w' \leftarrow (1-a)w + ae_j ).</td>
<td></td>
</tr>
<tr>
<td>7. ( U^* \leftarrow \text{invoke Algorithm 1 with weight vector } w' ).</td>
<td></td>
</tr>
<tr>
<td>8. if recall(( U^* ) ) ( &gt; ) recall(( U ) ) then</td>
<td></td>
</tr>
<tr>
<td>9. ( U \leftarrow U^* ), ( w \leftarrow w' ), ( E \leftarrow E \setminus e_j ).</td>
<td></td>
</tr>
<tr>
<td>10. else</td>
<td></td>
</tr>
<tr>
<td>11. break</td>
<td></td>
</tr>
<tr>
<td>12. return ( U ).</td>
<td></td>
</tr>
</tbody>
</table>

Again be exponential in \( m \), we leverage the fact that column importance is a static property of the data set (Observation (2) above), and thus in each iteration we “inherit” column-weights from previous iterations, and further scale them linearly relative to the observed importance of the new column added in this iteration (Line 6).

**Complexity of Algorithm 3**. The search algorithm in Algorithm 3 invokes single-column AUTOFJ in \( O(m^2g) \) times (where \( m \) is the number of input columns, and \( g \) the discretization step for weights), which is substantially better than the naïve \( O(g^m) \) we started with. Hence, the time complexity is \( O(m^2g(|L||L| + |S||R|\sqrt{|L|}) \). Its space complexity is \( O(m(|L| + |S||R|\sqrt{|L|}) \) since we need to precompute distances for all \( m \) columns. In practice, we observe that it terminates after a few iterations (only selecting a few columns from a wide table). This, together with other optimizations we propose, makes multi-column AUTOFJ very efficient.

**5 EXPERIMENTS**

We evaluate the effectiveness, efficiency, and robustness of fuzzy-join algorithms. All experiments are performed on a machine with two Intel Xeon E5-2673 v4 CPUs at 2.30GHz and 256GB RAM.

### 5.1 Single-Column Auto-FuzzyJoin

#### 5.1.1 Datasets

We constructed 50 diverse fuzzy-join datasets using DBPedia [33]. Specifically, we obtained multiple snapshots of DBPedia\(^6\) (from year 2013, 2014, 2015, 2016, etc.), which are harvested from snapshots of Wikipedia over time. Each entity in a DBPedia snapshot has a unique “entity-id”, an “entity-name” (from Wikipedia article titles), and an “entity-type” (e.g., Political Parties, Soccer Leagues, NCAA teams, Politicians, etc., which are extracted from Wikipedia info-boxes). Because these entity-names are edited by volunteers, their names can have minor changes over time (e.g., “2012 Wisconsin Badgers football team” and “2012 Wisconsin Badgers football season” are the titles/entity-names used in two different snapshots, referring to the same Wikipedia article/entity because they share the same unique “entity-id”).

For each DBPedia snapshot from a specific year, and for each entity-type, we build a table with names of all entities in that type (e.g., NCAA-Teams from the snapshot in year 2013). Two tables of the same type from different years can then be used as a fuzzy-join task (e.g., NCAA-Teams in year 2013 vs. 2016). Because the entity-id of the mentioned entities do not change over time, it allows us to automatically generate fuzzy-join ground-truth using entity-id.

We randomly select 50 entity-types for benchmarking. We use the 2013 snapshot as \( L \), and use the union of all other snapshots as \( R \), which would create difficult cases where multiple right records join with the same left record, as well as cases where a right record has no corresponding left record. We further remove equi-joins from all datasets that are trivial for fuzzy joins. These 50 data sets and their sizes are shown in the leftmost two columns of Table 2. We released this benchmark together with our AUTO-FUZZYJOIN code on GitHub\(^9\) to facilitate future research.

#### 5.1.2 Evaluation Metrics

We report quality of Fuzzy-Join algorithms, using the standard precision (P) and recall (R) metrics, defined in Equation (3) and (4) of Section 2. Recall that AUTOFJ automatically produces a solution that maximizes recall while meeting a certain precision target. In comparison, existing fuzzy-join approaches usually output (their own version of) similarity/probability scores for each tuple pair, and ask users to pick the right threshold. In order to compare, for each existing method, we search for the similarity (probability) threshold that would produce a precision score that is “closest to but not greater than” AUTOFJ, and report the corresponding recall score (which favors baselines). We call this recall score adjusted recall (AR).

For example, suppose AUTOFJ produces results with precision 0.91, recall 0.72. Suppose an existing baseline produces the following (P, R) values at different threshold-levels: \( \{(0.8, 0.8), (0.9, 0.7), (0.92, 0.6), (0.95, 0.5)\} \). The adjusted recall (AR) for this baseline will be reported as 0.7, for its corresponding precision (0.9) is “closest to but not greater than” the 0.91 precision produced by AUTOFJ. We can see that this reported AR clearly favors the baseline, but allows us to compare recall at a fixed precision target.

In addition to the AR, we also measure the quality of fuzzy-joins using Precision-Recall AUC score (PR-AUC), defined as the entire area under the Precision-Recall curves. This is a standard metric that does not require the thresholding procedure above.

#### 5.1.3 Single-Column Fuzzy Join Algorithms

- **AUTOFJ.** This is our method, and we use target precision \( \tau = 0.9 \), the step size for discretizing numeric parameters \( s = 50 \). Table 1 lists the parameter values used in experiments (c.f. Figure 2). In total, we consider 4 options for preprocessing, 2 for tokenization and 2 for token weights. For distance function, we consider 2 character-based distance, 8 set-based distance and 1 embedding distance. Among the 8 set-based functions, the first 5 are standard functions; while the last 3 are hybrid ones we added. In total we have \( 4 \times 2 + 4 \times 2 \times 2 \times 5 \).

---

\(^6\)http://downloads.dbpedia.org/

\(^9\)https://github.com/chu-data-lab/AutoFuzzyJoin

\(^10\)https://github.com/explosion/spacy-models/releases/tag/en_core_web_lg-2.3.0
2 × 8 + 4 × 1 = 140 join functions (note that the tokenization and token-weight parameters are only applicable to set-based distance).

- **Best Static Join Function (BSJ).** In this method, we evaluate the join quality of each individual join function from the space of 140 discussed above. We compute the Adjusted-Recall (AR) score of each join function on each data set, and report the join function that has the best average AR over 50 datasets. This can be seen as the best static join function, whereas AutoFJ produces dynamic join functions (different datasets can use different join functions).

- **Excel.** This is the fuzzy-join feature in Excel. The default parameter setting is carefully engineered and uses a weighted combination of multiple distance functions.

- **FuzzyWuzzy (FW).** This is a popular open-source fuzzy join package with 5K+ stars. It produces a score for every tuple pair based on an adapted and fine-tuned version of the edit distance.

- **ZEROER** [48]. This is a recent unsupervised entity resolution (ER) approach that requires zero labeled examples. It uses a generative model that is a variant of a Gaussian Mixture Model to predict the probability of a tuple pair being a match. The features used in ZEROER are generated by the MAGELLAN [31] package.

- **ECM** [23]: This is an unsupervised approach with the Fellegi and Sunter framework [28]. We use the implementation from [24] that uses binary features and Expectation-Conditional Maximization (ECM) algorithm. The features are generated by the MAGELLAN [31] package and binarized using the mean value as the threshold.

- **PPJoin (PP)** [49]: This is a set similarity join algorithm that employs several filtering techniques to optimize efficiency. We use an existing implementation and use Jaccard similarity.

- **MAGELLAN** [31]: This is a supervised approach that uses conventional ML models based on similarity values as features. We use the open-source implementation with random forest as the model. For each dataset, we randomly split the data into a training and a test set by 50%-50%. Note that 50% training data is generous given that the amount of available labeled data is usually much smaller in practice. The reported AR are the average results over 5 runs.

- **DeepMatcher (DM)** [40]: This is a supervised approach that uses a deep learning model with learned record embedding as features. We use the same setup and MAGELLAN in terms of train/test split. We use the open-source implementation with its default model.

- **Active Learning (AL).** This is an active learning based supervised approach. The algorithm interactively queries users to label new tuple pairs until 50% joined pairs in the data are labeled. We use the implementation from modAL [20] with default query strategy, and we use the same model and features as MAGELLAN.

- **Upper Bound of Recall (UBR).** There are many ground-truth pairs in L and R that are difficult for fuzzy-joins (e.g., (“Lita (wrestler), “Amy Dumas”), (“GLYX-13”, “Rapastinel”), etc.). These pairs have semantic relationships that are out of the scope of fuzzy-joins. To test the true upper-bound of fuzzy-joins, for each r we find its closest l ∈ L using all possible configurations C ∈ S, which collectively is the set of fuzzy-join pairs that can be produced. We call a ground-truth pair (l, r) feasible if it is in the set, and report the recall using all feasible ground-truth pairs. This gives us a true upper-bound of fuzzy-join on these data sets.

5.1.4 Single-Column Fuzzy Join Evaluation Results.

Overall Quality Comparison. Table 2 shows the overall quality comparison between AutoFJ and other approaches on 50 datasets. The average precision of AutoFJ is 0.886, which is very close to the target precision  𝑠  = 0.9. We compute the Pearson correlation coefficient between the actual precision and the estimated precision (PEPCC) over AutoFJ iterations for each dataset. As we can see in Table 2, the average PEPCC over all datasets is 0.894, which shows that the actual/estimated precision match well across iterations.

The average recall of AutoFJ is 0.624. That the average recall upper bound (UBR) is 0.834. AutoFJ produces about 75% of correct joins that can possibly be generated by any fuzzy-join program. As we can see, AutoFJ outperforms all other approaches on 21 out of 50 datasets. On average, the recall of AutoFJ is 0.062 better than Excel, the best among all unsupervised approaches, and 0.129 better than AL, the best among all supervised approaches that use 50% of joins as training data. To test the statistical significance of this comparison, we perform an upper-tailed T-Test over the 50 datasets, where the null hypothesis (H₀) states that the mean of AutoFJ’s recall is no better than that of a baseline’s AR. As shown in the second last row of Table 2, the p-values of all baselines are smaller than 0.003, showing that the differences are significant.

The last row of Table 2 shows the average PR-AUC scores of AutoFJ and other methods over 50 datasets. As we can see, the PR-AUC of AutoFJ is on average 0.057 better than Excel, the strongest unsupervised method, and 0.056 better than MAGELLAN, the method with the highest PR-AUC score among all supervised methods. This indicates that AutoFJ can outperform other baselines across different precision levels. The details of PR-AUC scores on each dataset can be found at the full version of this paper [36], where we show that AutoFJ outperforms all other methods in terms of PR-AUC on 28 out of 50 datasets.

Among all unsupervised baselines, Excel, as a commercial-grade tool that features carefully engineered weighted combination of multiple distance functions, performs the best. In fact, Excel is even better than BestStaticJF, the best statistic configuration tuned on the 50 datasets. We also observe that FW and ZEROER has generally worse performance than Excel and BestStaticJF, because FW and ZEROER use predetermined sets of similarity functions while Excel and BestStaticJF have various degrees of feature engineering. ECM and PPJoin under-perform other unsupervised methods, because ECM binarizes features and loses information, while PPJoin uses vanilla Jaccard similarity.

Among all supervised baselines that use 50% all joins as training data, AL performs best based on AR as it carefully selects training examples. The deep model DM performs poorly, which is not surprising as deep learning approaches typically require a large number of labeled examples to perform well.

It is also worth highlighting that AutoFJ (and Excel) outperforms the best supervised baseline even when 50% of all ground-truth labels are used as training data.

Ablation Study (1): Contribution of Union of Configurations. To study the benefit of using a set of configurations, we compare AutoFJ with AutoFJ-UC that only uses one single best configuration. Note that the single configuration selected by AutoFJ-UC can be different for each dataset. The column AutoFJ-UC in Table 2

---

10https://github.com/seatgeek/fuzzywuzzy
11https://github.com/usc-ist-i2/ppjoin
shows the quality of the best single configuration on each dataset. The average adjusted recall is 0.575, which is 0.049 lower than AutoFJ, but still higher than all other methods. This suggests that (1) dynamically using a single configuration is better than using any static configuration; and (2) dynamically selecting a union of configurations can further boost the performance.

Ablation Study (2): Contribution of Negative Rules. The column AutoFJ-NR in Table 2 shows the AR results of AutoFJ without negative rules. As we can see, without negative-rules, the average AR decreases to 0.608, which shows the benefit of negative-rules.

Robustness Test (1): Adding Irrelevant Records to the Right Table. We construct an adversarial test to evaluate the robustness of AutoFJ as follows. For each dataset, we insert irrelevant records to the R by randomly picking records from other datasets. Figure 6(a) shows the average precision and recall over 50 datasets with different amounts of irrelevant records. As we can see, even when 80% of records in the R are irrelevant, AutoFJ can still achieve an average precision of around 84% with recall almost unaffected.

Robustness Test (2): Zero Fuzzy Joins. We construct a second adversarial test, where the L and R are taken from different entity-type that are completely unrelated (e.g., L from "Satellites" and R from "Hospitals"), such that any joins produced are false positives. We construct 10 such cases. Figure 6(b) shows the false positive rate (defined as the number of false positives divided by the size of R) of AutoFJ and Excel, the best baseline. In all cases, the false positive rate of AutoFJ is below 5% and much smaller than Excel.

Robustness Test (3): L Incompleteness. In this work, we do not assume the reference table L to be complete, and we take this into account when estimating precision (c.f. Equation (9)). However, an extremely sparse L can affect our estimation. To test its robustness, we make the already incomplete L even more sparse by randomly removing records in L. Figure 6(c) shows the average performance when L is randomly removed.

Table 2: Performance evaluation on 50 single-column fuzzy join datasets.
of AutoFJ and Excel across 50 datasets with different amounts of records removed from L tables. As expected, the average precision decreases as L becomes more sparse. However, even with 30% L records removed, AutoFJ can still achieve precision of 0.81. In all cases, the recall of AutoFJ is still at least 0.051 higher than Excel.

**Sensitivity to Blocking.** Figure 6(d) shows the average running time of 50 datasets varying the blocking factor β, where \( β \times L \) is the number of left records kept for each right record. A smaller β gives faster algorithms, but potentially at the cost of join quality. As we can see, after β exceeds 1.0, e.g., we keep top 1.0 × 100 = 100 records for each right record if \( L = 100 \), the performance of AutoFJ remains almost unchanged even if we increase β further.

**Varying Target Precision.** Figure 7(a) shows the average precision and recall on 50 datasets, as we vary the precision target τ. Note that the two align very well (the correlation-coefficient of the two is 0.9939), suggesting that our precision estimation works as intended. Compared to other baseline methods, our method remain the best as we vary the target, and our algorithm consistently outperforms Excel, the strongest baseline, by at least 0.062.

**Efficiency Analysis.** Overall, AutoFJ finishes 15/50 data sets in 30 seconds, 33/50 in 1 minute, and 49/50 in 130 seconds. To compare the running time of AutoFJ with other methods, we bucketize 50 datasets into 5 groups based on the size of \( L \times R \). Figure 7(b) shows the average running time of AutoFJ and other methods over datasets in each group. As we can see, the running time of AutoFJ is comparable to other methods. PPJoin is the fastest method since it employs an efficient version of Jaccard similarity. DM is on average 10 times slower than other methods because it needs to train deep neural networks. AutoFJ is on average 2-3 times slower than ECM and Excel, but faster than ZeroER, Magellan and FW. AutoFJ is 2-3 times faster than AL.

**Varying Configuration Spaces.** We run AutoFJ using a varying number of configurations from the space listed in Table 1. The reduced configuration space is achieved by removing some options for the 4 parameters. For example, if we only use L and L+S+SR for pre-processing instead of all four options, the space reduces to 70 from 140. Figure 7(c) shows the average performance of AutoFJ over 50 datasets with different size of the configuration space. As we can see, the average precision is almost unchanged as we vary the space size, showing the accuracy of our precision estimation. The average recall decreases slightly with a smaller number of configurations, because the expressiveness of fuzzy-matching is reduced accordingly. We compute the AR of Excel and Magellan using the precision of AutoFJ with different configuration space.

As we can see, even with 24 configurations, the recall of AutoFJ is still 0.036 higher than Excel and 0.105 higher than Magellan.

Figure 7(d) shows the running time of each component of AutoFJ as we vary the configuration space. As we can see, the running time is greatly reduced as the configuration space shrinks. With 24 configurations, the algorithm becomes 2 times faster than using 140 configurations. Also, as we can see in Figure 7(d), the pre-computation for precision takes less than 10% of the overall time. In contrast, if we compute this repeatedly at every iteration (e.g., with 140 configurations, there are about 45 iterations on average for each dataset), our overall running time can be 6x slower (with this component taking 85% time).

### 5.2 Multi-Column Auto-FuzzyJoin

#### 5.2.1 Multi-Column Datasets

For multi-column fuzzy joins, we use 8 benchmark datasets in the entity resolution literature [31, 32, 40], as shown in Table 3.

As we can see, even with 24 configurations, the recall of AutoFJ is still 0.036 higher than Excel and 0.105 higher than Magellan.

#### 5.2.2 Multi-Column Fuzzy Join Algorithms

- **AutoFJ.** This is our proposed Algorithm 3, using precision target \( τ = 0.9 \), discretization steps \( s = 50 \), and the column-weight search steps \( g = 10 \). Given 140 join functions, and a table with \( m \) columns, we can in theory have as many as \( 140^m \) configurations. In our experiments, we add an additional constraint that distance functions considered in the same configuration should be the same across all columns. This is for efficiency considerations, but nevertheless produces fuzzy-joins with state-of-the-art quality. To handle missing values in the datasets, we treat missing values as empty strings, and assign maximum distances when comparing two missing values.

- **Excel, FW, ZeroER, ECM and PP, Magellan, DM, AL.** These are the same methods as we described in Section 5.1.3. Since Excel, FW and PP handle all columns in the same way, we invoke these methods with all columns concatenated.

#### 5.2.3 Multi-Column Fuzzy Join Evaluation Results

**Overall Quality Comparison.** Table 4(a) shows the overall quality comparison between AutoFJ and other methods on multi-column datasets. As we can see, AutoFJ remains the best method on average in the multi-column joins. The recall of AutoFJ on average is 0.13 better than Excel, the strongest unsupervised baseline,
and 0.014 better than AL, the strongest supervised method. AutoFJ outperforms all other methods on 3 out of 8 datasets and achieves comparable results to the best baseline on the remaining datasets. We perform upper-tailed T-Test to verify the statistical significance with the exception of [10, 11, 13, 19, 22, 25, 35, 39, 42, 43, 45, 46, 50].

The last row of Table 4(a) shows the average PR-AUC of AutoFJ and other methods. As we can see, AutoFJ significantly outperforms all other unsupervised methods and achieve comparative performance compared to supervised methods such as Magellan and AL that uses 50% joins as training data. The PR-AUC on each dataset can be found in the full version [36] of this paper.

Effectiveness of Column Selection. Table 4(a) reports the columns selected by AutoFJ and their corresponding weights. Observe that the selected columns are indeed informative attributes, such as Name and Director in Rotten Tomatoes-IMDB (RI) dataset (with Name being more important). Also note that AutoFJ is able to achieve these results typically using only one or two columns.

Robustness Test: Adding Random Columns. We test the robustness of AutoFJ on multi-column joins by adding adversarial columns with randomly-generated strings in both L and R tables. The length of each random string is between 10-50. Table 4(b) shows the change of performance of AutoFJ, Excel and AL after adding random columns. Since random columns do not provide any useful information, they are not selected by AutoFJ, and hence have no effect on our results. In contrast, as Excel and AL use all input columns, adding random columns does affect their results.

6 RELATED WORK

Fuzzy join, also known as entity resolution and similarity join, is a long-standing problem in data integration [26, 27], with a long line of research on improving the scalability of fuzzy-join algorithms (e.g., [10, 11, 13, 19, 22, 25, 35, 39, 42, 43, 45, 46, 50]).

Existing state-of-the-art in optimizing join quality are predominantly supervised methods (e.g., Magellan [31] and DeepMatcher [40]), which require labeled data of matches/non-matches to be provided before classifiers can be trained. In contrast, our proposed Auto-FuzzyJoin is unsupervised and mainly leverages structural properties of reference tables. Surprisingly, this unsupervised approach outperforms supervised methods even when 50% of ground-truth labels are used as training data.

Among unsupervised methods, our evaluation suggests that the carefully-tuned Excel (with default settings) is a strong baseline. It employs a variant of the generalized fuzzy similarity [17], which is a weighted combination of multiple distance functions. The weights and pre-processing parameters are carefully-tuned on English data.

Other entity-matching approaches include AutoEM [51] and Ditto [37], which uses pre-trained entity-type-models and language-models for entity-matching, respectively. ZeroER [48] is a recent unsupervised method that uses a predetermined set of features and Gaussian Mixture Model to determine matches.

Additional methods to facilitate complex table joins include methods that leverage search engines [15, 30, 34], and program-synthesis [47, 52].

7 CONCLUSIONS

In this paper, we propose an unsupervised Auto-FuzzyJoin to auto-program fuzzy joins without using labeled examples. We formalized this as an optimization problem that maximizes recall under a given precision constraint. Our results suggest that this unsupervised method is competitive even against state-of-the-art supervised methods. We believe unsupervised fuzzy entity matching is an interesting area that is still under-studied, and clearly worthy attention from the research community.