Paper Review - Auto-FuzzyJoin: Auto-Program Fuzzy Similarity Joins Without Labelled Examples

Peng Li, Xiang Cheng, Xu Chu, Yeye He, Surajit Chaudhuri

Carmel Gafa

April 20, 2025

Fuzzy-join (or similarity join)

Left Table

id	Isem
l1	Peppi Azzopardi
l2	Annetto Depasquale
l3	Karmenu Vassallo

Right Table

id	Isem
r1	Karmnu Vassallo
r2	Ġużeppi Azzopardi
r3	Annetto De Pasquale

Left Table

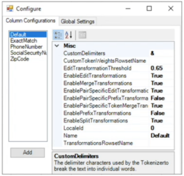
L-id	L-name	L-director	L-description
l1	Carrie	Brian De Palma	Carrie White is shy and outcast ...
l2	Vibes	Ken Kwapis	Psychics hired to find lost temple...

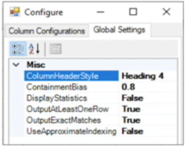
Right Table

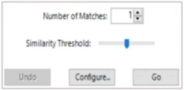
R-id	R-name	R-director	R-description
r1	Carrie	Brian DePalma	This classic horror movie based ...
r2	Vibes	Ken Kwapis	Two hapless psychics unwittingly...

- Fuzzy join takes two tables as inputs and identifies record pairs that refer to the same entity.
- As an example, l1 and r2 refer to the same person.
- The concept can be extended to records with multiple fields or attributes.

Fuzzy-join configuration







- Fuzzy-join has been integrated into many commercial applications
- These systems are often difficult to use due to the large number of configuration parameters.
- The extension in Microsoft Excel has 19 options that span across 3 dialog boxes.
  - 11 are binary, thus resulting in 2048 possible configuration scenarios.
  - 8 continuous, such as thresholds and biases.
- In order to execute quality Fuzzy-joins, these configurations require careful user setup to achieve high-quality results.

Theoretical foundation: fuzzy join mapping

Given a **reference table** L and a table R containing records that may be **imprecise** or noisy, a **fuzzy join mapping** J establishes approximate matches between them.

- J connects elements of R to similar elements in L based on a chosen **similarity measure** (e.g., Levenshtein distance, cosine similarity, Jaccard similarity).
- Each record $r \in R$ is mapped to at most one record $l \in L$, or **no match at all** (denoted by $\perp$).
- The join is **many-to-one** because multiple records in R can be associated with the same record in L, but each $r \in R$ has only **one** possible match.

Formally:

$$J : R \to L \cup \perp$$

Theoretical foundation: fuzzy join configuration space

A **fuzzy join** f compares two strings, r and l, by computing a distance score that reflects their similarity. The computation of this score is governed by a variety of parameters, forming a **parameter space**.

Each unique combination of these parameters defines a specific **join function** $f \in \mathcal{F}$, where $\mathcal{F}$ is the space of all possible join functions.

Example: fuzzy join distance score computation

Join Function: $f = (L, SP, EW, JD)$

- **L**: Lower-casing (Preprocessing)
- **SP**: Space Tokenization
- **EW**: Equal Weights
- **JD**: Jaccard Distance

Inputs:

- $l =$ `"2012 tigers lsu baseball team"`
- $r =$ `"2012 lsu baseball team"`

Tokenization (SP):

- $l \longrightarrow \{2012, \textit{tigers}, \textit{lsu}, \textit{baseball}, \textit{team}\}$
- $r \longrightarrow \{2012, \textit{lsu}, \textit{baseball}, \textit{team}\}$

Jaccard Distance:

- $A \cap B = \{2012, \textit{lsu}, \textit{baseball}, \textit{team}\} \longrightarrow |A \cap B| = 4$
- $A \cup B = \{2012, \textit{tigers}, \textit{lsu}, \textit{baseball}, \textit{team}\} \longrightarrow |A \cup B| = 5$
- Jaccard Similarity $= \frac{4}{5} = 0.8$
- Jaccard Distance $= 1 - 0.8 = 0.2$

Result: $f(l, r) = 0.2$

Theoretical foundation: threshold and join configuration

- Once the distance $f(l, r)$ is computed:
    - It is compared to a threshold **compared to a threshold** θ to decide whether to join the string pair l and r.
        - lower θ gives stricter matches
    - If $f(l, r) \leq \theta$, the pair is considered **a match**.
- Together, the function f and the threshold θ define what the authors call a **join configuration**:

$$C = \langle f, \theta \rangle$$

- This configuration encapsulates both:
    - How distance is computed.
    - When two strings are considered similar enough to be joined.

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

Theoretical foundation: fuzzy join mapping

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

$$J_C(r) = \underset{l \in L,\ f(l,r) \leq \theta}{\arg\min}\ f(l, r),\ \forall r \in R$$

That is

- For each record $r \in R$, find $l \in L$ that minimizes the distance $f(l, r)$, **only if** that distance is less than or equal to the threshold θ.
- If no such $l \in L$ exists such that $f(l, r) \leq \theta$, then $J_C(r)$ is maps to $\perp$ — i.e., no match for that record.

Theoretical foundation: the problem with single join configurations

Real-world data can exhibit **multiple types of variations simultaneously**, such as:

- **Typos**
- **Missing tokens**
- **Extraneous information**

As a result, relying on a **single join configuration** often fails to capture all valid matches, particularly when high **recall** is required.

To handle this diversity, the algorithm uses a **set of join configurations**:

$$U = \{C_1, C_2, \ldots, C_K\}$$

Instead of relying on a single configuration, the system computes join results from each one.
This approach allows the system to:

- Accommodate diverse types of variations.
- Improve overall recall by **combining multiple perspectives** on similarity (different parametrizations that are sensitive to different types of noise).

L-id	L-Table (Reference Table)		R-id	R-Table (Input Table)
l_1	2008 LSU Tigers baseball team	✓	r_1	2008 LSU baseball team
l_2	2008 LSU Tigers football team	✓	r_2	2008 LSU football team
l_3	2008 Mississippi State Bulldogs baseball team	✓	r_3	2008 Missisippi State Bulldog baseball team
l_4	2008 Mississippi State Bulldogs football team	✓	r_4	2008 Mississippi State Bulldog football team
l_5	...		r_5	...
l_6	2007 LSU Tigers football team	✗	r_6	2007 LSU Tigers baseball team
l_7	2007 Wisconsin Badgers football team	✗	r_7	2008 Wisconsin Badgers football team
l_8	2011 LSU Tigers football team	✗	r_8	2010 LSU Tigers football team
l_9	2007 LSU Tigers baseball team	✗	r_9	2007 LSU Tigers football team

- A **Jaccard distance** with threshold 0.2 works well for pairs like (l_1, r_1), which differ by only one or two tokens.
- However, for pairs like (l_3, r_3) with **spelling variations**, Jaccard similarity is not enough:
  - Jaccard distance $\approx 0.5 \rightarrow$ too high to match under the 0.2 threshold
  - A more suitable metric is **Edit Distance**, which can better align such pairs.

Theoretical foundation: fuzzy join via multiple configurations

- To handle diversity, the algorithm uses a **set of join configurations**:

$$U = \{C_1, C_2, \ldots, C_K\}$$

- Instead of relying on a single configuration, the system computes join results from each.

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:

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

This means that the overall result of the fuzzy join using configuration set U is the **union** of results from all individual configurations $C_i \in U$.

Each configuration $C_i \in U$ is designed to capture a **specific type of string variation** (e.g., typos, missing tokens, extra tokens).
Two records are considered **joined by the set U if and only if** they are joined by **at least one** configuration $C_i \in U$.

- Each configuration contributes **high-quality joins** targeted at particular data challenges.
- The overall join is more **robust and comprehensive**.

Theoretical foundation: evaluating join quality; Precision

Given two tables R and L, and a **space of join configurations** $\mathcal{S}$, the objective is to find a subset $U \subseteq \mathcal{S}$ that produces **good fuzzy join results**.
Let:

- J_U be the fuzzy join mapping induced by configuration set U
- J_G be the **ground truth** join mapping — the ideal join result

Precision measures how many of the predicted joins are correct:

$$\text{precision}(U) = \frac{\underbrace{|\{r \in R \mid J_U(r) \neq \emptyset,\ J_U(r) = J_G(r)\}|}_{\text{True Positives (TP)}}}{\underbrace{|\{r \in R \mid J_U(r) \neq \emptyset\}|}_{\text{TP + FP (all predicted joins)}}}$$

- **Numerator (TP)**: Records where a join was predicted and it matched the ground truth.
- **Denominator (TP + FP)**: All records where a join was predicted (correct or not).
- Only records with a prediction (i.e., $J_U(r) \neq \emptyset$) are evaluated in this precision formula.

Theoretical foundation: evaluating join quality; Recall

Recall measures how many of the correct (ground truth) joins were successfully predicted:

$$\text{recall}(U) = \underbrace{|\{r \in R \mid J_U(r) \neq \emptyset,\ J_U(r) = J_G(r)\}|}_{\text{True Positives (TP)}}$$

- This is the **absolute count of True Positives**, i.e., records for which:
  - A join was predicted ($J_U(r) \neq \emptyset$), and
  - It matches the ground truth ($J_U(r) = J_G(r)$)

False Negatives (FN) — cases where a correct join was missed — are defined as:

$$\text{FN} = |\{r \in R \mid J_G(r) \neq \emptyset,\ J_U(r) = \emptyset\}|$$

Note: The denominator $TP + FN$ is constant across all U for a fixed dataset, so it is omitted in comparisons.

Theoretical foundation: Estimating precision without labels

Traditional precision metrics require a labeled ground truth to evaluate the quality of predicted joins.

Auto-FuzzyJoin introduces an unsupervised method to estimate join precision, without labeled data.

- Uses a local geometric heuristic: the number of L records within a $2d$-ball around a matched reference point l
- Fewer neighbors imply higher confidence in the match (i.e., higher estimated precision)
- This estimation is:
  - **Data-driven**: only needs L and R
  - **Model-independent**: works with any join function f
  - **Efficient**: avoids costly labeling efforts

This idea enables precision-aware optimization without needing ground truth labels.

Theoretical foundation: estimating Precision/Recall for a single join configuration

Given:

- A **single join configuration** $C = \langle f, \theta \rangle$
- Two tables:
  - L: reference table
  - R: query table

Assumption: Complete Reference Table L

- L is assumed to contain **all possible true matches** for records in R, that is for each $r \in R$, there exists a correct match $l \in L$.
- This simplifies analysis by reducing the chance of missing true positives due to an incomplete reference.

Geometric View of the Distance Function f

- Join function f embeds records into a **metric space**.
- Records are **conceptually** modelled as points on a **unit grid**.
- Each $l \in L$ is surrounded by **close variants** (differing by a token, character, etc.).
- The distance between each l and the surrounding r's is exploited by θ to compute join pairs.

Analogy: Stars and Planets

- Reference records $l \in L$ are like **stars** on a grid.
- Query records $r \in R$ are like **planets** that orbit these stars.
- Identifying the best join $J_C(r)$ is like determining **which star a planet orbits**.

Theoretical foundation: safe joins and the geometry of fuzzy matching

Safe Joins with a Complete L

- Define the **grid width** w: typical distance between a record l and its closest neighbors in L.
- A join is considered **safe** if the distance $d = f(l, r)$ satisfies:

$$d < \frac{w}{2}$$

- This guarantees that r lies **closer to its true match** l than to any other reference point.

Why This Matters:

- Ensures high **precision** — avoiding false positives caused by ambiguous joins.
- Avoids joining r to an incorrect l' that lies at a similar distance.

Analogy: Stars and Planets

- A planet that lies **equidistant** between two stars (at $\frac{w}{2}$ each) **cannot be confidently claimed by either**.
- In fuzzy joining, such cases are inherently **ambiguous** and risky to resolve.

Theoretical foundation: estimating join precision (local heuristic)

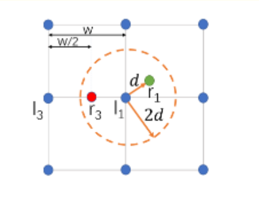
Given a query record $r \in R$ and its closest match $l \in L$, with distance $d = f(l, r)$, we can estimate how **precise** this join is — i.e., how likely it is that (l, r) is a **correct match**.

- The **more candidate records** in L that are close to r, the **less confident** we are about any one being the true match.
- So we count how many other records $l' \in L$ fall within the **2d-ball** centered at l:

$$\text{precision}(l, r) = \underbrace{\frac{1}{|\{l' \in L \mid f(l, l') \leq 2f(l, r)\}|}}_{\text{TP + FP (local competitors)}}$$

- A small $2d$-ball $\rightarrow$ high precision (few competitors).
- A large $2d$-ball $\rightarrow$ low precision (many competitors).

This provides a **data-driven estimate** of join quality **without needing ground truth**.



- To estimate the quality of joining r_1, we first find its nearest neighbor in L, which we'll call l_1.
- Compute the distance: $d = f(l_1, r_1)$.
- Draw a ball of radius $2d$ centered at l_1.
  - If no other L records fall in the ball $\rightarrow$ high confidence.
- In this case, the $2d$-ball contains only l_1:

$$\text{precision}(l_1, r_1) = \frac{1}{1} = 1$$

- **High confidence join.**

Theoretical foundation: When L is incomplete

Problem: When L is incomplete (i.e., some records are missing):

- Missing records in L result in **missing stars** in the grid.
- A record r may join to the wrong l, causing **false positives** and reducing **precision**.
- Example: If r_2 should match with l_2 (but l_2 is missing), it might instead match l_1 using $d = f(r_2, l_1)$.

Note:

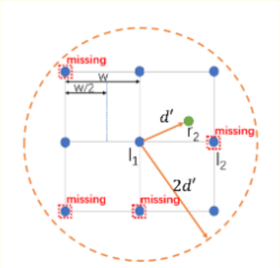
- Even if some records in L are missing, **safe decisions** can still be made.

Precision estimation:

- r_2 should match l_2 (missing), so l_1 becomes the fallback.
- The $2d'$-ball around l_1 contains **5 records**.
- Precision:

$$\text{precision}(l_1, r_2) = \frac{1}{5}$$

- $\Rightarrow$ **Low confidence join**



- r_2 should join with l_2, but l_2 is **missing**
- l_1 becomes the closest available record
- Compute distance $d' = f(l_1, r_2)$
- Draw a $2d'$-ball around l_1
  - If the ball includes many other L records $\rightarrow$ d' is too **lax**
  - Join becomes unreliable

Theoretical foundation: Estimating Precision and Recall for a configuration

A configuration $C = \langle f, \theta \rangle$ includes:

- A join function f
- A threshold θ

1. Local precision for a join

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

- $J_C(r) = l$: join match for $r \in R$
- Denominator = number of plausible alternatives

2. Expected true positives

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

3. Expected false positives

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

4. Overall Precision and Recall

$$\textbf{precision}(C) = \frac{TP(C)}{TP(C) + FP(C)} \qquad \textbf{recall}(C) = TP(C)$$

Note: Recall is estimated absolutely since ground truth is unavailable.

Understanding TP and FP contributions

True Positives (TP) and False Positives (FP) are calculated from the estimated precision of each join:

- If a configuration joins a record r with high estimated precision $\rightarrow$ contributes more to TP
- If a join has low estimated precision $\rightarrow$ contributes more to FP

Formula Review:

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

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

Implication:

- Adding a configuration to U increases recall (more joins)
- But can hurt precision if added joins are unreliable
- Greedy selection prefers configurations that give more TP per unit FP

Example: Precision and Recall estimation

Setup: Assume 3 records in R, joined to L using configuration $C = \langle f, \theta \rangle$.

Join Results:

- $J_C(r_1) = l_1$, $f(l_1, r_1) = 0.1$, 5 plausible matches $\Rightarrow$ precision$(r_1, C) = \frac{1}{5} = 0.20$
- $J_C(r_2) = l_2$, $f(l_2, r_2) = 0.05$, 2 plausible matches $\Rightarrow$ precision$(r_2, C) = \frac{1}{2} = 0.50$
- $J_C(r_3) = l_3$, $f(l_3, r_3) = 0.2$, 4 plausible matches $\Rightarrow$ precision$(r_3, C) = \frac{1}{4} = 0.25$

Estimated TP and FP:

$$TP(C) = 0.20 + 0.50 + 0.25 = 0.95$$
$$FP(C) = (1 - 0.20) + (1 - 0.50) + (1 - 0.25) = 2.05$$

Estimated Precision and Recall:

$$\text{precision}(C) = \frac{0.95}{0.95 + 2.05} = \frac{0.95}{3.00} \approx \mathbf{0.317}$$

$$\text{recall}(C) = TP(C) = \mathbf{0.95}$$

Note: This example assumes no ground truth; hence recall is based on expected TP count.

Theoretical foundation: Precision and Recall for a set of configurations

Let $U = \{C_1, C_2, \ldots, C_K\}$ be a set of configurations.

Case 1: No Conflicts in U

- Each record $r \in R$ is matched by at most one configuration:

$$\forall r \in R, \quad |J_U(r)| \leq 1$$

- Then:

$$TP(U) = \sum_{C \in U} TP(C), \quad FP(U) = \sum_{C \in U} FP(C)$$

Case 2: Conflicting Assignments in U

- Multiple configurations suggest different joins for the same r
- Resolve conflicts by:
  1. Compare precision scores: precision(r, C_i) vs. precision(r, C_j)
  2. Choose the match with higher precision
  3. Assign that join to $J_U(r)$
  4. Recompute $TP(U)$ and $FP(U)$

Final Estimates:

$$\text{precision}(U) = \frac{TP(U)}{TP(U) + FP(U)} \qquad \text{recall}(U) = TP(U)$$

Example: resolving conflicting joins from multiple configurations

Context: Two configurations propose different joins for the same record $r \in R$ using different string similarity methods.

Configurations:

- $C_1 = \langle f_1, \theta_1 \rangle$, where f_1 uses Jaccard distance over space-tokenized lowercase strings with equal weights.
- $C_2 = \langle f_2, \theta_2 \rangle$, where f_2 uses Cosine similarity over character trigrams with TF-IDF weighting.

Join Proposals for r:

- C_1: $J_{C_1}(r) = l_1$ with $\text{precision}(r, C_1) = \frac{1}{4} = 0.25$
- C_2: $J_{C_2}(r) = l_2$ with $\text{precision}(r, C_2) = \frac{1}{2} = 0.50$

Conflict Resolution Strategy:

1. Compare estimated precision:

$$\text{precision}(r, C_1) = 0.25 < \text{precision}(r, C_2) = 0.50$$

2. Assign $J_U(r) = l_2$ (higher-confidence match from C_2)

Effect:

- $TP(U)$ and $FP(U)$ incorporate only the winning match.
- Competing matches are discarded.

From theory to implementation

Before diving into the implementation of AutoFJ for the single-column case, let us briefly revisit the key theoretical ideas that drive the algorithm.

- A fuzzy join configuration is defined by a join function and a threshold $C = \langle f, \theta \rangle$, where f captures how similarity is computed, and θ determines when two strings are similar enough to join.
- Multiple such configurations are explored in order to maximize recall while satisfying a user-defined precision constraint τ.
- Precision is estimated without labeled data using a geometric heuristic based on how isolated a match is in the reference table L.
- The algorithm selects an optimal set of configurations $\mathcal{U} \subseteq \mathcal{F} \times \Theta$ through a greedy strategy that maximizes true positives while minimizing false positives.

Auto-FuzzyJoin Algorithm: single column case

Recall-Maximizing Fuzzy Join (RM-FJ) is **NP-hard**. Use a **greedy approximation algorithm** called `AutoFJ`.

Objective:

- Maximize recall $TP(U)$ subject to maintaining precision$(U) \geq \tau$

Greedy Strategy:

- Select configurations that:
  - Increase true positives (recall)
  - Minimize false positives (preserve precision)
- Guided by the **Profit Metric**:

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

Blocking Heuristic:

- To reduce the number of comparisons, apply **3-gram blocking**.
- Each string is decomposed into overlapping sequences of 3 characters (3-grams).
- Only record pairs that share at least one common 3-gram are considered for joining.
- This blocks out obviously dissimilar pairs and speeds up computation.
- Applied to both L–L and L–R candidate pairs:

$$LL, LR \longleftarrow \text{generate candidate pairs using 3-gram overlap}$$

Algorithm 1 AutoFJ for single column

Require: Tables L and R, precision target τ, search space S

$LL, LR \leftarrow$ apply blocking with $L-L$ and $L-R$
$LR \leftarrow$ Learn negative-rules from LL and apply rules on LR (Alg. 2)
Compute distance with different join functions $f \in S$
Pre-compute precision estimation for each configuration $C \in S$
$U \leftarrow \emptyset$
while $S \setminus U \neq \emptyset$ **do**
  $max\_profit \leftarrow 0$
  for all $C \in S \setminus U$ **do**
    if $profit(U \cup \{C\}) > max\_profit$ **then**
      $C^* \leftarrow C, max\_profit \leftarrow profit(U \cup \{C\})$
  if $precision(U \cup \{C^*\}) > \tau$ **then**
    $U \leftarrow U \cup \{C^*\}$
  else
    break
return U

Problem formulation and complexity

Goal: Identify a set of join configurations $U \subseteq \mathcal{S}$ such that:

- Maximizes recall: $TP(U)$
- Satisfies precision constraint: precision$(U) \geq \tau$

Formal problem definition: Recall-maximizing fuzzy join (RM-FJ)

Given reference table L, query table R, and configuration space $\mathcal{S}$, find a subset $U \subseteq \mathcal{S}$ to:

$$\max_{U \subseteq \mathcal{S}} TP(U) \quad \text{subject to} \quad \text{precision}(U) \geq \tau$$

Computational Complexity:

- The RM-FJ problem is shown to be **NP-hard**.
- Exact search over all subsets of $\mathcal{S}$ is computationally infeasible.
- Justifies use of **greedy approximation** (AutoFJ).

Blocking for efficient candidate generation

Motivation:

- Naively comparing every $r \in R$ with every $l \in L$ is computationally expensive.
- We use a **blocking** technique to generate a smaller candidate set for similarity evaluation.

Technique: 3-Gram Blocking

- Each string is decomposed into overlapping substrings of 3 characters (3-grams).
- Only consider (r, l) pairs that share at least one common 3-gram.
- Applied on both L–L (for negative rule learning) and L–R (for actual join candidates).

Impact:

- Reduces the number of unnecessary comparisons.
- Increases efficiency without significant recall loss.

1. 3-Gram blocking using TF-IDF

`LL, LR` $\leftarrow$ `apply 3-gram blocking on` L–L `and` L–R

This step implements the *blocking* stage described in the theoretical framework. The goal is to reduce the number of candidate pairs from $L \times R$ by quickly eliminating obviously dissimilar records. Blocking uses 3-gram tokenization and TF-IDF weighting to prioritize pairs that share meaningful substrings. Only pairs that pass this filter proceed to the more expensive distance computation phase.

Reference Table L:

l_1	`"john smith"`
l_2	`"jane smythe"`
l_3	`"alice johnson"`

Query Record r_1: `"jon smyth"`

Step 1: Preprocessing (P)

- Lowercasing (already lowercase)
- Add padding for 3-grams: e.g., `"john smith"` $\rightarrow$ `"##john#smith##"`

Step 2: Tokenization (T)

- r_1: `##j, #jo, jon, on# , n#s, #sm, smy, myt, yth, th#, h##`
- l_1, l_2: similar 3-gram sequences

Step 3: Token Weighting (W)

- Use TF-IDF to emphasize rare, meaningful trigrams (e.g., `smy`, `yth`)
- r_1–l_2: **High score(rare overlapping trigrams)**, r_1–l_1: Medium (more common overlap), r_1–l_3: Zero (no shared trigrams)

Blocking Result:

- Only compare r_1 with l_1, l_2 $\rightarrow$ prune l_3

The next step corresponds to the theoretical concept of *negative rule learning*, where patterns of dissimilarity within the reference table L are used to eliminate poor candidate matches before computing distances. These rules are designed to catch non-matching pairs that may appear similar due to superficial token overlap but are semantically distinct. Filtering based on these learned rules improves both computational efficiency and overall precision.

2. Optimization - filtering with negative rules

`LR ← Learn negative-rules from LL and apply rules on LR (Alg. 2)`

Assumption: Although 3-gram blocking may have pruned l_3, we assume here it was retained due to weak overlap, allowing us to illustrate negative-rule filtering.

Goal: Use **obvious non-matches** in L–L to learn rules that help **filter unlikely L–R pairs** before costly distance computations.

Step 1: Generate LL — Self-Join on L using 3-gram blocking

Pair	Shared 3-grams	Interpretation
l_1 vs l_2	`sm, smy, th`	Possibly similar
l_1 vs l_3	`jo, on`	Clearly different
l_2 vs l_3	Weak overlap	Probably different

Learn Negative Rule:
"If 3-gram overlap ≤ 2, treat as a non-match."

Step 2: Apply Rule on LR Candidate Pairs

Pair	Overlap	Apply Rule?	Keep?
r_1, l_1	~ 4	No	Yes
r_1, l_2	~ 5	No	Yes
r_1, l_3	~ 1	Yes	No

Effect: *Filter out clearly irrelevant pairs early — no need to compute Jaccard or Edit Distance!*

The next step directly implements the fuzzy join function $f(l, r)$ as defined in the theoretical framework. Each join function is composed of a preprocessing step P, a tokenization strategy T, a token weighting scheme W, and a distance metric D. The full join function is expressed as:

$$f(l, r) = D(W(T(P(l))), W(T(P(r))))$$

Multiple such functions are evaluated in parallel, each capturing different notions of similarity. The results will later be used to select the most effective join configurations under precision constraints.

3. Compute distances - apply join functions

```
Compute distance with different join functions f ∈ S
Pre-compute precision estimation for each configuration C ∈ S
```

Once candidate pairs are identified (via blocking and optional negative rules), we compute the actual similarity using multiple join functions $f \in \mathcal{S}$.

Each join function is defined by:

- Preprocessing (e.g., lowercasing, punctuation removal)
- Tokenization (e.g., char 3-grams, word tokens)
- Token weights (e.g., TF-IDF)
- Distance function (e.g., Jaccard, Cosine, Edit)

Example Candidate Pairs (after blocking):

r (query)	l (reference)
"jon smyth"	"john smith"
"jon smyth"	"jane smythe"

Join Functions in $\mathcal{S}$:

Function f	**Tokenizer**	**Distance**	**Description**
f_1	char 3-grams	Jaccard	Overlap in token sets
f_2	char 3-grams	Cosine (TF-IDF)	Weighted similarity
f_3	raw string	Levenshtein	Edit distance

Computed Scores:

Pair	f_1	f_2	f_3
jon vs john	0.4	0.5	2
jon vs jane	0.6	0.7	3

Note: Distances may follow different scales — lower often means more similar.

The next step implements the core optimization loop of AutoFJ, which is guided by the objective of maximizing recall while ensuring the estimated precision remains above a threshold τ. The goal is to identify a subset of join configurations $\mathcal{U} \subseteq \mathcal{F} \times \Theta$ that collectively yield the most high-quality joins. A greedy approximation is used due to the NP-hardness of the problem, incrementally adding the most profitable configuration C that increases true positives without violating the precision constraint.

4. Start of greedy algorithm

`Initialize:` $U \leftarrow \emptyset$

U will hold the **selected join configurations**:

$$C = \langle f, \theta \rangle$$

Each configuration includes:

- A join function $f \in \mathcal{F}$ (defined by P, T, W, D)
- A distance threshold θ (max allowed distance for a match)

Goal:

- Select a subset $U \subseteq \mathcal{S}$ from all candidate configurations
- Maximize recall: $TP(U)$
- Maintain precision: precision$(U) \geq \tau$

Example: Precomputed configuration set S

Config C	**Description**
$C_1 = \langle f_1, 0.37 \rangle$	Jaccard distance with $\theta = 0.37$
$C_2 = \langle f_2, 0.42 \rangle$	Cosine distance with $\theta = 0.42$
$C_3 = \langle f_3, 2 \rangle$	Edit distance with $\theta = 2$

These θ values were selected based on prior precision–recall evaluation for each f.

5. Main greedy loop

`Main Loop: while` $S \setminus U \neq \emptyset$ `do`

We continue as long as there are still unused configurations to consider.

Notation:

- S: full set of candidate configurations, each $C = \langle f, \theta \rangle$
- U: set of selected configurations
- $S \setminus U$: unused configurations

At each iteration:

1. Evaluate each $C \in S \setminus U$
2. Compute profit: how many true positives vs. false positives it contributes
3. Select the best configuration C^*
4. If precision$(U \cup \{C^*\}) \geq \tau$:

$$U \leftarrow U \cup \{C^*\}$$

Example state:

- $S = \{\langle f_1, 0.37\rangle, \langle f_2, 0.42\rangle, \langle f_3, 2\rangle\}$
- $U = \emptyset$

Loop continues while there are remaining candidates and precision can be preserved.

6. Find most promising configuration (profit heuristic)

```
max_profit ← 0
for all  C ∈ S \ U  do
    if  profit(U ∪ {C}) > max_profit  then
      C* ← C, max_profit ← profit(U ∪ {C})
```

Profit Formula:

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

Example:

Config C	TP	FP	Profit = TP / FP
C_1	4	2	2.0
C_2	5	5	1.0
C_3	3	1	**3.0**

After evaluation: $C^* = C_3$, $\texttt{max\_profit} = 3.0$

Heuristic: choose the configuration that gives the most recall "bang" per unit of precision "risk."

7. Precision constraint check & termination

`Check: if` precision$(U \cup \{C^*\}) > \tau$ `then` $U \leftarrow Ucup\{C^*\}$

After selecting the best candidate C^* (based on profit), we must verify that adding it to U preserves minimum required precision τ.

- If precision passes: add C^* to U
- Else: break — no remaining configs will satisfy the constraint

Example 1 (Pass):

Config	TP	FP	Profit	Precision	τ
C_3	3	1	3.0	**0.75**	0.7

$\Rightarrow$ Precision $> \tau \rightarrow$ Accept $\rightarrow U \leftarrow \{C_3\}$

Example 2 (Fail & Break):

Config	TP	FP	Profit	Precision	τ
C_3	3	1	3.0	**0.75**	0.8

$\Rightarrow$ Precision $< \tau \rightarrow$ Reject $\rightarrow$ Stop Loop

Greedy termination: If best config can't meet τ, no others will.

8. Return final join plan

```
Return: U
```

The greedy loop terminates when:

- $S \setminus U = \emptyset$ (all configs evaluated), or
- The best candidate fails the precision constraint

The algorithm returns U: a set of selected configurations:

- Each $C = \langle f, \theta \rangle$
- Maximizes recall while keeping precision$(U) \geq \tau$

Each configuration in U defines:

- A join function f (e.g., Jaccard, Cosine, Edit Distance)
- A threshold θ used to accept matches

Example Output:

- $U = \{\langle f_2 = \text{Cosine}, \theta = 0.5\rangle,\ \langle f_3 = \text{Edit}, \theta = 2\rangle\}$

These are used to perform the final fuzzy similarity join.

Example: Selecting the best match

1. Input Setup:

- Query record: $r_1 =$ `"jon smyth"`
- Reference table: $L = \{$`"john smith"`, `"jane smythe"`, `"alice johnson"`$\}$
- After blocking: candidates for r_1 are l_1 and l_2

2. Distance Results:

Join Function f	θ	$f(r_1, l_1)$	$f(r_1, l_2)$	Matches?
Jaccard (3-grams)	0.4	0.5	0.3	l_2 only
Cosine (TF-IDF)	0.5	0.6	0.4	l_2 only
Edit Distance	2.0	2	3	l_1 only

3. Final Configuration Set U:

- $U = \{\langle f_2 = \text{Cosine},\ \theta = 0.5\rangle,\ \langle f_3 = \text{Edit},\ \theta = 2\rangle\}$
- Under Cosine: $r_1 \mapsto l_2$
- Under Edit Distance: $r_1 \mapsto l_1$

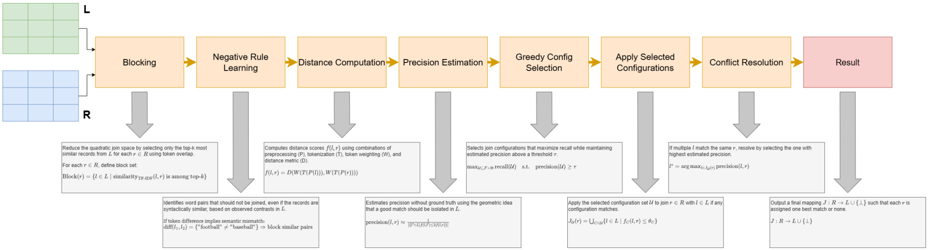
4. Conflict Resolution: Local Precision

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

Config C	Match	$f(l, r)$	$2d$-ball size	Precision
C_2 (Cosine)	l_2	0.4	5	$1/5 = 0.2$
C_3 (Edit)	l_1	2	2	$\mathbf{1/2 = 0.5}$

Result: *"jon smyth" is matched to "john smith" (Edit Distance), since it has higher estimated precision ($0.5 > 0.2$).*

AutoFJ pipeline



AutoFJ Architecture

AutoFJ Codebase Structure

```
autofj/
|-- 50-single-column-datasets.md          # Documentation describing benchmark datasets
|-- autofj.py                             # Main driver script for AutoFJ
|-- datasets.py                           # Loads and preprocesses datasets
|-- negative_rule.py                      # Learns rules to prevent false matches
|-- utils.py                              # General-purpose utility functions
|-- benchmark/                            # Contains all test datasets and benchmarks
|-- blocker/                              # Blocking component
|   |-- autofj_blocker.py                 # AutoFJ-specific record blocking
|   |-- blocker.py                        # General blocking logic
|-- optimizer/                            # Greedy optimization logic
|   |-- autofj_multi_column_greedy_algorithm.py   # Multi-column join optimizer
|   |-- autofj_single_column_greedy_algorithm.py  # Single-column join optimizer
 |-- join_function_space/                  # Parameter space for join functions
|   |-- autofj_join_function_space.py     # Constructs and manages the join function space
|   |-- options.py                        # Parameter definitions
|   |-- join_function/                    # Join function components
|   |   |-- autofj_join_function.py       # Encapsulates join logic
|   |   |-- distance_function.py          # Implements distance metrics
|   |   |-- join_function.py              # Computes scores for matching
|   |   |-- preprocessor.py               # Text cleaning and normalization
|   |   |-- tokenizer.py                  # Tokenization strategies
|   |   |-- token_weight.py               # Token weighting methods
```

Step 1: Start - Input Tables L and R

Function

Loads the reference table L and the noisy table R, possibly applying minimal preprocessing.

Implementation

`autofj/datasets.py`	Uses standard file loading and DataFrame operations to ingest raw datasets. It may include optional utilities to split or format tables.

Step 2: Blocking

Function

Reduces the number of comparisons by pruning unlikely matches based on token similarity.

Implementation

`autofj/blocker/autofj_blocker.py`	Main entry point for blocking.
`autofj/blocker/blocker.py`	Constructs token sets using 3-gram decomposition, computes TF-IDF scores for each token, and retrieves candidate pairs via cosine similarity. Function `block_L_L_and_L_R()` returns top-k most similar candidates from L for each $r \in R$ based on TF-IDF weighted token overlap.

Step 3: Negative Rule Learning

Function
Learns simple rules to eliminate false matches based on exclusive tokens (e.g., 2007 $\neq$ 2008) learned from intra-L variation.

Implementation

`autofj/negative_rule.py`

`learn_negative_rules(...)`	identifies discriminative tokens from record pairs in L differing by only one word.
`apply_negative_rules(...)`	removes candidate (l, r) pairs from $L \times R$ if they violate these rules.

Step 4: Distance Computation

Function
Computes similarity scores using various join function configurations $\langle P, T, W, D \rangle$.

Implementation
`autofj/join_function_space/autofj_join_function_space.py` – Iterates over all valid combinations of preprocessing (lowercase, strip punctuation), tokenization (space or 3-gram), weighting (equal or TF-IDF), and distances (Jaccard, Cosine, Edit).

Each component is modularized:

`preprocessor.py`	e.g., lowercase, remove punctuation
`tokenizer.py`	whitespace or n-gram based splitting
`token_weight.py`	raw or IDF weighting
`distance_function.py`	Jaccard, Edit, Cosine

Scores are stored in a large matrix for all candidate (l, r) pairs under all functions.

Step 5: Precision Estimation

Function
Estimates the reliability of each predicted join (l, r) using a geometric heuristic based on L.

Implementation
`autofj/autofj.py` – `estimate_precision(...)` checks how "isolated" l is in the embedding space: the fewer $l' \in L$ within a 2-ball of l, the higher the confidence.
This is a key innovation: it enables precision estimation without ground truth by assuming L has no duplicates.

Step 6: Greedy Configuration Selection

Function

Selects join configurations that maximize recall while meeting a minimum precision threshold τ.

Implementation

`autofj/optimizer/autofj_single_column_greedy_algorithm.py` – Implements a greedy loop where each candidate configuration $C = \langle f, \theta \rangle$ is scored using a profit function: $TP(C)/FP(C)$.
Configuration C^* is added to $\mathcal{U}$ only if the precision of $\mathcal{U} \cup \{C^*\}$ remains $\geq \tau$.

Step 7: Apply Selected Configurations

Function
Applies all configurations in $\mathcal{U}$ to generate potential matches for each $r \in R$.

Implementation

`autofj/autofj.py`	Applies each $\langle f, \theta \rangle \in \mathcal{U}$ to the blocked pairs. If multiple l satisfy $f(l, r) \leq \theta$, they are retained for conflict resolution.

Step 8: Conflict Resolution

Function
If multiple $l \in L$ match a single $r \in R$, pick the one with highest estimated precision.

Implementation

`autofj/autofj.py`	Function `get_final_join_result(...)` evaluates local precision of each match using the 2-ball heuristic. The l with the lowest neighborhood density is chosen.

Step 9: Output - Final Join Result

Function

Returns the final mapping $J : R \to L \cup \{\perp\}$ where each r is assigned the best matching l or none at all.

Implementation

`autofj.py`	Wraps the full join pipeline from blocking to conflict resolution and outputs the join set. Can be extended to include score thresholds, logs, and exporting joins.

Testing

Possible improvements