Fast and Precise Convolutional Jaro and Jaro-Winkler Similarity

Ondřej Rozinek
University of Pardubice
Pardubice, Czech Republic
ondrej.rozinek@gmail.com

Jan Mareš
University of Pardubice
Pardubice, Czech Republic
jan.mares@vscht.cz

***Abstract*—In the domain of character-based approximate string matching, edit distances such as Levenshtein have remained predominant despite their quadratic time complexity. This reality has prompted the adoption of more efficient metrics like Jaro and Jaro-Winkler. However, these methods often overlook the significance of character order within the matching window, which can adversely affect accuracy.**

For the first time, we introduce a novel class of character-based approximate string matching algorithms that leverage a convolutional kernel, surpassing the performance of existing state-of-the-art unsupervised character-based approximate string matching algorithms. This paper presents Convolutional Jaro (ConvJ) and Convolutional Jaro-Winkler (ConvJW), innovative similarity metrics designed to overcome these shortcomings. ConvJ and ConvJW utilize a convolutional approach with Gaussian weighting to effectively capture the positional proximity of matching characters, resulting in a more precise similarity evaluation. This method not only achieves computational efficiency comparable to that of Jaro and Jaro-Winkler but also surpasses the state-of-the-art in terms of F1-score, demonstrating faster execution times compared to the conventional Jaro and Jaro-Winkler implementations across various datasets.

Our extensive experimental analysis highlights the exceptional performance of ConvJ and ConvJW across a range of datasets. Remarkably, ConvJ exhibits a 7x faster execution time than the fast Jaro implementation and exceeds the state-of-the-art F1-score by a significant margin of 10% more than Jaro. By setting a new benchmark in unsupervised character-based approximate string matching, our research shows the new way for future exploration and development in this field. The ConvJ and ConvJW algorithms, characterized by their quasilinear time complexity and improved accuracy, provide a solid foundation for the advancement of string matching techniques. These developments hold promise for a broad spectrum of applications in data mining, bioinformatics, and related areas.

I. Introduction

Character-based similarity metrics represent critical tools in various domains, including data mining, bioinformatics, and natural language processing. Widely used methods such as Jaro [1] and Jaro-Winkler [2]–[5] offer efficient similarity evaluation, but they often neglect the significance of character order within the matching window. This disregard can potentially compromise accuracy, especially when dealing with strings where specific sequence holds considerable importance.

In this paper, we propose two novel metrics, Convolutional Jaro (ConvJ) and Convolutional Jaro-Winkler (ConvJW), that effectively address this limitation. Both metrics leverage a convolutional approach with a Gaussian weighting function to effectively capture the positional proximity of matching characters, leading to significantly enhanced accuracy compared to existing methods in the category of unsupervised character based approximate string matching.

The key contributions of this work comprise:

Improved accuracy: ConvJ and ConvJW achieve superior accuracy as measured by F1-score, surpassing the state-of-the-art in unsupervised character based approximate string matching.

Computational efficiency: Both metrics maintain computational efficiency comparable to Jaro and Jaro-Winkler, making them suitable for practical applications.

Faster execution: ConvJ exhibits even faster execution times compared to the standard Jaro implementation.

These characteristics render ConvJ and ConvJW ideal for tasks requiring high-performance string similarity calculations. They set the stage for continued investigation of convolutional similarity measures across diverse fields.

Despite different modern techniques the leading technological companies using Jaro and Jaro-Winkler in their products and it belongs to the most impactful and most used similarity function for task in deduplication and matching. We list some large products where are such techniques implemented:

A. Commercial products

- **Oracle Data Quality**: A component of Oracle's broader data management suite that offers data cleansing, profiling, and matching capabilities. It likely uses algorithms similar to Jaro-Winkler for its matching and deduplication processes to ensure high data quality across enterprise systems [6].
- **Talend Open Studio**: This open-source data integration platform includes the Jaro-Winkler similarity function in its Data Profiling and Data Quality, which can be used for data cleansing and matching. You can find the documentation for the Data Profiling and Data Quality solution here: [7].
- **IBM InfoSphere** This data integration platform includes the Jaro-Winkler similarity function in its Transformer stage, which can be used for data cleansing and matching. You can find the documentation for the Transformer stage here: [8].

- **Informatica Data Quality** This data integration platform includes the Jaro and Jaro-Winkler similarity functions in its Data Quality transformation, which can be used for data cleansing and matching. You can find the documentation for the Data Quality transformation here: [9].

B. Open-source libraries

- **Apache Lucene** This open-source search engine library includes the Jaro-Winkler similarity function in its FuzzyQuery class, which can be used for fuzzy search queries. You can find the documentation for the FuzzyQuery class here: [10].

This list is far from exhaustive; there are numerous other commercial products and open-source libraries that utilize the Jaro or Jaro-Winkler algorithms.

II. Related Work

String similarity metrics play a crucial role in various domains, including data mining, bioinformatics, and natural language processing. Several established metrics exist, each with its own strengths and weaknesses. Here, we discuss relevant approaches and highlight how our proposed Convolutional Jaro (ConvJ) and Convolutional Jaro-Winkler (ConvJW) builds upon them. See Fig. 1.

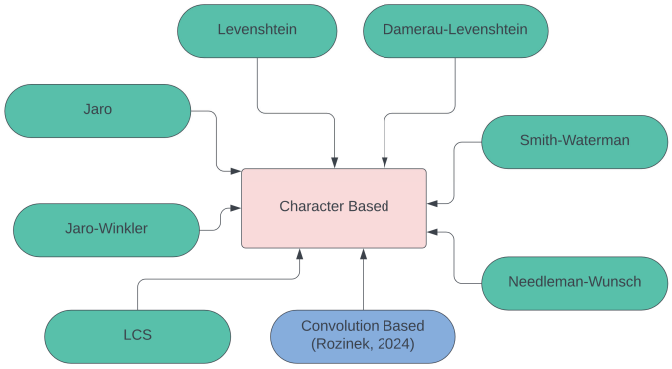

Fig. 1. This figure delineates the taxonomy of character-based approximate string matching algorithms, categorizing them by their methodological approach and operational characteristics. A newly introduced class, termed 'Convolution-Based', as discussed in this article, highlights the main contribution.

A. Established Edit Distance Measures

a) Levenshtein distance: This classic metric measures the minimum number of edit operations (insertions, deletions, substitutions) Required to transform one string into another [11]. While efficient and widely used, its computational complexity grows quadratically with string length, limiting its scalability for large datasets [12].

b) Damerau-Levenshtein distance: An extension of Levenshtein distance, it also considers transpositions as edit operations. This provides more flexibility but maintains quadratic complexity [13].

c) Needleman-Wunsch algorithm: This global alignment algorithm finds the optimal alignment between two strings, offering high accuracy but being computationally expensive for long strings [14].

d) Smith-Waterman algorithm: An improvement over the Needleman-Wunsch algorithm for local sequence alignment, the Smith-Waterman algorithm identifies the most similar segment between two sequences. It is particularly useful in bioinformatics for finding similar regions within genes or proteins. The algorithm's complexity remains quadratic, but it often performs better in practice by focusing on the most relevant segments of the sequences [15]. Recent advancements have aimed at optimizing its performance through parallel computing techniques and specialized hardware, such as GPUs and FPGAs, to handle larger datasets more efficiently [16], [17].

e) Longest Common Subsequence (LCS): The LCS problem involves finding the longest subsequence present in both sequences without disturbing the order of characters. This measure is crucial in diff utilities, version control systems, and understanding the similarity between texts. Unlike the edit distances, LCS focuses on similarity rather than the number of edits. The complexity of the basic LCS algorithm is also quadratic, but various optimizations and approximations have been proposed to improve its scalability and efficiency [18], [19].

B. Character-based Similarity Heuristics

a) Jaro similarity: This efficient metric focuses on matching characters within a limited window around their original positions. However, it does not consider the relative order of characters within the window, potentially impacting accuracy [1], [20].

b) Jaro-Winkler similarity: This variant of Jaro includes a prefix bonus to prioritize matches at the beginning of strings, improving performance for specific cases [2]. However, it inherits the limitations of Jaro regarding positional information [21].

C. Convolutional Approaches

Character-level convolutional neural networks (CNNs): These have been explored for string similarity, achieving high accuracy but often requiring large training datasets and computational resources [22]–[24].

Word embeddings with cosine similarity: Embedding methods learn vector representations of words, and cosine similarity measures the angle between them [25]. While effective for semantic similarity, they might not capture precise character-level differences relevant for our task [26].

Limitations of Existing Approaches: Edit distance measures are generally computationally expensive for large datasets [12]. Character-based metrics like Jaro and Jaro-Winkler neglect the importance of character order within the matching window. While CNNs offer high accuracy, they can be resource-intensive [22], [25]. Word embeddings might not be

optimal for capturing fine-grained character-level similarities [26].

Addressing the Limitations: ConvJ addresses these limitations by combining the efficient matching mechanism of Jaro with a convolutional approach. It incorporates Gaussian weighting to emphasize matches closer in position and leverages convolutions to capture the overall similarity landscape effectively. This results in a computationally efficient metric that considers both character matches and their relative order, leading to improved accuracy compared to existing methods.

III. Jaro Similarity Algorithm

The Jaro similarity metric, introduced by Matthew A. Jaro, serves as a measure for evaluating the similarity between two strings, denoted as $S1$ and $S2$. This metric is particularly useful in fields such as record linkage and spell checking, quantifying the degree of similarity on a scale from 0 to 1, where 0 indicates no similarity and 1 denotes an exact match.

Given strings $S1$ and $S2$, the Jaro similarity score, $s_J(S1, S2)$, is mathematically defined as follows:

$$s_J(S1,S2) = \begin{cases} 0 & \text{if } m = 0, \\ \frac{1}{3}\left(\frac{m}{|S1|} + \frac{m}{|S2|} + \frac{m-t}{m}\right) & \text{otherwise.} \end{cases} \quad (1)$$

Here, $|S1|$ and $|S2|$ represent the lengths of strings $S1$ and $S2$, respectively, m is the number of matching characters, and t is the half number of transpositions. A matching character is defined as one that is the same in both strings and is not farther than $\left\lfloor \frac{\max(|S1|,|S2|)}{2} \right\rfloor - 1$ positions away from its counterpart in the other string. A transposition is considered when two matching characters are in a different order in the two strings.

The computational complexity of the Jaro similarity Algorithm 1 is $\mathcal{O}(|S1||S2|)$, where $|S1|$ and $|S2|$ are the lengths of the two input strings. This efficiency makes it highly suitable for real-time data processing and large-scale data matching tasks. The Jaro similarity measure has found extensive applications in various domains requiring accurate string comparison, including database cleaning, information retrieval, and natural language processing. It also serves as the foundation for more sophisticated similarity metrics, such as the Jaro-Winkler distance, which introduces adjustments for common prefixes to increase precision in specific contexts.

Example 1. Jaro Similarity To demonstrate the Jaro similarity, consider the strings $S1$ = "MARTHA" and $S2$ = "MARHTA".

The matching characters are $m = 6$, as all characters in $S1$ match with those in $S2$, albeit in a slightly different order. The half number of transpositions t (where a transposition is a pair of matching characters in a different sequence between $S1$ and $S2$) is calculated as 1, since two characters ('R' and 'H') are out of order.

Thus, the Jaro similarity score s_J can be calculated as follows:

Algorithm 1 Jaro Similarity (implementation Rosetta [27])

Require: $S1, S2$ ▷ Two input strings
Ensure: $s_J(S1, S2)$ ▷ Normalized similarity score between 0 and 1

if $S1 =$ NULL or $S2 =$ NULL **then**
  return 0.0
end if
$|S1| \leftarrow$ length of $S1$
$|S2| \leftarrow$ length of $S2$
$w \leftarrow \max(|S1|, |S2|)/2 - 1$
Initialize $s1Matches[1 \ldots |S1|]$ to all **false**
Initialize $s2Matches[1 \ldots |S2|]$ to all **false**
$m \leftarrow 0$ ▷ Number of matches
$t \leftarrow 0$ ▷ Half the number of transpositions
for $i = 0$ to $|S1| - 1$ **do**
  for $j = \max(0, i - w)$ to $\min(i + w + 1, |S2|) - 1$ **do**
    if $s2Matches[j]$ is true or $S1[i] \neq S2[j]$ **then**
      continue
    end if
    $s1Matches[i] \leftarrow$ true
    $s2Matches[j] \leftarrow$ true
    $m \leftarrow m + 1$
    break
  end for
end for
if $m = 0$ **then**
  return 0.0
end if
$k \leftarrow 0$
for $i = 0$ to $|S1| - 1$ **do**
  if $s1Matches[i]$ **then**
    while $s2Matches[k]$ is false **do**
      $k \leftarrow k + 1$
    end while
    if $S1[i] \neq S2[k]$ **then**
      $t \leftarrow t + 1$
    end if
    $k \leftarrow k + 1$
  end if
end for
$s_J(S1, S2) \leftarrow \frac{1}{3}\left(\frac{m}{|S1|} + \frac{m}{|S2|} + \frac{m-t/2}{m}\right)$
return $s_J(S1, S2)$

$$s_J(S1,S2) = \frac{1}{3}\left(\frac{m}{|S1|} + \frac{m}{|S2|} + \frac{m-t}{m}\right) \quad (2)$$

$$= \frac{1}{3}\left(\frac{6}{6} + \frac{6}{6} + \frac{6-1}{6}\right) \quad (3)$$

$$= 0.944 \quad (4)$$

IV. Jaro-Winkler Similarity Algorithm

The Jaro-Winkler similarity metric enhances the Jaro similarity measure by giving more favorable scores to strings that match from the beginning for a set prefix length. This adjustment is particularly beneficial in applications where common prefixes are an indication of similarity, such as name matching in record linkage. The Algorithm 2 was developed by William E. Winkler [2] to improve the accuracy of the Jaro similarity metric in certain contexts.

The Jaro-Winkler similarity score, $s_{\text{JW}}(S1, S2)$, is calculated using the Jaro similarity score, $s_J(S1, S2)$, with an additional boost for common prefixes. The formula is given by:

$$s_{\text{JW}}(S1, S2) = s_J(S1, S2) + l \cdot p \cdot (1 - s_J(S1, S2)), \quad (5)$$

where l is the length of the common prefix up to a maximum of 4 characters, p is a scaling factor for how much the score is adjusted upwards for prefix similarity. A typical value for p is 0.1.

The common prefix length l is defined as the number of characters from the start of the strings that are identical, up to a maximum of 4 characters. This means that the maximum possible adjustment to the Jaro similarity score is $0.1 \times 4 \times (1 - s_J(S1, S2)) = 0.4 \times (1 - s_J(S1, S2))$, which can significantly influence the final similarity score for strings with common prefixes.

Algorithm 2 Jaro-Winkler Similarity (implementation)

Require: $S1, S2$ ▷ Two input strings
Ensure: $s_{\text{JW}}(S1, S2)$ ▷ Normalized similarity score between 0 and 1

```
s_J ← Jaro Similarity(S1, S2)
l ← 0
for i = 0 to min(min(|S1|, |S2|), 4) − 1 do
    if S1[i] = S2[i] then
        l ← l + 1
    else
        break
    end if
end for
p ← 0.1                                   ▷ Scaling factor
s_JW ← s_J + l · p · (1 − s_J)
return s_JW
```

The Jaro-Winkler similarity algorithm maintains the computational efficiency of the Jaro similarity, with an additional step to calculate the prefix length. This makes it equally suitable for real-time applications and large datasets where precise string matching is crucial. The introduction of the prefix scaling factor enhances the matching accuracy for strings with common beginnings, making it especially useful in the fields of data deduplication, record linkage, and information retrieval where such characteristics are common.

Example 2. Jaro-Winkler Similarity To illustrate the Jaro-Winkler similarity, we examine the strings $S1 =$ "DWAYNE" and $S2 =$ "DUANE".

First, calculate the Jaro similarity as above. Assuming $m = 4$ (matching characters excluding transpositions) and $t = 1$ (the 'W' and 'U' are transposed), the Jaro score is:

$$s_J(S1, S2) = \frac{1}{3}\left(\frac{4}{6} + \frac{4}{5} + \frac{4-1}{4}\right) \approx 0.822$$

The common prefix length l is 1 (for 'D'), and with $p = 0.1$ (the standard scaling factor), the Jaro-Winkler score s_{JW} becomes:

$$\begin{aligned} s_{\text{JW}}(S1, S2) &= s_J(S1, S2) + l \cdot p \cdot (1 - s_J(S1, S2)) && (6)\\ &= 0.822 + 1 \cdot 0.1 \cdot (1 - 0.822) && (7)\\ &= 0.822 + 0.018 && (8)\\ &= 0.840 && (9) \end{aligned}$$

This example illustrates how the Jaro-Winkler similarity provides a slight increase over the Jaro similarity for strings with a common prefix, emphasizing the importance of initial characters in certain contexts.

V. Convolutional Jaro (ConvJ) and Convolutional Jaro-Winkler (ConvJW)

A. Algorithm Overview

The ConvJ and ConvJW are innovative enhancements of the traditional Jaro and Jaro-Winkler similarity algorithms, incorporating a convolutional approach with Gaussian weighting to assess the similarity between two strings. This method extends the original Jaro algorithm by applying a Gaussian-weighted convolution operation, aiming to capture the positional proximity of matching characters with greater detailed similarity evaluation. See Algorithm 3 and Algorithm 4.

B. Gaussian Weighting

For any two positions i and j within strings S_1 and S_2, respectively, the Gaussian weight is computed as:

$$G(i, j) = \exp\left(-\frac{|i - j|^2}{2\sigma^2}\right) = \exp\left(-\frac{d^2}{2\sigma^2}\right), \quad (10)$$

where σ represents the standard deviation of the Gaussian kernel. This weighting function decreases the influence of character matches as their positional distance increases, with σ controlling the rate of this decrease.

C. Convolution Operation

The convolution operation aims to identify the optimal character match between strings S_1 and S_2 within a predefined sliding window, utilizing Gaussian weighting to balance proximity and character accuracy. This procedure is encapsulated by the calculation of M_w, representing the accumulated sum of optimal match evaluations across all character positions in S_1:

$$M_w = \sum_{i=0}^{|S_1|-1} \max_{j \in J(i)} \left\{ G(i,j) \cdot \delta_{S_1[i],S_2[j]} \right\}, \tag{11}$$

where $G(i,j)$ denotes the Gaussian weight function, emphasizing the impact of positional differences between characters at indices i in S_1 and j in S_2. The Kronecker delta function, $\delta_{S_1[i],S_2[j]}$, indicating an exact match between characters at positions i and j:

$$\delta_{S_1[i],S_2[j]} = \begin{cases} 1 & \text{if } S_1[i] = S_2[j], \\ 0 & \text{otherwise.} \end{cases} \tag{12}$$

The index set $J(i)$ defines the matching range for a character at position i, determined by:

$$J(i) = \{ j : \max(0, i-w) \leq j \leq \min(|S_2|-1, i+w) \}, \tag{13}$$

with w representing the half-window size, calculated based on the desired coverage percentage and the standard deviation (σ) of the Gaussian distribution. The calculation of w is informed by the z-score (Z), which is derived from the desired coverage percentage in a normal distribution. For a coverage of 99.9%, the z-score is determined as follows

$$Z = \Phi^{-1}\left(\frac{99.9\% + 1}{2} \right), \tag{14}$$

where Φ^{-1} denotes the inverse of the cumulative distribution function (CDF) for a standard normal distribution. Given $Z \approx 3.29$ for 99.9% coverage, w can be calculated by:

$$w = \lceil 3.29 \cdot \sigma \rceil, \tag{15}$$

where $\lceil \cdot \rceil$ denotes the ceiling function, ensuring that w is sufficiently large to include at least 99.9% of the normal distribution's weight, thus maximizing the likelihood of capturing the most relevant character matches within the window. This selection of w ensures that the convolution operation robustly accounts for both the precision of character matches and their positional proximity.

D. Misalignment Calculation

The ConvJ and ConvJW metrics enhance string similarity evaluation by incorporating a detailed assessment of character misalignment between strings S_1 and S_2. This approach extends beyond simple transposition counts to offer a granular examination of character positional deviations and their impact on similarity perception.

Misalignment is assessed using an adjusted inverse Kronecker delta function, $\bar{\delta}_{ij}$, to penalize positional discrepancies more effectively:

$$\bar{\delta}_{ij} = 1 - \delta_{ij}, \tag{16}$$

where δ_{ij} is defined as:

$$\delta_{ij} = \begin{cases} 1 & \text{if } i = j, \\ 0 & \text{otherwise,} \end{cases} \tag{17}$$

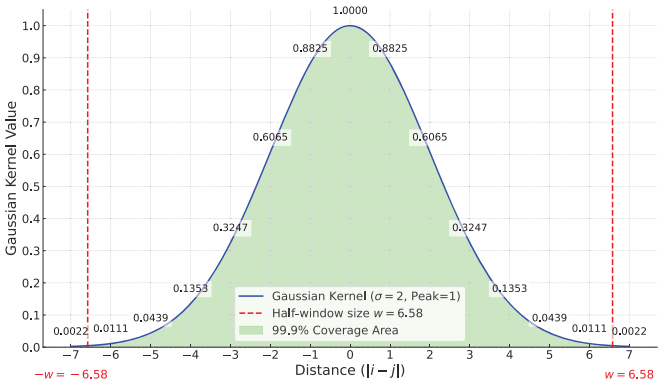

Fig. 2. The graph illustrates the Gaussian kernel's influence on convolution-based string similarity, with a standard deviation ($\sigma = 2$), highlighting the 99.9% coverage area critical for the ConvJ and ConvJW algorithms. The graph, marked with $-w$ and w to denote the half-window size, emphasizes the balance between character match precision and proximity.

indicating exact alignment of characters at index i in S_1 with index j in S_2, and 0 for misalignments. The complement, $\bar{\delta}_{ij}$, quantifies the degree of misalignment, which is then weighted by the Gaussian function, $G(i,j)$, to account for the significance of positional differences:

$$A_w = \sum_{(i,j) \in M} \bar{\delta}_{ij} \cdot G(i,j), \tag{18}$$

Here, A_w denotes the cumulative weighted sum of misalignments across all character position pairs within the set M. This measure not only identifies character matches but also captures the complex spatial dynamics of string similarity, enhancing the precision of the similarity score. The set M is defined as the collection of all character position pairs (i,j) where character i from string S_1 is evaluated against character j from string S_2. Formally, M can be represented as:

$$M = \{ (i,j) : i \in S_1, j \in S_2, \text{ and } |i-j| \leq w \}. \tag{19}$$

The parameter σ in the Gaussian weighting function adjusts the sensitivity to positional differences. A larger σ broadens the Gaussian distribution, accommodating misalignments over larger distances and thus penalizing distant mismatches less severely. In contrast, a smaller σ emphasizes immediate proximity, focusing the assessment on closely aligned character pairs. This flexibility allows for the algorithm to be tailored to different application needs, balancing precision with positional variance tolerance.

This approach ensures that the ConvJ and ConvJW metrics not only quantify character matches but also account for both character presence and their orderly sequence.

E. Similarity Score

The ConvJ similarity score is computed

$$s_{ConvJ} = \frac{1}{3}\left(\frac{M_w}{|S_1|} + \frac{M_w}{|S_2|} + \frac{M_w - A_w}{M_w} \right), \tag{20}$$

where M_w represents the sum of Gaussian-weighted matches for characters in S_1 against S_2, and A_w accounts for the weighted sum of misalignments between the two strings.

Algorithm 3 Convolutional Jaro Similarity with Gaussian Precomputation (ConvJ)

Require: S_1, S_2 ▷ Two input strings
Require: σ ▷ Standard deviation for Gaussian weighting
Ensure: s_{ConvJ} ▷ Normalized similarity score [0,1]

$w \leftarrow \lceil 3.29 \cdot \sigma \rceil$
$G \leftarrow$ PrecomputeGaussian(w, σ)
function PRECOMPUTEGAUSSIAN(w, σ)
  Initialize a 1D array $G[0 \dots w]$
  for $d = 0$ to w **do**
    $G[d] \leftarrow \exp\left(-\frac{d^2}{2\sigma^2}\right)$
  end for
  return G
end function
$M_w \leftarrow 0$ ▷ Sum of weights for matches
$A_w \leftarrow 0$ ▷ Sum of weights for misalignments
for $i = 0$ to $|S_1| - 1$ **do**
  $M(i) \leftarrow 0$
  for $j = \max(0, i - w)$ to $\min(|S_2|, i + w + 1) - 1$ **do**
    if $S_1[i] = S_2[j]$ **then**
      $weight \leftarrow G[|i - j|]$
      $M(i) \leftarrow \max(M(i), weight)$
      if $weight = 1.0$ **then**
        break ▷ Early termination
      end if
    end if
  end for
  $M_w \leftarrow M_w + M(i)$
  if $M(i) > 0$ and $S_1[i] \neq S_2[j]$ **then**
    $A_w \leftarrow A_w + M(i)$
  end if
end for
$s_{ConvJ} \leftarrow \frac{1}{3}\left(\frac{M_w}{|S_1|} + \frac{M_w}{|S_2|} + \frac{M_w - A_w}{M_w}\right)$
return s_{ConvJ}

F. Computational Complexity and Applications

The time complexity of both the Jaro and Jaro-Winkler similarity algorithms is $\mathcal{O}(|S_1||S_2|)$, where $|S_1|$ and $|S_2|$ denote the lengths of the two strings being compared. This complexity arises from the requirement to potentially compare each character in one string to every character in the other string within a certain matching window, scaling with the product of the lengths of the two strings.

The memory complexity for these algorithms is $\mathcal{O}(|S_1| + |S_2|)$. This is due to the need to store intermediate match information, such as matched characters, for each string. This storage is necessary to calculate the final similarity score, including handling transpositions for the Jaro algorithm and prefix similarity adjustments in the Jaro-Winkler extension.

These complexities suggest that while Jaro and Jaro-Winkler are relatively efficient for shorter strings, their performance may degrade for longer strings due to the quadratic nature of their time complexity. Despite this, they are popular for many practical applications involving string similarity and approximate matching due to their simplicity and effectiveness.

Algorithm 4 Convolutional Jaro-Winkler Similarity (ConvJW)

Require: S_1, S_2 ▷ Two input strings
Require: σ ▷ Standard deviation for Gaussian weighting
Require: $p = 0.1$ ▷ Prefix scaling factor
Ensure: s_{ConvJW} ▷ Normalized ConvJW similarity score between 0 and 1

$s_{ConvJ} \leftarrow$ ConvJ$(S1, S2, \sigma)$
$l \leftarrow 0$
for $i = 0$ to $\min(\min(|S1|, |S2|), 4) - 1$ **do**
  if $S1[i] = S2[i]$ **then**
    $l \leftarrow l + 1$
  else
    break
  end if
end for
$p \leftarrow 0.1$ ▷ Scaling factor
$s_{\text{ConvJW}} \leftarrow s_J + l \cdot p \cdot (1 - s_J)$
return s_{ConvJW}

The ConvJ algorithm introduces more efficient computational approach to approximate string similarity, characterized by a quasilinear time complexity with respect to the length of one string and the fixed window size. Specifically, the time complexity is denoted as $\mathcal{O}(|S_1|w)$, where $|S_1|$ is the length of the first string and w is the fixed window size utilized in the similarity calculation. This efficiency is achieved by limiting comparisons to a fixed window around each character, significantly reducing the number of operations compared to traditional quadratic approaches.

Memory complexity for ConvJ is primarily influenced by the storage of precomputed Gaussian weights and intermediate calculations. Given the fixed window size w, the memory requirement is $\mathcal{O}(w)$, accounting for the Gaussian weight array and temporary variables used during computation. This compact memory footprint makes ConvJ particularly suitable for applications with stringent memory constraints.

Example 3. Convolutional Jaro Similarity (ConvJ) Consider the strings $S1 =$ "MARTHA" and $S2 =$ "MARHTA" for calculating Convolutional Jaro similarity with $\sigma = 2$ and $w = 7$.

Given the matching window determined by $w = 7$, all characters are within this range due to the strings' lengths. The Gaussian weight for each character position difference (distance d) is calculated using $G(i, j) = \exp\left(-\frac{d^2}{2\sigma^2}\right)$, with $\sigma = 2$.

For simplicity, assume the Gaussian weights for matching characters (ignoring character order) result in a sum of weights $M_w = 5.8$ (a hypothetical value for illustrative purposes, reflecting the sum of Gaussian weights for matched characters). Assuming no transpositions for a direct match scenario, the

ConvJ score is computed as:

$$s_{ConvJ}(S1, S2) = \frac{1}{3}\left(\frac{M_w}{|S1|} + \frac{M_w}{|S2|} + \frac{M_w}{M_w}\right) \quad (21)$$

$$= \frac{1}{3}\left(\frac{5.8}{6} + \frac{5.8}{6} + 1\right) \quad (22)$$

$$\approx 0.967 \quad (23)$$

This score is slightly adjusted due to the convolutional matching process, illustrating the nuanced similarity assessment provided by ConvJ.

Example 4. Convolutional Jaro-Winkler Similarity (ConvJW) For the strings $S1$ = "DWAYNE" and $S2$ = "DUANE", using $\sigma = 2$ and $w = 7$, and considering the ConvJW approach:

First, calculate the ConvJ similarity as above. For this example, let's assume a hypothetical sum of Gaussian-weighted matches $M_w = 4.5$ (for illustrative purposes), with a common prefix length $l = 1$ ('D'), and a prefix scaling factor $p = 0.1$.

The ConvJW similarity incorporates both the ConvJ score and an adjustment for the common prefix. Assuming the ConvJ score is approximately 0.822 (for continuity with the Jaro example):

$$s_{ConvJW}(S1, S2) \quad (24)$$

$$= s_{ConvJ}(S1, S2) + l \cdot p \cdot (1 - s_{ConvJ}(S1, S2)) \quad (25)$$

$$= 0.822 + 1 \cdot 0.1 \cdot (1 - 0.822) \quad (26)$$

$$= 0.822 + 0.018 = 0.840 \quad (27)$$

This score demonstrates the slight boost provided by the Jaro-Winkler adjustment in the convolutional context, highlighting the initial character similarity's importance.

These examples illustrate how ConvJ and ConvJW assess string similarity, taking into account both the convolutional matching strategy and the significance of starting characters, especially when applying Gaussian weighting with specific σ and w parameters.

VI. Implementation Details

As a reference implementation of the Jaro similarity for our experiments, we utilized the Rosetta Code implementation [27] (see Algorithm 1) in the C# programming language. In our implementation, we have made several enhancements to improve execution time.

Our proposed *ConvJ*, detailed in Algorithm 3, introduces several optimizations over the standard Jaro similarity calculation, leading to significant improvements in computational efficiency, accuracy and execution speed. Key enhancements include:

1) **Elimination of Match Tracking Arrays:** Unlike the original Jaro algorithm, which utilizes boolean arrays to track character matches between strings (lines 7-8 of Algorithm 1), *ConvJ* foregoes this approach. This optimization reduces memory overhead and eliminates the need for multiple array access operations, thereby enhancing runtime performance.
2) **Integrated Match and Miasalignment Calculation:** *ConvJ* computes character matches and misalignments simultaneously within a single iteration through each string. This integration contrasts with the Jaro method's sequential process, which separately identifies matches and then calculates transpositions, thus reducing the overall computational steps required. Through the loop in lines 24-27 of Algorithm 3, for each character in S_1, we seek matches in S_2 within the window w, optimizing both match detection and misalignment evaluation.
3) **Precomputed Gaussian Weights:** The algorithm leverages a precomputed vector of 1D Gaussian weights based on the relative character positions within a predefined window size. This precalculation avoids repetitive weight computations during runtime, leading to a more efficient execution. We precompute these values based on the window size w and standard deviation σ (lines 1-3 and the *PrecomputeGaussian* function in lines 4-9 of Algorithm 3).
4) **Localized Comparison with Gaussian Weighting:** By applying Gaussian weights to character comparisons, *ConvJ* emphasizes closer character matches over distant ones. This approach not only aligns with the intuitive understanding of string similarity but also minimizes unnecessary computations for characters outside the maximum window size, further speeding up the algorithm.
5) **Early Termination on Perfect Character Match:** If a perfect match (character equality with maximum Gaussian weight) is discovered, our algorithm terminates the inner loop early (line 20 of Algorithm 3), sidestepping unnecessary comparisons. This optimization proves particularly efficacious for strings with high similarity, curtailing the average computation time.

These improvements collectively contribute to a more performant execution of the *ConvJ* similarity measurement, making it particularly suitable for applications requiring high-throughput processing of string comparisons.

VII. Experiments

A. Overview

We evaluated the effectiveness of our novel algorithms, ConvJ and ConvJW, on a suite of datasets, segmented into two categories: Dataset A and Dataset B. Datasets A, which are enumerated in Table I, span various entities characterized by single attributes, as adapted from Cohen (2003) [21]. For Dataset B [28], outlined in Table II, we specifically focused on the first attribute, namely the name and title, for our experiments. These datasets include complex records from the Abt-Buy, Amazon-Google Products and DBLP-ACM sources, which are designed for benchmarking entity resolution tasks. The datasets are part of a collection curated by the DBS

Uni Leipzig, aimed at facilitating research in the domain of entity resolution by providing a diverse set of scenarios where entities need to be matched across different data sources despite discrepancies in their representations [29].

TABLE I. DATASET A USED IN EXPERIMENTS FROM ORIGINAL SOURCES [21]

Name	Number of strings	Name	Number of strings
Animal	5,709	Game	911
Bird Kunkel	336	Park	654
Bird Nybird	982	Restaurant	863
Bird Scott1	38	Ucd-people	90
Bird Scott2	719	Census	841
Business	2,139		

TABLE II. DATASET B USED IN EXPERIMENTS FROM ORIGINAL SOURCES

Dataset	#Tuples	#True matches	#Attributes
Abt-Buy	1081-1092	1097	4
Amazon-GoogleProducts	1363-3226	1300	4
DBLP-ACM	2614-2294	2224	4

B. Evaluation Metrics

Our analysis leveraged the non-interpolated average precision, adopting methodologies from prior works [21], [26]. Precision and recall are defined as follows:

$$\text{Precision} = \frac{c(i)}{i}, \tag{28}$$

$$\text{Recall} = \frac{c(i)}{m}, \tag{29}$$

where $c(i)$ represents the count of correct matches up to rank i, and m denotes the total correct matches. The interpolated precision at recall level r maximizes precision for all ranks i satisfying $c(i)/m \geq r$. Performance comparison among similarity functions utilizes the maximum F1-score, calculated by:

$$\text{F1-score} = 2 \times \frac{(\text{Precision} \times \text{Recall})}{\text{Precision} + \text{Recall}} \times 100\%. \tag{30}$$

Results are depicted in Tables I and II for Dataset A and B, emphasizing the comparative analysis of current state-of-the-art character-based similarity functions.

C. Experimental Results and Discussion

In our comprehensive analysis, the Convolutional Jaro (ConvJ) and Convolutional Jaro-Winkler (ConvJW) metrics were assessed across multiple datasets to evaluate their performance in comparison with state-of-the-art character based approximate string matching such as Jaro [1], Jaro-Winkler [2], Levenshtein [11], Damerau-Levenshtein [13], Smith-Waterman [15], and Needleman-Wunsch [14]. The evaluation focused on the F1-score, a critical metric reflecting both precision and recall, to provide a balanced view of each algorithm's accuracy and efficiency.

Superior F1-Score Performance: The results, particularly highlighted in Tables III and IV, demonstrate the superior performance of ConvJ and ConvJW algorithms. ConvJW, with a σ setting of 2, achieved the highest F1-score of 86.32% on Dataset A, surpassing the traditional Jaro-Winkler score of 81.45%. This improvement emphasizes the efficacy of integrating a convolutional methodology and Gaussian weighting to accurately capture the proximity of character positions with improved precision. Similarly, on Dataset B, ConvJ with a σ setting of 0.5 achieved an F1-score of 89.13%, outperforming Jaro-Winkler's 86.17%, further affirming the robustness of our proposed metrics in varied dataset contexts.

Optimal Sigma (σ) Settings: The performance sensitivity to the σ parameter was evident, where ConvJ and ConvJW's effectiveness varied with changes in σ. For instance, ConvJ's performance peaked at a σ value of 0.5 on Dataset B, indicating the importance of tuning this parameter to balance between emphasizing close character matches and accommodating positional variances. This adaptability allows for fine-tuning the algorithms to match the specific requirements of different datasets, contributing significantly to their superior performance (Fig. 3 and Fig. 4).

TABLE III. COMPARISON OF CHARACTER-BASED SIMILARITY FUNCTIONS RANKED IN DESCENDING ORDER OF F1-SCORE FOR DATASET A

Similarity Function	F1-score
ConvJW ($\sigma = 2$)	86.32 %
ConvJW ($\sigma = 1$)	86.12 %
ConvJW ($\sigma = 0.5$)	86.12 %
ConvJ ($\sigma = 1$)	84.99 %
ConvJ ($\sigma = 2$)	84.72 %
ConvJ ($\sigma = 0.5$)	84.54 %
Jaro-Winkler	81.45 %
Damerau-Levenshtein	76.86 %
Levenshtein	76.83 %
Needleman-Wunsch	76.25 %
Smith-Waterman	75.71 %
Jaro	75.29 %

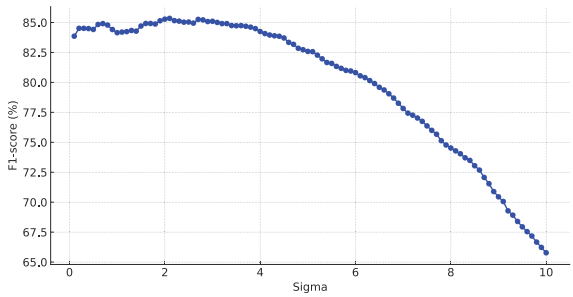

Fig. 3. F1-score performance as a function of the σ parameter in the ConvJ algorithm. The graph demonstrates the optimal σ value for maximizing the F1-score, illustrating the algorithm's sensitivity to this parameter for Dataset A.

D. Performance Analysis

Performance assessments were conducted on an Intel i7 11370H processor with 16GB RAM, comparing execution

TABLE IV. COMPARISON OF CHARACTER-BASED SIMILARITY FUNCTIONS RANKED IN DESCENDING ORDER FOR DATASET B

Similarity Function	F1-score
ConvJ ($\sigma = 0.5$)	89.13%
ConvJ ($\sigma = 1$)	89.03%
ConvJ ($\sigma = 2$)	88.55%
ConvJW ($\sigma = 0.5$)	88.38%
ConvJW ($\sigma = 1$)	88.22%
ConvJW ($\sigma = 2$)	87.99%
JaroWinkler	86.17%
Needleman-Wunsch	85.85%
Jaro	85.78%
Levenshtein	85.39%
Damerau-Levenshtein	85.37%
Smith-Waterman	84.38%

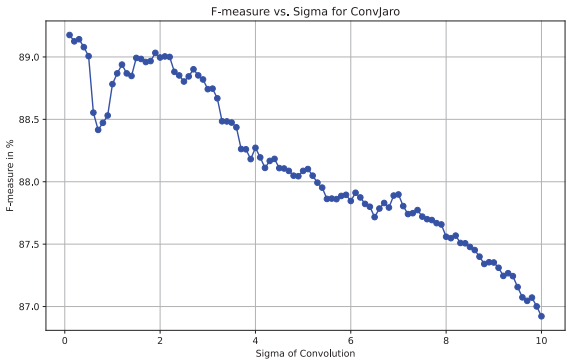

Fig. 4. F1-score performance as a function of the σ parameter in the ConvJ algorithm. The graph demonstrates the optimal σ value for maximizing the F1-score, illustrating the algorithm's sensitivity to this parameter for Dataset B.

TABLE V. PERFORMANCE OF STRING MATCHING ALGORITHMS ON DATASET A

Algorithm	Time (mm:ss:ms)	Performance Δ (%)
ConvJ ($\sigma = 0.5$)	0:02:412	0.00%
ConvJW ($\sigma = 0.5$)	0:02:660	+10.28%
ConvJ ($\sigma = 1$)	0:03:607	+49.54%
ConvJW ($\sigma = 1$)	0:03:841	+59.25%
ConvJ ($\sigma = 2$)	0:05:540	+129.68%
ConvJW ($\sigma = 2$)	0:05:577	+131.22%
Jaro	0:10:314	+327.61%
Jaro-Winkler	0:10:736	+345.11%
Levenshtein	0:33:629	+1294.24%
Damerau-Levenshtein	0:58:098	+2308.71%
Needleman-Wunsch	1:13:331	+2940.26%
Smith-Waterman	1:24:695	+3411.40%

TABLE VI. PERFORMANCE OF STRING MATCHING ALGORITHMS ON DATASET B

Algorithm	Time (mm:ss:ms)	Performance Δ (%)
ConvJ ($\sigma = 0.5$)	0:10:158	0.00%
ConvJW ($\sigma = 0.5$)	0:12:733	+25.34%
ConvJ ($\sigma = 1$)	0:15:948	+56.94%
ConvJW ($\sigma = 1$)	0:17:424	+71.54%
ConvJ ($\sigma = 2$)	0:25:537	+151.49%
ConvJW ($\sigma = 2$)	0:27:048	+166.32%
Jaro	1:21:675	+705.88%
Jaro-Winkler	1:33:368	+819.36%
Levenshtein	6:20:035	+3759.22%
Damerau-Levenshtein	11:20:111	+6708.99%
Needleman-Wunsch	15:45:774	+9296.85%
Smith-Waterman	16:38:783	+9815.41%

times and performance deltas to gauge the algorithms' efficiency. The summarized outcomes in Tables V and VI for Dataset A and B reveal the computational advantages of ConvJ and ConvJW against established string matching algorithms.

Computational Efficiency: Notably, ConvJ and ConvJW not only excelled in accuracy but also in computational efficiency. As evidenced in Tables V and VI, ConvJW ($\sigma = 0.5$) recorded the fastest execution time on Dataset B with 0:10:158, showcasing a substantial speed advantage over traditional metrics like Jaro (1:21:675) and Jaro-Winkler (1:33:368). This efficiency is paramount for large-scale applications, enabling rapid and precise string similarity assessments across extensive datasets.

VIII. CONCLUSION

In this paper, we introduced Convolutional Jaro (ConvJ) and Convolutional Jaro-Winkler (ConvJW), algorithms aimed at refining the precision of character-based approximate string matching. By integrating a convolutional methodology with Gaussian weighting, these algorithms more accurately assess the positional proximity of characters, a critical aspect often overlooked by traditional metrics. Our extensive evaluation across a variety of datasets revealed that ConvJ and ConvJW outperform existing methods, with ConvJ notably achieving execution speeds up to 7 times faster than the fastest known Jaro implementation and enhancing the F1-score by 10% compared to Jaro.

According to the results, it appears that ConvJ and ConJW exhibit superior performance for larger strings in comparison to traditional approaches. This improvement can be attributed to the fact that the window size is not proportional to the string size but is instead fixed and determined by the parameter σ, running in quasilinear time complexity $\mathcal{O}(|S_1|w)$.

The research establishes a new standard in the domain of character-based string matching, highlighting the potential of ConvJ and ConvJW to significantly improve upon the accuracy and efficiency of current approaches. The quasilinear time complexity and superior accuracy of these algorithms suggest a promising direction for future research, with implications for a wide range of applications in data mining, bioinformatics, and related fields. This work encourages further exploration into convolutional similarity measures and their potential to advance string matching techniques.

ACKNOWLEDGMENT

It was supported by the Erasmus+ project: Project number: 2022-1-SK01-KA220-HED-000089149, Project title: Including EVERyone in GREEN Data Analysis (EVERGREEN) funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily

reflect those of the European Union or the Slovak Academic Association for International Cooperation (SAAIC). Neither the European Union nor SAAIC can be held responsible for them.

REFERENCES

[1] M. A. Jaro, "Advances in record-linkage methodology as applied to matching the 1985 census of tampa, florida," *Journal of the American Statistical Association*, vol. 84, no. 406, pp. 414–420, 1989.

[2] W. E. Winkler, "String comparator metrics and enhanced decision rules in the fellegi-sunter model of record linkage." 1990.

[3] Y. Wang, J. Qin, and W. Wang, "Efficient approximate entity matching using jaro-winkler distance," in *International conference on web information systems engineering*. Springer, 2017, pp. 231–239.

[4] K. Dreßler and A.-C. Ngonga Ngomo, "On the efficient execution of bounded jaro-winkler distances," *Semantic Web*, vol. 8, no. 2, pp. 185–196, 2017.

[5] P. Pitchandi and M. Balakrishnan, "Document clustering analysis with aid of adaptive jaro winkler with jellyfish search clustering algorithm," *Advances in Engineering Software*, vol. 175, p. 103322, 2023.

[6] "Oracle database data quality operators documentation," https://docs.oracle.com/en/database/oracle/oracle-database/23/sqlrf/data-quality-operators.html, 2023, accessed: 2024-03-19.

[7] "Talend open studio documentation," https://help.talend.com/r/en-US/8.0/studio-user-guide/defining-matching-key, accessed: 2024-03-19.

[8] "Ibm infosphere documentation," https://www.ibm.com/docs/en/ignm/7.0.0?topic=overview-score-type, accessed: 2024-03-19.

[9] "Informatica data quality documentation," https://docs.informatica.com/, accessed: 2024-03-19.

[10] "Apache lucene documentation," https://lucene.apache.org, accessed: 2024-03-19.

[11] V. I. Levenshtein *et al.*, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet physics doklady*, vol. 10, no. 8. Soviet Union, 1966, pp. 707–710.

[12] G. Navarro, "A guided tour to approximate string matching," *ACM Computing Surveys*, vol. 33, no. 1, pp. 31–88, 2001.

[13] F. J. Damerau, "A technique for computer detection and correction of spelling errors," *Communications of the ACM*, vol. 7, no. 3, pp. 171–176, 1964.

[14] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of molecular biology*, vol. 48, no. 3, pp. 443–453, 1970.

[23] M. Zhang, J. Tang, and X. Zhang, "Text data processing and analysis: A brief survey," in *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, 2016, pp. 2065–2066.

[15] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *Journal of molecular biology*, vol. 147, no. 1, pp. 195–197, 1981.

[16] X. Fei, Z. Dan, L. Lina, M. Xin, and Z. Chunlei, "Fpgasw: accelerating large-scale smith–waterman sequence alignment application with backtracking on fpga linear systolic array," *Interdisciplinary Sciences: Computational Life Sciences*, vol. 10, pp. 176–188, 2018.

[17] M. Farrar, "Striped smith-waterman speeds database searches six times over other simd implementations," *Bioinformatics*, vol. 23, no. 2, pp. 156–161, 2007.

[18] L. Bergroth, H. Hakonen, and T. Raita, "A survey of longest common subsequence algorithms," *Proceedings of the Seventh International Symposium on String Processing and Information Retrieval (SPIRE 2000)*, pp. 39–48, 2000.

[19] D. Hirschberg, "A linear space algorithm for computing maximal common subsequences," *Communications of the ACM*, vol. 18, no. 6, pp. 341–343, 1975.

[20] P. Christen, "A comparison of personal name matching: Techniques and practical issues," in *Sixth IEEE International Conference on Data Mining-Workshops (ICDMW'06)*. IEEE, 2006, pp. 290–294.

[21] W. W. Cohen, P. Ravikumar, S. E. Fienberg *et al.*, "A comparison of string distance metrics for name-matching tasks." in *IIWeb*, vol. 3, 2003, pp. 73–78.

[22] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," *Advances in neural information processing systems*, vol. 26, 2013.

[24] A. Conneau, H. Schwenk, L. Barrault, and Y. Lecun, "Very deep convolutional networks for text classification," *arXiv preprint arXiv:1606.01781*, 2016.

[25] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, 2014, pp. 1532–1543.

[26] N. Gali, R. Mariescu-Istodor, D. Hostettler, and P. Fränti, "Framework for syntactic string similarity measures," *Expert Systems with Applications*, vol. 129, pp. 169–185, 2019.

[27] Rosetta Code, "Jaro similarity," https://rosettacode.org/wiki/Jaro_similarity, n.d., accessed: 2024-02-28.

[28] H. Köpcke, A. Thor, and E. Rahm, "Evaluation of entity resolution approaches on real-world match problems," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 484–493, 2010.

[29] DBS Universität Leipzig, "Benchmark datasets for entity resolution," https://dbs.uni-leipzig.de/en/research/projects/object_matching/benchmark_datasets_for_entity_resolution, 2023, accessed: 2023-02-21. [Online]. Available: https://dbs.uni-leipzig.de/en/research/projects/object_matching/benchmark_datasets_for_entity_resolution