Hybrid Algorithm for Approximate String Matching to be used for Information Retrieval

Surbhi Arora, Ira Pandey

Abstract— Conventional database searches require the user to hit a complete, correct query denying the possibility that any legitimate typographical variation or spelling error in the query will simply fail the search procedure. An approximate string matching engine, often colloquially referred to as fuzzy keyword search engine, could be a potential solution for all such search breakdowns. A fuzzy matching program can be summarized as the Google's 'Did you mean: ...' or Yahoo's 'Including results for ...'. These programs are entitled to be fuzzy since they don't employ strict checking and hence, confining the results to 0 or 1, i.e. no match or exact match. Rather, are designed to handle the concept of partial truth; be it DNA mapping, record screening, or simply web browsing. With the help of a 0.4 million English words dictionary acting as the underlying data source, thereby qualifying as Big Data, the study involves use of Apache Hadoop's MapReduce programming paradigm to perform approximate string matching. Aim is to design a system prototype to demonstrate the practicality of our solution.

Index Terms— Approximate String Matching, Fuzzy, Information Retrieval, Jaro-Winkler, Levenshtein, MapReduce, N-Gram, Edit Distance

—————————— ◆ ——————————

1 Introduction

FUZZY keyword search engine employs approximate search matching algorithms for Information Retrieval (IR). Approximate string matching involves spotting all text matching the text pattern of given search query but with limited number of errors [1]. The errors are expressed by a metric, edit distance between the spotted text and the search query text. This approach diverges from the exact string matching, wherein the search query results return either no text or either exactly matched text which is not approximated in any way. Since text data is pervasive and finding some specific information from variegated, inharmonious data sources involves identifying database records approximately similar to the base entity in order to achieve information integration.

In this paper, comparative study of few string similarity algorithms has been conducted, which perform "approximate string search" to extract all information relevant to the given search query from the underlying information pool as shown in Fig. 1.

Fig. 1. Fuzzy Search results for query string, "noting".

The fuzzy logic behind approximate string searching can be described by considering a fuzzy set F over a referential universal set U, characterized by a membership function, $m=\mu(F)$. The membership grade of each element $x \in U$ depends on the value of $m(x)$ which is a real number in the interval $[0,1]$ [2]. The membership grade can be regarded as edit distance [3], such that, $x \in U$ is,

1. **exact match**, if $m(x) = 1$
2. **no match**, if $m(x) = 0$
3. **partial match**, if $0 < m(x) < 1$ (fuzzy member in F)

Key contribution areas of this study are:

1. Understand the domain of the project, that is Big Data Analytics and decide technology platform used to build the desired product i.e. Fuzzy Keyword Search Engine.
2. Analyze a few known standard string matching algorithms and choose the most suitable string matching algorithm for the designing of the Fuzzy Search Engine.
3. Devise a hybrid approach for approximate string matching based on above comparative analysis.

2 Hadoop vs. Java IDE: Platform for Fuzzy Keyword Search Engine

Given a huge collection of dictionary words as underlying information pool, analysis of one of the standard approximate string matching algorithm, Jaro-Winkler Distance on a common Java IDE such as NetBeans versus on Hadoop clearly indicates that as the information pool increases, the execution time increases by manifold on Java IDE as compared to on Hadoop. The experimental observations, therefore conclude that MapReduce paradigm in Hadoop is better for voluminous data, since Mappers filter and transform input data and Reducers aggregate mappers' output, allowing parallel execution flow, thereby reducing the execution time

————————————————

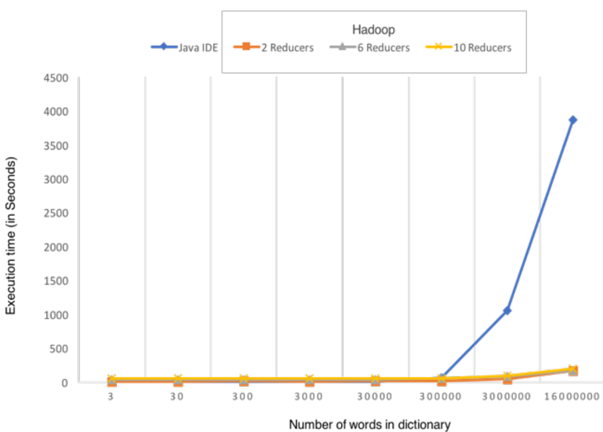
- *Surbhi Arora holds a bachelor's degree in IT and is currently working as a Software Engineer at Sprinklr, Gurgaon, India. PH- +91 9711177495. E-mail: arora.94surbhi@gmail.com*
- *Ira Pandey holds a bachelor's degree in IT and is currently working as a Software Engineer at Accenture, Pune, India, PH- +91 9971225539. E-mail: irap94@gmail.com*

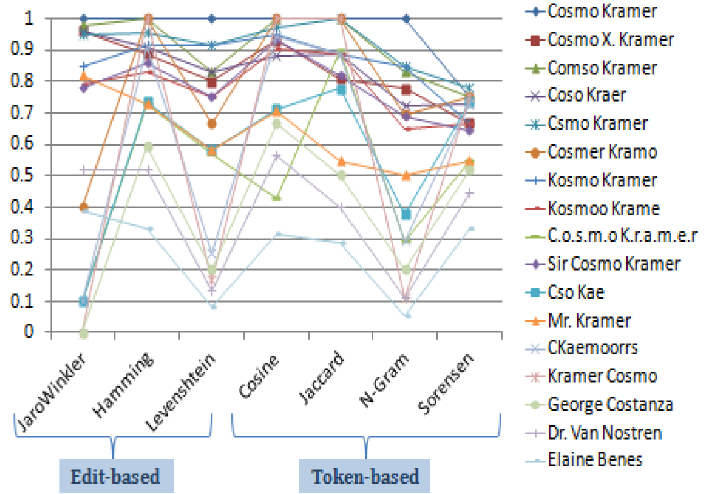
[4]. Hence, Fuzzy keyword search engine is implemented using Hadoop. Also, it was observed that, with a smaller information pool, increase in number of reducers in Map-Reduce Framework does not improve the performance of Fuzzy Keyword Search Engine, as combining results of all reducers becomes a time overhead.

TABLE 1
EXECUTION TIME - HADOOP VS. JAVA IDE

Words in Dictionary	Execution time in milliseconds			
	IDE (NetBeans)	Hadoop (Map-Reduce Paradigm)		
		2 Reducers	6 Reducers	10Reducers
3	4	12	33	53
30	5	12	34	53
300	5	13	32	52
3000	6	12	32	54
30000	10	13	40	53
300000	64	15	51	59
3000000	1057	41	88	97
16000000	3870	172	173	200

IJSER





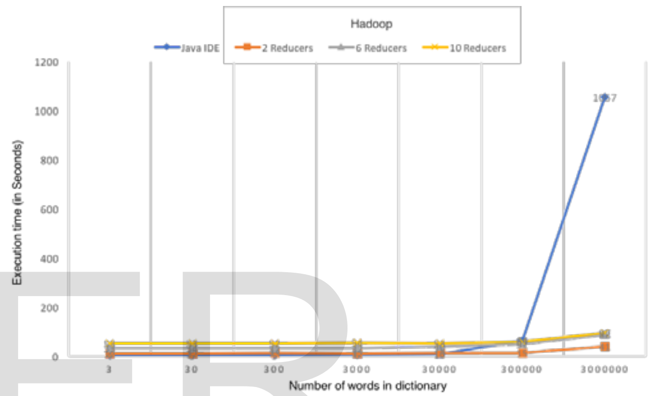


Fig. 2(a). Increase in execution time - Hadoop vs. Java IDE for up to 3 million words.

Fig. 2(b). Increase in execution time - Hadoop vs. Java IDE for up to 16 million words.

3 Proposed Methodology

3.1 Preliminaries

Edit Distance, d: Metric chosen to denote the similarity coefficient of the two strings being matched. Also, d=1, if the two strings match exactly; d=0, if there is no similarity between them; and otherwise 'd' takes a decimal value in the interval [0,1] depending upon their degree of match.

Fig. 3. Variations in String Distances, d, computed by comparing the given strings with query string "Cosmo Kramer" using various approximate string matching algorithms under study.

Threshold, t: Threshold, t acts as a filter to retrieve only relevant information. In order to retrieve top-k matched strings from a given string collection for a particular fuzzy query, the retrievals are made out of the sub-string collection that have string distance greater than or equal to the threshold value set, i.e.,

$$d \geq t \quad (1)$$

Strings with d< t are hence, pruned from the result set.

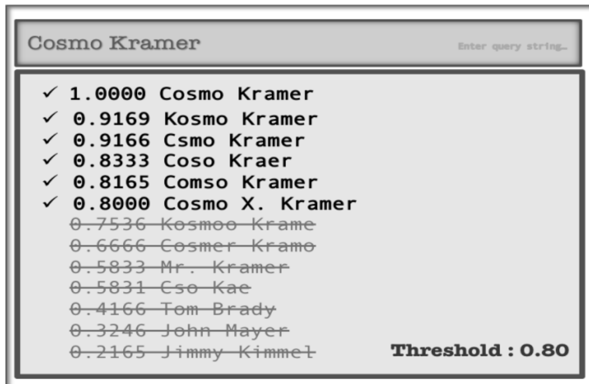


Fig. 4. Computation of String Distances, d, for strings in a given collection against the query string 'Cosmo Kramer' using a standard approximate string matching algorithm. Results are pruned in accordance with the threshold, t is set to 0.80

This study comparatively analyses the performance of some standard Edit-based and Token-based approximate string matching algorithms primarily on the basis of two parameters:

1. Quality of prediction, based on the relevance of retrieved results.
2. Performance, determined by execution time of the algorithm.

3.2 Measures for Quality of Prediction

The relevance of retrieved results is based on the values of these accuracy measures - Precision (*p*), Recall (*r*) and F-measure (*f*) [5].

1. Precision, p : proportion of retrieved records that are relevant [6].
2. Recall, r : proportion of relevant records actually retrieved [6].
3. F-measure, f : Harmonic mean of precision and recall.

Based on the given Fig. 5, accuracy measures are calculated as:

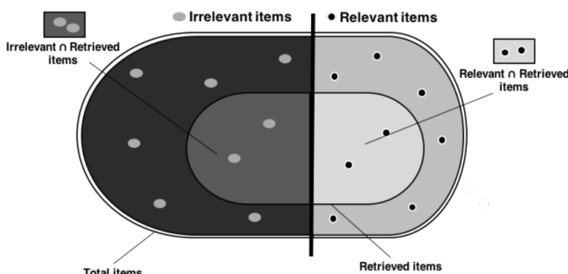


Fig. 5. Venn diagram: Accuracy measures for estimating 'quality of prediction'.

$$p = (retrieved \cap relevant)/retrieved \quad (2)$$

$$r = (retrieved \cap relevant)/relevant \quad (3)$$

$$f = (2 \times p \times r)/(p + r) \quad (4)$$

3.3 Estimating Accuracy measures

Considering a word list containing around 0.4 million English words, sorted alphabetically, as the data source, 15 query words are picked and a list of 10 relevant word searches from the data source (typographically similar to the query word) is pre-assumed for each of the query word. For each query word, string-distance, d is computed by executing the approximate string searching algorithm against each single word in the data source. Based on the string-distance (d) computed, threshold (t) set and pre-assumed list of relevant words, words in the data source are categorized into following:

1. **relevant**: If the word ∈ pre-assumed list of relevant words
2. **retrieved**: If d ≥ t
3. **retrieved ∩ relevant**: If the word ∈ pre-assumed list of relevant words and d ≥ t
4. **retrieved ∩ irrelevant**: If the word ∉ pre-assumed list of relevant words and d ≥ t
5. **irrelevant**: If the word ∉ pre-assumed list of relevant words

Precision, Recall and F-measures are calculated for each query word case using "(2)", "(3)" and "(4)" respectively. Accordingly, minimum, maximum and average values of accuracy measures are computed for each approximate string matching algorithm.

4 Approximate String Matching Algorithms: An Overview

Approximate String Matching algorithms are broadly categorized into edit-based and token-based measures [7].

Edit-based measures: Individual characters in the string are considered and String Distance, d is measured as the computation expense of converting one string to other.

Token-based measures: String is divided into n-grams, i.e. substrings of consecutive characters each of length n and String Distance, d is measured on basis of gram similarity between both the strings.

TABLE 2

APPROXIMATE STRING MATCHING ALGORITHMS

Algorithm Family	Algorithm	Time Complexity	Space Complexity
Exact Matching	Character comparison	O(min(n, m))	O (1)
Edit Based (Approximate String Matching)	Levenshtein Distance	O(nm)	O(m*n)
	Jaro-Winkler Distance	O(n+m)	
	Jaro Distance	O(n+m)	
	Hamming Distance	O(max(n, m))	O (1)
Token based (Approximate String Matching)	N-Gram Approach	O(n+m)	O(n+m)
	Cosine Similarity	O(n*m)	O(n+m)
	Jaccard Index	O(n+m)	O (1)
	Sorensen Coefficient	O(n+m)	O(1)

SUMMARY

4.1 Levenshtein Distance

Given two strings, *str1* and *str2* of length *l1* and *l2* respectively, Levenshtein distance between the two strings depends on the minimum number of insertion, deletion and substitution operations required for transforming *str1* to *str2*. Number of substitutions is computed using "(5)" and

IJSER

Levenshtein distance is computed using "(6)" [8].
$x \in$ str1 and $y \in$ str2, at position t

$$S = Min(l1, l2) - \sum \begin{cases} \exists x, \exists y, t \quad if\ x = y & 0 \\ otherwise & 1 \end{cases} \quad (5)$$

$$Lev(str1, str2) = Min(l1, l2) - \sum \begin{cases} \exists x, \exists y, t \quad if\ x = y & 0 \\ otherwise & 1 \end{cases} + Abs(l1 - l2) \quad (6)$$

4.2 Jaro and Jaro-Winkler Distance

Given two strings, *str1* and *str2* of length *l1* and *l2* respectively, wherein,
m = number of matching characters,
t = number of transposed characters,
Jaro distance, d_j can be calculated using "(7)". [7]

$$d_j = \frac{1}{3}\left(\frac{m}{l1} + \frac{m}{l2} + \frac{m - \frac{t}{2}}{m}\right) \quad (7)$$

Jaro-Winkler distance, d_w can be calculated using "(8)", given, [16]
l = prefix length (number of starting characters in both strings that matched, max length = 4), and
p = prefix weight (default = 0.1),

$$d_w = d_j + lp(1 - d_j) \quad (8)$$

4.3 Hamming Distance

Given two strings, *str1* and *str2* of equal length *l*, Hamming Distance, HD is the minimum number of single character substitutions to transform *str1* to *str2* [8].

$$HD = \sum \begin{cases} \exists x, \exists y, t \quad if\ x = y & 0 \\ otherwise & 1 \end{cases} \quad (9)$$

4.4 N-Gram Approach

Given two strings, *str1* and *str2* of any lengths, N-Gram similarity coefficient can be calculated using "(10)" [8], given,
n1 = number of N-Grams in str1,
n2 = number of N-Grams in str2, and
$n1 \cap n2$ *= number of common N-Grams in str1 and str2, where,*
N-Grams are defined as substrings of consecutive characters each of length N.

$$N - Gram(str1, str2) = \frac{1}{1 + n1 + n2 - 2 \times (n1 \cap n2)} \quad (10)$$

4.5 Cosine Similarity

Cosine similarity measure is a token-based measure, where the whole string is split either into words or into characters to form a vector out of each string based on frequency of each word/character in the string. Given two strings, *str1* and *str2*, cosine similarity can be expressed mathematically as, [9]

$$similarity(\overrightarrow{str1}, \overrightarrow{str2}) = cos\theta = \frac{\overrightarrow{str1} . \overrightarrow{str2}}{|\overrightarrow{str1}||\overrightarrow{str2}|} \quad (11)$$

$$\frac{\overrightarrow{str1} . \overrightarrow{str2}}{|\overrightarrow{str1}||\overrightarrow{str2}|} = \frac{\sum_{i=1}^{n} str1_i\, str2_i}{\sqrt{\sum_{i=1}^{n} str1_i^2} \times \sqrt{\sum_{i=1}^{n} str2_i^2}} \quad (12)$$

4.6 Jaccard Index

Given two strings, *str1* and *str2* of any lengths, Jaccard index, *J* can be calculated using "(13)", given,
$n1 \cap n2$ *= intersection of N-Grams in str1 and str2, and,*
$n1 \cup n2$ *= union of N-Grams in str1 and str2, and,* [10]

$$J(str1, str2) = \frac{|n1 \cap n2|}{|n1 \cup n2|} = \frac{|n1 \cap n2|}{|n1| + |n2| - |n1 \cap n2|} \quad (13)$$

4.7 Sorensen Coefficient

Given two strings, *str1* and *str2* of any lengths, Sorensen coefficient, *s* can be calculated using "(14)", given,
n_t *= number of character bigrams found in both strings,*
n_{str1} *= number of character bigrams found in str1, and*
n_{str2} *= number of character bigrams found in str2,* [7]

$$s = \frac{2n_t}{n_{str1} + n_{str2}} \quad (14)$$

5 Comparative Analysis of Existing Approximate String Matching Algorithms

5.1 Criteria for analyzing performance

1. Performance based on execution time
2. Performance based on accuracy measures: Precision, Recall and F-measure as described in Fig. 5.

5.2 Plotting average execution time

Average execution time for standard edit and token based approximate string matching algorithms is computed by taking mean of execution times of sample runs for each algorithm under study.

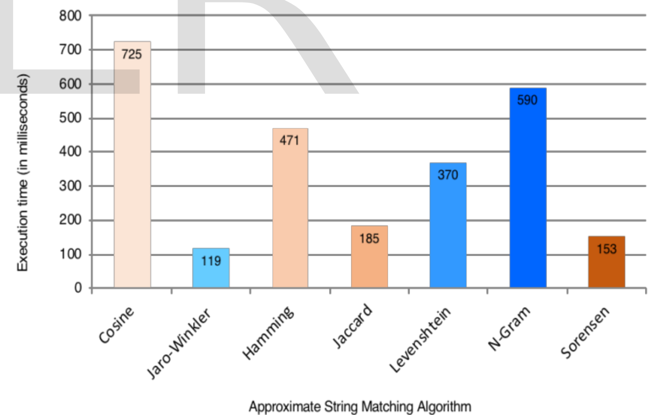
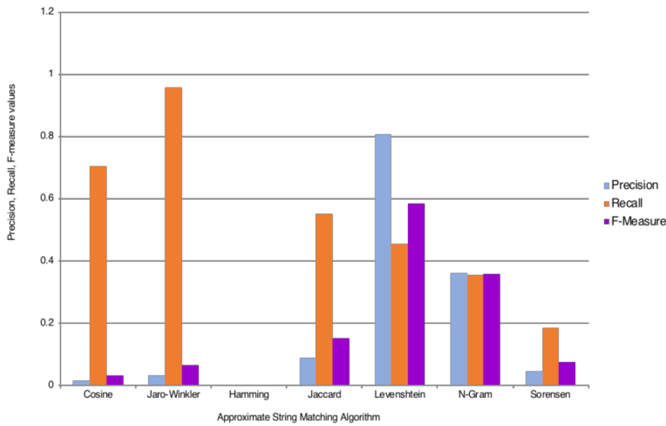


Fig. 6. Average execution time plot for various Approximate String Searching algorithms

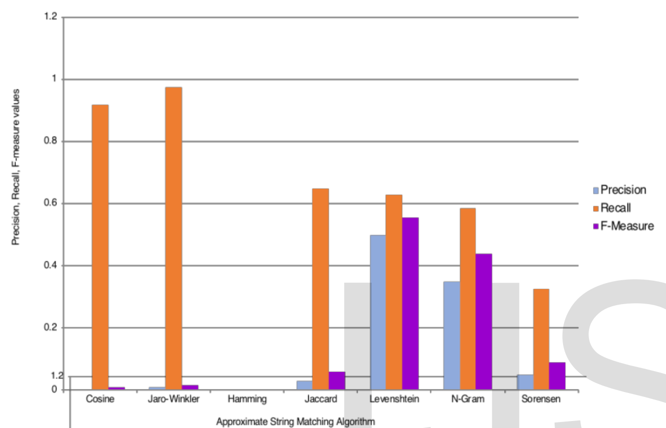
5.3 Plotting accuracy measures

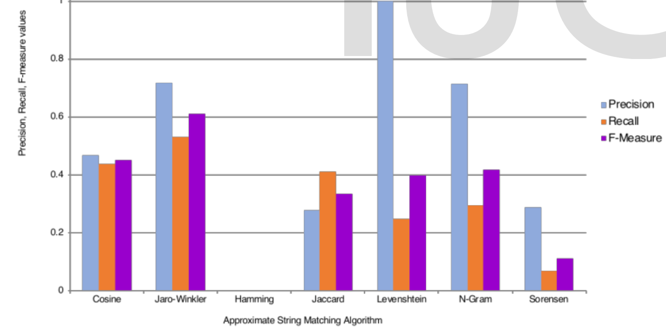
Precision values are undefined, i.e. n/a if all search results are pruned as their string-distances (d) are less than threshold (t) set, $d < t$. This can be inferred from "(2)".
Recall values are 0 if either all search results are pruned, since, $d < t$ or if none of the retrieved results belong to the pre-assumed relevant word list. This can be inferred from "(3)".

ISSN 2229-5518



F-measure values are undefined, n/a if the precision for the





corresponding search result is undefined. This can be inferred from "(4)".

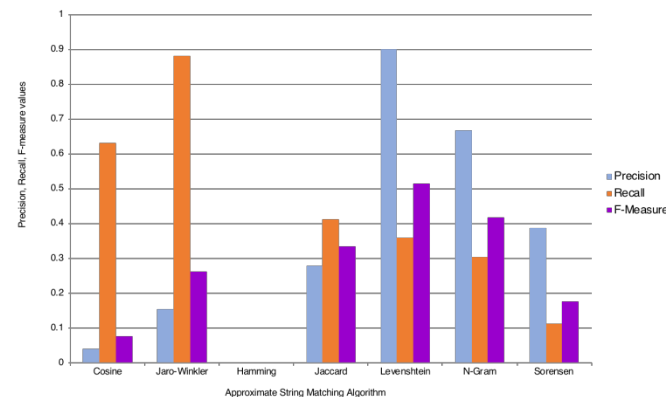


Fig. 7(a). Expected accuracy measures for threshold = 0.95

Fig. 7(b). Expected accuracy measures for threshold = 0.90

Fig. 7(c). Expected accuracy measures for threshold = 0.85

Fig. 7(d). Expected accuracy measures for threshold = 0.80

Fig. 7(e). Expected accuracy measures for threshold = 0.75

6 Results and Discussion

6.1 What should be threshold value for safe pruning?

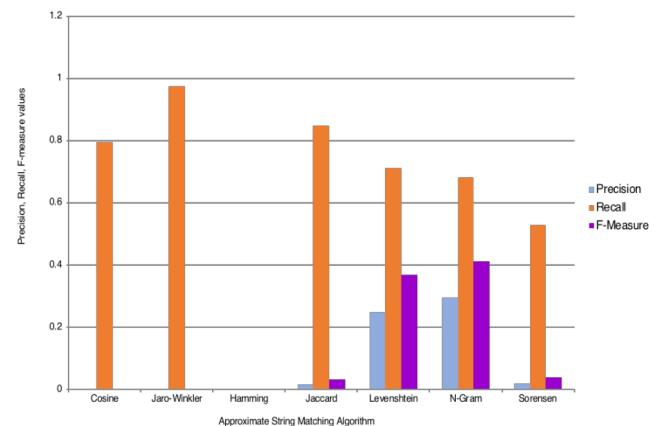
As the threshold value decreases, more results are retrieved and hence precision decreases and recall rate of algorithm increases. This can be inferred from "(2)" and "(3)". Referring to the plotted accuracy measures for different threshold values, **0.90 is assumed to be the working threshold for the algorithms**. Accuracy measures for threshold 0.90 are depicted in Fig. 7(b). Higher threshold values like 0.95 are not safe as there is risk of pruning the desired searches. Moreover, with lower thresholds, precision values stoop so low as even undesired searches are retrieved to a great extent, depicted by "(15)". Also, it was found that all algorithms except Hamming Distance, yield string-distances of all relevant searches for a given search string within this given threshold.

6.2 Which algorithms provide more relevant results?

Observations made from tabulated accuracy measures:

1. Levenshtein and N-Gram have higher precision rates than Jaro Winkler and the rest.
2. Jaro Winkler has the maximum recall rate but lower precision rates. Though it retrieves relevant searches in the given threshold, but the problem is it retrieves too many! This can be inferred from "(15)".

IJSER



3. Jaro-Winkler, Levenshtein and N-Gram have better accuracy rates than the rest.
4. Hamming distance has the worst since it needs lower threshold values to retrieve results. For threshold 0.90, its precision and f-measure values are undefined.

$$\downarrow p = \frac{retrieved \cap relevant}{retrieved \uparrow} \quad (15)$$

6.3 Which algorithms run faster?

Considering execution time for the algorithms: Jaro-Winkler,

Levenshtein and N-Gram with fairly better accuracy measures than the rest,

1. Jaro-Winkler has the least execution time as plotted in Fig. 6.
2. Levenshtein falls in the mid-range and N-Gram has slightly higher execution time.

7 DEVISED HYBRID APPROACH: LEVIGRAM

7.1 Computing Levigram similarity index

Levigram is hybrid of Jaro-Winkler, Levenshtein and N-Gram. In case of Levigram and Levenshtein, string-distance, *d* between two strings, *str1* and *str2*, is computed using dynamic programming from a tabular computation of matrix *L(n, m)*. The approach boils down to computing *L(i, j)* and *C(i, j)*, $\forall i \in (0,n)$ and $\forall j \in (0,m)$, where,

str1[i] = N-Gram at position i in str1
str2[j] = N-Gram at position j in str2
C(i,j) = Cost for transforming str1[i] to str2[j]

$$L(i,j) = L_{i,j} = \begin{cases} C_{i,j} + L_{i-1,j-1} & if\ str1[i] = str2[j] \\ C_{i,j} + \min(L_{i-1,j}, L_{i,j-1}, L_{i-1,j-1}) & else \end{cases} \quad (16)$$

Levenshtein algorithm is operated on individual characters of string, so N=1 for N-Gram, and cost for transforming is uniform, i.e.

$$cost_{i,j} = \begin{cases} 0 & if\ str1[i] = str2[j] \\ 1 & esle \end{cases} \quad (17)$$

and, Levenshtein distance is calculated as,

$$L(i,j) = L_{i,j} = \begin{cases} L_{i-1,j-1} & if\ str1[i] = str2[j] \\ 1 + \min(L_{i-1,j}, L_{i,j-1}, L_{i-1,j-1}) & else \end{cases} \quad (18)$$

Levigram, devised hybrid algorithm, is operated on bi-grams of string, so N=2 for N-Gram, and cost for transformation is not uniform. Then, cost is estimated by computing matches, $m_{i,j}$ and transpositions, $t_{i,j}$, $\forall i \in (0,n)$ and $\forall j \in (0,m)$, using Jaro-Winkler string-distance concept. Mismatches between str1[i] and str2[i] are calculated using ("19") and as,.

$$mismatches_{i,j} = \min(|str1|, |str2|) - (m_{i,j} - t_{i,j}) \quad (19)$$

Cost is calculated as,

$$cost_{i,j} = \begin{cases} 0 & if\ str1[i] = str2[j] \\ \dfrac{mismatches_{i,j}}{N - Gram} & esle \end{cases} \quad (20)$$

and, Levigram distance matrix is computed as,

$$L(i,j) = \begin{cases} L_{i-1,j-1} & if\ str1[i] = str2[j] \\ \dfrac{mismatches_{i,j}}{2} + \min(L_{i-1,j}, L_{i,j-1}, L_{i-1,j-1}) & else \end{cases} \quad (21)$$

$$Levigram\ Disatnce = L(n,m) = L[n][m] \quad (22)$$

$$Sim_{Levigram} = \frac{1 - L[n][m]}{\min(|str1|, |str2|) + 1} \quad (23)$$

7.2 Comparative analysis of Execution Time

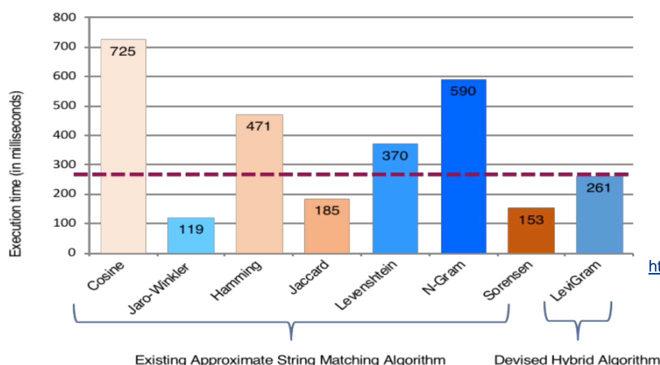


Fig. 8. Average execution time plot for various existing Approximate String Searching algorithms vs. Levigram, devised hybrid approach

Though Levigram doesn't have the least execution time but its dynamic approach beats Levenshtein and N-Gram.

7.3 Comparative analysis of Accuracy Measures

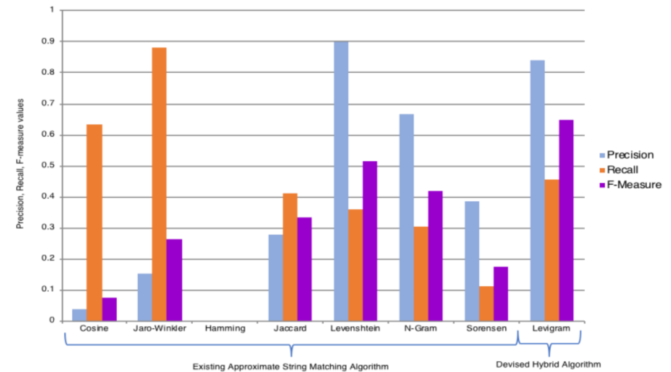


Fig. 9. Expected accuracy measures plot for standard Approximate String Searching algorithms under study vs. Levigram, devised hybrid approach

7.4 Levigram : Psuedo Algorithm

Given two strings, a and b, algorithm Levigram retuns edit distance bettween the two strings.

algorithm Levigram **is**
  input: strings a[1..length(a)], b[1..length(b)]
  output: float edit_dist
  nGram := 2
  a := "." + a
  b := "." +b
  n := a.length() - nGram + 1
  m := b.length() - nGram +1
  min_dist := min(a.length(),b.length())
  match_a : = ""
  match_b : = ""
  mRange := 0
let d[0..n+1, 0..m+1] be a 2-d array of integers, dimensions n+2, m+2
if n = 0 | | m = 0 **then**
  return 0.00;
for i := 0 **to** n **inclusive do**
  d[i, 0] := (float)i
for j := 0 **to** m **inclusive do**
  d[0, j] := (float)j
for i := 1 **to** n **inclusive do**
  a_i := a.substring(i-1, i-1+nGram)
  for j := 1 **to** m **inclusive do**
    b_j := b.substring(j-1, j-1+nGram)
    if a_i = b_j **then**
      cost := 0

IJSER

```
        d[i][j] := d[i-1][j-1] + cost
      else
        matches := getMatch(a_i,b_j)
        transpositions := 0
        if getMissmatch(b_j,a_i) > 0 then
           transpositions := (getMissmatch(a_i,b_j)/
              getMissmatch(b_j,a_i))
        missMatches := min_dist - (matches - transpositions)
        cost = missMatches/ (float)nGram
        d[i, j] := minimum(d[i−1, j−1] + cost,  //substitution
                 d[i,  j−1] + cost,  //insertion
                 d[i−1, j  ] + cost)  //deletion
levigram_edit_dist := 1-d[n][m]/(min_dist+1)
return levigram_edit_dist


function getMatch is
   input: strings a_comp, b_comp
   output: integer matches
matches := 0
mRange := max(a_comp.length(),b_comp.length())/2 - 1
for i := 0 to a_comp.length() inclusive do
   counter :=0
   while counter<=mRange & i>=0 & counter<=i inclusive do
     if a_comp.charAt(i)=b_comp.charAt(i-counter) inclusive do
        matches := matches +1
        match_a := match_a + a_comp.charAt(i)
        match_b := match_b + b_comp.charAt(i)
     counter := counter + 1
   counter :=1
   while counter<=mRange & i<b_comp.length() inclusive do
     if a_comp.charAt(i)=b_comp.charAt(i+counter) inclusive do
        matches := matches +1
        match_a := match_a + a_comp.charAt(i)
        match_b := match_b + b_comp.charAt(i)
     counter := counter + 1
return matches


function getMissMatch is
   input: strings a_comp, b_comp
   output: integer missMatches
transpositions := 0
for i := 0 to match_a.length() inclusive do
   counter :=0
   while counter<=mRange & i<match_b.length() & (counter+i)<
match_b.length() inclusive do
     if match_a.charAt(i)=match_b.charAt(i+counter) &counter>0
inclusive do
        transpositions := transpositions +1
     counter := counter + 1
   counter :=1
return transpositions
```

8 Conclusions

Comparative study of existing approximate string matching methods suggests that finding an unrivaled method with highest accuracy measures is not feasible. Some methods have high precision rates while others may have high recall rates. Hybrid approach seems to fit best in this case, producing all possible matches and pruning the undesired matches safely. Levigram, devised hybrid algorithm is a composite method, incorporating the attributes of both edit-based and token-based string distance methods. Token-based methods are likely to improve precision, since they operate on substrings of consecutive characters, producing bordering results. Moreover, the value of threshold could be adjusted to one's requirement of how many matches one needs to filter. Since, lowering the threshold, increases the number of matches retrieved and vice versa.

Acknowledgment

The authors wish to thank **Mrs. Neetika Bhandari**, their Faculty Advisor at Indira Gandhi Delhi Technical University, Delhi, India, for her guidance and constant supervision much required for completion of this study.

References

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

[2] S. Gottwald. "Logical Preliminaries: Basic Notations". Fuzzy sets and fuzzy logic. Verlag Vieweg, Braunschweig, 1993, pp. 1-3.

[3] Esko Ukkonen, "Algorithms for Approximate String Matching*", ELSEVIER, Information and Control vol. 64, no. 1-3, pp. 110-118, January-March 1985.

[4] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler, "The Hadoop Distributed File System," *Proc. IEEE 26th Symp. Mass Storage Systems and Technologies (MSST '10)*, pp. 1-10, May. 2010

[5] Tom Fawcett, "An introduction to ROC analysis", ELSEVIER, Pattern Recognition Letters vol. 27, no. 8, pp. 861-874, July 2006, doi:10.1016/j.patrec.2005.10.010.

[6] Patrick A.V. Hall, Geoff R. Dowling, "Approximate String Matching", ACM Computing Surveys, vol. 12, no. 4, pp. 381–402, December 1980.

[7] M.Bilenko, R. Mooney, William W. Cohen, Pradeep Ravikumar, and Stephen E. Fienberg, "Adaptive name matching in information integration", IEEE Intelligent Systems, vol. 18, no. 5, Sep/Oct 2003, doi:10.1109/MIS.2003.1234765.

[8] V.P. Sumathi, Dr. K. Kousalya, R. Kalaiselvi, "A Comparative study on Syntax Matching Algorithms in Semantic Web", WSEAS Transactions on COMPUTERS, EISSN: 2224 2872, vol. 4, 2015

[9] Anna Huang, "Similarity Measures for Text Document Clustering", NZCSRSC 2008, April 2008, Christchurch, New Zealand.

[10] Shraddha Pandit, and Suchita Gupta, "A comparative study on distance measuring approaches for clustering", IJORCS, vol. 2, no. 1, pp. 29-31. 2011.

IJSER