

# Comparison of String Similarity Algorithm in post-processing OCR

**Al Birr Karim Susanto, Nuraziz Muliadi, Bagus Nugroho, Muljono\***

*Department of Informatics Engineering, Dian Nuswantoro University*

*E-mail: [albirkarim1@gmail.com](mailto:albirkarim1@gmail.com), [nurazizmuliadi9@gmail.com](mailto:nurazizmuliadi9@gmail.com),*

*[bagoescomputer@gmail.com](mailto:bagoescomputer@gmail.com), [muljono@dsn.dinus.ac.id](mailto:muljono@dsn.dinus.ac.id)*

---

**Abstract** - The Optical Character Recognition (OCR) problem that often occurs is that the image used, has a lot of noise covering letters in a word partially. This can cause misspellings in the process of word recognition or detection in the image. After the OCR process, we must do some post-processing for correcting the word. The words will be corrected using a string similarity algorithm. So what is the best algorithm? We conducted a comparison algorithm including the Levenshtein distance, Hamming distance, Jaro-Winkler, and Sørensen – Dice coefficient. After testing, the most effective algorithm is the Sørensen-Dice coefficient with a value of 0.88 for the value of precision, recall, and F1 score.

**Keywords** - Post-processing OCR, Levenshtein distance, Hamming distance, Jaro-Winkler, Sørensen–Dice coefficient, Website.

## 1. INTRODUCTION

---

OCR (Optical Character Recognition) is a method that can be used as word recognition in an image. OCR technology has a broad range of applications in document processing. As in previous research [1], OCR recognized street names on street signs and digitized books to search for text in them digitally. However, some weaknesses occur, such as the image used has noise that can be annoying because it covers some of the letters in a word. It will result in a spelling error to make a word or sentence that is not following the original.

There are many OCR engines. One of them is Tesseract [2]. This library is developed using C and C++. This research uses the javascript version of tesseract known as tesseract.js made by Jerome Wu.

We will make post-processing the text by comparing several algorithms that are more effective in solving these problems. Not only better accuracy but the algorithm must be fast too. The speed of algorithm is very important if we deal with realtime OCR, like street sign recognition and etc.

The algorithms that will be compared are the Levenshtein Distance, Hamming distance, Jaro-Winkler, and Sørensen – Dice coefficient. From the above problems, we hoped that this research can be considered to increase the effectiveness and accuracy in processing when using algorithms in post-processing.

## 2. RESEARCH METHOD

### 2.1. Methodology

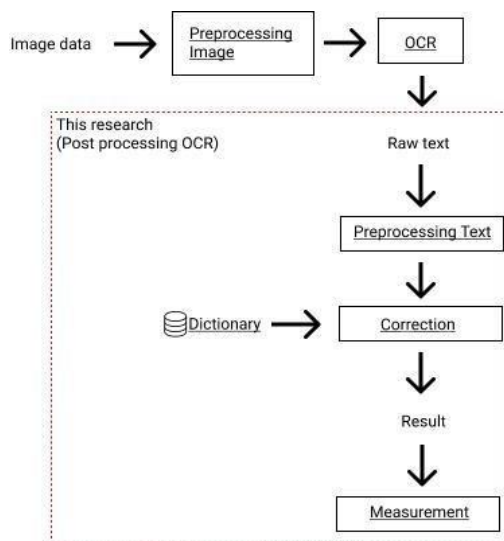


Figure 1. Methodology.

In this method, image data is entered, then the image will be processed, and when the image has been processed, it will be used as OCR (Optical Character Recognition), in post-processing, OCR data will be used as raw text, then pre-processing the text, in the correction section, dictionary data will be entered and used. to correct the pre-processing part, after the correction is complete then the results are created and then the data will be measured.

### 2.2. Dataset

In this study, the authors used image data that contained writing in it with various kinds of noise. The dataset is obtained from the public Kaggle dataset [8].

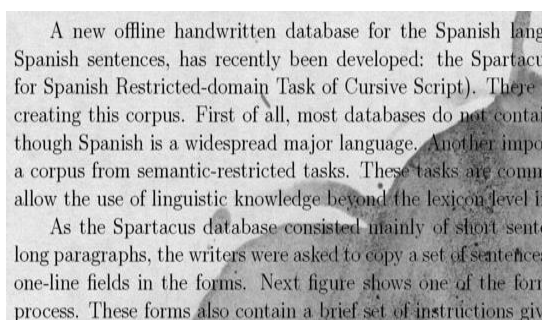


Figure 2. Source image.

### 2.3. Application & OCR library

This research uses javascript as a programming language and a string similarity node package (library) to make it easier. The image will be recognized (OCR) by javascript on the client-side. By using the library npm tesseract.js version 2.1.1[9] and using OEM.TESSERACT\_LSTM\_COMBINED as recognition mode [10].

## 2.4. Preprocessing Text

After some image is recognized, text from the result OCR is preprocessed [11] [12]

with:

1. Remove punctuation
2. Convert to lower case
3. Tokenization

## 2.5. Correction

There is a database of correct words, each word from the OCR results will be searched for the level of similarity with the words in the database with some of these algorithms. The term in the correct database with the highest level of similarity will be used to replace the phrase OCR result earlier[18].

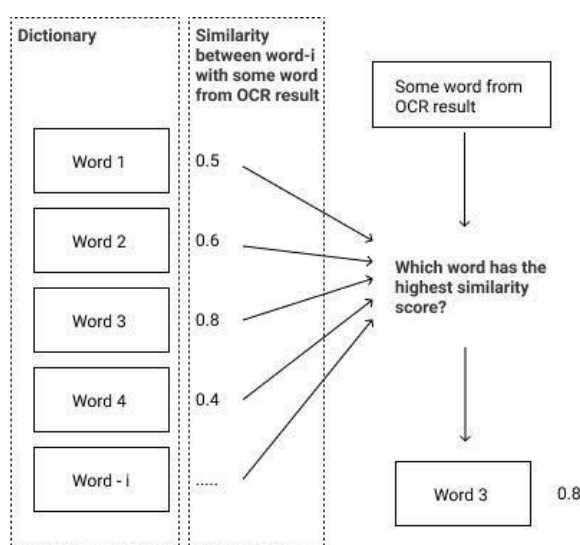


Figure 3. Correction

## 2.6. Algorithm

### Levenshtein Distance

$$\text{lev}(a, b) = \begin{cases} |a| & \text{if } |b| = 0, \\ |b| & \text{if } |a| = 0, \\ \text{lev}(\text{tail}(a), \text{tail}(b)) & \text{if } a[0] = b[0] \\ 1 + \min \begin{cases} \text{lev}(\text{tail}(a), b) \\ \text{lev}(a, \text{tail}(b)) \\ \text{lev}(\text{tail}(a), \text{tail}(b)) \end{cases} & \text{otherwise.} \end{cases} \quad (1)$$

This algorithm is named after the Soviet mathematician Vladimir Levenshtein, who considered this distance in 1965. This algorithm is based on edit distance, which counts how many steps a string can take to become another string (insertions, deletions, or substitutions) [13]. The author uses the NPM (Node Package Manager). The package name is js-levenshtein version 1.1.6.



Figure 4. Correction.

An example of the first word, namely Nachto, in this part of the sentence there is an incorrect word, so the character "a" is edited to "i" and the character "c" to "g", after editing the character "o" will be deleted. then the levenshtein distance is 3. That is 2 edits and 1 deletion.

### Hamming Distance

American mathematician Richard Hamming proposed this method in 1950 [14]. Hamming distance algorithm is a fast algorithm because it uses simple calculations. This research using node package hamming version 0.0.2. Example :

String 1: "abcd"

String 2: "dbek"

Table 1. Hamming Distance Illustration

Letter position	0	1	2	3
String 1	a	b	c	d
String 2	d	b	e	k
Is Equal?	No	Yes	No	No

Count the "No" result from the table, so the hamming distance of that example is 3.

### Jaro Winkler

The Jaro – Winkler distance is based on Jaro distance. This algorithm is reinvented by William E. Winkler in 1990.[15] Also this algorithm performing better measurement of lexical similarity and suitable for short string.[16]

The jaro distance  $d_j$  of two given strings  $s_1$  and  $s_2$  is

$$d_j = \frac{1}{3} \left( \frac{m}{|s_1|} + \frac{m}{|s_2|} + \frac{m-t}{m} \right) \quad (2)$$

Where:

- $d_j$  is result distance
- $s_i$  is the length of the string  $s_i$
- $m$  is the number of matching characters in both  $s_1$  and  $s_2$
- $t$  is half the number of transpositions (compare the  $i$ -th character of  $s_1$  and the  $i$ -th character of  $s_2$  divided by 2)

And the Jaro Winkler its self uses a prefix scale  $p$  which gives more favorable ratings to strings that match from the beginning for a set prefix length  $l$ . Given two strings  $s_1$  and  $s_2$ , their Jaro–Winkler similarity  $\text{sim}_w$  is:

$$sim_w = sim_j + lp(1 - sim_j) \quad (3)$$

Where:

- $sim_j$  is the Jaro similarity result for strings  $s_1$  and  $s_2$ .
- $l$  is the length of the common prefix at the start of the string. The maximum length is 4 characters.
- $p$  is a constant scaling factor for how much the score is adjusted upwards for having common prefixes. The default value for this constant is  $p = 0.1$ .

then jaro – winkler distance  $d_w$  is defined as  $d_w = 1 - sim_w$ . Using the node package jaro-winkler version 0.2.8.

### Sørensen–Dice coefficient

Sørensen – Dice index is a method used to compare the level of similarity of two strings. This method was published by Sørensen and Lee Raymond Dice in 1948 and 1945, respectively. This formula is similar to the Jaccard formula. However, the difference lies in the presence of a double match in Dice’s coefficient. The following is Dice’s coefficient formula:

$$dice(A, B) = \frac{2|A \cap B|}{|A| + |B|} \quad (4)$$

where  $|X|$  ( $|Y|$ , resp.) is the cardinality of the set  $X$  ( $Y$ , resp.), i.e., the number of elements in the set. That is, the Sørensen-Dice index is equal to twice the ratio of the number of elements appearing in both sets to the sum of the number of elements in each set. We remark that in the context the sets will be instantiated by entities in SAO structure, and the elements will be instantiated by terms or words accordingly where  $|X|$  ( $|Y|$ , resp.) is the cardinality of the set  $X$  ( $Y$ , resp.), i.e., the number of elements in the set. That is, the Sørensen-Dice index is equal to twice the ratio of the number of elements appearing in both sets to the sum of the number of elements in each set. We remark that in the context the sets will be instantiated by entities in SAO structure, and the elements will be instantiated by terms or words accordingly This research using the node package dice-coefficient version 2.0.0. In this library, the input string is converted to bigram. Example :

"somestring"

Become

"so", "me", "st", "ri", "ng"

Then the bigram output is used as input of the Sorensen – dice algorithm.

### 3. RESULTS AND DISCUSSION

---

We compared the results before and after post-processing. The evaluation is carried out based on the Precision, Recall, F-measure, and processing speed. Precision is part of the text that is right on top of all the text detected in the image. The recall is a portion of the text which corrects all the text in the data set. F-measure is a combination of recall and precision. Measurement of F-Measure / F1 Score is used because it is to overcome the uneven distribution of data.

$$Precision = \frac{TP}{(TP+FP)} \quad (5)$$

$$Recall = \frac{TP}{(TP+FN)} \quad (6)$$

$$Fmeasure = \frac{2 Precision \cdot Recall}{Precision+Recall} \quad (7)$$

The time in here is the total time it takes for the algorithm to process comparing 337 string data. Time is in milliseconds. This study uses an AMD A4 CPU (4 Core) with 8 RAM and SSD. For software, we use Linux Xubuntu 20.04, Chrome Version 90.0.4430.212 (Official Build) (64-bit).

Different browsers will result in different speeds and accuracy because different browsers handle images and their color profiles differently. That means an image can be rendered differently depending on the browser. Sometimes using chrome are better than firefox and vice versa [17].

Table 2. Performance Before post-processing

Precision	Recall	F1
0.73	0.65	0.69

In table two are examples of using precision, recall, and also Fmeasure which uses speed examples in browsers. The precision that is owned is 0.73 then for recall it is 0.65 and for measurement it is 0.69.

Here is the comparison between the algorithm:

Table 3. Performance after post-processing

Algorithm	Precision	Recall	F1	Time (ms)
Levenshtein distance	0.83	0.85	0.84	156
Hamming distance	0.65	0.68	0.66	61
Jaro Winkler distance	0.84	0.85	0.85	173
Sørensen – Dice coefficient	0.88	0.88	0.88	256

In this table a comparison of the algorithms used to edit a sentence is based on precision, recall, distance, and also the time it takes for the algorithm to edit. In the performance table using the formula:

$$Precision = \frac{TP}{(TP + FP)}$$

$$Recall = \frac{TP}{(TP + FN)}$$

$$Fmeasure = \frac{2 Precision \cdot Recall}{Precision + Recall}$$

Description:

TP = True Positive(Correct result).

FP = False Positive(Unexpected result).

FN = False Negative(Missing result).

#### 4. CONCLUSION

---

Of the four algorithms that we compare, the Sørensen-Dice coefficient ranks first in the F1 – Score with 0.88 but takes longer than other algorithms. Levenshtein distance and Jaro Winkler have a close result, but Levenshtein is faster.

The fastest algorithm is Hamming distance because it only uses simple operations. But this algorithm has a drawback, namely that the length of the strings must be the same. This is something that cannot be applied if the string is the result of the OCR process. Sometimes the resulting word has various kinds of additional lengths (noise letter).

We suggest not to use hamming distance because it performs badly in this case. If you need a better result and not considering speed, use the Sorensen dice. If you need a fast algorithm with an F1 score that is not much different, use the Levenshtein algorithm.

## REFERENCES

- [1] "Levensthein distance as a post-process to improve the performance of OCR in written road signs." <https://ieeexplore.ieee.org/document/8280534> (accessed May 26, 2021).
- [2] R. Smith, "An Overview of the Tesseract OCR Engine," in Ninth International Conference on Document Analysis and Recognition (ICDAR 2007), Sep. 2007, vol. 2, pp. 629–633. doi: 10.1109/ICDAR.2007.4376991.
- [3] H. Hu, L. Zhang, and J. Wu, "Hamming distance based approximate similarity text search algorithm," in 2015 Seventh International Conference on Advanced Computational Intelligence (ICACI), Mar. 2015, pp. 1–6. doi: 10.1109/ICACI.2015.7184772.
- [4] K. Manaf, S. Pitara, B. Subaeki, R. Gunawan, Rodiah, and Bakhtiar, "Comparison of Carp Rabin Algorithm and Jaro-Winkler Distance to Determine The Equality of Sunda Languages," in 2019 IEEE 13th International Conference on Telecommunication Systems, Services, and Applications (TSSA), Oct. 2019, pp. 77–81. doi: 10.1109/TSSA48701.2019.8985470.
- [5] V. R. Chifu, I. Salomie, E. Şt. Chifu, B. Izabella, C. B. Pop, and M. Antal, "Cuckoo search algorithm for clustering food offers," in 2014 IEEE 10th International Conference on Intelligent Computer Communication and Processing (ICCP), Sep. 2014, pp. 17–22. doi: 10.1109/ICCP.2014.6936974.
- [6] E. Brajković and D. Vasić, "Tree and word embedding based sentence similarity for evaluation of good answers in intelligent tutoring system," in 2017 25th International Conference on Software, Telecommunications and Computer Networks (SoftCOM), Sep. 2017, pp. 1–5. doi: 10.23919/SOFTCOM.2017.8115592.
- [7] M. Pikies and J. Ali, "String similarity algorithms for a ticket classification system," in 2019 6th International Conference on Control, Decision and Information Technologies (CoDIT), Apr. 2019, pp. 36–41. doi: 10.1109/CoDIT.2019.8820497.
- [8] "Denoising Dirty Documents." <https://kaggle.com/c/denoising-dirty-documents> (accessed Jul. 17, 2021).
- [9] "tesseract.js," npm. <https://www.npmjs.com/package/tesseract.js> (accessed Jul. 18, 2021).
- [10] "tesseract.js/api.md at master · naptha/tesseract.js," GitHub. <https://github.com/naptha/tesseract.js> (accessed Jul. 23, 2021).
- [11] C. A. B. de Mello, A. L. I. de Oliveira, and W. P. dos Santos, Eds., Digital document analysis and processing. New York: Nova Science Publishers, 2012.
- [12] J. Mei, A. Islam, A. Moh'd, Y. Wu, and E. E. Milios, "MiBio: A dataset for OCR post-processing evaluation," Data Brief, vol. 21, pp. 251–255, Dec. 2018, doi: 10.1016/j.dib.2018.08.099.
- [13] S. Rani and J. Singh, "Enhancing Levenshtein's Edit Distance Algorithm for Evaluating Document Similarity," in Computing, Analytics and Networks, Singapore, 2018, pp. 72–80. doi: 10.1007/978-981-13-0755-3\_6.

- [14] R. W. Hamming, "Error detecting and error correcting codes," Bell Syst. Tech. J., vol. 29, no. 2, pp. 147–160, Apr. 1950, doi: 10.1002/j.1538-7305.1950.tb00463.x.
- [15] "Jaro–Winkler distance," Wikipedia. May 30, 2021. Accessed: Jul. 19, 2021. [Online]. Available:  
[https://en.wikipedia.org/w/index.php?title=Jaro%E2%80%93Winkler\\_distance&oldid=1025977252](https://en.wikipedia.org/w/index.php?title=Jaro%E2%80%93Winkler_distance&oldid=1025977252)
- [16] H. Gueddah, A. Yousfi, and M. Belkasmi, "The filtered combination of the weighted edit distance and the Jaro-Winkler distance to improve spellchecking Arabic texts," in 2015 IEEE/ACS 12th International Conference of Computer Systems and Applications (AICCSA), Nov. 2015, pp. 1–6. doi: 10.1109/AICCSA.2015.7507128.
- [17] "Image To Text Conversion With React And Tesseract.js (OCR)," Smashing Magazine. <https://www.smashingmagazine.com/2021/06/image-text-conversion-react-tesseract-js-ocr/> (accessed Jul. 19, 2021).
- [18] Penerapan OCR (Optical Character Recognition) Pada Sistem Akuisisi Dokumen Jabatan Fungsional Dosen - UMM Institutional Repository. (2020, July 24). UMM Institutional Repository. Retrieved December 26, 2022, from <https://eprints.umm.ac.id/63700/>