Filtering Noisy Text to Accelerate Classification

D. Shapiro[a,b,*], H. Qassoud[a,b], M. Bolic[a]

[a] *Department of Electrical Engineering, 161 Louis Pasteur, University of Ottawa, Ottawa, Canada, K1N 6N5*
[b] *Clockrr Inc., Ottawa, Ontario, Canada*

Abstract

A new text comparison algorithm called LazyJaroWinkler is described in this work. For the noisy text filtering problem considered, it outperforms state of the art approaches on execution time. The balance between execution time, precision and recall is application-specific. LazyJaroWinkler and many other approaches were compared using more than a million configurations, and the top performing hyperparameter configurations are reported here. To facilitate further study and analysis, a database model is made available to the research community, in addition to the source code and algorithm description. This work may interest engineers and data scientists developing classifiers for noisy text or accelerating slow neural networks by sanitizing raw input data. On 3000 samples of realistic data, Jaro-Winkler provided the highest recall ($\approx 65\%$) with a low execution time (1.35s $\pm$ 0.09, n=10), while LazyJaroWinkler provided the lowest execution time (0.67s $\pm$ 0.14, n=10) at the expense of lower recall ($\approx 11\%$).

Keywords: Fuzzy string matching, neural network input filtering, text similarity, Jaro-Winkler, hyperparameters, modeling, memoization
2010 MSC: 62M45, 68T05, 68T10, 68T45

[*]Corresponding author
Email addresses: `dshap092@eecs.uottawa.ca` (D. Shapiro), `hqass076@uottawa.ca` (H. Qassoud), `mbolic@eecs.uottawa.ca` (M. Bolic)

1. Introduction

To motivate this work, consider a character-level text classification task performed by a deep neural network on noisy text. How can this task be accelerated? A neural network classifier takes candidate text as input, and outputs a classification, where each class trained into the neural network is associated with a unique keyword. The text comparison algorithms described in this work can quickly discard candidate text prior to classification when the candidate text is not similar to any keywords known to the classifier. Design space exploration of these various text comparison algorithms was performed with the objective to discover hyperparameter configurations that result in low latency and high precision. In the process of carrying out this comparison, a new approach was devised as an extension of JaroWinkler [1].

Neural-network-based classification relies heavily on slow mathematical operations such as multiplication and division. Character-level keyword recognition in noisy text is one case where much of the data fed to a neural network classifier may be irrelevant, and processing the irrelevant text is computationally expensive. Consider, for example, a situation where noisy text is recognized by a Deep Neural Network (DNN) as one of several keywords, or as noise. The noisy text is the output of an Optical Character Recognition (OCR) system which has processed a computer screen image into text. Classification of words in the OCR system's output should only be performed on text that is likely to be classified with some degree of confidence. This discarding of candidate text with a filtering algorithm prior to the neural network classification saves time. However, how can this filtering algorithm know which parts of the OCR output are similar to keywords that the neural network was trained to recognize? Is that not the purpose of the DNN classifier? The process of discarding input data prior to reaching the DNN is analogous to a funnel, where the first classifier discards clearly irrelevant text, and the DNN carefully classifies the surviving text. To decide which components of the text to discard, this initial classifier should use some objective measure such as the similarity between the candidate text and the set of keywords that the DNN was trained to classify. The filter should minimize latency while maintaining high precision and recall. The trade-off between execution time, precision, and recall is made on a case by case basis depending on the application.

To motivate the development of the filter, consider the execution time for this real-time classification task without acceleration. As depicted in Figure 1, each task involves 3,000 text snippets to be classified into one of 300 classes, where 99.5% of the candidate input texts are not part of any class trained into the DNN. The expected latency for this task should be under 1 second on average. As a first experiment, a DNN was deployed with 784 inputs, 625 neurons in the first hidden layer, 625 neurons in the second hidden layer, and 5000 output neurons. The DNN was trained to recognize text containing spelling errors. 10 samples were collected for CPU and GPU execution time for this character-level text classification task. The mean execution time required to classify the 3,000 candidate text strings was 433.9 seconds on a CPU (1.80GHz Intel® Xeon®), and 333.1 seconds on a GRID K520 GPU. This task is accelerated in this work by filtering out candidate input text that is not likely to be classified accurately. As described in the following sections, this approach can be implemented to complete the task in under 1 second using only one computer, rather than a large GPU cluster.

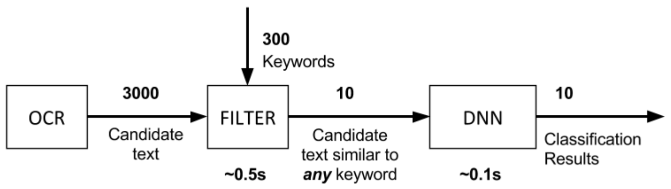

Figure 1: Text Classification Task.

The contribution of this work is to describe a new text filtering algorithm that is faster than previously reported work, and to provide a database model for evaluating text filtering approaches. The prior art is discussed in Section 2, followed by an overview of the various options for implementing this approach in Section 3. Next, performance experiments are detailed in Section 4. Section 5 contains a summary of this work and a discussion of future research directions.

2. Prior Art

The similarity of two text strings can be measured in terms of lexical or semantic similarity. Lexical similarity involves comparing sequences of characters to measure similarity, while semantic similarity involves studying a corpus to determine how similarly two words are used [2]. Hybrid measures are also possible [3]. For example, [4] mixed semantic text similarity extracted from a corpus of text with lexical similarity obtained from comparing character sequences to improve document comparison. Text similarity metrics are applied in many fields including, for example, spelling correction, where an autocorrect tool recommends words with low edit distance from a non-dictionary word [5], and plagiarism detection, where similar texts are flagged for review [6].

Matching noisy text to a list of keywords is also referred to as fuzzy string matching, string matching allowing errors, and approximate string matching [7]. The Levenshtein distance is a lexical approach to quantifying text similarity by finding the minimum edit distance between two text strings [8]. The idea is to count the minimum number of operations (insertions, deletions and substitutions) required to transform one string of text into the other. The Needleman-Wunsch distance (also known as Sellers Algorithm) adds to the Levenshtein distance the idea of a distance dependent substitution cost [9]. It searches for approximate substring matches. The Smith-Waterman distance is a metric for genetic sequence matching [10]. It finds the longest common sub-expression between two strings with a smaller penalty than the Levenshtein distance for prefixes and suffixes adjacent to the common substring [11].

Calculating an approximate edit distance rather than a true minimum edit distance can save execution time at the cost of accuracy [12]. Several approximate edit distance approaches such as [13], [14] and [15] provide better worst-case execution times than previous work due to improvements in estimating edit distance. The approach in these works was to reduce the per-iteration algorithmic complexity, emphasizing algorithmic improvements and utilizing low latency binary operations such as shifting, rather than slow mathematical operations such as multiplication, exponentiation and division.

The L1 distance (also called city block distance or block distance) is the sum of the absolute difference in value between words P and Q for each dimension d [16]. It measures the length of a line connecting P and Q drawn along one dimension at a time. The L2 distance (also called euclidean distance) is the square root of the sum of the absolute difference squared in each dimension [16]. The L2 distance is the length of the line segment $\overline{PQ}$ in multidimensional space. Cosine similarity computes the normalized dot product of P and Q [17].

Word embedding is a technique for encoding the relationships between words [18]. At the character level this means encoding letters as vectors, and words as sequences of vectors. This letter by letter encoding was used in this work to model the L1 and L2 distances. A similarity score is formed as the distance (e.g. L1 or L2) between points for each word in the constructed model.

$$Jaro(P,Q) = \frac{1}{3}(\frac{|\sigma|}{|P|} + \frac{|\sigma|}{|Q|} + \frac{|\sigma| - 0.5t}{|\sigma|}) \tag{1}$$

$$JaroWinkler(P,Q) = Jaro(P,Q) + |\rho| f(1 - Jaro(P,Q)) \tag{2}$$

$$ratio(P,Q) = \frac{|P| + |Q| - Levenshtein(P,Q)}{|P| + |Q|} \tag{3}$$

The Jaro similarity is another edit distance metric shown in Equation 1 [19] [20]. It considers four quantities: the length of both strings to be compared ($|P|$ and $|Q|$), the number of "matching" characters σ, and t defined as the number of transpositions (characters that match but are out of sequence) [11]. Calculating σ, matching is true if two characters are not more distant that half the length of the shorter string i.e. for P_i and Q_j, $|i - j| \leq \frac{min(|P|,|Q|)}{2}$ [11].

Jaro-Winkler extends the Jaro similarity with the concept of rewarding a common prefix shared between the two strings [1]. As in Equation 2, the longest common prefix ρ and scaling factor f are used to increase the similarity of strings sharing a common prefix [11].

In this work the ratio metric implementation from [21] is considered, where the metric shown in Equation 3 is followed. The sum of the length of both compared strings ($|p|+|q|$) less the edit distance (Levenshtein distance) is divided by the length of both strings ($|p| + |q|$).

Locality Sensitive Hashing (LSH) involves constructing a one way hash function which quickly reduces an input string to a fixed-length hash with the special property that similar strings generate hash outputs with low Hamming distance between them [22]. The idea behind these hashes is to reduce the dimensionality of the data while preserving the similarity features encoded into the data. Several LSH hashes such as SimHash [23] and MinHash [24] are studied in detail in the literature. Among many other applications, these hashes are used to build an index for comparing documents to discover approximate similarity [25].

Some distance measurement algorithms do not apply to the fuzzy string matching problem considered in this work. For example, comparison algorithms such as mean squared error and Hamming distance work well for computing image similarity for images with identical dimensions, but do not work well for string similarity computations as the strings are often of different length. Truncating one of the strings to perform the comparison on equal length strings causes unacceptable information loss. q-grams [26] also called n-grams [27] are small substring snippets of a fixed length created from each of the strings to be compared [11]. This string representation is an efficient data structure for computing token-based similarity. The Jaccard similarity takes into account string similarity and diversity of two tokenized strings [11]. Tokenization-based methods Jaccard similarity, Monge-Elkan similarity, and n-grams were not considered for this work.

Semantic text similarity measurement is an evolving field with ongoing metric quality problems for various applications such as constructing word embeddings [28]. Lexical similarity metrics also suffer from quality problems when comparing short strings. In such cases the compared strings have a low edit distance even though they are not related. For example, the word ‘a’ is only one substitution away from the word ‘I’. Therefore consider that this is an imperfect measure of text similarity. It does not account for context as, for example, a distributed word embedding does [18]. In fact, each of the string comparison algorithms presented in this work has some drawbacks and advantages based upon the constraints of the application and the data used to evaluate the algorithm’s performance. The performance results presented in this work are expected to vary under datasets that differ significantly from those described below. This reality recalls the “no free lunch theorem” for effective optimization algorithms, which states that “for any algorithm, any elevated performance over one class of problems is offset by performance over another class” [29].

Regarding precision and recall, let TN be the number of true negative classifications. Let TP be the number of true positive classifications. Let FN be the number of false negative classifications. Let FP be the number of false positive classifications. The equations for precision and recall from [30] are:

$$Precision(P) = TN/(TN + FP) \tag{4}$$

$$Recall(R) = TP/(TP + FN) \tag{5}$$

3. Text Filter Overview

This work is concerned with dropping text that is too dissimilar from anything trained into a DNN. The similarity between one text string and many keywords is therefore at the center of this problem. The algorithm (hereafter “the filter”) which filters the candidate input text (hereafter *ocr*) checks that the similarity between *ocr* and any keyword (hereafter *keyword*) is sufficiently high to justify DNN classification. Once the filter has identified that the *ocr* is similar to any *keyword*, it stops analyzing and proceeds to pass the text to be classified by the DNN. Generally, the vast majority of *ocr* input to the filter is dropped.

Tables 1 and 2 list the filter implementation approaches considered. The approaches listed in Table 1 were implemented in a similar way, shown in Algorithm 1, which loops through the *keywords* with each *ocr* looking for the first exact match (line 4) or sufficiently similar *keyword* (line 8). The function $FUNC$ configured by hyperparameters $PARAMS$ compares *keyword* and *ocr* where the result is then compared using operation OP in relation to the hyperparameter $LIMIT$. The $LIMIT$ defines the similarity threshold past which the filter should let an *ocr* pass to the DNN.

Consider an example of how Algorithm 1 is used in this work to implement a specific approach. The package [31] is an implementation of the edit distance algorithm described in [13]. Its minimum edit distance function *editdistance.eval* is used as $FUNC$ with no additional hyperparameters $PARAMS$. The resulting minimum edit distance must be less edits than some constant defined by hyperparameter $LIMIT$, and therefore OP is the operation $<=$.

The text filtering approaches in Table 1 contain hyperparameters that can take on a range of values, each affecting the execution time, precision, and recall of the implementation. Table 1 lists these hyperparameters and their value ranges for each algorithm.

The L1, L2, and substring approaches were implemented using variations of Algorithm 1. L1 and L2 implementations followed a character embedding approach for encoding text into feature vectors [36]. The model for encoding text was designed with a feature vector of length 100, and trained on the sequence of characters in each of the keywords. Word vectors were composed of the sum of the character vectors, divided by the number of characters in the word. The *keyword* vectors were stored in a dictionary for fast retrieval during string comparison.

The substring approach checks if any *keyword* is a substring of the input text *ocr*. The advantage of this approach is high speed and the ability to identify keywords in an *ocr* with characters added to the start or end of the *keyword*, while the disadvantage is a an inability to spot *ocr* with containing *keyword* text with one or more spelling errors. While Algorithm 1 requires at least one

Algorithm 1: A Generic Filtering Algorithm

```
Input: $keywords, ocrs$; Classification threshold $LIMIT$; 0 or more
       additional hyperparameters $PARAMS$; Function reference
       $FUNC$; Binary comparison operation in the set $\{<=, >=\}$ $OP$;
       OCR output text snippets $ocrs$
Output: List of text strings to be classified by a neural network
        $DNNinput$
1  ed($keywords, ocrs$):
2  for each $ocr \in ocrs$ do
3      for each $keyword \in keywords$ do
4          if $keyword == ocr$ then
5              $DNNinput.append(ocr)$
6              break
7          end
8          else if $OP(FUNC(keyword, ocr, PARAMS), LIMIT)$ then
9              $DNNinput.append(ocr)$
10             break
11         end
12     end
13 end
```

hyperparameter $LIMIT$, this substring approach contains no hyperparameters.

An LSH SimHash implementation that varies the length of the hash string is described in [35]. Conveniently, the comparison function compares one input text string (*ocr*) with all of the encoded text (*keywords*) with one function call to an index (*get_near_dups*()).

3.1. LazyJaroWinkler: A novel filter implementation

LazyJaroWinkler (Algorithm 2) is an extension of Jaro-Winkler. LazyJaroWinkler was optimized for speed using loop optimization, memoization, and low latency binary operations to compare strings and compute a similarity

Table 1: Hyperparameter configurations for approaches implemented using Algorithm 1

ID	**FUNC**	**FUNC Implementation**	**PARAMS (range, step size)**	**OP**	**LIMIT (range, step size)**
1	Edit Distance [13]	[31]	-	$\leq$	(0-10, 1)
2	Edit Distance [8]	[21]	-	$\leq$	(0-100, 1)
3	Ratio	[21]	-	$\geq$	(0-1, 0.1)
4	Jaro	[21]	-	$\geq$	(0-1, 0.1)
5	Jaro	[32]	-	$\geq$	(0-1, 0.1)
6	Jaro-Winkler	[21]	prefix_weight (0-1, 0.1)	$\geq$	(0-1, 0.1)
7	Jaro-Winkler	[32]	long_tolerance (0-1,1)	$\geq$	(0-1, 0.1)
8	Needleman-Wunsch	[33]	gap_cost (0.5-3,0.5)	$\geq$	(5-15, 1)
11	Cosine	[33]	-	$\geq$	(0.5-1,0.01)

Table 2: Hyperparameter configurations for approaches implemented using variations of Algorithm 1

ID	**Name**	**Implementation**	**PARAMS (range, step size)**	**LIMIT (range, step size)**
9	L1	[34]	windowSize(1-10,1)	(0.0001-0.1,0.01)
10	L2	[34]	windowSize(1-10,1)	(0.0001-0.1,0.01)
12	Substring	-	-	-
13	LazyJaroWinkler	Algorithm 2	swb (0-3,1), wshift(0-4,1), minsim(0-0.9,0.1), mindist(0-9,1)	(0-9,1)
14	SimHash	[35]	width(1-5,1), k (1-30,1)	-

score between *ocr* and *keywords*. The filter should improve execution time when an exact match is very unlikely. Therefore, checking for an exact match in Algorithm 2 is deeper into the loop structure. Under the most common condition when *ocr* is not similar to any *keyword*, the match failure should occur quickly in a low latency outer loop, rather than computing a more expensive metric such as edit distance during every iteration through the *keywords*. Dictionaries can be employed to store the results of common operations using memoization, further accelerating the most common case where *ocr* and *keyword* are not similar.

In Algorithm 2, for each candidate *ocr*, if it is an exact match for a keyword recognized by the DNN (*keyword*), then the candidate *ocr* is allowed to pass the filter and the loop skips to evaluate the next candidate *ocr* (lines 12 to 15). The loop optimization on lines 1 to 4 memorizes a binary-based frequency representation $binFreq()$ of each keyword known to the DNN and the number of high bits in that binary representation. These values are then available in all of the following loops without incurring a computation penalty. Loop optimization by precomputation is performed on lines 6, 7, 9, and 10. After removing non-alphanumeric characters, if the candidate *ocr* is more than *swb* characters shorter than a given *keyword*, or if it is 2^{wshift} times longer, then it likely will not match. Shorter texts are penalized much more heavily because information in the candidate *ocr* was lost in the OCR process, whereas longer strings can result from the merging of several words, which happens on occasion and does not imply a loss of information. The similarity between the candidate *ocr* and *keyword* is computed at line 16 using Algorithm 4. Algorithm 4 computes the number of bits in common between the $binFreq()$ representations of *keyword* and the candidate *ocr*, and then divides this by the number of high bits in the *keyword*. If the two are sufficiently similar, a second check measures how different the candidate *ocr* and *keyword* are at line 17. At this point in Algorithm 2 on line 18, the filtering algorithm has established that the candidate *ocr* is similar enough to *keyword* that it is worth spending the time on a relatively slow and more accurate comparison. A candidate text string *ocr* passes the filter if the minimum edit distance is less than $LIMIT$ (line 19).

$binFreq()$ (Algorithm 3) creates an integer for a word containing one 3-bit frequency bin for each of the 26 letters in the alphabet. Each bin inside the integer saturates after 3 occurrences of the corresponding letter within the word. For example, the $binFreq()$ binary representation of "a" is 001 while "aa" 011, and abab is 011 011. A word with many of the same letter quickly saturates the corresponding bin. For example, "baaaaab" becomes 011 111, the same as "baaaab" and "baaab". The number of bits in the XOR of two $binFreq()$ results gives a rough score for the distance between words (Algorithm 2 line 17). Also, the number of bits in the AND of two $binFreq()$ results gives a rough score for the similarity as it keeps only the bits in common (Algorithm 4 line 6). One drawback of this approach is that numbers and other characters are not represented in the letter frequency analysis, and so numerical strings are not seen as similar. For example, $binFreq('a123456')$ AND $binFreq('b123456')$ becomes 000 001 AND 001 000, which resolves to 0. One would expect these 2 similar strings to have a higher similarity score than 0. Testing with realistic data, increasing the number of bins per character to more than 3 did not significantly improve accuracy but did increase the execution time. Less than 3 bins decreased accuracy. Similarly representing all characters with their own bin had a negative impact on execution time while not significantly improving accuracy.

Regarding the design of Algorithm 2, it is important to consider the 5 hyperparameters that affect execution time as well as the precision and recall of the filter. The hyperparameters for Algorithm 2 are *minsim* (line 16), *mindist* (line 18), *swb* (set on line 7 and applied on line 12 to skip over *ocr* too short in length compared to the DNN *keyword*), $wshift$ (line 10 to skip over *ocr* too long in comparison to the DNN *keyword*), and $LIMIT$. $LIMIT$ performs the same threshold role as it does for Jaro-Winkler (IDs 6 and 7). A Jaro-Winkler score of 0 means that the 2 values are not similar, while larger values imply increased similarity. Therefore, a lower $LIMIT$ causes the algorithm to allow more results to pass the filter, possibly increasing recall, but likely decreasing precision.

As more results pass the filter and are processed by the DNN, execution time increases. *minsim* filters out *ocr* which have a large edit distance from the DNN *keywords*. It is compared against an approximation of the percentage of bits in common between two binary letter frequency encoded words. Increasing the similarity threshold causes the execution time of Algorithm 2 to decrease and the number of *ocr* allowed to pass the filter decreases. Also, increasing the similarity threshold generally tends to decrease recall while increasing precision. *mindist* is used to skip *ocr* that contain too many differences from the *keyword* in question. It is compared to a count of the number of bits that are different between the binary letter frequency encoded *ocr* and *keyword*. Increasing *mindist* causes the execution time to increase and the number of *ocr* allowed to pass the filter increases. Also, increasing *mindist* generally increases recall while decreasing the precision. Increasing *swb* allows *ocr* much shorter that the *keyword* and causes the execution time to increase while the number of *ocr* allowed to pass the filter increases. This is particularly problematic when *ocr* is much shorter than *keyword*, indicating that *ocr* is likely irrelevant text, and so increasing *swb* generally increases recall while decreasing precision. $wshift$ filters out *ocr* with a length much longer than the length of the keyword in question. Increasing $wshift$ allows longer and longer *ocr* to pass the filter, causing the execution time to increase. Also, increasing $wshift$ generally increases recall while decreasing the precision.

4. Performance Evaluation

In this section, design space exploration for the filter implementation is documented. The objective is to discover which algorithms and hyperparameter configurations result in low latency, pass simple tests, and result in high recall and precision. Four datasets were constructed that correspond to interesting extreme cases within the space of all possible inputs to the filter from the OCR software. These corner cases are sanity checks to make sure that the algorithm for each approach handles an easy to verify task correctly and in reasonable time. A fifth dataset containing real world data was created to model realistic conditions. These 5 datasets were used to evaluate the many hyperparameter settings for each of the approaches to implementing the filter. Dataset 1 contained 300 *keyword* and 3,000 *ocr*, with 0 *ocr* similar to any *keyword*. In this case the filter should reject all 3,000 *ocr*. Dataset 2 contained 300 *keyword* and 3,000 *ocr*, with each *ocr* randomly assigned a *keyword*. In this case the filter should accept all 3,000 *ocr*. Dataset 3 contained 300 *keyword* and 3,000 *ocr*, where all *ocr* randomly assigned a *keyword* with one added spelling error (e.g. 'NullPointerException' became 'NullPointerExc3ption'). In this case, the filter should accept all 3,000 *ocr*. Datasets 1, 2, and 3 were constructed such that *keyword* terms and *ocr* candidate text strings were 15 characters in length (lowercase letters or numbers), with the goal of avoiding the problem with editdistance mentioned earlier, where the algorithm replaces one string with the other when comparing two short strings. Allowing the text strings to contain numbers models realistic data more closely. Dataset 4 contained 300 *keyword* and 3,000 *ocr*, where all *ocr* and *keyword* were composed of 5 randomly selected lowercase letters. In this case the filter should reject all 3,000 *ocr*. Finally, Dataset 5 contained 299 *keyword* and 3,000 *ocr*, with 496 of the *ocr* similar to any *keyword*.

Each dataset was processed 10 times using each filter implementation in each of the hyperparameter configurations listed in Tables 1 and 2. However, NeedlemanWunsch (ID 8) is not listed, as high execution times such as 634 seconds per iteration rendered the algorithm too slow to be useful as a filter. Across the 5 datasets ≈1 million observations were recorded for LazyJaroWinkler.

Tables 3 and 4 present the hyperparameter configurations for each filter implementation, selecting the observations where the number of results was closest to 3,000 for datasets 1 and 2, and the number of results was closest to zero for datasets 3 and 4. These top results were then sorted by execution time, and the lowest execution time observation for dataset 5 was selected. Each observation involved executing the algorithm ten times for each scenario (Datasets 1, 2, 3, 4, and 5). Table 5 contains the analysis of the observations, reporting the precision and recall for each dataset. The observations were collected on 10 Virtual Machines (VMs) within a cloud computing infrastructure and using task to VM assignment based on the identifier of the VM. Each VM was configured with SSD storage, 512MB RAM, and a 2GHz 64-bit Xeon® CPU. Results were stored by the worker thread on each VM into a centralized database which could then be analyzed further using SQL commands. The complete dataset characterizing the solution space including the views extracting the best hyperparameter configurations is available for further study at [37].

Two broad groups emerge from the data: the first group with IDs {1, 2, 9, 10, 11, 12, 14} generated lower quality results, while the second group with IDs {3, 4, 5, 6, 7, 13} generated more promising results. In the first group, serious deficiencies in execution time, precision, and/or recall are immediately apparent. Specifically, for IDs 9, 10, 11, and 14 the average execution times for Dataset 5 were above 5 seconds (See Table 3). Word embedding was slow due to ad-hoc model generation each time a string list arrives from the OCR. Also, the word embedding approach does not tolerate one-off spelling mistakes and so required a large corpus that well-characterizes the words to be compared. For IDs 1 and 2 the precision for Dataset 4 was lower as a result of the weakness of the edit distance approach discussed in the prior art regarding short length strings (See Table 4). For ID 12 the recall on Dataset 2 was 0 as even small spelling errors were missed by the substring matching approach (See Table 5).

The second group scored well on execution time, precision, and accuracy for Datasets 1 to 4, and so they should be compared based upon their execution time, precision, and recall for the realistic Dataset 5. ID 13 (LazyJaroWinler) had the best execution time in the second group, while ID 7 (Jaro-Winkler) had the highest recall in the second group. LazyJaroWinler's aggressive approach in the outer loops resulted in the false rejection of a small amount of data from Dataset 2. Jaro-Winkler provided the highest recall (≈65%) with a low execution time (≈1.35s), while LazyJaroWinkler provided half the execution time (≈0.67s) at the expense of five sixths the recall (≈11%). By simply discarding randomly half of the input data, Jaro-Winkler can achieve a similar execution time to LazyJaroWinkler with a recall of 0% for the discarded half of the data and ≈65% for the second half, averaging to ≈33% recall. The advantage of LazyJaroWinkler over discarding data is that all data is processed and therefore the closest matches to the known classes of data are returned, whereas discarding data may miss strong matches. Randomly discarding samples from the input data mitigates this problem somewhat, as the classification task executes every second.

Algorithm 2: LazyJaroWinkler

```
Input: $LIMIT$; $swb$; $wshift$; $minsim$; $mindist$; Keywords that neural
       network was trained to classify are stored in $keywords.values()$,
       while the indexes of these entries are stored in $keywords.keys()$;
       OCR output text $ocrs$
Output: List of text strings to be classified by a neural network
        $DNNinput$
for each $keyword \in keywords.values()$ do
    $binDict[keyword] \leftarrow binFreq(keyword)$
    $freqDict[keyword] \leftarrow bin(binDict[keyword]).count(1)$
end
for each $ocr \in ocrs$ do
    $sw \leftarrow ocr.keepOnlyLetters()$
    $swLen \leftarrow swb + Legth(sw)$
    for each $keyword \in keywords.values()$ do
        $kLen \leftarrow Legth(keyword)$
        $kLenS \leftarrow kLen << wshift$
        if $NOT\ (kLen > swLen\ OR\ kLenS < swLen)$ then
            if $keyword == ocr$ then
                $DNNinput.append(ocr)$
                break
            end
            if $similarity(keyword, ocr) > minsim$ then
                $distance \leftarrow$
                 $binary(binDict[keyword] \oplus binFreq(ocr)).count(1)$
                if $distance <= mindist$ then
                    if $jaro\_winkler(keyword, ocr) >= LIMIT$ then
                        $DNNinput.append(ocr)$
                        break
                    end
                end
            end
        end
    end
end
```

Algorithm 3: Binary Letter Frequency Encoding. $0b111$, $0b011$, and $0b001$ are binary bit masks.

Input: Any text string for which a similarity will later be computed: *word*

Output: Integer representation of a binary field representing the frequency of each letter in *word*: *intermediate*

```
binFreq($word$):
    if $word \in binDict.keys()$ then
        return $binDict[word]$
    end
    $letters = \{ a, b, ..., z \}$
    $intermediate \leftarrow 0$, $iteration \leftarrow 0$, $bits \leftarrow 3$
    for each $letter \in letters$ do
        $count \leftarrow word.count(letter)$
        if $count >= 3$ then
            $intermediate \leftarrow intermediate|(0b111 << (iteration * bits))$
        else if $count == 2$ then
            $intermediate \leftarrow intermediate|(0b011 << (iteration * bits))$
        else if $count == 1$ then
            $intermediate \leftarrow intermediate|(0b001 << (iteration * bits))$
        $iteration \leftarrow iteration + 1$
    end
    $binDict[word] \leftarrow intermediate$
    return $intermediate$
```

Algorithm 4: Similarity. Computes the bit-level similarity between input text and a word from a neural network's lexicon.

```
Input:  A word in a neural network's lexicon: $keyword$; A substring
        from the OCR text $ocr$
Output: Similarity score between 0 and 1
similarity($keyword, ocr$):
    if $ocr \in binDict.keys()$ then
        $freqOCR \leftarrow binDict[ocr]$
    else
        $freqOCR \leftarrow binFreq(ocr)$
    $same \leftarrow (binDict[keyword]$ & $freqOCR)$
    if $same \in cntDict.keys()$ then
        $sameCount \leftarrow cntDict[same]$
    else
        $sameCount \leftarrow binary(same).count(1)$
        $cntDict[same] \leftarrow sameCount$
    return $sameCount/freqDict[keyword]$
```

Table 3: Mean ($\bar{x}$) and standard deviation (σ) of 10 execution time samples for each filter's most effective hyperparameter configuration.

		Dataset 1 (Exact matches) Exec. Time (s)		**Dataset 2** (One letter changed) Exec. Time (s)		**Dataset 3** (No matches) Exec. Time (s)		**Dataset 4** (No matches & short strings) Exec. Time (s)		**Dataset 5** (Screen capture) Exec. Time (s)	
ID	**Hyperparameters**	$\bar{x}$	σ	$\bar{x}$	σ	$\bar{x}$	σ	$\bar{x}$	σ	$\bar{x}$	σ
1	LIMIT=2	2.37	0.17	2.08	0.22	4.18	0.28	1.47	0.12	3.71	0.35
2	LIMIT=2	0.67	0.07	0.61	0.07	1.10	0.12	0.41	0.04	1.62	0.12
9	LIMIT=0.049; windowSize=1	0.05	0.01	9.87	1.45	11.83	1.67	13.08	3.03	11.71	1.11
10	LIMIT=0.007; windowSize=1	0.04	0.00	14.74	1.32	19.39	1.41	20.77	1.62	18.97	1.46
11	LIMIT=0.84	4.20	0.22	3.99	0.20	8.12	0.84	5.99	0.21	7.59	0.44
12	-	0.13	0.01	0.07	0.01	0.12	0.02	0.10	0.02	0.11	0.02
14	width=3; k=17	19.81	2.02	21.49	1.44	25.21	2.33	24.04	2.10	24.14	2.30
3	LIMIT=0.6	0.61	0.06	0.54	0.03	1.06	0.10	0.49	0.03	1.29	0.13
4	LIMIT=0.8	0.47	0.05	0.39	0.03	0.85	0.19	0.46	0.04	0.82	0.09
5	LIMIT=0.8	0.73	0.06	0.66	0.05	1.32	0.22	0.63	0.06	1.47	0.16
6	LIMIT=1; prefix_weight=0.2	0.47	0.05	0.46	0.15	0.83	0.12	0.48	0.04	0.81	0.09
7	LIMIT=0.8; long_tolerance=0	0.75	0.06	0.72	0.21	1.32	0.27	0.65	0.06	1.35	0.09
13	LIMIT=9; swb=1; wshift=1; minsim=0.7; mindist=6	0.79	0.09	0.70	0.06	1.55	0.26	1.27	0.12	0.67	0.14

5. Conclusions and Future Work

The contribution of this work was to describe LazyJaroWinkler, and to provide a model for evaluating text filtering approaches. LazyJaroWinkler is a new approach that breaks from loops as soon as a similarity match is identified. Memoization limits the overall execution time, and when there is no match, most computation involves low level binary operations on integers. The takeaway advice to the designer of a filter for a text classifier is to first select an algorithm having high recall and precision for extreme cases, and then to consider the trade-off between recall and execution time when selecting a particular algorithm.

Future research directions for this initial work are to investigate further the applications of LazyJaroWinkler as compared to Jaro and Jaro-Winkler. Perhaps there are additional heuristics that can achieve the low execution time and high precision of LazyJaroWinkler without the degradation in recall reported in this work. Finally, this work provides a reusable starting point for additional simulation-based quantitative analysis of string comparison approaches.

References

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

[2] W. H. Gomaa, A. A. Fahmy, A survey of text similarity approaches, International Journal of Computer Applications 68 (13).

[3] W. Cohen, P. Ravikumar, S. Fienberg, A comparison of string metrics for matching names and records, in: Kdd workshop on data cleaning and object consolidation, Vol. 3, 2003, pp. 73–78.

[4] A. Islam, D. Inkpen, Semantic text similarity using corpus-based word similarity and string similarity, ACM Transactions on Knowledge Discovery from Data (TKDD) 2 (2) (2008) 10.

Table 4: Mean ($\bar{x}$) and standard deviation (σ) for 10 samles counting the number of *ocr* that pass the filter for each filter's most effective hyperparameter configuration. The the same hyperparameter settings were used in Table 3.

	Dataset 1 (Exact matches)		**Dataset 2** (One letter changed)		**Dataset 3** (No matches)		**Dataset 4** (No matches & short strings)		**Dataset 5** (Screen capture)	
ID	$\bar{x}$	σ	$\bar{x}$	σ	$\bar{x}$	σ	$\bar{x}$	σ	$\bar{x}$	σ
1	3000.00	0.00	3000.00	0.00	0.00	0.00	465.90	21.47	65.00	0.00
2	3000.00	0.00	3000.00	0.00	0.00	0.00	465.90	21.47	65.00	0.00
9	3000.00	0.00	1551.50	326.21	490.80	98.62	27.30	7.32	331.20	68.05
10	3000.00	0.00	1661.70	348.77	390.50	44.12	13.90	8.56	329.60	11.82
11	3000.00	0.00	2997.00	6.75	2.40	1.65	154.20	11.68	237.00	0.00
12	3000.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	805.00	0.00
14	3000.00	0.00	2086.60	84.23	104.00	9.31	122.20	11.04	534.00	0.00
3	3000.00	0.00	3000.00	0.00	0.00	0.00	44.10	5.22	359.00	0.00
4	3000.00	0.00	3000.00	0.00	0.00	0.00	43.80	4.59	238.00	0.00
5	3000.00	0.00	3000.00	0.00	0.00	0.00	43.40	5.17	251.00	0.00
6	3000.00	0.00	3000.00	0.00	0.00	0.00	0.00	0.00	232.00	0.00
7	3000.00	0.00	3000.00	0.00	0.00	0.00	105.20	7.81	378.00	0.00
13	3000.00	0.00	2995.00	12.69	0.00	0.00	0.00	0.00	62.00	0.00

Table 5: Analysis of observations: Average execution time, precision, and recall were calculated and are presented here. Precision and accuracy was 100% for all filter implementations for Dataset 1 and so those results are omitted.

	Dataset 2		**Dataset 3**		**Dataset 4**		**Dataset 5**	
ID	**Prec. (%)**	**Rec. (%)**	**Prec. (%)**	**Rec. (%)**	**Prec. (%)**	**Rec. (%)**	**Prec. (%)**	**Rec. (%)**
1	100.00%	100.00%	100.00%	N/A	84.47%	N/A	99.76%	11.90%
2	100.00%	100.00%	100.00%	N/A	84.47%	N/A	99.76%	11.90%
9	100.00%	51.72%	83.64%	N/A	99.09%	N/A	95.77%	47.98%
10	100.00%	55.39%	86.98%	N/A	99.54%	N/A	96.61%	44.96%
11	100.00%	99.90%	99.92%	N/A	94.86%	N/A	97.48%	35.08%
12	0.00%	0.00%	100.00%	N/A	100.00%	N/A	100.00%	89.31%
14	100.00%	69.55%	96.53%	N/A	95.93%	N/A	93.49%	74.80%
3	100.00%	100.00%	100.00%	N/A	98.53%	N/A	97.72%	63.71%
4	100.00%	100.00%	100.00%	N/A	98.54%	N/A	99.20%	43.95%
5	100.00%	100.00%	100.00%	N/A	98.55%	N/A	99.16%	45.77%
6	100.00%	100.00%	100.00%	N/A	100.00%	N/A	98.04%	36.90%
7	100.00%	100.00%	100.00%	N/A	96.49%	N/A	98.08%	65.52%
13	100.00%	99.83%	100.00%	N/A	100.00%	N/A	99.76%	11.29%

[5] E. Brill, R. C. Moore, An improved error model for noisy channel spelling correction, in: Proceedings of the 38th Annual Meeting on Association for Computational Linguistics, Association for Computational Linguistics, 2000, pp. 286–293.

[6] Z. Su, B. R. Ahn, K. Y. Eom, M. K. Kang, J. P. Kim, M. K. Kim, Plagiarism detection using the levenshtein distance and smith-waterman algorithm, in: Innovative Computing Information and Control, 2008. ICICIC '08. 3rd International Conference on, 2008, pp. 569–569. doi:10.1109/ICICIC.2008.422.

[7] G. Navarro, A guided tour to approximate string matching, ACM computing surveys (CSUR) 33 (1) (2001) 31–88.

[8] V. Levenshtein, Binary codes capable of correcting deletions, insertions, and reversals, in: Soviet Physics-Doklady, Vol. 10, 1966.

[9] S. B. Needleman, C. D. Wunsch, A general method applicable to the search for similarities in the amino acid sequence of two proteins, Journal of molecular biology 48 (3) (1970) 443–453.

[10] T. F. Smith, M. S. Waterman, Identification of common molecular subsequences, Journal of molecular biology 147 (1) (1981) 195–197.

[11] F. Naumann, M. Herschel, An introduction to duplicate detection, Synthesis Lectures on Data Management 2 (1) (2010) 1–87.

[12] E. Ukkonen, Algorithms for approximate string matching, Information and control 64 (1-3) (1985) 100–118.

[13] H. Hyyrö, Explaining and extending the bit-parallel approximate string matching algorithm of myers, Technical Report A-2001-10, Department of Computer and Information Sciences, University of Tampere, Tampere, Finland (2001).

[14] A. Andoni, R. Krauthgamer, K. Onak, Polylogarithmic approximation for edit distance and the asymmetric query complexity, in: Foundations of Computer Science (FOCS), 2010 51st Annual IEEE Symposium on, IEEE, 2010, pp. 377–386.

[15] G. Li, D. Deng, J. Wang, J. Feng, Pass-join: A partition-based method for similarity joins, Proceedings of the VLDB Endowment 5 (3) (2011) 253–264.

[16] S.-H. Cha, Comprehensive survey on distance/similarity measures between probability density functions, International Journal of Mathematical Models And Methods In Applied Sciences 1 (4) (2007) 301–307.

[17] V. Q. Malic, Analyzing historians' perspectives using semantic measures, IConference 2016 Proceedings.

[18] J. Turian, L. Ratinov, Y. Bengio, Word representations: a simple and general method for semi-supervised learning, in: Proceedings of the 48th annual meeting of the association for computational linguistics, Association for Computational Linguistics, 2010, pp. 384–394.

[19] M. A. Jaro, Probabilistic linkage of large public health data files, Statistics in medicine 14 (5-7) (1995) 491–498.

[20] M. A. Jaro, Advances in record-linkage methodology as applied to matching the 1985 census of tampa, florida, Journal of the American Statistical Association 84 (406) (1989) 414–420.

[21] D. Necas, E. Mtt, M. Ohtamaa, et al., python-Levenshtein: The levenshtein python c extension module contains functions for fast computation [Online; accessed 2016-09-01] (2010–).
URL `https://github.com/ztane/python-Levenshtein`

[22] P. Indyk, R. Motwani, Approximate nearest neighbors: Towards removing the curse of dimensionality, in: Proceedings of the Thirtieth Annual ACM Symposium on Theory of

Computing, STOC '98, ACM, New York, NY, USA, 1998, pp. 604–613. doi:10.1145/276698.276876.
URL http://doi.acm.org/10.1145/276698.276876

[23] M. S. Charikar, Similarity estimation techniques from rounding algorithms, in: Proceedings of the thiry-fourth annual ACM symposium on Theory of computing, ACM, 2002, pp. 380–388.

[24] A. Shrivastava, P. Li, In defense of minhash over simhash., in: AISTATS, 2014, pp. 886–894.

[25] G. S. Manku, A. Jain, A. Das Sarma, Detecting near-duplicates for web crawling, in: Proceedings of the 16th international conference on World Wide Web, ACM, 2007, pp. 141–150.

[26] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, L. Pietarinen, D. Srivastava, Using q-grams in a dbms for approximate string processing, IEEE Data Eng. Bull. 24 (4) (2001) 28–34.

[27] G. Kondrak, N-gram similarity and distance, in: Proceedings of the 12th International Conference on String Processing and Information Retrieval, SPIRE'05, Springer-Verlag, Berlin, Heidelberg, 2005, pp. 115–126. doi:10.1007/11575832_13.
URL http://dx.doi.org/10.1007/11575832_13

[28] M. Faruqui, Y. Tsvetkov, P. Rastogi, C. Dyer, Problems with evaluation of word embeddings using word similarity tasks, arXiv preprint arXiv:1605.02276.

[29] D. H. Wolpert, W. G. Macready, No free lunch theorems for optimization, IEEE transactions on evolutionary computation 1 (1) (1997) 67–82.

[30] C. Sammut, G. I. Webb, Encyclopedia of machine learning, Springer Science & Business Media, 2011.

[31] H. Tanaka, editdistance: 0.3.1, https://pypi.python.org/pypi/editdistance (2016).

[32] J. Turk, M. Stephens, et al., jellyfish: a library for doing approximate and phonetic matching of strings., [Online; accessed 2016-09-01] (2015–).
URL https://github.com/jamesturk/jellyfish

[33] A. H. Hitawala, P. Konda, py-stringmatching, [Online; accessed 2016-11-03] (2016).
URL https://pypi.python.org/pypi/py_stringmatching

[34] E. Jones, T. Oliphant, P. Peterson, et al., SciPy: Open source scientific tools for Python, [Online; accessed 2016-09-01] (2001–).
URL http://www.scipy.org/

[35] Leon, simhash 1.7.0, [Online; accessed 2016-11-07] (2016).
URL https://pypi.python.org/pypi/simhash/1.7.0

[36] R. Řehůřek, P. Sojka, Software Framework for Topic Modelling with Large Corpora, in: Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks, ELRA, Valletta, Malta, 2010, pp. 45–50, http://is.muni.cz/publication/884893/en.

[37] D. Shapiro, H. Qassoud, M. Lemay, M. Bolic, Neural network text filtering, [Online; accessed 2016-11-03] (2016).
URL https://github.com/hamzaqassoud/Neural_Network_Filtering