

# Smash: A Dictionary-Free String Distance Metric that Considers Acronyms, Abbreviations, and Typos

*Joshua Wu  
Aditya Parameswaran, Ed.  
Dixin Tang, Ed.*



Electrical Engineering and Computer Sciences  
University of California, Berkeley

Technical Report No. UCB/EECS-2023-167

<http://www2.eecs.berkeley.edu/Pubs/TechRpts/2023/EECS-2023-167.html>

May 12, 2023

Copyright © 2023, by the author(s).  
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

### Acknowledgement

I would like to thank my advisor Aditya Parameswaran for his guidance during my candidacy. He is a thoroughly knowledgeable and caring instructor, and I could not have asked for a better advisor for my time here. I would also like to thank Dr. Dixin Tang, who has supported me throughout my candidacy. I am grateful to have learned much about the research process and received much support from him.

I would like to thank Nithin Chalapathi for his assistance with the project and support throughout my candidacy.

I would lastly like to acknowledge the NACDL and Big Local News at Stanford for kickstarting this project. In particular, I would like to thank Julie Ciccolini, Tristan Chambers, Lisa Pickoff-White, and Cheryl Philips for their involvement in and contributions to the project.

---

# SMASH: A Dictionary-Free String Distance Metric that Considers Acronyms, Abbreviations, and Typos

Joshua Wu

---

## Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

### Committee



---


Aditya Parameswaran  
Research Advisor

05/12/2023

---

(Date)

★ ★ ★ ★ ★ ★ ★



---

Dixin Tang  
Second Reader

05/12/2023

---

(Date)

SMASH: A Dictionary-Free String Distance Metric that Considers Acronyms,  
Abbreviations, and Typos

by

Joshua Wu

A thesis submitted in partial satisfaction of the

requirements for the degree of

Master of Science

in

Electrical Engineering and Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Aditya Parameswaran, Chair  
Dr. Dixin Tang

Spring 2023

SMASH: A Dictionary-Free String Distance Metric that Considers Acronyms,  
Abbreviations, and Typos

Copyright 2023  
by  
Joshua Wu

## Abstract

SMASH: A Dictionary-Free String Distance Metric that Considers Acronyms,  
Abbreviations, and Typos

by

Joshua Wu

Master of Science in Electrical Engineering and Computer Science

University of California, Berkeley

Professor Aditya Parameswaran, Chair

String matching is the operation of identifying and matching similar strings according to a similarity function or distance metric. It is a historically difficult problem that has many real world applications such as data cleaning, entity deduplication, and search. Traditional string distance metrics only consider either acronyms, abbreviations, or typos, but do not consider all three together. Prior research has attempted to address string matching with acronyms and abbreviations by leveraging a dictionary of generated synonyms. However, this approach is limited because it must trade off between potentially leaving incorrect rules in the dictionary or potentially removing correct rules during a refinement procedure.

We propose SMASH, a new character-based string distance metric for applications in string matching. We also propose an efficient dynamic programming algorithm that leverages SMASH and captures abbreviations, acronyms, and misspellings, three of the most common data modifications in real world data. We evaluate our metric on real world datasets to show that our metric's accuracy outperforms that of state-of-the-art string distance metrics and similarity measures. In particular, we evaluate SMASH on police roster data from the NACDL (National Association of Criminal Defense Lawyers) and show that it achieves the highest F-score among different metrics on a police roster dataset as well as on other benchmark datasets. We also show that SMASH is practical to integrate into a data cleaning workflow.

# Contents

<b>Contents</b>	<b>i</b>
<b>List of Figures</b>	<b>iii</b>
<b>List of Tables</b>	<b>iv</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Data Cleaning for NACDL Database Ingestion . . . . .	1
1.2 The Problematic Case: Title Matching . . . . .	3
1.3 Our Approach . . . . .	5
<b>2 Related Work</b>	<b>7</b>
<b>3 String Metrics</b>	<b>10</b>
3.1 Problem Formulation . . . . .	10
3.2 Our Similarity Metric . . . . .	11
<b>4 The Algorithm</b>	<b>13</b>
4.1 Proof of Correctness . . . . .	15
4.2 Complexity Analysis . . . . .	17
4.3 Stop Words . . . . .	18
<b>5 Integration with OpenRefine</b>	<b>19</b>
5.1 Implementation . . . . .	19
5.2 Demo . . . . .	19
<b>6 Experiments and Results</b>	<b>24</b>
6.1 Experimental Settings . . . . .	24
6.2 Evaluation Against Baselines . . . . .	26
6.3 Evaluation Against State-of-the-Art . . . . .	27
6.4 Further Experiments . . . . .	30
<b>7 Future Work</b>	<b>36</b>

<b>8 Conclusion</b>	<b>37</b>
<b>Bibliography</b>	<b>38</b>



# List of Figures

1.1	Interface for our cleaner app . . . . .	2
1.2	Sample warnings from our cleaner app . . . . .	2
1.3	Some groups of equivalent police titles that get matched through running our workflow . . . . .	4
1.4	Motivating examples for designing our metric . . . . .	5
2.1	An example that showcases the affine gap distance . . . . .	7
2.2	A subset of rewriting rules generated by <code>pkduck</code> [15] for police officer title data . . . . .	8
3.1	The intuition for capturing acronyms, abbreviations, and typos between two strings . . . . .	11
4.1	Visualization of three iterations of the step-by-step process in the loop of the SMASH algorithm. . . . .	14
4.2	Continuation of previous visualization: cached results are used when $k$ advances. . . . .	15
5.1	Found clusters using the key collision method and fingerprint as the keying function . . . . .	20
5.2	Found clusters using nearest neighbor clustering with Levenshtein distance . . . . .	21
5.3	Found clusters using nearest neighbor clustering with SMASH distance . . . . .	22
5.4	Text facet interface in OpenRefine . . . . .	23
6.1	Sample longform and shortform string pairs in <code>LARGE DISEASE</code> . . . . .	25
6.2	Sample similar string pairs in <code>SMALL DISEASE</code> taken from the <code>pkduck</code> paper . . . . .	25
6.3	Sample similar string pairs in <code>LOCATION</code> taken from the <code>pkduck</code> paper . . . . .	25
6.4	Comparison of SMASH, Jaccard, Levenshtein, and Affine Gap using the capture counts metric defined earlier on the large disease dataset . . . . .	27
6.5	Runtime comparison with sampled rows from the large disease dataset . . . . .	28
6.6	Runtime comparison across word length with sampled pairs from the large disease dataset . . . . .	29
6.7	Most successful example of ChatGPT string matching . . . . .	32
6.8	ChatGPT only outputs a few correct pairs . . . . .	32
6.9	ChatGPT avoids the task . . . . .	33
6.10	ChatGPT outputs something completely untrue . . . . .	33

# List of Tables

6.1	Comparison of SMASH against pkduck on the SMALL DISEASE and LOCATION datasets . . . . .	29
6.2	Comparison of pkduck refiner on POLICE ROSTER . . . . .	30
6.3	Final results . . . . .	34
6.4	Final results, varied across threshold . . . . .	35

## Acknowledgments

I would like to thank my advisor Aditya Parameswaran for his guidance during my candidacy. He is a thoroughly knowledgeable and caring instructor, and I could not have asked for a better advisor for my time here. I am extremely grateful for his willingness to give me this opportunity.

I would also like to thank Dr. Dixin Tang, who has supported me throughout my candidacy. I am grateful to have learned much about the research process and received much support from him.

I would like to thank Nithin Chalapathi for his assistance with the project and support throughout my candidacy.

I would lastly like to acknowledge the NACDL and Big Local News at Stanford for kick-starting this project and increasing the accountability of government institutions by making it easier to access public data. In particular, I would like to thank Julie Ciccolini, Tristan Chambers, Lisa Pickoff-White, and Cheryl Philips for their involvement in and contributions to the project.

# Chapter 1

## Introduction

String data is ubiquitous in datasets across domains and often contains problematic inconsistencies due to human oversight. In this project, we worked with public defender datasets from the National Association of Criminal Defense Lawyers' for the purpose of cleaning the data before it is ingested into their database of police misconduct. With police officer data aggregated from sources nationwide, the wide range of sources alongside manual mistakes in data entry results in many unwanted modifications to the string data to be ingested into the database. For example, perhaps there are extra spaces in a value ("Sheriff's Office" vs. " Sheriff's Office "), or the value is missing apostrophes where they should have them ("Sheriff's Office" vs. "Sheriff S Office"). In either of these cases, ingesting the data into the database would be actively harmful because if a user makes a selection on "Sheriff's Office" or joins on the appropriate column, these other cases would not be included in the result even though they are referring to the same entity.

Here, we will first discuss common problems the NACDL faced and potential solutions we implemented. Then, we will turn our attention to focus on one specific difficult problem of string matching.

### 1.1 Data Cleaning for NACDL Database Ingestion

The National Association of Criminal Defense Lawyers hosts the Full Disclosure Project, a public database housing information for public defenders. They aim to make police information more transparent, allowing lawyers to easily get access to any history of misconduct for a particular officer. However, because of the various errors in their data, much manual labor has to be expended to comb through and clean the data before it can be added to their database. The NACDL approached us to help build tools that can help reduce the manual labor required. They handed us a list of problems, and we were very quickly able to create a simple application, seen in Figures 1.1 and 1.2, that addresses these issues one by one.

The following is a summary of the issues they have brought forward to us and our approach to reduce the manual effort required in ingesting new data.

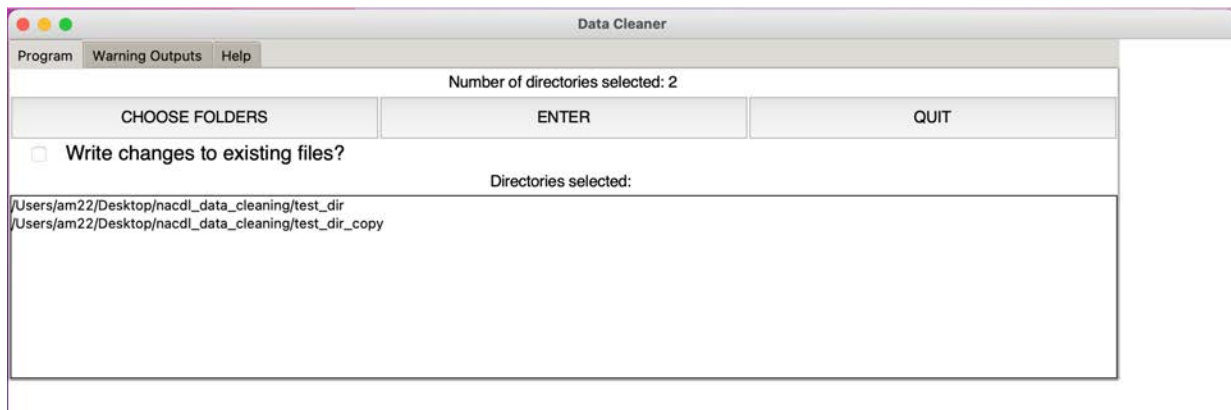


Figure 1.1: Interface for our cleaner app

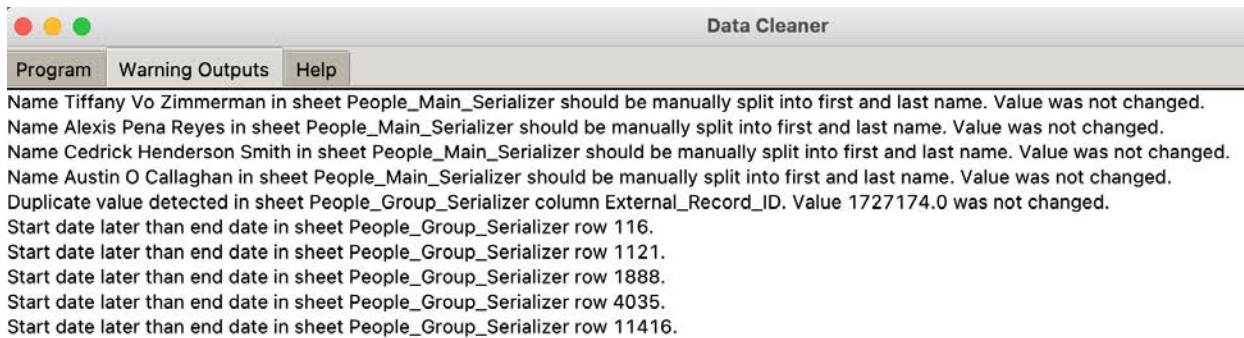


Figure 1.2: Sample warnings from our cleaner app

- **Extraneous whitespace between words and around words:** Automatically detect and remove whitespace, modifying values.
- **Missing apostrophes:** Detect potential cases and output a warning with the sheet name and value, does not modify values. For example, “Sheriff S Office” is likely an incorrectly parsed version of “Sheriff’s Office.”
- **Invalid phone numbers:** Detect invalid phone numbers with a regex and output a warning, does not modify values.
- **Invalid emails:** Detect invalid emails with a regex and output a warning, does not modify values.

- **Certain columns need values to be distinct:** Run a check on those columns and output warnings, does not modify values. For example, we would not want there to be multiple records that have the same id.
- **Alias columns have extraneous values:** Some records have both a name field (e.g. “John Smith”) as well as an alias field, which is a list of aliases for the same entity (e.g. [“Johnny”, “Mr. Smith”]). For each record, we make sure that the alias field does not contain the main name nor any duplicate aliases that are just in different case, removing these values if they exist. In the previous example, “john smith” and “mr. smith” would be removed from the list if they were also present.
- **Relation’s own *external\_id* present in fields where they do not belong:** Remove a record’s own *external\_id* from columns such as *belongs\_to*. For example, if a police officer record’s *external\_id* field is 0 and *reports\_to* field is [0, 71, 53], then we would want to remove 0 from the *reports\_to* field because it does not make sense to report to oneself.
- **Invalid *external\_ids*:** Check that each referenced *external\_id* exists, and output a warning if it does not.
- **Name field exists but not first or last name:** Automatically create first name and last name columns, populating the values based on the name column; output warning and leave cell blank if the name field does not contain a two-word string.
- **Invalid dates:** Verifies that dates are integers with  $1 \leq month \leq 12$  and days are valid for the given month; ensures that the *start\_date* field for a record is less than or equal to the *end\_date* field.

However, we soon realized that there was one problem that required the most amount of human effort and was going to be the most difficult to solve: title matching. The rest of this report will focus on addressing the issue of title matching, and more broadly, approximate string matching.

## 1.2 The Problematic Case: Title Matching

The wide range of data sources from which the dataset was constructed alongside manual mistakes in data entry resulted in multiple string variations for the same officer title. The dataset may identify one officer as having rank “Lieutenant” and another having rank “Lt.” In reality, both strings represent the same rank but one string has been abbreviated. This is just one example of many in which the same entity is being represented by multiple different strings because of data inconsistency.

The primary problem we tackle is cleaning a dataset with such inconsistencies. The traditional approach is to have an expert expend many work days to create a dictionary of

1	Motor Carrier Inspector III Motor Carrier Inspector mci1 mci3	3	Deputy Marshall Dmrsl Dpty Mrsl
2	Assistant Park Manager apkmgr aprmngr atpm	4	Special Agent in Charge Sag ic sac

Figure 1.3: Some groups of equivalent police titles that get matched through running our workflow

rules establishing equivalency between a string and its modified form, for example, having a mapping from “Lt.” to “Lieutenant.” This process often takes days, according to the person working with the data at a specific midwestern public defender office we worked with. To automate part of the data cleaning process, we can look to string matching or approximate string joins.

Approximate string join (ASJ) is an operation that identifies and matches similar strings and is adopted in a variety of applications, such as data cleaning and integration, record linkage, information retrieval, and recommendation systems [5, 2, 9, 21, 11]. In data cleaning, ASJ can match similar strings from different data sources that refer to the same entity, which improves data quality and reduces errors.

ASJ evaluates the similarity of two strings using a similarity function (e.g., Jaccard similarity [18]) or distance metric (e.g., Levenshtein distance [19]). Two strings are regarded as a match if their similarity score is higher than (or their distance metric is smaller than) a user input threshold.

Real-world datasets pose significant challenges to accurately measure the similarity of two strings since they include strings that use various forms of acronyms and abbreviations to represent the same entities and have many typos due to human mistakes in data entry. The police dataset includes a lot of acronyms (e.g., “school resource officer” vs. “sro”), abbreviations (e.g., “deputy marshall” vs. “dpty mrsl”), and typos (e.g., “sergeant” vs. “sargeant”). Several more examples of real equivalent titles we discovered in the police officer dataset may be seen in Figures 1.3 and 1.4.

Traditional similarity measures, such as Levenshtein distance [19] and affine gap distance [3], only consider the scenarios that include typos or acronyms, but do not consider the scenarios that involve acronyms, typos, and abbreviations together. We will discuss previous work on similarity measures in detail in Chapter 2.

	Long String	Short String
Acronym	school resouce officer	sro
Abbreviation-1	<b>deputy marshall</b>	dpty mrsl
Abbreviation-2	dpty <b>marshall</b>	<b>deputy</b> mrsl
Typo	inspector	<b>ims</b> pector
Mixed-1	<b>ins</b> pector	<b>ims</b>
Mixed-2	<b>assistant park manager</b>	apmngr

Figure 1.4: Motivating examples for designing our metric

### 1.3 Our Approach

We propose SMASH, the first dictionary-free string distance metric that considers typos, acronyms, and abbreviations. SMASH is able to effectively capture these cases without the limitations of dictionary-based methods which we will discuss in Chapter 2. Given a long and a short string, the intuition of SMASH is that for every word in the long string, some representation of it (the full word or its abbreviation) should appear as a substring in the short string. Therefore, we first partition the short string into  $k$  substrings, where  $k$  equals the number of words in the long string. The metric SMASH is defined as the minimal sum of the distances between each word in the long string and its corresponding substring. The distance between a word and a substring will be computed based on a combination of traditional measures, such as affine gap [3] or subsequence [20]. Next, we develop a novel dynamic programming algorithm to compute SMASH and extend this algorithm to consider stop words (e.g., “in” or “at”).

We also introduce a new clustering workflow by creating an extension package introducing the SMASH metric to OpenRefine, an open-source data cleaning tool. Since SMASH is good at assigning a string and its acronym, abbreviation, or misspelled form to be close to each other, OpenRefine clustering produces reasonably good clusters. This process allows users to save time compared to manual standardization. The full specifics and an illustrative demo can be found in Chapter 5.

Finally, our experiments on real-world datasets show that SMASH outperforms traditional measures and a state-of-the-art dictionary-based metric. Specifically, SMASH achieves the highest F-score out of all metrics we test on all four test datasets LARGE DISEASE, SMALL DISEASE, LOCATION, and POLICE ROSTER: 0.55, 0.89, 0.86, and 0.84, respectively. In comparison, the next best F-scores on these datasets across different string matching methods and a variety of thresholds are only 0.34, 0.83, 0.85, and 0.59, respectively.

In summary, our contributions are as follows:

1. A new string distance metric, SMASH, that captures all three of abbreviations, acronyms, and misspellings at once without the use of dictionaries.



2. An efficient dynamic programming algorithm that implements the aforementioned metric running in polynomial time.
3. An optional metric improvement in the form of stop words.
4. A new OpenRefine extension package and user walkthrough for how to use our metric.
5. Evaluation results on four real world datasets: POLICE ROSTER, LARGE DISEASE, SMALL DISEASE, and LOCATION against four benchmarks: Levenshtein, Affine Gap, Jaccard, and pkduck showing both the practical speed and efficacy of our metric.

The rest of this thesis is organized as follows: In Chapter 2, we discuss the limitations of prior work to understand why SMASH is necessary. In Chapter 3, we provide some background on string matching and how SMASH calculates the distance between strings. Then, we explain the dynamic programming algorithm for computing the SMASH metric and its implementation details in Chapter 4. In Chapter 5, we discuss how the SMASH metric is implemented in OpenRefine and give a demo of the data cleaning process. We describe a variety of experiments we conduct to assess SMASH’s efficacy and their results in Chapter 6. We briefly discuss potential improvements to SMASH and other potential interesting lines of work in string matching in Chapter 7. Finally, we conclude this report in Chapter 8.

# Chapter 2

## Related Work

Various approaches to string similarity metrics exist, including character and token-based approaches [8, 17, 10], syntactic and synonym-based approaches [15, 16, 14, 13], and phonetic-based approaches [6].

Character-based approaches use only the individual characters of the two strings to measure their similarity. For example, one of the most well known character based distance measures is Levenshtein distance or edit distance [19]. This metric measures the similarity of two strings by counting the minimal number of insertions, deletions, or substitutions required to edit one string to match the other.

Although Levenshtein distance is still widely used in applications to this day and easily captures small typos, it is especially ineffective in capturing acronyms. In practice, clustering using Levenshtein distance will match acronyms not with the correct longer string, but rather with other short words. This happens because it could take many deletions to get from the long string to the short string compared to a few substitutions to get from one short string to another.

An alternative is the Affine Gap distance metric [3], which modifies edit distance by assigning a smaller penalty to continuous insertions or deletions compared to the initial insertion or deletion. Figure 2.1 gives a visual example of how it is computed. This formulation is good at capturing acronyms since the characters that follow the first letters of each word in the longer string are treated as “gaps” in the shortened string and are penalized at a

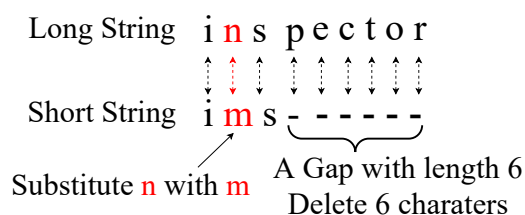


Figure 2.1: An example that showcases the affine gap distance

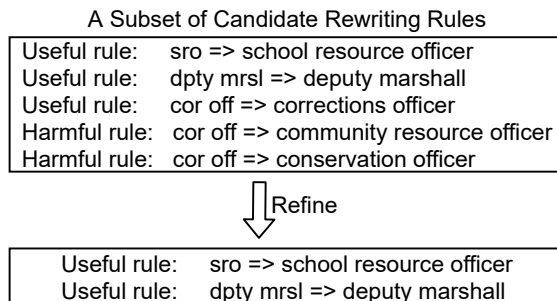


Figure 2.2: A subset of rewriting rules generated by `pkduck` [15] for police officer title data

discount. In practice, however, the out-of-the-box Affine Gap metric does not apply to our problem in cleaning the police roster dataset, due in large part to abbreviations. Because abbreviations can take characters from any position in the longer string in addition to the first letters, gaps would oftentimes not be long enough to benefit from the discount, resulting in a string and its abbreviation having a large distance.

Other methods in the literature include token-based approaches where each string would be split into tokens of 1 or more characters, and those tokens would then be compared to measure distance. For example, the Jaccard [18] method splits each string into a set of  $n$ -grams and then takes the proportion of the size of the intersection of both sets over the size of the union of both sets. Both character-based similarity and token-based similarity approaches have also been labeled as syntactic similarity approaches, because they use only some combination of the characters found in the two strings to determine distance.

Like Levenshtein however, Jaccard is unable to capture acronyms. For example, the bigrams “sr” and “ro” from “sro” appear nowhere in the set of bigrams formed from “school resource officer.” Even the high performing fuzzy matching algorithms such as FuzzyED [17], which combine characteristics from both edit distance and Jaccard, fail to capture all acronyms since neither edit distance nor Jaccard can capture acronyms that are highly condensed, and, at the same time, do not share tokens with the longer string.

In contrast to syntactic similarity, much research has also been focused on semantic similarity, which seeks to rank the similarity of strings based on their meaning. These approaches often store rules for rewriting a shorter string to a longer string (e.g., “sro” → “school resource officer”). To evaluate the similarity of two strings, this line of research adopts a semantic dictionary to rewrite the two strings and adopts Jaccard similarity or its variant to compute the similarity.

We opted not to explore synonym-based approaches because of three main reasons. First, the target datasets which we intend to clean largely contain strings consisting of individual words or short phrases. As such, there is not as much opportunity to use the rules to replace one word with another. Consequently, there is not as much benefit to be gained from the synonym rules. Besides, if there were many opportunities to use the rules, we would more

or less already have what we desire: a ground truth dictionary mapping each word to its modified form! Such a dictionary would likely have to be manually constructed, however, which brings us back to the original problem. The second reason is that synonyms are often collected from data freely available on the web, while we are working in a low data environment. Even if it would be helpful to have certain rules for multi-word strings, it is not guaranteed that we can even find them due to the nature of how arbitrarily these acronyms and abbreviations are created. We could scrape the web and fairly easily discover a rule mapping “VLDB” to “Very Large Data Bases,” for example, but will likely fail to find anything suggesting that “apr” is equivalent to “Assistant Park.”

However, the synonyms do not necessarily have to be manually constructed or scraped from the web. There exist dictionary-based approaches that can generate their own dictionary of synonyms [15]. Even if this is the case, there is a more fundamental limitation to dictionary-based approaches: *one short string may map to multiple long strings, which significantly reduces the precision, but refining the dictionary may wrongly delete useful rules, leading to a low recall score.* Consider the example in Figure 2.2, which shows a subset of rewriting rules that are generated by a state-of-the-art dictionary-based approach, `pkduck`, for the police officer title dataset before and after the refinement. `pkduck` generates candidate rewriting rules based on the longest common sequence of each pair of strings, where a short string maps to many long strings. For the example in Figure 2.2, “cor off” maps to “corrections officer”, “community resource officer”, and “conservation officer”. But the latter two rules are incorrect because “cor off” should only map to “corrections officer”. Therefore, they adopt several refinement rules, where one refinement is to discard the rewriting rules if the ratio between the number of consonants of the short and the long strings are smaller than a pre-defined threshold (0.6 by default) based on the assumption that an abbreviated short string should include a large fraction of consonants from the long string. This refinement rule, while discarding the harmful rules, will also discard the useful rules (e.g., “cor off” → “corrections officer” in Figure 2.2 is discarded because its ratio is  $\frac{4}{11} = 0.37$  and smaller than 0.6). Note that `pkduck` uses a guardrail rule to ensure the acronym rules are maintained (e.g., “sro” → “school resource officer”).

Finally, phonetic-based approaches seek to match strings by how they sound. These approaches have merit in easily catching typos that people can make naturally by confusing homophones for example. However, it isn’t hard to see how the phonetics for an acronym or abbreviation could be vastly different from those of the original string, not to mention that many acronyms would be difficult to fit an accurate pronunciation to in the first place. For these reasons, we opt to explore a character-based approach.

# Chapter 3

## String Metrics

We will begin by introducing some definitions to formalize the problem. Then, we will explain SMASH, our new string distance metric.

### 3.1 Problem Formulation

We define a string  $s$  to be a sequence of characters where  $s[i]$  refers to the  $i$ th character of  $s$  and  $s[i, j]$  refers to the substring of  $s$  from the  $i$ th character to the  $j$ th character. For example, if  $s_1 = \text{“Police Officer”}$ , then  $s_1[7] = \text{‘O’}$ . Note that the space is not skipped. We define a modification as any operation which changes the characters in a string. We refer to entities as real world concepts which are represented by strings. Multiple strings may refer to the same entity in a given dataset, and entity resolution is a primary goal of data cleaning in the context of this report. We also define string clustering as the process by which similar strings are grouped into clusters according to a string distance metric.

Our dataset cleaning problem can be split into two parts: the clustering algorithm  $C$  and the string metric  $m$ . The objective is to design  $m$  such that clustering with  $C$  and  $m$  produces as many clusters containing strings that all refer to the same entity as possible. We additionally want to minimize the number of incorrect clusters to make the process as straightforward as possible for the user, since the end goal is to maximize the time the user can save by using  $m$  over manually constructing the ground truth.

In this paper, we will not focus on  $C$ ; we only focus on  $m$ . In Section 5 where  $C$  is applicable, we use  $k$ -means clustering. We want to design  $m$  such that the distance  $d = m(s_1, s_2)$  is minimized if  $s_1$  is a modified version of  $s_2$ , both referring to the same entity, and maximized otherwise. In this project, the modifications we focus on are abbreviations, acronyms, and misspellings. This way, if  $s_1$  is an abbreviation of  $s_2$  for example, they are likely to be close according to  $m$  and thereby put in the same cluster. If the two strings are unrelated, it is unlikely that there will be a cluster grouping them together.

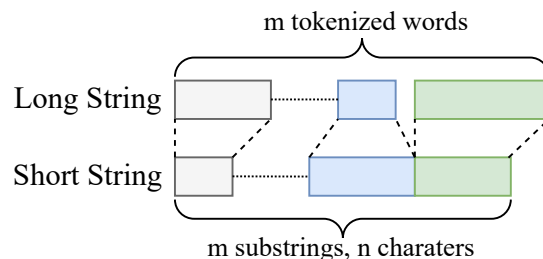


Figure 3.1: The intuition for capturing acronyms, abbreviations, and typos between two strings

## 3.2 Our Similarity Metric

The new SMASH metric is designed with the goals outlined in the previous section in mind and addresses a shortcoming in existing work: the lack of a metric that can simultaneously capture abbreviations, acronyms, and misspellings.

The core intuition behind the algorithm is that for every word in the longer string, some representation of it should appear as a substring in the shorter string. This intuition actually comes from our desire to capture abbreviations and acronyms. Figure 3.1 visualizes this intuition. We found that for acronyms, the first letter of each word would be taken to form the acronym: “School **R**esource **O**fficer”  $\rightarrow$  “**SRO**”. On the other hand, for abbreviations, the first letter along with some other subsequence of characters of each word would be taken to represent that word in the abbreviation. One example is “**A**ssistant **P**ark **M**anager”  $\rightarrow$  “**APrMngr**”.

We represent the longer string as a sequence of words and the shorter string as a sequence of characters. At a high level, the algorithm will match each word in the longer string to contiguous substrings of characters in the shorter string. For each of these pairings, it computes a distance based on the rules explained below and returns the sum of these distances across pairs.

The distance between a word  $w$  from the longer string and a substring  $s$  of the shorter string is defined as follows:

- If their first letters of  $w$  and  $s$  don’t match, return a large number to signify no match. Once again, we do this because we know that acronyms and abbreviations will both have the first letter of each word in the short form. Unfortunately, our metric is less resilient to typos that occur in the first letter of a word as a result.
- Otherwise, if either  $w$  or  $s$  is a subsequence of the other, return 0. We do this because it would satisfy the conditions of being an abbreviation: both the first letter and some subsequence appears in the shortened string.

- Finally, the base case is to simply return the affine gap distance [3, 1] between  $w$  and  $s$ . We opt to use affine gap over Levenshtein because we still want to capture near abbreviations and acronyms. This choice makes the metric more resilient to certain typos. As an example, imagine if “ins”, an abbreviation for “inspector”, was misspelled “ims”. In this case, to get from “inspector” to “ims,” we would make one substitution “m”  $\rightarrow$  “n” and six deletions for “p” through “r.” The affine gap metric is able to discount the removal of “pector,” making it more likely to capture this case than standard edit distance.

One final addition we made to the metric was adding a functionality to skip words, specifically short words or words added to the stop words list as discussed in Section 4.3. If we did not implement word skipping, pairs like (“Special Agent in Charge”, “sac”) would not be matched because there is no “i” corresponding to “in” in the short word. Cases of suffixes that do not change the entity the string refers to such as “mci3” not having a second “i” to match with “Motor Carrier Inspector III” are similarly problematic. If we have the option to skip short words when necessary, a lot of these additional cases are captured.

Section 4 describes the algorithm that is used to implement this metric in code and goes into more detail on certain aspects of the metric.

# Chapter 4

## The Algorithm

We present a dynamic programming algorithm with the following substructure to efficiently calculate the SMASH metric:

$$score(s_1[1, k], s_2[1, p]) = \min(\text{dist}(s_1[k], s_2[p', p]) + score(s_1[1, k-1], s_2[1, p']) \text{ for } p' \text{ in } [1, p]) \quad (4.1)$$

Here, *dist* refers to the distance calculated according to our rules and  $score(s_1[1, k-1], s_2[1, p'])$  is the call to the subproblem whose results are precomputed and memoized.

The algorithm is implemented with two main functions. The *smash* function primarily does preprocessing on the input strings  $s_1$  and  $s_2$  then calls the *dp* function. Notably, we make sure that  $s_1$  is always the longer string and  $s_2$  is the shorter string. The *dp* function is defined to return the distance between the  $s_1$  to the  $k$ th word and the  $s_2$  up to the  $p$ th character, so the *smash* function calls *dp* with the arguments  $k$ =number of words in  $s_1$ ,  $p$ =number of characters in  $s_2$ .

The base cases for the *dp* function are when  $k$  or  $p$  are 0. If they are both 0 (empty string matched with the empty string), then we return 0. If only one is 0, then we have part of one string being matched to the empty string, so we just count that as not a match and return a large number. After checking for base cases, we first initialize the distance to be infinity or make another call to *dp* if the  $k$ th word can be skipped. This way, if no shorter distance is found that includes the  $k$ th word, the distance without it is returned. Next, the algorithm essentially loops to find the best way to match the  $k$ th word of  $s_1$  to a substring of  $s_2$  that ends at the  $p$ th character by selecting the match resulting in the lowest total distance. We do not limit this substring to be up to the same length as the word, allowing the algorithm to search to the beginning of  $s_2$  to start this substring. The objective is to find the best point to start this substring such that the sum of the distance between the substring and the  $k$ th word and the distance between everything that comes before the substring and all the words that come before the  $k$ th word is minimized. The latter comes from the recursive step and is memoized. The former is calculated as explained in Section 3.2. The total distance is calculated by the sum of these two values and the returned value of the *dp* function is the



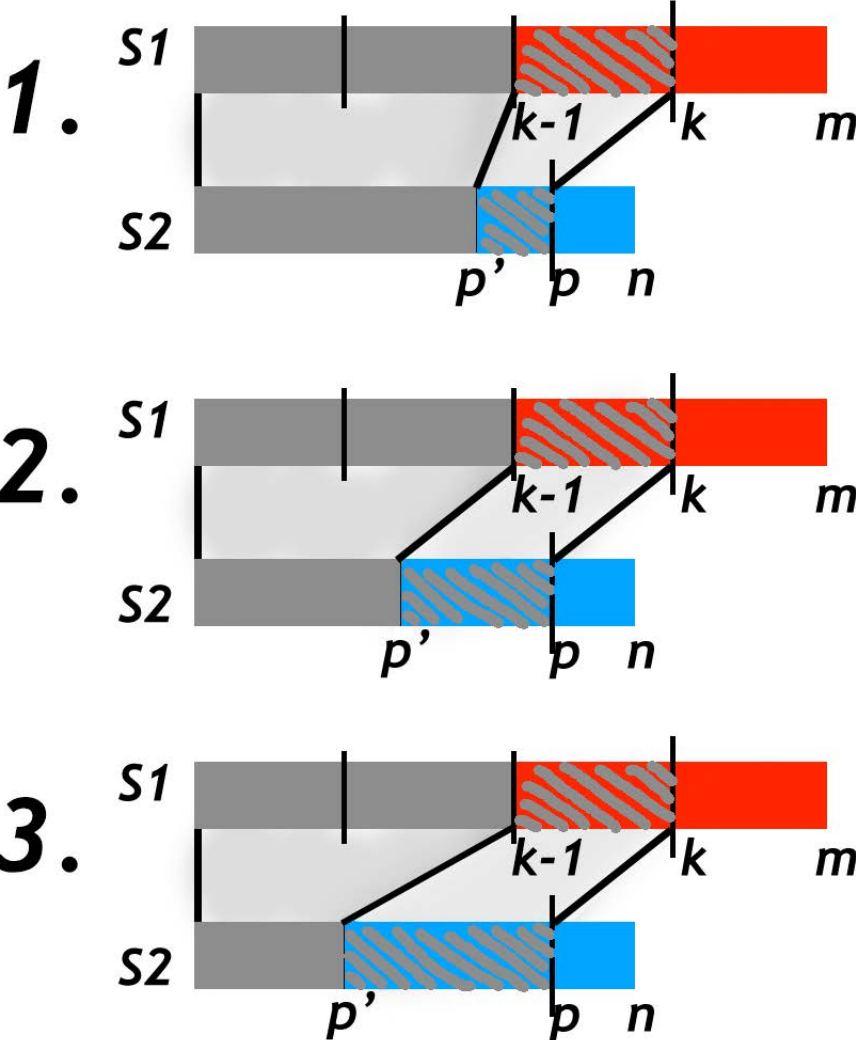


Figure 4.1: Visualization of three iterations of the step-by-step process in the loop of the SMASH algorithm.

smallest distance found across all starting points for the substring. Pseudocode for the two functions is provided below and a visualization of the algorithm is provided in Figures 4.1 and 4.2.

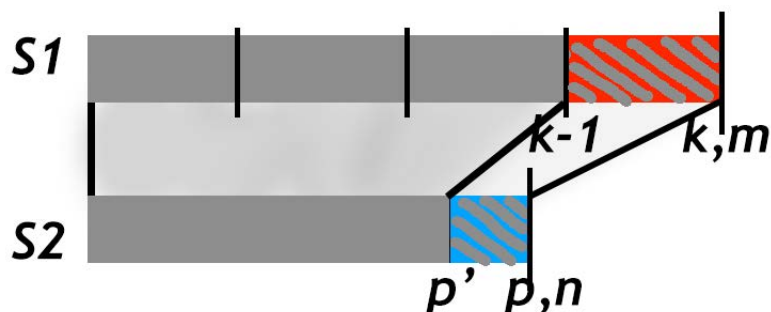


Figure 4.2: Continuation of previous visualization: cached results are used when  $k$  advances.

---

**Algorithm 1** smash
 

---

Input: Strings  $s_1$  and  $s_2$   
 Output: distance between them  
 $s_1, s_2 = \text{trim}(\text{lower}(s_1)), \text{trim}(\text{lower}(s_2))$   
**if**  $\text{length}(s_2) > \text{length}(s_1)$  **then**  
      $s_1, s_2 = s_2, s_1$   
**if**  $s_1[0] \neq s_2[0]$  **then**  
     return  $\infty$   
 $\text{arr1}, \text{arr2} \leftarrow$  arrays containing the words in  $s_1, s_2$   
**if** both arrays have a suffix **then**  
     remove last word from  $\text{arr1}$  and  $s_2$   
 add spaces to the end of each word in  $\text{arr1}$   
 return  $dp(\text{length}(\text{arr1}), \text{length}(s_2))$

---

## 4.1 Proof of Correctness

*Proof.* We will prove the correctness of the SMASH algorithm through induction.

**Base case:** The base case is when  $p = 0$  or  $k = 0$ . If both of them are 0, then we should return 0. We can think of this case as the algorithm having just matched the last word in  $str1$  to the last set of characters in  $str2$  in which case there should be no additional penalty. Alternatively, we can think of it as matching an empty word to the empty string which should trivially get a distance of 0. If only one of  $p$  or  $k$  is 0, however, we should return a large number to signify that there was not a match. If  $p$  is 0 and  $k$  is not, then all of the characters in  $str2$  have been used up in matching to words in  $str1$ , but there are still words left unmatched. If  $k$  is 0 and  $p$  is not, then all of the words in  $str1$  have been matched to a substring of  $str2$ , but there is still a substring of characters in  $str2$  that has

---

**Algorithm 2** dp

---

Input: Integers  $k$  and  $p$ Output: the distance between  $s_1$  up to the  $k$ th word and  $s_2$  up to the  $p$ th character**if**  $k = 0 \wedge p = 0$  **then**

return 0 (base case)

**if**  $k = 0 \vee p = 0$  **then**    return  $\infty$  (base case) $word \leftarrow arr1[k]$  $rest \leftarrow s_2[:p]$  $substr \leftarrow ""$ **if** we can skip the current word because it is short or in an ignore list **then**     $mindist \leftarrow dp(k - 1, p)$ **else**     $mindist \leftarrow \infty$ **while**  $p \neq s_2[0]$  **do**    move  $p$  one character to the left    **if** we can ignore  $p$  (e.g. if  $p$  is punctuation) **then**

continue

 $substr \leftarrow p + substr$     **if**  $word[0] \neq substr[0]$  **then**         $value \leftarrow \infty$     **else if**  $substr$  is a subsequence of  $word$  or vice versa **then**         $value \leftarrow 0$     **else**         $value \leftarrow$  Affine Gap Distance between  $word$  and  $substr$     **if**  $arr1[k - 1][0] = rest[p]$  **then**         $mindist \leftarrow \min(mindist, value + dp(k - 1, p))$     **else**

skip making the recursive call

return  $mindist$ 

---

not been matched to any word. In either of these cases, it would be best to return a penalty representing a “no match.”

**Inductive hypothesis:** Assume that for all  $p$  and  $k$  up to an arbitrary  $p = p'$  and  $k = k'$ , the  $dp$  function will return the correct distance between the first  $k$  words of  $str1$  and the first  $p$  characters of  $str2$ .

**Inductive step:** We will prove that the hypothesis holds for  $p = p + 1$  and  $k = k + 1$ .

$p + 1$  **case:** We will show that  $dp(k, p + 1)$  will return the correct result. If word  $k$  can be skipped, the distance is initialized to  $dp(k - 1, p + 1)$  which ends up being another version of the  $p + 1$  case or the base case if  $k$  reaches 0. In the loop, we calculate  $mindist = \min(mindist, value + dp(k - 1, p'))$  for  $p' < p + 1$ , where  $value$  is the distance between the  $k$ th word and the substring from  $p'$  to  $p + 1$ . We know  $value$  will be correct because it is calculated through the rules we define.  $dp(k - 1, p')$  will also be correct because  $p'$  will range from 0 to  $p$ , all of which are covered under the inductive hypothesis. For each iteration of the loop, the total distance found will be the sum of the two distances and thus also be correct according to how we defined the metric. Finally,  $dp(k, p + 1)$  will return the minimum of these correct distances and thus also yield the correct distance between the  $k$  words and  $p + 1$  characters.

$k + 1$  **case:** We will show that  $dp(k + 1, p)$  will return the correct result. If word  $k + 1$  can be skipped, the distance is initialized to  $dp(k, p)$  which is a correct distance by the inductive hypothesis. In the loop, we calculate  $mindist = \min(mindist, value + dp((k + 1) - 1, p')) = \min(mindist, value + dp(k, p'))$  for some  $p' < p$ , where  $value$  is the distance between the  $k + 1$ th word and the substring from  $p'$  to  $p$ . We know  $value$  will be correct because it is calculated through the rules we define, and  $dp(k, p')$  yields a correct result through the inductive hypothesis.  $dp(k + 1, p)$  will return the minimum of the sums between these correct distances, thus also yielding a correct distance between the  $k + 1$  words and  $p$  characters.  $\square$

## 4.2 Complexity Analysis

The time complexity of dynamic programming problems can be expressed as the number of unique subproblems times  $\times$  time taken per state. For SMASH, each subproblem is defined as finding the distance between  $s_1$  up to the  $k$ th word and  $s_2$  up to the  $p$ th character. If  $s_1$  has  $m$  words and  $s_2$  has  $n$  characters, then the total number of subproblems is  $mn$ .

For each subproblem, in the worst case, the algorithm must scan through the entirety of  $s_2$  in the while loop, until  $p$  becomes the first character of  $s_2$ . Appending to *substr* and checking the first characters are constant time operations. Checking whether one substring is a subsequence of the other is linear in the length of the larger of the two inputs. The affine gap algorithm is linear in the length of its inputs. In the worst case, the input to the affine gap algorithm would be the entirety of  $s_1$  and entirety of  $s_2$ , so  $O(mn)$ . Thus, the overall runtime complexity is  $O(mn \times n \times mn) = O(m^2n^3)$ .

### 4.3 Stop Words

A problem that may occasionally occur in string matching is when a word appears in one modification of a string but is completely absent in the other. Examples include suffixes (e.g. “Motor carrier inspector III” vs. “mci”) or common words such as “the” (e.g. “transient osteoporosis of the hip” vs. “toh”). In these cases, we can simply choose to “ignore” such short words by creating a branch in the algorithm that explores the path in which that word is skipped, essentially matching it to nothing. However, what if it is a longer word that is missing from the modified form? We can know “123 Detective Squad” to be equivalent to “123DET”, but the word “Squad” is not represented at all in the abbreviation.

To address the problem, we introduce a list of stop words: we can have a list of words such that if any word appears within the list, we are free to skip that word, essentially matching it to the empty string within the modified representation. In the following section, we discuss the integration of SMASH in the data cleaning software OpenRefine [12]. We have additionally found that it is possible to modify OpenRefine such that it can generate and append to a stop word list from user actions: if the user manually standardizes “AUD&ACC” to “Audits & Accounts Section”, for example, we will be able to recognize that “Section” does not appear in the modified representation and thus does not contribute to the meaning of the string in a way that impacts its equivalence to its modification. Thus, we then add “Section” to the stop words list so that next time it may be skipped.

## Chapter 5

# Integration with OpenRefine

To investigate the practical applications of SMASH, we created an extension package for OpenRefine [12], an open source data cleaning software. The extension allows users to select SMASH as one of the metrics for clustering string values in OpenRefine. In this section, we will discuss implementation details and present a walkthrough detailing how a user would use our algorithm to clean string data in OpenRefine.

### 5.1 Implementation

We create a Java file that containing a single class that implements the algorithm in Section 4. OpenRefine has an internal interface `SimilarityDistance` which we implement with our class. In the `controller.js` file within the extension package, we add SMASH to the internal `DistanceFactory` object to ensure that it shows up in the dropdown menu. We finally package everything into the file structure OpenRefine desires for their extensions.

To implement the stop words list update, we directly edited the `MassEditOperation.java` file in the source code such that whenever a mass edit occurs, we call a new function `updateIgnoreList` with the old string and the new string. This function determines whether the stop words list, which we have implemented as a simple text file, should be updated. If so, the function writes the word to ignore to a new line in the file.

### 5.2 Demo

To begin cleaning with OpenRefine, the user will first create a project from the dataset they wish to clean. After creating a new project, users can then cluster values of a specific column by clicking on the arrow next to the column name, going to “Edit cells,” and then clicking “Cluster and edit.”

As seen in Figures 5.1, 5.2, using the default methods does not produce very useful clusters for the police roster dataset we are using for demonstration. Many of them produce an extremely limited selection of clusters, while Levenshtein especially tends to just group

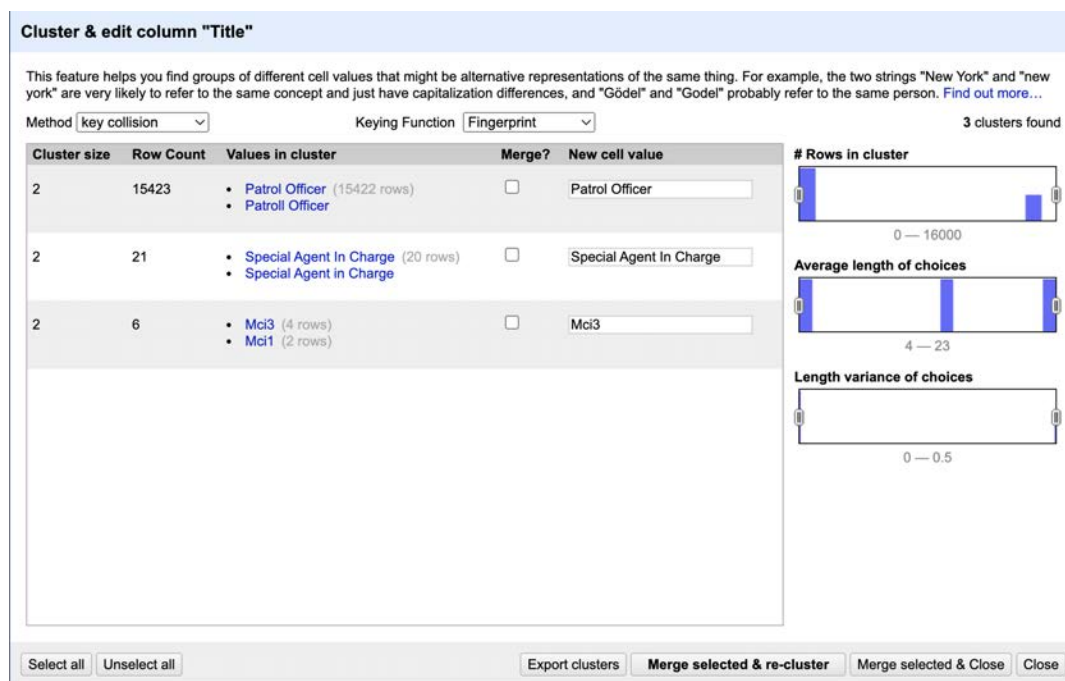


Figure 5.1: Found clusters using the key collision method and fingerprint as the keying function

the shortest titles together. We can change the distance metric to SMASH if the extension is installed by selecting it from the dropdown with the nearest neighbors method selected.

Figure 5.3 shows some of the initial clusters produced by using nearest neighbors clustering with SMASH. OpenRefine lists the largest clusters at the top so they are more likely to contain incorrect equivalencies. Users will scroll toward the bottom where the finer grain clusters are located and check the boxes next to clusters they would like to merge along the way. By default, OpenRefine assigns the most common string in the cluster as the name for the new group, but the new name can be manually edited as the user wishes. Once the user is done picking clusters, they will click the “Merge selected & re-cluster” button. This will standardize every value in each cluster to the value entered as the name of the group and then generate new clusters. Cleaning the data is often an iterative process and the user may have to re-cluster multiple times. To get more or different clusters to show up, users can also use the interactive bars to the right or change the radius. We recommend users to start with a small radius ( $radius \approx 1.0$ ) and increase the radius after merging all desired clusters should they desire to find more clusters.

It is possible that there are still equivalent strings that have not been captured by the clustering process. In this case, users are able to open text facets for the column by clicking the arrow next to the column name, going to “Facet,” and clicking on “Text facet.” Figure 5.4

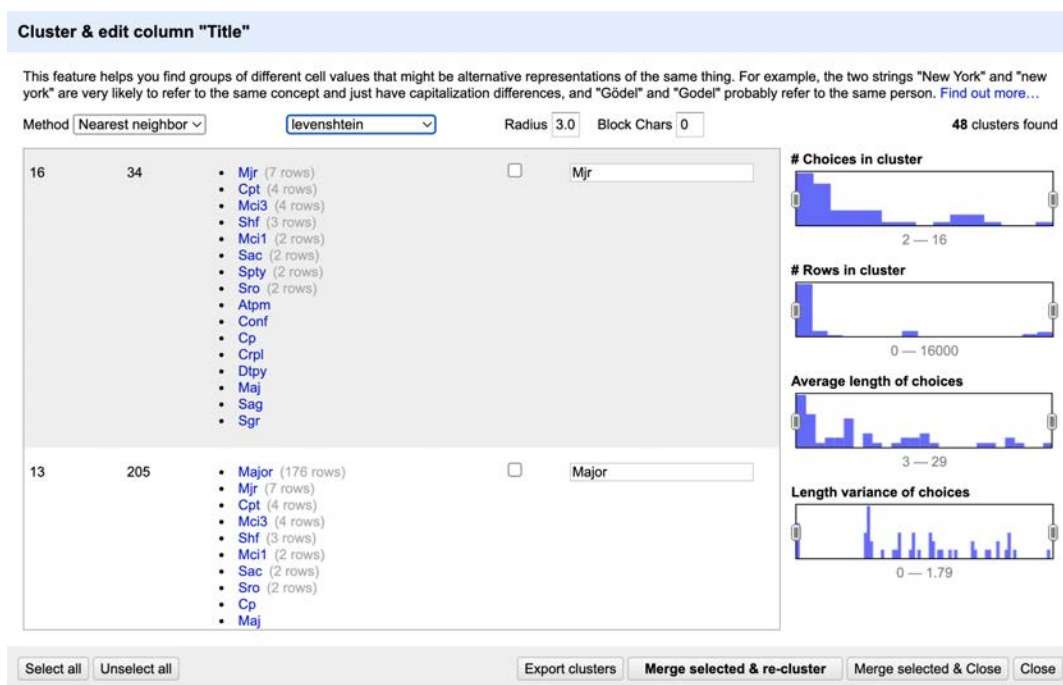


Figure 5.2: Found clusters using nearest neighbor clustering with Levenshtein distance

shows the box that appears upon opening text facets. If the user wants to change “Dpty Chf” to “Deputy Chief”, they can do so by clicking edit and entering the desired value. If the stop words list is implemented, this action will cause any “ignored” words to be appended to the list.

Finally, users can revert to any point in their edit history in the “Undo / Redo” tab. They can also extract all changes, including those applied through the text facets feature, up to their current point in the edit history as a json object. This json can be saved and reused later to repeat operations on a different dataset.



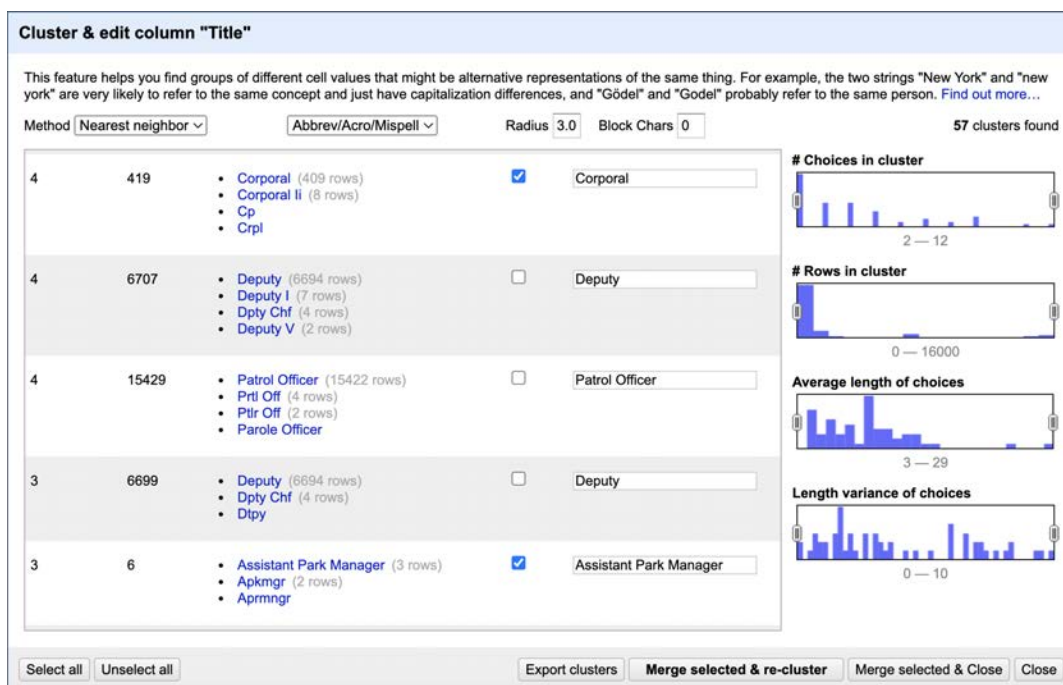


Figure 5.3: Found clusters using nearest neighbor clustering with SMASH distance



Figure 5.4: Text facet interface in OpenRefine

# Chapter 6

## Experiments and Results

### 6.1 Experimental Settings

We evaluate SMASH on four different real life datasets:

1. `POLICE ROSTER` contains 31,516 rows of police officer data with 154 distinct values for “Title” column. `POLICE ROSTER` is the same dataset of police officer titles that has been mentioned in previous sections. We have manually constructed a ground truth for this dataset. However, not every string in the dataset has a modified form, so the ground truth only contains the modified forms present in the dataset and their corresponding standard forms. We also use this dataset as a motivating example for adopting our OpenRefine workflow in Section 5.
2. `LARGE DISEASE` contains 405,543 rows of medical data relating to disease. Each row has a short form of the medical term, its corresponding long form, and some additional metadata. There are no misspellings in this dataset, but it includes many acronyms and abbreviations.
3. `SMALL DISEASE` contains 634 distinct disease names along with their short forms. Like the `LARGE DISEASE` dataset, it contains acronyms and abbreviations.
4. `LOCATION` contains 112,394 distinct location names (street names, city names, etc.) that were collected by Tao et al. [15] from 7,515 tables crawled from Data.gov. The authors sampled approximately 5% of these strings (5677 distinct strings) and manually paired strings that referred to the same location to construct a ground truth dataset containing 116 equivalent pairs of location names. It should be noted that this dataset contains some equivalencies that are not in line with our task of cleaning string data that has primarily been modified by acronyms, abbreviations, and misspellings. We will touch upon this note again in future sections.

As seen in Figures 1.3, 6.1, 6.2, and 6.3, which show some sample strings from the above four datasets, both abbreviations and acronyms are common across all four datasets.

1	acquired immunodeficiency syndrome aids
2	atrial fibrillation afib
3	Moderate Pulmonary Embolism Treated with Thrombolysis mopett

Figure 6.1: Sample longform and shortform string pairs in LARGE DISEASE

1	cmt disease charcot marie tooth disease
2	ic/pbs interstitial cystitis/painful bladder syndrome

Figure 6.2: Sample similar string pairs in SMALL DISEASE taken from the pkduck paper

1	martin luther king ave se mlk avenue se
2	historic columbia river hwy e hist col rvr hwy
3	broadway ste brdway suite

Figure 6.3: Sample similar string pairs in LOCATION taken from the pkduck paper

## 6.2 Evaluation Against Baselines

We begin evaluation by comparing against three baseline metrics described in Section 2: Levenshtein, Affine Gap, and Jaccard. For Affine Gap, we use the default parameters of  $matchWeight = 1$ ,  $mismatchWeight = 11$ ,  $gapWeight = 10$ ,  $spaceWeight = 7$ , and  $abbreviation\_scale = .125$ . For Jaccard, we use n-grams of 3.

We will briefly return to the motivations behind the SMASH algorithm: the need for a general purpose algorithm that can clean messy string datasets. With this in mind, a good metric would be one that makes the workflow easy for the user when integrated into their data cleaning process such as discussed for OpenRefine in Section 5. In OpenRefine, the user is presented with possible clusters depending on how close the strings are to each other. Then, any modified forms of a string should ideally be closer to the correct string than any other strings such that the user would be presented with clusters containing the string and its modified forms with minimal tweaking from the user.

We now present the capture counts metric for evaluation with the above reasoning as motivation. First, we define a longform as an unmodified string, and a shortform as a shorter modified form of the corresponding longform string. For a given dataset with defined longform and shortform pairs, we compute the full distance matrix containing the pairwise distances between strings. Then for each shortform, if the correct longform appears in the set of  $k$  closest words, then we count that shortform as “captured.” In practice, we sampled 500 rows from LARGE DISEASE five times and averaged the results across trials. In Figure 6.4, we show the capture counts for a few metrics, varied across  $k$ . As we can see, SMASH performs far above existing common distance metrics in this regard. Even at  $k = 1$ , SMASH captures nearly 300 shortforms, meaning that out of the 500 shortforms, over half of them have the correct longform as the closest word according to SMASH.

Next, we will look at the runtime performance. First, we sampled some amount of rows from LARGE DISEASE. Then for every metric, we time how long it takes to compute the distance matrix on the sampled shortforms and longforms. Figure 6.5 shows the runtime across number of sampled rows. While it is asymptotically more complex, in practice, SMASH ends up taking similar time to run as affine gap.

We also investigated how the runtime scales with word length. In the next experiment, we again sample 500 rows from LARGE DISEASE and compute the distance matrix. For each comparison, we keep track of the length of the longer string and how long it took to do the comparison. For each length, we average the runtimes of the comparisons and further average the results across 100 trials with different sampled rows. Figure 6.6 displays the results of this experiment. While the other three metrics display a clear trend in how their runtime varies across word length, has a very noisy and jagged graph. We believe the jagged trend can be explained by the different heuristics we have included into the algorithm, allowing many comparisons to stop short regardless of their length.

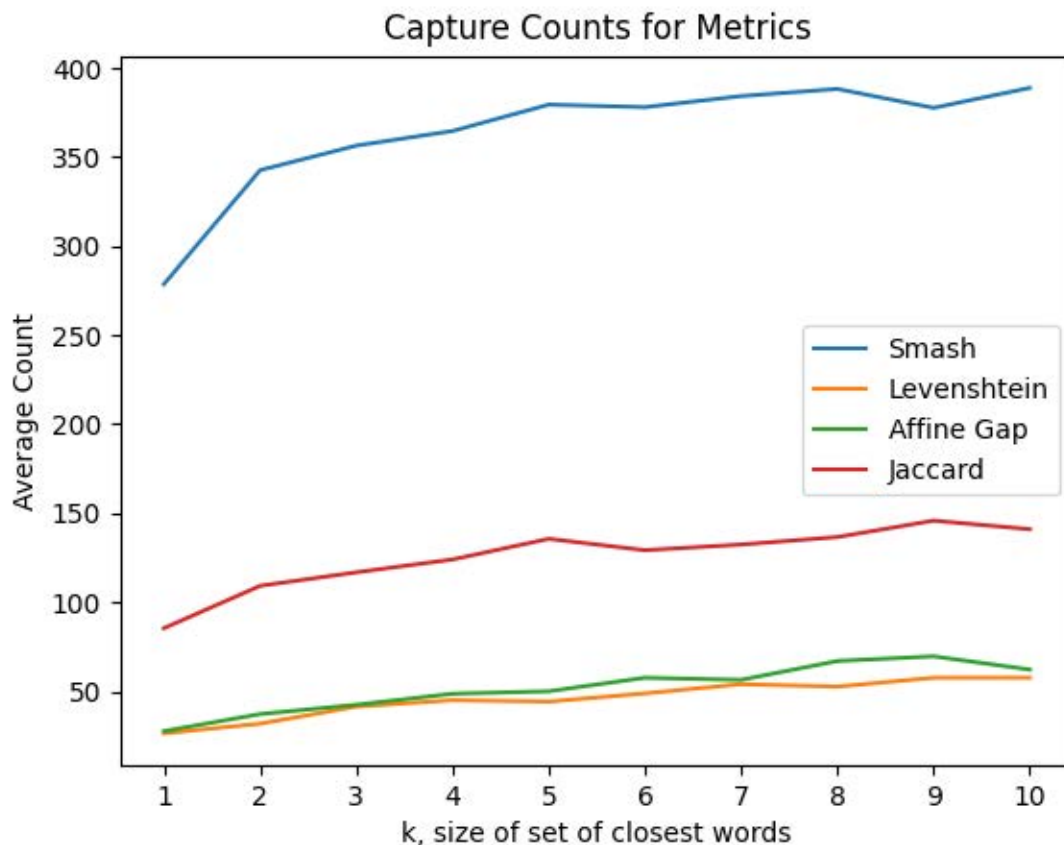


Figure 6.4: Comparison of SMASH, Jaccard, Levenshtein, and Affine Gap using the capture counts metric defined earlier on the large disease dataset

### 6.3 Evaluation Against State-of-the-Art

Next, we test SMASH against the `pkduck` algorithm from [15]. The `pkduck` metric is a synonym based approach that is rather unique in that it generates its own dictionary of synonyms based on finding the longest common subsequence between a pair of strings and checking whether it is a close match to the shorter of the two strings. In the paper, the authors run experiments on the `SMALL DISEASE` and `LOCATION` datasets, recording the precision, recall, and F-Measure. These measures also make sense to compare in the context of our `OpenRefine` workflow because high recall would indicate higher chance of users seeing correct matches, and high precision would indicate lower chance of seeing incorrect matches, reducing the time spent sorting through matches. We compiled their best results on the two datasets from the paper and compare them against ours in Table 6.1. We have a threshold

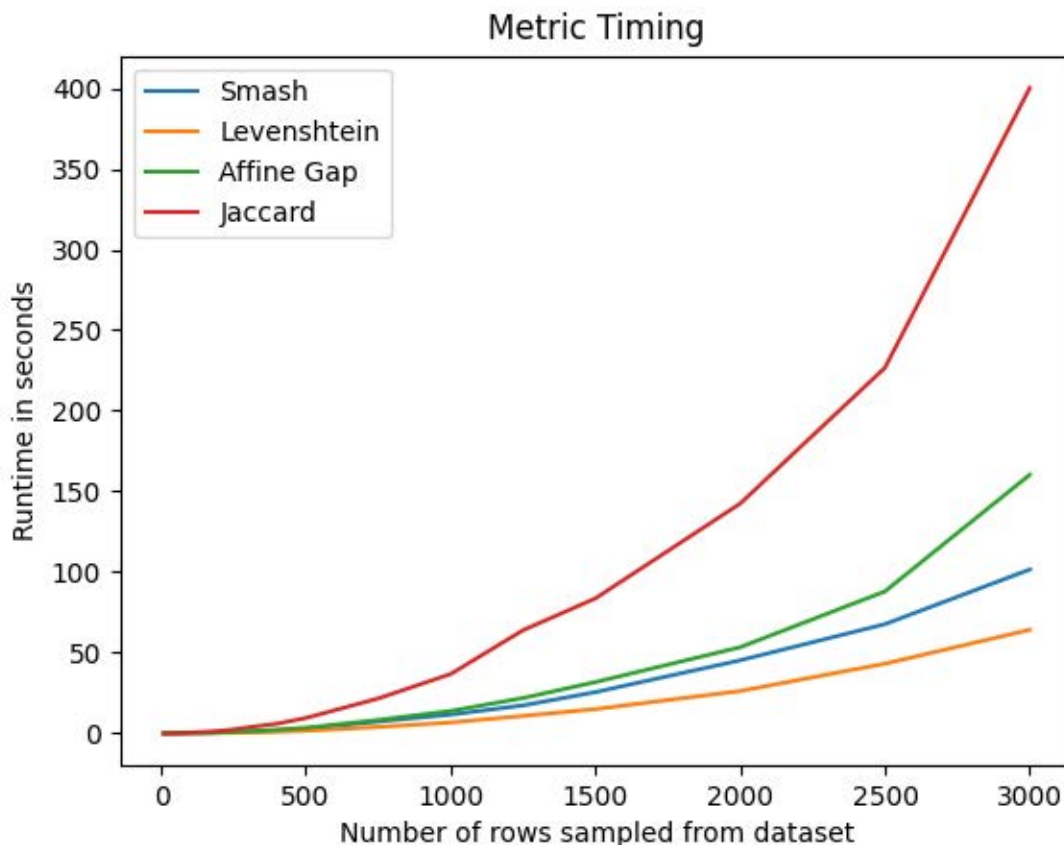


Figure 6.5: Runtime comparison with sampled rows from the large disease dataset

distance below which any string is counted as a positive. We then count the true positives, false positives, and false negatives for each string to calculate the statistics and select the threshold producing the highest F-Measure. In Table 6.1, we can see that SMASH has a higher F-score than pkduck on both test datasets.

## Rule Refiner

To corroborate the validity of our results above, we next decided to manually label the ground truth for POLICE ROSTER and run both SMASH and pkduck on the labeled POLICE ROSTER dataset. To our surprise, pkduck performed rather poorly, so we decided to investigate why pkduck was missing cases that it should theoretically capture. For example, it was missing the rule that “Dmrsl” maps to “Deputy Marshall” despite the former being a direct subsequence of the latter. We found that the culprit was the rule refiner that they implemented as part

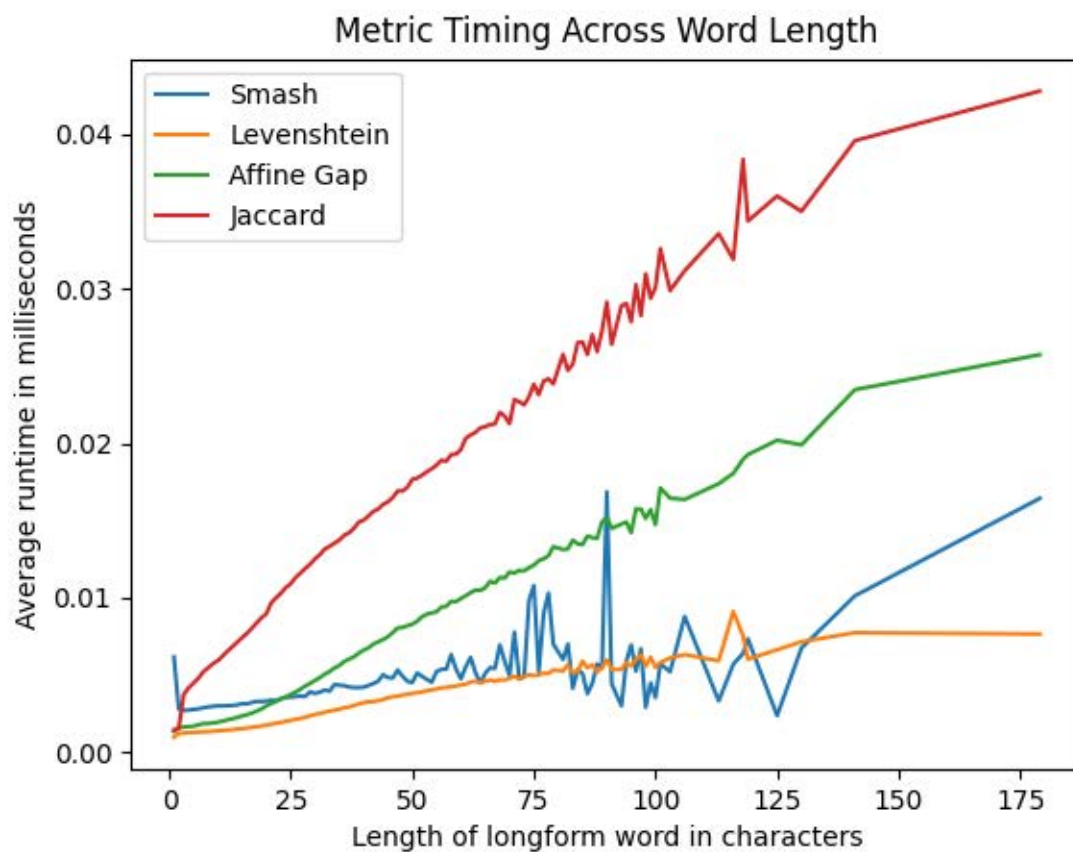


Figure 6.6: Runtime comparison across word length with sampled pairs from the large disease dataset

	Precision (disease)	Recall (disease)	F-Measure (disease)	Precision (location)	Recall (location)	F-Measure (location)
pkduck	0.99	0.72	0.83	0.46	0.55	0.5
smash	0.90	0.88	0.89	0.95	0.78	0.86

Table 6.1: Comparison of SMASH against pkduck on the SMALL DISEASE and LOCATION datasets



	Precision (Refiner on)	Recall (Refiner on)	F-Measure (Refiner on)	Precision (Refiner off)	Recall (Refiner off)	F-Measure (Refiner off)
$\theta = 0.7$	0.83	0.33	0.48	0.73	0.50	0.59
$\theta = 0.8$	0.83	0.25	0.38	0.72	0.43	0.54
$\theta = 0.9$	1.00	0.25	0.40	0.79	0.43	0.56

Table 6.2: Comparison of pkduck refiner on POLICE ROSTER

of pkduck.

The rule refiner is put in place to filter out any unlikely rules that were generated by the LCS process. Some examples they use in the paper are “ay” = “dictionary” and “ia” = “information”. How it works in practice is that they check that the ratio of number of consonants in the short form to number of consonants in the long form is greater than or equal to 0.6. This explains why the previously mentioned case was missing from the rules in the dictionary. We decided to compare the performance of pkduck on the POLICE ROSTER dataset with the refiner on and off. As seen in Table 6.2, it achieves slightly higher performance without the refiner on POLICE ROSTER. However, turning the refiner off does lower precision, as the dictionary is populated with incorrect rules. For example, we saw “cor off” = “conservation officer” and “sro” = “supervisor” were rules in the dictionary when their correct matches should be “corrections officer” and “school resource officer”, respectively.

This example highlights the brittleness of dictionaries and gives some theoretical justification for taking a character-based approach like SMASH as opposed to a synonym-based approach to string metrics. Empirically, it can be noted that SMASH outperforms pkduck whether or not the refiner is turned off. For future experiments, we leave the rule refiner turned on.

## 6.4 Further Experiments

### Tuning Jaccard

It should be noted that while the authors of the pkduck paper found the best performance of Jaccard on LOCATION to have F-score=0.3 with  $\theta = 0.7$ , we found that by increasing the cutoff to 0.6 and using n-grams of size 3, it is able to achieve statistics of p=0.94, r=0.83, and F=0.88 which outperforms SMASH. However, as noted earlier in this section, we found that the data in LOCATION does not fully reflect the problem we try to solve with SMASH. The cases that SMASH does not capture in this dataset can be summarized as follows: 1. extra word ex. “south blue island” vs “south blue island avenue”, and 2. rearranged words “b ave suite” vs “ave ste b”. The first case can be remedied through our stop words list as explained in Section 4, while the second case is out of scope for now. Furthermore, Jaccard’s performance on the SMALL DISEASE dataset, which moreso contains data in line

with SMASH’s intended use, is capped at F-score  $\approx 0.1$ .

We decided to do further investigation on Jaccard following its somewhat surprising performance on the LOCATION dataset. Firstly, we note that it may be unreasonable to expect the users to tune the metric’s threshold for each individual dataset but rather use some common threshold that should work around equally as well across datasets. We found that while Jaccard’s performance can vary wildly across different threshold values and datasets, SMASH performs reasonably well across a wider range of settings. For future experiments, we will select the threshold  $T = 1$  as the threshold for SMASH, as we find that using  $T = 1$  rather than the dataset optimal threshold does not result in a significant loss in performance on any of our four test datasets. For Jaccard, we use  $T = 0.3$  as it is the best performing threshold from the pkduck paper.

Secondly, we have earlier selected n-grams of size 3 for Jaccard. This was because  $n = 3$  allowed for the best performance on the LARGE DISEASE dataset when testing with the capture counts metric. However, we later found that using  $n = 1$ , or character-wise Jaccard, allowed for higher performance on both SMALL DISEASE and POLICE ROSTER when using a threshold of 0.3.

Finally, while we were using Jaccard with n-grams, it is very common to use word-wise Jaccard, where the tokens are the words within a string rather than n-grams. In fact, this is what the pkduck paper used for Jaccard in their experiments. We thought it would be beneficial to test word-wise Jaccard because there are some settings where it may outperform character-wise Jaccard, such as the LARGE DISEASE datasets where strings that have many words are a more common appearance (“Toll-IL-1 receptor domain-containing adaptor-inducing IFN-beta” vs. “toll/interleukin-1 receptor domain-containing adapter-inducing interferon beta” for example). The performance of Jaccard under these conditions can be found in Figure 6.3.

## Comparison against LLMs

Recent advances in large language models, such as ChatGPT and the GPT-3 model ChatGPT is built on [4], have the potential to significantly enhance the task of cleaning string data. In this section, we finally compare the efficacy of string matching with LLMs to that of SMASH.

First, we attempt to use the ChatGPT interface to generate matching string pairs from a list of assorted longforms and shortforms. To prompt the model, we first instruct it of its task: “I will give you a list of strings. Can you tell me which of them are equivalent?” and optionally include some example inputs and outputs. We gain a few insights from this procedure. Figure 6.7 shows an instance of successful string matching. We sample 50 random shortform-longform pairs from LARGE DISEASE, shuffle them, and then provide them as the list of strings to ChatGPT. The model’s output includes 44 matched pairs, of which 42 are correct.

However, our experiments also show that ChatGPT is extremely unreliable, especially when the number of input strings grows larger. For example, when attempting to feed

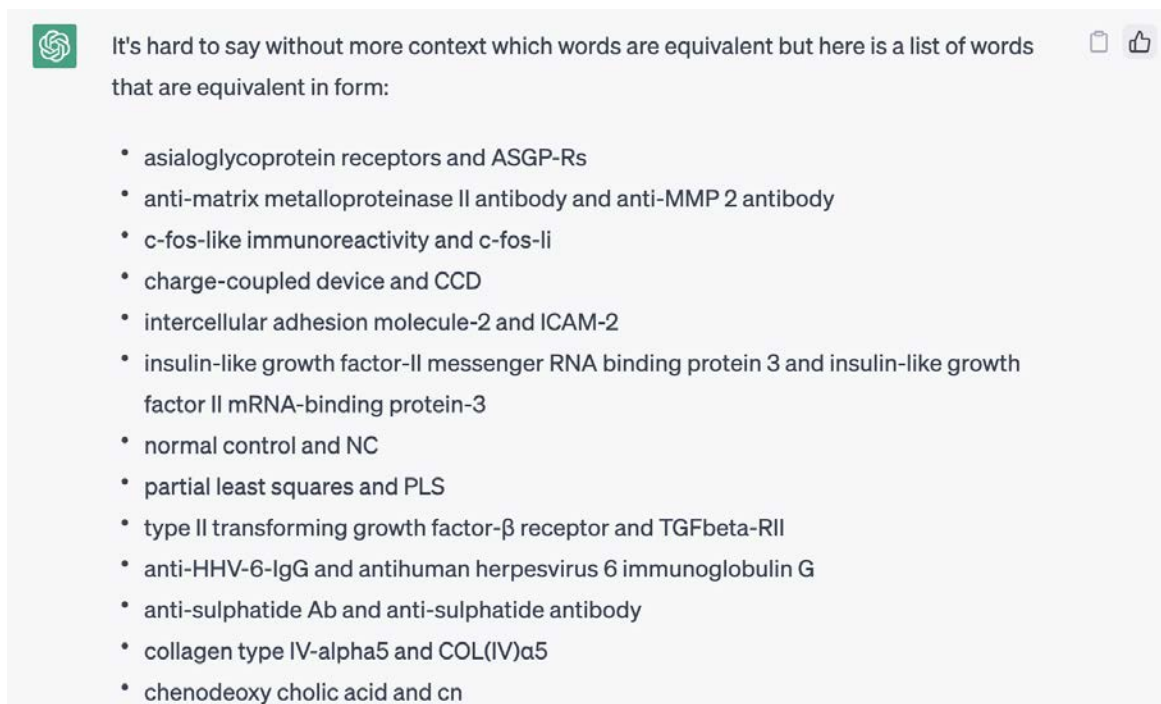


Figure 6.7: Most successful example of ChatGPT string matching

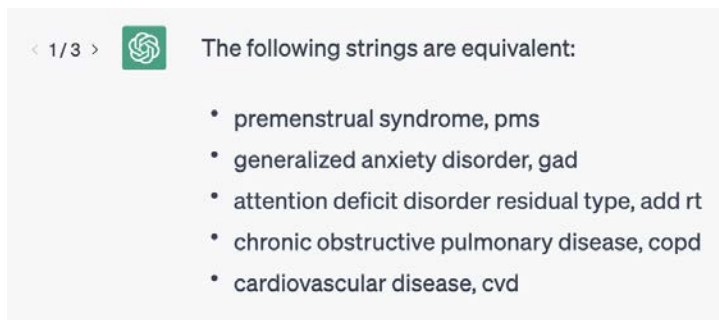


Figure 6.8: ChatGPT only outputs a few correct pairs

SMALL DISEASE as the input list of strings to ChatGPT, Figures 6.8, 6.9, and 6.10 showcase the common outputs. The model will sometimes output only a few correct strings. Most commonly, it will output some explanation for why the task cannot be done without further context, and rarely, it will completely fail and assert that all of the strings are referring to distinct entities.

Because of ChatGPT's unreliability, we will instead turn our attention elsewhere for the

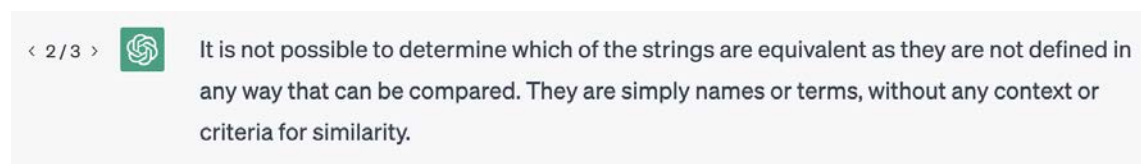


Figure 6.9: ChatGPT avoids the task

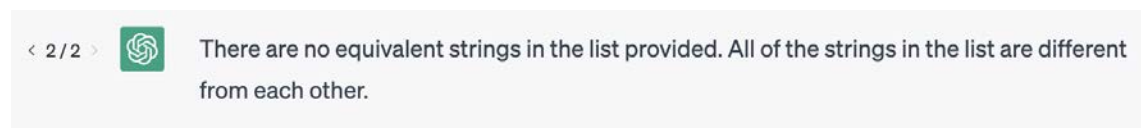


Figure 6.10: ChatGPT outputs something completely untrue

remaining experiments. We introduce GPT for Sheets [7], a ChatGPT extension for Google Sheets. GPT for Sheets includes a *GPT\_MAP* function that uses the *text-ada-002-embeddings* model under the hood. It would make sense for us to compare against this model because the latent embeddings of these LLMs likely contain some semantic insights about how words and potential modified forms of those words would relate to each other.

The *GPT\_MAP* function finds the  $k$  closest longforms for each shortform and optionally outputs a similarity score between 0 and 1 for each shortform longform pair. By turning similarity scores on and setting  $k = \text{len}(\text{longforms})$ , we can compute the full distance matrix using the embeddings model and find precision, recall, and F-score using a threshold as usual. The results can be found in Table 6.3. It should be noted that the optimal threshold for the GPT-based metric varies wildly across datasets. Should we use the optimal threshold for each dataset, its performance with  $T = 0.21$  on SMALL DISEASE would improve to  $F \approx 0.54$  and its performance with  $T = 0.11$  on LOCATION would match that of SMASH's. We selected  $T = 0.15$  to show results for GPT that are reasonable for all datasets.

## Final Results

Table 6.3 shows the collective results of our experiments. Because the LARGE DISEASE dataset was still too large to compute a distance matrix for in a reasonable amount of time, we sampled 30,000 rows or  $\approx 7.4\%$  of the dataset to run the experiments with. Despite reducing the size of the dataset, *GPT\_MAP* was still unable to compute the distance matrix and instead produced an error so we are unable to report those results. For Affine Gap and Levenshtein, the threshold was chosen in a similar way to GPT. While the optimal threshold was very different for Levenshtein across datasets, the optimal threshold for Affine Gap was very similar on each dataset.

Results												
Distance Metric	Large Disease			Small Disease			Location			Police		
	P	R	F	P	R	F	P	R	F	P	R	F
Smash (T=1)	0.47	0.66	<b>0.55</b>	0.89	0.88	<b>0.89</b>	0.95	0.78	<b>0.86</b>	0.94	0.77	<b>0.84</b>
Levenshtein (T=14)	0.79	0.22	0.34	0.93	0.07	0.13	0.12	0.99	0.22	0.32	0.8	0.46
Affine Gap (T=1.5)	0.93	0.08	0.14	0.58	0.43	0.5	0.95	0.78	0.72	0.84	0.42	0.56
Jac-Word (T=0.3)	1	0	0	1	0	0	0.53	0.21	0.3	0.98	0.12	0.21
Jac-Char (T=0.3)	0.96	0.05	0.1	0.99	0.02	0.04	0.71	0.86	0.78	0.86	0.39	0.54
pkduck (T=0.3)	0.12	0.15	0.13	0.88	0.74	0.81	0.76	0.55	0.64	0.83	0.33	0.48
GPT (T=0.15)	ERR	ERR	ERR	0.98	0.11	0.19	0.59	0.95	0.73	0.53	0.57	0.55

Table 6.3: Final results

The precision, recall, and F-score for a given metric and dataset is calculated by finding the distance from each shortform in the dataset to every longform in the dataset according to the metric. Then, the true positives, false positives, and false negatives are found using the computed distances, ground truth, and threshold. We bring this up because this formulation is different from the `pkduck` paper’s experiments on all datasets but `SMALL DISEASE` so we rerun their experiments on `LOCATION` using their code.

The key takeaway from Table 6.3 is that using the same threshold, `SMASH` is able to achieve the highest F-score on each of the four evaluation datasets.

To match the experimental setup of the `pkduck` paper we also convert `SMASH` to a similarity measure, which we will define as having a range from 0 to 1, with 1 indicating that the two strings are most similar and 0 indicating that they are completely dissimilar. We do this by feeding the outputs of the `SMASH` distance metric into an exponential decay function. For a given pair of strings, if their similarity is above a threshold  $\theta$ , they will count as a match. For this final experiment, we vary  $\theta$  in the range [0.7, 0.8, 0.9] and calculate PRF across the same four datasets for the distance metrics that can be equivalently defined as similarity measures. Table 6.4 displays the results of this experiment. The key takeaway here is that while `SMASH` performs well across thresholds, none of the other similarity measures are able to beat `SMASH`’s best F-scores even when using different thresholds.

Results Varied Across Threshold													
Similarity Measure	$\theta$	Large Disease			Small Disease			Location			Police		
		P	R	F	P	R	F	P	R	F	P	R	F
Smash	0.7	0.29	0.73	0.41	0.65	0.89	0.75	0.89	0.78	0.83	0.77	0.82	0.8
	0.8	0.41	0.68	0.51	0.85	0.88	0.87	0.95	0.78	<b>0.86</b>	0.94	0.77	<b>0.84</b>
	0.9	0.48	0.65	<b>0.55</b>	0.89	0.88	<b>0.89</b>	0.95	0.78	0.86	0.94	0.75	0.83
Jaccard-Word	0.7	1	0	0	1	0	0	0.99	0.22	0.35	0.98	0.12	0.21
	0.8	1	0	0	1	0	0	0.99	0.22	0.35	0.98	0.03	0.06
	0.9	1	0	0	1	0	0	1	0.01	0.02	0.98	0.03	0.06
Jaccard-Char	0.7	0.96	0.05	0.1	0.99	0.02	0.04	0.7	0.86	0.77	0.86	0.4	0.55
	0.8	0.99	0.02	0.04	1	0	0	0.92	0.72	0.8	0.89	0.28	0.43
	0.9	1	0.01	0.01	1	0	0	0.99	0.41	0.58	0.98	0.23	0.38
pkduck	0.7	0.12	0.15	0.13	0.88	0.74	0.81	0.76	0.55	0.64	0.83	0.33	0.48
	0.8	0.16	0.12	0.14	0.97	0.72	0.83	0.94	0.28	0.44	0.83	0.25	0.38
	0.9	0.19	0.1	0.13	0.99	0.72	0.83	0.97	0.26	0.41	1	0.25	0.4
GPT	0.7	ERR	ERR	ERR	0.01	1	0.01	0.01	1	0.02	0.07	0.97	0.12
	0.8	ERR	ERR	ERR	0.7	0.41	0.51	0.17	0.99	0.29	0.26	0.83	0.41
	0.9	ERR	ERR	ERR	1	0.02	0.05	0.91	0.8	0.85	0.87	0.45	0.59

Table 6.4: Final results, varied across threshold

# Chapter 7

## Future Work

The mismatch of the LOCATION dataset to SMASH’s use case, while introducing some problems we considered out of scope, highlights some limitations of character-based string matching that could be further explored in future work. Those that come after us are free to look into the more general cases in string data cleaning which we elected not to prioritize in lieu of focusing on abbreviations, acronyms, and misspellings, such as the cases of extraneous stop words and swapped words mentioned in Section 6. SMASH may also be tweaked to be less absolute in requiring the first letters to match; this would help capture cases where the misspelling occurs in the first letter (“apple” vs. “bpple”, for example). It may also be interesting to investigate whether it would be possible to adopt insights from semantic similarity approaches and use existing dictionaries of synonyms alongside SMASH to capture more cases.

Despite the unreliability of current techniques we saw in Section 6, large language models still hold much potential in the field of string data cleaning. These models are trained on an extremely large corpus of data and would be able to make use of contextual information which is missing with a character based approach. They could learn that “Lt.” is equivalent to “Lieutenant” from the fact that they appear in the same contexts in the training data, for example. More notably, this allows them to recognize that “Skin Cancer” is equivalent to “NMSC” for example, a match that would be difficult to find with character-based approaches. We have also seen from the embedding model’s ability to capture many cases on LOCATION that some semantic insights are indeed being captured in the embeddings and that they may be useful in capturing cases that metrics like SMASH are unable to by design.

These large language models can be used to generate text that is semantically similar to the input string, which can be used to detect and correct typos and spelling errors. Additionally, they can be used to identify and remove irrelevant information, such as stop words or unnecessary punctuation, from the input string. For another example, if the input string contains a date, the model can generate text that contains a consistent and plausible date format. As the field of language models continues to evolve, it is likely that even more sophisticated techniques for cleaning string data will emerge.

## Chapter 8

# Conclusion

In this report, we explored data cleaning with the NACDL’s Full Disclosure Project and discussed the problem of matching equivalent police titles. To address this problem, we presented SMASH, a new character-based string distance metric that can robustly detect abbreviations, acronyms, and misspellings of words and short phrases without the use of dictionaries. We explained how SMASH recursively matches the words in the longer string to substrings in the shorter string and the manner in which it assigns a score for the matched strings. We looked at the field of string similarity measures and highlighted the limitations of the current approaches with respect to our use cases. We explored the workflow introduced by SMASH’s integration with OpenRefine and explained how this process simplifies data cleaning. Experimental results on four datasets show that SMASH beats out not only the standard baselines, but also the state-of-the-art and embedding based approaches in approximate string matching. We additionally show that SMASH is efficient and scalable with memoization.



# Bibliography

- [1] *affinegap*. An open source Python library that implements the affine gap penalty string distance. <https://github.com/dedupeio/affinegap>.
- [2] Arvind Arasu, Surajit Chaudhuri, and Raghav Kaushik. “Transformation-based framework for record matching”. In: *2008 IEEE 24th International Conference on Data Engineering*. IEEE. 2008, pp. 40–49.
- [3] Mikhail Bilenko and Raymond J Mooney. “Employing Trainable String Similarity Metrics for Information Integration.” In: *IIWeb*. 2003, pp. 67–72.
- [4] Tom Brown et al. “Language Models are Few-Shot Learners”. In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle et al. Vol. 33. Curran Associates, Inc., 2020, pp. 1877–1901. URL: [https://proceedings.neurips.cc/paper\\_files/paper/2020/file/1457c0d6bfc4967418bfb8ac142f64a-Paper.pdf](https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfc4967418bfb8ac142f64a-Paper.pdf).
- [5] Ahmed K Elmagarmid, Panagiotis G Ipeirotis, and Vassilios S Verykios. “Duplicate record detection: A survey”. In: *IEEE Transactions on knowledge and data engineering* 19.1 (2006), pp. 1–16.
- [6] Junior Ferri, Hegler Tissot, and Marcos Didonet Del Fabro. “Integrating approximate string matching with phonetic string similarity”. In: *Advances in Databases and Information Systems: 22nd European Conference, ADBIS 2018, Budapest, Hungary, September 2–5, 2018, Proceedings 22*. Springer. 2018, pp. 173–181.
- [7] *GPT for Work*. A ChatGPT extension for Google Sheets and Docs. <https://gptforwork.com/>.
- [8] Oktie Hassanzadeh, Mohammad Sadoghi, and Renée J Miller. “Accuracy of Approximate String Joins Using Grams.” In: *QDB*. 2007, pp. 11–18.
- [9] Liang Jin, Chen Li, and Sharad Mehrotra. “Efficient record linkage in large data sets”. In: *Eighth International Conference on Database Systems for Advanced Applications, 2003.(DASFAA 2003). Proceedings*. IEEE. 2003, pp. 137–146.
- [10] Chen Li, Jiaheng Lu, and Yiming Lu. “Efficient merging and filtering algorithms for approximate string searches”. In: *2008 IEEE 24th International Conference on Data Engineering*. IEEE. 2008, pp. 257–266.

- [11] Jiaheng Lu et al. “Synergy of Database Techniques and Machine Learning Models for String Similarity Search and Join”. In: *Proceedings of the 28th ACM International Conference on Information and Knowledge Management, CIKM 2019, Beijing, China, November 3-7, 2019*. Ed. by Wenwu Zhu et al. ACM, 2019, pp. 2975–2976. DOI: 10.1145/3357384.3360319. URL: <https://doi.org/10.1145/3357384.3360319>.
- [12] *OpenRefine*. An open source data cleaning application. <https://github.com/OpenRefine/OpenRefine>.
- [13] Gwangho Song, Kyuseok Shim, and Hongrae Lee. “Substring similarity search with synonyms”. In: *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE. 2021, pp. 2003–2008.
- [14] Gwangho Song et al. “String joins with synonyms”. In: *Database Systems for Advanced Applications: 25th International Conference, DASFAA 2020, Jeju, South Korea, September 24–27, 2020, Proceedings, Part III 25*. Springer. 2020, pp. 389–405.
- [15] Wenbo Tao, Dong Deng, and Michael Stonebraker. “Approximate String Joins with Abbreviations”. In: *Proceedings of the VLDB Endowment Volume 11, Issue 1*. ACM. 2017, pp. 53–65.
- [16] Jin Wang et al. “An Efficient Sliding Window Approach for Approximate Entity Extraction with Synonyms.” In: *EDBT*. 2019, pp. 109–120.
- [17] Zeyi Wen et al. “2ED: An Efficient Entity Extraction Algorithm Using Two-Level Edit-Distance”. In: *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 2019, pp. 998–1009. DOI: 10.1109/ICDE.2019.00093.
- [18] Wikipedia contributors. *Jaccard index*. [Online; accessed 27-February-2023]. 2023. URL: [https://en.wikipedia.org/wiki/Jaccard\\_index](https://en.wikipedia.org/wiki/Jaccard_index).
- [19] Wikipedia contributors. *Levenshtein distance*. [Online; accessed 27-February-2023]. 2023. URL: [https://en.wikipedia.org/wiki/Levenshtein\\_distance](https://en.wikipedia.org/wiki/Levenshtein_distance).
- [20] Wikipedia contributors. *Subsequence*. [Online; accessed 27-February-2023]. 2023. URL: <https://en.wikipedia.org/wiki/Subsequence>.
- [21] Chuan Xiao et al. “Efficient similarity joins for near-duplicate detection”. In: *ACM Transactions on Database Systems (TODS)* 36.3 (2011), pp. 1–41.