2013-12-11

Record Linkage

Stasha Ann Bown Larsen
Brigham Young University - Provo

Record Linkage

Stasha Ann Bown Larsen

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

Jeffrey Humpherys, Chair
Christopher Grant
Shue-Sum Chow

Department of Mathematics

Brigham Young University

December 2013

ABSTRACT

Record Linkage

Stasha Ann Bown Larsen
Department of Mathematics, BYU
Master of Science

This document explains the use of different metrics involved with record linkage. There are two forms of record linkage: deterministic and probabilistic. We will focus on probabilistic record linkage used in merging and updating two databases. Record pairs will be compared using character-based and phonetic-based similarity metrics to determine at what level they match. Performance measures are then calculated and Receiver Operating Characteristic (ROC) curves are formed. Finally, an economic model is applied that returns the optimal tolerance level two databases should use to determine a record pair match in order to maximize profit.

Keywords: Probabilistic record linkage, Character-based similarity metrics, Phonetic-based similarity metrics, ROC curves

Acknowledgments

My thesis would not be possible without the love and support of so many wonderful people in my life. The first person that I would like to thank is my husband, Mike. You have been my rock and best friend through graduate school. I am so grateful that I am able to share my life with you.

The second person that I would like to thank is my mother. Thank you Mom for teaching me the value of hard work and the importance of getting an education. You are the strongest person I know and I admire you so much. I would also like to thank my amazing siblings Dave, Cyd (and Capella), Ryan (and Adrienne), and Sheila. Family is one of life's greatest blessings and I am so thankful that each one of you are in my life. Next I would like to thank my second family, the Larsens. Michelle, Kent, Elizabeth, and Ruby, thank you for always treating me like a daughter and sister. I love being apart of your family and appreciate your love and kindness so much.

Finally, I would like to thank my advisor Jeff. You gave me my first research job in the math department, and I never had any reason to want to work with anyone else. Thank you for always believing in me, even when I did not believe in myself. (It was because of you and your encouragement that I entered graduate school to begin with.) I will miss working with you and hearing your words of wisdom. But please know that I will always remember the best piece of advice you ever gave me: "Stasha, if you want to get good at tennis you have to play with people that are better than you."

Contents

List of Tables

List of Figures

Chapter 1. Background Information

1.1 Definition

Record linkage is the process of determining if data records from the same or different data sources refer to the same entity. Data records can include names, addresses, dates of birth, social security numbers, occupation, and marital status. Data sources are books, databases, and websites. An entity is usually a person, but it could also be an organization or a company.

We will focus on record linkage in databases. Databases are sets of structured and properly segmented data records. Record linkage is similar to mirror detection and anaphora resolution. Mirror detection deals with detecting similar or identical web pages, whereas anaphora resolution works on locating different mentions of the same entity in text [1, 2]. Algorithms that are used for mirror detection and anaphora resolution do have some applications to finding duplicates in databases, but we will not focus on these methods.

1.2 History

The first major scholarly article on record linkage, entitled "Record Linkage", was in 1946 by Halbert L. Dunn [3]. Dunn's initial work on record linkage was added upon when Howard Borden Newcombe, J. M. Kennedy, S.J. Axford, and A.P. James proposed a probabilistic approach in their 1959 article "Automatic Linkage of Vital Records" [4]. The probabilistic approach to record linkage was then formalized by Ivan Fellegi and Alan Sunter in "A Theory For Record Linkage" in 1969 [5]. Fellegi and Sunter showed that the probabilistic decision rule they described in their paper was optimal when the attributes being compared were conditionally independent. Their findings are the foundation for many record linkage applications still used today.

Record linkage has various names among different research disciplines. In the Artificial Intelligence community it is called database hardening [6] and name matching [7]. In the database community it is called data deduplication [8], instance identification [9], and merge-

purge [10]. In statistics it is also called record matching [5]. Record linkage is also called coreference resolution, duplicate record detection, and identity uncertainty [11].

Record linkage can be done without a computer. However, computers are used for better quality control, central supervision of processing, consistency, reproducibility of results, and speed.

1.3 Why is Record Linkage Important?

Record linkage is needed to join datasets based on entities that may or may not have a common identifier, or to remove redundant entries in a single dataset. Data sources we are interested in usually lack a global identifier, otherwise we could join data records on this identifier and easily remove duplicate entities, which would make the datasets not interesting. Moreover, records are not consistent across different data sources. For example, there could be data entry errors resulting in misspellings, invalid data given field constraints, or simply different conventions for storing the same information. When data sources are managed independently the structure, semantics, and assumptions about the data are very likely to be different.

1.4 Data Heterogeneity

Data heterogeneity refers to potential systematic differences that occur in data from different data sources [12]. Resolving such differences is referred to as data cleaning, data pre-processing, data scrubbing, data standardization, or ETL (extraction, transformation and loading) [13]. The two types of data heterogeneity are structural and lexical. Structural heterogeneity happens when fields in different data sources are structured differently. For example, in one data source an employee's address can be found in the field called *address*. However, in another data source an employee's address can be located in multiple fields called *street*, *apartment*, *city*, *state*, and *zip_code*. Lexical heterogeneity happens when fields across

data sources are the same, but the data representing the entity is different. An example of this would be in one data source the field *date_of_birth* represents information in the form Month day, year (example April 11, 1973) and the field *date_of_birth* in another data source represents information in the form month/day/year (example 04/11/73).

To address the problems of data heterogeneity the data must first be prepared, which generally consists of three steps: parsing, data transformation, and standardization [11]. Parsing is the process of locating, identifying, and isolating individual data elements within a record so that comparisons can be made between individual elements, rather than the whole record. Data transformation is the process of converting data in a field from one type to another, renaming a field, range checking, and dependency checking. Standardization is the process of transforming the data into a consistent format, so that records that are the same no longer look different. For example, all names in the data source can be given in the order last name, first name, all dates are given in the form MM/DD/YYYY, and all states are represented by their two letter abbreviation. Standardization is done through rule based data transformations or lexicon based tokenization and probabilistic hidden Markov models. Standardizing the data so that it looks the same is easier to do than to normalize names given in the data. This is because there is more than one way to spell many names. For example, Aaron and Erin are both pronounced the same, but do not have the same spellings. To help normalize names, first and last names can be run through a phonetic algorithm like Soundex, NYSIIS, or Double Metaphone to help determine which names really are the same.

After the data has been prepared field matching occurs next. Field matching is done in order to get rid of duplicate records that occur as a result of misspellings and different conventions being used to record the same information. Character-based similarity metrics can be used to help find misspelled words. Examples of these metrics include: edit distance, affine gap distance, Smith-Waterman distance, Jaro distance, and q-gram distance.

Chapter 2. Character-Based Similarity Metrics

2.1 Edit Distance

The edit distance (also called Levenshtein distance) metric is the minimum number of single character edit operations needed for strings s and t, in order to transform the string s into string t [14]. Edit distance is effective at finding typographical errors, but it is not effective at finding other types of mismatches. For example, it cannot tell that two names are identical if in one database the name is written last name, first name and in the other database it is written first name last name. It also does not work well if strings have been truncated or shortened.

Copy, insert, delete, and substitute are the four types of edit operations. Copy copies an identical character from the string, insert inserts a character into the string, delete deletes a character from the string, and substitute substitutes one character in the string with a different character. Copying a character from string s to t costs 0, deleting a character from string s costs 1, inserting a character from string t costs 1, and substituting one character for another costs 1 [15]. Below we give three examples of calculating the edit distance between two strings.

Example 2.1. distance(Stasha Bown, Sttasha Brown)

string s = Stasha Bown

string t = Sttasha Brown

operation CCICCCC CICCC, where C = copy and I = insert

cost (cumulative) 0011111 12222

Example 2.2. distance(SStasha Bawn, Stasha Bown)

string s = SStasha Bawn

string t = Stasha Bown

operation CDCCCCC CSCC, where C = copy, D = delete, and S = substitute

cost (cumulative) 0111111 1222

Example 2.3. distance(SStasha Bawn, Sttasha Brown)

string s = SStasha Bawn

string t = Sttasha Brown

operation CSCCCCC CSICC, where C = copy, I = insert, and S = substitute

cost (cumulative) 0111111 12333

Considering the costs we defined above, if string $s = s_1 s_2 \cdots s_m$ and string $t = t_1 t_2 \cdots t_n$, the score for the best alignment for comparing $s_1, s_2, \ldots, s_m$ to $t_1, t_2, \ldots, t_n$ where $1 \leq i \leq m$ and $1 \leq j \leq n$ is

$$D(i,j) = min \begin{cases} D(i-1, j-1) & \text{if } s_i = t_j, \text{ copy} \\ D(i-1, j-1) + 1 & \text{if } s_i \neq t_j, \text{ substitute} \\ D(i-1, j) + 1 & \text{insert} \\ D(i, j-1) + 1 & \text{delete} \end{cases}$$

This can be simplified to

$$D(i,j) = min \begin{cases} D(i-1, j-1) + d(s_i, t_j) & \text{for a substitution or copy} \\ D(i-1, j) + 1 & \text{insert} \\ D(i, j-1) + 1 & \text{delete} \end{cases}$$

Where $d(a, b)$ is a distance function on characters that is related to typo frequencies. Thus $d(a,b) = \begin{cases} 0 & \text{when } a = b \\ 1 & \text{otherwise.} \end{cases}$ We also have that $D(i, 0) = i$ for i inserts and $D(0, j) = j$ for j deletes [15].

Example 2.4. An example of the edit distance is given below where the strings DAVID and SDDAVD are being compared and $d(a,b) = \begin{cases} 0 & \text{when } a = b \\ 1 & \text{otherwise.} \end{cases}$

	D	A	V	I	D
S	1	2	3	4	5
D	1	2	3	4	4
D	2	2	3	4	4
A	3	2	3	4	5
V	4	3	2	3	4
D	5	4	3	3	3

Note that the edit distance for comparing DAVID and SDDAVD is 3 (the value found in the lower right hand corner of the table). These distance values were formed as a result of comparing the following characters and finding the minimum value.

	s_1	s_2	s_3	s_4	s_5
t_1	d, s [1] min val = 1	a, s [2, 2] min val = 2	v, s [3, 3] min val = 3	i, s [4, 4] min val = 4	d, s [5, 5] min val = 5
t_2	d, d [2, 1] min val = 1	a, d [3, 2, 2] min val = 2	v, d [4, 3, 3] min val = 3	i, d [5, 4, 4] min val = 4	d, d [6, 5, 4] min val = 4
t_3	d, d [2, 2] min val = 2	a, d [3, 3, 2] min val = 2	v, d [4, 3, 3] min val = 3	i, d [5, 4, 4] min val = 4	d, d [5, 5, 4] min val = 4
t_4	d, a [3, 4] min val = 3	a, a [3, 4, 2] min val = 2	v, a [4, 3, 3] min val = 3	i, a [5, 4, 4] min val = 4	d, a [5, 5, 5] min val = 5
t_5	d, v [4, 5] min val = 4	a, v [3, 5, 4] min val = 3	v, v [4, 4, 2] min val = 2	i, v [5, 3, 4] min val = 3	d, v [6, 4, 5] min val = 4
t_6	d, d [5, 5] min val = 5	a, d [4, 6, 5] min val = 4	v, d [3, 5, 4] min val = 3	i, d [4, 4, 3] min val = 3	d, d [5, 4, 3] min val = 3

Table 2.1: Edit distance table comparing string s = DAVID and string t = SDDAVD.

The more different two strings are, the larger their edit distance will be. For example, if the strings STASHA and DSSTSGA are compared they would have an edit distance of 4. It could be that these two strings were both suppose to be STASHA. However, as a result of the d key being pressed by accident and then over-correcting we ended up with DSS at the beginning instead. Furthermore, the letter a in the middle is missing and the g key near the end was hit instead of the h key. So many typos in such a small string are unlikely, but mistakes do happen. To get a better understanding of why distance(STASHA, DSSTSGA) = 4, please see the edit distance table below.

	S	T	A	S	H	A
D	1	2	3	4	5	6
S	1	2	3	3	4	5
S	2	2	3	3	4	5
T	3	2	3	4	4	5
S	4	3	3	3	4	5
G	5	4	4	4	4	5
A	6	5	4	5	5	4

These distance values were formed as a result of comparing the following characters and finding the minimum value.

	s_1	s_2	s_3	s_4	s_5	s_6
t_1	s, d [1] min val = 1	t, d [2, 2] min val = 2	a, d [3, 3] min val = 3	s, d [4, 4] min val = 4	h, d [5, 5] min val = 5	a, d [6, 6] min val = 6
t_2	s, s [2, 1] min val = 1	t, s [3, 2, 2] min val = 2	a, s [4, 3, 3] min val = 3	s, s [5, 4, 3] min val = 3	h, s [6, 4, 5] min val = 4	a, s [7, 5, 6] min val = 5
t_3	s, s [2, 2] min val = 2	t, s [3, 3, 2] min val = 2	a, s [4, 3, 3] min val = 3	s, s [4, 4, 3] min val = 3	h, s [5, 4, 4] min val = 4	a, s [6, 5, 5] min val = 5
t_4	s, t [3, 4] min val = 3	t, t [3, 4, 2] min val = 2	a, t [4, 3, 3] min val = 3	s, t [4, 4, 4] min val = 4	h, t [5, 5, 4] min val = 4	a, t [6, 5, 5] min val = 5
t_5	s, s [4, 4] min val = 4	t, s [3, 5, 4] min val = 3	a, s [4, 4, 3] min val = 3	s, s [5, 4, 3] min val = 3	h, s [5, 4, 5] min val = 4	a, s [6, 5, 5] min val = 5
t_6	s, g [5, 6] min val = 5	t, g [4, 6, 5] min val = 4	a, g [4, 5, 4] min val = 4	s, g [4, 5, 4] min val = 4	h, g [5, 5, 4] min val = 4	a, g [6, 5, 5] min val = 5
t_6	s, a [6, 7] min val = 6	t, a [5, 7, 6] min val = 5	a, a [5, 6, 4] min val = 4	s, a [5, 5, 5] min val = 5	h, a [5, 6, 5] min val = 5	a, a [6, 6, 4] min val = 4

Table 2.2: Edit distance table comparing string s = STASHA and string t = DSSTSGA.

The edit distance of two strings can be calculated by hand or using a computer. The python code that I used to calculate the edit distance metric is located in the appendix. It is important to note that the boundary conditions for when i = 0 or j = 0 have to be handled differently. Furthermore, when s_i and t_j are being compared we have to take into account how far into the string we are.

2.2 Affine Gap Distance

The affine gap distance metric is an extension of the edit distance metric [16]. With affine gap distance there are two additional operations called open gap and extend gap.

The cost of a gap of n characters is $nG = A + (n-1)B$, where A is the cost of opening

the gap and B is the cost of continuing the gap. The affine gap distance $D(i, j)$ can be calculated as follows:

$$D(i,j) = max \begin{cases} D(i-1, j-1) + d(s_i, t_j) \\ SI(i-1, j-1) + d(s_i, t_j) \\ TJ(i-1, j-1) + d(s_i, t_j) \end{cases}$$

where $d(s_i, t_j)$ is an arbitrary distance function on characters that is related to typo frequencies.

$$SI(i,j) = max \begin{cases} D(i-1, j) - A \\ SI(i-1, j) - B \end{cases}$$

which is the best score for which s_i is aligned with a gap and

$$TJ(i,j) = max \begin{cases} D(i, j-1) - A \\ TJ(i, j-1) - B \end{cases}$$

which is the best score for which t_j is aligned with a gap [15].

2.3 Smith-Waterman Distance

The Smith-Waterman distance metric is an extension of the edit and affine gap distance metrics [17]. With the Smith-Waterman distance metric, mismatches in the middle of the string have a higher cost than mismatches at the beginning and end. This allows the metric to have better substring matching.

The value of the Smith-Waterman distance metric is given by

$$D(i,j) = max \begin{cases} 0 & \text{start over} \\ D(i-1,j-1) - d(s_i,t_j) & \text{for a substitution or copy} \\ D(i-1,j) - G & \text{insert} \\ D(i,j-1) - G & \text{delete} \end{cases}$$

Where the distance is the maximum value over all i, j in the table of $D(i,j)$ [15]. Note that $d(s_i,t_j)$ is an arbitrary distance function on characters that is related to typo frequencies and the term G is a penalty cost for having to perform an insertion or deletion. It is also important to note that the distance cannot be negative, so a start over value of zero is used. If the distance does become a negative value, then the start over option is applied and the distance value becomes zero.

Example 2.5. An example of the Smith−Waterman distance metric is given below where $G = 1, d(c,c) = -2, d(c,d) = 1$, and start over is not used. e.g.,

$$D(i,j) = max \begin{cases} D(i-1,j-1) + 2 & \text{for a copy when } s_i = t_j \\ D(i-1,j-1) - 1 & \text{for a substitution when } s_i \neq t_j \\ D(i-1,j) - 1 & \text{insert} \\ D(i,j-1) - 1 & \text{delete} \end{cases}$$

	D	A	V	I	D
S	-1	-2	-3	-4	-5
D	+1	0	-1	-2	-2
D	0	0	-1	-2	0
A	-1	+2	+1	0	-1
V	-2	+1	+4	+3	+2
D	-3	0	+3	+3	+5

Note that the Smith-Waterman distance for comparing DAVID and SDDAVD is 5 (the value found in the lower right hand corner of the table). These distance values were formed as a result of comparing the following characters and finding the maximum value.

	s_1	s_2	s_3	s_4	s_5
t_1	d, s [-1] max val = -1	a, s [-2, -2] max val = -2	v, s [-3, -3] max val = -3	i, s [-4, -4] max val = -4	d, s [-5, -5] max val = -5
t_2	d, d [-2, 1] max val = 1	a, d [-3, 0, -2] max val = 0	v, d [-4, -1, -3] max val = -1	i, d [-5, -2, -4] max val = -2	d, d [-6, -3, -2] max val = -2
t_3	d, d [0, 0] max val = 0	a, d [-1, -1, 0] max val = 0	v, d [-2, -1, -1] max val = -1	i, d [-3, -2, -2] max val = -2	d, d [-3, -3, 0] max val = 0
t_4	d, a [-1, -4] max val = -1	a, a [-1, -2, 2] max val = 2	v, a [-2, 1, -1] max val = 1	i, a [-3, 0, -2] max val = 0	d, a [-1, -1, -3] max val = -1
t_5	d, v [-2, -5] max val = -2	a, v [1, -3, -2] max val = 1	v, v [0, 0, 4] max val = 4	i, v [-1, 3, 0] max val = 3	d, v [-2, 2, -1] max val = 2
t_6	d, d [-3, -3] max val = -3	a, d [0, -4, -3] max val = 0	v, d [3, -1, 0] max val = 3	i, d [2, 2, 3] max val = 3	d, d [1, 2, 5] max val = 5

Table 2.3: Smith-Waterman distance table comparing string s = DAVID and string t = SDDAVD.

Example 2.6. If the start over value of 0 is used, then the same two strings being compared have the following distance table.

	D	A	V	I	D
S	0	0	0	0	0
D	+1	0	0	0	+2
D	0	0	0	0	+2
A	0	+2	+1	0	+1
V	0	+1	+4	+3	+2
D	0	0	+3	+3	+5

Similarly we see that the Smith-Waterman distance for comparing DAVID and SDDAVD when the start over value of 0 is used is 5 (the value found in the lower right hand corner of the table).

Example 2.7. If the strings $s =$ STASHA and $t =$ WTAAHZ where compared using the Smith-Waterman distance and start over is not used along with $G = 1, d(c, c) = -1, d(c, d) = 1, D(i, 0) = i$, and $D(0, j) = j$. Then

$$D(i,j) = max \begin{cases} D(i-1, j-1) + 1 & \text{for a copy when } s_i = t_j \\ D(i-1, j-1) - 1 & \text{for a substitution when } s_i \neq t_j \\ D(i-1, j) - 1 & \text{insert} \\ D(i, j-1) - 1 & \text{delete} \end{cases}$$

Then string $s =$ STASHA and string $t =$ WTAAHZ have the following distance table.

	S	T	A	S	H	A
W	-1	-2	-3	-4	-5	-6
T	-2	0	-1	-2	-3	-4
A	-3	-1	+1	0	-1	-2
A	-4	-2	0	0	-1	-2
H	-5	-3	-1	-1	+1	0
Z	-6	-4	-2	0	0	0

Note that the Smith-Waterman distance for comparing STASHA and WTAAHZ is 0 (the value found in the lower right hand corner of the table). These distance values were formed as a result of comparing the following characters and finding the maximum value.

	s_1	s_2	s_3	s_4	s_5	s_6
t_1	s, w [-1] max val = -1	t, w [-2, -2] max val = -2	a, w [-3, -3] max val = -3	s, w [-4, -4] max val = -4	h, w [-5, -5] max val = -5	a, w [-6, -6] max val = -6
t_2	s, t [-2, -2] max val = -2	t, t [-3, -3, 0] max val = 0	a, t [-1, -4, -3] max val = -1	s, t [-2, -5, -4] max val = -2	h, t [-3, -6, -5] max val = -3	a, t [-4, -7, -6] max val = -4
t_3	s, a [-3, -3] max val = -3	t, a [-4, -1, -3] max val = -1	a, a [-2, -2, 1] max val = 1	s, a [0, -3, -2] max val = 0	h, a [-1, -4, -3] max val = -1	a, a [-2, -5, -2] max val = -2
t_4	s, a [-4, -4] max val = -4	t, a [-5, -2, -4] max val = -2	a, a [-3, 0, 0] max val = 0	s, a [-1, -1, 0] max val = 0	h, a [-1, -2, -1] max val = -1	a, a [-2, -3, 0] max val = 0
t_5	s, h [-5, -5] max val = -5	t, h [-6, -3, -5] max val = -3	a, h [-4, -1, -3] max val = -1	s, h [-2, -1, -1] max val = -1	h, h [-2, -2, 1] max val = 1	a, h [0, -1, -2] max val = 0
t_6	s, z [-6, -6] max val = -6	t, z [-7, -4, -6] max val = -4	a, z [-5, -2, -4] max val = -2	s, z [-3, -2, -2] max val = -2	h, z [-3, 0, -2] max val = 0	a, z [-1, -1, 0] max val = 0

Table 2.4: Smith-Waterman distance table comparing string s = STASHA and string t = WTAAHZ.

Example 2.8. If the start over value of 0 is used, then the same two strings being compared have the following distance table.

	S	T	A	S	H	A
W	0	0	0	0	0	0
T	0	+1	0	0	0	0
A	0	0	+2	+1	0	+1
A	0	0	+1	+1	0	+1
H	0	0	0	0	+2	+1
Z	0	0	0	0	+1	+1

Then the Smith-Waterman distance for comparing STASHA and WTAAHZ when the start over value of 0 is used is 1 (the value found in the lower right hand corner of the table).

The python code that I used to calculate the Smith-Waterman distance metric is given in the appendix.

2.4 Jaro Distance

The Jaro distance metric is a string comparison algorithm that is primarily used for first and last name comparison [18]. Given a string s and a string t, c is said to be a common character in s and t if $s_i = c, t_j = c$, and $|i - j| < \frac{min(|s|,|t|)}{2}$ (or $i - H \leq j \leq i + H$ where $H = \frac{min(|s|,|t|)}{2}$). Furthermore, the characters d and e are said to be a transposition of each other if the characters d and e are both characters in string s and string t, but d and e appear in different orders in s and t. This leads to the Jaro distance metric of strings s and t to be $Jaro(s,t)$ = average of $\frac{\#common}{|s|}$, $\frac{\#common}{|t|}$, and $\frac{0.5\#transpositions}{\#common}$ [15].

More formally, the Jaro distance metric can be viewed as follows. Let $s' = s'_1, s'_2, ..., s'_k$ and $t' = t'_1, t'_2, ..., t'_l$ where s' consists of the characters that s has in common with t and t' consists of the characters that t has in common with s. A transposition in s' in position i occurs when $s'_i < t'_i$. And a transposition in t' in position i occurs when $s'_i > t'_i$. If we let $T_{s',t'}$ be one half of the number of transpositions in s' and t', then $Jaro(s,t) = \frac{1}{3}\left(\frac{|s'|}{|s|} + \frac{|t'|}{|t|} + \frac{|s'|-T_{s',t'}}{|s'|}\right)$ [19].

A variant on the Jaro distance metric can be achieved by weighting the errors that occur earlier in the string more heavily [15].

Example 2.9. Find the Jaro distance between *Stasha* and *Satsha*.

$s = Stasha$, $|s| = 6$

$t = Satsha$, $|t| = 6$

$H = \frac{min(|s|,|t|)}{2} = \frac{min(6,6)}{2} = 3$

$s' = Stasha$, $|s'| = 6$

$t' = Satsha$, $|t'| = 6$

$T_{s',t'} = \frac{1}{2}(2) = 1$

$Jaro(s,t) = \frac{1}{3}\left(\frac{6}{6} + \frac{6}{6} + \frac{6-1}{6}\right) = \frac{17}{18} = .9\overline{4}$

Example 2.10. Find the Jaro distance between *Stasha Bown* and *SStasja Brpwm*.

$s = Stasha\ Bown$, $|s| = 11$

$t = SStasja\ Brpwm$, $|t| = 13$

$H = \frac{min(|s|,|t|)}{2} = \frac{min(11,13)}{2} = 5.5$

$s' = Stasa\ Bw$, $|s'| = 8$

$t' = Stasa\ Bw$, $|t'| = 8$

$T_{s',t'} = \frac{1}{2}(0) = 0$

$Jaro(s,t) = \frac{1}{3}\left(\frac{8}{11} + \frac{8}{13} + \frac{8-0}{8}\right) = 0.7\overline{808857}$

The python code used to calculate the Jaro distance between *Stasha* and *Satsha* along with *Stasha Bown* and *SStasja Brpwm* is given in the appendix. The python code is based on code written in C for calculating the Jaro distance from the jellyfish library available for download at https://github.com/sunlightlabs/jellyfish.

2.5 Q-gram Distance

A q-gram is a character substring of length q [20]. If two strings s and t are similar, then they will have a number of q-grams in common. The more q-grams two strings have in common, the more likely that the two strings match. A q-gram is formed for a string s by sliding a window of length q over the characters of s. At the beginning and end of a string, fewer than q characters will be present. An option available to fix this problem is to pad the string s at the beginning and the end with $q-1$ padding characters that are not part of the original alphabet [21]. Otherwise, q-grams are formed using only the characters given in the string s. Letter q-grams (q-grams that are of length one, two, or three) can also be used to help fix spelling mistakes and for text recognition. Q-grams can be extended to positional q-grams, which record the position of each q-gram in string s [22].

Example 2.11. Let the string $s = STASHA$.

If $q = 1$, then the q-grams of string s are equal to the set $S = \{S, T, A, S, H, A\}$.

If $q = 2$, then the q-grams of string s are equal to the set $S = \{ST, TA, AS, SH, HA\}$.

If $q = 3$, then the q-grams of string s are equal to the set $S = \{STA, TAS, ASH, SHA\}$.

If $q = 4$, then the q-grams of string s are equal to the set $S = \{STAS, TASH, ASHA\}$.

If $q = 5$, then the q-grams of string s are equal to the set $S = \{STASH, TASHA\}$.
If $q = 6$, then the q-gram of string s is equal to the set $S = \{STASHA\}$.

The python code used to generate the q-grams of a string is given in the appendix.

Q-grams can be used as the basis for a distance metric. To calculate the q-gram distance between string s and string t with a q-gram of size n the following steps are taken. First, the q-grams of each string must be found. The set of q-grams of size n for the string s will be called set S and the set of q-grams of size n for the string t will be called set T. Second, the set U, which contains the q-grams that S and T have in common is formed. The size of the set U will be denoted by u. Third, the set V is formed which contains the unique q-grams from sets S and T. The size of the set V will be denoted by v. If the set V is the empty set, then $v = 0$. The q-gram distance is then equal to $\frac{u}{v}$, if $v \neq 0$ and is equal to 0 otherwise. Note that the q-gram distance will be between 0 and 1 inclusive, where two strings are not similar if they have a q-gram distance of 0 and a q-gram distance of 1 represents a perfect match. Therefore, the closer to 1 the q-gram distance of two strings is, the more likely the strings are to be a match. The q-gram distance can then be multiplied by 100 to get a percentage value for how similar two strings are.

Example 2.12. Let the string $s = STASHA$, string $t = SSTASGA$, and $q = 4$. Then the q-grams of string s are equal to the set $S = \{STAS, TASH, ASHA\}$. And the q-grams of string t are equal to the set $T = \{SSTA, STAS, TASG, ASGA\}$. Then the set U, which contains the q-grams that S and T have in common is equal to $U = \{STAS\}$. There are 6 unique q-grams possible between the two strings and $|U| = 1$. So the strings have a 16.67% probability of being a match.

The python code used to generate the q-gram distance between $STASHA$ and $SSTASGA$ when $q = 4$ is given in the appendix.

It is important to note that character-based similarity metrics are good at finding typographical errors and not at detecting rearrangements of words. Token-based similarity

metrics work to resolve this problem. Token-based similarity metrics include atomic strings, WHIRL, and Q-grams with $tf.idf$.

Chapter 3. Token-Based Similarity Metrics

3.1 Atomic Strings

Atomic strings are sequences of numbers and letters delimited by punctuation characters. Two strings are said to match if one is the prefix of the other or they are equal to each other. In this metric, how similar two fields are to each other is given by their number of matching atomic strings divided by their average number of atomic strings. The algorithm used for matching fields based on atomic strings was formed by Monge and Elkan [23].

3.2 WHIRL

WHIRL computes the similarity of two fields by looking at the cosine similarity between strings s and t, along with the $tf.idf$ weighting scheme [24]. First each string s is separated into words w and each word is assigned a weight $v(w) = log(tf_w + 1) \cdot log(idf_w)$. tf_w is the number of times that w appears in the field and $idf_w = \frac{|D|}{n_w}$, where n_w is the number of records in database D that contain w. If w appears a large number of times in the field, then the $tf.idf$ weight for a word w in a field is high. If w is a lesser used word, then idf_w is high. Then the cosine similarity of string s and t is $\frac{\sum_{j=1}^{|D|} v_s(j) \cdot v_t(j)}{||v_s||_2 \cdot ||v_t||_2}$.

WHIRL works well for a large variety of entries and allows for word order to vary. For example, it sees first name last name as the same as last name, first name. Furthermore, frequent words have a minimal effect on the similarity of two strings since they have a low idf weight. An example of this would be if there was a database that had titles attached to peoples' names. However, WHIRL is not good at finding spelling errors, especially if there are multiple spelling errors in the database. This system was formed by William W. Cohen.

3.3 Q-GRAMS WITH $tf.idf$

Q-grams with $tf.idf$ is an extension of the WHIRL system [25]. This metric works on handling spelling errors by using q-grams as tokens instead of words. This makes it so that spelling errors have a minimal effect on the set of common q-grams for two strings s and t. Furthermore, q-grams with $tf.idf$ is able to handle the insertion and deletion of words in strings without too much difficulty.

Character-based similarity metrics and token-based similarity metrics view a database as being string based. For strings that are written similarly these are good metrics to use. However, some strings in a database may be phonetically similar but are not very similar under character-based or token-based metrics. Phonetic similarity metrics that try to match such strings are Soundex, New York State Identification and Intelligence System (NYSIIS), Oxford Name Compression Algorithm (ONCA), Metaphone, and Double Metaphone.

Chapter 4. Phonetic-Based Similarity Metrics

4.1 Soundex

The most common phonetic similarity metric is Soundex. It is mainly used to match surnames and is widely used in genealogy. This is because Soundex works well for caucasian surnames and surnames of different origins that are not heavily based on vowels, since it ignores vowels. It was developed by Robert C. Russell and Margaret K. Odell [26, 27].

Soundex groups phonetically similar consonants together and assigns those letters a number. Only the first letter in the surname is kept, all W's and H's are ignored, and the remaining letters are changed to numbers, except for vowels which serve as separators. Repeated numbers are then located and consolidated to just one representation of that number. After that the vowels which served as separators are then removed. Lastly, only the letter and first three numbers are kept.

It is worth noting that more than three numbers can be kept. However, it is customary that only the letter and first three numbers are kept. (This is the default setting in the code.) If more or less numbers are desired to be kept or removed, this must be specified when performing Soundex on a given dataset. If there are fewer than three numbers given for a surname, then still keep the first letter and any numbers that are present and fill in the rest of the spaces with zeros.

Example 4.1. Soundex on the surname Simoneit

(i) Keep the first letter S. If there were any W's or H's, they would be ignored at this point. Since there are no instances of W's or H's, move on to step two.

(ii) The following letters get assigned to the following numbers.

B, F, P, V to 1

C, G, J, K, Q, S, X, Z to 2

D, T to 3

L to 4

M, N to 5

R to 6

So Simoneit become SI5O5EI3.

(iii) All the vowels which served as separators are dropped. Thus SI5O5EI3 becomes S553.

(iv) Since S553 has only the first letter and three numbers, it is in the proper form. Meaning that no numbers need to be truncated and no zeros need to be added for padding.

An image of a 1920 U.S. Census index card for a family with the last name Simoneit is given in figure 4.1 [28]. The Soundex form of the family's name (S553) is given in the top left corner of the card.

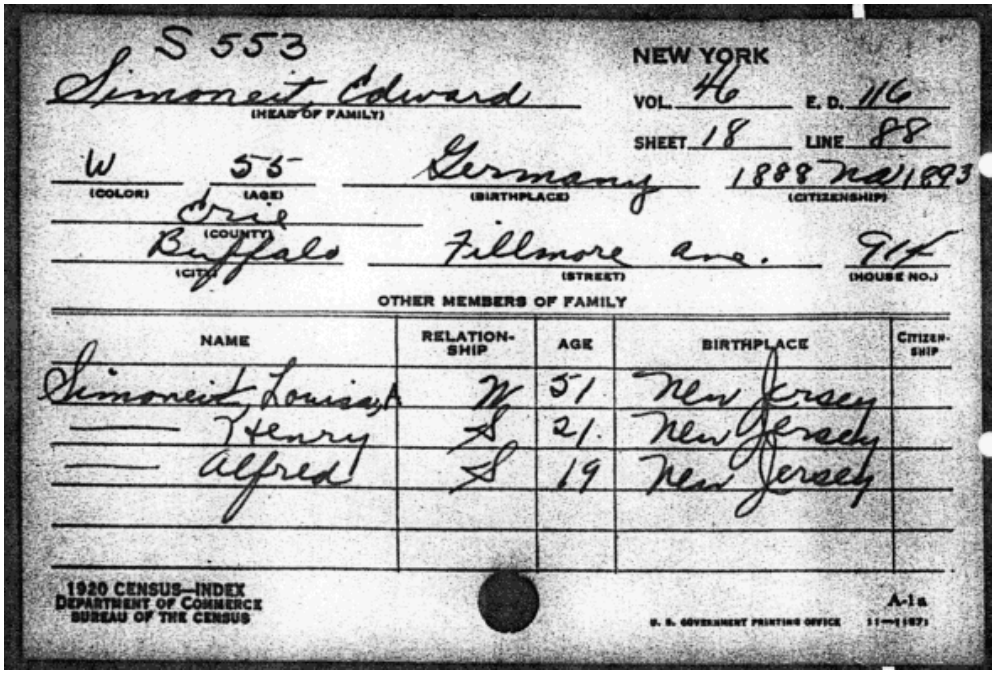


Figure 4.1: Image of a 1920 U.S. Census index card for a family with the last name Simoneit.

Example 4.2. Soundex on the surname Avio

(i) Keep the first letter A. If there were any W's or H's, they would be ignored at this point. Since there are no instances of W's or H's, move on to step two.

(ii) The following letters get assigned to the following numbers.

B, F, P, V to 1

C, G, J, K, Q, S, X, Z to 2

D, T to 3

L to 4

M, N to 5

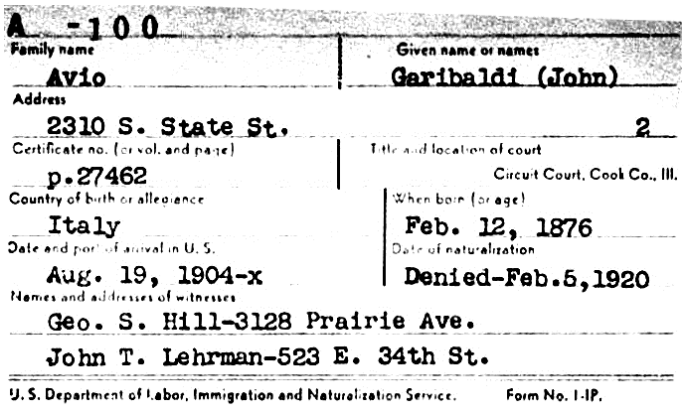
R to 6

So Avio become A1IO.

(iii) All the vowels which served as separators are dropped. Thus A1IO becomes A1.

(iv) Since A1 has only the first letter and one number, then it is not in the proper form. (Meaning that zeros need to be added for padding.) After adding two zeros, A1 becomes A100 and is now in standard Soundex form.

An image of a U.S. Department of Labor, Immigration, and Naturalization Service card for a man with the last name Avio is given in figure 4.2 [29]. The Soundex form of the man's surname (A100) is given in the top left corner of the card.


A -100

Family name: Avio

Given name or names: Garibaldi (John)

Address: 2310 S. State St. 2

Certificate no. (or vol. and page): p.27462

Title and location of court: Circuit Court, Cook Co., Ill.

Country of birth or allegiance: Italy

When born (or age): Feb. 12, 1876

Date and port of arrival in U. S.: Aug. 19, 1904-x

Date of naturalization: Denied-Feb.5,1920

Names and addresses of witnesses: Geo. S. Hill-3128 Prairie Ave.

John T. Lehrman-523 E. 34th St.

U. S. Department of Labor, Immigration and Naturalization Service. Form No. I-IP.

Figure 4.2: Image of a U.S. Department of Labor, Immigration, and Naturalization Service card for a man with the last name Avio.

The python code used to find the Soundex representation of a name is given in the appendix. The Soundex python code was written by Gregory Jorgensen and can also be found at http://code.activestate.com/recipes/52213-soundex-algorithm/.

4.2 New York State Identification and Intelligence System (NYSIIS)

NYSIIS extends upon Soundex and takes into account vowels, changing most of the vowels to A [30]. Furthermore, NYSIIS does not change phonetically similar consonants to the same number like Soundex does, but instead it replaces consonants with other phonetically similar consonants. NYSIIS was invented by Robert Taft and is used by the New York State Division of Criminal Justice Services.

The steps as given in *Name Search Techniques* by Taft are [30]:

(i) Translate the first character or characters of the name to the following:

MAC to MCC

KN to N

K to C

PH and PF to FF

SCH to SSS

(ii) Translate the last character of the name to the following:

EE to Y

IE to Y

DT, RT, RD, NT, ND to D

(iii) The first character of the name becomes equal to the first character of the key.

(iv) Translate the remaining characters in the name by the following rules, moving forward by one character each time.

(a) EV to AF else A, E, I, O, and U to A

(b) Q to G

Z to S

M to N

(c) KN to N else K to C

(d) SCH to SSS

PH to FF

(e) H to previous character, if the previous character or next is not a vowel

(f) W to A, if the previous character is a vowel

(g) Add current to key if current is not the same as the last key character.

(v) If the last character is S, then remove it.

(vi) If the last characters are AY, then replace AY with Y.

(vii) If the last character is A, then remove it.

(viii) Append the translated key to value from step 3 (removed first character).

(ix) If the result is longer than six characters, then truncate it to the first six characters. Note that this step is only needed for true NYSIIS and that some versions use the full key.

Example 4.3. NYSIIS on the surname Simoneit

(i) Since Simoneit does not start with any of these letter combinations, is remains unchanged for the moment.

(ii) Simoneit does not end with any of these letter combinations, so again Simoneit remains unchanged.

(iii) Key = S

(iv) E, I, and O are replaced by A, so SIMONEIT becomes SAMANAT. And M is replaced by N, so SAMANAT becomes SANANAT.

(v) The remaining steps do not apply, so the NYSIIS representation for Simoneit is SANANAT.

Example 4.4. NYSIIS on the name Kristie

(i) The first letter K is changed to C, so KRISTIE becomes CRISTIE.

(ii) The last letters IE are changed to Y, so CRISTIE becomes CRISTY.

(iii) Key = C

(iv) I is replaced by A, so CRISTY becomes CRASTY.

(v) The remaining steps do not apply, so the NYSIIS representation for KRISTIE is CRASTY.

The python code used to find the NYSIIS representation of a name is given in the appendix. The code can also be found at http://utilitymill.com/edit/nysiis.

4.3 Oxford Name Compression Algorithm (ONCA)

ONCA is another extension of Soundex. First ONCA uses a British version of NYSIIS and then those results are run through Soundex [31].

4.4 Metaphone and Double Metaphone

Metaphone is not an extension of Soundex like NYSIIS and ONCA, but is seen as an alternative to Soundex [32]. It uses sixteen consonant sounds that describe a number of sounds used in many words. The consonants are 0BFHJKLMNPRSTWXY, where 0 represents th, X represents sh and ch, and the remaining fourteen consonants represent their standard English pronunciations. Metaphone's advantage over other metrics is that these consonant sounds are used in many English and non-English words. It is important to note that the vowels A, E, I, O, and U are only used if they are at the beginning of the word, otherwise they are dropped.

Double Metaphone is an extension of Metaphone that allows for more than one encoding of names that have multiple possible pronunciations [33]. All possible encodings are tested

when trying to see if names are similar, which allows for more matches to be found. Both Metaphone and Double Metaphone were proposed by Lawrence Philips.

The steps for transforming a word into its approximate phonetic representation are [34]:

(i) Drop duplicate adjacent letters, except C.

(ii) If the word begins with AE, GN, KN, PN, or WR, drop the first letter.

(iii) Drop B if after M at the end of the word.

(iv) C transforms to X if followed by IA or H, unless in the latter case it is part of -SCH-, in which case it transforms to K.
C transforms to S if followed by I, E, or Y.
Otherwise, C transforms to K.

(v) D transforms to J if followed by GE, GI, or GY.
Otherwise, D transforms to T.

(vi) Drop G if followed by H and H is not at the end or before a vowel.
Drop G if followed by N or NED and is at the end.

(vii) G transforms to J if before I, E, or Y, and it is not in GG.
Otherwise G transforms to K.

(viii) Drop H if after a vowel and not before a vowel.

(ix) CK transforms to K.

(x) PH transforms to F.

(xi) Q transforms to K.

(xii) S transforms to X if followed by H, IA, or IO.

(xiii) T transforms to X if followed by IA or IO.

TH transforms to 0.

Drop T if followed by CH.

(xiv) V transforms to F.

(xv) WH transforms to W if at the beginning.

Drop W if not followed by a vowel.

(xvi) X transforms to S if at the beginning.

Otherwise, X transforms to KS.

(xvii) Drop Y if not followed by a vowel.

(xviii) Z transforms to S.

(xix) Drop all vowels other than at the beginning of the word.

Example 4.5. The Double Metaphone representation of Simoneit is SMNT.

Example 4.6. The Double Metaphone representation of Stasha is STX.

The python code used to find the Double Metaphone representation of a name is given in the appendix. The code can also be found at https://github.com/dracos/double-metaphone /blob/master/metaphone.py.

Many methods exist for finding similarities in string-based data. However, there is not much work that has been done to find similarities in numeric data. This is because in general numbers are treated like strings and compared using the similarity metrics discussed previously or a range analysis is run on the numbers to locate numbers with similar values.

Chapter 5. Deterministic and Probabilistic Record Linkage

5.1 Deterministic Record Linkage

Once the data has been prepared, there are two approaches to record linkage: deterministic and probabilistic [35]. Deterministic record linkage is the simpler of the two kinds of record linkage. Rules are coded for what an acceptable match will be. For example, records are joined when entities share the same name and date of birth or when entities have the same social security number. When it works, a deterministic record linkage algorithm is very fast. However, such algorithms can be difficult to tune because all cases that arise in the data source must be covered. Any missing coverage will result in duplicate records being left in, and any incorrect or overzealous rules will cause incorrect linking to occur.

Interesting data sources are usually missing a global key in all or some of their entries. When such a key is missing in a data source and deterministic record linkage is to take place, Wang and Madnick proposed using rules formed by experts that give a set of characteristics that then are able to form a key for each data record [9]. For example, a rule that would give a characteristic that would help form a key would be: if age < 18, then status = student, else status = non-student. The work of Wang and Madnick was expanded upon by Lim, Prabhakar, Srivastava, and Richardson who suggested that the answers to all the rules must always be correct [36]. This makes it so that the rules are not defined ad-hoc, and makes it easier to form a key and join the records. Their work was then further developed by Hernandez and Stolfo who worked to find similarities between records using similarity techniques and string comparison techniques [10].

5.2 Probabilistic Record Linkage

Probabilistic record linkage takes into account a wider range of potential identifiers [37]. Weights are calculated for each identifier based on its believed ability to correctly identify a match or a non-match. These weights are then used to calculate the probability that two

records correspond to the same entity. Pairs that have probabilities above a certain cut off point are considered to be matches, while pairs that have probabilities below another cut off point are considered to be non-matches. Pairs that have probabilities between the two cut off points are said to be possible matches. Possible matches can be linked or not depending on the requirements given. This is a key difference between probabilistic record linkage and deterministic record linkage. Deterministic requires a correct and comprehensive set of rules, while probabilistic can get by with weights, a definition of columns to compare, and some tuning, and can even be set to give an operator the hard ones.

5.3 Notation

Let S and T denote the two tables that we wish to join. Without loss of generality, we can assume that S and T have n fields that can be compared. Let the set M denote the record pairs that represent a match (e.g. they are the same entity) and U denote the record pairs that represent a non-match (e.g. they are different entities). Then for $s \in S$ and $t \in T$, the tuple pair $< s, t >$ is either an element of M or U [11].

Each tuple pair $< s, t >$ is then represented by the vector $x = [x_1, x_2, \ldots, x_n]^T$, where n is the number of fields being compared between S and T and x_i is the level of agreement of the i^{th} field between the records s and t. It is often the case that binary values are used for x_i, where $x_i = 1$ if the records s and t are the same in field i and $x_i = 0$ if the records are not the same in field i. Then x is assigned to be an element of M or U. Another way of looking at x is that it is the input to a decision rule that assigns x to M or U. If x is assumed to be a random vector whose density function is different for M and U and the density functions for M and U are known, then finding duplicate records becomes a Bayesian problem.

5.4 Bayesian Statistics and Record Linkage

Let $x =< s, t >$ where $s \in S$ and $t \in T$. In order to know if $< s, t >$ represents a match or a non-match a decision rule is defined. Defining a decision rule based on probabilities is given by

$$< s, t > \in \begin{cases} M & \text{if } p(M|x) \geq p(U|x) \\ U & \text{otherwise.} \end{cases}$$

Bayes Theorem tells us that $p(M|x) = \frac{p(x|M)p(M)}{p(x)}$ and $p(U|x) = \frac{p(x|U)p(U)}{p(x)}$, provided $p(x) \neq 0$. Then applying Bayes Theorem this is equivalent to

$$< s, t > \in \begin{cases} M & \text{if } l(x) = \frac{p(x|M)}{p(x|U)} \geq \frac{p(U)}{p(M)} \\ U & \text{otherwise} \end{cases}$$

where $l(x)$ is the likelihood ratio. This new form for whether $< s, t > \in M$ or U is called Bayes test for minimum error. Bayes test for minimum error is an optimal classifier, however, it is rarely the case that the distributions of $p(x|M)$ and $p(x|U)$ as well as the priors $p(M)$ and $p(U)$ are known.

This is where Naive Bayes comes into play. With Naive Bayes we can compute the distributions of $p(x|M)$ and $p(x|U)$ by assuming conditional independence and claiming that the probabilities $p(x_i|M)$ and $p(x_j|M)$ along with $p(x_i|U)$ and $p(x_j|U)$ are independent if $i \neq j$. Since independence is being assumed, then we obtain

$$p(x|M) = \prod_{i=1}^{n} p(x_i|M)$$

and

$$p(x|U) = \prod_{i=1}^{n} p(x_i|U).$$

$p(x|M)$ and $p(x|U)$ can also be estimated using the general expectation maximization algorithm when it is not reasonable to assume conditional independence [38]. Winkler proposed

using the general unsupervised expectation maximization algorithm along with the following conditions: more than five percent of the data contains matches, the matching pairs are well-separated from the other classes, the rate of typographical errors is low, there are sufficiently many redundant identifiers to overcome errors in other fields of the record, and the estimates computed under the conditional independence assumption result in good classification performance [39].

Sometimes fields have missing values and Du Bois suggested a method that takes missing values into account so mismatches do not occur [40]. With the previous algorithms x is of length n, where n is the number of fields being compared between the two tables that we wish to join. With Du Bois's algorithm the comparison vector is called x^* and is of length $2n$ such that $x^* = (x_1, x_2, \ldots, x_n, x_1y_1, x_2y_2, \ldots, x_ny_n)$, where $y_i = 1$ if the i^{th} field in both records is present and $y_i = 0$ otherwise. Finding duplicate records is then improved, because mismatches from missing data are ignored. Du Bois also did work with finding the distributions of $p(x_iy_i|M)$ and $p(x_iy_i|U)$ by using a training set of pre-labeled record pairs.

While having a decision rule based on probabilities does work, it is not the best way to create a decision rule [11]. This is because different consequences can occur for misclassifications of x into M or U. Thus it is reasonable to assign a cost c_{ij} to each situation, where c_{ij} is the cost of having x in class i when x should really be in class j. This then leads us to the expected costs of x being in M and U given by $r_M(x)$ and $r_U(x)$, where

$$r_M(x) = c_{MM} \cdot p(M|x) + c_{MU} \cdot p(U|x)$$

$$r_U(x) = c_{UM} \cdot p(M|x) + c_{UU} \cdot p(U|x).$$

Then the decision rule for $x \in M$ is

$$< s, t > \in \begin{cases} M & \text{if } r_M(x) < r_U(x) \\ U & \text{otherwise.} \end{cases}$$

And the minimum cost decision rule is then

$$< s,t >\in \begin{cases} M & \text{if } l(x) > \frac{(c_{MU}-c_{UU})\cdot p(U)}{(c_{UM}-c_{MM})\cdot p(M)} \\ U & \text{otherwise.} \end{cases}$$

Note that the minimum error and minimum cost decision rules are the same when $c_{UM} - c_{MM} = c_{MU} - c_{UU}$ [41].

When the likelihood ratio is close to the threshold the error of a decision is high [41]. To counteract this, Fellegi and Sunter proposed adding a reject class R to M and U [5]. R contains record pairs that could or could not be a match. They are assigned to R so that they may be looked over manually to determine if they are matches or not. The reject region is defined by setting bounds on the conditional error for M and U. Whereas the reject probability is the probability of sending a record pair to an expert for review.

ALIAS is a learning-based duplicate records detection algorithm that implements a reject region in order to reduce the size of the training set [8]. For record pairs that certainly are a match or not a match they are assigned to M or U like previous algorithms do. However, when a match is uncertain ALIAS requires that an operator labels the record pair as a match or not. ALIAS's way of handling uncertain pairs is similar to Fellegi and Sunter's algorithm where uncertain record pairs are put into a reject region for a person to classify later. The ALIAS algorithm was invented by Sarawagi and Bhamidipaty. Sarawagi and Bhamidipaty's work was further expanded upon by Tejada, Knoblock, and Minton who used decision trees to help the algorithm learn the rules for matching records with more than one field [42].

Chapter 6. Efficiency in Finding Duplicate Records

The metrics and probabilities that we were looking at previously focused on how good of a comparison we were making. However, improving the quality of the comparison is just one part of the process We also have to see how effective the process is. For example, if we wanted to compare every record in table S with every record in table T it would require $|S| \cdot |T|$ comparisons, which is very expensive. Another point to keep in mind is that a single record generally has multiple fields. Comparing more fields proportionally increases the cost of each comparison. The final cost is more like $|S| \cdot |T| \cdot n$, where n is the number of fields compared.

A technique used to reduce the number of comparisons made is blocking [11]. Blocking is the process of subdividing files into a set of mutually exclusive subsets, referred to as blocks, where there are no matches between the blocks. For example, to form name blocks programs such as Soundex, NYSIIS, ONCA, Metaphone, or Double Metaphone can be used. These programs should be run multiple times on different blocking fields in order to decrease the rate of incorrect mismatches.

Another technique called the sorted neighborhood approach proposed by Hernandez and Stolfo consists of three steps: create key, sort data, and merge [10]. Sorted neighborhood is greatly dependent on the key that is chosen and assumes that duplicate records will be near each other in the sorted list and then compared in the merge step. This led Hernandez and Stolfo to expand their technique and create a multi-pass approach. This makes it so multiple keys can be created and run to eliminate as many duplicate records as possible.

Using transitivity in finding duplicate records was implemented by Monge and Elkan [43]. Transitivity in duplicate records means that if record a is the same as record b and record b is the same as record c, then record a is the same as record c. This then led Monge and Elkan to create a union-find structure where duplicate records are merged into a cluster and only a representative of the cluster is kept for future comparisons.

McCallum, Nigam, and Ungar suggested finding duplicate records by using two compar-

isons techniques, which is supposed to led to better results for finding duplicate records [44]. The first comparison groups records into overlapping clusters called canopies. The canopies are supposed to be a loose comparison framework that is formed quickly. Then a second more detailed comparison is done within the canopies. McCallum, Nigam, and Ungar's work with using canopies was extended by Cohen and Richman [45]. Their work was then again added upon by Gravano, Ipeirotis, Jagadish, Koudas, Muthukrishnan, and Srivastava [46]. Similarly Chaudhuri, Ganjam, Ganti, and Motwan extended the work of record linkage using canopies by suggesting the use of an indexable canopy [47]. Lastly Baxter, Christen, and Churches did work comparing the use of blocking and canopies [48].

Chapter 7. Tools Available to Find Duplicate Records

The Freely Extensible Biomedical Record Linkage (Febrl) system is an open source data cleaning toolkit [49]. It standardizes the data records using hidden-Markov models and then finds the duplicate records using edit distance, Jaro distance, or q-gram distance. In order to help with finding similar names Febrl uses phonetic similarity metrics Soundex, NYSIIS, and Double Metaphone. These phonetic similarity metrics use a reversed version of the name string in order to avoid the problem of errors in the first letter of a name.

TAILOR is a toolbox that allows users to try different duplicate record detection techniques on the same dataset [50]. Different techniques have varying levels of accuracy and completeness for the same dataset. TAILOR gives the user the ability to compare the different techniques against each other by giving the user the statistics of how effective each method was for finding duplicate records.

To find similar strings in two data records WHIRL uses a $tf.idf$ token-based similarity metric [51]. However, this toolkit is only available for academic and research proposes. Toolkits that are similar to WHIRL that are available to the public include The Flamingo Project, WizSame by WizSoft, and SoftTF.IDF.

BigMatch uses blocking techniques discussed earlier to find duplicate records in very large datasets [52]. It is meant to serve as the first step in a two part process that would then run a more complex algorithm to find duplicate records. It is interesting to note that the US Census Bureau uses BigMatch to detect duplicate records in its data.

Chapter 8. The Algorithm

8.1 Traditional Record Linkage

Traditionally record linkage algorithms determine if two records are a match or not at or above a given tolerance level. This tolerance level is provided by the user and allows the user to determine at what threshold they want records to be matched or not. When the algorithm is than run on two databases it returns true or false for each record pair. Returning true means that that record pair was a match at or above that tolerance level, while returning false means that it was not. From there the number of true positives, false positives, true negatives, and false negatives can be calculated for known data pairs.

Running each of the algorithms repeatedly at different tolerance levels is expensive in terms of time and machine use. Therefore, I took a different approach with finding when records matched or not. This new approach made it so that each of the algorithms only had to be run once per given set of algorithm parameters and algorithm threshold, but output the results for multiple tolerance levels.

8.2 The New Approach

There are two parts to the new testing system; a frontend that the user controls and a backend that runs the actual merge and updates. The frontend also keeps track of all the datasets, stats, and outputs of the mergers and updates that are to be run. The worker is given instructions about what datasets to use and what operations to do on them. When the worker is handling a job: it loads the data, runs the merge or update, then uploads the results. Of these three steps the key one is actually running the merge or update. It is also the most interesting. The way that this is done is different depending if it is a merge or an update job, although the comparison code is the same either way.

For merge jobs, each row in the first dataset is compared with each row in the second dataset, like a Cartesian product. For an update job the matcher iterates over each row in

the dataset and compares it with all previous rows. So that every row is compared with every other row, exactly once. The matcher is also able to find duplicate entries in a single dataset. We do not look at examples of this type of datasets here, but it is a possiblity.

When a character-based similarity metric is used the parameters passed in indicate whether each record pair being compared is a match or not. When a phonetic-based similarity metric is used it compares the outputs to see if they are the same. If the two strings are identical, then those fields are marked as a match.

When two rows are being compared the new algorithm looks at the ratio of the number of matched columns to the number of possible matched columns for a given entry. The matcher keeps a running total of how many rows matched at each tolerance level. The comparison function returns the ratio of matched columns over possible matched columns. The matcher is able to tell whether any two rows are supposed to match because it is given a lookup table. The lookup table contains the row ids (i.e. column name *rec_id*) that should match with any given row.

8.3 When a Field is Missing Information

With some records, fields are missing because the information was not provided to begin with or fields were dropped due to a data entry error in the duplicate record. In either case if data is missing from a given field or two fields that are to be compared against each other, the algorithm skips over that field or fields and does not count it as a match or non-match, nor does it count towards the total possible matched columns. Below is an example where an original database entry and its duplicate entry are missing information.

rec_id	given_name	surname	street_number	address_1	address_2	suburb	postcode	state	date_of_birth	age	phone_number	soc_sec_id
600	caitlin	bruty	548	clive steele avenue		port augusta	2322	qld	19921019	31		2913484
601	caitlin	bruty	548	clive steele avenue		port augusta	2322	qld	19912019	31		2913484

Table 8.1: Example of an original database entry with rec_id 600 with its duplicate database entry with rec_id 601 where both database entries are missing data in more than one field.

In the above example both the original database entry and its duplicate are missing data

in columns *address_2* and *phone_number*. The duplicate entry differs from the original entry in the tenth column *date_of_birth* where the original date of birth provided is 19921019 and the duplicate date of birth provided is 19912019. When calculating if these two records are a match only the ten columns that have data both in the original entry and in the duplicate entry would be taken into account.

8.4 The Frontend of the Testing System

The frontend of the matching algorithm uses the following steps to run a merge or updating job.

(i) Name the merge or updating job

(ii) Select dataset a

(iii) Select dataset b

(iv) Select the algorithm to use for comparing records.

The algorithms that are available to choose from are:

(a) Edit distance

(b) Smith-Waterman distance

(c) Jaro distance

(d) q-gram distance

(e) Soundex

(f) NYSIIS

(g) Double Metaphone

(v) Enter the algorithm parameters and threshold. For

(a) edit distance

i. parameters: none

ii. threshold: upper bound for match (a perfect match is 0)

(b) Smith-Waterman distance

i. parameters: G, lower bound, upper bound (inclusive)

ii. threshold: has no effect

(c) Jaro distance

i. parameters: lower bound, upper bound (inclusive)

ii. threshold: has no effect

(d) q-gram distance

i. parameters: q, the q-gram size to use

ii. threshold: lower bound for match, the upper bound is automatically 1

(e) Soundex

i. parameters: the maximum length of the final Soundex string that is returned. The option defaults to 4, which is the classical behavior of the algorithm.

ii. threshold: has no effect

(f) NYSIIS

i. parameters: none

ii. threshold: has no effect

(g) Double Metaphone

i. parameters: none

ii. threshold: has no effect

(vi) Enter in the name of the columns that are to be compared. Note that the version of Soundex that we are using ignores non-alphabetic characters. NYSIIS replaces numbers with a blank space and special characters with nothing. Double Metaphone does not handle numbers and special characters and it is run under the assumption that the input is letters only.

(vii) Once all the options are selected, the job is saved. (At this time the status of the job is "Ready".)

(viii) A new page will appear that shows all the information for the given job and gives the user a chance to make any changes before running the job.

(ix) If changes need to be made the edit button is selected. After the changes have been made, they are saved and ready to be reviewed again.

(x) Once the job has been reviewed and requires no additional edits, the run button is pushed.

(xi) The job is then placed in the queue and waits for an open worker to pick it up. (At this time the status of the job is changed to "Queued".)

(xii) Once a worker has picked up the job, it is removed from the queue and the job's running status is updated every ten seconds. (At this time the status of the job is changed to "Running" and next to the word Running a percentage of the number of comparisons made over the total number of comparison required is given.)

(xiii) After the job has finished running the data is uploaded. (At this time the status of the job is changed to "Uploading" and next to the word Uploading a percentage appears with how much of the final results have been uploaded.)

(xiv) Once the final results have been uploaded the job is complete. The output, json, and csv files can be downloaded and the graphs may be viewed. (At this time the status of the job is changed to "Complete".)

(xv) If for any reason the job fails the worker will stop working on it and delete the job before picking up a new job from the queue. (At this time the status of the job is changed to "Failed".)

Chapter 9. Performance Measures

We measure how well a record linkage algorithm performs by looking at the following metrics [53]:

(i) True positives, denoted by n_m

The number of record pairs correctly linked.

(ii) False positives (Type I error), denoted by n_{fp}

The number of record pairs incorrectly linked.

(iii) True negatives, denoted by n_u

The number of record pairs correctly not linked.

(iv) False negatives (Type II error), denoted by n_{fn}

The number of record pairs incorrectly not linked.

(v) The total number of matched record pairs, denoted by N_m and equal to $n_m + n_{fn}$.

(vi) The total number of non-matched record pairs, denoted by N_u and equal to $n_u + n_{fp}$.

(vii) Sensitivity, denoted by $\frac{n_m}{N_m}$

The number of record pairs correctly linked divided by the total number of matched record pairs.

Note that sensitivity measures the percentage of correctly classified record matches.

It is equivalent to recall.

(viii) Specificity, denoted by $\frac{n_u}{N_u}$

The number of record pairs correctly not linked divided by the total number of non-matched record pairs.

Note that specificity measures the percentage of correctly classified non-matches.

(ix) Match rate, denoted by $\frac{n_m+n_{fp}}{N_m}$

The total number of linked record pairs divided by the total number of matched record pairs.

(x) Positive predictive value (ppv), denoted by $\frac{n_m}{n_m+n_{fp}}$

The number of correctly linked record pairs divided by the total number of linked record pairs.

This is equivalent to precision.

Additional performance criteria can be made in terms of time taken and the number of records requiring manual review [53]. With time taken this is usually dominated by the number of record comparisons being performed. If there are a large number of records that need to be compared, more time will be needed to make the comparisons than if a small number of records needed to be compared. The number of records requiring manual review is important because reviewing records is time-consuming (humans are slow), expensive (humans are expensive), and error prone (humans are inconsistent).

9.1 Receiver Operating Characteristic (ROC) Curve

A Receiver Operating Characteristic (ROC) curve is a graphical plot that shows how well a classification system performs at different thresholds. A ROC curve is formed by plotting sensitivity $= \frac{n_m}{N_m}$ on the y-axis against 1− specificity $= 1 - \frac{n_u}{N_u} = \frac{n_{fp}}{N_u}$ on the x-axis. The ratio of the number of true positives to the number of true positives plus the number of false negatives is called the true positive rate (TPR), and the ratio of the number of false positives to the number of false positives plus the number of true negatives is called the false positive rate (FPR).

A ROC curve is also known as a relative operating characteristic curve. This is because it compares TPR (on the y-axis) and FPR (on the x-axis), which are two operating characteristics, as the threshold changes. Since TPR = sensitivity and FPR = 1 - specificity

this means that we are looking at a trade off between true positives (sensitivity) and false positives (specificity). If an ideal prediction method was used then we would get a point at (0,1) on the ROC curve. Having the curve pass through (0,1) means that there were no false negatives and the method had one hundred percent sensitivity. Being at the point (0,1) also means that there were no false positives and that the method had one hundred percent specificity. Obtaining this point also means that a perfect classification was reached. Perfect classification is not always possible, but we do want to stay above the line of no discrimination. The line of no discrimination is the diagonal line going from the bottom left of the graph to the top right corner of the graph.

The ROC curve also allows us to select an optimal model to classify the data without introducing a cost parameter. Later on we will introduce an economic model and cost parameters that affect what the optimal cutoff point will be. Until then, we will examine ROC curves without cost parameters.

9.2 Applications of ROC Curves

An example of when a ROC curve would be used is for determining the effectiveness of a medical test. Sensitivity would be the percentage of people who are sick that are correctly identified as being sick and specificity would be the percentage of people who are not sick who are correctly identified as being healthy. More formally, sensitivity equals the probability of a positive test, given that the person is sick and specificity equals the probability of a negative test, given that the person was healthy. Thus if the medical test was one hundred percent sensitive, then all the people who were sick were correctly identified as being sick. If the medical test was one hundred percent specific, then all the people who were healthy were not identified as being sick.

Example 9.1. A medical test is given to a group of 5000 individuals. Of the 5000 people that have the test administered to them only 100 of them are actually sick. The table below shows the number of false negatives, false positives, true negatives, and true positives

obtained. From there we are able to calculate the sensitivity and specificity of the test.

level	comp_specificity	false_negatives	false_positives	sensitivity	specificity	total_people_tested	total_sick	total_not_sick	true_negatives	true_positives
0	1	0	4900	1	0	5000	100	4900	0	100
0.083333	0.860408	0	4216	1	0.139592	5000	100	4900	684	100
0.166667	0.851837	0	4174	1	0.148163	5000	100	4900	726	100
0.250000	0.688571	1	3374	0.990000	0.311429	5000	100	4900	1526	99
0.333333	0.419388	4	2055	0.960000	0.580612	5000	100	4900	2845	96
0.416667	0.197347	10	967	0.900000	0.802653	5000	100	4900	3933	90
0.500000	0.075306	31	369	0.690000	0.924694	5000	100	4900	4531	69
0.583333	0.017755	56	87	0.440000	0.982245	5000	100	4900	4813	44
0.666667	0.001633	80	8	0.200000	0.998367	5000	100	4900	4892	20
0.750000	0.000204	94	1	0.060000	0.999796	5000	100	4900	4899	6

Table 9.1: Stats for a medical test.

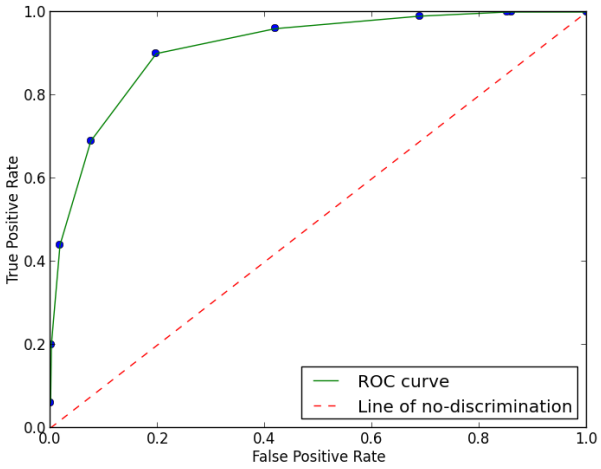


Figure 9.1: Example of a ROC curve with medical test data.

From the ROC curve we see that this particular medical test is not very sensitive nor is it very specific in determining who is sick and who is not sick. This is evidenced by the fact that as the number of false positives decreased so do the number of true positives. The test does perform better than just a random guess, as the ROC curve stays above the line of no-discrimination.

In the lower range of tolerance levels from 0 to $0.41\bar{6}$ the number of false positives is quite high. In this range the number of true positives decreases from 100 to 90 while the number of false positives decreases from 4900 to 967. The next tolerance level of 0.5 serves as our tipping point in that after this level the number of true positives and false positives rapidly decrease. This test might not be very sensitive nor specific, but it could be a quick

and inexpensive test that is run as the first part of a multi-stage testing sequence. This is because if this test returns the results that you are not sick, you are most likely not sick. However, if this test returns the results that you are sick, you may or may not be sick. A second more through and expensive test could then be run just on those that have positives test results.

Example 9.2. A medical test is given to a group of 5000 individuals. Of the 5000 people that have the test administered to them only 50 of them are actually sick. The table below shows the number of false negatives, false positives, true negatives, and true positives obtained. From there we are able to calculate the sensitivity and specificity of the test.

level	comp_specificity	false_negatives	false_positives	sensitivity	specificity	total_people_tested	total_sick	total_not_sick	true_negatives	true_positives
0	1	0	4950	1	0	5000	50	4950	0	50
0.083333	0.954949	0	4727	1	0.045051	5000	50	4950	223	50
0.166667	0.820404	0	4061	1	0.179596	5000	50	4950	889	50
0.250000	0.713333	0	3531	1	0.286667	5000	50	4950	1419	50
0.333333	0.425657	0	2107	1	0.574343	5000	50	4950	2843	50
0.416667	0.312525	0	1547	1	0.687475	5000	50	4950	3403	50
0.500000	0.067475	0	334	1	0.932525	5000	50	4950	4616	50
0.583333	0.011313	0	56	1	0.988687	5000	50	4950	4894	50
0.666667	0.000808	0	4	1	0.999192	5000	50	4950	4946	50
0.916667	0	0	0	1	1	5000	50	4950	4950	50
1	0	11	0	0.780000	1	5000	50	4950	4950	39

Table 9.2: Stats for a medical test.

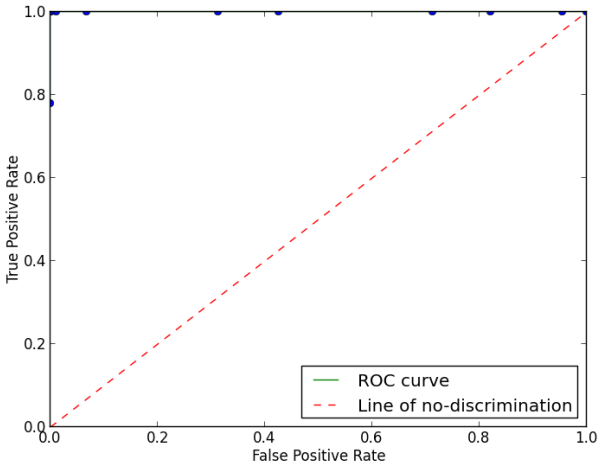


Figure 9.2: Example of a ROC curve with medical test data.

From the ROC curve we see that this medical test is very sensitive and very specific in determining who is sick and who is not sick. This is evidenced by the fact that there is a

point were there are no false positives and no false negatives. We can also see this visually by observing that the ROC curve goes through the perfect classification point at (0,1).

From the stats table for the medical test we are able to see that for levels 0 through $0.91\bar{6}$ there are no false negatives. So if an individual takes the test and it comes back that they are not sick, then they are most likely not sick. However, the test does have a high false positive rate when the tolerance level is low. It is only at the level $0.91\bar{6}$ that the test has perfect classification. After we go over that threshold false negatives occur. Since this test performs so well for determining who is not sick at almost all tolerance levels it would be recommended for widespread use.

Chapter 10. Merging Two Databases

10.1 Examples of Applications of Record Linkage

Some applications of record linkage can be found in the process of maintaining insurance policy, census records, political party membership, and mass marketing mailing list databases. Most health insurance companies have an open enrollment period every year, during which current members can retain, change, or drop their current insurance policy. New members can also join during this time. After this open enrollment period has ended, the company needs to update its database. Ideally after the merge takes place there will only be one policy in the database for each individual or family. The presence of duplicate records could cause errors when filing an individual's claim after they have gone to the doctor or could cause a person to be told by their doctor that they have to pay out of pocket for their visit or at a higher rate because the doctor appears to be out of network due to the error in the insurance company's database.

It is also important to update the census record database correctly. Census records have an impact on electoral districts, therefore, it is extremely important that when the new census records come in every ten years that they are carefully merged with the old database and no duplicate entries are formed. This can be hard to do since not everyone in the United States has a unique name and some individuals do not have a social security number.

Because political parties often contact people to solicit financial donations and other political participation, they must be particularly careful when updating their databases. If one of their members changed political parties, names, address, or phone number and this information was not updated in the database and the member was contacted for a donation, the likelihood of the person giving a donation greatly decreases. This cost is a significant loss in comparison to the cost the party paid for making the phone call.

A final example of where record linkage occurs is with mass marketing mailing list databases, in particular coupon mailers. This example is interesting because, incorrect

records do not lead to huge losses, since if a company spells an individual's name wrong or mails to the wrong address the recipient may look at the coupon mailer despite the mistake. So in this case it is okay to be wrong some of the time, because the company is still reaching possible clients and can still make money even if they did not reach the original intended recipient.

If we then compare the example of mass marketing mailing list databases to that of the political party an interesting insight can be made: Sometimes it pays to be right, but it is okay if the company does not reach everyone as long as who they reach is truly the right person, while other times it is okay to be wrong about which records are linked together as long as they reach as many people as possible. This is because different applications of record linkage are sensitive to different costs. Later on in Chapter 12 we will look at an economic model that examines the trade offs between sensitivity and specificity.

10.2 Background Information on the Datasets

The column names of the data in the databases are *rec_id*, *given_name*, *surname*, *street_number*, *address_1*, *address_2*, *suburb*, *postcode*, *state*, *date_of_birth*, *age*, *phone_number*, and *soc_sec_id*. The columns *rec_id*, *street_number*, *postcode*, *date_of_birth*, *age*, *phone_number*, and *soc_sec_id* contain only numbers, while the columns *given_name*, *surname*, *address_1*, *suburb*, and *state* contain only letters. The column *address_2* is the only column to contain both numbers and letters.

Knowing what kind of data is in each column is important because not all of the similarity metrics handle letters, numbers, and special characters in the same way. For example edit distance, Smith-Waterman distance, Jaro Distance, and q-gram distance treat letters, numbers, and special characters the same. The implementation of Soundex that we are using ignores all non-alphabetic characters. NYSIIS replaces digits with a blank space and special characters with nothing. Double Metaphone does not handle numbers and special characters. The algorithm is run under the assumption that the input string is a single word

or name. If an algorithm that cannot handle numbers or special characters is given a column that contains numbers or special characters, the job will give poor or incorrect results.

A duplicate database was generated from an original database for testing purposes. This way we know what records should match and which ones should not. This also allows us to be able to control how many duplicate entries there are and what kinds of mutations can be made to the duplicate entries.

The duplicate entries were formed by making one change in one of the fields from the original entry. A change can be: a character inserted, a character deleted, a character changed, the transposition of two characters, the reordering of two words (example: First Street becomes Street First), the use of a nickname, and the dropping of a field. Examples of an original entry and its duplicate entry are given below.

rec_id	given_name	surname	street_number	address_1	address_2	suburb	postcode	state	date_of_birth	age	phone_number	soc_sec_id
60	mikaela	nes	36	beirne street		raby bay	2160	nt	19990509	26	08 73958146	1057852
61	mikaela	nrs	36	beirne street		raby bay	2160	nt	19990509	26	08 73958146	1057852

Table 10.1: Example of an original database entry with rec_id 60 with its duplicate database entry with rec_id 61.

Notice that in the above example the difference between the original entry and the duplicate entry is in the third column *surname*. In the original entry the surname is nes, whereas in the duplicate entry the surname is spelled nrs. The letters e and r are right next to each other on the keyboard, so an error like this is a simple mistake to make.

rec_id	given_name	surname	street_number	address_1	address_2	suburb	postcode	state	date_of_birth	age	phone_number	soc_sec_id
280	max	davis	25	mimosa close	loxley road	croydon	2170	nsw	19780307	36	08 94579332	4928485
281	max	davis	25	mimosa close	loxley road	croydon	2170	nsw	19780377	36	08 94579332	4928485

Table 10.2: Example of an original database entry with rec_id 280 with its duplicate database entry with rec_id 281.

In the above example the difference between the original entry and the duplicate entry is in the tenth column *date_of_birth*. In the original entry the date of birth is 19780307, and in the duplicate entry the date of birth provided is 19780377. The numbers 0 and 7 are not close to each other on the keyboard, so an error like this is harder to justify.

To examine the problem of merging two databases together, I used the following databases:

(i) 500 duplicates with 500 originals

(ii) 1250 duplicates with 1250 originals

(iii) 2500 duplicates with 2500 originals

(iv) 5000 duplicates with 5000 originals

Every original entry in database one has a single corresponding duplicate entry in database two. Therefore, there can be at most 500 true matches obtained when 500 duplicates are merged with 500 originals, 1250 true matches obtained when 1250 duplicates are merged with 1250 originals, 2500 true matches obtained when 2500 duplicates are merged with 2500 originals, and 5000 true matches obtained when 5000 duplicates are merged with 5000 originals.

The similarity metrics that I used when evaluating how similar two strings are during the process of merging databases were:

(i) edit distance

(ii) Jaro distance

(iii) Smith-Waterman distance

(iv) q-gram distance

(v) Soundex

(vi) NYSIIS

(vii) Double Metaphone

Recall that edit distance, Jaro distance, Smith-Waterman distance, and q-gram distance are character-based similarity metrics so they can run comparisons on any given column,

whereas Soundex, NYSIIS, and Double Metaphone are phonetic-based similarity metrics and are designed to make comparisons between strings that contain only letters. Therefore, they will only be given the columns *given_name*, *surname*, *address_1*, *suburb*, and *state*, to run comparisons on.

10.3 Merge 500 Duplicates with 500 Originals

Example 10.1. merge 500 duplicates with 500 originals using edit distance

dataset 1:	dataset_A_500_original
dataset 2:	dataset_A_500_duplicate
method:	edit distance
algorithm threshold:	2.0 (upper bound for match, note that a perfect match is 0)
final dataset:	dataset_A_500_org_500_dup_ed

level	comp_specificity	false_negatives	false_positives	sensitivity	specificity	total_matches	total_non_matches	true_negatives	true_positives
0	1	0	249500	1	0	500	249500	0	500
0.083333	0.988685	0	246677	1	0.011315	500	249500	2823	500
0.166667	0.876196	0	218611	1	0.123804	500	249500	30889	500
0.250000	0.562910	0	140446	1	0.437090	500	249500	109054	500
0.333333	0.227082	0	56657	1	0.772918	500	249500	192843	500
0.416667	0.062112	0	15497	1	0.937888	500	249500	234003	500
0.500000	0.013375	0	3337	1	0.986625	500	249500	246163	500
0.583333	0.002244	0	560	1	0.997756	500	249500	248940	500
0.666667	0.000140	0	35	1	0.999860	500	249500	249465	500
0.916667	0	0	0	1	1	500	249500	249500	500
1	0	107	0	0.786000	1	500	249500	249500	393

Runtime	Comparisons	Comparisons per Second
81 seconds	250000	3086.4198

Table 10.3: Stats for dataset_A_500_org_500_dup_ed.

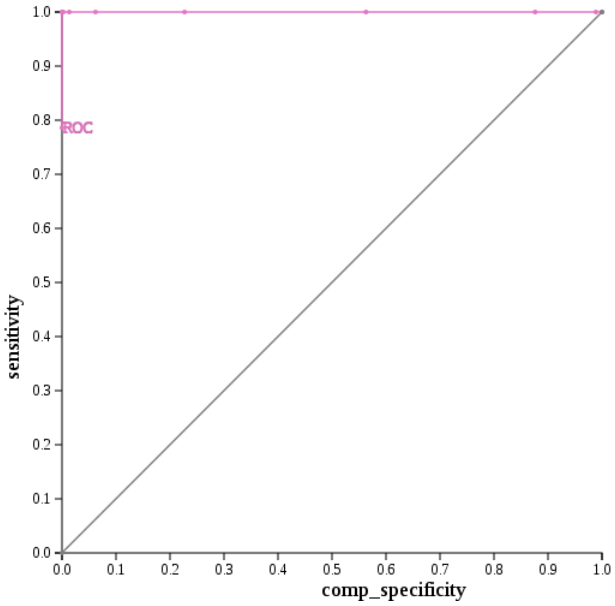


Figure 10.1: ROC curve for dataset_A_500_org_500_dup_ed.

Example 10.2. merge 500 duplicates with 500 originals using Soundex

dataset 1: dataset_A_500_original

dataset 2: dataset_A_500_duplicate

method: Soundex

final dataset: dataset_A_500_org_500_dup_soundex

level	comp_specificity	false_negatives	false_positives	sensitivity	specificity	total_matches	total_non_matches	true_negatives	true_positives
0	1	0	249500	1	0	500	249500	0	500
0.200000	0.197210	0	49204	1	0.802790	500	249500	200296	500
0.400000	0.002569	0	641	1	0.997431	500	249500	248859	500
0.600000	0.000020	0	5	1	0.999980	500	249500	249495	500
0.800000	0	0	0	1	1	500	249500	249500	500
1	0	109	0	0.782000	1	500	249500	249500	391

Runtime	Comparisons	Comparisons per Second
13 seconds	250000	19230.7692

Table 10.4: Stats for dataset_A_500_org_500_dup_soundex.

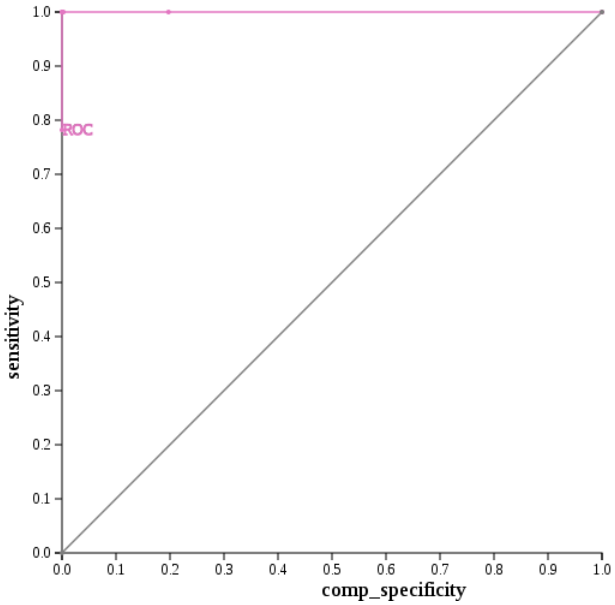


Figure 10.2: ROC curve for dataset_A_500_org_500_dup_soundex.

For dataset_A_500_original and dataset_A_500_duplicate the similarity metrics I used were edit distance and Soundex. Their ROC curves when run on these datasets are almost identical and both of them pass through the optimal point (0,1). These results are not surprising since the algorithm threshold for edit distance was set at 2.0. Recall that the algorithm threshold for edit distance is the upper bound for a match and that a perfect match has

an algorithm threshold of 0. Since a duplicate is formed by making one mutation in one of the fields for each entry, then most duplicate entries should match with the corresponding original entry at an algorithm threshold of 2.0. Duplicate entries that are more difficult to match are those were nicknames have replaced given names, words have been reordered, or fields have been dropped.

Edit distance's cut off level is $0.91\bar{6}$, whereas Soundex is $0.8\bar{3}$. Having similar cut off levels for edit distance and Soundex is to be expected. This is because given the conditions for these two example datasets, both should be able to find the correct matches easily except in the cases of nickname replacement, reordering of words, and dropped fields. This is because edit distance does not work well when strings have been truncated and neither similarity metric can tell that two words are the same if their order has been switched.

Additionally, Soundex has a harder time performing well if fields are dropped because it has less fields to compare to begin with. Of the twelve columns only five of them contain fields that only have letters in them – edit distance has two and a half times the number of columns to work with then Soundex does. Thus when a field is dropped it has a bigger impact on Soundex then it does on edit distance. In the end Soundex was able to get the maximum number of true positives without any false negatives at a slightly lower level and in less runtime then edit distance. Thus the phonetic-based similarity metric performed better at merging dataset_A_500_original and dataset_A_500_duplicate together.

10.4 Merge 1250 Duplicates with 1250 Originals

Example 10.3. merge 1250 duplicates with 1250 originals using Jaro distance

dataset 1:	dataset_A_1250_original
dataset 2:	dataset_A_1250_duplicate
method:	Jaro distance
algorithm parameters:	0.75, 1.0 (lower bound, upper bound inclusive)
final dataset:	dataset_A_1250_org_1250_dup_jd

level	comp_specificity	false_negatives	false_positives	sensitivity	specificity	total_matches	total_non_matches	true_negatives	true_positives
0	1	0	1561251	1	0	1249	1561251	0	1249
0.083333	0.505374	0	789015	1	0.494626	1249	1561251	772236	1249
0.166667	0.124357	0	194152	1	0.875643	1249	1561251	1367099	1249
0.250000	0.016000	0	24980	1	0.984000	1249	1561251	1536271	1249
0.333333	0.001179	0	1840	1	0.998821	1249	1561251	1559411	1249
0.416667	0.000056	0	88	1	0.999944	1249	1561251	1561163	1249
0.500000	0.000002	0	3	1	0.999998	1249	1561251	1561248	1249
0.583333	0.000001	0	1	1	0.999999	1249	1561251	1561250	1249
0.666667	0.000001	2	1	0.998399	0.999999	1249	1561251	1561250	1247
0.750000	0.000001	28	1	0.977582	0.999999	1249	1561251	1561250	1221
0.833333	0.000001	144	1	0.884708	0.999999	1249	1561251	1561250	1105
0.916667	0	531	0	0.574860	1	1249	1561251	1561251	718
1	0	1050	0	0.159327	1	1249	1561251	1561251	199

Runtime	Comparisons	Comparisons per Second
173 seconds	1562500	9031.7919

Table 10.5: Stats for dataset_A_1250_org_1250_dup_jd.

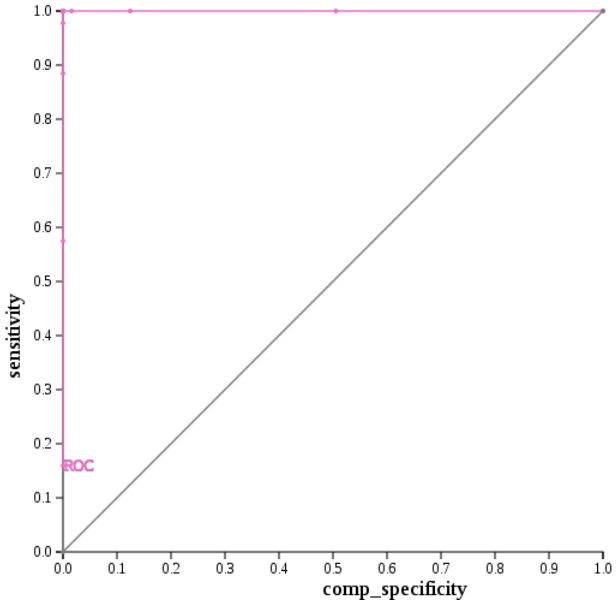


Figure 10.3: ROC curve for dataset_A_1250_org_1250_dup_jd.

Example 10.4. merge 1250 duplicates with 1250 originals using NYSIIS

dataset 1: dataset_A_1250_original

dataset 2: dataset_A_1250_duplicate

method: NYSIIS

final dataset: dataset_A_1250_org_1250_dup_nysiis

level	comp_specificity	false_negatives	false_positives	sensitivity	specificity	total_matches	total_non_matches	true_negatives	true_positives
0	1	0	1561251	1	0	1249	1561251	0	1249
0.200000	0.202827	0	316664	1	0.797173	1249	1561251	1244587	1249
0.400000	0.001896	0	2960	1	0.998104	1249	1561251	1558291	1249
0.600000	0.000005	0	8	1	0.999995	1249	1561251	1561243	1249
0.800000	0.000001	0	1	1	0.999999	1249	1561251	1561250	1249
1	0.000001	500	1	0.599680	0.999999	1249	1561251	1561250	749

Runtime	Comparisons	Comparisons per Second
71 seconds	1562500	22007.0423

Table 10.6: Stats for dataset_A_1250_org_1250_dup_nysiis.

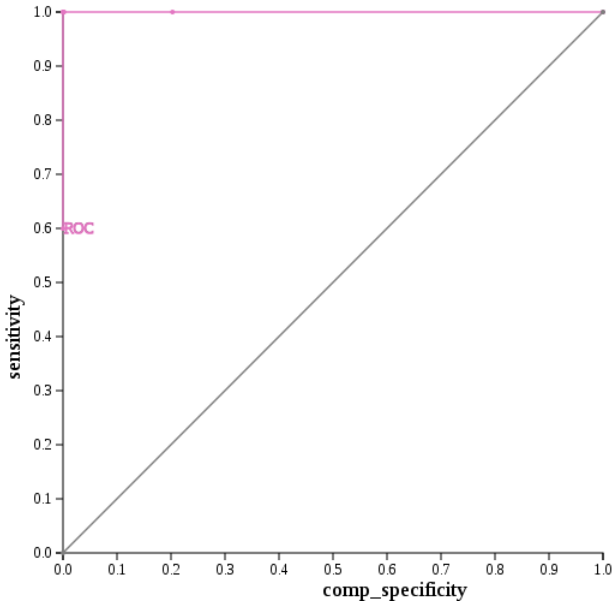


Figure 10.4: ROC curve for dataset_A_1250_org_1250_dup_nysiis.

For dataset_A_1250_original and dataset_A_1250_duplicate the similarity metrics I used were Jaro distance and NYSIIS. Their ROC curves when run on these datasets are very similar and both of them pass through the optimal point (0,1). These results are not surprising since the algorithm parameters for Jaro distance were set at 0.75 and 1.0 inclusive, where 0.75 is the lower bound for a match and 1.0 is the upper bound for a match. Since

a duplicate is formed by making one mutation in one of the fields for each entry, then most duplicate entries should match with their original entry with algorithm parameters of 0.75 and 1.0.

Duplicate entries that are more difficult to match using Jaro distance are those where letters have been inserted or fields have been dropped. In contrast, NYSIIS has more difficulty with given names that have been replaced with nicknames, reordered words, and dropped fields. This is because Jaro distance takes substrings into account, so nicknames, reordering words, and deleted characters have less of an effect on it than on NYSIIS. Dropped fields affects both Jaro distance and NYSIIS when it comes to matching an original entry with its duplicate, but missing fields affect NYSIIS more. This is because NYSIIS can only make comparisons with letters and only five of the twelve columns contain strings that have just letters. Therefore, a dropped field that should contain a string of letters in it is seen as being a bigger loss of information when running NYSIIS on the dataset, than Jaro distance.

Jaro distance's cut off level is $0.58\bar{3}$, while NYSIIS is 0.8. Having such different cut off levels for Jaro distance and NYSIIS is to be expected, since Jaro distance can handle these kinds of mutations better then NYSIIS. This is because NYSIIS is more sensitive to nickname replacement, dropped fields, and letters being inserted or transposed then Jaro distance is. Both Jaro distance and NYSIIS managed to be optimal and NYSIIS's runtime was nearly half of that of Jaro distance, but Jaro distance was able to get the maximum number of true positives without any false negatives at a much lower level. Thus we have that the character-based similarity metric performed better when merging dataset_A_1250_original and dataset_A_1250_duplicate.

10.5 Merge 2500 Duplicates with 2500 Originals

Example 10.5. merge 2500 duplicates with 2500 originals using Smith-Waterman distance

dataset 1:	dataset_A_2500_original
dataset 2:	dataset_A_2500_duplicate
method:	Smith-Waterman distance
algorithm parameters:	1, 6, 10 (G, lower bound, upper bound)
final dataset:	dataset_A_2500_org_2500_dup_swd

level	comp_specificity	false_negatives	false_positives	sensitivity	specificity	total_matches	total_non_matches	true_negatives	true_positives
0	1	0	6247500	1	0	2500	6247500	0	2500
0.083333	0.440671	18	2753090	0.992800	0.559329	2500	6247500	3494410	2482
0.166667	0.090378	299	564639	0.880400	0.909622	2500	6247500	5682861	2201
0.250000	0.009227	1192	57648	0.523200	0.990773	2500	6247500	6189852	1308
0.333333	0.000465	2116	2904	0.153600	0.999535	2500	6247500	6244596	384
0.416667	0.000007	2449	46	0.020400	0.999993	2500	6247500	6247454	51
0.500000	0	2495	0	0.002000	1	2500	6247500	6247500	5

Runtime	Comparisons	Comparisons per Second
895 seconds	6250000	6983.2402

Table 10.7: Stats for dataset_A_2500_org_2500_dup_swd.

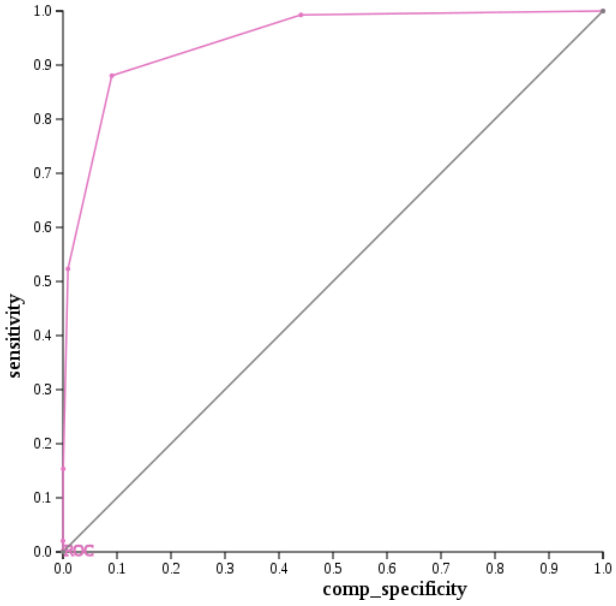


Figure 10.5: ROC curve for dataset_A_2500_org_2500_dup_swd.

Example 10.6. merge 2500 duplicates with 2500 originals using Double Metaphone

dataset 1: dataset_A_2500_original

dataset 2: dataset_A_2500_duplicate

method: Double Metaphone

final dataset: dataset_A_2500_org_2500_dup_dm

level	comp_specificity	false_negatives	false_positives	sensitivity	specificity	total_matches	total_non_matches	true_negatives	true_positives
0	1	0	6247500	1	0	2500	6247500	0	2500
0.200000	0.195716	0	1222737	1	0.804284	2500	6247500	5024763	2500
0.400000	0.002056	0	12847	1	0.997944	2500	6247500	6234653	2500
0.600000	0.000005	0	29	1	0.999995	2500	6247500	6247471	2500
0.800000	0	0	0	1	1	2500	6247500	6247500	2500
1	0	862	0	0.655200	1	2500	6247500	6247500	1638

Runtime	Comparisons	Comparisons per Second
308 seconds	6250000	20292.2078

Table 10.8: Stats for dataset_A_2500_org_2500_dup_dm.

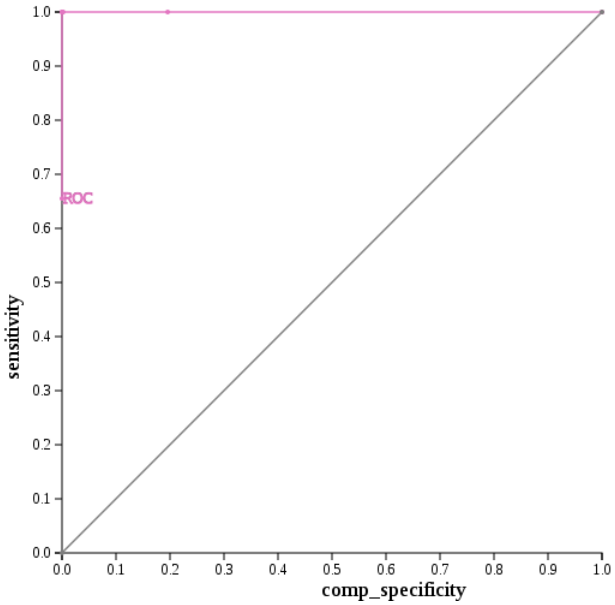


Figure 10.6: ROC curve for dataset_A_2500_org_2500_dup_dm.

For dataset_A_2500_original and dataset_A_2500_duplicate the similarity metrics I used were Smith-Waterman distance and Double Metaphone. Their ROC curves are less similar than the previous examples–only the ROC curve for Double Metaphone passes through the optimal point (0,1). The ROC curve for Smith-Waterman distance does not pass through the optimal point because it does not have a clear cut off level in which there are no false

negatives and the maximum number of true positives possible on the same level. The closest we were able to get the ROC curve for Smith-Waterman distance to the one for Double Metaphone was using the algorithm parameters G = 1, lower bound = 6, and upper bound = 10 (inclusive).

When the original data was mutated to produce the duplicate entries, mutations could occur at any point in a field. Smith-Waterman distance is more sensitive to mutations that occur in the middle of the string then those that occur at the beginning or end of the string. This property has the advantage of making Smith-Waterman good at substring matching, but if random mutations are introduced into the data in the middle of strings, then Smith-Waterman is not as useful of a similarity metric to use. Double Metaphone's main drawback is that it cannot deal with numbers, which reduces the pool of fields it has to compare on. Dropped fields also decrease the number of columns to compare on and make it harder to tell if two records should be linked or not. Double Metaphone's runtime was also nearly a third of that of Smith-Waterman distnace. Since Smith-Waterman distance does not have a clear cut off level and Double Metaphone does at $0.8\bar{3}$, then we have that the phonetic-based similarity metric was better at merging dataset_A_2500_original and dataset_A_2500_duplicate.

10.6 Merge 5000 Duplicates with 5000 Originals

Example 10.7. merge 5000 duplicates with 5000 originals using q-gram distance

dataset 1:	dataset_A_5000_original
dataset 2:	dataset_A_5000_duplicate
method:	q-gram distance
algorithm parameters:	1 (the size of q)
algorithm threshold:	0.75 (lower bound for match – upper bound is automatically set to 1.0)
final dataset:	dataset_A_5000_org_5000_dup_qgd

level	comp_specificity	false_negatives	false_positives	sensitivity	specificity	total_matches	total_non_matches	true_negatives	true_positives
0	1	0	24995000	1	0	5000	24995000	0	5000
0.083333	0.396341	0	9906543	1	0.603659	5000	24995000	15088457	5000
0.166667	0.073514	0	1837476	1	0.926486	5000	24995000	23157524	5000
0.250000	0.007297	0	182387	1	0.992703	5000	24995000	24812613	5000
0.333333	0.000420	0	10503	1	0.999580	5000	24995000	24984497	5000
0.416667	0.000014	0	359	1	0.999986	5000	24995000	24994641	5000
0.500000	0.000000	0	2	1	1.000000	5000	24995000	24994998	5000
0.583333	0	1	0	0.999800	1	5000	24995000	24995000	4999
0.666667	0	17	0	0.996600	1	5000	24995000	24995000	4983
0.750000	0	151	0	0.969800	1	5000	24995000	24995000	4849
0.833333	0	823	0	0.835400	1	5000	24995000	24995000	4177
0.916667	0	2531	0	0.493800	1	5000	24995000	24995000	2469
1	0	4419	0	0.116200	1	5000	24995000	24995000	581

Runtime	Comparisons	Comparisons per Second
3434 seconds	25000000	7280.1398

Table 10.9: Stats for dataset_A_5000_org_5000_dup_qgd.

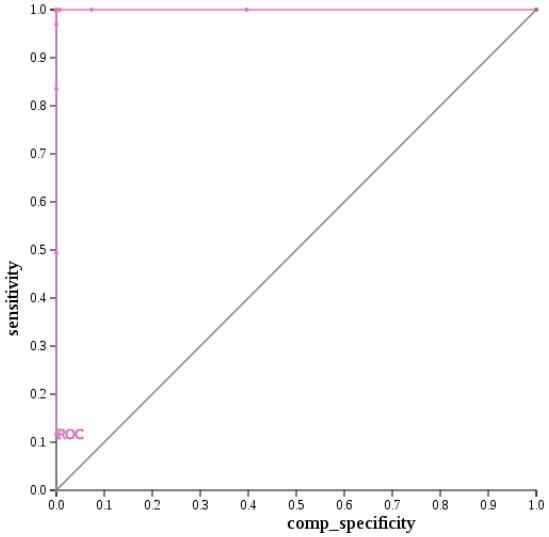


Figure 10.7: ROC curve for dataset_A_5000_org_5000_dup_qgd.

Example 10.8. merge 5000 duplicates with 5000 originals using Soundex

dataset 1: dataset_A_5000_original

dataset 2: dataset_A_5000_duplicate

method: Soundex

final dataset: dataset_A_5000_org_5000_dup_soundex

level	comp_specificity	false_negatives	false_positives	sensitivity	specificity	total_matches	total_non_matches	true_negatives	true_positives
0	1	0	24995000	1	0	5000	24995000	0	5000
0.200000	0.198132	0	4952303	1	0.801868	5000	24995000	20042697	5000
0.400000	0.002594	0	64826	1	0.997406	5000	24995000	24930174	5000
0.600000	0.000011	0	274	1	0.999989	5000	24995000	24994726	5000
0.800000	0	0	0	1	1	5000	24995000	24995000	5000
1	0	1124	0	0.775200	1	5000	24995000	24995000	3876

Runtime	Comparisons	Comparisons per Second
1108 seconds	25000000	22563.1769

Table 10.10: Stats for dataset_A_5000_org_5000_dup_soundex.

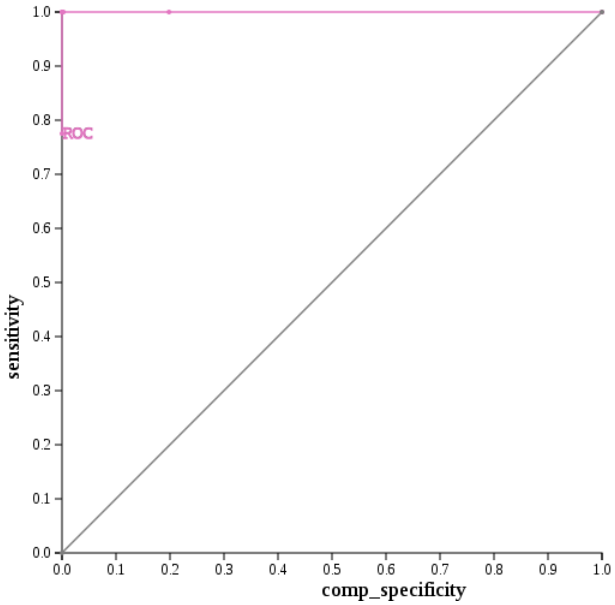


Figure 10.8: ROC curve for dataset_A_5000_org_5000_dup_soundex.

For dataset_A_5000_original and dataset_A_5000_duplicate the similarity metrics I used were q-gram distance and Soundex. Their ROC curves are almost identical and both pass through the optimal point (0,1). These results are not surprising because when the size of q is 1, as it is in this example, then the q-gram distance behaves similar to edit distance in that it looks at character relationships between two strings. In a previous example we

looked at the ROC curves for dataset_A_500_original and dataset_A_500_duplicate where the similarity metrics edit distance and Soundex were used. Since q-gram distance is similar to edit distance when $q = 1$, then we should expect to see similar results. However, q-gram distance also shares some properties similar to those of Jaro distance, in that the algorithm has a threshold. In this example the lower bound for a match was set at 0.75. For q-gram distance the upper bound for a match is automatically set to 1.0.

Q-gram distance's cut off level is 0.5, whereas Soundex is $0.8\bar{3}$. It is interesting that there is such a large difference between the optimal cut off levels of the two metrics. If q-gram distance behaved more like edit distance, then we might have expected the difference to be small. However, it seems that the properties that make q-gram distance similar to Jaro distance also make it have a lower cut off level.

It should be noted that both q-gram distance and Soundex performed optimally. Also, we only looked it $q = 1$. If q were to equal a different value maybe Soundex would have performed better with merging these two databases together. Soundex was able to have the faster runtime and completed the merge job in a third of the time that it took q-gram distance to run. Given that $q = 1$ and a lower bound for a match set at 0.75 we have that q-gram distance was able to get the maximum number of true positives without any false negatives at a significantly lower level. Thus we have that the character-based similarity metric was better at merging dataset_A_5000_original and dataset_A_5000_duplicate.

10.7 Compare Character-Based and Phonetic-Based Similarity Metrics

For each dataset (dataset_A_500, dataset_A_1250, dataset_A_2500, dataset_A_5000) I used a character-based similarity metric (edit distance, Jaro distance, Smith-Waterman distance, q-gram distance) and a phonetic similarity metric (Soundex, NYSIIS, Double Metaphone). I did this to examine how a particular character-based similarity metric and a particular phonetic-based similarity metric compare when used to merge each dataset. Any of the seven metrics can be used on any of the datasets, though a character-based similarity metric must be given the additional parameters that it uses.

Experimenting with these datasets individually provided interesting results, but raises more questions. The main one being how does increasing the size of the dataset affect the algorithm's effectiveness and runtime? Algorithm may be extremely effective at finding duplicate records in a small dataset, but could return multiple errors, become much less effective, or take a very long time to run for a larger dataset. I approached this question by looking at the speed and effectiveness of algorithms as dataset size increases. To analyze this progression, I chose the Jaro distance and Double Metaphone algorithms as representative character-based and phonetic-based similarity metrics.

Example 10.9. merge 500 duplicates with 500 originals using Jaro distance

dataset 1:	dataset_A_500_original
dataset 2:	dataset_A_500_duplicate
method:	Jaro distance
algorithm parameters:	0.85, 1.0 (lower bound, upper bound inclusive)
final dataset:	dataset_A_500_org_500_dup_jd

level	comp_specificity	false_negatives	false_positives	sensitivity	specificity	total_matches	total_non_matches	true_negatives	true_positives
0	1	0	249500	1	0	500	249500	0	500
0.083333	0.229142	0	57171	1	0.770858	500	249500	192329	500
0.166667	0.013359	0	3333	1	0.986641	500	249500	246167	500
0.250000	0.000269	0	67	1	0.999731	500	249500	249433	500
0.333333	0.000008	0	2	1	0.999992	500	249500	249498	500
0.500000	0	0	0	1	1	500	249500	249500	500
0.583333	0	1	0	0.998000	1	500	249500	249500	499
0.666667	0	2	0	0.996000	1	500	249500	249500	498
0.750000	0	14	0	0.972000	1	500	249500	249500	486
0.833333	0	73	0	0.854000	1	500	249500	249500	427
0.916667	0	261	0	0.478000	1	500	249500	249500	239
1	0	435	0	0.130000	1	500	249500	249500	65

Runtime	Comparisons	Comparisons per Second
30 seconds	250000	8333.3333

Table 10.11: Stats for dataset_A_500_org_500_dup_jd.

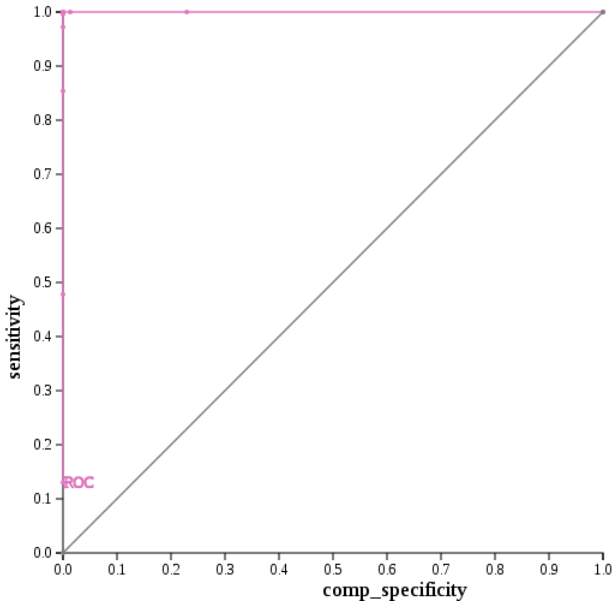


Figure 10.9: ROC curve for dataset_A_500_org_500_dup_jd.

Example 10.10. merge 500 duplicates with 500 originals using Double Metaphone

dataset 1: dataset_A_500_original

dataset 2: dataset_A_500_duplicate

method: Double Metaphone

final dataset: dataset_A_500_org_500_dup_dm

level	comp_specificity	false_negatives	false_positives	sensitivity	specificity	total_matches	total_non_matches	true_negatives	true_positives
0	1	0	249500	1	0	500	249500	0	500
0.200000	0.192244	0	47965	1	0.807756	500	249500	201535	500
0.400000	0.002116	0	528	1	0.997884	500	249500	248972	500
0.600000	0.000020	0	5	1	0.999980	500	249500	249495	500
0.800000	0	0	0	1	1	500	249500	249500	500
1	0	170	0	0.660000	1	500	249500	249500	330

Runtime	Comparisons	Comparisons per Second
13 seconds	250000	19230.7692

Table 10.12: Stats for dataset_A_500_org_500_dup_dm.

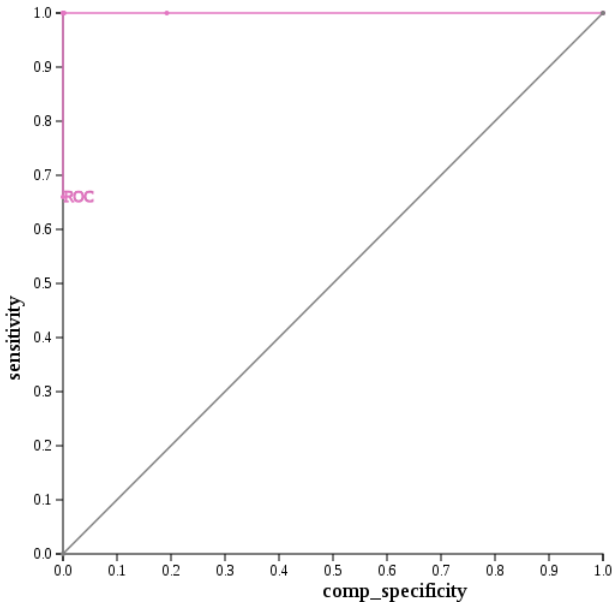


Figure 10.10: ROC curve for dataset_A_500_org_500_dup_dm.

Example 10.11. merge 1250 duplicates with 1250 originals using Jaro distance

dataset 1:	dataset_A_1250_original
dataset 2:	dataset_A_1250_duplicate
method:	Jaro distance
algorithm parameters:	0.85, 1.0 (lower bound, upper bound inclusive)
final dataset:	dataset_A_1250_org_1250_dup_jd

level	comp_specificity	false_negatives	false_positives	sensitivity	specificity	total_matches	total_non_matches	true_negatives	true_positives
0	1	0	1561251	1	0	1249	1561251	0	1249
0.083333	0.240923	0	376142	1	0.759077	1249	1561251	1185109	1249
0.166667	0.013824	0	21583	1	0.986176	1249	1561251	1539668	1249
0.250000	0.000338	0	527	1	0.999662	1249	1561251	1560724	1249
0.333333	0.000008	0	12	1	0.999992	1249	1561251	1561239	1249
0.583333	0.000001	0	1	1	0.999999	1249	1561251	1561250	1249
0.666667	0.000001	2	1	0.998399	0.999999	1249	1561251	1561250	1247
0.750000	0.000001	37	1	0.970376	0.999999	1249	1561251	1561250	1212
0.833333	0.000001	183	1	0.853483	0.999999	1249	1561251	1561250	1066
0.916667	0	592	0	0.526021	1	1249	1561251	1561251	657
1	0	1077	0	0.137710	1	1249	1561251	1561251	172

Runtime	Comparisons	Comparisons per Second
168 seconds	1562500	9300.5952

Table 10.13: Stats for dataset_A_1250_org_1250_dup_jd.

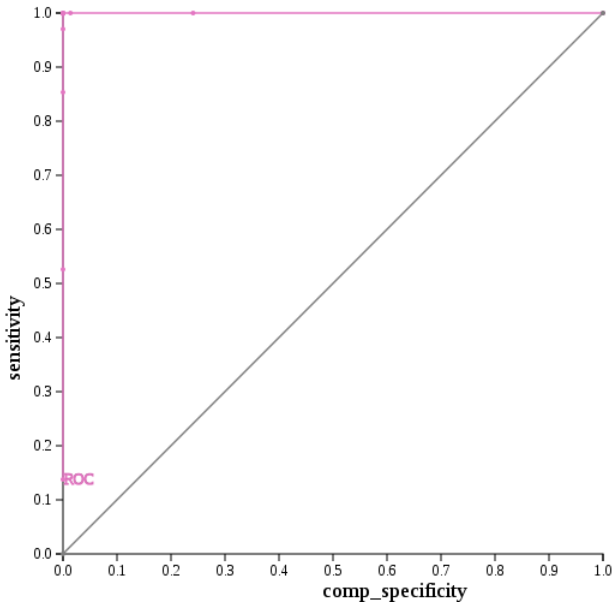


Figure 10.11: ROC curve for dataset_A_1250_org_1250_dup_jd.

Example 10.12. merge 1250 duplicates with 1250 originals using Double Metaphone

dataset 1: dataset_A_1250_original

dataset 2: dataset_A_1250_duplicate

method: Double Metaphone

final dataset: dataset_A_1250_org_1250_dup_dm

level	comp_specificity	false_negatives	false_positives	sensitivity	specificity	total_matches	total_non_matches	true_negatives	true_positives
0	1	0	1561251	1	0	1249	1561251	0	1249
0.200000	0.206356	0	322173	1	0.793644	1249	1561251	1239078	1249
0.400000	0.002101	0	3280	1	0.997899	1249	1561251	1557971	1249
0.600000	0.000007	0	11	1	0.999993	1249	1561251	1561240	1249
0.800000	0.000001	0	1	1	0.999999	1249	1561251	1561250	1249
1	0.000001	430	1	0.655725	0.999999	1249	1561251	1561250	819

Runtime	Comparisons	Comparisons per Second
77 seconds	1562500	20292.2078

Table 10.14: Stats for dataset_A_1250_org_1250_dup_dm.

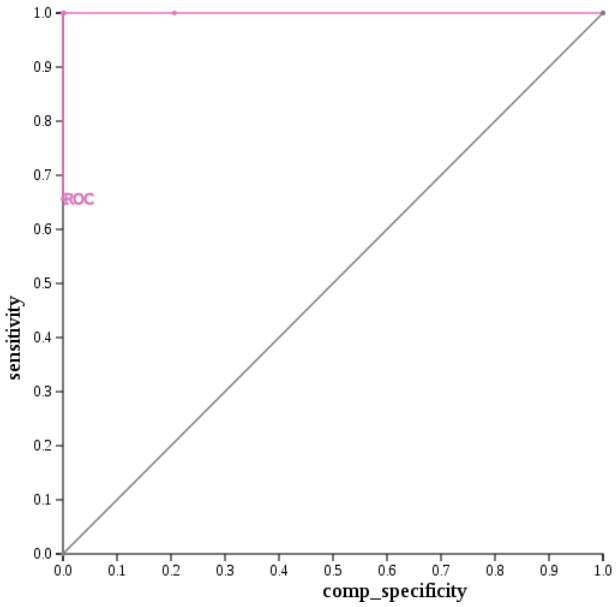


Figure 10.12: ROC curve for dataset_A_1250_org_1250_dup_dm.

Example 10.13. merge 2500 duplicates with 2500 originals using Jaro distance

dataset 1:	dataset_A_2500_original
dataset 2:	dataset_A_2500_duplicate
method:	Jaro distance
algorithm parameters:	0.85, 1.0 (lower bound, upper bound inclusive)
final dataset:	dataset_A_2500_org_2500_dup_jd

level	comp_specificity	false_negatives	false_positives	sensitivity	specificity	total_matches	total_non_matches	true_negatives	true_positives
0	1	0	6247500	1	0	2500	6247500	0	2500
0.083333	0.229632	0	1434624	1	0.770368	2500	6247500	4812876	2500
0.166667	0.013099	0	81833	1	0.986901	2500	6247500	6165667	2500
0.250000	0.000291	0	1820	1	0.999709	2500	6247500	6245680	2500
0.333333	0.000003	0	17	1	0.999997	2500	6247500	6247483	2500
0.583333	0	0	0	1	1	2500	6247500	6247500	2500
0.666667	0	12	0	0.995200	1	2500	6247500	6247500	2488
0.750000	0	78	0	0.968800	1	2500	6247500	6247500	2422
0.833333	0	371	0	0.851600	1	2500	6247500	6247500	2129
0.916667	0	1217	0	0.513200	1	2500	6247500	6247500	1283
1	0	2201	0	0.119600	1	2500	6247500	6247500	299

Runtime	Comparisons	Comparisons per Second
672 seconds	6250000	9300.5952

Table 10.15: Stats for dataset_A_2500_org_2500_dup_jd.

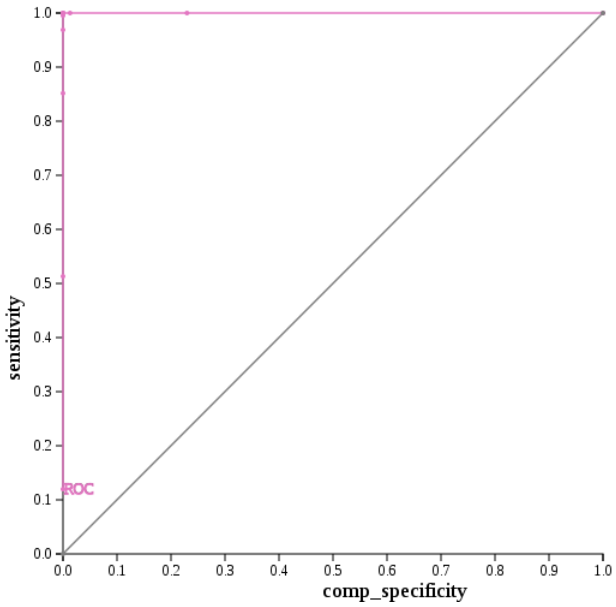


Figure 10.13: ROC curve for dataset_A_2500_org_2500_dup_jd.

Example 10.14. merge 2500 duplicates with 2500 originals using Double Metaphone

dataset 1: dataset_A_2500_original

dataset 2: dataset_A_2500_duplicate

method: Double Metaphone

final dataset: dataset_A_2500_org_2500_dup_dm

level	comp_specificity	false_negatives	false_positives	sensitivity	specificity	total_matches	total_non_matches	true_negatives	true_positives
0	1	0	6247500	1	0	2500	6247500	0	2500
0.200000	0.195716	0	1222737	1	0.804284	2500	6247500	5024763	2500
0.400000	0.002056	0	12847	1	0.997944	2500	6247500	6234653	2500
0.600000	0.000005	0	29	1	0.999995	2500	6247500	6247471	2500
0.800000	0	0	0	1	1	2500	6247500	6247500	2500
1	0	862	0	0.655200	1	2500	6247500	6247500	1638

Runtime	Comparisons	Comparisons per Second
308 seconds	6250000	20292.2078

Table 10.16: Stats for dataset_A_2500_org_2500_dup_dm.

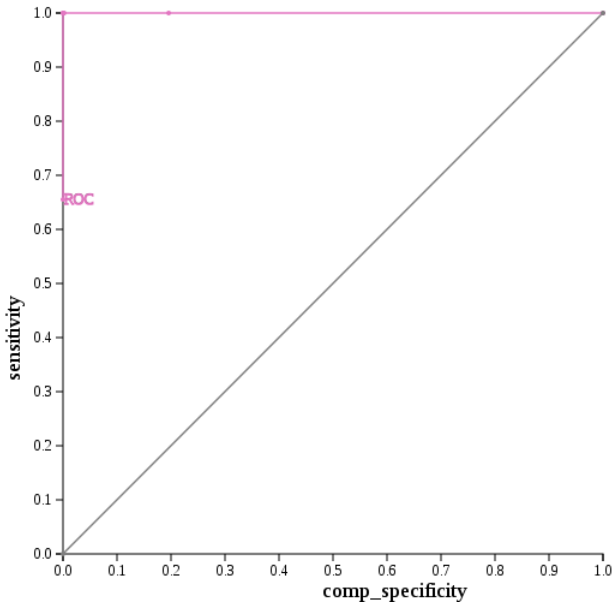


Figure 10.14: ROC curve for dataset_A_2500_org_2500_dup_dm.

Example 10.15. merge 5000 duplicates with 5000 originals using Jaro distance

dataset 1:	dataset_A_5000_original
dataset 2:	dataset_A_5000_duplicate
method:	Jaro distance
algorithm parameters:	0.85, 1.0 (lower bound, upper bound inclusive)
final dataset:	dataset_A_5000_org_5000_dup_jd

level	comp_specificity	false_negatives	false_positives	sensitivity	specificity	total_matches	total_non_matches	true_negatives	true_positives
0	1	0	24995000	1	0	5000	24995000	0	5000
0.083333	0.229163	0	5727928	1	0.770837	5000	24995000	19267072	5000
0.166667	0.013145	0	328552	1	0.986855	5000	24995000	24666448	5000
0.250000	0.000302	0	7556	1	0.999698	5000	24995000	24987444	5000
0.333333	0.000003	0	80	1	0.999997	5000	24995000	24994920	5000
0.500000	0	0	0	1	1	5000	24995000	24995000	5000
0.583333	0	1	0	0.999800	1	5000	24995000	24995000	4999
0.666667	0	17	0	0.996600	1	5000	24995000	24995000	4983
0.750000	0	145	0	0.971000	1	5000	24995000	24995000	4855
0.833333	0	802	0	0.839600	1	5000	24995000	24995000	4198
0.916667	0	2475	0	0.505000	1	5000	24995000	24995000	2525
1	0	4377	0	0.124600	1	5000	24995000	24995000	623

Runtime	Comparisons	Comparisons per Second
2666 seconds	25000000	9377.3443

Table 10.17: Stats for dataset_A_5000_org_5000_dup_jd.

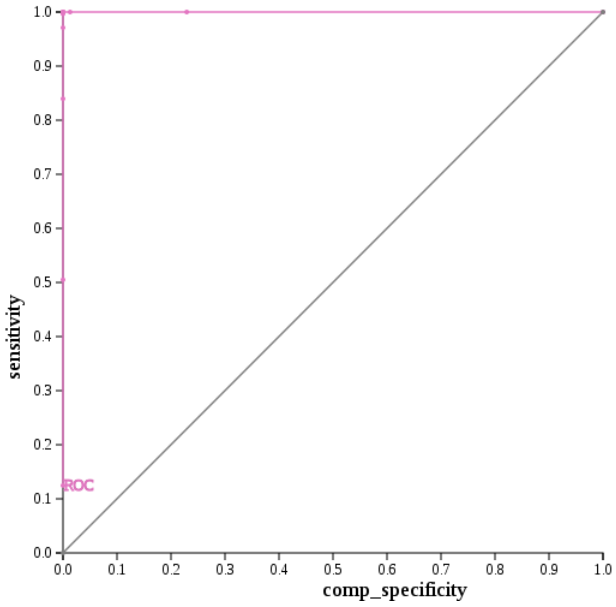


Figure 10.15: ROC curve for dataset_A_5000_org_5000_dup_jd.

Example 10.16. merge 5000 duplicates with 5000 originals using Double Metaphone

dataset 1: dataset_A_5000_original

dataset 2: dataset_A_5000_duplicate

method: Double Metaphone

final dataset: dataset_A_5000_org_5000_dup_dm

level	comp_specificity	false_negatives	false_positives	sensitivity	specificity	total_matches	total_non_matches	true_negatives	true_positives
0	1	0	24995000	1	0	5000	24995000	0	5000
0.200000	0.194661	0	4865554	1	0.805339	5000	24995000	20129446	5000
0.400000	0.002058	0	51436	1	0.997942	5000	24995000	24943564	5000
0.600000	0.000007	0	168	1	0.999993	5000	24995000	24994832	5000
0.800000	0	0	0	1	1	5000	24995000	24995000	5000
1	0	1701	0	0.659800	1	5000	24995000	24995000	3299

Runtime	Comparisons	Comparisons per Second
1214 seconds	25000000	20593.0807

Table 10.18: Stats for dataset_A_5000_org_5000_dup_dm.

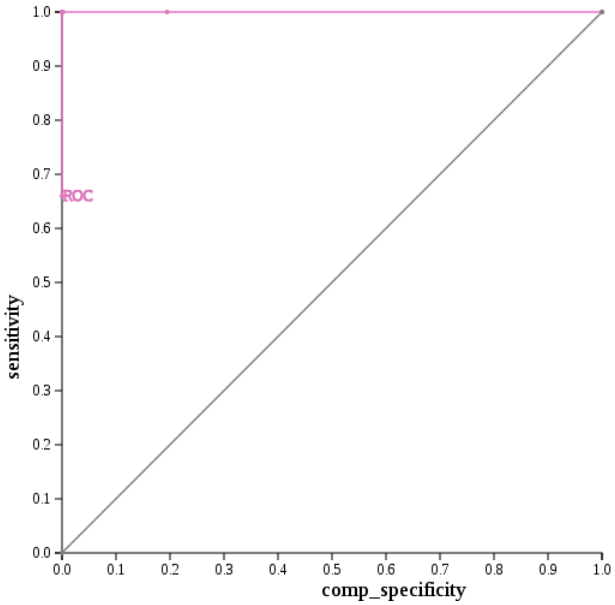


Figure 10.16: ROC curve for dataset_A_5000_org_5000_dup_dm.

The ROC curves for all eight of the datasets are almost identical. All of the ROC curves go straight up from the point (0,0), pass through the optimal point (0,1), and then go straight to the right to the point (1,1). So even if the size of the dataset is increased, the algorithms are still very effective at finding correct record pairs. Jaro distance's cut off levels for the four dataset are: 0.5, $0.58\bar{3}$, $0.58\bar{3}$, and 0.5 respectively with algorithm parameters 0.85 set

as the lower bound and 1.0 set as the upper bound, inclusive, while Double Metaphone's cut off levels are: 0.8, 0.8, $0.8\bar{3}$, and 0.8 respectively. Jaro distance was able to get the maximum number of true positives without any false negatives at a significantly lower level then Double Metaphone for each of the four datasets, so the character-based similarity metric performed better for these datasets than the phonetic-based similarity metric.

It should also be noted that neither similarity metric had problems with scalability. As the size of the datasets increased, it did take longer to run each comparison, but each comparison was able to finish running in a timely manner. The smaller datasets finished running in a matter of seconds, whereas the larger datasets took a few minutes and the largest dataset took under forty-four minutes to run. If the size of the datasets were to be increased, the algorithms will take longer to run on them, but would still be able to run the analysis necessary to determine matches or non-matches and at what tolerance levels.

Chapter 11. Updating a Database With Multiple Duplicate Entries

To examine the problem of updating a database with another database that has multiple duplicate entries, four forms of the updating database are used. The format of the updating database will be described in greater details in the examples to follow.

11.1 Updating a Database With One, Two, or Three Duplicate Entries

Example 11.1. Form 1

Dataset two consists of the following entries:

(i) 500 new original entries

(ii) 500 duplicates from the 4000 and 500

  (a) from the 4000 original entries

    i. 250 one time duplicates

    ii. 150 two time duplicates

    iii. 51 three time duplicates

  (b) from the 500 new original entries

    i. 49 one time duplicate

dataset 1:	dataset_A_4000_originals
dataset 2:	dataset_A_1000_update_form1
method:	Jaro distance
algorithm parameters:	0.75, 1.0 (lower bound, upper bound inclusive)
final dataset:	dataset_A_4000_org_1000_dup_form1_jd

level	comp_specificity	false_negatives	false_positives	sensitivity	specificity	total_matches	total_non_matches	true_negatives	true_positives
0	1	0	3999549	1	0	451	3999549	0	451
0.083333	0.493256	0	1972801	1	0.506744	451	3999549	2026748	451
0.166667	0.118701	0	474752	1	0.881299	451	3999549	3524797	451
0.250000	0.014813	0	59247	1	0.985187	451	3999549	3940302	451
0.333333	0.001046	0	4185	1	0.998954	451	3999549	3995364	451
0.416667	0.000046	0	182	1	0.999954	451	3999549	3999367	451
0.500000	0.000001	0	4	1	0.999999	451	3999549	3999545	451
0.583333	0.000000	0	1	1	1.000000	451	3999549	3999548	451
0.666667	0	4	0	0.991131	1	451	3999549	3999549	447
0.750000	0	20	0	0.955654	1	451	3999549	3999549	431
0.833333	0	76	0	0.831486	1	451	3999549	3999549	375
0.916667	0	223	0	0.505543	1	451	3999549	3999549	228
1	0	398	0	0.117517	1	451	3999549	3999549	53

Runtime	Comparisons	Comparisons per Second
424 seconds	4000000	9433.9623

Table 11.1: Stats for dataset_A_4000_org_1000_dup_form1_jd.

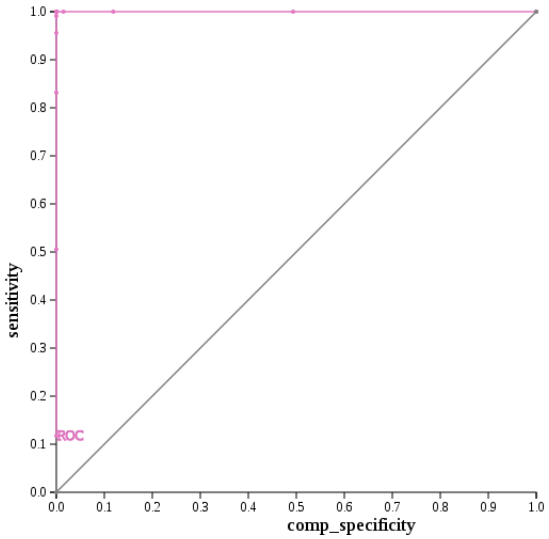


Figure 11.1: ROC curve for dataset_A_4000_org_1000_dup_form1_jd.

Example 11.2. Form 2

Dataset two consists of the following entries:

(i) 500 new original entries

(ii) 500 duplicates from the 4000

(a) 299 one time duplicates

(b) 150 two time duplicates

(c) 51 three time duplicates

dataset 1: dataset_A_4000_originals

dataset 2: dataset_A_1000_update_form2

method: Soundex

final dataset: dataset_A_4000_org_1000_dup_form2_soundex

level	comp_specificity	false_negatives	false_positives	sensitivity	specificity	total_matches	total_non_matches	true_negatives	true_positives
0	1	0	6247500	1	0	2500	6247500	0	2500
0.200000	0.195716	0	1222737	1	0.804284	2500	6247500	5024763	2500
0.400000	0.002056	0	12847	1	0.997944	2500	6247500	6234653	2500
0.600000	0.000005	0	29	1	0.999995	2500	6247500	6247471	2500
0.800000	0	0	0	1	1	2500	6247500	6247500	2500
1	0	862	0	0.655200	1	2500	6247500	6247500	1638

Runtime	Comparisons	Comparisons per Second
308 seconds	6250000	20292.2078

Table 11.2: Stats for dataset_A_4000_org_1000_dup_form2_soundex.

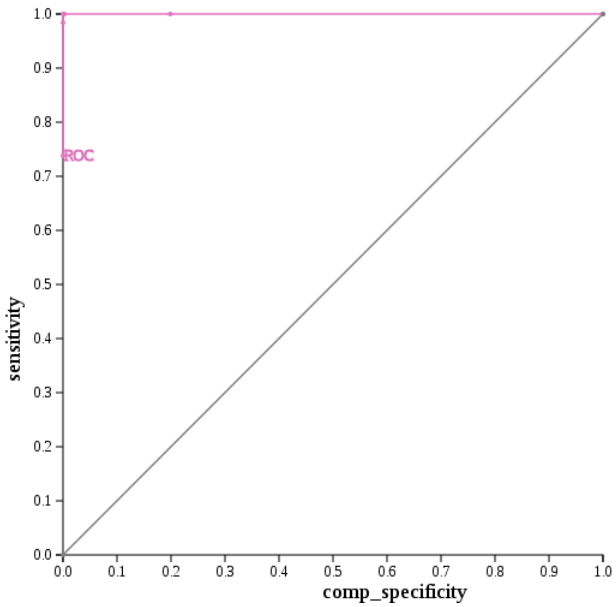


Figure 11.2: ROC curve for dataset_A_4000_org_1000_dup_form2_soundex.

Example 11.3. Form 3

Dataset two consists of the following entries:

(i) 525 new original entries

(ii) 475 duplicates from the 4000

(a) 274 one time duplicates

(b) 150 two time duplicates

(c) 51 three time duplicates

dataset 1:	dataset_A_4000_originals
dataset 2:	dataset_A_1000_update_form3
method:	q-gram distance
algorithm parameters:	1 (the size of q)
algorithm threshold:	0.7 (lower bound for match – upper bound is automatically set to 1.0)
final dataset:	dataset_A_4000_org_1000_dup_form3_qgd

level	comp_specificity	false_negatives	false_positives	sensitivity	specificity	total_matches	total_non_matches	true_negatives	true_positives
0	1	0	3999525	1	0	475	3999525	0	475
0.083333	0.490924	0	1963464	1	0.509076	475	3999525	2036061	475
0.166667	0.122668	0	490612	1	0.877332	475	3999525	3508913	475
0.250000	0.016759	0	67028	1	0.983241	475	3999525	3932497	475
0.333333	0.001306	0	5223	1	0.998694	475	3999525	3994302	475
0.416667	0.000064	0	255	1	0.999936	475	3999525	3999270	475
0.500000	0.000001	0	3	1	0.999999	475	3999525	3999522	475
0.583333	0	0	0	1	1	475	3999525	3999525	475
0.666667	0	3	0	0.993684	1	475	3999525	3999525	472
0.750000	0	21	0	0.955789	1	475	3999525	3999525	454
0.833333	0	82	0	0.827368	1	475	3999525	3999525	393
0.916667	0	262	0	0.448421	1	475	3999525	3999525	213
1	0	419	0	0.117895	1	475	3999525	3999525	56

Runtime	Comparisons	Comparisons per Second
551 seconds	4000000	7259.5281

Table 11.3: Stats for dataset_A_4000_org_1000_dup_form3_qgd.

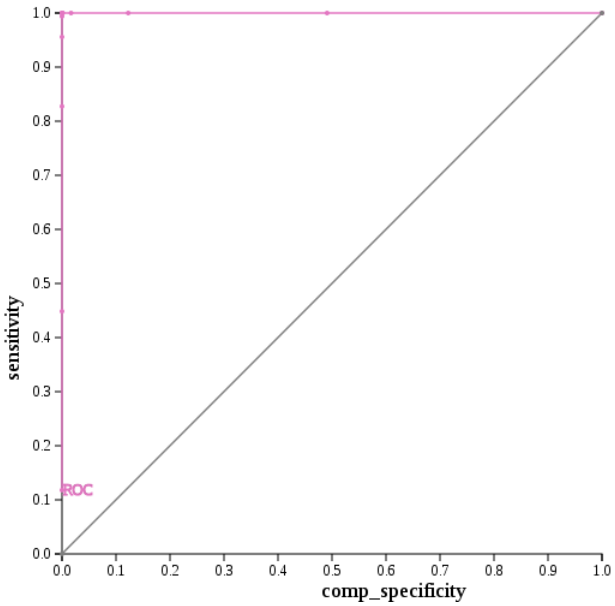


Figure 11.3: ROC curve for dataset_A_4000_org_1000_dup_form3_qgd.

Example 11.4. Form 4

Dataset two consists of the following entries:

(i) 549 new original entries

(ii) 451 duplicates from the 4000

(a) 250 one time duplicates

(b) 150 two time duplicates

(c) 51 three time duplicates

dataset 1:	dataset_A_4000_originals
dataset 2:	dataset_A_1000_update_form4
method:	Double Metaphone
final dataset:	dataset_A_4000_org_1000_dup_form4_dm

level	comp_specificity	false_negatives	false_positives	sensitivity	specificity	total_matches	total_non_matches	true_negatives	true_positives
0	1	0	3999549	1	0	451	3999549	0	451
0.200000	0.194672	0	778601	1	0.805328	451	3999549	3220948	451
0.400000	0.002123	0	8493	1	0.997877	451	3999549	3991056	451
0.600000	0.000004	0	15	1	0.999996	451	3999549	3999534	451
0.800000	0	16	0	0.964523	1	451	3999549	3999549	435
1	0	191	0	0.576497	1	451	3999549	3999549	260

Runtime	Comparisons	Comparisons per Second
196 seconds	4000000	20408.1633

Table 11.4: Stats for dataset_A_4000_org_1000_dup_form4_dm.

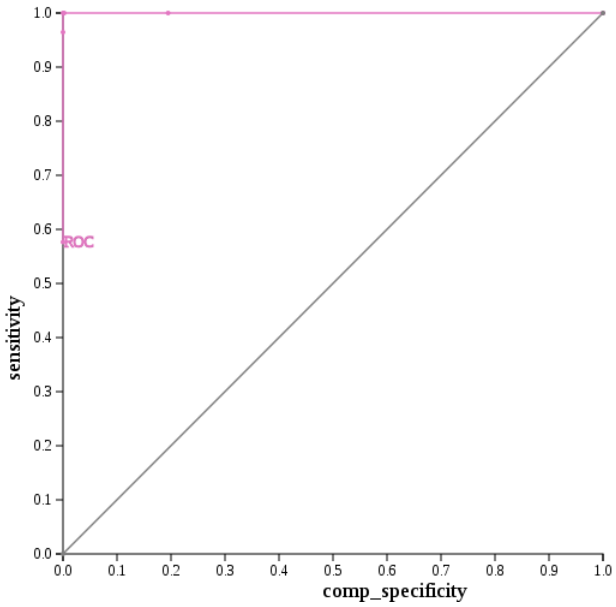


Figure 11.4: ROC curve for dataset_A_4000_org_1000_dup_form4_dm.

In all four of the above examples the resulting dataset has a near-perfect ROC curve. Meaning that the ROC curve goes through or almost goes through the optimal point (0,1). Also in their respective stats tables there is a clear cut off point in which there are zero false negatives and the maximum number of true positives possible on the same level. We are getting these results because of how the duplicate entries are formed.

Recall that a duplicate entry is formed by making one change in one of the fields. A change consists of one of the following actions: inserting a single character, deleting a single character, replacing a single character, transposing two characters, or transposing two words (example: First Street becomes Street First), replacing a name with a nickname, or even dropping a field. The database I used in the previous examples had only been mutated very

lightly, which made it easy for the algorithms to produce near-perfect ROC curves. Therefore, I created my own mutator program mutator.py (which can be found in the appendix) that performs the same kinds of changes mentioned above, except for nickname replacement.

This program is configurable; the user can choose how many rows to mutate, how many times they should be mutated, and how many duplicates should be created for each original. The program mutates the original dataset entries with letters, numbers, and special characters. The letters can be either capital or lowercase from the standard English alphabet, the number selection goes from zero to nine inclusive, and the special characters available are a space, period, exclamation point, and question mark. The probability of a character being inserted is 20%, a character being deleted is 20%, a character being replaced is 20%, a character being moved is 20%, reordering words is $13.\bar{3}\%$ (i.e. 2 in 15), and a dropped field is $6.\bar{6}\%$ (i.e. 1 in 15).

Creating the mutated dataset using mutator.py led to more interesting results, where the ROC curves are more curved and there is no longer a clear, ideal cutoff point in which there are zero false negatives and the maximum number of true positives possible on the same level. Below is a sequence where I progressively broke a dataset (creating a dataset with one or multiple duplicate entries from the original dataset that have been mutated more than once) and merged it with the original dataset.

11.2 Updating a Database With Duplicate Entries That Have Multiple Mutations

In this series of examples we will update a dataset with duplicate entires that have been mutated by letters, numbers, and special characters. This is an important series of examples to look at because previuosly mutated datasets only looked at the mutations that invloved letters. Numbers and special characters are also keys that can be selected by mistake when another key was met to be pressed, so seeing how these affect the number of true positive and true negatives obtained at each tolerance level should be interesting.

Example 11.5. Step one in breaking the dataset

dataset 1:	dataset_A_500_original
dataset 2:	dataset_A_from_500_org_100_dup_1021_1031_1041_1051_1061_1071 _1081_1091_10101_10111
method:	edit distance
algorithm threshold:	2.0
final dataset:	dataset_A_500_org_100a_dup_ed

Dataset 2 was formed by the following process: ./mutator.py dataset_A_500_original.csv rec_id dataset_A_from_500_org_100_dup_1021_1031_1041_1051_1061_1071_1081_1091_10101_10111 10:2:1 10:3:1 10:4:1 10:5:1 10:6:1 10:7:1 10:8:1 10:9:1 10:10:1 10:11:1

(i) 10 rows are randomly selected and then 1 duplicate record for each of the rows was created with 2 mutations.

(ii) 10 rows are randomly selected (from the remaining 490 unchosen rows) and then 1 duplicate record for each of the rows was created with 3 mutations.

(iii) 10 rows are randomly selected (from the remaining 480 unchosen rows) and then 1 duplicate record for each of the rows was created with 4 mutations.

(iv) 10 rows are randomly selected (from the remaining 470 unchosen rows) and then 1 duplicate record for each of the rows was created with 5 mutations.

(v) 10 rows are randomly selected (from the remaining 460 unchosen rows) and then 1 duplicate record for each of the rows was created with 6 mutations.

(vi) 10 rows are randomly selected (from the remaining 450 unchosen rows) and then 1 duplicate record for each of the rows was created with 7 mutations.

(vii) 10 rows are randomly selected (from the remaining 440 unchosen rows) and then 1 duplicate record for each of the rows was created with 8 mutations.

(viii) 10 rows are randomly selected (from the remaining 430 unchosen rows) and then 1 duplicate record for each of the rows was created with 9 mutations.

(ix) 10 rows are randomly selected (from the remaining 420 unchosen rows) and then 1 duplicate record for each of the rows was created with 10 mutations.

(x) 10 rows are randomly selected (from the remaining 410 unchosen rows) and then 1 duplicate record for each of the rows was created with 11 mutations.

level	comp_specificity	false_negatives	false_positives	sensitivity	specificity	total_matches	total_non_matches	true_negatives	true_positives
0	1	0	49900	1	0	100	49900	0	100
0.083333	0.986333	0	49218	1	0.013667	100	49900	682	100
0.166667	0.883006	0	44062	1	0.116994	100	49900	5838	100
0.250000	0.623607	0	31118	1	0.376393	100	49900	18782	100
0.333333	0.315591	0	15748	1	0.684409	100	49900	34152	100
0.416667	0.112004	0	5589	1	0.887996	100	49900	44311	100
0.500000	0.026814	0	1338	1	0.973186	100	49900	48562	100
0.583333	0.003287	0	164	1	0.996713	100	49900	49736	100
0.666667	0.000381	1	19	0.990000	0.999619	100	49900	49881	99
0.750000	0	7	0	0.930000	1	100	49900	49900	93
0.833333	0	17	0	0.830000	1	100	49900	49900	83
0.916667	0	44	0	0.560000	1	100	49900	49900	56
1	0	76	0	0.240000	1	100	49900	49900	24

Runtime	Comparisons	Comparisons per Second
18 seconds	50000	2777.7778

Table 11.5: Stats for dataset_A_500_org_100a_dup_ed.

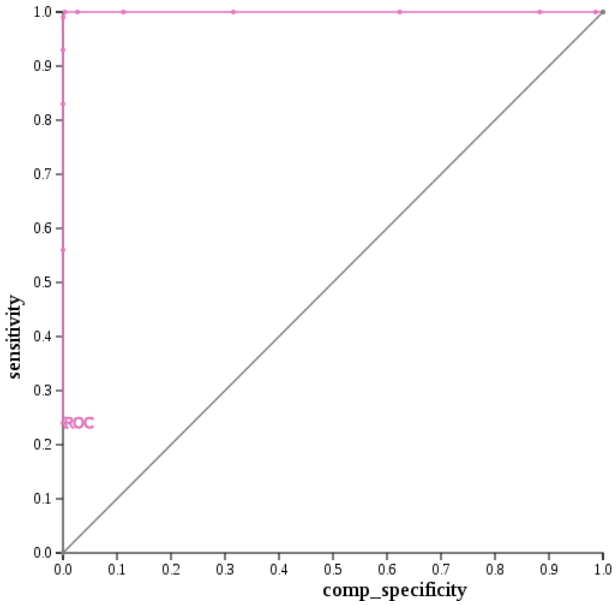


Figure 11.5: ROC curve for dataset_A_500_org_100a_dup_ed.

Example 11.6. Step two in breaking the dataset

dataset 1:	dataset_A_500_original
dataset 2:	dataset_A_from_500_org_100_dup_10121_10131_10141_10151_10161 _10171_10181_10191_10201_10211
method:	edit distance
algorithm threshold:	2.0
final dataset:	dataset_A_500_org_100b_dup_ed

Dataset 2 was formed by the following process: ./mutator.py dataset_A_500_original.csv rec_id dataset_A_from_500_org_100_dup_10121_10131_10141_10151_10161_10171_10181_10191_10201_10211 10:12:1 10:13:1 10:14:1 10:15:1 10:16:1 10:17:1 10:18:1 10:19:1 10:20:1 10:21:1

(i) 10 rows are randomly selected and then 1 duplicate record for each of the rows was created with 12 mutations.

(ii) 10 rows are randomly selected (from the remaining 490 unchosen rows) and then 1 duplicate record for each of the rows was created with 13 mutations.

(iii) 10 rows are randomly selected (from the remaining 480 unchosen rows) and then 1 duplicate record for each of the rows was created with 14 mutations.

(iv) 10 rows are randomly selected (from the remaining 470 unchosen rows) and then 1 duplicate record for each of the rows was created with 15 mutations.

(v) 10 rows are randomly selected (from the remaining 460 unchosen rows) and then 1 duplicate record for each of the rows was created with 16 mutations.

(vi) 10 rows are randomly selected (from the remaining 450 unchosen rows) and then 1 duplicate record for each of the rows was created with 17 mutations.

(vii) 10 rows are randomly selected (from the remaining 440 unchosen rows) and then 1 duplicate record for each of the rows was created with 18 mutations.

(viii) 10 rows are randomly selected (from the remaining 430 unchosen rows) and then 1 duplicate record for each of the rows was created with 19 mutations.

(ix) 10 rows are randomly selected (from the remaining 420 unchosen rows) and then 1 duplicate record for each of the rows was created with 20 mutations.

(x) 10 rows are randomly selected (from the remaining 410 unchosen rows) and then 1 duplicate record for each of the rows was created with 21 mutations.

level	comp_specificity	false_negatives	false_positives	sensitivity	specificity	total_matches	total_non_matches	true_negatives	true_positives
0	1	0	49900	1	0	100	49900	0	100
0.083333	0.991242	0	49463	1	0.008758	100	49900	437	100
0.166667	0.904749	0	45147	1	0.095251	100	49900	4753	100
0.250000	0.701683	0	35014	1	0.298317	100	49900	14886	100
0.333333	0.406713	0	20295	1	0.593287	100	49900	29605	100
0.416667	0.161703	1	8069	0.990000	0.838297	100	49900	41831	99
0.500000	0.044890	5	2240	0.950000	0.955110	100	49900	47660	95
0.583333	0.009820	13	490	0.870000	0.990180	100	49900	49410	87
0.666667	0.001222	32	61	0.680000	0.998778	100	49900	49839	68
0.750000	0.000100	62	5	0.380000	0.999900	100	49900	49895	38
0.833333	0.000020	86	1	0.140000	0.999980	100	49900	49899	14
0.916667	0	97	0	0.030000	1	100	49900	49900	3
1	0	99	0	0.010000	1	100	49900	49900	1

Runtime	Comparisons	Comparisons per Second
16 seconds	50000	3125.0000

Table 11.6: Stats for dataset_A_500_org_100b_dup_ed.

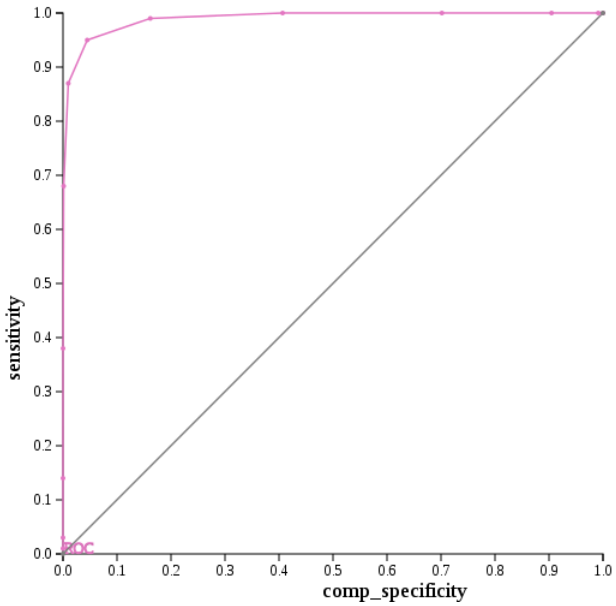


Figure 11.6: ROC curve for dataset_A_500_org_100b_dup_ed.

Example 11.7. Step three in breaking the dataset

dataset 1:	dataset_A_500_original
dataset 2:	dataset_A_from_500_org_100_dup_10221_10231_10241_10251_10261 _10271_10281_10291_10301_10311
method:	edit distance
algorithm threshold:	2.0
final dataset:	dataset_A_500_org_100c_dup_ed

Dataset 2 was formed by the following process: ./mutator.py dataset_A_500_original.csv rec_id dataset_A_from_500_org_100_dup_10221_10231_10241_10251_10261_10271_10281_10291_10301_10311 10:22:1 10:23:1 10:24:1 10:25:1 10:26:1 10:27:1 10:28:1 10:29:1 10:30:1 10:31:1

(i) 10 rows are randomly selected and then 1 duplicate record for each of the rows was created with 22 mutations.

(ii) 10 rows are randomly selected (from the remaining 490 unchosen rows) and then 1 duplicate record for each of the rows was created with 23 mutations.

(iii) 10 rows are randomly selected (from the remaining 480 unchosen rows) and then 1 duplicate record for each of the rows was created with 24 mutations.

(iv) 10 rows are randomly selected (from the remaining 470 unchosen rows) and then 1 duplicate record for each of the rows was created with 25 mutations.

(v) 10 rows are randomly selected (from the remaining 460 unchosen rows) and then 1 duplicate record for each of the rows was created with 26 mutations.

(vi) 10 rows are randomly selected (from the remaining 450 unchosen rows) and then 1 duplicate record for each of the rows was created with 27 mutations.

(vii) 10 rows are randomly selected (from the remaining 440 unchosen rows) and then 1 duplicate record for each of the rows was created with 28 mutations.

(viii) 10 rows are randomly selected (from the remaining 430 unchosen rows) and then 1 duplicate record for each of the rows was created with 29 mutations.

(ix) 10 rows are randomly selected (from the remaining 420 unchosen rows) and then 1 duplicate record for each of the rows was created with 30 mutations.

(x) 10 rows are randomly selected (from the remaining 410 unchosen rows) and then 1 duplicate record for each of the rows was created with 31 mutations.

level	comp_specificity	false_negatives	false_positives	sensitivity	specificity	total_matches	total_non_matches	true_negatives	true_positives
0	1	0	49900	1	0	100	49900	0	100
0.083333	0.986293	0	49216	1	0.013707	100	49900	684	100
0.166667	0.893267	0	44574	1	0.106733	100	49900	5326	100
0.250000	0.678838	1	33874	0.990000	0.321162	100	49900	16026	99
0.333333	0.405912	4	20255	0.960000	0.594088	100	49900	29645	96
0.416667	0.193948	10	9678	0.900000	0.806052	100	49900	40222	90
0.500000	0.074128	31	3699	0.690000	0.925872	100	49900	46201	69
0.583333	0.017555	56	876	0.440000	0.982445	100	49900	49024	44
0.666667	0.001603	80	80	0.200000	0.998397	100	49900	49820	20
0.750000	0.000180	94	9	0.060000	0.999820	100	49900	49891	6

Runtime	Comparisons	Comparisons per Second
15 seconds	50000	3333.3333

Table 11.7: Stats for dataset_A_500_org_100c_dup_ed.

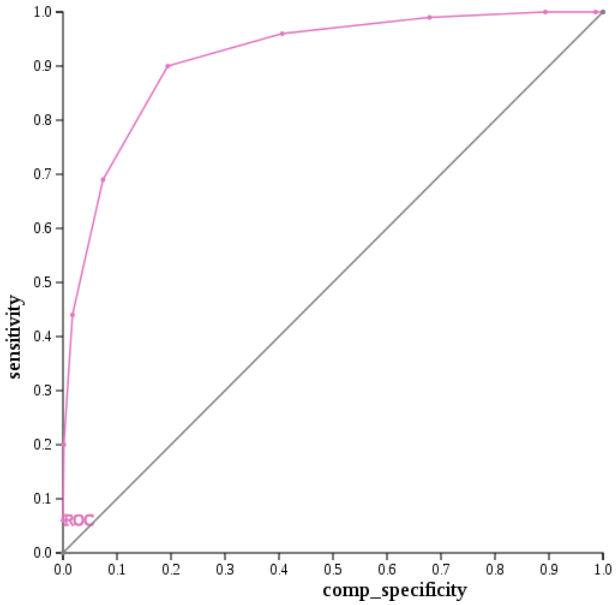


Figure 11.7: ROC curve for dataset_A_500_org_100c_dup_ed.

Example 11.8. Step four in breaking the dataset

dataset 1:	dataset_A_500_original
dataset 2:	dataset_A_from_500_org_100_dup_10321_10331_10341_10351_10361 _10371_10381_10391_10401_10411
method:	edit distance
algorithm threshold:	2.0
final dataset:	dataset_A_500_org_100d_dup_ed

Dataset 2 was formed by the following process: ./mutator.py dataset_A_500_original.csv rec_id dataset_A_from_500_org_100_dup_10321_10331_10341_10351_10361_10371_10381_10391_10401_10411 10:32:1 10:33:1 10:34:1 10:35:1 10:36:1 10:37:1 10:38:1 10:39:1 10:40:1 10:41:1

(i) 10 rows are randomly selected and then 1 duplicate record for each of the rows is created that has 32 mutations in it.

(ii) 10 rows are randomly selected (from the remaining 490 unchosen rows) and then 1 duplicate record for each of the rows was created with 33 mutations.

(iii) 10 rows are randomly selected (from the remaining 480 unchosen rows) and then 1 duplicate record for each of the rows was created with 34 mutations.

(iv) 10 rows are randomly selected (from the remaining 470 unchosen rows) and then 1 duplicate record for each of the rows was created with 35 mutations.

(v) 10 rows are randomly selected (from the remaining 460 unchosen rows) and then 1 duplicate record for each of the rows was created with 36 mutations.

(vi) 10 rows are randomly selected (from the remaining 450 unchosen rows) and then 1 duplicate record for each of the rows was created with 37 mutations.

(vii) 10 rows are randomly selected (from the remaining 440 unchosen rows) and then 1 duplicate record for each of the rows was created with 38 mutations.

(viii) 10 rows are randomly selected (from the remaining 430 unchosen rows) and then 1 duplicate record for each of the rows was created with 39 mutations.

(ix) 10 rows are randomly selected (from the remaining 420 unchosen rows) and then 1 duplicate record for each of the rows was created with 40 mutations.

(x) 10 rows are randomly selected (from the remaining 410 unchosen rows) and then 1 duplicate record for each of the rows was created with 41 mutations.

level	comp_specificity	false_negatives	false_positives	sensitivity	specificity	total_matches	total_non_matches	true_negatives	true_positives
0	1	0	49900	1	0	100	49900	0	100
0.083333	0.993287	0	49565	1	0.006713	100	49900	335	100
0.166667	0.940501	0	46931	1	0.059499	100	49900	2969	100
0.250000	0.791463	5	39494	0.950000	0.208537	100	49900	10406	95
0.333333	0.540381	16	26965	0.840000	0.459619	100	49900	22935	84
0.416667	0.299980	36	14969	0.640000	0.700020	100	49900	34931	64
0.500000	0.133487	58	6661	0.420000	0.866513	100	49900	43239	42
0.583333	0.051483	77	2569	0.230000	0.948517	100	49900	47331	23
0.666667	0.013687	93	683	0.070000	0.986313	100	49900	49217	7
0.750000	0.001844	98	92	0.020000	0.998156	100	49900	49808	2
0.833333	0.000140	99	7	0.010000	0.999860	100	49900	49893	1

Runtime	Comparisons	Comparisons per Second
15 seconds	50000	3333.3333

Table 11.8: Stats for dataset_A_500_org_100d_dup_ed.

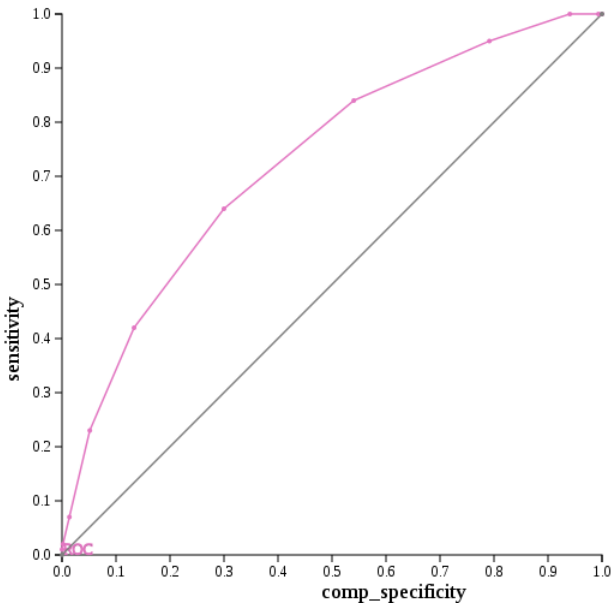


Figure 11.8: ROC curve for dataset_A_500_org_100d_dup_ed.

Edit distance was used to compare each pair of datasets. Since we are using a character-based similarity metric to compare the two databases, letters, numbers, and special characters were used to mutate the original data entries to create duplicates. From dataset dataset_A_500_original only twenty percent of the original entries were used to make duplicates. Each duplicate corresponded to one original entry and the duplicate had between two and forty-one mutations performed on it. As the number of mutations increases, the ROC curve moves closer to the line of no-discrimination, and looks more like an actual curve.

11.3 Comparing Character-Based and Phonetic-Based Similarity Metrics When There is One Duplicate Entry With Multiple Mutations

We now know that mutating the duplicates has an effect on the ROC curve. This raises the question how much of an effect does mutating the duplicates have and does the number of duplicates play a role in the shape of the ROC curve as well? We also want to compare a character-based similarity metric with a phonetic-based similarity metric as the number of mutations and duplicates are manipulated. This will show us whether the two metrics perform similarly under these varying conditions, and how much of an effect the number of mutations and duplicates have on performance.

In this first case dataset one will consist of 2500 original entries and dataset set two will consist of 250 duplicate entries. The duplicate entries have been mutated two to forty-one times. Mutations were formed by inserting a single letter, deleting a single letter, replacing a single letter, tranposing two letters, transposing two words, or dropping a field.

Example 11.9. Step one in breaking the dataset with a character-based similarity metric

dataset 1:	dataset_A_2500_original
dataset 2:	dataset_A_from_2500_org_250_dup_25_1_2_11
method:	q-gram distance
algorithm parameters:	1 (the size of q)
algorithm threshold:	0.75 (lower bound for match – upper bound is automatically set to 1.0)
final dataset:	dataset_A_2500_org_250a_dup_qgd

Dataset 2 was formed by the following process: ./mutator.py dataset_A_2500_original.csv rec_id dataset_A_from_2500_org_250_dup_25_1_2_11 25:2:1 25:3:1 25:4:1 25:5:1 25:6:1 25:7:1 25:8:1 25:9:1 25:10:1 25:11:1

level	comp_specificity	false_negatives	false_positives	sensitivity	specificity	total_matches	total_non_matches	true_negatives	true_positives
0	1	0	624750	1	0	250	624750	0	250
0.083333	0.317570	0	198402	1	0.682430	250	624750	426348	250
0.166667	0.045863	0	28653	1	0.954137	250	624750	596097	250
0.250000	0.003585	0	2240	1	0.996415	250	624750	622510	250
0.333333	0.000158	0	99	1	0.999842	250	624750	624651	250
0.416667	0.000005	0	3	1	0.999995	250	624750	624747	250
0.500000	0	1	0	0.996000	1	250	624750	624750	249
0.583333	0	11	0	0.956000	1	250	624750	624750	239
0.666667	0	30	0	0.880000	1	250	624750	624750	220
0.750000	0	77	0	0.692000	1	250	624750	624750	173
0.833333	0	145	0	0.420000	1	250	624750	624750	105
0.916667	0	202	0	0.192000	1	250	624750	624750	48
1	0	241	0	0.036000	1	250	624750	624750	9

Runtime	Comparisons	Comparisons per Second
93 seconds	625000	6720.4301

Table 11.9: Stats for dataset_A_2500_org_250a_dup_qgd.

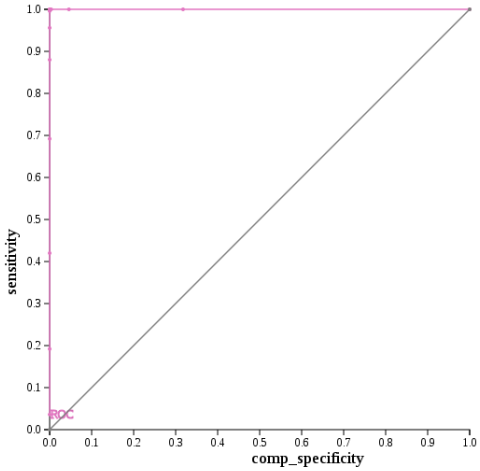


Figure 11.9: ROC curve for dataset_A_2500_org_250a_dup_qgd.

Example 11.10. Step one in breaking the dataset with a phonetic-based similarity metric

dataset 1: dataset_A_2500_original

dataset 2: dataset_A_from_2500_org_250_dup_25_1_2_11

method: NYSIIS

final dataset: dataset_A_2500_org_250a_dup_nysiis

Dataset 2 was formed by the following process: ./mutator.py dataset_A_2500_original.csv rec_id dataset_A_from_2500_org_250_dup_25_1_2_11 25:2:1 25:3:1 25:4:1 25:5:1 25:6:1 25:7:1 25:8:1 25:9:1 25:10:1 25:11:1

level	comp_specificity	false_negatives	false_positives	sensitivity	specificity	total_matches	total_non_matches	true_negatives	true_positives
0	1	0	624750	1	0	250	624750	0	250
0.200000	0.133812	4	83599	0.984000	0.866188	250	624750	541151	246
0.400000	0.001681	15	1050	0.940000	0.998319	250	624750	623700	235
0.600000	0.000011	57	7	0.772000	0.999989	250	624750	624743	193
0.800000	0	139	0	0.444000	1	250	624750	624750	111
1	0	213	0	0.148000	1	250	624750	624750	37

Runtime	Comparisons	Comparisons per Second
30 seconds	625000	20833.3333

Table 11.10: Stats for dataset_A_2500_org_250a_dup_nysiis.

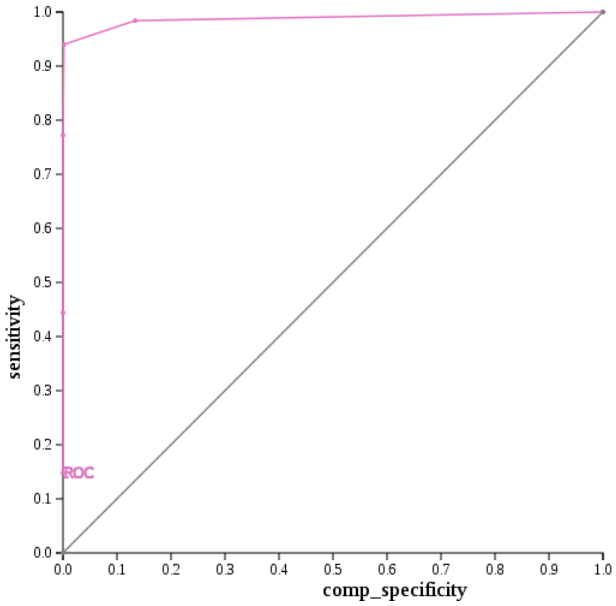


Figure 11.10: ROC curve for dataset_A_2500_org_250a_dup_nysiis.

Example 11.11. Step two in breaking the dataset with a character-based similarity metric

dataset 1:	dataset_A_2500_original
dataset 2:	dataset_A_from_2500_org_250_dup_25_1_12_21
method:	q-gram distance
algorithm parameters:	1 (the size of q)
algorithm threshold:	0.75 (lower bound for match – upper bound is automatically set to 1.0)
final dataset:	dataset_A_2500_org_250b_dup_qgd

Dataset 2 was formed by the following process: ./mutator.py dataset_A_2500_original.csv rec_id dataset_A_from_2500_org_250_dup_25_1_12_21 25:12:1 25:13:1 25:14:1 25:15:1 25:16:1 25:17:1 25:18:1 25:19:1 25:20:1 25:21:1

level	comp_specificity	false_negatives	false_positives	sensitivity	specificity	total_matches	total_non_matches	true_negatives	true_positives
0	1	0	624750	1	0	250	624750	0	250
0.083333	0.214580	0	134059	1	0.785420	250	624750	490691	250
0.166667	0.018948	0	11838	1	0.981052	250	624750	612912	250
0.250000	0.000869	2	543	0.992000	0.999131	250	624750	624207	248
0.333333	0.000026	6	16	0.976000	0.999974	250	624750	624734	244
0.416667	0.000002	18	1	0.928000	0.999998	250	624750	624749	232
0.500000	0	45	0	0.820000	1	250	624750	624750	205
0.583333	0	98	0	0.608000	1	250	624750	624750	152
0.666667	0	158	0	0.368000	1	250	624750	624750	92
0.750000	0	210	0	0.160000	1	250	624750	624750	40
0.833333	0	236	0	0.056000	1	250	624750	624750	14
0.916667	0	248	0	0.008000	1	250	624750	624750	2

Runtime	Comparisons	Comparisons per Second
88 seconds	625000	7102.2727

Table 11.11: Stats for dataset_A_2500_org_250b_dup_qgd.

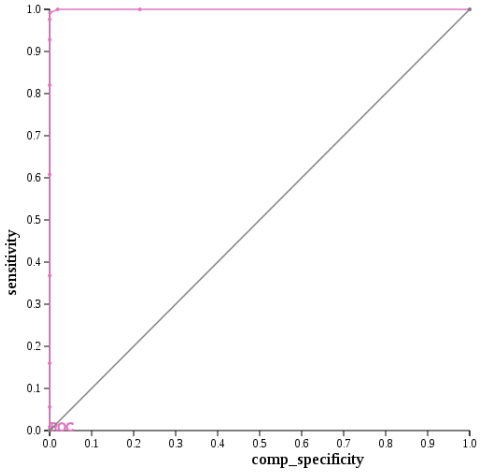


Figure 11.11: ROC curve for dataset_A_2500_org_250b_dup_qgd.

Example 11.12. Step two in breaking the dataset with a phonetic-based similarity metric

dataset 1: dataset_A_2500_original

dataset 2: dataset_A_from_2500_org_250_dup_25_1_12_21

method: NYSIIS

final dataset: dataset_A_2500_org_250b_dup_nysiis

Dataset 2 was formed by the following process: ./mutator.py dataset_A_2500_original.csv rec_id dataset_A_from_2500_org_250_dup_25_1_12_21 25:12:1 25:13:1 25:14:1 25:15:1 25:16:1 25:17:1 25:18:1 25:19:1 25:20:1 25:21:1

level	comp_specificity	false_negatives	false_positives	sensitivity	specificity	total_matches	total_non_matches	true_negatives	true_positives
0	1	0	624750	1	0	250	624750	0	250
0.200000	0.084579	29	52841	0.884000	0.915421	250	624750	571909	221
0.400000	0.000994	116	621	0.536000	0.999006	250	624750	624129	134
0.600000	0.000002	204	1	0.184000	0.999998	250	624750	624749	46
0.800000	0	240	0	0.040000	1	250	624750	624750	10

Runtime	Comparisons	Comparisons per Second
29 seconds	625000	21551.7241

Table 11.12: Stats for dataset_A_2500_org_250b_dup_nysiis.

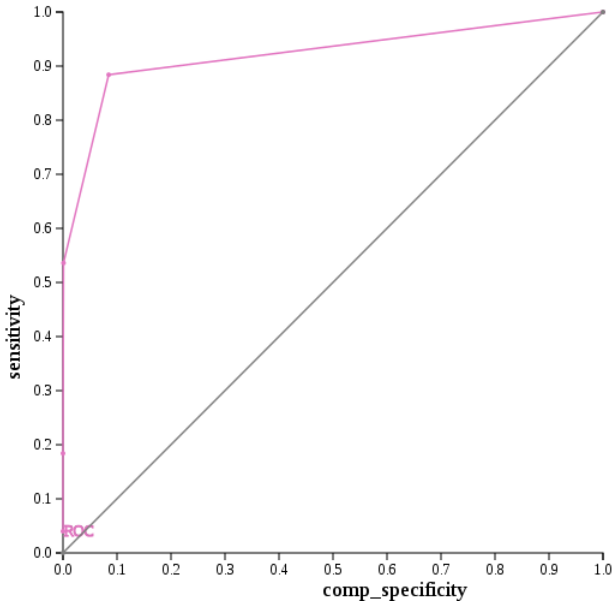


Figure 11.12: ROC curve for dataset_A_2500_org_250b_dup_nysiis.

Example 11.13. Step three in breaking the dataset with a character-based similarity metric

dataset 1:	dataset_A_2500_original
dataset 2:	dataset_A_from_2500_org_250_dup_25_1_22_31
method:	q-gram distance
algorithm parameters:	1 (the size of q)
algorithm threshold:	0.75 (lower bound for match – upper bound is automatically set to 1.0)
final dataset:	dataset_A_2500_org_250c_dup_qgd

Dataset 2 was formed by the following process: ./mutator.py dataset_A_2500_original.csv rec_id dataset_A_from_2500_org_250_dup_25_1_22_31 25:22:1 25:23:1 25:24:1 25:25:1 25:26:1 25:27:1 25:28:1 25:29:1 25:30:1 25:31:1

level	comp_specificity	false_negatives	false_positives	sensitivity	specificity	total_matches	total_non_matches	true_negatives	true_positives
0	1	0	624750	1	0	250	624750	0	250
0.083333	0.142359	0	88939	1	0.857641	250	624750	535811	250
0.166667	0.007856	1	4908	0.996000	0.992144	250	624750	619842	249
0.250000	0.000266	11	166	0.956000	0.999734	250	624750	624584	239
0.333333	0.000002	41	1	0.836000	0.999998	250	624750	624749	209
0.416667	0	93	0	0.628000	1	250	624750	624750	157
0.500000	0	147	0	0.412000	1	250	624750	624750	103
0.583333	0	196	0	0.216000	1	250	624750	624750	54
0.666667	0	227	0	0.092000	1	250	624750	624750	23
0.750000	0	245	0	0.020000	1	250	624750	624750	5

Runtime	Comparisons	Comparisons per Second
88 seconds	625000	7102.2727

Table 11.13: Stats for dataset_A_2500_org_250c_dup_qgd.

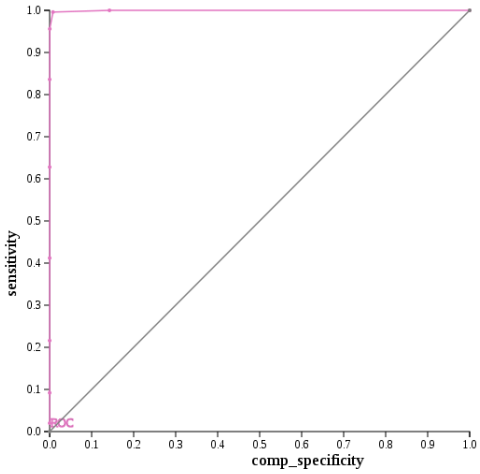


Figure 11.13: ROC curve for dataset_A_2500_org_250c_dup_qgd.

Example 11.14. Step three in breaking the dataset with a phonetic-based similarity metric

dataset 1: dataset_A_2500_original

dataset 2: dataset_A_from_2500_org_250_dup_25_1_22_31

method: NYSIIS

final dataset: dataset_A_2500_org_250c_dup_nysiis

Dataset 2 was formed by the following process: ./mutator.py dataset_A_2500_original.csv rec_id dataset_A_from_2500_org_250_dup_25_1_22_31 25:22:1 25:23:1 25:24:1 25:25:1 25:26:1 25:27:1 25:28:1 25:29:1 25:30:1 25:31:1

level	comp_specificity	false_negatives	false_positives	sensitivity	specificity	total_matches	total_non_matches	true_negatives	true_positives
0	1	0	624750	1	0	250	624750	0	250
0.200000	0.060154	89	37581	0.644000	0.939846	250	624750	587169	161
0.400000	0.000659	193	412	0.228000	0.999341	250	624750	624338	57
0.600000	0	247	0	0.012000	1	250	624750	624750	3

Runtime	Comparisons	Comparisons per Second
29 seconds	625000	21551.7241

Table 11.14: Stats for dataset_A_2500_org_250c_dup_nysiis.

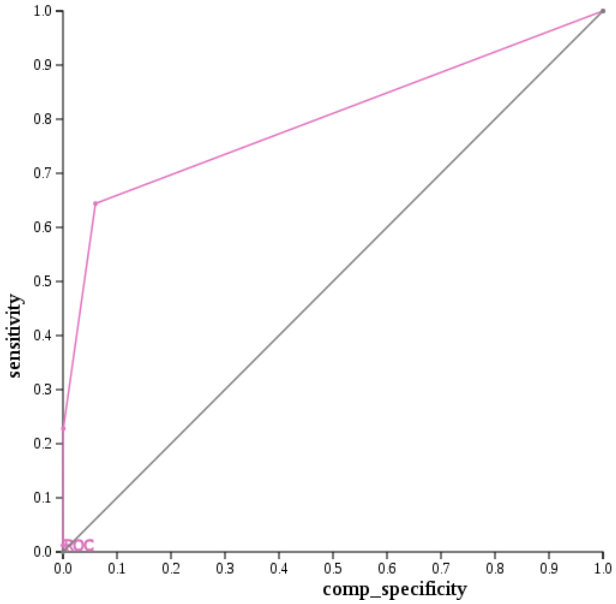


Figure 11.14: ROC curve for dataset_A_2500_org_250c_dup_nysiis.

Example 11.15. Step four in breaking the dataset with a character-based similarity metric

dataset 1:	dataset_A_2500_original
dataset 2:	dataset_A_from_2500_org_250_dup_25_1_32_41
method:	q-gram distance
algorithm parameters:	1 (the size of q)
algorithm threshold:	0.75 (lower bound for match – upper bound is automatically set to 1.0)
final dataset:	dataset_A_2500_org_250d_dup_qgd

Dataset 2 was formed by the following process: ./mutator.py dataset_A_2500_original.csv rec_id dataset_A_from_2500_org_250_dup_25_1_32_41 25:32:1 25:33:1 25:34:1 25:35:1 25:36:1 25:37:1 25:38:1 25:39:1 25:40:1 25:41:1

level	comp_specificity	false_negatives	false_positives	sensitivity	specificity	total_matches	total_non_matches	true_negatives	true_positives
0	1	0	624750	1	0	250	624750	0	250
0.083333	0.089111	2	55672	0.992000	0.910889	250	624750	569078	248
0.166667	0.002985	13	1865	0.948000	0.997015	250	624750	622885	237
0.250000	0.000054	65	34	0.740000	0.999946	250	624750	624716	185
0.333333	0	129	0	0.484000	1	250	624750	624750	121
0.416667	0	197	0	0.212000	1	250	624750	624750	53
0.500000	0	227	0	0.092000	1	250	624750	624750	23
0.583333	0	245	0	0.020000	1	250	624750	624750	5
0.666667	0	249	0	0.004000	1	250	624750	624750	1

Runtime	Comparisons	Comparisons per Second
87 seconds	625000	7183.9080

Table 11.15: Stats for dataset_A_2500_org_250d_dup_qgd.

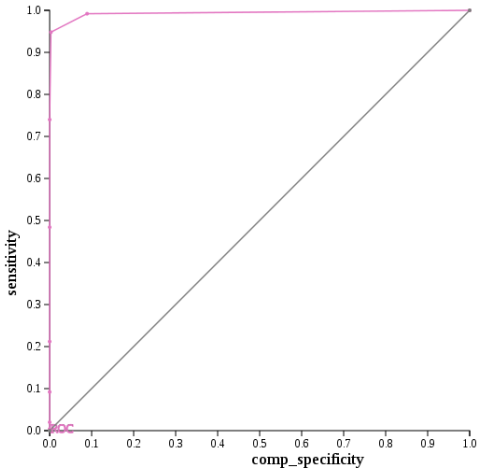


Figure 11.15: ROC curve for dataset_A_2500_org_250d_dup_qgd.

Example 11.16. Step four in breaking the dataset with a phonetic-based similarity metric

dataset 1: dataset_A_2500_original

dataset 2: dataset_A_from_2500_org_250_dup_25_1_32_41

method: NYSIIS

final dataset: dataset_A_2500_org_250d_dup_nysiis

Dataset 2 was formed by the following process: ./mutator.py dataset_A_2500_original.csv rec_id dataset_A_from_2500_org_250_dup_25_1_32_41 25:32:1 25:33:1 25:34:1 25:35:1 25:36:1 25:37:1 25:38:1 25:39:1 25:40:1 25:41:1

level	comp_specificity	false_negatives	false_positives	sensitivity	specificity	total_matches	total_non_matches	true_negatives	true_positives
0	1	0	624750	1	0	250	624750	0	250
0.200000	0.055920	149	34936	0.404000	0.944080	250	624750	589814	101
0.400000	0.000667	237	417	0.052000	0.999333	250	624750	624333	13
0.600000	0.000003	249	2	0.004000	0.999997	250	624750	624748	1

Runtime	Comparisons	Comparisons per Second
31 seconds	625000	20161.2903

Table 11.16: Stats for dataset_A_2500_org_250d_dup_nysiis.

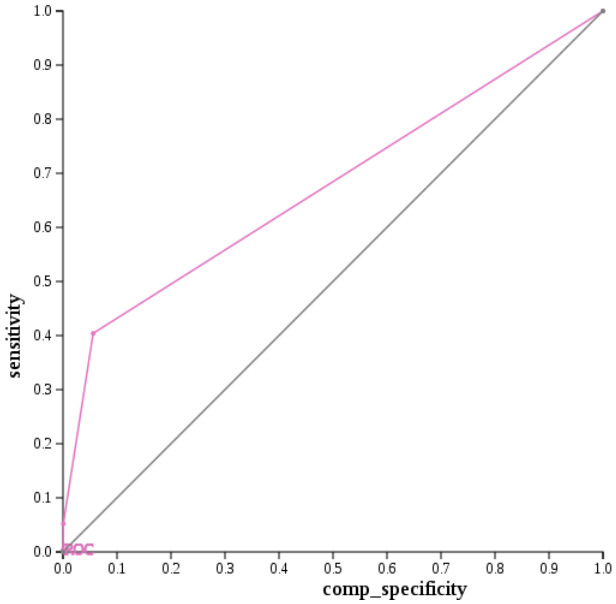


Figure 11.16: ROC curve for dataset_A_2500_org_250d_dup_nysiis.

Looking over the ROC curves for q-gram distance and NYSIIS, the ROC curves for q-gram distance stayed more consistent and had points closer to (0,1), while the ROC curves for NYSIIS kept getting closer and closer to the line of no-discrimination as the number of mutations increased. I believe that we are seeing these results because q-gram distance is a character-based similarity metric, so it looks at the individual characters that make up each word in each field, whereas NYSIIS is a phonetic-based similarity metric, so it looks at each word as a whole in each field. Since the mutations in each duplicate entry were put in randomly and phoneticly change the makeup of a string, NYSIIS has a harder time knowing when two strings are the same. Q-gram distance's ability to look at individual characters and substrings helps it see when a change has been made, but if duplicate entries have been mutated too much it has a difficult time finding true matches.

11.4 Comparing Character-Based and Phonetic-Based Similarity Metrics When There are a Varying Number of Duplicate Entries With Multiple Mutations

With the previous examples the character-based similarity metric performed better in comparison to the phonetic-based similarity metric where each original entry in dataset one had one duplicate entry in dataset two. Recall that in the previous series the duplicate entries ranged from being manipulated two times to forty-one times. In this series of examples we will look at what happens when the originals in the first dataset have multiple duplicates in the mutated dataset. This series will show us whether or not the character-based similarity metric continues to out-perform the phonetic-based similarity metric as the number of duplicates and mutations change.

Example 11.17. Step one in breaking the dataset with a character-based similarity metric

dataset 1:	dataset_A_2500_original
dataset 2:	dataset_A_from_2500_org_5000_dup_25_20_2_11
method:	q-gram distance
algorithm parameters:	1 (the size of q)
algorithm threshold:	0.75 (lower bound for match – upper bound is automatically set to 1.0)
final dataset:	dataset_A_2500_org_5000_dup_qgd

Dataset 2 was formed by the following process: ./mutator.py dataset_A_2500_original.csv rec_id dataset_A_from_2500_org_5000_dup_25_20_2_11 25:2:20 25:3:20 25:4:20 25:5:20 25:6:20 25:7:20 25:8:20 25:9:20 25:10:20 25:11:20

level	comp_specificity	false_negatives	false_positives	sensitivity	specificity	total_matches	total_non_matches	true_negatives	true_positives
0	1	0	12494150	1	0	5850	12494150	0	5850
0.083333	0.320783	597	4007908	0.897949	0.679217	5850	12494150	8486242	5253
0.166667	0.046266	823	578057	0.859316	0.953734	5850	12494150	11916093	5027
0.250000	0.003548	847	44328	0.855214	0.996452	5850	12494150	12449822	5003
0.333333	0.000146	850	1818	0.854701	0.999854	5850	12494150	12492332	5000
0.416667	0.000002	855	31	0.853846	0.999998	5850	12494150	12494119	4995
0.500000	0	896	0	0.846838	1	5850	12494150	12494150	4954
0.583333	0	1053	0	0.820000	1	5850	12494150	12494150	4797
0.666667	0	1473	0	0.748205	1	5850	12494150	12494150	4377
0.750000	0	2344	0	0.599316	1	5850	12494150	12494150	3506
0.833333	0	3690	0	0.369231	1	5850	12494150	12494150	2160
0.916667	0	4954	0	0.153162	1	5850	12494150	12494150	896
1	0	5659	0	0.032650	1	5850	12494150	12494150	191

Runtime	Comparisons	Comparisons per Second
1781 seconds	12500000	7018.5289

Table 11.17: Stats for dataset_A_2500_org_5000_dup_qgd.

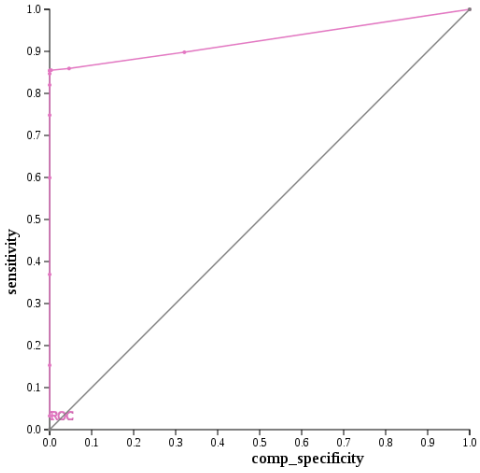


Figure 11.17: ROC curve for dataset_A_2500_org_5000_dup_qgd.

Example 11.18. Step one in breaking the dataset with a phonetic-based similarity metric

dataset 1: dataset_A_2500_original

dataset 2: dataset_A_from_2500_org_5000_dup_25_20_2_11

method: NYSIIS

final dataset: dataset_A_2500_org_5000_dup_nysiis

Dataset 2 was formed by the following process: ./mutator.py dataset_A_2500_original.csv rec_id dataset_A_from_2500_org_5000_dup_25_20_2_11 25:2:20 25:3:20 25:4:20 25:5:20 25:6:20 25:7:20 25:8:20 25:9:20 25:10:20 25:11:20

level	comp_specificity	false_negatives	false_positives	sensitivity	specificity	total_matches	total_non_matches	true_negatives	true_positives
0	1	0	12494150	1	0	5850	12494150	0	5850
0.200000	0.138755	813	1733630	0.861026	0.861245	5850	12494150	10760520	5037
0.400000	0.001435	1142	17934	0.804786	0.998565	5850	12494150	12476216	4708
0.600000	0.000004	2055	50	0.648718	0.999996	5850	12494150	12494100	3795
0.800000	0	3645	0	0.376923	1	5850	12494150	12494150	2205
1	0	5178	0	0.114872	1	5850	12494150	12494150	672

Runtime	Comparisons	Comparisons per Second
563 seconds	12500000	22202.4867

Table 11.18: Stats for dataset_A_2500_org_5000_dup_nysiis.

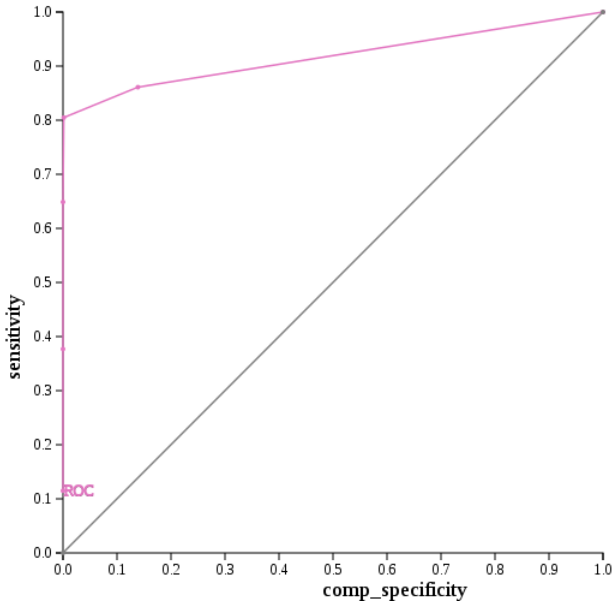


Figure 11.18: ROC curve for dataset_A_2500_org_5000_dup_nysiis.

Example 11.19. Step two in breaking the dataset with a character-based similarity metric

dataset 1:	dataset_A_2500_original
dataset 2:	dataset_A_from_2500_org_3750_dup_25_15_12_21
method:	q-gram distance
algorithm parameters:	1 (the size of q)
algorithm threshold:	0.75 (lower bound for match – upper bound is automatically set to 1.0)
final dataset:	dataset_A_2500_org_3750_dup_qgd

Dataset 2 was formed by the following process: ./mutator.py dataset_A_2500_original.csv rec_id dataset_A_from_2500_org_3750_dup_25_15_12_21 25:12:15 25:13:15 25:14:15 25:15:15 25:16:15 25:17:15 25:18:15 25:19:15 25:20:15 25:21:15

level	comp_specificity	false_negatives	false_positives	sensitivity	specificity	total_matches	total_non_matches	true_negatives	true_positives
0	1	0	9370475	1	0	4525	9370475	0	4525
0.083333	0.212296	618	1989312	0.863425	0.787704	4525	9370475	7381163	3907
0.166667	0.019023	758	178258	0.832486	0.980977	4525	9370475	9192217	3767
0.250000	0.000862	785	8081	0.826519	0.999138	4525	9370475	9362394	3740
0.333333	0.000019	836	175	0.815249	0.999981	4525	9370475	9370300	3689
0.416667	0.000000	1020	1	0.774586	1.000000	4525	9370475	9370474	3505
0.500000	0	1460	0	0.677348	1	4525	9370475	9370475	3065
0.583333	0	2241	0	0.504751	1	4525	9370475	9370475	2284
0.666667	0	3125	0	0.309392	1	4525	9370475	9370475	1400
0.750000	0	3894	0	0.139448	1	4525	9370475	9370475	631
0.833333	0	4340	0	0.040884	1	4525	9370475	9370475	185
0.916667	0	4493	0	0.007072	1	4525	9370475	9370475	32

Runtime	Comparisons	Comparisons per Second
1315 seconds	9375000	7129.2776

Table 11.19: Stats for dataset_A_2500_org_3750_dup_qgd.

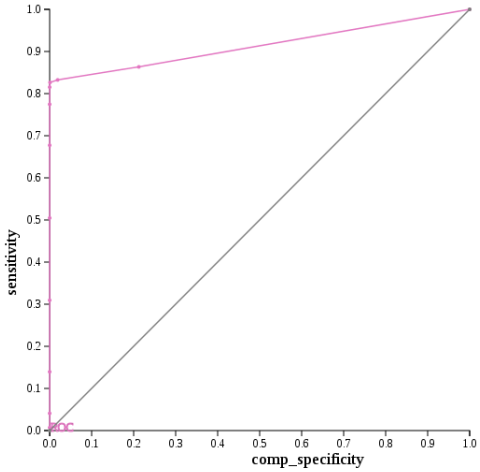


Figure 11.19: ROC curve for dataset_A_2500_org_3750_dup_qgd.

Example 11.20. Step two in breaking the dataset with a phonetic-based similarity metric

dataset 1: dataset_A_2500_original

dataset 2: dataset_A_from_2500_org_3750_dup_25_15_12_21

method: NYSIIS

final dataset: dataset_A_2500_org_3750_dup_nysiis

Dataset 2 was formed by the following process: ./mutator.py dataset_A_2500_original.csv rec_id dataset_A_from_2500_org_3750_dup_25_15_12_21 25:12:15 25:13:15 25:14:15 25:15:15 25:16:15 25:17:15 25:18:15 25:19:15 25:20:15 25:21:15

level	comp_specificity	false_negatives	false_positives	sensitivity	specificity	total_matches	total_non_matches	true_negatives	true_positives
0	1	0	9370475	1	0	4525	9370475	0	4525
0.200000	0.086135	1155	807123	0.744751	0.913865	4525	9370475	8563352	3370
0.400000	0.000955	2507	8950	0.445967	0.999045	4525	9370475	9361525	2018
0.600000	0.000003	3766	29	0.167735	0.999997	4525	9370475	9370446	759
0.800000	0	4368	0	0.034696	1	4525	9370475	9370475	157
1	0	4510	0	0.003315	1	4525	9370475	9370475	15

Runtime	Comparisons	Comparisons per Second
423 seconds	9375000	22163.1206

Table 11.20: Stats for dataset_A_2500_org_3750_dup_nysiis.

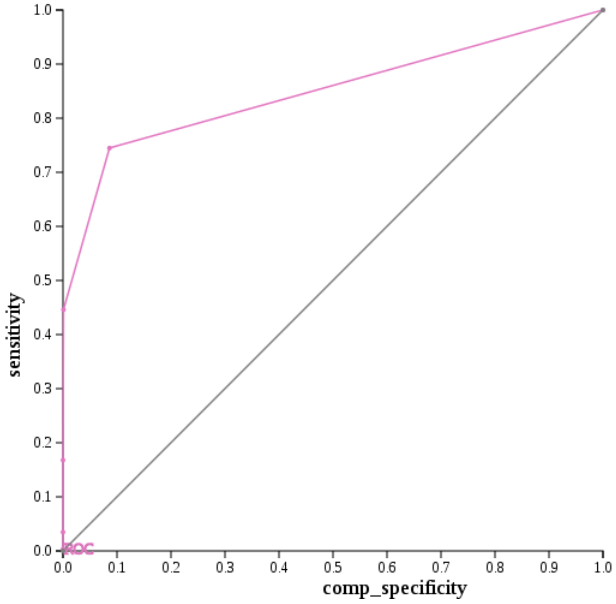


Figure 11.20: ROC curve for dataset_A_2500_org_3750_dup_nysiis.

Example 11.21. Step three in breaking the dataset with a character-based similarity metric

dataset 1:	dataset_A_2500_original
dataset 2:	dataset_A_from_2500_org_1250_dup_25_5_22_31
method:	q-gram distance
algorithm parameters:	1 (the size of q)
algorithm threshold:	0.75 (lower bound for match – upper bound is automatically set to 1.0)
final dataset:	dataset_A_2500_org_1250_dup_qgd

Dataset 2 was formed by the following process: ./mutator.py dataset_A_2500_original.csv rec_id dataset_A_from_2500_org_1250_dup_25_5_22_31 25:22:5 25:23:5 25:24:5 25:25:5 25:26:5 25:27:5 25:28:5 25:29:5 25:30:5 25:31:5

level	comp_specificity	false_negatives	false_positives	sensitivity	specificity	total_matches	total_non_matches	true_negatives	true_positives
0	1	0	3123750	1	0	1250	3123750	0	1250
0.083333	0.145462	2	454386	0.998400	0.854538	1250	3123750	2669364	1248
0.166667	0.008408	15	26266	0.988000	0.991592	1250	3123750	3097484	1235
0.250000	0.000242	69	755	0.944800	0.999758	1250	3123750	3122995	1181
0.333333	0.000004	216	14	0.827200	0.999996	1250	3123750	3123736	1034
0.416667	0	456	0	0.635200	1	1250	3123750	3123750	794
0.500000	0	779	0	0.376800	1	1250	3123750	3123750	471
0.583333	0	1028	0	0.177600	1	1250	3123750	3123750	222
0.666667	0	1174	0	0.060800	1	1250	3123750	3123750	76
0.750000	0	1226	0	0.019200	1	1250	3123750	3123750	24
0.833333	0	1247	0	0.002400	1	1250	3123750	3123750	3

Runtime	Comparisons	Comparisons per Second
436 seconds	3125000	7167.4312

Table 11.21: Stats for dataset_A_2500_org_1250_dup_qgd.

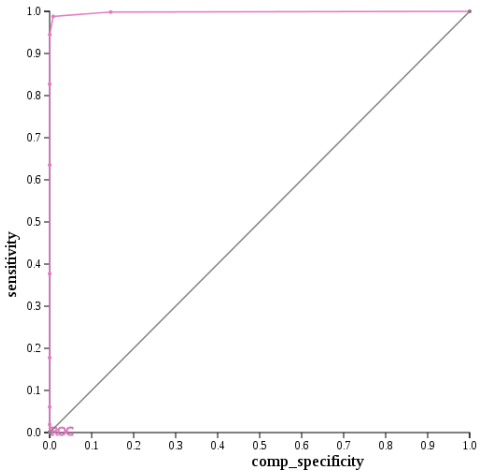


Figure 11.21: ROC curve for dataset_A_2500_org_1250_dup_qgd.

Example 11.22. Step three in breaking the dataset with a phonetic-based similarity metric

dataset 1: dataset_A_2500_original

dataset 2: dataset_A_from_2500_org_1250_dup_25_5_22_31

method: NYSIIS

final dataset: dataset_A_2500_org_1250_dup_nysiis

Dataset 2 was formed by the following process: ./mutator.py dataset_A_2500_original.csv rec_id dataset_A_from_2500_org_1250_dup_25_5_22_31 25:22:5 25:23:5 25:24:5 25:25:5 25:26:5 25:27:5 25:28:5 25:29:5 25:30:5 25:31:5

level	comp_specificity	false_negatives	false_positives	sensitivity	specificity	total_matches	total_non_matches	true_negatives	true_positives
0	1	0	3123750	1	0	1250	3123750	0	1250
0.200000	0.069344	475	216612	0.620000	0.930656	1250	3123750	2907138	775
0.400000	0.000841	996	2626	0.203200	0.999159	1250	3123750	3121124	254
0.600000	0.000003	1221	10	0.023200	0.999997	1250	3123750	3123740	29
0.800000	0	1249	0	0.000800	1	1250	3123750	3123750	1

Runtime	Comparisons	Comparisons per Second
141 seconds	3125000	22163.1206

Table 11.22: Stats for dataset_A_2500_org_1250_dup_nysiis.

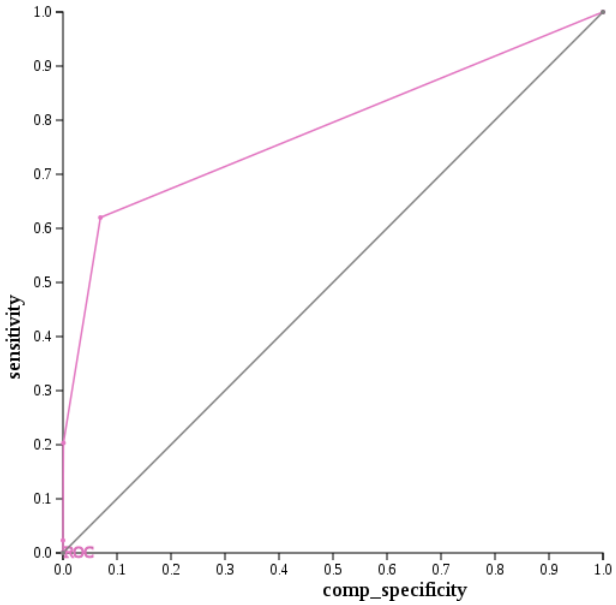


Figure 11.22: ROC curve for dataset_A_2500_org_1250_dup_nysiis.

Example 11.23. Step four in breaking the dataset with a character-based similarity metric

dataset 1:	dataset_A_2500_original
dataset 2:	dataset_A_from_2500_org_500_dup_25_2_32_41
method:	q-gram distance
algorithm parameters:	1 (the size of q)
algorithm threshold:	0.75 (lower bound for match – upper bound is automatically set to 1.0)
final dataset:	dataset_A_2500_org_500_dup_qgd

Dataset 2 was formed by the following process: ./mutator.py dataset_A_2500_original.csv rec_id dataset_A_from_2500_org_500_dup_25_2_32_41 25:32:2 25:33:2 25:34:2 25:35:2 25:36:2 25:37:2 25:38:2 25:39:2 25:40:2 25:41:2

level	comp_specificity	false_negatives	false_positives	sensitivity	specificity	total_matches	total_non_matches	true_negatives	true_positives
0	1	0	1249500	1	0	500	1249500	0	500
0.083333	0.093437	9	116750	0.982000	0.906563	500	1249500	1132750	491
0.166667	0.003409	44	4260	0.912000	0.996591	500	1249500	1245240	456
0.250000	0.000077	127	96	0.746000	0.999923	500	1249500	1249404	373
0.333333	0.000001	257	1	0.486000	0.999999	500	1249500	1249499	243
0.416667	0	372	0	0.256000	1	500	1249500	1249500	128
0.500000	0	451	0	0.098000	1	500	1249500	1249500	49
0.583333	0	486	0	0.028000	1	500	1249500	1249500	14
0.666667	0	497	0	0.006000	1	500	1249500	1249500	3

Runtime	Comparisons	Comparisons per Second
172 seconds	1250000	7267.4419

Table 11.23: Stats for dataset_A_2500_org_500_dup_qgd.

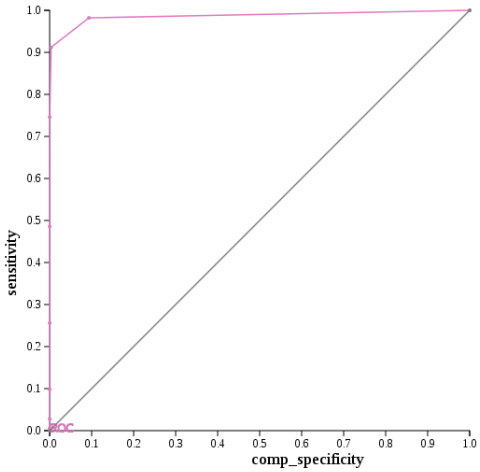


Figure 11.23: ROC curve for dataset_A_2500_org_500_dup_qgd.

Example 11.24. Step four in breaking the dataset with a phonetic-based similarity metric

dataset 1: dataset_A_2500_original

dataset 2: dataset_A_from_2500_org_500_dup_25_2_32_41

method: NYSIIS

final dataset: dataset_A_2500_org_500_dup_nysiis

Dataset 2 was formed by the following process: ./mutator.py dataset_A_2500_original.csv rec_id dataset_A_from_2500_org_500_dup_25_2_32_41 25:32:2 25:33:2 25:34:2 25:35:2 25:36:2 25:37:2 25:38:2 25:39:2 25:40:2 25:41:2

level	comp_specificity	false_negatives	false_positives	sensitivity	specificity	total_matches	total_non_matches	true_negatives	true_positives
0	1	0	1249500	1	0	500	1249500	0	500
0.200000	0.059808	291	74730	0.418000	0.940192	500	1249500	1174770	209
0.400000	0.000819	460	1023	0.080000	0.999181	500	1249500	1248477	40
0.600000	0.000004	498	5	0.004000	0.999996	500	1249500	1249495	2

Runtime	Comparisons	Comparisons per Second
58 seconds	1250000	21551.7241

Table 11.24: Stats for dataset_A_2500_org_500_dup_nysiis.

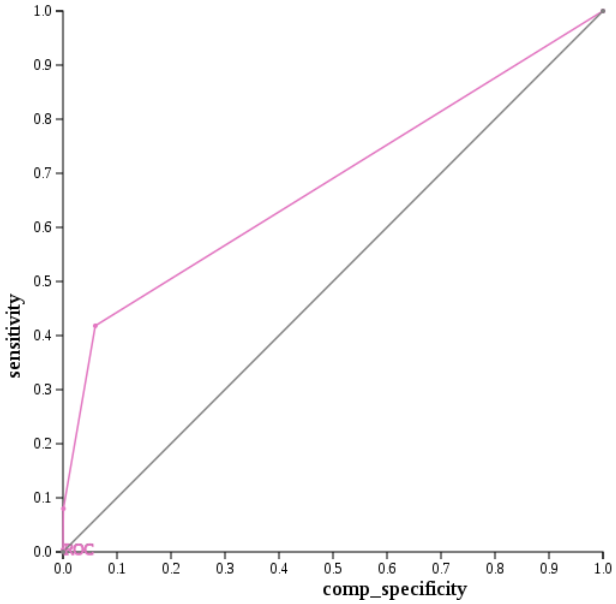


Figure 11.24: ROC curve for dataset_A_2500_org_500_dup_nysiis.

The ROC curves for when the second dataset is dataset_A_from_2500_org_5000_dup_25_20_2_11 and dataset_A_from_2500_org_3750_dup_25_15_12_21 look fairly similar for both q-gram distance and NYSIIS. Neither of them have a point close to the perfect classification point (0,1) on the ROC curve, but all of the ROC curves stay way above the line of no-discrimination. However, when the mutated dataset is dataset_A_from_2500_org_1250_dup_25_5_22_31 and dataset_A_from_2500_org_500_dup_25_2_32_41 the ROC curves for the character-based similarity metric and the phonetic-based similarity metric do not look similar. In fact, q-gram distance has a better ROC curve in both of the latter cases than NYSIIS does. In the last two cases there were not many duplicates, but there were a high number of mutations. The fact that the ROC curves looked similar when there were a high number of duplicates and a low numbers of mutations, this then leads us to examine the effects of a high number of duplicates and mutations.

11.5 Comparing Character-Based and Phonetic-Based Similarity Metrics When There are Multiple Duplicate Entries With Multiple Mutations

We have already looked at what happens when dataset two has one duplicate for each original entry with two to forty-one mutations and when dataset two has a varying number of duplicate entries. This last series of examples will than allow us to look at what happens when all the originals have a large number of duplicates and manipulations performed on them.

Example 11.25. Step one in breaking the dataset with a character-based similarity metric

dataset 1:	dataset_A_2500_original
dataset 2:	dataset_A_from_2500_org_2500_dup_25_10_2_11
method:	q-gram distance
algorithm parameters:	1 (the size of q)
algorithm threshold:	0.75 (lower bound for match – upper bound is automatically set to 1.0)
final dataset:	dataset_A_2500_org_2500a_dup_qgd

Dataset 2 was formed by the following process: ./mutator.py dataset_A_2500_original.csv rec_id dataset_A_from_2500_org_2500_dup_25_10_2_11 25:2:10 25:3:10 25:4:10 25:5:10 25:6:10 25:7:10 25:8:10 25:9:10 25:10:10 25:11:10

level	comp_specificity	false_negatives	false_positives	sensitivity	specificity	total_matches	total_non_matches	true_negatives	true_positives
0	1	0	6247500	1	0	2500	6247500	0	2500
0.083333	0.323640	0	2021938	1	0.676360	2500	6247500	4225562	2500
0.166667	0.046454	0	290222	1	0.953546	2500	6247500	5957278	2500
0.250000	0.003531	0	22057	1	0.996469	2500	6247500	6225443	2500
0.333333	0.000141	0	881	1	0.999859	2500	6247500	6246619	2500
0.416667	0.000003	3	16	0.998800	0.999997	2500	6247500	6247484	2497
0.500000	0	25	0	0.990000	1	2500	6247500	6247500	2475
0.583333	0	115	0	0.954000	1	2500	6247500	6247500	2385
0.666667	0	337	0	0.865200	1	2500	6247500	6247500	2163
0.750000	0	766	0	0.693600	1	2500	6247500	6247500	1734
0.833333	0	1392	0	0.443200	1	2500	6247500	6247500	1108
0.916667	0	2064	0	0.174400	1	2500	6247500	6247500	436
1	0	2408	0	0.036800	1	2500	6247500	6247500	92

Runtime	Comparisons	Comparisons per Second
886 seconds	6250000	7054.1761

Table 11.25: Stats for dataset_A_2500_org_2500a_dup_qgd.

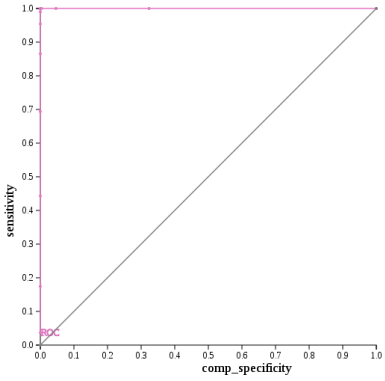


Figure 11.25: ROC curve for dataset_A_2500_org_2500a_dup_qgd.

Example 11.26. Step one in breaking the dataset with a phonetic-based similarity metric

dataset 1: dataset_A_2500_original

dataset 2: dataset_A_from_2500_org_2500_dup_25_10_2_11

method: NYSIIS

final dataset: dataset_A_2500_org_2500a_dup_nysiis

Dataset 2 was formed by the following process: ./mutator.py dataset_A_2500_original.csv rec_id dataset_A_from_2500_org_2500_dup_25_10_2_11 25:2:10 25:3:10 25:4:10 25:5:10 25:6:10 25:7:10 25:8:10 25:9:10 25:10:10 25:11:10

level	comp_specificity	false_negatives	false_positives	sensitivity	specificity	total_matches	total_non_matches	true_negatives	true_positives
0	1	0	6247500	1	0	2500	6247500	0	2500
0.200000	0.145619	15	909753	0.994000	0.854381	2500	6247500	5337747	2485
0.400000	0.001443	145	9016	0.942000	0.998557	2500	6247500	6238484	2355
0.600000	0.000004	581	26	0.767600	0.999996	2500	6247500	6247474	1919
0.800000	0	1387	0	0.445200	1	2500	6247500	6247500	1113
1	0	2153	0	0.138800	1	2500	6247500	6247500	347

Runtime	Comparisons	Comparisons per Second
283 seconds	6250000	22084.8057

Table 11.26: Stats for dataset_A_2500_org_2500a_dup_nysiis.

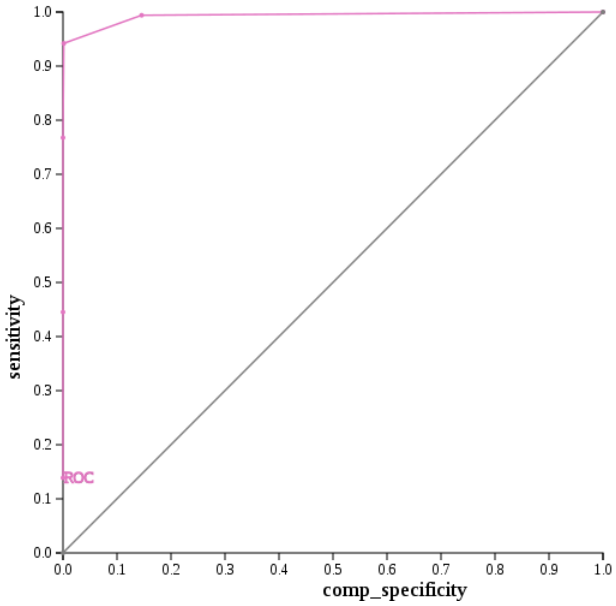


Figure 11.26: ROC curve for dataset_A_2500_org_2500a_dup_nysiis.

Example 11.27. Step two in breaking the dataset with a character-based similarity metric

dataset 1:	dataset_A_2500_original
dataset 2:	dataset_A_from_2500_org_2500_dup_25_10_12_21
method:	q-gram distance
algorithm parameters:	1 (the size of q)
algorithm threshold:	0.75 (lower bound for match – upper bound is automatically set to 1.0)
final dataset:	dataset_A_2500_org_2500b_dup_qgd

Dataset 2 was formed by the following process: ./mutator.py dataset_A_2500_original.csv rec_id dataset_A_from_2500_org_2500_dup_25_10_12_21 25:12:10 25:13:10 25:14:10 25:15:10 25:16:10 25:17:10 25:18:10 25:19:10 25:20:10 25:21:10

level	comp_specificity	false_negatives	false_positives	sensitivity	specificity	total_matches	total_non_matches	true_negatives	true_positives
0	1	0	6247500	1	0	2500	6247500	0	2500
0.083333	0.210896	0	1317570	1	0.789104	2500	6247500	4929930	2500
0.166667	0.018759	2	117199	0.999200	0.981241	2500	6247500	6130301	2498
0.250000	0.000881	9	5506	0.996400	0.999119	2500	6247500	6241994	2491
0.333333	0.000021	48	133	0.980800	0.999979	2500	6247500	6247367	2452
0.416667	0.000000	172	1	0.931200	1.000000	2500	6247500	6247499	2328
0.500000	0	533	0	0.786800	1	2500	6247500	6247500	1967
0.583333	0	1042	0	0.583200	1	2500	6247500	6247500	1458
0.666667	0	1640	0	0.344000	1	2500	6247500	6247500	860
0.750000	0	2106	0	0.157600	1	2500	6247500	6247500	394
0.833333	0	2379	0	0.048400	1	2500	6247500	6247500	121
0.916667	0	2481	0	0.007600	1	2500	6247500	6247500	19
1	0	2499	0	0.000400	1	2500	6247500	6247500	1

Runtime	Comparisons	Comparisons per Second
875 seconds	6250000	7142.8571

Table 11.27: Stats for dataset_A_2500_org_2500b_dup_qgd.

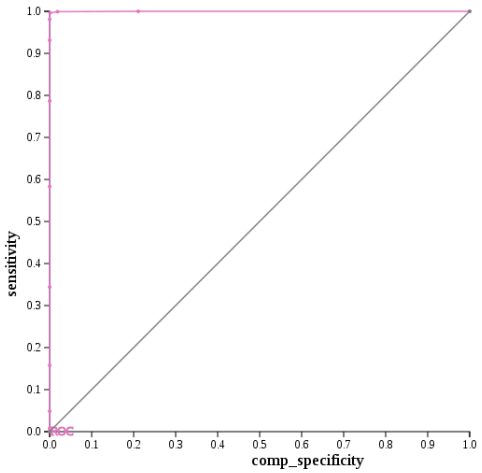


Figure 11.27: ROC curve for dataset_A_2500_org_2500b_dup_qgd.

Example 11.28. Step two in breaking the dataset with a phonetic-based similarity metric

dataset 1: dataset_A_2500_original

dataset 2: dataset_A_from_2500_org_2500_dup_25_10_12_21

method: NYSIIS

final dataset: dataset_A_2500_org_2500b_dup_nysiis

Dataset 2 was formed by the following process: ./mutator.py dataset_A_2500_original.csv rec_id dataset_A_from_2500_org_2500_dup_25_10_12_21 25:12:10 25:13:10 25:14:10 25:15:10 25:16:10 25:17:10 25:18:10 25:19:10 25:20:10 25:21:10

level	comp_specificity	false_negatives	false_positives	sensitivity	specificity	total_matches	total_non_matches	true_negatives	true_positives
0	1	0	6247500	1	0	2500	6247500	0	2500
0.200000	0.083919	334	524284	0.866400	0.916081	2500	6247500	5723216	2166
0.400000	0.000948	1204	5921	0.518400	0.999052	2500	6247500	6241579	1296
0.600000	0.000004	2024	25	0.190400	0.999996	2500	6247500	6247475	476
0.800000	0	2406	0	0.037600	1	2500	6247500	6247500	94
1	0	2497	0	0.001200	1	2500	6247500	6247500	3

Runtime	Comparisons	Comparisons per Second
281 seconds	6250000	22241.9929

Table 11.28: Stats for dataset_A_2500_org_2500b_dup_nysiis.

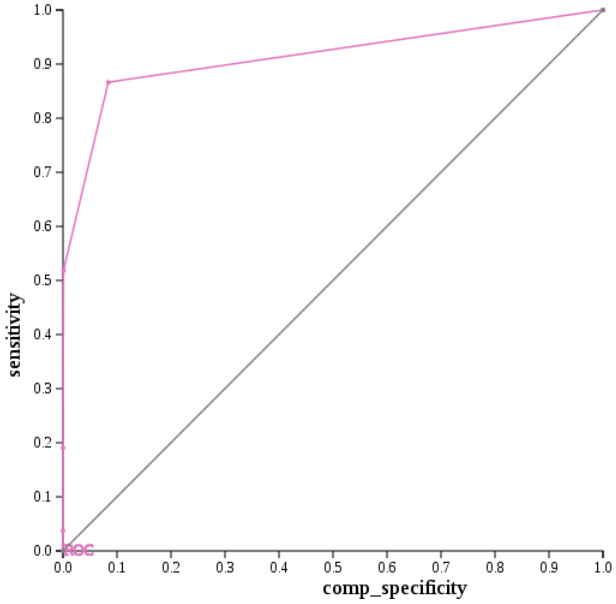


Figure 11.28: ROC curve for dataset_A_2500_org_2500b_dup_nysiis.

Example 11.29. Step three in breaking the dataset with a character-based similarity metric

dataset 1:	dataset_A_2500_original
dataset 2:	dataset_A_from_2500_org_2500_dup_25_10_22_31
method:	q-gram distance
algorithm parameters:	1 (the size of q)
algorithm threshold:	0.75 (lower bound for match – upper bound is automatically set to 1.0)
final dataset:	dataset_A_2500_org_2500c_dup_qgd

Dataset 2 was formed by the following process: ./mutator.py dataset_A_2500_original.csv rec_id dataset_A_from_2500_org_2500_dup_25_10_22_31 25:22:10 25:23:10 25:24:10 25:25:10 25:26:10 25:27:10 25:28:10 25:29:10 25:30:10 25:31:10

level	comp_specificity	false_negatives	false_positives	sensitivity	specificity	total_matches	total_non_matches	true_negatives	true_positives
0	1	0	6247500	1	0	2500	6247500	0	2500
0.083333	0.140478	2	877634	0.999200	0.859522	2500	6247500	5369866	2498
0.166667	0.008145	31	50885	0.987600	0.991855	2500	6247500	6196615	2469
0.250000	0.000245	141	1528	0.943600	0.999755	2500	6247500	6245972	2359
0.333333	0.000004	430	22	0.828000	0.999996	2500	6247500	6247478	2070
0.416667	0	943	0	0.622800	1	2500	6247500	6247500	1557
0.500000	0	1538	0	0.384800	1	2500	6247500	6247500	962
0.583333	0	2062	0	0.175200	1	2500	6247500	6247500	438
0.666667	0	2357	0	0.057200	1	2500	6247500	6247500	143
0.750000	0	2459	0	0.016400	1	2500	6247500	6247500	41
0.833333	0	2494	0	0.002400	1	2500	6247500	6247500	6
0.916667	0	2499	0	0.000400	1	2500	6247500	6247500	1

Runtime	Comparisons	Comparisons per Second
868 seconds	6250000	7200.4608

Table 11.29: Stats for dataset_A_2500_org_2500c_dup_qgd.

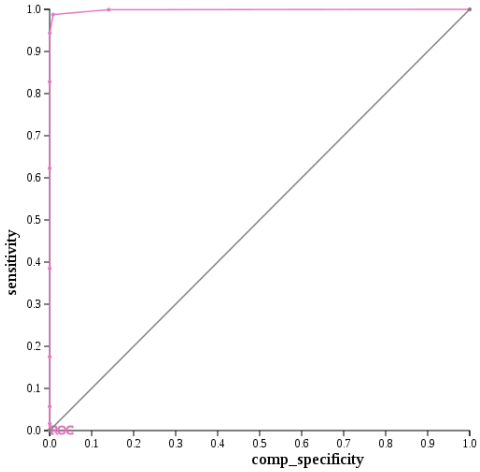


Figure 11.29: ROC curve for dataset_A_2500_org_2500c_dup_qgd.

Example 11.30. Step three in breaking the dataset with a phonetic-based similarity metric

dataset 1: dataset_A_2500_original

dataset 2: dataset_A_from_2500_org_2500_dup_25_10_22_31

method: NYSIIS

final dataset: dataset_A_2500_org_2500c_dup_nysiis

Dataset 2 was formed by the following process: ./mutator.py dataset_A_2500_original.csv rec_id dataset_A_from_2500_org_2500_dup_25_10_22_31 25:22:10 25:23:10 25:24:10 25:25:10 25:26:10 25:27:10 25:28:10 25:29:10 25:30:10 25:31:10

level	comp_specificity	false_negatives	false_positives	sensitivity	specificity	total_matches	total_non_matches	true_negatives	true_positives
0	1	0	6247500	1	0	2500	6247500	0	2500
0.200000	0.066194	897	413547	0.641200	0.933806	2500	6247500	5833953	1603
0.400000	0.000825	1990	5154	0.204000	0.999175	2500	6247500	6242346	510
0.600000	0.000004	2410	22	0.036000	0.999996	2500	6247500	6247478	90
0.800000	0	2497	0	0.001200	1	2500	6247500	6247500	3

Runtime	Comparisons	Comparisons per Second
281 seconds	6250000	22241.9929

Table 11.30: Stats for dataset_A_2500_org_2500c_dup_nysiis.

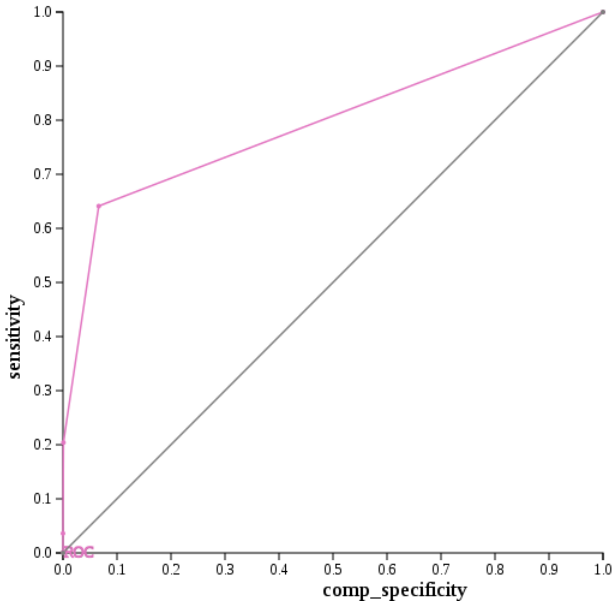


Figure 11.30: ROC curve for dataset_A_2500_org_2500c_dup_nysiis.

Example 11.31. Step four in breaking the dataset with a character-based similarity metric

dataset 1:	dataset_A_2500_original
dataset 2:	dataset_A_from_2500_org_2500_dup_25_10_32_41
method:	q-gram distance
algorithm parameters:	1 (the size of q)
algorithm threshold:	0.75 (lower bound for match – upper bound is automatically set to 1.0)
final dataset:	dataset_A_2500_org_2500d_dup_qgd

Dataset 2 was formed by the following process: ./mutator.py dataset_A_2500_original.csv rec_id dataset_A_from_2500_org_2500_dup_25_10_32_41 25:32:10 25:33:10 25:34:10 25:35:10 25:36:10 25:37:10 25:38:10 25:39:10 25:40:10 25:41:10

level	comp_specificity	false_negatives	false_positives	sensitivity	specificity	total_matches	total_non_matches	true_negatives	true_positives
0	1	0	6247500	1	0	2500	6247500	0	2500
0.083333	0.082036	28	512517	0.988800	0.917964	2500	6247500	5734983	2472
0.166667	0.002761	183	17250	0.926800	0.997239	2500	6247500	6230250	2317
0.250000	0.000042	624	260	0.750400	0.999958	2500	6247500	6247240	1876
0.333333	0.000000	1239	2	0.504400	1.000000	2500	6247500	6247498	1261
0.416667	0	1837	0	0.265200	1	2500	6247500	6247500	663
0.500000	0	2235	0	0.106000	1	2500	6247500	6247500	265
0.583333	0	2429	0	0.028400	1	2500	6247500	6247500	71
0.666667	0	2482	0	0.007200	1	2500	6247500	6247500	18
0.750000	0	2498	0	0.000800	1	2500	6247500	6247500	2

Runtime	Comparisons	Comparisons per Second
855 seconds	6250000	7309.9415

Table 11.31: Stats for dataset_A_2500_org_2500d_dup_qgd.

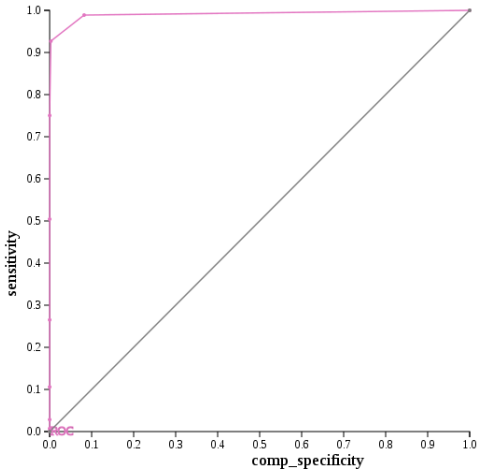


Figure 11.31: ROC curve for dataset_A_2500_org_2500d_dup_qgd.

Example 11.32. Step four in breaking the dataset with a phonetic-based similarity metric

dataset 1: dataset_A_2500_original

dataset 2: dataset_A_from_2500_org_2500_dup_25_10_32_41

method: NYSIIS

final dataset: dataset_A_2500_org_2500d_dup_nysiis

Dataset 2 was formed by the following process: ./mutator.py dataset_A_2500_original.csv rec_id dataset_A_from_2500_org_2500_dup_25_10_32_41 25:32:10 25:33:10 25:34:10 25:35:10 25:36:10 25:37:10 25:38:10 25:39:10 25:40:10 25:41:10

level	comp_specificity	false_negatives	false_positives	sensitivity	specificity	total_matches	total_non_matches	true_negatives	true_positives
0	1	0	6247500	1	0	2500	6247500	0	2500
0.200000	0.058276	1490	364079	0.404000	0.941724	2500	6247500	5883421	1010
0.400000	0.000749	2312	4678	0.075200	0.999251	2500	6247500	6242822	188
0.600000	0.000007	2476	43	0.009600	0.999993	2500	6247500	6247457	24
0.800000	0	2499	0	0.000400	1	2500	6247500	6247500	1

Runtime	Comparisons	Comparisons per Second
279 seconds	6250000	22401.4337

Table 11.32: Stats for dataset_A_2500_org_2500d_dup_nysiis.

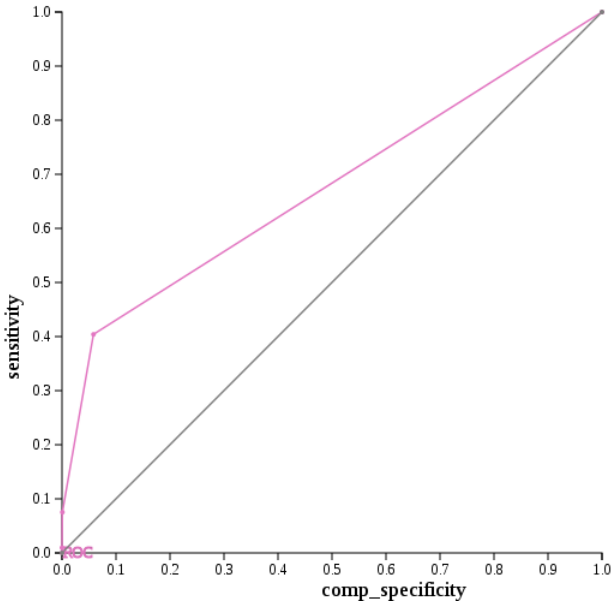


Figure 11.32: ROC curve for dataset_A_2500_org_2500d_dup_nysiis.

The ROC curves for q-gram distance stayed consistent as the number of mutations increased and the number of duplicates stayed the same. For NYSIIS, however, the ROC curves moved closer and closer to the line of no-discrimination as the number of mutations increased and the number of duplicates stayed the same. The results for this series were surprisingly similar to those found in the first series of examples where there was only one duplicate and the number of mutations increased. Furthermore, the number of duplicates had less of an effect on the ROC curve than the number of mutations did.

Chapter 12. Economic Model

When datasets are being linked for a company there is a payoff for being right (correctly linking two records that should be linked) and a penalty for being wrong (linking two records together that should not be linked). Lets say that for every record that is correctly linked the company receives a pay off of b dollars and for every record that they incorrectly link they lose a dollars. An example of this would be that when a company links two records if they were a correct match then they still have that individual's business and when they link incorrect individuals they lose that person's future business with them. Using this information we can then build an economic model that will allow us to find what tolerance level we need to merge two datasets together in order to maximize our profit.

Being right corresponds to the number of true positives and being wrong corresponds to the number of false positives. Recall that sensitivity = true positives/(true positives + false negatives) and specificity = true negatives/(true negatives + false positives). Now the y-axis on the ROC curve corresponds to sensitivity and the x-axis corresponds to 1-specificity or what we called comp_specificity where comp_specificity = false positives/(false positives + true negatives). Applying this information we then are able to say that not only does being right correspond to the number of true positives, but it also relates to the y-axis and sensitivity, while being wrong corresponds to the number of false positives and relates to the x-axis and comp_specificity. Then b is how much weight we put on being right and a is how much we put on being wrong.

Using this information we are then able to create a profit function that is dependent on how many records were linked correctly, how many were linked incorrectly, the cost given to being right, and the penalty given for being wrong. Thus the profit function is linear and is equal to $a * (1 - comp\_specificity) + b * sensitivity$.

The bounded region that the profit function is supposed to be maximized over is created by the line of no-discrimination (the diagonal line going from the point (0,0) to (1,1)) and the ROC curve. The points (0,0), (1,1), and those of the form (comp_specificity, sensitivity)

are the vertices of this bounded region. It is important to note that this bounded region is concave, thus we can maximize a function over it. The point that gives the profit function the greatest value with the given cost of being right and penalty for being wrong, maximizes the profit function.

Since $profit = a * (1 - comp\_specificity) + b * sensitivity$, where $comp\_specificity$ represents the x-axis and $sensitivity$ represents the y-axis, then the slope of the profit equation is $\frac{a}{b}$. Using this information we then can form the equation that maximizes profit using point slope form. Thus maximum profit is obtained when

$$y = m * (x - comp\_specificity[max\_profit\_index]) + sensitivity[max\_profit\_index].$$

With the above method we are looking at true positives and false positives indirectly in terms of sensitivity and specificity. This then leads to the question what if we perform a linear transformation and look at the number of true positives and false positives directly? The complete number of negatives is equal to the number of false positives plus the number of true negatives. Whereas the complete number of positives is equal to the number of true positives plus the number of false negatives.

Recall that sensitivity = true positives/(true positives + false negatives) and corresponds to the y-axis and 1-specificity = comp_specificity = false positives/(false positives + true negatives) and corresponds to the x-axis. Then this would mean that the number of false positives are represented on the x-axis and the number of true positives are represented on the y-axis. To convert to the false positive cost simply multiply $-a$ by the complete number of positives and to convert to the true positives cost multiply b by the complete number of negatives.

The profit function is then equal to the false positive cost times the false positives plus the true positive cost times the true positives. After all of the possible profits are calculated, the index of the maximum profit is found. This then tells us what false positive and true positive count gave us the greatest profit. From there we can then find the slope of the line

that corresponds to the maximum profit equation. Since true positives are represented on the x-axis and false positives are represented on the y-axis, then the slope of the line is -false positive cost/true positives cost. Using point-slope form we can then form an equation to the line that will give us the maximum profit. So

$$y_2 = m_2 * (x_2 - false\_positives[max\_profit\_index\_2]) + true\_positives[max\_profit\_index\_2].$$

Below we will use the datasets from breaking a dataset and merging it with its original to find what equation maximizes the profit function and at what tolerance level should the datasets be merged at. Note that for all of the examples $a = 7$, $b = 2$, and both methods are used. The python code profit.py used to generate these results can be found in the appendix.

Example 12.1.

dataset 1:	dataset_A_500_original
dataset 2:	dataset_A_from_500_org_100_dup_1021_1031_1041_1051_1061_1071 _1081_1091_10101_10111
method:	edit distance
algorithm threshold:	2.0
final dataset:	dataset_A_500_org_100a_dup_ed

Data set 2 was formed by the following process: ./mutator.py dataset_A_500_original.csv rec_id dataset_A_from_500_org_100_dup_1021_1031_1041_1051_1061_1071_1081_1091_10101_10111 10:2:1 10:3:1 10:4:1 10:5:1 10:6:1 10:7:1 10:8:1 10:9:1 10:10:1 10:11:1

level	comp_specificity	false_negatives	false_positives	sensitivity	specificity	total_matches	total_non_matches	true_negatives	true_positives
0	1	0	49900	1	0	100	49900	0	100
0.083333	0.986333	0	49218	1	0.013667	100	49900	682	100
0.166667	0.883006	0	44062	1	0.116994	100	49900	5838	100
0.250000	0.623607	0	31118	1	0.376393	100	49900	18782	100
0.333333	0.315591	0	15748	1	0.684409	100	49900	34152	100
0.416667	0.112004	0	5589	1	0.887996	100	49900	44311	100
0.500000	0.026814	0	1338	1	0.973186	100	49900	48562	100
0.583333	0.003287	0	164	1	0.996713	100	49900	49736	100
0.666667	0.000381	1	19	0.990000	0.999619	100	49900	49881	99
0.750000	0	7	0	0.930000	1	100	49900	49900	93
0.833333	0	17	0	0.830000	1	100	49900	49900	83
0.916667	0	44	0	0.560000	1	100	49900	49900	56
1	0	76	0	0.240000	1	100	49900	49900	24

Runtime	Comparisons	Comparisons per Second
18 seconds	50000	2777.7778

Table 12.1: Stats for dataset_A_500_org_100a_dup_ed.

After running profit.py on the conditions given we learn that the equation that maximizes profit is: $y = 3.5(x - 0.000381) + 0.99$ and the tolerance level should be set at 0.6666666667 in order to maximize profit using the first method. Using the second method where false positives and true positives have weights directly applied to them the equation that maximizes profit is: $y = 0.007014(x - 19) + 99$ and the tolerance level should be set at 0.6666666667 in order to maximize profit.

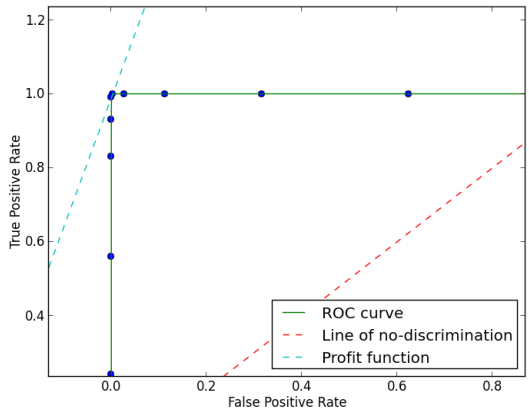


Figure 12.1: ROC curve with profit function for dataset_A_500_org_100a_dup_ed using method one.

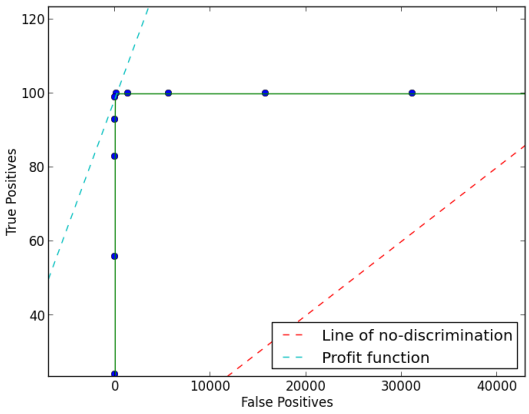


Figure 12.2: The profit function for dataset_A_500_org_100a_dup_ed using method two.

Example 12.2.

dataset 1:	dataset_A_500_original
dataset 2:	dataset_A_from_500_org_100_dup_10121_10131_10141_10151_10161 _10171_10181_10191_10201_10211
method:	edit distance
algorithm threshold:	2.0
final dataset:	dataset_A_500_org_100b_dup_ed

Data set 2 was formed by the following process: ./mutator.py dataset_A_500_original.csv rec_id dataset_A_from_500_org_100_dup_10121_10131_10141_10151_10161_10171_10181_10191_10201_10211 10:12:1 10:13:1 10:14:1 10:15:1 10:16:1 10:17:1 10:18:1 10:19:1 10:20:1 10:21:1

level	comp_specificity	false_negatives	false_positives	sensitivity	specificity	total_matches	total_non_matches	true_negatives	true_positives
0	1	0	49900	1	0	100	49900	0	100
0.083333	0.991242	0	49463	1	0.008758	100	49900	437	100
0.166667	0.904749	0	45147	1	0.095251	100	49900	4753	100
0.250000	0.701683	0	35014	1	0.298317	100	49900	14886	100
0.333333	0.406713	0	20295	1	0.593287	100	49900	29605	100
0.416667	0.161703	1	8069	0.990000	0.838297	100	49900	41831	99
0.500000	0.044890	5	2240	0.950000	0.955110	100	49900	47660	95
0.583333	0.009820	13	490	0.870000	0.990180	100	49900	49410	87
0.666667	0.001222	32	61	0.680000	0.998778	100	49900	49839	68
0.750000	0.000100	62	5	0.380000	0.999900	100	49900	49895	38
0.833333	0.000020	86	1	0.140000	0.999980	100	49900	49899	14
0.916667	0	97	0	0.030000	1	100	49900	49900	3
1	0	99	0	0.010000	1	100	49900	49900	1

Runtime	Comparisons	Comparisons per Second
16 seconds	50000	3125.0000

Table 12.2: Stats for dataset_A_500_org_100b_dup_ed.

After running profit.py on the conditions given we learn that the equation that maximizes profit is: $y = 3.5(x - 0.00982) + 0.87$ and the tolerance level should be set at 0.5833333333 in order to maximize profit using method one. Using method two where false positives and true positives have weights directly applied to them the equation that maximizes profit is: $y = 0.007014(x - 490) + 87$ and the tolerance level should be set at 0.5833333333 in order to maximize profit.

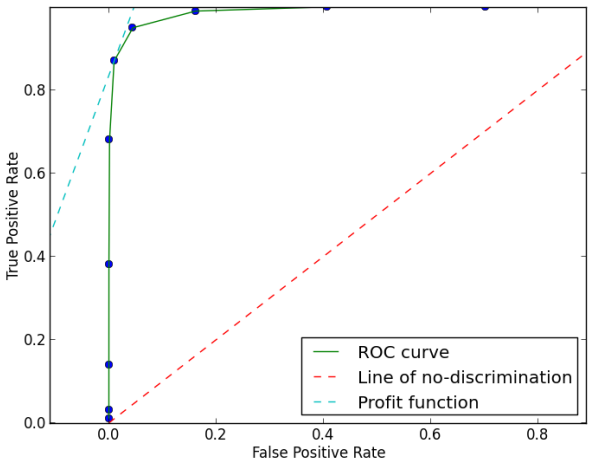


Figure 12.3: ROC curve with profit function for dataset_A_500_org_100b_dup_ed using method one.

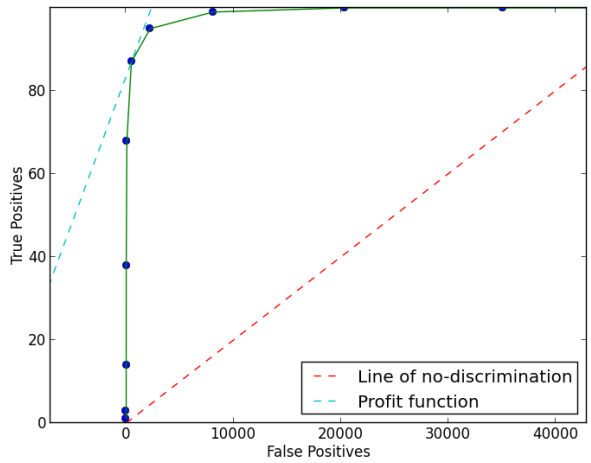


Figure 12.4: The profit function for dataset_A_500_org_100b_dup_ed using method two.

Example 12.3.

dataset 1:	dataset_A_500_original
dataset 2:	dataset_A_from_500_org_100_dup_10221_10231_10241_10251_10261 _10271_10281_10291_10301_10311
method:	edit distance
algorithm threshold:	2.0
final dataset:	dataset_A_500_org_100c_dup_ed

Data set 2 was formed by the following process: ./mutator.py dataset_A_500_original.csv rec_id dataset_A_from_500_org_100_dup_10221_10231_10241_10251_10261_10271_10281_10291_10301_10311 10:22:1 10:23:1 10:24:1 10:25:1 10:26:1 10:27:1 10:28:1 10:29:1 10:30:1 10:31:1

level	comp_specificity	false_negatives	false_positives	sensitivity	specificity	total_matches	total_non_matches	true_negatives	true_positives
0	1	0	49900	1	0	100	49900	0	100
0.083333	0.986293	0	49216	1	0.013707	100	49900	684	100
0.166667	0.893267	0	44574	1	0.106733	100	49900	5326	100
0.250000	0.678838	1	33874	0.990000	0.321162	100	49900	16026	99
0.333333	0.405912	4	20255	0.960000	0.594088	100	49900	29645	96
0.416667	0.193948	10	9678	0.900000	0.806052	100	49900	40222	90
0.500000	0.074128	31	3699	0.690000	0.925872	100	49900	46201	69
0.583333	0.017555	56	876	0.440000	0.982445	100	49900	49024	44
0.666667	0.001603	80	80	0.200000	0.998397	100	49900	49820	20
0.750000	0.000180	94	9	0.060000	0.999820	100	49900	49891	6

Runtime	Comparisons	Comparisons per Second
15 seconds	50000	3333.3333

Table 12.3: Stats for dataset_A_500_org_100c_dup_ed.

After running profit.py on the conditions given we learn that the equation that maximizes profit is: $y = 3.5(x - 0.074128) + 0.69$ and the tolerance level should be set at 0.5 in order to maximize profit using the first method. Using the second method where false positives and true positives have weights directly applied to them the equation that maximizes profit is: $y = 0.007014(x - 19) + 99$ and the tolerance level should be set at 0.6666666667 in order to maximize profit.

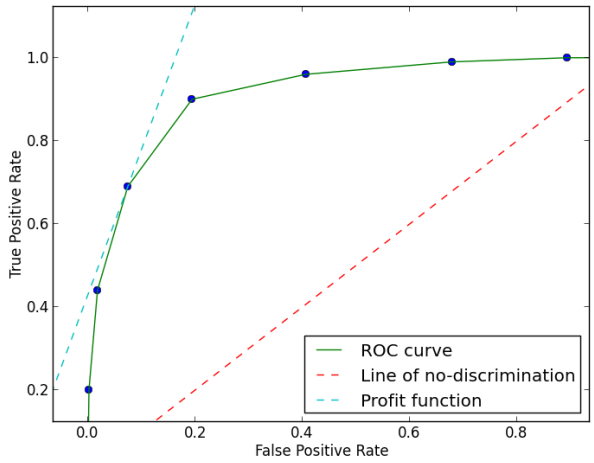


Figure 12.5: ROC curve with profit function for dataset_A_500_org_100c_dup_ed using method one.

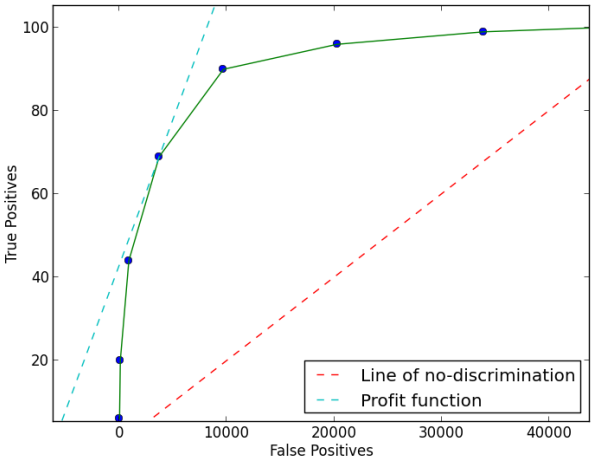


Figure 12.6: The profit function for dataset_A_500_org_100c_dup_ed using method two.

Example 12.4.

dataset 1:	dataset_A_500_original
dataset 2:	dataset_A_from_500_org_100_dup_10321_10331_10341_10351_10361 _10371_10381_10391_10401_10411
method:	edit distance
algorithm threshold:	2.0
final dataset:	dataset_A_500_org_100d_dup_ed

Data set 2 was formed by the following process: ./mutator.py dataset_A_500_original.csv rec_id dataset_A_from_500_org_100_dup_10321_10331_10341_10351_10361_10371_10381_10391_10401_10411 10:32:1 10:33:1 10:34:1 10:35:1 10:36:1 10:37:1 10:38:1 10:39:1 10:40:1 10:41:1

level	comp_specificity	false_negatives	false_positives	sensitivity	specificity	total_matches	total_non_matches	true_negatives	true_positives
0	1	0	49900	1	0	100	49900	0	100
0.083333	0.993287	0	49565	1	0.006713	100	49900	335	100
0.166667	0.940501	0	46931	1	0.059499	100	49900	2969	100
0.250000	0.791463	5	39494	0.950000	0.208537	100	49900	10406	95
0.333333	0.540381	16	26965	0.840000	0.459619	100	49900	22935	84
0.416667	0.299980	36	14969	0.640000	0.700020	100	49900	34931	64
0.500000	0.133487	58	6661	0.420000	0.866513	100	49900	43239	42
0.583333	0.051483	77	2569	0.230000	0.948517	100	49900	47331	23
0.666667	0.013687	93	683	0.070000	0.986313	100	49900	49217	7
0.750000	0.001844	98	92	0.020000	0.998156	100	49900	49808	2
0.833333	0.000140	99	7	0.010000	0.999860	100	49900	49893	1

Runtime	Comparisons	Comparisons per Second
15 seconds	50000	3333.3333

Table 12.4: Stats for dataset_A_500_org_100d_dup_ed.

After running profit.py on the conditions given we learn that the equation that maximizes profit is: $y = 3.5(x - 0.051483) + 0.23$ and that the tolerance level should be set at 0.5833333333 in order to maximize profit using method one. Using method two where false positives and true positives have weights directly applied to them the equation that maximizes profit is: $y = 0.007014(x - 2569) + 23$ and the tolerance level should be set at 0.6666666667 in order to maximize profit.

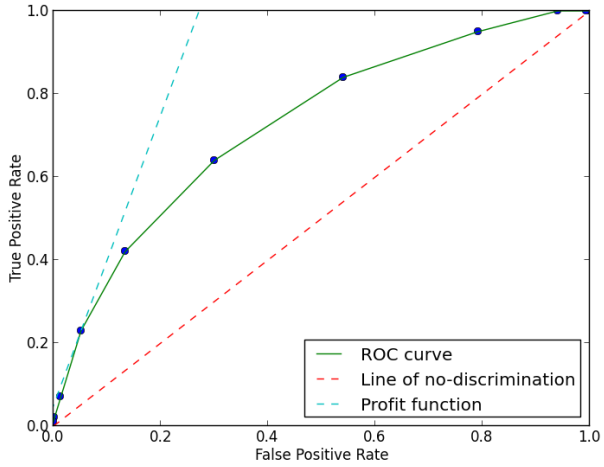


Figure 12.7: ROC curve with profit function for dataset_A_500_org_100d_dup_ed using method one.

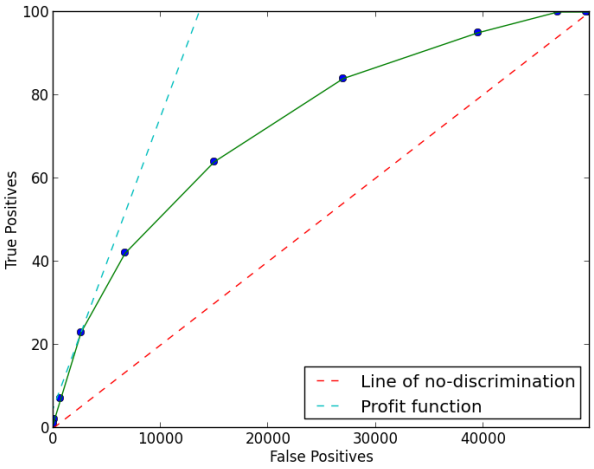


Figure 12.8: The profit function for dataset_A_500_org_100d_dup_ed using method two.

It is interesting to note that the curves for the two methods are the same, except for the scaling. The scaling is due to a linear transformation that maps the data points between the two graphs. Thus whether we look directly at the false positives and true positives, or indirectly in terms of comp_specificity and sensitivity, we get the same results.

Appendix A. Edit Distance Python Code

```
# EditDistance.py

def ed(s, t):
    n = len(s)
    m = len(t)

    if n == 0 or m == 0:
        return None

    if n < m:
        return ed(t, s)

    d = [[0 for j in range(m)] for i in range(n)]

    for i in range(n):
        for j in range(m):
            if s[i] == t[j]:
                cost = 0
            else:
                cost = 1

            vals = []
            if i > 0:
                vals.append(d[i-1][j] + 1) # insert
            if j > 0:
                vals.append(d[i][j-1] + 1) # delete
            if i > 0 and j > 0:
                vals.append(d[i-1][j-1] + cost) # substitute or copy
            elif i > 0:
                # cost of skipping the first i values in the string s
                vals.append(cost + i)
```

```
            elif j > 0:
                # cost of skipping the first j values in the string t
                vals.append(cost + j)
            else:
                vals.append(cost) # special case when i = 0 and j = 0

            d[i][j] = min(vals)

    return d[n-1][m-1]

if __name__ == '__main__':
    #s = 'DAVID'
    #t = 'SDDAVD'

     s = 'STASHA'
     t = 'DSSTSGA'

     print ed(s, t)
```

Appendix B. Smith-Waterman Distance Python Code

```
# SmithWatermanDistance.py

def sw(s, t, G):
    n = len(s)
    m = len(t)

    if n == 0 or m == 0:
        return None
    elif n < m:
        return sw(t, s, G)

```

```
    prev = [2 if s[0] == t[0] else 0]
    prev_append = prev.append
    for j in xrange(1,m):
        prev_append(max(-((-2 if s[0] == t[j] else 1) + j), prev[-1] - G,
            0))

    for i in xrange(1,n):
        curr_s = s[i]
        curr = [max(-((-2 if curr_s == t[0] else 1) + i), prev[0] - G, 0)]
        curr_append = curr.append
        for j in xrange(1, m):
            cost = -2 if curr_s == t[j] else 1
            curr_append(max(0, prev[j] - G, curr[-1] - G, prev[j-1] - cost
                ))

        prev = curr

    return prev[-1]

if __name__ == '__main__':
    s = 'david'
    t = 'sddavd'
    G = 1

    #s = 'stasha'
    #t = 'wtaahz'
    #G = 1

    print sw(s, t, G)
```

Appendix C. Jaro Distance Python Code

```
# JaroDistance.py

def jd(s, t):
    if len(s) == 0 or len(t) == 0:
        return 0

    s_flag = [False for i in range(len(s))]
    t_flag = [False for j in range(len(t))]

    search_range = max(0, (min(len(s), len(t)) / 2) - 1)

    common_chars = 0
    for i in range(len(s)):
        for j in range(max(i - search_range, 0), min(i + search_range + 1,
            len(t))):
            if (not t_flag[j]) and t[j] == s[i]:
                t_flag[j] = True
                s_flag[i] = True
                common_chars += 1
                break

    if common_chars == 0:
        return 0

    k = trans_count = 0
    for i in range(len(s)):
        if s_flag[i]:
            for j in range(k, len(t)):
                if t_flag[j]:
                    k = j + 1
                    break
```

```
            if s[i] != t[j]:
                trans_count += 1
    trans_count /= 2

    return ((common_chars / float(len(s))) + (common_chars / float(len(t))
        ) + (float(common_chars - trans_count) / common_chars)) / 3

if __name__ == '__main__':
    #s = 'Stasha'
    #t = 'Satsha'

    s = 'Stasha Bown'
    t = 'SStasja Brpwm'

    print jd(s, t)
```

Appendix D. Q-gram Python Code

```
#Qgram.py

def qg(s, q):
     vals = []
     n = len(s)

     for i in range(n):
          if n-i+1 > q:
               vals.append(s[i:i+q])
     return vals

if __name__ == '__main__':
     s = 'stasha'
     q = 3
```

```
15
16      print qg(s, q)
```

Appendix E. Q-gram Distance Python Code

```
1  #QgramDistance.py
2
3  def qgd(s, t, q):
4       vals1 = []
5       vals2 = []
6       n = len(s)
7       m = len(t)
8
9       for i in range(n):
10           if n-i+1 > q:
11                vals1.append(s[i:i+q])
12
13      for j in range(m):
14           if m-j+1 > q:
15                vals2.append(t[j:j+q])
16
17      intersect = set(vals1).intersection(vals2) # intersection of vals1
           and vals2
18      uniq = list(set(vals1 + vals2)) # unique elements in vals1 and vals2
19      qgramdist = 1.0*len(intersect)/len(uniq)*100
20      return "%.2f" %qgramdist
21
22 if __name__ == '__main__':
23      s = 'stasha'
24      t = 'sstasga'
25      q = 4
26
```

```
27      print qgd(s, t, q)
```

Appendix F. Soundex Python Code

```
1  # Soundex.py
2
3  def soundex(name, len=4):
4      """ soundex module conforming to Knuth's algorithm
5          implementation 2000-12-24 by Gregory Jorgensen
6          public domain
7      """
8
9      # digits holds the soundex values for the alphabet
10     digits = '01230120022455012623010202'
11     sndx = ''
12     fc = ''
13
14     # translate alpha characters in name to soundex digits
15     for c in name.upper():
16         if c.isalpha():
17             if not fc: fc = c   # remember first letter
18             d = digits[ord(c)-ord('A')]
19             # duplicate consecutive soundex digits are skipped
20             if not sndx or (d != sndx[-1]):
21                 sndx += d
22
23     # replace first digit with first alpha character
24     sndx = fc + sndx[1:]
25
26     # remove all 0s from the soundex code
27     sndx = sndx.replace('0','')
28
```

```
    # return soundex code padded to len characters
    return (sndx + (len * '0'))[:len]

if __name__ == '__main__':
     name = 'Simoneit'
     #name = 'Avio'

     print soundex(name)
```

Appendix G. NYSIIS Python Code

```
# NYSIIS.py

#import line added for compatibility with code execution change.
#You can remove all but what is needed.
import calendar, datetime, difflib, math, random, re, string, time, urllib

"""NYSIIS phonetic code algorithm
adapted from http://metagram.webreply.com/downloads/nysiis.py
"""

_DIGITre = re.compile(r"\d")
_JRSRre = re.compile(r" [JS]R$")
_VIre = re.compile(r" [VI]+$")
_VOWELWre = re.compile(r"[AEIOU]+W")
_VOWELre = re.compile(r"[AEIOU]+")
_AHHAre = re.compile(r"AH|HA")
_DUPSre = re.compile(r"([A-Z])\1+")

_suffixMap = {
    "IX": "IC",
    "EX": "EC",
```

```
    "YE": "Y",
    "EE": "Y",
    "IE": "Y",
    "DT": "D",
    "RT": "D",
    "RD": "D",
    "NT": "N",
    "ND": "N",
}

def prepare(s):
    """Return a string with stripped whitespace, no numbers or punctuation
       , collapsed consecutive spaces, and converted to uppercase"""
    s = s.strip().upper()
    for char in s:
        if char in '1234567890':
            s = s.replace(char, ' ')
        elif char in '!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~':
            s = s.replace(char, '')
    s = " ".join(s.split())
    return s

def nysiis(name):
    """Calculates the NYSIIS phonetic code for the given name"""
    if _DIGITre.search(name):
        raise Exception("ERROR: Numerics found in NYSIIS input!")
    name = prepare(name)
    name = _JRSRre.sub("", name) #drop JR/SR suffixes
    name = _VIre.sub("", name) #drop trailing roman numerals
    name = name.rstrip("SZ") #remove trailing SZ
    if name.startswith("MAC"): #MAC -> MC
        name = "MC" + name[3:]
    if name.startswith("PF"): #PF -> F
```

```
54          name = name[1:]
55      #change spelling on some suffixes
56      if len(name) > 2:
57          suffix = name[-2:]
58          name = name[:-2] + _suffixMap.get(suffix, suffix)
59          #change 'EV' to 'EF' if not at start of name
60          if len(name) > 2 and name[2:].find("EV") > -1:
61              name = name[:2] + name[2:].replace("EV", "EF")
62          #save first vowel for later
63          firstletter = name[0]
64          #remove any 'W' that follows a vowel
65          name = _VOWELWre.sub("A", name)
66          #replace all vowels with 'A'
67          name = _VOWELre.sub("A", name)
68          #change GHT->GT, DG->G, PH->F
69          for pattern, replacement in [("GHT","GT"), ("DG","G"), ("PH","F")
                ]:
70              name = name.replace(pattern, replacement)
71          #change non-initial AH, or HA to just A
72          name = name[0] + _AHHAre.sub("A", name[1:])
73          #change KN->N, K->C
74          name = name.replace("KN", "N")
75          name = name.replace("K", "C")
76          #change non initial M->N, Q->G
77          if len(name) > 1:
78              name = name[0] + name[1:].replace("M", "N")
79              name = name[0] + name[1:].replace("Q", "G")
80          for pattern, replacement in [("SH",  "S"),
81                                       ("SCH", "S"),
82                                       ("YW",  "Y")]:
83              name = name.replace(pattern, replacement)
84          #if not first or last character, change 'Y' to 'A'
85          if len(name) > 2:
```

```
86              first = name[0]
87              mid = name[1:]
88              last = ""
89              if len(mid) > 2:
90                  last = mid[-1]
91                  mid = mid[:-1]
92              name = first + mid.replace("Y","A") + last
93          #change 'WR' to 'R'
94          name = name.replace("WR", "R")
95          #if not first character, change 'Z' to 'S'
96          name = name[0] + name.replace("Z", "S")
97          if name[-2:] == "AY":
98              name = name[:-2] + "Y"
99          #remove trailing vowels
100         while name and name[-1] == "A":
101             name = name[:-1]
102         #remove all duplicated letters
103         name = _DUPSre.sub(r"\1", name)
104         #if first char of original name was a vowel, append it to the code
105         if firstletter in "AEIOU":
106             name = firstletter + name[1:]
107
108     return name
109
110 #Conditional execution added to satisfy import.
111 #http://featurelist.org/features/details/238
112 #Further refinements made for use as a module
113 #and to allow execution from within Python 3.0
114 if __name__ == '__builtin__':
115     try:
116         print (nysiis(INPUT))
117     except:
118         import sys
```

```
119         print (sys.exc_info()[1].args[0])
120
121 #name = 'Simoneit'
122 name = 'Kristie'
123
124 print nysiis(name)
```

Appendix H. Double Metaphone Python Code

```
1  # DoubleMetaphone.py
2
3  #!python
4  #coding= utf-8
5  # This script implements the Double Metaphone algorythm (c) 1998, 1999 by
       Lawrence Philips
6  # it was translated to Python from the C source written by Kevin Atkinson
       (http://aspell.net/metaphone/)
7  # By Andrew Collins - January 12, 2007 who claims no rights to this work
8  # http://www.atomodo.com/code/double-metaphone/metaphone.py/view
9  # Tested with Pyhon 2.4.3
10 # Updated Feb 14, 2007 - Found a typo in the 'gh' section
11 # Updated Dec 17, 2007 - Bugs fixed in 'S', 'Z', and 'J' sections. Thanks
       Chris Leong!
12 # Updated June 25, 2010 - several bugs fixed thanks to Nils Johnsson for a
        spectacular
13 #       bug squashing effort. There were many cases where this function
       wouldn't give the same output
14 #       as the original C source that were fixed by his careful attention
       and excellent communication.
15 #       The script was also updated to use utf-8 rather than latin-1.
16 def dm(st) :
17         """dm(string) -> (string, string or None)
```

```
18          returns the double metaphone codes for given string - always a
               tuple
19          there are no checks done on the input string, but it should be a
               single word or name."""
20          vowels = ['A', 'E', 'I', 'O', 'U', 'Y']
21          st = st.decode('utf-8', 'ignore')
22          st = st.upper() # st is short for string. I usually prefer
               descriptive over short, but this var is used a lot!
23          is_slavo_germanic = (st.find('W') > -1 or st.find('K') > -1 or st.
               find('CZ') > -1 or st.find('WITZ') > -1)
24          length = len(st)
25          first = 2
26          st = ('-') * first + st + (' ' * 5)  # so we can index beyond the
               begining and end of the input string
27          last = first + length -1
28          pos = first # pos is short for position
29          pri = sec = '' # primary and secondary metaphone codes
30          #skip these silent letters when at start of word
31          if st[first:first+2] in ["GN", "KN", "PN", "WR", "PS"] :
32                  pos += 1
33          # Initial 'X' is pronounced 'Z' e.g. 'Xavier'
34          if st[first] == 'X' :
35                  pri = sec = 'S' #'Z' maps to 'S'
36                  pos += 1
37          # main loop through chars in st
38          while pos <= last :
39                  #print str(pos) + '\t' + st[pos]
40                  ch = st[pos] # ch is short for character
41                  # nxt (short for next characters in metaphone code) is set
                        to  a tuple of the next characters in
42                  # the primary and secondary codes and how many characters
                       to move forward in the string.
43                  # the secondary code letter is given only when it is
```

```
                different than the primary.
44              # This is just a trick to make the code easier to write
                and read.
45              nxt = (None, 1) # default action is to add nothing and
                move to next char
46              if ch in vowels :
47                      nxt = (None, 1)
48                      if pos == first : # all init vowels now map to 'A'
49                              nxt = ('A', 1)
50              elif ch == 'B' :
51                      #"-mb", e.g", "dumb", already skipped over... see
                        'M' below
52                      if st[pos+1] == 'B' :
53                              nxt = ('P', 2)
54                      else :
55                              nxt = ('P', 1)
56              elif ch == 'C' :
57                      # various germanic
58                      if (pos > (first + 1) and st[pos-2] not in vowels
                        and st[pos-1:pos+2] == 'ACH' and \
59                        (st[pos+2] not in ['I', 'E'] or st[pos-2:pos+4]
                            in ['BACHER', 'MACHER'])) :
60                              nxt = ('K', 2)
61                      # special case 'CAESAR'
62                      elif pos == first and st[first:first+6] == 'CAESAR
                        ' :
63                              nxt = ('S', 2)
64                      elif st[pos:pos+4] == 'CHIA' : #italian 'chianti'
65                              nxt = ('K', 2)
66                      elif st[pos:pos+2] == 'CH' :
67                              # find 'michael'
68                              if pos > first and st[pos:pos+4] == 'CHAE'
                                    :
```

```
69                          nxt = ('K', 'X', 2)
70                  elif pos == first and (st[pos+1:pos+6] in
                        ['HARAC', 'HARIS'] or \
71                        st[pos+1:pos+4] in ["HOR", "HYM", "HIA"
                            , "HEM"]) and st[first:first+5] != '
                            CHORE' :
72                          nxt = ('K', 2)
73                  #germanic, greek, or otherwise 'ch' for '
                        kh' sound
74                  elif st[first:first+4] in ['VAN ', 'VON ']
                         or st[first:first+3] == 'SCH' \
75                        or st[pos-2:pos+4] in ["ORCHES", "
                            ARCHIT", "ORCHID"] \
76                        or st[pos+2] in ['T', 'S'] \
77                        or ((st[pos-1] in ["A", "O", "U", "E"]
                            or pos == first) \
78                        and st[pos+2] in ["L", "R", "N", "M", "
                            B", "H", "F", "V", "W", " "]) :
79                          nxt = ('K', 1)
80                  else :
81                          if pos > first :
82                                  if st[first:first+2] == '
                                        MC' :
83                                          nxt = ('K', 2)
84                                  else :
85                                          nxt = ('X', 'K',
                                                2)
86                          else :
87                                  nxt = ('X', 2)
88          #e.g, 'czerny'
89          elif st[pos:pos+2] == 'CZ' and st[pos-2:pos+2] !=
                'WICZ' :
90                  nxt = ('S', 'X', 2)
```

```
91                      #e.g., 'focaccia'
92                      elif st[pos+1:pos+4] == 'CIA' :
93                              nxt = ('X', 3)
94                      #double 'C', but not if e.g. 'McClellan'
95                      elif st[pos:pos+2] == 'CC' and not (pos == (first
                            +1) and st[first] == 'M') :
96                              #'bellocchio' but not 'bacchus'
97                              if st[pos+2] in ["I", "E", "H"] and st[pos
                                    +2:pos+4] != 'HU' :
98                                      #'accident', 'accede' 'succeed'
99                                      if (pos == (first +1) and st[first
                                            ] == 'A') or \
100                                         st[pos-1:pos+4] in ['UCCEE', '
                                                UCCES'] :
101                                             nxt = ('KS', 3)
102                                     #'bacci', 'bertucci', other
                                            italian
103                                     else:
104                                             nxt = ('X', 3)
105                             else :
106                                     nxt = ('K', 2)
107                     elif st[pos:pos+2] in ["CK", "CG", "CQ"] :
108                             nxt = ('K', 'K', 2)
109                     elif st[pos:pos+2] in ["CI", "CE", "CY"] :
110                             #italian vs. english
111                             if st[pos:pos+3] in ["CIO", "CIE", "CIA"]
                                    :
112                                     nxt = ('S', 'X', 2)
113                             else :
114                                     nxt = ('S', 2)
115                     else :
116                             #name sent in 'mac caffrey', 'mac gregor
117                             if st[pos+1:pos+3] in [" C", " Q", " G"] :
```

```
118                                     nxt = ('K', 3)
119                             else :
120                                     if st[pos+1] in ["C", "K", "Q"]
                                            and st[pos+1:pos+3] not in ["CE
                                            ", "CI"] :
121                                             nxt = ('K', 2)
122                                     else : # default for 'C'
123                                             nxt = ('K', 1)
124             elif ch == u'  ' :
125                     nxt = ('S', 1)
126             elif ch == 'D' :
127                     if st[pos:pos+2] == 'DG' :
128                             if st[pos+2] in ['I', 'E', 'Y'] : #e.g. '
                                    edge'
129                                     nxt = ('J', 3)
130                             else :
131                                     nxt = ('TK', 2)
132                     elif st[pos:pos+2] in ['DT', 'DD'] :
133                             nxt = ('T', 2)
134                     else :
135                             nxt = ('T', 1)
136             elif ch == 'F' :
137                     if st[pos+1] == 'F' :
138                             nxt = ('F', 2)
139                     else :
140                             nxt = ('F', 1)
141             elif ch == 'G' :
142                     if st[pos+1] == 'H' :
143                             if pos > first and st[pos-1] not in vowels
                                    :
144                                     nxt = ('K', 2)
145                             elif pos < (first + 3) :
146                                     if pos == first : #'ghislane',
```

```
                                    ghiradelli
147                                     if st[pos+2] == 'I' :
148                                             nxt = ('J', 2)
149                                     else :
150                                             nxt = ('K', 2)
151                             #Parker's rule (with some further
                                refinements) - e.g., 'hugh'
152                             elif (pos > (first + 1) and st[pos-2] in [
                                'B', 'H', 'D'] ) \
153                                or (pos > (first + 2) and st[pos-3] in
                                    ['B', 'H', 'D'] ) \
154                                or (pos > (first + 3) and st[pos-4] in
                                    ['B', 'H'] ) :
155                                     nxt = (None, 2)
156                             else :
157                                     # e.g., 'laugh', 'McLaughlin', '
                                        cough', 'gough', 'rough', '
                                        tough'
158                                     if pos > (first + 2) and st[pos-1]
                                         == 'U' \
159                                        and st[pos-3] in ["C", "G", "L"
                                            , "R", "T"] :
160                                             nxt = ('F', 2)
161                                     else :
162                                             if pos > first and st[pos
                                                -1] != 'I' :
163                                                     nxt = ('K', 2)
164                     elif st[pos+1] == 'N' :
165                             if pos == (first +1) and st[first] in
                                vowels and not is_slavo_germanic :
166                                     nxt = ('KN', 'N', 2)
167                             else :
168                                     # not e.g. 'cagney'
```

```
169                                 if st[pos+2:pos+4] != 'EY' and st[
                                        pos+1] != 'Y' and not
                                        is_slavo_germanic :
170                                         nxt = ('N', 'KN', 2)
171                                 else :
172                                         nxt = ('KN', 2)
173                         # 'tagliaro'
174                         elif st[pos+1:pos+3] == 'LI' and not
                                is_slavo_germanic :
175                                 nxt = ('KL', 'L', 2)
176                         # -ges-,-gep-,-gel-, -gie- at beginning
177                         elif pos == first and (st[pos+1] == 'Y' \
178                                or st[pos+1:pos+3] in ["ES", "EP", "EB", "EL",
                                        "EY", "IB", "IL", "IN", "IE", "EI", "ER"]) :
179                                 nxt = ('K', 'J', 2)
180                         # -ger-,  -gy-
181                         elif (st[pos+1:pos+2] == 'ER' or st[pos+1] == 'Y')
                                 \
182                                and st[first:first+6] not in ["DANGER", "RANGER
                                        ", "MANGER"] \
183                                and st[pos-1] not in ['E', 'I'] and st[pos-1:
                                        pos+2] not in ['RGY', 'OGY'] :
184                                 nxt = ('K', 'J', 2)
185                         # italian e.g, 'biaggi'
186                         elif st[pos+1] in ['E', 'I', 'Y'] or st[pos-1:pos
                                +3] in ["AGGI", "OGGI"] :
187                                 # obvious germanic
188                                 if st[first:first+4] in ['VON ', 'VAN ']
                                        or st[first:first+3] == 'SCH' \
189                                        or st[pos+1:pos+3] == 'ET' :
190                                         nxt = ('K', 2)
191                                 else :
192                                         # always soft if french ending
```

```
193                                     if st[pos+1:pos+5] == 'IER ' :
194                                             nxt = ('J', 2)
195                                     else :
196                                             nxt = ('J', 'K', 2)
197                             elif st[pos+1] == 'G' :
198                                     nxt = ('K', 2)
199                             else :
200                                     nxt = ('K', 1)
201                     elif ch == 'H' :
202                             # only keep if first & before vowel or btw. 2
                                vowels
203                             if (pos == first or st[pos-1] in vowels) and st[
                                pos+1] in vowels :
204                                     nxt = ('H', 2)
205                             else : # (also takes care of 'HH')
206                                     nxt = (None, 1)
207                     elif ch == 'J' :
208                             # obvious spanish, 'jose', 'san jacinto'
209                             if st[pos:pos+4] == 'JOSE' or st[first:first+4] ==
                                 'SAN ' :
210                                     if (pos == first and st[pos+4] == ' ') or
                                        st[first:first+4] == 'SAN ' :
211                                             nxt = ('H',)
212                                     else :
213                                             nxt = ('J', 'H')
214                             elif pos == first and st[pos:pos+4] != 'JOSE' :
215                                     nxt = ('J', 'A') # Yankelovich/Jankelowicz
216                             else :
217                                     # spanish pron. of e.g. 'bajador'
218                                     if st[pos-1] in vowels and not
                                        is_slavo_germanic \
219                                        and st[pos+1] in ['A', 'O'] :
220                                             nxt = ('J', 'H')
```

```
221                             else :
222                                     if pos == last :
223                                             nxt = ('J', ' ')
224                                     else :
225                                             if st[pos+1] not in ["L",
                                                    "T", "K", "S", "N", "M"
                                                    , "B", "Z"] \
226                                                and st[pos-1] not in ["
                                                       S", "K", "L"] :
227                                                     nxt = ('J',)
228                                             else :
229                                                     nxt = (None, )
230                     if st[pos+1] == 'J' :
231                             nxt = nxt + (2,)
232                     else :
233                             nxt = nxt + (1,)
234             elif ch == 'K' :
235                     if st[pos+1] == 'K' :
236                             nxt = ('K', 2)
237                     else :
238                             nxt = ('K', 1)
239             elif ch == 'L' :
240                     if st[pos+1] == 'L' :
241                             # spanish e.g. 'cabrillo', 'gallegos'
242                             if (pos == (last - 2) and st[pos-1:pos+3]
                                    in ["ILLO", "ILLA", "ALLE"]) \
243                                    or ((st[last-1:last+1] in ["AS", "OS"]
                                           or st[last] in ["A", "O"]) \
244                                    and st[pos-1:pos+3] == 'ALLE') :
245                                     nxt = ('L', '', 2)
246                             else :
247                                     nxt = ('L', 2)
248                     else :
```

```
249                         nxt = ('L', 1)
250                 elif ch == 'M' :
251                     if st[pos+1:pos+4] == 'UMB' \
252                        and (pos + 1 == last or st[pos+2:pos+4] == 'ER'
                              ) \
253                        or st[pos+1] == 'M' :
254                         nxt = ('M', 2)
255                     else :
256                         nxt = ('M', 1)
257                 elif ch == 'N' :
258                     if st[pos+1] == 'N' :
259                         nxt = ('N', 2)
260                     else :
261                         nxt = ('N', 1)
262                 elif ch == u'  ' :
263                     nxt = ('N', 1)
264                 elif ch == 'P' :
265                     if st[pos+1] == 'H' :
266                         nxt = ('F', 2)
267                     elif st[pos+1] in ['P', 'B'] : # also account for
                            "campbell", "raspberry"
268                         nxt = ('P', 2)
269                     else :
270                         nxt = ('P', 1)
271                 elif ch == 'Q' :
272                     if st[pos+1] == 'Q' :
273                         nxt = ('K', 2)
274                     else :
275                         nxt = ('K', 1)
276                 elif ch == 'R' :
277                     # french e.g. 'rogier', but exclude 'hochmeier'
278                     if pos == last and not is_slavo_germanic \
279                        and st[pos-2:pos] == 'IE' and st[pos-4:pos-2]
```

```
                    not in ['ME', 'MA'] :
280                         nxt = ('', 'R')
281                 else :
282                         nxt = ('R',)
283                 if st[pos+1] == 'R' :
284                         nxt = nxt + (2,)
285                 else :
286                         nxt = nxt + (1,)
287         elif ch == 'S' :
288                 # special cases 'island', 'isle', 'carlisle', '
                    carlysle'
289                 if st[pos-1:pos+2] in ['ISL', 'YSL'] :
290                         nxt = (None, 1)
291                 # special case 'sugar-'
292                 elif pos == first and st[first:first+5] == 'SUGAR'
                     :
293                         nxt =('X', 'S', 1)
294                 elif st[pos:pos+2] == 'SH' :
295                         # germanic
296                         if st[pos+1:pos+5] in ["HEIM", "HOEK", "
                            HOLM", "HOLZ"] :
297                                 nxt = ('S', 2)
298                         else :
299                                 nxt = ('X', 2)
300                 # italian & armenian
301                 elif st[pos:pos+3] in ["SIO", "SIA"] or st[pos:pos
                    +4] == 'SIAN' :
302                         if not is_slavo_germanic :
303                                 nxt = ('S', 'X', 3)
304                         else :
305                                 nxt = ('S', 3)
306                 # german & anglicisations, e.g. 'smith' match '
                    schmidt', 'snider' match 'schneider'
```

```
307                     # also, -sz- in slavic language altho in hungarian
                            it is pronounced 's'
308                     elif (pos == first and st[pos+1] in ["M", "N", "L"
                            , "W"]) or st[pos+1] == 'Z' :
309                             nxt = ('S', 'X')
310                             if st[pos+1] == 'Z' :
311                                     nxt = nxt + (2,)
312                             else :
313                                     nxt = nxt + (1,)
314                     elif st[pos:pos+2] == 'SC' :
315                             # Schlesinger's rule
316                             if st[pos+2] == 'H' :
317                                     # dutch origin, e.g. 'school', '
                                            schooner'
318                                     if st[pos+3:pos+5] in ["OO", "ER",
                                            "EN", "UY", "ED", "EM"] :
319                                             # 'schermerhorn', '
                                                    schenker'
320                                             if st[pos+3:pos+5] in ['ER
                                                    ', 'EN'] :
321                                                     nxt = ('X', 'SK',
                                                            3)
322                                             else :
323                                                     nxt = ('SK', 3)
324                                     else :
325                                             if pos == first and st[
                                                    first+3] not in vowels
                                                    and st[first+3] != 'W'
                                                    :
326                                                     nxt = ('X', 'S',
                                                            3)
327                                             else :
328                                                     nxt = ('X', 3)
```

```
                elif st[pos+2] in ['I', 'E', 'Y'] :
                        nxt = ('S', 3)
                else :
                        nxt = ('SK', 3)
            # french e.g. 'resnais', 'artois'
            elif pos == last and st[pos-2:pos] in ['AI', 'OI']
                :
                nxt = ('', 'S', 1)
            else :
                nxt = ('S',)
                if st[pos+1] in ['S', 'Z'] :
                        nxt = nxt + (2,)
                else :
                        nxt = nxt + (1,)
        elif ch == 'T' :
            if st[pos:pos+4] == 'TION' :
                nxt = ('X', 3)
            elif st[pos:pos+3] in ['TIA', 'TCH'] :
                nxt = ('X', 3)
            elif st[pos:pos+2] == 'TH' or st[pos:pos+3] == '
                TTH' :
                # special case 'thomas', 'thames' or
                    germanic
                if st[pos+2:pos+4] in ['OM', 'AM'] or st[
                    first:first+4] in ['VON ', 'VAN '] \
                    or st[first:first+3] == 'SCH' :
                        nxt = ('T', 2)
                else :
                        nxt = ('O', 'T', 2)
            elif st[pos+1] in ['T', 'D'] :
                nxt = ('T', 2)
            else :
                nxt = ('T', 1)
```

```
358                 elif ch == 'V' :
359                         if st[pos+1] == 'V' :
360                                 nxt = ('F', 2)
361                         else :
362                                 nxt = ('F', 1)
363                 elif ch == 'W' :
364                         # can also be in middle of word
365                         if st[pos:pos+2] == 'WR' :
366                                 nxt = ('R', 2)
367                         elif pos == first and (st[pos+1] in vowels or st[
                               pos:pos+2] == 'WH') :
368                                 # Wasserman should match Vasserman
369                                 if st[pos+1] in vowels :
370                                         nxt = ('A', 'F', 1)
371                                 else :
372                                         nxt = ('A', 1)
373                         # Arnow should match Arnoff
374                         elif (pos == last and st[pos-1] in vowels) \
375                            or st[pos-1:pos+5] in ["EWSKI", "EWSKY", "OWSKI
                                  ", "OWSKY"] \
376                            or st[first:first+3] == 'SCH' :
377                                 nxt = ('', 'F', 1)
378                         # polish e.g. 'filipowicz'
379                         elif st[pos:pos+4] in ["WICZ", "WITZ"] :
380                                 nxt = ('TS', 'FX', 4)
381                         else : # default is to skip it
382                                 nxt = (None, 1)
383                 elif ch == 'X' :
384                         # french e.g. breaux
385                         nxt = (None,)
386                         if not(pos == last and (st[pos-3:pos] in ["IAU", "
                               EAU"] \
387                            or st[pos-2:pos] in ['AU', 'OU'])):
```

```
                        nxt = ('KS',)
                if st[pos+1] in ['C', 'X'] :
                        nxt = nxt + (2,)
                else :
                        nxt = nxt + (1,)
        elif ch == 'Z' :
                # chinese pinyin e.g. 'zhao'
                if st[pos+1] == 'H' :
                        nxt = ('J',)
                elif st[pos+1:pos+3] in ["ZO", "ZI", "ZA"] \
                   or (is_slavo_germanic and pos > first and st[
                      pos-1] != 'T') :
                        nxt = ('S', 'TS')
                else :
                        nxt = ('S',)
                if st[pos+1] == 'Z' :
                        nxt = nxt + (2,)
                else :
                        nxt = nxt + (1,)
        # ----------------------------------
        # --- end checking letters------
        # ----------------------------------
        #print str(nxt)
        if len(nxt) == 2 :
                if nxt[0] :
                        pri += nxt[0]
                        sec += nxt[0]
                pos += nxt[1]
        elif len(nxt) == 3 :
                if nxt[0] :
                        pri += nxt[0]
                if nxt[1] :
                        sec += nxt[1]
```

```
420                         pos += nxt[2]
421         if pri == sec :
422                 return (pri, None)
423         else :
424                 return (pri, sec)
425         return pri
426
427 if __name__ == '__main__' :
428         names = {'maurice':('MRS', None),'aubrey':('APR', None),'cambrillo
                ':('KMPRL','KMPR')\
429         ,'heidi':('HT', None),'katherine':('K0RN','KTRN'),'Thumbail':('0
                MPL','TMPL')\
430                 ,'catherine':('K0RN','KTRN'),'richard':('RXRT','RKRT'),'
                    bob':('PP', None)\
431         ,'eric':('ARK', None),'geoff':('JF','KF'),'Through':('0R','TR'), '
                Schwein':('XN', 'XFN')\
432                 ,'dave':('TF', None),'ray':('R', None),'steven':('STFN',
                    None),'bryce':('PRS', None)\
433         ,'randy':('RNT', None),'bryan':('PRN', None),'Rapelje':('RPL',
                None)\
434                 ,'brian':('PRN', None),'otto':('AT', None),'auto':('AT',
                    None), 'Dallas':('TLS', None)\
435         , 'maisey':('MS', None), 'zhang':('JNK', None), 'Chile':('XL',
                None)\
436         ,'Jose':('HS', None), 'Arnow':('ARN','ARNF'), 'solilijs':('SLLS',
                None)\
437                 , 'Parachute':('PRKT', None), 'Nowhere':('NR', None), 'Tux
                    ':('TKS', None)}
438         for name in names.keys() :
439                 assert (dm(name) == names[name]), 'For "%s" function
                    returned %s. Should be %s.' % (name, dm(name), names[
                    name])
440
```

```
#st = 'Stasha'
#st = 'Simoneit'

#print dm(st)
```

Appendix I. Mutator Python Code

```
#!/usr/bin/env python
from __future__ import print_function

import random
import csv
import argparse
import json

# Running this should be as easy as marking it as executable (chmod +x
    mutator.py), and running it like so:
# ./mutator.py <file> <id_field> <out_filename> <mutation> [<mutation>
    ...]
# Where
# - file is the input file to read and select from
# - id_field is the name of the column that has the IDs (assumed to be
    integers)
# - out_filename is the beginning part of the name of the output (so if
    you wanted the output to be "mutated_dataset.csv", then say "
    mutated_dataset" here)
# - mutation looks like this: <n>:<m>:<d>, where n is the number of rows
    to select, m is the number of mutations to do, and d is the number of
    duplicates to generate for each row.  You can specify multiple
    mutations, although the rows that are selected for each mutation are
    removed from the pool so that they won't be selected later.
# Example: 100:3:2 200:5:3 first selects 100 rows from the input and
```

```
    creates 2 duplicates for each of those rows, mutating each 3 times, and
     then selects 200 other rows from the input and creates 3 duplicates
    for each of the rows, mutating each 5 times. (The output would have a
    total of (100*2) + (200*3) = 800 rows).
# An oracle file is automatically generated to go along with the dataset.
# Take the output name, and add ".oracle.json" instead of ".csv".
#
# Example command:
# ./mutator.py dataset_A_1000.csv rec_id dups_A_1000 300:6:2 100:7:4
# this will read dataset_A_1000.csv, output to dups_A_1000.csv, write an
    oracle to dups_A_1000.oracle.json, use "rec_id" as the ID column, and
    select 300 rows, make 2 duplicates with 6 mutations, and then select
    100 rows, make 4 duplicates with 7 mutations.
#
# You can optionally specify a -s <SEED> or --seed <SEED> option, which
    expects an integer, and is used as the seed for the random number
    generator.
# You can also specify -e <FIELD> or --exclude-field <FIELD> option, which
     excludes a field from being mutated.

parser = argparse.ArgumentParser(description='Mutate a Dataset')
parser.add_argument('-s', '--seed', type=int, help='Random Seed', default=
    None)
parser.add_argument('-e', '--exclude-field', type=str, action='append',
    default=[])
parser.add_argument('file', type=str, help='File to Mutate')
parser.add_argument('id_field', type=str, help='ID Field')
parser.add_argument('out_filename', type=str, help='Name of output (add .
    csv or .oracle.json)')
parser.add_argument('mutation', type=str, nargs='+')

def mutate(filename, config):
    random.seed(config.get('seed', None))
```

```
37      print('Random Seeded')
38      records, fields = load_records(filename)
39
40      id_field = config['id_field']
41      print('Loaded Records')
42      print('ID Field: %s' % id_field)
43      print('Excluded Fields: %s' % ', '.join(config['exclude_fields']))
44      mutateable_fields = filter(lambda f: (f != config['id_field']) and not
            f in config['exclude_fields'], fields)
45
46      out_records = []
47      oracle = {}
48
49      for in_record_count, mutation_count, duplicates_count in config['
            mutations']:
50          print("select %d records, create %d duplicates with %d mutations"
                % (in_record_count, duplicates_count, mutation_count))
51          generate_mutations(out_records, oracle, records, id_field,
                mutateable_fields, in_record_count, mutation_count,
                duplicates_count)
52
53      write_output(config['out_filename'], fields, out_records, oracle)
54
55  def load_records(filename):
56      with open(filename, 'r') as inFile:
57          reader = csv.DictReader(inFile)
58          return [record for record in reader], reader.fieldnames
59
60  def write_output(out_filename, fields, out_records, oracle):
61      with open(out_filename + '.csv', 'w') as csv_out:
62          writer = csv.DictWriter(csv_out, fields)
63          writer.writeheader()
64          for r in out_records:
```

```
65              writer.writerow(r)
66      with open(out_filename + '.oracle.json', 'w') as json_out:
67          json.dump(oracle, json_out)
68
69
70  def generate_mutations(out_records, oracle, in_records, id_field,
        mutateable_fields, in_count, mutation_count, dup_count):
71      while in_count > 0 and len(in_records) > 0:
72          idx = random.randrange(len(in_records))
73          record = in_records[idx]
74          in_records = in_records[:idx] + in_records[idx + 1:]
75
76          if not record[id_field] in oracle:
77              oracle[record[id_field]] = []
78
79          for d in xrange(dup_count):
80              out_record = mutate_record(dict(record), mutateable_fields,
                    mutation_count)
81              out_record[id_field] = str(int(out_record[id_field]) + d)
82              oracle[record[id_field]].append(out_record[id_field])
83              out_records.append(out_record)
84
85          in_count -= 1
86      random.shuffle(out_records)
87
88  class InvalidMutationException(Exception):
89      pass
90
91
92  CHARSET = 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789
        .!?'
93  def random_char(charset=CHARSET):
94      return random.choice(charset or CHARSET)
```

```
95
96 def letter_insert(field, charset=None):
97     if field is None:
98         raise InvalidMutationException()
99     position = random.randrange(len(field) + 1)
100     return field[:position] + random_char(charset) + field[position:]
101
102 def letter_delete(field):
103     if field is None or len(field) == 0:
104         raise InvalidMutationException()
105     position = random.randrange(len(field))
106     if position == len(field) - 1:
107         return field[:position]
108     return field[:position] + field[position + 1:]
109
110 def letter_replace(field, charset=None):
111     if field is None or len(field) == 0:
112         raise InvalidMutationException()
113     position = random.randrange(len(field))
114     return field[:position] + random_char(charset) + field[position + 1:]
115
116 def letter_move(field):
117     if field is None or len(field) < 2:
118         raise InvalidMutationException()
119     src_position = random.randrange(len(field))
120     dst_position = random.randrange(len(field) - 1)
121     char = field[src_position]
122     res = field[:src_position] + field[src_position + 1:]
123     return res[:dst_position] + char + res[dst_position:]
124
125 def drop_field(field):
126     if field is None or field == '':
127         raise InvalidMutationException()
```

```
    return ''

def reorder_words(field):
    words = field.split(' ')
    if len(words) < 2:
        raise InvalidMutationException()
    src_position = random.randrange(len(words))
    dst_position = random.randrange(len(words) - 1)
    w = words[src_position]
    res = words[:src_position] + words[src_position + 1:]
    return ' '.join(res[:dst_position] + [w] + res[dst_position:])

MUTATORS = (
    letter_insert, letter_insert, letter_insert,
    letter_delete, letter_delete, letter_delete,
    letter_replace, letter_replace, letter_replace,
    letter_move, letter_move, letter_move,
    drop_field,
    reorder_words, reorder_words
)
def mutate_record(record, fields, count=1):
    while count > 0:
        try:
            field = random.choice(fields)
            record[field] = random.choice(MUTATORS)(record[field])
            count -= 1
        except InvalidMutationException:
            pass
    return record

if __name__ == '__main__':
    args = parser.parse_args()

```

```
    mutate(args.file, {
        'seed': args.seed,
        'id_field': args.id_field,
        'exclude_fields': args.exclude_field,
        'mutations': map(lambda m: map(int, m.split(':')), args.mutation),
        'out_filename': args.out_filename
    })
```

Appendix J. Economic Model Python Code

```
# profit.py

import numpy as np
from matplotlib import pyplot as plt
import argparse
import csv

parser = argparse.ArgumentParser(description='Plot an Economic Model
    optimization against a ROC curve')
parser.add_argument('file', type=str, help='Run Data to Plot')
parser.add_argument('a', type=float, help='penalty for being wrong (
    positive value)')
parser.add_argument('b', type=float, help='reward for being right (
    positive value)')

def load_data(filename):
    data = []
    with open(filename, 'r') as infile:
        for item in csv.DictReader(infile):
            data.append(item)
    return data

```

```
20 def plot(data, a, b):
21     # First Method
22     comp_specificity = [float(item['comp_specificity']) for item in data]
23     sensitivity = [float(item['sensitivity']) for item in data]
24     level = [float(item['level']) for item in data]
25 
26     x = np.arange(0.0, 1.1, 0.1)
27     y = np.arange(0.0, 1.1, 0.1)
28 
29     plt.figure(0)
30 
31     plt.plot(comp_specificity, sensitivity, 'o')
32     plt.plot(comp_specificity, sensitivity, '-', label='ROC curve')
33     plt.plot(x, y, '--', label="Line of no-discrimination")
34 
35     plt.xlim([0,1])
36     plt.ylim([0,1])
37 
38     plt.xlabel('False Positive Rate')
39     plt.ylabel('True Positive Rate')
40 
41     n = len(sensitivity)
42     profit = [0]*n
43 
44     for i in range(0, n):
45         profit[i] = a*(1 - comp_specificity[i]) + b*sensitivity[i]
46 
47     max_profit_index = profit.index(max(profit))
48 
49     profit_x = np.arange(-1.0, 1.1, 0.1)
50 
51     # m = slope
52     m = float(a)/b
```

```
53
54      # profit_y = m*(profit_x - comp_specificity_of_max_profit) +
            sensitivity_of_max_profit
55      profit_y = m*(profit_x - comp_specificity[max_profit_index]) +
            sensitivity[max_profit_index]
56
57      print "profit_y = %f (x - %f) + %f" %(m, comp_specificity[
            max_profit_index], sensitivity[max_profit_index])
58      print "optimal tolerance level:", level[max_profit_index]
59
60      plt.plot(profit_x, profit_y, '--', label="Profit function")
61
62      plt.legend(loc = 'lower right')
63
64      # Second Method
65      false_positives = [float(item['false_positives']) for item in data]
66      true_positives = [float(item['true_positives']) for item in data]
67
68      c_negatives = float(data[0]['false_positives']) + float(data[0]['
            true_negatives'])
69      c_positives = float(data[0]['true_positives']) + float(data[0]['
            false_negatives'])
70      x_2 = np.arange(0.0, c_negatives * 1.1, c_negatives / 10.0)
71      y_2 = np.arange(0.0, c_positives * 1.1, c_positives / 10.0)
72
73      plt.figure(1)
74
75      plt.plot(false_positives, true_positives, 'o')
76      plt.plot(false_positives, true_positives, '-')
77      plt.plot(x_2, y_2, '--', label="Line of no-discrimination")
78
79      plt.xlim([0,c_negatives])
80      plt.ylim([0,c_positives])
```

```
81
82      plt.xlabel('False Positives')
83      plt.ylabel('True Positives')
84
85      fp_cost = -1 * a * c_positives
86      tp_cost = b * c_negatives
87
88      profit_2 = [0] * n
89      for i in range(0, n):
90          profit_2[i] = (fp_cost * false_positives[i]) + (tp_cost *
                true_positives[i])
91
92      max_profit_index_2 = profit_2.index(max(profit_2))
93
94      profit_x_2 = np.arange(-1.0 * c_negatives, 1.1 * c_negatives, 0.1 *
            c_negatives)
95
96      # m = slope
97      m_2 = float(-fp_cost)/tp_cost
98
99      # profit_y = m*(profit_x - comp_specificity_of_max_profit) +
            sensitivity_of_max_profit
100     profit_y_2 = m_2*(profit_x_2 - false_positives[max_profit_index_2]) +
            true_positives[max_profit_index_2]
101     print "fp_cost = %f" % fp_cost
102     print "tp_cost = %f" % tp_cost
103     print "profit_y_2 = %f (x - %f) + %f" %(m_2, false_positives[
            max_profit_index_2], true_positives[max_profit_index_2])
104     print "optimal tolerance level for profit_y_2:", level[
            max_profit_index_2]
105
106     plt.plot(profit_x_2, profit_y_2, '--', label="Profit function")
107
```

```
108     plt.legend(loc = 'lower right')
109
110     plt.show()
111
112 if __name__ == '__main__':
113     args = parser.parse_args()
114
115     print args.file
116     data = load_data(args.file)
117
118     plot(data, args.a, args.b)
```

Bibliography

[1] Junghoo Cho, Narayanan Shivakumar, and Hector Garcia-Molina. Finding Replicated Web Collections. In *Proceedings 2000 ACM SIGMOD International Conference Management of Data*, pages 355–366, 2000.

[2] Ruslan Mitkov. *Anaphora Resolution: The State of the Art*. Longman, 1999.

[3] Halbert L. Dunn. Record Linkage. *American Journal of Public Health*, 36(12):1412–1416, 1946.

[4] H.B. Newcombe, J.M. Kennedy, S.J. Axford, and A.P. James. Automatic Linkage of Vital Records. *Science*, 130(3381):954–959, 1959.

[5] I.P. Fellegi and A.B Sunter. A Theory for Record Linkage. *Journal of the American Statistical Association*, 64(328):1183–1210, 1969.

[6] William W. Cohen, Henry Kautz, and David McAllester. Hardening Soft Information Sources. In *Proceedings Sixth ACM SIGKDD International Conference Knowledge Discovery and Data Mining*, pages 255–259, 2000.

[7] Mikhail Bilenko, Raymond Mooney, William Cohen, Pradeep Ravikumar, and Fienberg Fienberg. Adaptive Name Matching in Information Integration. *IEEE Intelligent Systems*, 18(5):16–23, 2003.

[8] Sunita Sarawagi and Anuradha Bhamidipaty. Interactive Deduplication Using Active Learning. In *Proceedings Eighth ACM SIGKDD International Conference Knowledge Discovery and Data Mining*, pages 269–278, 2002.

[9] Y. Richard Wang and Stuart E. Madrick. The Inter-Database Instance Identification Problem in Integrating Autonomous Systems. In *Proceedings Fifth IEEE International Conference Data Engineering*, pages 46–55, 1989.

[10] Mauricio A. Hernandez and Salvatore J. Salvatore. Real-world Data is Dirty: Data Cleansing and the Merge/Purge Problem. *Data Mining and Knowledge Discovery*, 2(1):9–37, 1998.

[11] Ahmen K. Elmagarmid, Panagiotis G. Ipeirotis, and Vassilois S. Verykios. Duplicate Record Detection: A Survey. *IEEE Transactions on Knowledge and Data Engineering*, 19(1):1–16, 2007.

[12] Abhirup Chatterjee and Arie Segev. Data Manipulation in Heterogeneous Databases. *ACM SIGMOD Record*, 20(4):64–68, 1991.

[13] Peter Christen and Tim Churches. Febrl - Freely Extensible Biomedical Record Linkage. open sorce release 0.3.1, 2005.

[14] Vladimir I. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions, and Reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.

[15] William W. Cohen. Record Linkage Tutorial: Distance Metrics for Text. Lecture from https://www.cs.cmu.edu/∼wcohen/, January 2010.

[16] M. S. Waterman, T. F. Smith, and W. A. Beyer. Some Biological Sequence Metrics. *Advances in Mathematics*, 20(4):367–387, 1976.

[17] T. F. Smith and M. S. Waterman. Identification of Common Molecular Subsequences. *Journal of Molecular Biology*, 147(1):195–197, 1981.

[18] Matthew A. Jaro. Unimatch: A Record Linkage System: Users Manual. Technical report, US Bureau of the Census, Washington, D.C., 1976.

[19] Nick Koudas, Sunita Sarawagi, and Divesh Srivastava. Record Linkage: Similarity Measures and Algorithms. In *Proceeding of the 2006 ACM SIGMOD International Conference on Management of Data*, pages 802–803, 2006.

[20] J. R. Ullmann. A Binary n-Gram Technique for Automatic Correction of Substitution, Deletion, Insertion, and Reversal Errors in Words. *The Computer Journal*, 20(2):141–147, 1977.

[21] Esko Ukkonen. Approximate String Matching with q-Grams and Maximal Matches. *Theoretical Computer Science*, 92(1):191–211, 1992.

[22] Erkki Sutinen and Jorma Tarhio. On Using q-Gram Locations in Approximate String Matching. In *Proceedings Third Annual European Symposium Algorithms*, pages 327–340, 1995.

[23] Alvaro E. Monge and Charles P. Elkan. The Field Matching Problem: Algorithms and Applications. In *Proceedings Second International Conference Knowledge Discovery and Data Mining*, pages 267–270, 1996.

[24] William W. Cohen. Integration of Heterogeneous Databases Without Common Domains Using Queries Based on Textual Similarity. In *Proceedings 1998 ACM SIGMOD International Conference Management of Data*, pages 201–212, 1998.

[25] Luis Gravano, Panagiotis G. Ipeirotis, Nick Koudas, and Divesh Srivastava. Text Joins in an RDBMS for Web Data Integration. In *Proceedings 12th International World Wide Web Conference*, pages 90–101, 2003.

[26] Robert C. Russell and Margaret K. Odell. U.S. Patent 1,261,167. http://patft.uspto.gov/netahtml/srchnum.htm, April 1918.

[27] Robert C. Russell and Margaret K. Odell. U.S. Patent 1,435,663. http://patft.uspto.gov/netahtml/srchnum.htm, November 1922.

[28] Robert Korn. http://freepages.genealogy.rootsweb.ancestry.com/∼korn/pafg04.htm, 2001.

[29] FamilySearch. Illinois, Northern District Naturalization Index, 1840-1950. http://familysearch.org/learn/wiki/en/Illinois,_Northern_District,_Indexes_to_Naturalization_Records_(FamilySearch_Historical_Records), 2013.

[30] Robert L. Taft. Name Search Techniques. Technical Report 1, New York State Identification and Intelligence System, Albany, New York, 1970.

[31] L. E. Gill. OX-LINK: The Oxford Medical Record Linkage System. In *Proceedings International Record Linkage Workshop and Exposition*, pages 15–33, 1997.

[32] Lawrence Philips. Hanging on the Metaphone. *Computer Language Magazine*, 7(12):39–44, 1990.

[33] Lawrence Philips. The Double Metaphone Search Algorithm. *C/C++ Users Journal*, 18(5), 2000.

[34] Metaphone. http://en.wikipedia.org/wiki/Metaphone, September 2013.

[35] Shanti Gomatam, Randy Carter, Mario Ariet, and Glenn Mitchell. An Empirical Comparison of Record Linkage Procedures. *Statistics in Medicine*, 21(10):1485–1496, 2002.

[36] Ee-Peng Lim, Satya Prabhakar, Jaideep Srivastava, and James Richardson. Entity Identification in Database Integration. In *Proceedings Ninth IEEE International Conference Data Engineering*, pages 294–301. IEEE Computer Society Press, 1993.

[37] Vassilios S. Verykios, George V. Moustakides, and Mohamed G. Elfeky. A Bayesian Decision Model for Cost Optimal Record Matching. *The VLDB Journal*, 12(1):28–40, 2003.

[38] William E. Winkler. Improved Decision Rules in the Fellegi-Sunter Model of Record Linkage. Statistical Research Report Series RR93/12, US Bureau of the Census, Washington, D.C., 1993.

[39] William E. Winkler. Methods for Record Linkage and Bayesian Networks. Statistical Research Report Series RRS2002/05, US Bureau of the Census, Washington, D.C., 2002.

[40] N. S. D'Andrea Du Bois Jr. A Solution to the Problem of Linking Multivariate Documents. *Journal of the American Statistical Association*, 64(325):163–174, 1969.

[41] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification, 2nd Ed.* Wiley Interscience Publication, 2001.

[42] Sheila Tejada, Craig A. Knoblock, and Steven Minton. Learning Object Identification Rules for Information Integration. *Information Systems*, 26(8):607–633, 2001.

[43] Alvaro Monge and Charles P. Elkan. An Efficient Domain-Independent Algorithm for Detecting Approximately Duplicate Database Records. In *Proceedings Second ACM SIGMOD Workshop Research Issues in Data Mining and Knowledge Discovery*, pages 23–29, 1997.

[44] Andrew McCallum, Kamal Nigam, and Lyle H. Ungar. Efficient Clustering of High-Dimensional Data Sets with Application to Reference Matching. In *Proceedings Sixth ACM SIGKDD International Conference Knowledge Discovery and Data Mining*, pages 169–178, 2000.

[45] William Cohen and Jacob Richman. Learning to Match and Cluster Large High-Dimensional Data Sets for Data Integration. In *Proceedings Eighth ACM SIGKDD International Conference Knowledge Discovery and Data Mining*, 2002.

[46] Luis Gravano, Panagiotis G. Ipeirotis, H. V. Jagadish, Nick Koudas, S. Muthukrishnan, and Divesh Srivastava. Approximate String Joins in a Database (Almost) for Free. In *Proceedings 27th International Conference of Very Large Databases*, pages 491–500, 2001.

[47] Surajit Chaudhuri, Kris Ganjam, Venkatesh Ganti, and Rajeev Motwani. Robust and Efficient Fuzzy Match for Online Data Cleaning. In *Proceedings 2003 ACM SIGMOD International Conference Management of Data*, pages 313–324, 2003.

[48] Rohan Baxter, Peter Christen, and Tim Churches. A Comparison of Fast Blocking Methods for Record Linkage. In *Proceedings ACM SIGKDD 2003 Workshop Data Cleaning, Record Linkage, and Object Consolidation*, pages 25–27, 2003.

[49] Peter Christen. Febrl - An Open Source Data Cleaning Deduplication and Record Linkage System with a Graphical User Interface. In *ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD '08)*, 2008.

[50] Mohamed G. Elfeky, Vassilios S. Verykios, and Ahmed K. Elmagarmid. TAILOR: A Record Linkage Toolbox. In *Proceedings 18th IEEE International Conference Data Engineering*, pages 17–28, 2002.

[51] William W. Cohen. Data Integration Using Similarity Joins and a Word-Based Information Representation Language. In *ACM Transactions on Information Systems*, volume 18, pages 288–321, 2000.

[52] William E. Yancey. Bigmatch: A Program for Extracting Probable Matches from a Large File for Record Linkage. Statistical Research Report Series RRC2002/01, US Bureau of the Census, Washington, D.C., 2002.

[53] Lifang Gu, Rohan Baxter, Deanne Vickers, and Chris Rainsford. Record Linkage: Current Practice and Future Directions. Technical report, CSIRO Mathematical and Information Sciences, 2003.