Utrecht University

Identifying Historical Person Names using Weighted Edit Distance

Author:
Richard Oosterlaken
3952029

Supervisor:
dr. ir. G. Bloothooft

Second Supervisor:
dr. A.J. Feelders

Bachelor Thesis Artificial Intelligence, 7.5 ECTS

10 July, 2018

Abstract

In the process of automated record linkage, dealing with name variation is often done via limited means, such as an edit distance plus a threshold value. However, names vary in ways that default similarity measures can not reliably coped with.

In an effort to overcome this threshold, an alternative, 'weighted' edit distance is proposed. This weighted edit distance would assign costs to operations based on previously seen operations that transform names into their known variants. Names often vary in similar ways, by adding the same suffixes, to name an example. Operations that transform names into their name variants are therefore likely to be similar to the operations that would be seen between names and their yet unseen name variants.

In this paper, methods are defined that gather the data required to create a cost model that assigns costs for the operations of a weighted edit distance. Suggestions were then given on how to implement a cost model and a weighted edit distance based on this data, as well as how to test these implementations.

Contents

1 Introduction

Names, though being one of a person's most identifying properties, can often be used interchangeably while referring to the same person. The variation in names an individual can have can hinder identification, because it takes additional information, like age, place of birth and family ties, to determine definitively that sources refer to the same person.

In the Netherlands, registering a name in the Civil Registration became mandatory in 1811 (Bloothooft and Schraagen, 2015). Most of this civil registration has since been digitised and is made public under the *WieWasWie* domain (www.wiewaswie.nl). The civil registration contains birth, marriage and death certificates. A civil certificate holds information on its event, including names of the relevant person, names of their parents, ages, dates and locations. There has been an ongoing genealogical effort to link these digitised civil certificates to one another through automated means, a process known as 'record linkage'.

In the past, there was a lack of caution to make sure that a name did not differ between certificates. Consequently, there may be inconsistency in the certificates referring to the same individual. For example, a person can be named *Johannes* in the birth certificate and *Jan* in the marriage certificate. In addition, digitisation is a laborious and error prone process, which introduces a lot of spelling variation, mainly typos, in the civil certificates. Linking of certificates becomes a non-trivial process, because the same individual can be referred to with different names, differently spelled names or misspelled names. Bloothooft and Schraagen (2015) require at least five names per certificate, the names of a person and their parents. Matching four of those can be enough to link certificates with confidence. The non-matching name then generates a 'name variant'; a name that can be used interchangeably to refer to the same person.

To solve the problem of variation in names between certificates, efforts are being made to create a list of variants for every name. The list can then be referenced every time that the similarity between a pair of names is questionable. Names, in this case, of course include first names and surnames. Record linkage would be sped up significantly and become more accurate, too. However, identification of these name variants is hard to automate and an automated way to identify yet unseen name variants is not known (Bloothooft and Schraagen, 2015).

A common method to measure the similarity between words or to circumvent spelling variation is to calculate an 'edit distance' (Levenshtein, 1966). An edit distance function defines different edit 'operations' that will alter a word in certain ways, like deleting or inserting a letter. Each of these operations has a predefined cost, which is chosen to be 1 in most cases. Edit distance functions calculate the minimum cost to transform a source word into a target word, using the operations that are defined beforehand. The available operations differ per type of edit distance.

The most widespread implementation of an edit distance is the 'Levenshtein distance' (Levenshtein, 1966; Wagner and Fischer, 1974). Operations used in the Levenshtein distance are *insertion*, *deletion* and *substitution*, which respectively add, remove and replace a single letter in the source word. A 'path' is a sequence of operations used to transform a source word into a target word. Edit distances are non-unique, as multiple paths can exist to make the same transformation. For example, Fig. 1 shows the name *Willemine* edited into the name *Wilmina*. Assuming the cost of all the operations is 1, there are two possible paths for this transformation; one that deletes the first L and one that deletes the second.

Digitised records contain a lot of typos where two keys were accidentally pressed in the wrong order, for example *Theodora* can be mistakenly spelled as *Thoedora*. An extension of the Levenshtein distance that also deals with swapping adjacent characters in the source word is called the 'optimal string alignment distance' (Wagner and Lowrance,

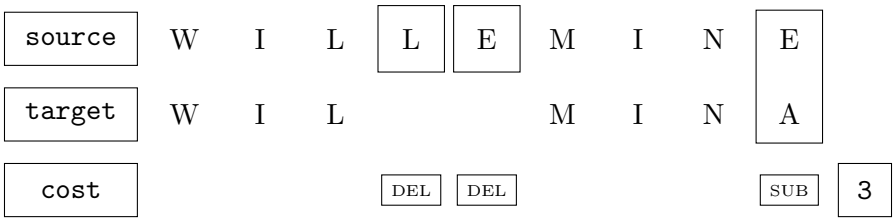

Figure 1: Levenshtein distance for *Willemine* and *Wilmina*, including operations. 'DEL' and 'SUB' are short for deletion and substitution respectively. The first L could have been deleted instead of the second one, which would have resulted in a different path with the same distance.

1975; Kruskal, 1983). The swapping operation, or *transposition*, is demonstrated in Fig. 2. Note that transposition is equal to the deletion of a character, followed by the insertion of that same character, one position further in the word. The optimal string alignment distance defaults the cost of transposition to 1, making the operation path cheaper than the one the Levenshtein distance would take.

Edit distances use a calculation table that gets filled in and reveals the final edit distance. A calculation table can be backtracked through to efficiently see the paths that were taken to make a transformation.

Often when measuring similarity between words, a hard, static threshold is defined. If the edit distance measure is below the threshold, words are considered similar. Bloothooft and Schraagen (2015) report how the edit distance between two names can be very high, yet the names will commonly be used interchangeably between records. The names *Jan* and *Johannes* are a good example of this. Contradictory, it is mentioned that a very short edit distance does not necessarily imply that two names are variants, like the surnames *Vos* and *Bos*. This results in simple edit distance implementations being unreliable metrics for similarity in record linkage.

This paper proposes an edit distance algorithm that learns from operations extracted from the edit distance calculation tables between name variants in the Dutch certificate data. A cost model will be created that decides the cost of the operations in this edit distance, given a certain context. The costs will be influenced by how often the same type of operation is seen in a similar context. The proposed edit distance then simply references this cost model with some contextual insight, before deciding on the execution of an operation.

The 'weighted edit distance' will have the same operations as the optimal string alignment distance. The optimal string alignment distance is favoured over the Levenshtein distance for defining name variant similarity. This because, assuming all operation costs are 1, the transposition operation can solve adjacent character typing errors in name variants cheaper than the Levenshtein distance does.

The similarity value calculated by the weighted edit distance will be influenced by the amount of operations that are likely to be used to transform a name into one of its variants. For example, in the Dutch certificate data, the surnames *Vos* and *Bos* are never

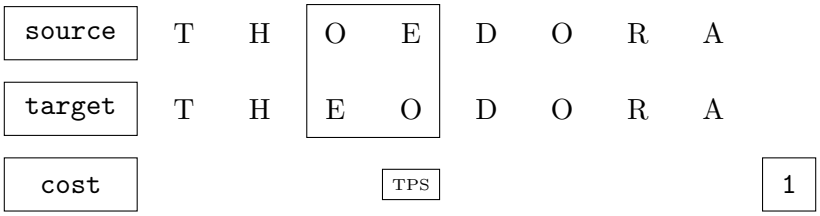

Figure 2: Optimal string alignment distance for *Thoedora* and *Theodora*. 'TPS' is short for transposition.

Marriage certificate
Date: February 23, 1840
Municipality: Sijbekarspel
Bridegroom: Pieter Houtlosser
Age: 32
Father: Jacob Houtlosser
Mother: Aafje Spruit
Bride: Aaltje Kort
Age: 22
Father: Jan Kort
Mother: Aafje Vorst

Figure 3: Example of data extracted from a wedding certificate.

used interchangeably. However, *Vosch* is a name variant of *Vos*, and *Bosch* is a name variant of *Bos*. The insertions of '*CH*' at the end is shared between these two pairs of variants, so these should be recognised and assigned low costs. Meanwhile, these variants do not provide any support for '*V*' into '*B*' substitution, so the cost for that operation shouldn't be altered until it is seen between other name variants.

The weighted edit distance will be able to deal with the limitations of a static edit distance threshold, by becoming more dynamic in the usage of certain operations than regular edit distance measures currently are.

The name variant data that the weighted edit distance is trained on will be separated in female first names, male first names and surnames. Splitting the name variant data is chosen for, because some combinations of operations might be more likely to happen to surnames as compared to first names, and vice-versa. For example, adding '*TJE*' at the end would be uncommon for surnames, but usual for female first names.

The structure of this paper is as follows. The name variant data used is elaborated on in Sect. 2. Sect. 3 describes the methods that were created to extract operations in their context from the differences between name variants. In Sect. 4 the results of applying these methods is shown, resulting in an operation count. Another analysis was done to extract substrings from the known name variants, to get an understanding of the combinations of characters that was seen in the name variants in general. Sect. 5 suggests implementations for a few types of cost models. It also elaborates on a draft of the weighted edit distance, and how this could be tested. The weighted edit distance is discussed in Sect. 6, as well as any limitations and future work. Finally, Sect. 7 concludes the paper.

2 Material

The weighed edit distance uses operations between known name variants as 'learning' data. Bloothooft and Schraagen (2015) extracted over half a million pairs of name variants from birth, marriage and death certificates in the *WieWasWie* database (previously known as *Genlias*), the digitised version of the Dutch Civil Registration. These name variants have been filtered Amongst the information that is stored in a record are a date and the names of the people involved, as well as the age and sex of the main person. The main person to whom a certificate refers may also be called the 'ego'. Fig. 3 shows an example of data collected from a marriage certificate. Most of the events in the certificates took place in the nineteenth or early twentieth century.

Two certificates can be cross-referenced to derive a link between them, provided enough of the certificates' values match. Bloothooft and Schraagen (2015) describe how accurate

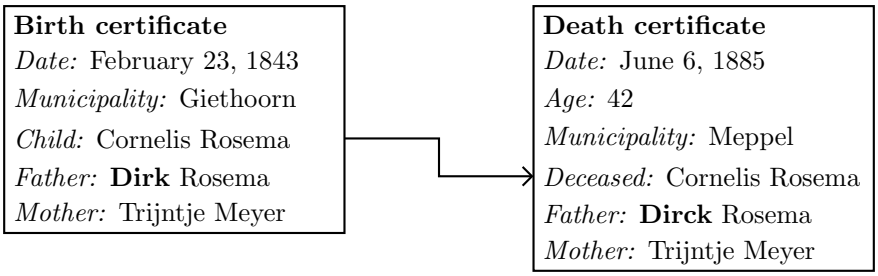

Figure 4: Example of matching birth and marriage certificate data for the person *Cornelis Rosema*. How the father's name is spelled as *Dirk* in the birth certificate, yet as *Dirck* in the death certificate.

linkage required an exact match of the year of birth and four out of at least five names of the ego and the parents. Composite names, such as *Wilhelmina Maria Keijzers*, were also considered and provide a stronger link between two certificates as more names can be exactly matched. The ego's first name always had to be an exact match, because otherwise the names of siblings could be mistaken for name variants. Note that the requirement of matching the ego's first name exactly, as well as three out of at least four other names was specific to the Dutch certificate data, and may differ for sets of similar certificates in other countries.

To extract name variant pairs, the dates of birth and all the names between linked certificates were considered. Dissimilarities between the names in certificates were identified and recorded as name variant pairs, where the Levenshtein distance and some re-alignment were used for special cases considering composite names (Bloothooft and Schraagen, 2015). An example match for the birth and death certificates of the ego *Cornelis Rosema* is shown in Fig. 4. The certificates do not match on the father's first name, which then produces the name variant pair *Dirk*/*Dircк*.

The lists of name variant pairs that was created by Bloothooft and Schraagen (2015) is a count of occurrences of name variant pairs throughout the *WieWasWie* database. All the name variants were recorded in uppercase, making the name variants essentially case-insensitive. It list contains some metadata per name variant pair. The metadata included the amount of occurrences of the pair, as well as the occurrences of the two name variants by themselves. Not all the names in the *WieWasWie* data were extracted. Therefore, it is also described how often variants occurred independently in that database. Finally, it is mentioned how many names matched when the name variant pair was extracted. Table 1 shows an example of three rows in the female name variant pair list.

Table 1: An example slice of the female name variant pair data by Bloothooft and Schraagen (2015). The ID column was omitted.

Variant 1	Variant 2	Sex	Pairs	Independently In Pairs		Independently In *WieWasWie*		Names
				Variant 1	Variant 2	Variant 1	Variant 2	
HENDRIKAE	HENDRIKA	V	1	1	8948	1	453890	4
TETTJE	TEDJE	V	1	473	6	6896	26	4
JAANTJE	JANTJE	V	103	884	4627	9609	304638	7
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

The data was made bidirectional, meaning every entry occurs twice, but once with the name variants' order reversed. This was done to simplify manual look-up in the files. The bidirectional trait was left in on purpose, because it provides some code efficiency later. The list was manually split in male and female first names and surnames. Separation simplifies creation of cost models per type of name. The total amount of name variant pairs and their numbers per type of name is shown in Table 2. The count is separated

in types and tokens to give an idea of how many different kinds of pairs were found how often.

Table 2: Amounts of name variant pairs (types) and their cumulative amount of occurrences (tokens).

Male First Names		Female First Names		Surnames	
Types	Tokens	Types	Tokens	Types	Tokens
51,188	418,706	83,032	653,202	328,411	913,793

* The occurrence of an uneven amount of pairs in the surnames file might raise suspicion. A scan revealed this is caused by the name variant pair *Slöetjes/Slöetjes*, which is not a pair of variants to begin with, but exactly equal names. It is unsure how this entry made it into this list, as it is the only variant with equal names in it. The entry is left in, because the equal names mean no operations will be extracted anyway.

3 Methods

This section describes the methods to extract operation data from the datasets of name variant pairs. The operation data that is extracted contains the operation's type (insertion, deletion, substitution or transposition), characters that are altered and the context that the operation was done in. The context is the combination of the character on the left side of the operation and the character on the right side of the operation. For example, the context for '*J*' deletion in the variant *Rietje* is done in the context (*T,E*). Storing the context in which an operation is done allows for very specific querying of the amount of operations that occur in a specific context, or how often a specific operation occurs in some context. Most edit distances do not consider contextual information during calculation, but for name variants.

The data is cleaned beforehand, described in Sect. 3.1, which describes how the name variant pairs are filtered of punctuation, digits and diacritics. Next, the edit distance described in Sect. 3.2 is used to generate a calculation table between the variants of all the name variant pairs. The paths are then extracted from the calculation table using path tracing, which is described in Sect. 3.3. The extracted paths, however, contain subsequent operations that lose their context because they are not considered in unison with one another. For example, during the insertion of '*T*' '*J*' and '*E*' at the end of a name, the '*J*' and '*E*' insertions do not necessarily know that a '*T*' is ahead of them in the path. These operations need to be considered in context of one another to signify their impact on the transformation of name variants. This loss of context is solved through the combination of operations in a path, or path collapsing, as described in Sect. 3.4. The combined operations in the collapsed paths can then be used as a basis for future operation cost.

3.1 Data Cleaning and Preparation

The data is cleaned with the purpose to reduce the alphabet of characters the operations are trained on. Note beforehand that this is not essential, but is done because it is assumed that it provides a clearer prototype of the weighted edit distance. The cleaning methods mentioned here can be safely skipped without hindering the workings of the rest of the methods.

The first cleaning process is normalising the diacritics found in name variants. For example, normalising the diacritics name *Joänna* would strip the umlaut away, resulting in the name *Joanna*. Diacritics are usually beneficial to consider. However, as Table 3 shows, it does not reduce the dataset too drastically. The occurrences of diacritics that did exist were usually replaceable through substitutions that removed diacritics. Most of these substitutions were unique, especially when separated by context.

Table 3: Diacritics normalisation results. Shows how many name variant pairs were normalised, how many became equal as a result and how many were finally loaded.

Male First Name Pairs			**Female First Name Pairs**			**Surname Pairs**		
Normalised	Equal	Loaded	Normalised	Equal	Loaded	Normalised	Equal	Loaded
1,488	670	50,426	1,222	560	82,372	14,031	8,685	324,497

Table 4: Results of the removal of pairs containing punctuation and digits.

Male First Name Pairs		**Female First Name Pairs**		**Surname Pairs**	
Total Available	Filtered	Total Available	Filtered	Total Available	Filtered
50,426	762	82,372	660	324,497	3,914

The second cleaning process is removing the name variants that contain punctuation and digits. Removal was preferred, considering how this cuts down on more singleton operation occurrences later. The argument can be made that the weighted edit distance needs this data to learn how to deal with typos by itself. An analysis of the lost name variants because of this punctuation filtering (Table 4) shows that name variant pairs containing punctuation and digits make up around $\pm 1.5\%$ of the data.

Sometimes the pair ends up having two equal name variants after applying the cleaning processes. Operations are directly created because of differences between variants. There are no differences between the variants in this case, so these pairs end up not creating any operations for training the weighted edit distance. Pairs like these can and will be safely ignored. However, it should be kept in mind that the new pairs still contribute to the total amount of pairs and occurrences of variants and such. In this proposal of the weighted edit distance, the only value used for cost calculation is the amount of independent occurrences of a name variant pair, and remains unused in the case of a similar name variant pair.

It is important to note again that the cleaning processes described here are not essential. If the cleaning processes were to be skipped, it would not change the workings of the prototype. Rather, it would introduce some more nuance towards certain operations, since the singleton operation occurrences would then be available to weigh costs against. Typo's would probably be handled better, as well as languages that use a lot of diacritics by default, such as French or Spanish.

3.2 Edit Distance Measure

For the purpose of creating a list of operations seen between all name variants, an edit distance is calculated between the variants of every name variant pair. An edit distance measure calculates the minimum cost to transform a source word into a target word (Levenshtein, 1966). A transformation is made using predefined operations that have a certain cost. The possible operations can differ between edit distance definitions.

The usual edit distance measure used for calculating the similarity between words is called the 'Levenshtein distance' (Levenshtein, 1966; Wagner and Fischer, 1974). The operations that are available to this definition of the edit distance are *insertion*, *deletion* and *substitution*. Note that substitution is simply an insertion followed by a deletion or vice versa. The costs of all operations, however, is usually chosen to be 1. Then, a swap is cheaper to execute than a subsequent insertion and deletion are. Fig. 1 in the introduction shows an example of how the edit distance between the names *Willemine* and *Wilmina* can be derived.

The implementation of the Levenshtein edit creates a 'calculation table'. An example of an initiated and finished calculation table can be found in Fig. 5. Note that the black area around the table is metadata, indicating what characters in the source and target word each cell refers to.

This table is first padded with some initial values in the top-most row and the left-most

		J	A	N
	0	1	2	3
J	1			
O	2			
H	3			
N	4			

		J	A	N
	0	1	2	3
J	1	0	1	2
O	2	1	1	2
H	3	2	2	2
N	4	3	3	2

Figure 5: A Levenshtein distance calculation table for the transformation of the name *Jan* into the name *John*.

	s	i
	d	C

i	Insertion
d	Deletion
s	Substitution

Figure 6: Cells that are used to calculate the value of the cell $\mathcal{C}$ when using the Levenshtein distance.

column, starting with 0 in the top left of the table and adding 1 for every step taken in this row and column. Assume $\mathcal{C}$ is the current cell being calculated. Then, for every cell $\mathcal{C}$ starting in the top left of the blank space, the cells left, above and diagonally left-above $\mathcal{C}$ are considered. To calculate the new value for $\mathcal{C}$, the values in the considered cells are added by their operations costs. If $\mathcal{C}$ corresponds to a matching character in the source and target words, the value of the cell left-above $\mathcal{C}$ will simply be copied. Otherwise, if $\mathcal{C}$ does not correspond to matching characters, the following three values will be calculated:

- For the cell above $\mathcal{C}$, the insertion cost is added;
- For the cell left of $\mathcal{C}$, the deletion cost is added;
- For the cell left-above $\mathcal{C}$, the substitution cost is added.

Fig. 6 shows a visual representation of some arbitrary $\mathcal{C}$ and the cells the different operation costs are derived from.

The three new costs resulting from the additions are weighed against each other. The minimum value amongst the operations is the new value for $\mathcal{C}$. This process is repeated until all blank spaces are filled, revealing the minimum edit distance in the bottom right corner of the calculation table.

The Levenshtein distance does a decent job in identifying similarity between two words. However, the name variants that are dealt with here often contain simple typing errors that could sometimes be fixed by simply reversing two adjacent characters. An edit distance measure that combines the Levenshtein distance with an operation that performs adjacent character *transposition* is called the 'optimal string alignment distance' (Wagner and Lowrance, 1975; Kruskal, 1983).

Unlike the other operations, transposition works by considering not one, but two characters. Because at least two characters need to be considered in for the transposition

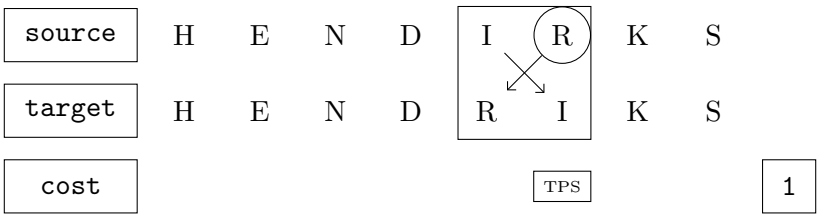

Figure 7: Example of a succeeding transposition check. Produces an *IR*-transposition.

operation, it starts being considered after the second character is reached in both the source and target. A check is done for every cell. This check matches the current character in the source word to the previous character in the target word, and the previous character in the source word to the current character in the target word. An example of the check is illustrated in Fig. 7.

If the transposition check passes, the operation cost for the transposition is added to the value in the cell two steps diagonally left-above the current one, $\mathcal{C}$, as shown in Fig. 8. Transposition is included when calculating the minimum cost. With that, transposition can then be considered to define the value of $\mathcal{C}$, alongside insertion, deletion and substitution.

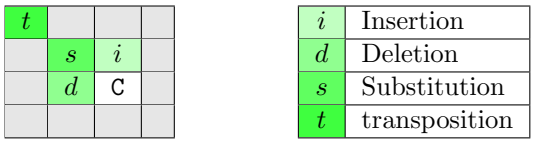

Figure 8: Cells that are used to calculate the value of the cell $\mathcal{C}$ when using the optimal string alignment distance.

Cost for transposition is also assumed to be defaulted to 1, like the rest of the operations. The calculation tables in Fig. 9 show how transposition reduces the cost of transforming *Marai* into *Maria*, by comparing the Levenshtein distance and the optimal string alignment distance calculation matrices. When applied to the name variants, the source word is the first variant, and the target word is the second variant.

		M	A	R	A	I
	0	1	2	3	4	5
M	1	0	1	2	3	4
A	2	1	0	1	2	3
R	3	2	1	0	1	2
I	4	3	2	1	1	1
A	5	4	3	2	1	2

		M	A	R	A	I
	0	1	2	3	4	5
M	1	0	1	2	3	4
A	2	1	0	1	2	3
R	3	2	1	0	1	2
I	4	3	2	1	1	1
A	5	4	3	2	1	1

Figure 9: The calculation tables for the same source and target words, *Marai* and *Maria*. The left table was filled using the Levenshtein distance, the right using the optimal string alignment distance.

3.3 Path Tracing

The operations and their contexts are used to calculate the operation costs for the weighted edit distance. The context in this case is derived from the source word and only has a length of 1 on either side. When an operation is performed at the start or end of the source word, it is not possible to derive context at the left or right side. In such cases, the symbol `#` will be used, which signifies that the operation took place at the start or end of a name. The calculation tables that are produced when an edit distance is calculated, can be backtracked through to collect the sequences of operations, or 'paths', that can be taken. Fig. 9 highlights the paths taken to calculate the edit distance between *Marai* and *Maria*. Note how the gap in the second table's path trace indicates transposition.

Different paths can result in the same edit distance, which can also be seen in the left calculation table in Fig. 9. Note that the cells relevant to the paths are overlapping in the figure. The table was generated using the Levenshtein distance and therefore lacks transposition. The lack of a transposition operation means that *Marai* can only be transformed into *Maria* in two ways. Either (1) removing the '*a*' before the '*i*' and then inserting an

		I	E	P	K	J	E
	0	1	2	3	4	5	6
Y	1	1	2	3	4	5	6
P	2	2	2	2	3	4	5
K	3	3	3	3	2	3	4
J	4	4	4	4	3	2	3
E	5	5	4	5	4	3	2

Path 1

Operation	Characters	Context
Deletion	I	#_E
Substitution	$E \rightarrow Y$	I_P
Match	P	E_K
Match	K	P_J
Match	J	K_E
Match	E	J_#

Path 2

Operation	Characters	Context
Substitution	$I \rightarrow Y$	#_E
Deletion	E	I_P
Match	P	E_K
Match	K	P_J
Match	J	K_E
Match	E	J_#

Figure 10: The calculation table for the edit distance between *Ypkje* and *Iepkje* and the two paths of operations extracted from this table. The only difference between these paths is the order of deletion and substitution.

'a' after the 'i', or (2) inserting an 'i' before the 'a' and then removing the 'i' after the 'a'.

No decision can be made about which of the paths that can be taken to transform names into their variants is the best, as they all result in the same similarity value. This means all of the paths are equally valuable.

A path tracer algorithm is created for the purpose of collecting all possible paths from a calculation table. The operations that can occur in the produced paths are insertion, deletion, substitution and transposition, as well as a *match* operation that was created, which signifies that 'nothing happened' in the paths for a certain letter. The match operations will not be included when training from the operation data, as they do not alter the source string in any way. However, match operations imply that a free move was made in the path of a calculation table. This could be valuable, therefore the match operations are also stored.

The operations are respresented as a data structure that allows storage of the operation's type, its context and the characters that the operation alters.

The algorithm works as follows. Assume $\mathcal{P}$ is the path currently being traced. The current cell $\mathcal{C}$ will be considered. Tracing starts at the most bottom-right cell, where the edit distance value is stored, hence C is initialised as the most bottom-right cell. Application of this algorithm on the calculation table for the name variant pair *Iepkje*/*Ypkje* is illustrated in Fig. 10.

The starting point of the tracing loop is the consideration if $\mathcal{C}$ is the most top-left cell in the table, as no more steps can be taken in that case. The source and target characters that $\mathcal{C}$ relates to will be derived via a simple index lookup. If these characters are the same, a match operation is inserted at the front of $\mathcal{P}$. If no match operation can be derived, the prior knowledge of a cell's origins (Fig. 8) is used to derive the cells that $\mathcal{C}$ could have originated from, or 'origin cells'. An operation is valid if $\mathcal{C}$ minus the operation's cost matches the operation's origin cell value. Multiple operations can be valid here. For every valid operation, the context is collected through another character lookup. If the character lookup falls outside the range of the name variant, the symbol '#' is used. The symbol '#' indicates the start/end of a word in context and must be a character that is not found in any the the name variants to ensure its validity. If only one operation was valid, it is inserted at the front of $\mathcal{P}$. Otherwise, if multiple operations were valid, Finally, $\mathcal{C}$ is set to the newly inserted operation's origin cell, after which the tracing loop continues

		P	I	E	T	E	R
	0	1	2	3	4	5	6
P	1	0	1	2	3	4	5
I	2	1	0	1	2	3	4
E	3	2	1	0	1	2	3
T	4	3	2	1	0	1	2
E	5	4	3	2	1	0	1
R	6	5	4	3	2	1	0
T	7	6	5	4	3	2	1
J	8	7	6	5	4	3	2
E	9	8	7	6	5	4	3

Path

Operation	Characters	Context
Match	*P*	*#_I*
Match	*I*	*P_E*
Match	*E*	*I_T*
Match	*T*	*E_E*
Match	*E*	*T_R*
Match	*R*	*E_#*
Insert	*T*	*R_#*
Insert	*J*	*R_#*
Insert	*E*	*R_#*

Figure 11: Optimal string alignment distance for *Pieter* and *Pietertje*. The context of the three insertions is the same; the operations do not take the other operations into account.

from the starting point.

If $\mathcal{P}$ is completed, it is stored as such. If there are any unfinished paths left, one of them will be taken out and $\mathcal{P}$ will be reinitialised as this unfinished path, after which the tracing loop will continue from the starting point. If there are no unfinished paths left, the algorithm will terminate.

The result of this algorithm is a list of minimum cost paths, where each path describes a sequence of operations with context. Applying the operations of one of these paths in order would transform the same source word into the same target.

3.4 Path Collapsing

Context generated on a per-operation basis has limited meaning in the case of subsequent operations. For example, consider Fig. 11 where *Pieter* transformed into *Pietertje*. The first of the three insertions is contextually correct. However, the last two insertions have context that is not correct at the point where the operation would be executed. This is caused by the path tracing not considering operations that have already been done to the source string.

This problem can be remedied by combining the three subsequent insertions into one insertion of '*TJE*' in (*R,#*) context. A path collapsing algorithm is created that combines all the subsequent operations in a path, excluding match, into a single operation.

The path collapsing algorithm works as follows. Every type of operation must be combinable with every other type of operation. Naturally, another operation type results from the combination of two operations. A definition is made that decides which operation type results from the combination of every subsequent pair of two operations. The definition is shown in Table 5. Note that in most cases this results in substitution.

The match operation is not included in the collapsing data, since it does not affect the

Table 5: The type of operation that is created by combining two operation types.

	Insertion	**Deletion**	**Subtitution**	**Transposition**
Insertion	Insertion	-	Substitution	Substitution
Deletion	-	Deletion	Substitution	Substitution
Substitution	Substitution	Substitution	Substitution	Substitution
Transposition	Substitution	Substitution	Substitution	Substitution

Default Path		
Operation	Characters	Context
Match	*P*	*#_I*
Match	*I*	*P_E*
Match	*E*	*I_T*
Match	*T*	*E_E*
Match	*E*	*T_R*
Match	*R*	*E_#*
Insert	*T*	*R_#*
Insert	*J*	*R_#*
Insert	*E*	*R_#*

Collapsed Path		
Operation	Characters	Context
Match	*P*	*#_I*
Match	*I*	*P_E*
Match	*E*	*I_T*
Match	*T*	*E_E*
Match	*E*	*T_R*
Match	*R*	*E_#*
Insert	*TJE*	*R_#*

Figure 12: The default and collapsed variants of the path shown in Fig. 11.

source string in any way. The path collapsing algorithm iterates through some path $\mathcal{P}$, starting at the first operation in the path. While iterating through a path, the current operation $\mathcal{X}$ and the subsequent operation $\mathcal{Y}$ are considered. Then, one of the following scenarios occurs:

- $\mathcal{X}$ is a match operation. In this case, $\mathcal{X}$ will immediately be reassigned as $\mathcal{Y}$ and $\mathcal{Y}$ will be reassigned as its successor in the path.
- $\mathcal{X}$ is a non-match operation and $\mathcal{Y}$ is a match operation. In this case, $\mathcal{X}$ will immediately be reassigned as $\mathcal{Y}$ and $\mathcal{Y}$ will be reassigned as its successor in the path.
- $\mathcal{X}$ is a non-match operation and $\mathcal{Y}$ is a non-match operation. In this case, and only this case, combination will occur. The right side context of $\mathcal{X}$ is replaced by the right side context of $\mathcal{Y}$. The characters that the operations apply to will be combined. $\mathcal{X}$ will now be the combined operation. $\mathcal{Y}$ will be reassigned as its successor in the path. This way, subsequent operations can be iteratively collapsed into a single operation.

The loop repeats itself, considering two operations at a time, until $\mathcal{X}$ is the last operation in the path, at which point no more collapsing can be done. An example of the algorithm's work is shown in Fig. 12.

4 Results

This section describes the results gathered through applying edit distance calculation, path extraction and path collapsing on the name variant pairs in the different datasets (male first name, female first name and surname variants).

For all the name variants pairs in a dataset, two counts are made: an operation count and a substring count. The operation count stores the total number of occurrences of operations per altered character(s), per context. The substring count stores substrings with various lengths of name variants, which serves as a way to see how often a certain combination of characters occurs in the dataset.

The counts can then be used in unison to create a probability value for any given operation occurring in some specific context.

4.1 Operation Count

The operation count shows how often operations occur in the dataset. Every name variant pair has multiple occurrences (Table 1, 'Pairs'). Multiple paths can also be extracted per name variant. This introduces the problem that operations that occur in only a few of the paths would be counted equally as much as the operations that occur in all of the paths.

Table 6: Numbers of different types of operations when separated by altered characters and context. The numbers of tokens have been rounded to integers.

	Male First Names		**Female First Names**		**Surnames**	
	Types	Tokens	Types	Tokens	Types	Tokens
Insertions	3,200	123,137	3,845	180,160	7,107	253,912
Deletions	3,200	123,137	3,845	180,160	7,107	253,912
Substitutions	14,128	232,413	21,560	428,055	52,469	442,992
Transpositions	634	2,831	768	4,580	1,850	6,399

For example, assume two name variant pairs p and q. Say p produced four paths, and q produced one path. If an operation occurs in two of the paths in p, but also in the single path of q, then the operation in p only has half the significance compared to the one in q. To remedy this, a normalisation is applied that scales an operation's value based on its occurrences in the paths produced by the relevant name variant, the formula of which can be seen in Eq. 1. For the previously given example of the name variant pairs p and q, this would result in a value of 0.5 for the operation in p and a value of 1 for the same operation in q.

$$\text{value}(\textit{operation}) = \frac{\text{number of paths with } \textit{operation} \text{ occurrences}}{\text{total number of paths}} \tag{1}$$

The operation count will separate operations by characters affected and context, allowing for very specific lookups to be made that target a certain operation on certain characters in a certain context. Applying the normalisation formula to all the operations produced by all the name variant pairs gives us a count that has fractional values. Lookups in the count will result in a value that refers to how relatively often an operation was used in the complete dataset. Table 6 shows the results of these counts for the three different types of name variants. Note that the number of substitutions is substantially larger than the other numbers of operations, which is caused by path collapsing.

4.2 Substring Count

The substring count shows how often a substring occurs amongst all the name variants in the dataset. Ultimately, this count serves as a frame of reference, which operations can be weighed against to get some perspective of an operation's significance in the dataset.

The dataset is bidirectional, so only one name variant per pair is analysed for this purpose. Which of the two name variants per pair is analysed does not matter, as long as the choice for first or second variant remains the same for all the name variant pairs. The number of times a substring is counted is the number of times the name variant pair it was extracted from occurs.

The count might be queried for context-related information, so all name variants are surrounded by the name start/end symbol '`#`' again, to cover for these cases. The name variants are then iterated through and split in chunks of lengths 2 to n. The variable n is based on the length of the largest character alteration done by the operations in the

Table 7: Substrings that are generated for the name variant name variant *Rik*, assuming that the largest altered character length in operations is at least 5.

2	3	4	5
#R	#RI	#RIK	#RIK#
RI	RIK	RIK#	
IK	IK#		
K#			

Table 8: Largest substring size per name variant type, as well as one the operations and name variants that caused this.

Male First Names	Female First Names	Surnames
Largest substring size:		
13	12	9
Example Operation:		
Substitution BARTHOLOMEE→MEEUWIS Context: #_#	Insertion/Deletion NCATHARINA Context: A_#	Substitution DELAHAY→HAIJ Context: #_E
Name variant pair:		
Bartholomee/Meeuwis	Antoinettancatharina/Antoinetta	Delahaye/Haije
Largest substring produced:		
#BARTHOLOMEE#	ANCATHARINA#	#DELAHAYE

operation count, including the size of the context (plus two, in this case). This way, every operation including its context can be queried for in the substring count. An example of the substrings extracted from a name variant can be found in Table 7.

Table 8 shows the biggest character operations and the largest substring they generated. The substrings shown in Table 8 have some interesting aspects. The female first name variant *Antoinettancatharina* seems like an accidental combination of the names *Antoinetta* and *Catharina*. The operation simply removes the latter name plus the leftover '*n*'. The male first name *Bartholomee* is substituted entirely for its name variant *Meeuwis*. Consider the name variant pair's calculation table in Fig. 13. Any combination of 4 characters in *Bartholomee* can be removed while the other 7 are substituted, all resulting in the same cost.

		B	A	R	T	H	O	L	O	M	E	E
	0	1	2	3	4	5	6	7	8	9	10	11
M	1	1	2	3	4	5	6	7	8	8	9	10
E	2	2	2	3	4	5	6	7	8	9	8	9
E	3	3	3	3	4	5	6	7	8	9	9	8
U	4	4	4	4	4	5	6	7	8	9	10	9
W	5	5	5	5	5	5	6	7	8	9	10	10
I	6	6	6	6	6	6	6	7	8	9	10	11
S	7	7	7	7	7	7	7	7	8	9	10	11

Figure 13: Calculation table for the optimal string alignment distance between *Bartholomee* and *Meeuwis*. There are a lot of paths available to travel through this table.

4.3 Using Counts to Calculate Operation Probabilities

Table 9 shows some notable operations for female first name variants and surname variants, per type of operation.

The 'Fraction' column in this table is an estimation of how likely the operation is to occur in its given context. For example, the first operation is '*N*' insertion in (*E,#*) context. '*N*' insertion in (*E,#*) context occurs ±25,000 times and the fraction is only 10.5%. The same exact operation scores almost twice as high for surnames (20.7%), suggesting that inserting '*N*' behind '*E*' at the end of a surname has a higher chance of resulting in a name variant than when the insertion is done at the end of a female first name.

To calculate the fraction value for an operation two queries are made. First, the operation count for the number of occurrences of this operation their given context. Second,

Table 9: Some notable operations per type of operation, for female first names and surnames. Deletions were omitted, because they are the inverse of insertions and therefore have the same results.

Character(s)	Context	Occurrences	Fraction	Example
Female First Names - Insertions				
N	(*E,#*)	24,959	0.105	*Aaltje – Aaltjen*
E	(*I,N*)	5,774	0.061	*Rina – Riena*
H	(*O,A*)	5,364	0.643	*Joanna – Johanna*
OH	(*J,A*)	3,039	0.091	*Janna – Johanna*
DA	(*I,#*)	1,493	0.232	*Ali – Alida*
HA	(*T,R*)	1,376	0.028	*Catrina – Catharina*
Female First Names - Substitutions				
S→Z	(*I,A*)	15,085	0.591	*Lisa – Liza*
A→E	(*N,#*)	14,557	0.120	*Anne – Anna*
J→I	(*T,E*)	5,373	0.048	*Antje – Antie*
A→TJE	(*N,#*)	4,348	0.036	*Anna – Antje*
LE→HEL	(*L,M*)	2,581	0.206	*Willemien – Wilhelmien*
ET→IG	(*N,J*)	2,126	0.370	*Annetje – Annigje*
Female First Names - Transpositions				
IE	(*R,N*)	133	0.038	*Rientje – Reintje*
RE	(*D,K*)	75	0.325	*Henderkien – Hendrekien*
AN	(*I,#*)	70	0.470	*Christian – Christina*
Surnames - Insertions				
N	(*E,#*)	15,417	0.207	*Linde – Linden*
S	(*R,#*)	8,127	0.096	*Kuiper – Kuipers*
J	(*I,N*)	7,401	0.081	*Bruin – Bruijn*
EN	(*S,#*)	2,158	0.012	*Jans – Jansen*
CH	(*S,#*)	1,666	0.009	*Bos – Bosch*
DAL	(*N,#*)	11	0.000	*Bloemen – Bloemendal*
Surnames - Substitutions				
R→N	(*E,#*)	3,887	0.053	*Kortlever – Kortleven*
T→D	(*R,#*)	3,488	0.245	*Weert – Weerd*
G→K	(*N,#*)	2,644	0.131	*Wenting – Wentink*
G→CH	(*E,T*)	823	0.464	*Knegt – Knecht*
Z→JS	(*I,E*)	616	0.082	*Keizer – Keijser*
RITS→SEN	(*R,#*)	2	0.005	*Gerrits – Gersen*
Surnames - Transpositions				
RU	(*B,G*)	230	0.107	*Verbrug – Verburg*
TS	(*R,#*)	83	0.012	*Voorts – Voorst*
IE	(*R,N*)	75	0.048	*Rienders – Reinders*

the substring count for the number of seen substrings of the operation in its surrounding context before altering the characters. The operation count was extracted from exactly the same dataset as the substring count. This means that every possible operation in a certain context is also represented in the substrings, with at least as much occurrences. This assures a fractional value between 0 and 1.

As an example for calculating the fraction value, consider inserting '*TJE*' at the end of the name *Trijn*, creating its variant *Trijntje*.

Two lookups are made. The operation count returns that for '*TJE*' insertion in (*N,#*) context, ±123 operations were counted for female first name variants. The substring lookup checks for the number of occurrence of the characters before the operation was made. In the case of insertion, this is just the context, as this is where the new characters will be inserted. So, for the example, the occurrences of the substring '*N#*' are looked up, which returned 54,478 occurrences.

So 54,478 female first name variants ended in '*N*' and in approximately 123 of these cases an insertion of '*TJE*' was made, resulting in a fraction of 0.0023, or 0.23%.

5 Implementation Suggestions

In this section, suggestions are given for two different cost model implementations. Using this cost model, it is then described how this model could be applied in edit distance calculation, to create the weighted edit distance. Finally, it will be described how the edit distance can be tested.

5.1 Cost Model

When deciding operation costs, different approaches can be taken. Two implementations of cost models are suggested in this section. The first implementation is very binary, because it simply assigns operation costs based on whether or not an operation was seen during the operation count. The second implementation uses both the operation count and the substring count to calculate a possibility value for an operation, which is then used to decide the operation's cost.

5.1.1 Binary Cost Model

For any particular name variant type, the occurrence of an operation in the operation count signifies that this operation transforms a known name variant into another known name variant. This means that operations that are not seen in the operation count do not necessarily transform name variants into other name variants.

A straightforward cost model that can be made simply checks for occurrence of an operation in the operation count. This model assign low costs to operations that are found in the operation count and high costs to operations that are not. As the actual value of a cost is is usually chosen to be 1, the most similar cost assignment would be to assign 0 for operations that are counted and 1 for operations that are not.

5.1.2 Possibility-based Cost Model

A more sophisticated approach to assign a cost to an operation would be to calculate some value that indicates how likely an operation is to be made. In Sect. 4.3 it was explained how such a value could be calculated by combining the results of lookups in the operation and substring counts. A threshold then decides if an operation's possibility to occur in its suggested context is high enough. If so, the operation will be assigned a cost of 0, otherwise, a cost of 1 will be assigned.

Cost assignment here is still very binary. It is possible to create some sort of step function that smooths cost assignment. For example, if the threshold was put at 0.2, a fraction of 0.15 will go under the threshold, meaning its operation did not occur enough in its given context. Because the fraction range for high costs will be [0, 0.2] in this case, the cost would be a 1. Assigning a cost of 0.5 to the fraction range of [0.1, 0.2], however, would punish the operation a little bit for not occurring quite as often as requested. Introducing something like this would of course increase the complexity of this implementation, which might not be beneficial.

5.2 Weighted Edit Distance

The end goal of this paper was to define an edit distance measure that can detect the similarity between unknown name variants by learning from the operations used to transform known name variants. Now that it is possible to create a cost model that does precisely this, a weighted edit distance implementation can be suggested.

The weighted edit distance would be similar to the optimal string alignment distance. The operations that are supported – insertion, deletion, substitution and transposition –

would remain the same. The cost calculation for the operation is where the optimal string alignment distance and the weighted edit distance would start to differ.

The cost calculation can be approached in two ways, because there are multiple ways to consider subsequent operations during calculation of the edit distance. During a standard calculation of an edit distance, usage of operations is only decided through the values in the calculation table. The cheapest operations are then chosen to calculate the next value with, after which the operation itself is forgotten about. The weighted edit distance should keep subsequent operations and their context in mind as its cost model holds information based on those operations.

An approach to considering subsequent operations is to store operations previously seen and retroactively calculate the overall cost of the operations together. For example, consider the case where a name variant similarity calculation was done and the characters '*TJE*' would have to be inserted to transform a name into its variant. When the final '*E*' is considered for insertion, the weighted edit distance should be able to recall the two past insertions done and recognise it as a '*TJE*' insertion at the end. This approach would then allow cost calculation to rectify itself after a combination of operations is encountered. The cost of the '*TJE*' insertion would then be added to the value found in the calculation table on the location before the '*TJE*' was inserted. This is similar to how transposition can bridge the substitution operation in a calculation table.

5.3 Testing

It is, of course, important to know how well a weighted edit distance evaluates name variant similarity as opposed to regular edit distance measures. To test how well an edit distance evaluates the similarity between yet unknown name variants, a dataset of name variants reviewed by experts as valid is required.

The dataset we used for training contained valid name variants (Bloothooft and Schraagen, 2015). However, not every link between every name variant in a cluster of name variants was found. For example, the variant pairs *Mueller/Muller* and *Mulder/Muller* both exist, but the pair *Mueller/Mulder* does not. This means that there are yet unknown name variant pairs already existent in our original dataset, which could be utilised to test a prototype of the weighted edit distance.

Unknown pairs of which the name variants are in the same cluster should be classified as similar. Unknown name variant pairs of which the variants lie in different clusters should be classified as not similar.

The amount of pairs correctly classified can then be used as a measure to define which edit distance, or the model that assigns its costs, is better than another.

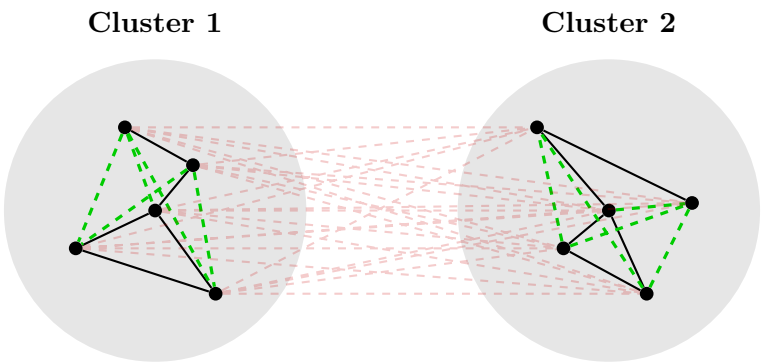

Figure 14: Two clusters of name variants. Solid links indicate an original name variant pair and dashed links are pairs that might be evaluated during testing. Green indicates similarity, while red indicates non-similarity.

Fig. 14 shows how potential links between two example clusters of name variants should be evaluated as similar or not similar, depending on if the link is between name variants of the same cluster or not.

6 Discussion

For its specific domain, the weighted edit distance will likely provide better nuance in spelling variation detection, and with that name variant identification, than regular edit distances do. The operation count suggests that predictable patterns can already be found often by extracting and combining operations from name variant pairs, which seems promising. So, ultimately, the weighed edit distance could most likely be successfully used to circumvent the limitation of the static threshold used for name variant identification.

6.1 Similar Work

Other efforts to identify name variants have also been made. An example is the approach that turned names into their base form, to then compare these, instead of trying to find a way to detect the variation via a similarity measure (Kemenade, 2016). Other research might reveal that variation in names contains more features than is currently assumed, in which case a probabilistic learning model-approach might prove more successful.

The solution for calculating a similarity measure tyhat was proposed in this paper might be applied to any domain with a similar threshold issue. However, alterations might be required, as not every similarity problem has need for a transposition operation, nor the given type of cost model or the described kind of utilisation of this cost model. Most of the components of the weighted edit distance can be changed without hindering the functionality of the approach as a whole, so ultimately there is a lot of flexibility when adapting the weighted edit distance for other purposes.

6.2 Limitations and Future Work

When an operation is never seen before, the suggested implementation of the weighted edit distance will assign a high cost to this operation. As it's counted in the same manner, this is true for contexts too. This raises the issue that, if an operation happens hundreds of times in all kinds of contexts, it might be considered a good operation for the weighted edit distance's purpose. However, if one context happens to have never been encountered, the weighted edit distance would immediately assume it is unlikely, and assign a high cost. If further research reveals that this is a problem, a possible solution for this would be to generalise the way in which operation characters and context are looked up in the operation count. Instead of looking for an exact match of the operation's type, the altered characters and the context, the characters and context could allow regular expression-based lookup support. For example, a deletion of the characters described by "`[0-9]*`", which captures all digits, could then be given as the lookup value, which would somehow combine all the results of 'digit deletion' in general.

The way context of operations is considered can likely be altered. Currently the context uses only 1 character on each side of the operation, which might turn out not to be enough to reliably confirm possible name variants. Names are short, meaning consideration of one character wide context is at least likely to be significant, as there is not much else that can be said about the operations, in a way. However, it might be valuable to consider context on a wider scale.

Language is something that should be taken into account too. Every observation made here is based on Dutch names, and therefore, Dutch naming habit through the years. Other languages might use more diacritics in characters, or even a completely different type of

alphabet, like how Asian languages usually use symbols and combinations of those symbols. In general terms, the weighted edit distance approach should work, as long as there are character-based patterns that can be identified and extracted, however, adaptations might be required to allow functionality for names in other languages.

7 Conclusion

The goal of the weighted edit distance was to overcome the static threshold that limits default similarity measures in current efforts in record linkage, namely that they are too unreliable to deal with all the variation that occurs in names between records. The variation between names has several origins, mostly different names, spelling variation and spelling mistakes made during digitisation. The proposed way to do this was to create a more dynamic edit distance, that assigns low costs to operations that are often used to transform names into their known name variants. This edit distance would naturally assign high costs to operations that are never seen to transform names into variants.

A series of methods were given to extract operations from a set of pairs of known name variants. First, the optimal string alignment distance between all name variants was calculated. The paths that were taken during the edit distance calculation were then extracted from the calculation table that gets generated during edit distance calculation. Sub operations in these paths were combined into singular operations that altered multiple characters. A collection of operations was then created by counting the occurrences of every operation.

Substrings of different lengths of the name variants were extracted and counted as well, to serve as a lookup table for different combinations of characters.

It was then described how the collection of operations and substrings could be used to create various types of cost models. These cost models can then assign costs to operations, creating the weighted edit distance. Finally, a test method was provided that tests how well the weighted edit distance would evaluate yet unseen name variants as opposed to other edit distances.

The operation count contained a lot of the common edits made between name variants, and as the count lies at the base of cost decision for the weighted edit distance, the weighted edit distance seems likely to yield positive results. This effect was seen already in the operation count that drives the cost model. However, to be certain about this, actual implementation and testing would be required.

References

Gerrit Bloothooft and Marijn Schraagen. Learning name variants from inexact high-confidence matches. In *Population reconstruction*, pages 61–83. Springer, 2015.

Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710, 1966.

Robert A Wagner and Michael J Fischer. The string-to-string correction problem. *Journal of the ACM (JACM)*, 21(1):168–173, 1974.

Robert A Wagner and Roy Lowrance. An extension of the string-to-string correction problem. *Journal of the ACM (JACM)*, 22(2):177–183, 1975.

Joseph B Kruskal. An overview of sequence comparison: Time warps, string edits, and macromolecules. *SIAM review*, 25(2):201–237, 1983.

J van Kemenade. Training a name-variant model using historical data. B.S. thesis, Universiteit Utrecht, 2016.

Source Code

The Python-based source code for this paper can be found at:
`https://github.com/rajoosterlaken/weighted_edit_distance`