Institutionen för datavetenskap
Department of Computer and Information Science

Master’s thesis

Design and Implementation of a Name Matching Algorithm for Persian Language

by

Leila Momeninasab

LIU-IDA/LITH-EX-A--13/061--SE

2013-10-30
Design and Implementation of a Name Matching Algorithm for Persian Language

by

Leila Momeninasab

LIU-IDA/LITH-EX-A--13/061--SE

2013-10-30

Supervisors: Jalal Maleki, Nima Amirsheshkari
Examiner: Lars Ahrenberg
Abstract

Name matching plays a vital and crucial role in many applications. They are for example used in information retrieval or deduplication systems to do comparisons among names to match them together or to find the names that refer to identical objects, persons, or companies. Since names in each application are subject to variations and errors that are unavoidable in any system and because of the importance of name matching, so far many algorithms have been developed to handle matching of names. These algorithms consider the name variations that may happen because of spelling, pattern or phonetic modifications. However most existing methods were developed for use with the English language and so cover the characteristics of this language. Up to now no specific one has been designed and implemented for the Persian language. The purpose of this thesis is to present a name matching algorithm for Persian. In this project, after consideration of all major algorithms in this area, we selected one of the basic methods for name matching that we then expanded to make it work particularly well for Persian names. This proposed algorithm, called Persian Edit Distance Algorithm or shortly PEDA, was built based on the characteristics of the Persian language and it compares Persian names with each other on three levels: phonetic similarity, character form similarity and keyboard distance, in order to give more accurate results for Persian names. The algorithm gets Persian names as its input and determines their similarity as a percentage in the output. In this thesis three series of experiments have been accomplished in order to evaluate the proposed algorithm. The f-measure average shows a value of 0.86 for the first series and a value of 0.80 for the second series results. The first series of experiments have been repeated with Levenshtein as well, and have 33.9% false negatives on average while PEDA has a false negative average of 6.4%. The third series of experiments shows that PEDA works well for one edit, two edits and three edits with true positive average values of 99%, 81%, and 69% respectively.
Acknowledgements

I would like to express my appreciation to:

- My supervisor, Jalal Maleki, for his valuable support and guidance throughout this work. He was always there when I needed him. His advice and comments to my technical questions, my report by proofreading and my presentation were very precise and useful. I learned a lot not only from his tips and technical views, but also from his friendly, supportive and humble personality.
- Professor Lars Ahrenberg as my examiner who gave me completely new perspectives on my research with his knowledge. Many thanks for giving me this opportunity.
- Dr. Nima Amirshekari who was my best consultant in this research. I am really grateful for his supports and encouragements when being far from university caused me difficulties in focusing and carrying on my thesis.
- David Hall for his valuable and precise comments in proofreading of this thesis. I am really grateful to him.
- All of my friends who supported me and gave me help during my study, especially to Gun Austrin, who was beside me on a difficult period of my life during my study, to Farah who is as my mother and opened a new window to the world to me, to calmness and energy, to dr. Asadpour who I am proud of having him not only as my consultant, but also as my friend and all of my friends in Linköping.
- My family, especially my parents for their constant support. My deepest gratitude to them.
# Contents

Abstract .................................................................................................................................................. 5
Acknowledgements .............................................................................................................................. 6
List of tables ........................................................................................................................................... 9
List of figures ......................................................................................................................................... 10
Chapter 1: Introduction ........................................................................................................................ 11
  Motivation ........................................................................................................................................... 11
  Overview and Contributions .............................................................................................................. 11
  Scope of this thesis ............................................................................................................................. 13
  Thesis assumption ............................................................................................................................. 13
  Evaluation methods ........................................................................................................................... 13
  Outline .............................................................................................................................................. 14
Chapter 2: Literature review ................................................................................................................. 16
  Name variations ................................................................................................................................. 16
  Name matching algorithms ............................................................................................................... 17
    Phonetic matching algorithms ........................................................................................................ 18
    String distance algorithms ........................................................................................................... 19
    Token based algorithms ............................................................................................................... 26
  Persian language ............................................................................................................................... 27
  Persian alphabet ............................................................................................................................... 27
Chapter 3: Method ............................................................................................................................... 29
  A name matching algorithm for Persian language .......................................................................... 29
  Levels of similarity in Persian language .......................................................................................... 30
    Form similarity ............................................................................................................................. 30
    Phonetic similarity ........................................................................................................................ 34
    Keyboard similarity ....................................................................................................................... 37
  Persian Edit Distance Algorithm’s Core ............................................................................................ 39
    Cost of insertion and deletion operations ..................................................................................... 40
    Cost of replacement operation ..................................................................................................... 41
    The similarity of two names ......................................................................................................... 43
Implementation of PEDA ................................................................. 44
Name data sets .............................................................................. 47
Parameters ...................................................................................... 48
Chapter 4: Findings and discussion .................................................. 50
Matching Results ............................................................................ 50
  First series of experiments ............................................................. 51
  Comparison with Levenshtein ......................................................... 55
  Second series of experiments ......................................................... 56
  Third series of experiments ............................................................. 59
Chapter 5: Conclusion and future work ............................................. 60
  Conclusion .................................................................................... 60
  Future Work ................................................................................. 60
Bibliography ..................................................................................... 62
List of tables

Table 1. Predicted matches statuses ................................................................. 13
Table 2. English alphabet encoding table .......................................................... 18
Table 3. Persian alphabet ................................................................................. 28
Table 4. Letters not included in Persian but they can be typed on Persian keyboard ................................................................. 28
Table 5. Form similarity in Persian alphabet (between all position-dependent letter forms) ................................................................. 32
Table 6. Form similarity in Persian alphabet (between origin letter forms) ......................................................................................... 33
Table 7. Phonetic similarity in Persian alphabet (between all position-dependent letter forms) ................................................................. 36
Table 8. Phonetic similarity in Persian alphabet (between origin letter forms) ......................................................................................... 37
Table 9. The performance measures for the first series of experiments ................................................................................................. 54
Table 10. Levenshtein results in first series of experiments ........................................ 55
Table 11. PEDA results in first series of experiments ................................................ 55
Table 12. The performance measures of Levenshtein in the first series of experiments ................................................................. 56
Table 13. The performance measures for the second series of experiments ................................................................................................. 56
Table 15. The true positives and false positives for four executions........................................ 59
Table 16. Some examples PEDA works not enough efficient on them ........................................ 59
List of figures

Figure 1. Overview of implemented PEDA in this thesis .......................................................... 12
Figure 2. A snapshot of the score matrix with filled first row and column ................................... 20
Figure 3. Considering three neighbors to calculate value cell .................................................... 20
Figure 4. Score matrix shows the optimum path and cost ........................................................... 21
Figure 5. A substitution matrix .................................................................................................. 22
Figure 6. The first row and first column initialization of score matrix .......................................... 23
Figure 7. Score matrix ............................................................................................................... 23
Figure 8. The local alignment done by SW .................................................................................. 25
Figure 9. Persian letters on the keyboard in their neutral forms ................................................ 37
Figure 10. Letters and sounds typed when SHIFT key is pressed ................................................ 38
Figure 11. The initial forms of Persian letters on the keyboard axis .............................................. 38
Figure 12. The medial forms of Persian letters on the keyboard layout ....................................... 38
Figure 13. The end forms of Persian letters on the keyboard layout ........................................... 38
Figure 14. The end forms of Persian letters on the keyboard layout when SHIFT key is pressed .... 39
Figure 15. The score matrix for موراد and مراد ................................................................. 43
Figure 16. Separate first names and last names ......................................................................... 44
Figure 17. Not separate first names and last names .................................................................... 45
Figure 19. The threshold divides the output matches into positives and negatives ....................... 50
Figure 20. The bar chart of performance measures for the first series of experiments .................. 55
Figure 21. The bar chart of performance measures for the second series of experiments ............. 57
Chapter 1: Introduction

Motivation

Discovery and matching of names, personal names or company names is used in an increasing number of applications and it constitutes a central part of many applications, for example, in text or web mining, information retrieval, data cleansing, or automatic spell checking and search engines. If only exact matching was available in these types of applications it would not be possible to deal with name variations, which unavoidably occur in the data and names in real world data sets. In order to get more accurate results, an approximate name matching should be applied instead of exact matching. Although name matching is used in many applications, in this thesis the main reason and motivation of providing a name matching algorithm for Persian language, is the increasing demand of name matching from the financial industries, such as the banking industry and especially in E-banking. In inter-branch and inter-bank processes, bank data in different places should be centralized, and in these situations the problem of identifying identical persons arises. Also in fraud detection, customer relationship management, anti-money laundering, credit scoring, unit beneficiary and in many other use cases, where the data is Persian, there is a great need for a name matching algorithm that works well and effective for Persian names. Obviously this algorithm will lead to better data quality and lower costs since less manual labor is required when merging data from different sources.

Overview and Contributions

This thesis makes a contribution to the area of name matching algorithms. These kinds of algorithms take a pair of names as its input and give a score of similarity or dissimilarity of these words as its output. In any natural language names do not have any known, regulated structure. Essentially they are a combination of letters in the language. Thus the name matching algorithms must investigate the similarity between letters or the combination of letters to be able to say how similar two names are to each other. This fact makes them most dependent of the nature of each language and its linguistic properties.

The first goal of this thesis is to design and implement an algorithm that can match names in Persian. Figure 1 illustrates an overview of the implemented algorithm in this thesis:
A phonetic similarity rule connects a pair of letters and shows how much they are similar phonetically. The phonetic similarity value is between 0 for no similarity to 1 for being equal.

A form similarity rule connects two Persian letters and says how much they are similar to each other according to their forms. The form similarity value is between 0 for no similarity to 1 for being equal.

A keyboard similarity rule also connects two letters together based on their proximity on the keyboard. The keyboard similarity value is between 0 for two letters which are far apart on the keyboard and 1 for two letters which are on the same place on the keyboard.

Source list includes all names for which we want to find names identical to them in the watch list. Includes all names which we want to find among them names identical to the names in watch list. Source list and watch list are just two different data sources which we want to match.

Watch list includes the names which we want to find in the source list. Watch list and source list are just two different data sources which we want to match.
• *Matched names list* is the result of matching names in the watch list with the names in the source list. For every matching that the algorithm performs, it returns the distance between those two names and their degree of similarity.

• *PEDA Core* is the core of the algorithm implemented in this thesis.

The second goal is to give statistical measurements of how effective this algorithm is.

**Scope of this thesis**

As a Persian letter may have different forms depending on its position inside a word, the similarity rules between letters should be defined accurately among pairs of position-related letter forms and then used in the algorithm. In order to achieve this, in implementation, for detecting position forms of letters there is a need to a text render work before PEDA. But here because of two reasons we skip this part and define the rules between original forms of letters, the first reason is that getting involved with a text render is out of the scope of this thesis and the other one is that the algorithm gets slower, although it gives more accurate results, using the rules defined of pairs of position-related letter forms.

**Thesis assumption**

In this thesis, the following assumptions are made:

• The default of keyboard layout in this thesis is for Windows-based PCs whit its language set to Persian. But the proposed algorithm has the capability of migration to another keyboard layout.

• The performance measures, recall and precision are of equal weight.

**Evaluation methods**

There are two statuses for a match returned by the algorithm of this thesis. If it is a correct match and its members refer to the same entity in reality, it will be a true match called a *true match or true positive* (TP). If it is not a correct match in reality, it will be a non true match, called a *false match or false positive* (FP). Additionally for a non-match returned by the algorithm there will be two statuses, as well. If its members denote the same entity in real environment, while they are connected together as a non-match in the result, then this non match will not be correct and called a *false non-match or false negative* (FN). If this non-match is a correct non-match in real too, then it will be called a *true Non-Match or true negative* (TN). In table 1 can be seen a short summary of the predicted matches statuses by the algorithm:

<table>
<thead>
<tr>
<th>Actual Matches</th>
<th>Algorithm Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Names match</td>
<td>Matches</td>
</tr>
<tr>
<td></td>
<td>True Positives</td>
</tr>
<tr>
<td></td>
<td>False Negatives</td>
</tr>
<tr>
<td>Names do not match</td>
<td>False Positives</td>
</tr>
<tr>
<td></td>
<td>True Negatives</td>
</tr>
</tbody>
</table>

Table 1. Predicted matches statuses
To assess the match accuracy of the algorithm in this thesis, two statistical measures of performance are applied here, which are calculated based on true positive, false positive, true negative and false negative cases. They are:

- **Precision** (P): is the ratio of true positives to predicted positive cases (Powers, 2011).

\[
Precision = \frac{TP}{TP + FP}
\]

- **Recall** (R): is the proportion of correctly predicted positive cases to real positive cases (Powers, 2011).

\[
Recall = \frac{TP}{TP + FN}
\]

In simple terms, recall shows the rate of real positives by the algorithm. On the other hand precision shows how many of the predicted positive matches that are true and real. In most cases precision and recall are important at the same level, but if a method tries to forecast less false positives to increase the precision, it will return more false negatives as well and recall will decrease. The opposite is the same too, so while these two parameters influence each other, their effects are reverse (Chinchor, 1992; Frakes & Ricardo, 1992).

Additionally, in order to have an overview on performance we use another metric called f-measure that mixes the precision and recall into a single measure (Chinchor, 1992). Recall and precision can have different weights and different level of importance. Thus a general formula for f-measure is (Chinchor, 1992):

\[
f - measure = \frac{(\beta^2 + 1.0) \times P \times R}{\beta^2 \times P + R}
\]

Where \( \beta \) is recall’s weight, that shows its importance relative to the importance of precision. If we consider an equal significance for both recall and precision, \( \beta = 1 \) and f-measure is a harmonic mean of precision and recall as follow (Sasaki, 2007):

\[
f - measure = \frac{2 \times P \times R}{P + R}
\]

Here, f-measure reflects the effect of both recall and precision metrics fairly (Chinchor, 1992). As our test data is not balanced, actually it is better to use the \( \beta \) varied \( f - measure \), but because defining a value for \( \beta \) is not an easy task, we use the basic version of \( f - measure \).

**Outline**

The contents of this thesis are organized as follows.
Chapter 2 consists of an overview on name variations and different types of name matching algorithm. The most known algorithms are described here. In the following there is an introduction to Persian language.

Chapter 3 explains the proposed algorithm of this thesis in details, its different parts and then the implementation is described briefly. Next data sets and parameters are described as well.

Chapter 4 is about the findings and evaluation of the proposed algorithm. It discusses the results.

Chapter 5 gives the conclusion and mentions several suggestions for future work.
Chapter 2: Literature review

This chapter gives an overview of different kinds of existing name matching algorithms and their usage to give the reader a background for the proposed algorithm in this master thesis and gives the basic reasons why the characteristics of the Persian language calls for a specially designed algorithm. It begins with an overview on different variations which may occur in names. It continues with naming the categories of name matching algorithms, and then goes deeper in each category with describing their most important, basic, algorithms and the type of name variations each of them can handle. Finally, we mention the algorithm inspired implementation of PEDA.

Name variations

When dealing with names, the occurrence of variations in identical names is inescapable. There are different types of this natural phenomenon which can be divided into the following groups. The categorization is based on work from articles (Patman & Schaefer, 2006; Branting, 2003; Miller & Arehart, 2008):

1. **Spelling variations**
   Include:
   - **Transcription errors**: come into being when letters are interchanged or misplaced inside the name caused by typographical errors, substituted by the other letters like in case of Smyth and Smith, added to names such as Smythe, or omitted, e.g. Collins and Colins (Lait & Randell, 1998).
   - **Alternative spellings**: occur in circumstances where there are more than one correct spelling for a name, e.g. Jennifer and Jenifer.
   - **Transliteration**: appear when a name is written using an alphabet that differs from the alphabet it is originally written with. For example, the originally Arabic name of “حسين” can be typed in English in these ways Husayn or Husein.
   - **Silent consonants**: arise in cases a name contained some silent consonants, is written without those letters. For example “Coghburn” may be spelled as “Coburn,” or “Deighton” may be recorded as “Dayton” (Patman & Schaefer, 2006).

   On the whole, spelling variations are the ones which happen to the names without destroying their phonetic structures. Nevertheless, their matching cannot carry out only through exact matching methods and needs their own specific matching solutions. Misreading, or mishearing, by either a human or an unmanned device cause the variations of mentioned category (Lait & Randell, 1998).

2. **Fielding variations**
   Different components of names can come into various orders among cultures. For instance the form of “first-middle-last” may be used in one culture while another culture uses “last-
first-middle”. In these circumstances name components may inserted in wrong fields while they are transferred from one database to another one. The name “Mohamed Afzal Aziz” might be mapped to the format of “first-middle-last” with “Mohamed” as the first name, “Afzal” as the middle name, and “Aziz” as the last name while somewhere else it might map to a different format like “first-last” with “Mohamed Afzal” as the first name and “Aziz” in the last-name field (Patman & Schaefer, 2006).

3. Name equivalence
For some people different names can be used to refer to the same. For instance the nickname of one person can use instead of his or her first name. In some culture when people get married, they may change their last name into the last name of their partner. It may possible for a person to change his/her own name during his/her life (Patman & Schaefer, 2006).

4. Short forms
- Initials, e.g. John Smith may be written in a shorter form as J Smith.
- Abbreviations, e.g. Muhammad can be shortened to Mhd.

5. Segmentation
In some languages like Arabic and Persian several names can put together and make a new one. These components can be written separately or together, e.g. Mohamed Amin = Mohamedamin, both forms are correct variants of the same name. In a language like Arabic, the particles and names can also be segmented in writing, e.g. Abd Al Rahman = Abdal Rahman.

6. Translation
Some names may have structural or phonetic variations when introduced to a different language, e.g. Joseph in English is equivalent to the Italian name Giuseppe.

7. Missing or extra elements
First names or surnames composed of more than one component, names or particles, may be written in full or not, e.g. John Charles Smith written as John Smith.

8. Punctuation
In some cases, punctuation may be used to show the separate parts of a name, e.g., “Owens Corning” vs. “Owens-Corning”; “IBM” vs. “I.B.M.”

The different types of variations are described above to give an overview of all possible changes which may happen to the names.

Name matching algorithms

The name matching algorithms are categorized into three main groups depending on the method of matching, i.e. whether they use phonetic similarity, pattern similarity or use names as arrangement of parts to detect names that refer to the identical objects.
Phonetic matching algorithms

These algorithms retrieve and match the names according to their pronunciations. Thus the names which fall in the same group have similar sounds, despite of the differences in their spelling. These kinds of algorithms are a good solution to deal with spelling variations in name matching. Mostly they convert names to codes according to how the names are pronounced, and the best known of these types of algorithms is Soundex, the earliest matching algorithm, developed in the early 20th century as a facility for manual filing of U.S. Census records (Patman & Schaefer, 2006). The later phonetic encoding algorithms, called Phonex, Phonix, NYSIIS, and Double-Metaphone are widely based on Soundex but make some variations into it to improve the output and overcome its limitations.

Soundex

In Soundex (Russel, 1918; Christen, 2006), the letters of English language are classified according to their phonetic similarities, as described for the first time by Russell in 1918, as can be seen in the table 2:

<table>
<thead>
<tr>
<th>Letter</th>
<th>Soundex Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>a, e, i, o, u, y</td>
<td>1</td>
</tr>
<tr>
<td>b, f, p, v</td>
<td>2</td>
</tr>
<tr>
<td>c, g, k, q, s, x, z</td>
<td>3</td>
</tr>
<tr>
<td>d, t</td>
<td>4</td>
</tr>
<tr>
<td>l</td>
<td>5</td>
</tr>
<tr>
<td>m</td>
<td>6</td>
</tr>
<tr>
<td>n</td>
<td>7</td>
</tr>
<tr>
<td>r</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 2. English alphabet encoding table

Here, the name’s first letter and 3 digits put together to generate the corresponding code for each name. The digits are the conversion of the name’s letters except the first one by applying the rules in table 2 (Christen, 2006) in addition some defined rules. For example, Russell (Russel, 1918) says: “In class of 3, the digraph ‘gh’ is not considered representative of the class, as the same is usually silent, as in the name ‘Wright’. Final ‘s’ and ‘z’ are disregarded as the omission or addition of the final sibilant is immaterial in the pronunciation of a name.” Finally, all name codes are compared together and the same codes are grouped together. We expect that all members of each group refer to identical objects.

Limitations / Drawbacks

- **Dependence on initial letter.** Soundex is not capable to find the identical names where the name variation takes place in their first letters. For instance, Soundex will not return “Korbin” and “Corbin” as a match to the user (Patman & Schaefer, 2006).
- **Unranked, unordered outputs.** As Soundex puts the same codes on the same groups, actually in last step it does an exact matching, it would not return a similarity degree of found pairs (Patman & Schaefer, 2006).
Due to phonetic matching algorithms working based on phonetic similarities, they are fit to the transcription and spelling names variations when these variations do not destroy phonetic structure of the names.

**String distance algorithms**

String distance is a non-negative integer that measures the difference (similarity / dissimilarity) between two strings. The string distance algorithms perform an approximate string matching that operate on two string inputs, $s$ and $t$, and calculate their distance. They are designed mainly to handle typographical and spelling errors. The most widely known of these algorithms is called Levenshtein distance (White, 2004) or Edit distance

**Levenshtein distance**

The algorithm is given two words, one as source ($s$) and the other as target ($t$), then by examining all possible ways between the two, this method attempts to find the least cost of converting $s$ to $t$. The Levenshtein distance between two words is defined as the minimum number of edits needed to transform one string into the other. Three types of edit operations, substitution, deletion, and insertion are considered in Levenshtein. Each edit operation has its own cost. The deletion and insertion costs are 1. The substitutions cost for two different characters are 1, otherwise it is 0 (White, 2004).

To discover the minimum cost, the algorithm constructs a score matrix whose rows are mapped to the letters of the source, and whose columns are mapped to the letters of the target except the first row and first column which are reserved as an empty source word and as an empty target word, respectively. The cell at $(i, j)$ determines the minimum cost of transformation a substring $(1, i)$ of $s$ into a substring $(1, j)$ of $t$. In the next step the Levenshtein algorithm starts to complete the matrix. First of all, it fills the first row as an empty source word. Here the optimum transformation of $s$ into $t$ equals to insert 0, 1... $j$ letters into $t$. Levenshtein does the same with the first column as an empty target where in the least transformation cost is the cost of deleting 0, 1... $i$ letters from $s$. For instance in transformation of $pzzel$ to $puzzle$, figure 2 (White, 2004) shows a snapshot of the matrix with filled first row and column:
A name matching algorithm for Persian language

Then in the next step, the values in each remaining cell are calculated by considering its three neighbors as shown in figure 3 and in the following formula:

\[
M(i, j) = \min \left\{ M(i - 1, j - 1) + \text{substitution cost} \right. \\
M(i - 1, j) + \text{delete cost} \right. \\
M(i, j - 1) + \text{insert cost}
\]

Levenshtein continues to fill the score matrix. In the end, the cost in the bottom right-hand cell, for the given example in figure 4 (White, 2004), the value of 3, shows the Levenshtein distance.
Figure 4. Score matrix shows the optimum path and cost

On the score matrix, we can start a walk on the top left-hand cell and continue to reach the bottom right-hand cell through a minimum-cost path. There might be more than one optimum path in the matrix. In fact, each optimal path illustrates a set of edit operations which transform the source word into the target word with minimum cost. For instance the path depicted with dotted light arrows in figure 4 can be read as (White, 2004):

- Substitute p with p (cost 0)
- Insert u (cost 1)
- Substitute z with z (cost 0)
- Substitute z with z (cost 0)
- Insert l (cost 1)
- Substitute e with e (cost 0)
- Delete l (cost 1)

The Needleman-Wunsch

This technique (NW) (Likic, n.d.; McLysaght, n.d.), is used for sequence comparison to find the optimal global alignment between the two. The elements of two sequences are joined up in an alignment. With respect to order, elements can be skipped (Covington, 2004). To skip an element, a gap (a blank character) is inserted in either or both of sequences (Likic, n.d.; McLysaght, n.d.; Covington, 2004).

Example: Suppose two sequences of DNAs are CCCTAGGTCCCA and CGGGTATCCAA. One possible alignment is:

CCC–TAGGTCCCA
CGGGTA––T–CCAA

There are three different types of sequence alignment (Likic, n.d.):

- Global alignment: The best alignment over the entire length of two sequences. Example:
**SIMILARITY**

**PI-LLAR---**

- **Local alignment**: The best alignment over a subsequence of entire lengths, possibly more than one. For example, the local alignment of SIMILARITY and PILLAR could be as:

  MILAR
  ILLAR

- **Multiple sequence alignment**: The best alignment of more than two sequences. For example:

  SIMILARITY
  PI-LLAR---
  --MOLARITY

For Needleman-Wunsch, there are two methods to explore the optimal match/alignment of two sequences (Backofen, 2010):

a. Calculate maximal similarity or score

In this method (Likic, n.d.; McLysaght, n.d.), first of all a *similarity or a substitution matrix* is defined which shows the score or similarity between all symbols, two by two. A gap penalty is also defined here. To see how it works we follow an example in which, the similarity matrix between any symbols pair of A, G, C, and T is defined as:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>G</th>
<th>C</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>5</td>
<td>0</td>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>G</td>
<td>0</td>
<td>7</td>
<td>-3</td>
<td>-2</td>
</tr>
<tr>
<td>C</td>
<td>-1</td>
<td>-3</td>
<td>12</td>
<td>-1</td>
</tr>
<tr>
<td>T</td>
<td>0</td>
<td>-2</td>
<td>-1</td>
<td>5</td>
</tr>
</tbody>
</table>

*Figure 5. A substitution matrix*

Additionally we assume the gap penalty as zero.

Then, to achieve the goal of finding the highest score alignment between the two sequences, a two-dimensional array of $S$ called *score matrix* is allocated and indexed by symbols of each sequence, $s$ sequence letters are mapped to the rows and $t$ sequence letters are mapped to the columns, in addition to a row and a column with zero indices. As the algorithm progresses this matrix is initialized in the first row and first column by the following formulas:

$$S_{0,j} = j * g$$
$$S_{i,0} = i * g$$

Where the $g$ is the gap penalty. In this order, the initialization of the score matrix for our example with two sequences of $s = AT$ and $t = AAGT$ is as in figure 6:
NW continues to fill the remaining cells of matrix with the following formula:

\[
S(i, j) = \max \begin{cases} 
1. S(i-1, j-1) + \text{substitution similarity matrix}(s_i, t_j) \\
2. S(i-1, j) + g \\
3. S(i, j-1) + g 
\end{cases}
\]

Which for the mentioned example result as in the figure below:

Once the score matrix is computed, the most bottom right cell gives the maximum score among all possible alignments. The alignment with computed maximum score is constructed back from the highest scoring cell through a pathway. As on each cell, its value is compared with three possible sources (diagonal-Match, left-Insert, and above-Delete), to see which it comes from. If Match, then \( s_i \) and \( t_j \) are aligned, if Delete, then \( s_i \) is aligned with a gap, and if Insert, then \( t_j \) is aligned with a gap. Thus, the best alignment from figure 7 is:

**S:** A--T  
**T:** AAGT

b. Calculate minimal alignment distance

It is also possible to find the best match or alignment of two sequences through computing minimal alignment distance equals to edit distance in Needleman-Wunsch (Backofen, 2010).

- Here, instead of considering similarity in construction of substitution matrix we define substitution cost of all symbols, two by two. For two identical symbols this cost is zero.
- Matrix \( (D_{i,j}) \) is where \( D_{i,j} \) shows the lowest distance of \( a_1..a_i, b_1..b_j \).
Each entry in matrix of $D_{i,j}$ is calculated as:

$$D(i,j) = \min \begin{cases} 
1. & D(i-1, j-1) + \text{Substitution cost matrix}(s_i, t_j) \\
2. & D(i-1, j) + g \\
3. & D(i, j-1) + g
\end{cases}$$

Where $g$ is the gap cost.

**Limitations / Drawbacks**

The Needleman-Wunsch works well for two similar length sequences which have high similarity across their lengths (Likic, n.d.).

**Smith-Waterman algorithm**

In Smith-Waterman (SW) (Smith & Waterman, 1981) for all possible length subsequences generated from the two main subsequences, the similarity is examined and the subsequence that maximizes the overall two sequences similarity is returned. In other words, it finds the best local alignment (Smith-Waterman algorithm, n.d.; McLysaght, n.d.).

SW works in the same way as Needleman-Wunsch with a small difference. It turns the cell value to zero when the calculated value is a negative number. This makes the local alignment visible to the user (Smith-Waterman algorithm, n.d.). So, each cell of the score matrix is computed as:

$$S(i, 0) = 0$$

$$S(0, j) = 0$$

$$S(i, j) = \max \begin{cases} 
0 \\
1. & S(i-1, j-1) + s(s_i, t_j) \quad \text{if } s_i t_j \\
2. & S(i-1, j) + g \quad \text{if } s_i - \\
3. & S(i, j-1) + g \quad \text{if } -t_j
\end{cases}$$

Where item 2 is the cost of insertion of a gap into $s$ and item 3 is the cost of insertion of a gap into $t$.

SW continues to fill the score matrix. In the end, the best local alignment is found by backtracking on the completed score matrix starting on the maximum cell value and ending at a cell with zero value (Smith-Waterman algorithm, n.d.). The following example describes this clearer. The local alignment between PAWHEAE and HEAGAWGHEE is:

$$S= \text{AWGHE}$$

$$T= \text{AW_HE}$$

As extracted from figure 8 (Ayguade, Navarro, & Jimenez-Gonzalez, 2007).
A name matching algorithm for Persian language

Limitations / Drawbacks

Smith-Waterman is suitable for very different sequences which in are different in their letters and lengths (Likic, n.d.).

Gotoh

Smith-Waterman is optimized into Gotoh method (Backofen, 2010). Gotoh introduces a higher cost for starting a new gap in a sequence alignment than for continuing an existing gap. It defines a linear function for the gap penalty, which is called affine gap.

Limitations / Drawbacks

Gotoh is an improved alternative to the Smith-Waterman which has better scaling (Gotoh, 1982) which is specialized to the properties of biological sequences.

Jaro-Winkler

Jaro-Winkler (JW) (Jaro, 1989; Winkler, 2006; Winkler, 1990), measures the similarity of two strings. It came after Jaro to make a difference to give more score to the strings which are similar in their beginnings. Two identical strings score 1 in Jaro-Winkler whereas two totally different strings score 0. The Jaro distance $d_j$ of two given strings $S_1$ and $S_2$ is

$$d_j = \begin{cases} 
0 & \text{if } m = 0 \\
\frac{1}{3} \left( \frac{m}{|s_1|} + \frac{m}{|s_2|} + \frac{0.5t}{m} \right) & \text{otherwise}
\end{cases}$$
Where:
- \(|s_1|\) and \(|s_2|\) are lengths of \(s_1\) and \(s_2\), respectively.
- \(m\) is the number of common characters between string \(s_1\) and \(s_2\) where the distance for common is half of the minimum length of \(s_1\) and \(s_2\).
- \(t\) is the number of transpositions

Given two strings \(S_1\) and \(S_2\), their Jaro-Winkler distance \(d_w\) is:

\[
d_w = d_j + (lp(1 - d_j))
\]

Where:
- \(d_j\) is the Jaro distance for strings \(S_1\) and \(S_2\)
- \(l\) is the length of common prefix at the start of the string up to a maximum of 4 characters
- \(p\) is a constant scaling factor for how much the score is adjusted upwards for having prefixes.

**Limitations / Drawbacks**

Although Jaro-Winkler is simple and fast, it only concerns the common characters and doesn’t take into account the different levels of similarities between non-common characters.

**Token based algorithms**

These kinds of algorithms examine the strings as a group of words or tokens. Character arrangement is the subject they concern to handle and they do not deal with typographical errors. Therefore they give more accurate results when they treat different orders of characters such as “William Smith” vs. “Smith William”. One of the most known algorithms of this group is called Q-gram.

**Q-gram**

Every Q-gram is a sequence of Q letters from a longer word. Here, words are divided into Q-gram fragments. The similarity of two words is the number of fragments they have in common. For instance, the string “Nelson”, for \(Q = 2\), has the following q-grams: NE EL LS SO ON and Neilsen’s 2-grams are: NE EI IL LS SE EN. Therefore the 2-gram of these two names is 2 equal to their common 2-grams, NE and LS.

**Limitations / Drawbacks**

Q-gram works well for fielding variation but this type of name variations generally occurs rarely, and encompasses a small percentage of all total variations which may affect the names, so applying it where name variations happen more because of other reasons, would not give a good and desirable result.
Persian language

Before going deep into the details of the algorithm proposed in this thesis, there is a need to have an introduction of the Persian alphabet, its writing system and have a look at letter similarities from different aspects. This section is essential, since most of name matching algorithms are basically language dependent and in this thesis the aim is to introduce a name matching algorithm for Persian. Therefore this part provides a necessary view of the features of Persian which are a foundation of our work. Persian language is an Iranian language, as described more in the following:

Persian alphabet

The Persian language is often written using the Persian/Farsi alphabet which has 33 Persian letters, written from right to left as following:

Some experts say Hamza, second from right is not a letter and it is only used in writing system.

Different writing systems are used for writing Persian. This project is aimed at Perso-Arabic which is a writing system based on Arabic script, Perso-Arabic is the standard for writing Persian and is the script system used officially in Iran. Persian according to this system is written from right to left and cursively. Cursively means that every letter takes different shapes based on its position in the word. Usually a Persian letter has 3 variations, one at the beginning of the word, another in the middle of the word, and the other one at the end of the word. Table 3 shows the Persian alphabet with their different shapes.

<table>
<thead>
<tr>
<th>Persian Alphabet</th>
<th>Variations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Initial</td>
</tr>
<tr>
<td>ا</td>
<td>ا</td>
</tr>
<tr>
<td>ء</td>
<td>ء</td>
</tr>
<tr>
<td>ب</td>
<td>ب</td>
</tr>
<tr>
<td>ج</td>
<td>ج</td>
</tr>
<tr>
<td>د</td>
<td>د</td>
</tr>
</tbody>
</table>
In addition to the above letters, there are some letters which can be typed through Persian keyboard but they are not Persian, as listed in table 4:

<table>
<thead>
<tr>
<th>Some Arabic letters</th>
<th>Variations</th>
<th>Initial</th>
<th>Medial</th>
<th>End</th>
</tr>
</thead>
<tbody>
<tr>
<td>ج</td>
<td>ج</td>
<td>ج</td>
<td>ج</td>
<td></td>
</tr>
<tr>
<td>ن</td>
<td>ن</td>
<td>ن</td>
<td>ن</td>
<td></td>
</tr>
<tr>
<td>ه</td>
<td>ه</td>
<td>ه</td>
<td>ه</td>
<td></td>
</tr>
<tr>
<td>ي</td>
<td>ي</td>
<td>ي</td>
<td>ي</td>
<td></td>
</tr>
</tbody>
</table>

Although these extra letters are not included in Persian, they are considered in our algorithm as well, because they can be typed wrongly instead of Persian letters in a name by the operator when the keyboard is set to Persian.
Chapter 3: Method

A name matching algorithm for Persian language

Since in the beginning there was no idea about what the best solution could be, in the first stage, we started to study the earliest algorithms in name matching. The goal set on covering different kinds of solutions and methods for name matching, at least the most known in order to achieve an overview on subject, name matching algorithms, in order to guide us towards finding out a solution for this topic thesis. Considering what was said in previous section, we can see there are different kinds of name variations and, various name matching algorithms. But because of variety in name variations, each name matching method focuses on some particular name changing and actually there is no one covering all aspects of variations. There are some primary mechanisms to perform approximate name matching, while many of the methods were established based on one of these fundamental and main techniques. Some came into existence to combine two or more of these main solutions. All of the latter methods were developed to improve the earlier ones in order to give more accurate results or work more effective. Thus there might be many editions and variations of one main algorithm. Moreover, because the name matching algorithms are language dependent, and many existing algorithms put English as their target, a number of subsequent algorithms were constructed to adapt to a specific language. As we also can see in some examples in the previous, every method has its own advantages and disadvantages and there is no one to treat all types of name variations and that works well for all languages. Therefore we decided to design a new method based on a primary algorithm in a way that make it suitable for Persian. In our literature study on name matches, we encountered an algorithm called Arabic Edit Distance Algorithm (AEDA) (Abdel Ghafour, El-Batawissy, & Heggazy, 2011) developed mainly for the Arabic language with its own extended version of Levenshtein. Since Persian and Arabic are close linguistically, we made AEDA as the basis for our work in this thesis. But AEDA has another strong advantage that persuaded us more and more to make it as our foundation. It uses three kinds of similarity: phonetic, form and keyboard, while as what described in previous chapter, most of algorithms focused and developed only based on one of these similarities.

PEDA has the same structure and components as AEDA, but PEDA is founded on characteristics of Persian while AEDA is on Arabic. In this thesis, phonetic similarities in PEDA are extracted based on how the Persian letters are pronounced which their pronunciations differ mainly with the pronunciation of Arabic letters, although they have most of letters in common. In addition the form similarity between Persian letters are made based on our own rules and considerations, while we didn’t know how they are grouped in AEDA, in addition that Persian has extra letters as well. And finally keyboard similarities are computed based on Persian keyboard layout which it may close to Arabic keyboard. Some similarities are based on grammar and points in Persian writing while the grammars of these two languages are basically
A name matching algorithm for Persian language

PEDA as AEDA are made from the following components which will be described in following sections:

- Phonetic similarity
- Form similarity
- Keyboard similarity
- PEDA's Core (Extended Levenshtein)

Levels of similarity in Persian language

Persian letters can be compared to each other from three aspects of similarity. They are:

Form similarity

As mentioned in previous sections, there are 33 Persian letters. A few have two forms, some of them have three forms and some more than 3 forms. Totally there are 142 forms which represent the Persian alphabet. The major cause of this variety is that letters have different shapes depending on their position in a word. If we examine the form appearances, we notice that there is some kind of similarity among groups of them. That is why when an operator read a text written by hand he/she may wrongly get one letter instead of another one. This phenomenon is more probable in dealing with names. Because the meaning of names is not a matter and does not help the operator to detect the exact letters included. Therefore the form similarity is one of the sources which may cause variations in identical names. To be capable to take into consideration the form similarity/dissimilarity in our approximate matching algorithm, here, we extract all possible form similarities among Persian letters. Actually finding form similarity between letters always is not an easy task. Because we cannot always say that why two letters seems to be similar by specific reasons. In most of cases it is just a perception that we notice a pair looks like each other. Despite of this challenging matter, we tried to discover similar pairs and classify them based on specific and exact reasons whenever it was possible. In order to be able to measure the form similarity, the important point for us was that to use a scoring system and assign each group a score which in fact is a definition of similarity value for all included pairs of this group. On the other hand a group score shows the percentage of its similarity compared to other groups. The goal was that to give more score to the group in which the members of each pair have more similarity with each other compared to the pair members of another group which have lower similarity to each other. That was the reason that for example we defined the score of 0.8 for the second group in table 5. The matter was that to give this group a score value lower than the first group score and higher than the third group score. Thus, the score value of 0.8 was our randomly chose between 1 and 0.6, in addition we tried to choose the middle value in a range. But it could be for example 0.9 instead of 0.8. This flexibility in score selection is really an advantage for our algorithm. This feature gives the algorithm a potential adjustable capability to the real data, perhaps will be implemented in newer PEDA versions. In this thesis, we define each similarity as a pair of letters and their similarity value, called as a rule or similarity rule as well. In the result, we put together all
observed similarity rules in number of ranking groups depend on their equivalences’ degrees as in table 5.

<table>
<thead>
<tr>
<th>No.</th>
<th>Similar Groups</th>
<th>Similarity Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>(ک - ل) (ک - ل)</td>
<td>1</td>
</tr>
<tr>
<td>2.</td>
<td>(ک - گ) (ک - گ)</td>
<td>0.8</td>
</tr>
<tr>
<td>3.</td>
<td>(ز - ز) (ز - ز)</td>
<td>0.6</td>
</tr>
<tr>
<td>4.</td>
<td>(ج - ح) (ج - ح)</td>
<td>0.4</td>
</tr>
<tr>
<td>5.</td>
<td>(ه - ی) (ک - ل)</td>
<td>0.2</td>
</tr>
</tbody>
</table>
In construction of table 5, all Persian letters’ forms are used in addition to the forms which do not belong to the set of Persian letters but that can be typed using a Persian keyboard. Table 5 depicts that all pairs of letters are placed into 6 categories according to a similarity index starting from 1 and ending at 0. The letters connected to each other with similarity index of 1 are considered completely the same. Actually they belong to their own correspondent Persian letter, in cases more than one shape is used for a Persian letter, like the letter of گ that has two shapes of گ – گ. Also this category includes any pairs which can replace each other without making any differences in Persian writing, like the pair of ی - ی. The second group is made of pairs which have only one difference in their appearances such as an extra dot. The third one, the group with similarity index of 0.6, is composed of the couples which have two differences compared to each other. The fourth group, are the ones which have three differences. And the group with 0.2 similarity index connects every two forms which are similar in a sensible way recognized by the human judgment not because of a clear and explicable reason. The all possible remaining pairs fall into the last group with a similarity index of 0.

Table 5 was used as the form similarity reference in this project until the middle of work. Then we switched to a new version. The reason was that actually every Persian name is stored inside an application or a database as a sequence of letters in their neutral forms. But at the time it is coming on a screen to be shown to the end user, a text editor get involved to parse the strings and detect the letter positions inside the words, then according to their position select the appropriate letter forms and finally join them together and present to the user on the screen. Thus by the usage of what is kept for a name, the position-dependant letter forms are not available. But so far all form similarity rules have been defined on position-dependent letter forms. At this point there are two possible solutions to continue the work:

1. Apply a text editor in the middle, in order to parse names and detect the letter positions inside the words, and then give the following component the position-dependent letter forms. In this solution, table 5 can be used as the form similarity reference when needed.

2. As can be seen in the table 3, almost 80%-90% of rules are in the same group for all position-dependent forms of a letter. So another solution would be to combine all related rules of each
specific Persian letter pair from table 5 into a new one rule that is a couple of letters in their neutral forms beside their similarity value.

The first solution is more accurate than the second one, but it is slower and more costly as well because it needs the inclusion of a text render. In this thesis the second solution has been followed.

To make the new version of the form similarity table, for each pair of Persian letters, the similarity average of all their position-dependent forms from table 5 is calculated, then a new rule is included where two neutral letter forms beside the computed similarity average is made and put in the corresponding index in new table, table 6, instead of all related ones in table 5. This process is continued for all pair of Persian letters and the result is table 6. To illustrate a sample, in table 5 we have the two rules of (ب - ن) and (ﺒ - ﻧ) in similarity index of 0.2. Also the rule of (ب - ﻧ) falls into the last category. All of these rules belong to the pair (ب - ن). The similarity average for all position-dependent forms of this pair of letters in table 5 is calculated as (0.2 + 0.2 + 0)/3 = 0.13333333 almost equals to 0.14. Thus the new rule is the pair (ب - ﻧ) with a similarity index of 0.14.

### Form Similarity in Persian Alphabet (between origin letter forms)

<table>
<thead>
<tr>
<th>No.</th>
<th>Similar Groups</th>
<th>Similarity Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(ب - ن) (ب - ن) (ب - ن) (ب - ن) (ب - ن) (ب - ن)</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>(ب - ن) (ب - ن) (ب - ن) (ب - ن) (ب - ن) (ب - ن)</td>
<td>0.8</td>
</tr>
<tr>
<td>3</td>
<td>(ب - ن) (ب - ن) (ب - ن) (ب - ن) (ب - ن) (ب - ن)</td>
<td>0.54</td>
</tr>
<tr>
<td>4</td>
<td>(ب - ن) (ب - ن) (ب - ن) (ب - ن) (ب - ن) (ب - ن)</td>
<td>0.6</td>
</tr>
<tr>
<td>5</td>
<td>(ب - ن) (ب - ن) (ب - ن) (ب - ن) (ب - ن) (ب - ن)</td>
<td>0.4</td>
</tr>
<tr>
<td>6</td>
<td>(ب - ن) (ب - ن) (ب - ن) (ب - ن) (ب - ن) (ب - ن)</td>
<td>0.27</td>
</tr>
<tr>
<td>7</td>
<td>(ب - ن) (ب - ن) (ب - ن) (ب - ن) (ب - ن) (ب - ن)</td>
<td>0.2</td>
</tr>
<tr>
<td>8</td>
<td>(ب - ن) (ب - ن) (ب - ن) (ب - ن) (ب - ن) (ب - ن)</td>
<td>0.14</td>
</tr>
<tr>
<td>9</td>
<td>(ب - ن) (ب - ن) (ب - ن) (ب - ن) (ب - ن) (ب - ن)</td>
<td>0.07</td>
</tr>
</tbody>
</table>

Table 6. Form similarity in Persian alphabet (between origin letter forms)
**Phonetic similarity**

Another aspect of similarity can be in the pronunciation of two different letters. In order to extract the phonetic similarities between Persian letters and build their corresponding rules, the sound of alphabet and the way of their production in vocal tract system is considered as the main factor. We used a scoring system like the one used for form similarity as well to be able to measure the phonetic similarity inside the algorithm. We examined the sound of all Persian letters and then made the rules as in the following table:

<table>
<thead>
<tr>
<th>No.</th>
<th>Similar Groups</th>
<th>Similarity Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>(ک - ک) (ک - گ) (ی - ی) (ی - ی)</td>
<td>1</td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(ک - ک)</td>
<td>(ی - ی)</td>
</tr>
<tr>
<td>(ک - گ)</td>
<td>(ی - ی)</td>
</tr>
<tr>
<td>(ی - ی)</td>
<td>(ک - ک)</td>
</tr>
<tr>
<td>(ک - گ)</td>
<td>(ی - ی)</td>
</tr>
<tr>
<td>(ی - ی)</td>
<td>(ک - گ)</td>
</tr>
<tr>
<td>(ی - ی)</td>
<td>(ی - ی)</td>
</tr>
<tr>
<td>(ک - ک)</td>
<td>(ی - ی)</td>
</tr>
<tr>
<td>(ک - گ)</td>
<td>(ی - ی)</td>
</tr>
<tr>
<td>(ی - ی)</td>
<td>(ک - ک)</td>
</tr>
<tr>
<td>(ک - گ)</td>
<td>(ی - ی)</td>
</tr>
<tr>
<td>(ی - ی)</td>
<td>(ک - ک)</td>
</tr>
<tr>
<td>(ک - گ)</td>
<td>(ی - ی)</td>
</tr>
<tr>
<td>(ی - ی)</td>
<td>(ک - ک)</td>
</tr>
<tr>
<td>(ک - گ)</td>
<td>(ی - ی)</td>
</tr>
<tr>
<td>(ی - ی)</td>
<td>(ک - ک)</td>
</tr>
<tr>
<td>(ک - گ)</td>
<td>(ی - ی)</td>
</tr>
<tr>
<td>(ی - ی)</td>
<td>(ک - ک)</td>
</tr>
<tr>
<td>(ک - گ)</td>
<td>(ی - ی)</td>
</tr>
<tr>
<td>(ی - ی)</td>
<td>(ک - ک)</td>
</tr>
<tr>
<td>(ک - گ)</td>
<td>(ی - ی)</td>
</tr>
<tr>
<td>(ی - ی)</td>
<td>(ک - ک)</td>
</tr>
<tr>
<td>(ک - گ)</td>
<td>(ی - ی)</td>
</tr>
<tr>
<td>(ی - ی)</td>
<td>(ک - ک)</td>
</tr>
<tr>
<td>2.</td>
<td>(ع - ع) (ع - ع)</td>
</tr>
<tr>
<td>-----</td>
<td>----------------</td>
</tr>
<tr>
<td>3.</td>
<td>(ب - ب) (ب - ب)</td>
</tr>
<tr>
<td>4.</td>
<td>(ت - ت) (ت - ت)</td>
</tr>
</tbody>
</table>

A name matching algorithm for Persian language 35
In order to extract phonetic similarities and classify them to make table 7, we placed the letters with completely equal sounds in Persian into the highest group, the one with the similarity index of 1. The pairs placed into the second group with a similarity index of 0.8 don’t have the same sound per se, but they can be used instead of each other in a word without any effect on its phonetic structure. For the fourth group with a similarity index of 0.4, we selected the pair of letters whose sounds physically is produced approximately in the same way, maybe with a little difference. The sounds of letters in the pair (ب - پ) from this group are explosive, obstructive and both lips are used in their production. The pair (ت - د) is selected for the fourth group as well, because the included letters are explosive and both are produced using teeth. The pairs of (ض - ض)، (س - ض)، (س - ط)، (س - ز)، (ت - ض)، (ت - ط)، (ت - ز)، (ص - ض)، (ص - ط)، (ص - ز)، and (ص - ض) include letters which all are rubbing and sibilant. The pair of (ج - چ) falls into the fourth group as well, because both letters’ sound is explosive and rubbing. The letters in the pair (ز - ش) from this group are also rubbing and breathing. The pair (ف - و) is placed into the fourth category as well due to rubbing and in their production lips and teeth are used. The letters (ک - گ) are pronounced approximately in a similar way, both are explosive and the palate is used in their construction. Thus this falls into the fourth group. (ک - گ) is the same as (ک - گ). Finally, the letters of (م - ن) put together as a pair into class of 4, as they are nasal. But the pair (ت - چ) which is placed in the sixth group, might be used interchangeably but we actually do not have the letter چ in the Persian alphabet.

Because of the same reason mentioned in the form similarity part, we removed all pairs of position-dependent forms from table 7 and just kept the pairs of base letters in table 8:

<table>
<thead>
<tr>
<th>Pair</th>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>ج - چ</td>
<td>1.0</td>
</tr>
<tr>
<td>ز - ش</td>
<td>0.8</td>
</tr>
<tr>
<td>ف - و</td>
<td>0.8</td>
</tr>
<tr>
<td>ک - گ</td>
<td>0.6</td>
</tr>
<tr>
<td>م - ن</td>
<td>0.4</td>
</tr>
<tr>
<td>ت - چ</td>
<td>0.2</td>
</tr>
<tr>
<td>ن - ت</td>
<td>0.1</td>
</tr>
<tr>
<td>Any other pair of Persian letters</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Table 7. Phonetic similarity in Persian alphabet (between all position-dependent letter forms)
**Phonetic Similarity in Persian Alphabet (between origin letter forms)**

<table>
<thead>
<tr>
<th>No.</th>
<th>Similar Groups</th>
<th>Similarity Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(ک - ک) (ی - ی) (ا - ع) (ا - ی) (ا - ی) (ا - ی)</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>(و - ی) (ت - ط)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(ت - س - ص)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(ز - ن - ض - ط)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(خ - گ) (خ - گ) (ع - گ) (غ - ی)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(و - ی) (ع - ی)</td>
<td>0.8</td>
</tr>
<tr>
<td></td>
<td>(ی - ی)</td>
<td>0.6</td>
</tr>
<tr>
<td></td>
<td>(ب - پ) (ت - د)</td>
<td>0.4</td>
</tr>
<tr>
<td></td>
<td>(ت - س - ص - ز - ن - ض - ط)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(ج - ژ) (ز - ژ) (ف - ژ) (ک - گ) (ک - گ) (م - ن)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(ن - ی)</td>
<td>0.2</td>
</tr>
<tr>
<td></td>
<td>(ه - ت)</td>
<td>0.1</td>
</tr>
<tr>
<td>Any other combination of Persian letters</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Table 8. Phonetic similarity in Persian alphabet (between origin letter forms)

It is necessary to mention that the assigned scores all are our definition.

**Keyboard similarity**

Since one of the reasons for name variations is the proximity of keys on the keyboard, this section discusses this issue based on the keyboard layout for Persian used in Microsoft Windows. Following are all diagrams which show all Persian letter forms with their positions on a Persian keyboard axis:

![Persian keyboard layout](image_url)

Figure 9. Persian letters on the keyboard in their neutral forms
Figure 10. Letters and sounds typed when SHIFT key is pressed

Figure 11. The initial forms of Persian letters on the keyboard axis

Figure 12. The medial forms of Persian letters on the keyboard layout.

Figure 13. The end forms of Persian letters on the keyboard layout.
The closer two letters are on the keyboard, the more probable is the operator to make a mistake. In other words the similarity of two keys increases if they are closer on the keyboard layout. Thus we can say that the keyboard similarity has a diverse relationship with distance as it is presented in the following formula:

$$Sim_{kb}(a, b) = 1 - \frac{\sqrt{(x_a - x_b)^2 + (y_a - y_b)^2}}{\psi}$$

Where $x_a$ & $x_b$ are the positions of any two keys (a & b) on X axis, $y_a$ & $y_b$ are the positions of the two keys (a & b) on Y axis, and $\psi$ is the maximum possible distance on Persian Keyboard, this distance is approximately 12 units which is the distance between ‘پ’ & ‘ض’ (Abdel Ghafoor, El-Batawissy, & Heggazy, 2011).

**Persian Edit Distance Algorithm’s Core**

Persian Edit Distance Algorithm, PEDA for short, proposed in this thesis is designed and implemented mainly to measure the similarity between two Persian names and match the ones with a high enough similarity as identical names. The idea of PEDA is taken from Arabic Edit Distance Algorithm (AEDA) that is based on the Levenshtein algorithm. Levenshtein uses dynamic programming to calculate the minimum cost of transforming string $s$ into string $t$. AEDA make some changes to Levenshtein and is extended to make it suitable for Arabic names. PEDA shares traits and characteristics with AEDA, but it stands on the features and properties of the Persian language. In order to compute the minimum transformation cost between two strings, Levenshtein counts the least numbers of edit operations needed to transform $s$ into $t$, while it regards the same cost for all operations. Basically Levenshtein is a distance based algorithm that compares the strings’ patterns, character by character. But PEDA like AEDA considers different costs for Insert, Delete and Substitution operations to give more accurate results. PEDA takes into consideration three aspects of similarity described in previous section to compute the cost of edit operations.

Before discussing PEDA in details, we take a look at the pseudo code of the core algorithm:
A name matching algorithm for Persian language

```c

decimal PEDA(char s[1..m], char t[1..n])
{
    declare decimal Cost[0..m, 0..n] // Cost Matrix
    for i from 0 to m Cost[i, 0] = i
    for j from 0 to n Cost[0, j] = j
    for j from 1 to n {
        for i from 1 to m {
            Cost[i, j] = min (Cost[i - 1, j] + f_idc(s[i], s[i - 1]),
                                Cost[i, j - 1] + f_idc(t[j], t[j - 1]),
                                Cost[i - 1, j - 1] + f_rel(s[i], t[j]))
        }
    }
    return Cost[m, n]
}
```

As the pseudo code shows, PEDA just like Levenshtein maps the letters of both input names to the rows and columns of a score matrix. The matrix has an extra row and column in zero indices, as well. PEDA walks through the cells and fills them. After the first row and column are filled, the code continues to complete remaining cells. Each cell is filled with the minimum of its three neighbors values added to their corresponding costs. A cell, the cross of the i\textsuperscript{th} row and j\textsuperscript{th} column of the matrix, is the representative of the letter s\textsubscript{i} from string s and the letter of t\textsubscript{j} from string t. Actually on each cell, the code examines which edit operation makes the minimum number of total edit operations to transform s into t up to that cell, that is for s\textsubscript{1} ... s\textsubscript{i} to t\textsubscript{1} ... t\textsubscript{j}.

The most right-bottom cell in the completed matrix returns the minimum cost needed to transform s into t.

In the following section we will illustrate in detail how the algorithm calculates the insertion, deletion and replacement costs:

**Cost of insertion and deletion operations**

The cost of insertion and deletion operations is a value between 0 and 1. Where 0 shows minimum and 1 shows maximum cost. Our major aim is to calculate the cost according to the Persian language’s properties. To achieve this goal, grammar and features of this language are analyzed in order to extract the rules which may affect the cost of insertion and deletion operations. At first we define that the insertion or deletion cost of a blank equals to zero beneficial to segmentations which might occur in names. Thus if a blank appears accidentally inside a name, our method will be able to treat it. The second issue is that if Hamza letter is placed after long vowels (alif, ya, and waw) in a name, the Hamza can be removed without any negative effect. Both names, with and without Hamza are correct. Therefore the cost of insertion or deletion of one Hamza after long vowels is defined as zero. The next principle, in Persian words, instead of a duplicate letter, a diacritic called “Tashdid” is used above the letter which is pronounced twice. Almost in all cases, especially in names, “Tashdid” is also dropped from the name. For example “محمد” should be written as “محمد” but it is written in form of “محمّد” in personal names and the letter of “م” is repeated just in pronunciation. Thus the
insertion or deletion cost of duplicate letters is set to a value smaller than one. In addition, in Persian long vowel sounds are used as consonant letters or to show the sounds. This is the reason that we defined another parameter to give lower insertion or deletion cost for long vowels. The cost for any other cases is defined as 1. These issues are formulated in the following mathematical formula to be used in the algorithm.

\[
\begin{align*}
    f_{idc}(a_i, a_{i-1}) = \begin{cases} 
    0 & \text{if } a_i = \text{blank} \\
    0 & \text{if } a_i \neq a_{i-1} \quad a_i = * \quad a_{i-1} \in \xi = \{\text{alif, ya, waw}\} \\
    \theta & \text{if } a_i = a_{i-1} \\
    \mu & \text{if } a_i \in \xi = \{\text{alif, ya, waw}\} \\
    1 & \text{Otherwise}
    \end{cases}
\end{align*}
\]

Where \( f_{idc}(a_i, a_{i-1}) \) is the cost of inserting or deleting the letter \( a_i \), \( \theta \) is the cost of inserting or deleting the letter \( a_i \) if it is equal to the previous letter \( a_{i-1} \) and it is smaller than 1. The symbol \( \xi \) is the set of long vowels. The symbol \( \mu \) is the insertion or deletion cost of long vowels.

**Cost of replacement operation**

As described in previous sections the similarity of letters can be seen from three main aspects. In replacement operation, logically the replacing cost of two similar letters should be smaller than the replacing cost of two different letters. To reflect this fact in addition to consider all similarity perspectives in PEDA, a proper formula should be defined. To see different levels of similarity, getting an average between them can be a solution. But there are many formulas of measures of central tendency, like arithmetic mean, geometric mean, harmonic mean, median, weighted mean and so on. To find out which one is most suited in our algorithm to return the influence of all similarity aspects in calculating the final transformation cost, a review of these arithmetical measures is given here:

- **Arithmetic mean**
  
  It is the sum of a collection of values divided by the number of values in the collection. In other words the arithmetic mean for the values of \( x_1, ..., x_n \), denoted by \( \bar{x} \) is defined via the expression (Medhi, 1992)

  \[
  \bar{x} = \frac{x_1 + x_2 + ... + x_n}{n}
  \]

- **Geometric mean**
  
  It is a type of mean or average, which indicates the central tendency or typical value of a set of numbers by using the product of their values as opposed to the arithmetic mean which uses their sum. The geometric mean is defined as the \( n \)th root of the product of the numbers where \( n \) is the count of numbers (Geometric mean, n.d.).
- **Harmonic mean**
  The harmonic mean for a set of observations is such that its reciprocal is the arithmetic mean of the reciprocals of the values of the observations (Medhi, 1992).

- **Median**
  The median of a set of measurements or observations $x_1, \ldots, x_n$ is the middle value when the measurements are arranged in an array according to their order of magnitude. If $n$ is odd, the middle value which is the $(n/2 + 1)$th in the ascending order of magnitude, is unique and is the median. If $n$ is even, there are two middle values and the average of these values is the median (Medhi, 1992).

- **Weighted mean**
  Is an average in which some data points count more strongly than others, in that they are given more weight in the calculation (Arithmetic mean, n.d.).

As the main purpose is to consider all aspects of similarity in calculating the cost of replacement, the median cannot be a proper average for our situation, because as we can see from its definition, all observations do not participate in calculation of median (Medhi, 1992). As such it cannot be a representative (Medhi, 1992) for our three aspects. Also the geometric mean is useful in dealing with ratios, rates etc (Medhi, 1992). While the three values of phonetic similarity, form similarity and keyboard distance are not a type of ratio or rates. The harmonic mean the same as geometric mean is proper for the average of rates (Harmonic mean, n.d.). On the other hand the arithmetic mean which its definition is precise and exact is computed easy. It also has a determinate value and it is a point of balance (Medhi, 1992). Therefore it can be a good solution here. But a more flexible formula in which gives the user more power to adapt the formula to different situations and get more precise results depends on those situations, makes PEDA more powerful too. This more flexible formula can be the weighted mean in that data points with higher weights contributes more in the average. The usage of the weighted formula gives the user a capability to adjust the algorithm parameters depending on different operational environments by giving different weights to phonetic, form, or keyboard similarities and their respective costs. Given the discussion, the below formula is applied to calculate the cost of replacement for each pair $(a,b)$ of input letters:

$$f_{rc}(a, b) = \begin{cases} \frac{\alpha(a, b) \cdot \omega + \beta(a, b) \cdot \lambda + \gamma(a, b) \cdot \sigma}{\omega + \lambda + \sigma} & \text{if } a \neq b \\ 0 & \text{Otherwise} \end{cases}$$

Where $f_{rc}(a, b)$ is a weighted average formula, for the replacement cost for two letters, $a$ and $b$. The same as insertion/deletion cost, replacement cost ranges from 0 to 1, as well. $\alpha, \beta$ and $\gamma$ are replacement costs for phonetic, form and keyboard distance similarity respectively. Each cost is equal to $1 - \text{similarity index}$, where similarity index is picked up from the similarity tables defined earlier in previous sections. $\omega, \lambda \text{ and } \sigma$ are the respective weights of costs. The different weights ($\omega, \lambda \text{ and } \sigma$) reflect the impact of different similarity levels. Depending on the
environment and situation a type of error or variation may occur more than the others. For example if data entry is done by reading from a hand written form, the probability of form variation gets higher than the phonetic or keyboard variation. On the other hand, if reception of data is done through a verbal communication or over the telephone, it increases the probability of phonetic errors. Therefore these weights are used to adapt the formula to the operational environment and get more accurate results depending on different situation.

To clarify our point, the following example shows the calculation of the replacement cost between م and ه:

- Phonetic cost is calculated as $\alpha = 1 - \text{phonetic similarity index}$. From table 8 the phonetic similarity index between م and ه is 0 and hence $\alpha(م, ه)$ will be 1.
- Form similarity cost is calculated as $\beta = 1 - \text{form similarity index}$. From table 6 it is 0.07 for these two letters and hence $\beta(م, ه)$ will be 0.93.
- The keyboard distance between م and ه is 0.92 so the keyboard cost will be 0.08.
- By considering equal weights for all operations, the replacement cost between م and ه is calculated as $(1*1 + 1*0.93 + 1*0.08)/(1+1+1) = 0.67$

Back to the PEDA core, for each pair of names it constructs a score matrix regarding the costs, to give an example, suppose that PEDA compares two names of $s = مراد$ into $t = موراذ$, then the constructed score matrix is:

<table>
<thead>
<tr>
<th></th>
<th>م</th>
<th>و</th>
<th>ر</th>
<th>ا</th>
<th>د</th>
</tr>
</thead>
<tbody>
<tr>
<td>م</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0.15</td>
<td>1.15</td>
<td>1.3</td>
<td>2.3</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0.77778</td>
<td>0.15</td>
<td>0.3</td>
<td>1.3</td>
</tr>
<tr>
<td>3</td>
<td>1.15</td>
<td>0.92778</td>
<td>0.3</td>
<td>0.15</td>
<td>1.00595</td>
</tr>
<tr>
<td>4</td>
<td>2.15</td>
<td>1.87222</td>
<td>1.3</td>
<td>0.99444</td>
<td>0.57778</td>
</tr>
</tbody>
</table>

Figure 15. The score matrix for مراد and موراذ

The minimum distance calculated for transformation of مراد into موراذ is the value of bottom right-hand cell equal to 0.57778.

**The similarity of two names**

What is calculated by PEDA is the minimum transformation cost for two names. But finally we want to determine the percentage of their similarity. To compute the value of their similarity we can use following formula (Christen, 2006):

$$\text{Sim}(s, t) = 100 - \frac{\text{minimum distance} \times 100}{\max(|s|, |t|)}$$
Where |s| is the length of source string and |t| is the length of target string. This formula is actually based on Jaccard similarity metric which takes into consideration the maximum string length. Basically in Levenshtein we consider the maximum length because if one string is longer than another string, deletion or insertion cost of extra letters are reflected in calculating final edit distance.

In previous example, the percentage of similarity according to the formula is approximately 88%:

\[
\text{Sim}(\text{موردادر}) = 100 - \frac{0.57778 \times 100}{\max(4,5)}
\]

**Implementation of PEDA**

In this thesis, we implemented PEDA inside a windows desktop application by Java Swing and SQL Server 2008 as its database from scratch. A system overview is depicted in figure 1. In order to have flexibility and capability of performing different types of matching, as it will be described in the following, we put different sections into different tabs inside our application.

![Figure 16. Separate first names and last names](image-url)
In figure 16, the first tab in the main window of our program displays the section in which end user can select a watch list included first name and last name in separate fields. And then load a source list with the same format by sending a query to the database. The records of watch list and source list can be seen in two data grids and user can scroll among their records. The user can press the “Apply PEDA” button and then PEDA starts working. In this part, each first name of watch list is compared with all first names in source list and each last name of watch list is compared with all last names in source list. The results are saved inside the database and they can be retrieved inside a data grid by sending a query to the database after PEDA completes its work.

![Figure 16. Watch list and source list](image)

Figure 17 displays another part of the program, where the formats of watch list and source list are different compared to the previous section and include just one field for a name. The name can be first name, last name or concatenated first name and last name. Each name from watch list is compared with all names in source list and the results are saved inside the database.

![Figure 17. Not separate first names and last names](image)

In order to prepare a capability inside the program to adjust PEDA with different values of parameters or similarity rules, we added extra forms for these actions as well, shown by following figures:
A name matching algorithm for Persian language
Name data sets

The base for the experiments in this thesis is a data set of 1 million records of first names and last names extracted from a bank customer data set collected from different banks in Iran. Because there were no access to the prepared data sets for test included Persian names, we used a portion of real data. To measure the accuracy of PEDA algorithm, three series of experiments performed which will be described in the following:

1. Firstly 1000 records are selected randomly from the 1 million records. Although our major data set consists of both first names and last names, we picked up only the data of first names to use in our experiments. Then among the randomly chosen 1000 records, the redundant first names are removed and replaced with the other randomly selected first names from the main data set. We repeated the redundancy checking to make our data set from 1000 numbers of completely different records. After that, we divided these 1000 records into 10 data sets of 100 records. Then we made a copy of all these 100 record data sets and applied some modification to all their records. Next in ten iterations, we executed the program, each time with one of these modified data sets of 100 records as the watch list against those 1000 records as the source list. Therefore in every execution, 100,000 matches were carried out by the PEDA and the
results were put in the database. Totally 10 result sets, each one included 100,000 records are produced in these series of experiments to be used in evaluation.

2. In another experimental series, 100 records of first names are selected randomly from the 1 million records. From these 100 records, we connected 50 ones to their known identical records in order to make 50 known true matches of first names. Then the remaining 50 records of these 100 records are connected to another 50 completely different names in order to make 50 known false matches of first names. Thus in this way a 100 matches are created included 50 known true matches and 50 known false matches. Then we disconnected all the known true and false matches into two data sets, each included 100 first names. Then, we put away one of these data sets to use later in our program as the source list. Next we change the place of records in the second data set so that all names are placed randomly. Then we divide it to 10 data sets of 10 records. Next we execute the program for ten iterations and each time we use one of 10 records as the watch list against the first data set of 100 records which we kept away as the source list. In each execution 1000 matches were carried out by the PEDA. It should be mentioned that in selection of 50 known true matches we tried to cover all different variation of names listed in previous chapters such as spelling, segmentation and so on.

3. On the other series of experiments, we wanted to evaluate how PEDA works for different number of edits. We made four groups; each one includes 100 of known true matches. The first category consists of true matches that differ from each other just as one edit – one insertion or one deletion. The second one includes the known true matches which differ from each other in just 2 edits and so on. The last class may include true matches which differ from each other in more than 4 edits. Also in construction of the last group we used full names in some records. Because the difference between two strings or names should be a reasonable number compared to their length. For example if two names of 5 characters differ in 4 characters from each other, it is obvious that no algorithm can detect if they were identical or not.

Parameters

As PEDA needs some settings, we defined a default value for each parameter in our implementation. And then we performed all our experiments with fixed parameter values in order to judge the results in the end. Because a little change in each parameter can affect on all results. Therefore they should remain the same during all experiments. The list of parameters is:

Psi (ψ): The maximum possible distance on Persian keyboard.

Theta (θ): The cost of inserting a letter after a letter to which it is equivalent.

Mu (μ): The cost of inserting a long vowel.
Omega (ω): The weight of phonetic similarity.

Lambda (λ): The weight of form similarity.

Sigma (σ): The weight of keyboard similarity.

We defined the default values of 12, 0.25, 0.15, 1, 1, and 1 for psi, theta, mu, omega, lambda, and sigma respectively. The psi is defined regard to Microsoft keyboard with its language set to Persian. Theta is a random chosen value smaller than 1 and bigger than 0. Mu is another randomly chosen value smaller than 1 and bigger than zero. But we defined theta bigger than mu because theta similarity is stronger than mu similarity. For our experiments we picked up the values of theta and mu the same as their corresponding values in AEDA to be able to compare our results with AEDA as well. We defined 1 for all three weights to evaluate the algorithm with a balance participation of all three similarity levels. These parameters can adjust to the real data in different and real executions.
Chapter 4: Findings and discussion

In this section we discuss the results of several series of comparison experiments performed on PEDA. The aim of these experiments is to see how effectively PEDA works and its accuracy.

Matching Results

To evaluate the algorithm through precision, recall and f-measure, there is a need to count true positives, false positives, true negatives, and false negatives in the result records. To be able to decide the status of outputs as TPs, FPs, TNs, and FNs, a threshold should be defined as well. Here, we defined the threshold as 80% for similarity in our experiments. This threshold divides the output into matches with equal or higher 80% similarity as positives and matches with similarity of lower than 80% as Negatives.

![Figure 18. The threshold divides the output matches into positives and negatives](image)

Defining a threshold is not an easy and specific task, as a little change in it may influence a lot in the accuracy of results. A higher value for threshold increases the loss of more true positives, although it reduces the false positives. On the other hand, a lower value for threshold gives more true positive matches but it also increases the number of false positives. Thus selection of a suitable threshold is challenging. In fact because there is no rule for defining the threshold, it should be adjusted by more runs on a specific data set. The selection of 80% as a threshold for the experiments in this thesis is based on just a guess not because of a specific rule. The results of the experiments described earlier, are as follows:
First series of experiments

Here, we have 10 sets of results each one included 100,000 records. The true matches and false matches were not known from before. Thus in order to count true positives and false positives in addition to true negatives and false negatives a human judgment should be applied to detect these issues and we do this. As in each execution the algorithm matches a watch list of 100 records against 1000 records of source list, and because we take the watch list randomly from the source list and applied some changes to their records, If the algorithm works with complete performance, all of these modified 100 records in watch list and their corresponding records in source list should be classified into positives. Therefore the ones with similarity equal to or larger than threshold are true positives and the ones with similarity smaller than threshold are false negatives. About false negatives, actually they should be counted in 100,000 numbers of result matches with similarity lower than of 80% - the defined threshold. But this is too much for a human to examine this number of matches in 10 times execution in order to count false negatives. Thus we used the watch list as the source for counting the false negatives. As each record of watch list with its corresponding record in source list which has similarity below 80% is considered as a false negative. The false positives are detected and counted by a human judgment among the matches with equal or higher similarity of 80%. And all of this process is repeated in 10 iterations. The counting is formulated as:

\[
\text{Positives} = \text{The number of matches with equal or higher than 80\% similarity} \\
\text{Negatives} = \text{The number of matches with lower than 80\% similarity} \\
100000 = \text{Total of positives and negatives} \\
\text{True positives} = \text{Number of records from watch list which have a similarity equal or above the threshold in their matching with their corresponding records in the source list} + \text{The matches with similarity equal to or above the threshold which is detected by the human judgment as true} \\
\text{False positives} = \text{Number of matches with similarity equal to or above the threshold which are detected as false matching by human judgment} \\
\text{True negatives} = \text{Number of matches with similarity lower than the threshold} - \text{false negatives} \\
\text{False negatives} = \text{Number of records from the watch list with their corresponding records in the source list which their matches have a similarity lower than the threshold} \\
\]

The following plots show the true positive percentages, false positive percentages on total positive matches in each execution. Also there are plots of true negative percentages and false negative percentages on total negative matches in each execution. These figures can show us how PEDA works at a glance. The lower the percentages of false positives and false negatives are, the more accurate the algorithm is.
A name matching algorithm for Persian language
In addition to using accurate measures for showing how PEDA works, we calculated the precisions and recalls for all ten executions as well. Then f-measures are calculated based on them as in Table 9 below:

<table>
<thead>
<tr>
<th>PEDA</th>
<th>Precision</th>
<th>Recall</th>
<th>f-measure</th>
</tr>
</thead>
<tbody>
<tr>
<td>DS01</td>
<td>0.8</td>
<td>0.9</td>
<td>0.85</td>
</tr>
<tr>
<td>DS02</td>
<td>0.77</td>
<td>0.95</td>
<td>0.85</td>
</tr>
<tr>
<td>DS03</td>
<td>0.89</td>
<td>0.95</td>
<td>0.92</td>
</tr>
<tr>
<td>DS04</td>
<td>0.81</td>
<td>0.97</td>
<td>0.88</td>
</tr>
<tr>
<td>DS05</td>
<td>0.73</td>
<td>0.96</td>
<td>0.83</td>
</tr>
<tr>
<td>DS06</td>
<td>0.79</td>
<td>0.95</td>
<td>0.86</td>
</tr>
<tr>
<td>DS07</td>
<td>0.79</td>
<td>0.99</td>
<td>0.88</td>
</tr>
<tr>
<td>DS08</td>
<td>0.79</td>
<td>0.96</td>
<td>0.87</td>
</tr>
<tr>
<td>DS09</td>
<td>0.75</td>
<td>0.95</td>
<td>0.84</td>
</tr>
<tr>
<td>DS10</td>
<td>0.7</td>
<td>0.95</td>
<td>0.81</td>
</tr>
<tr>
<td>f-measure mean</td>
<td></td>
<td></td>
<td>0.86</td>
</tr>
</tbody>
</table>

Table 9. The performance measures for the first series of experiments

Since f-measure is an accuracy measure between 0 and 1, the higher the values, the better and more accurate are the results. The mean of f-measures on all ten executions as can be seen in Table 9 which equals to 0.86, says that accuracy is almost high and reasonable to trust to the results. To have a better visualization on the performance measures and to be able to compare them side by side refer to Figure 15:
As we can see in figure 15 the values of precisions, recalls and f-measures are acceptable and near to each other for these series of experiments.

**Comparison with Levenshtein**

In order to determine how PEDA works effectively, we have done the first series of experiments with Levenshtein algorithm as well. We expect larger false positives with Levenshtein, as PEDA has changed Levenshtein in a way to detect more true positives.

<table>
<thead>
<tr>
<th>Levenshtein</th>
<th>DS01</th>
<th>DS02</th>
<th>DS03</th>
<th>DS04</th>
<th>DS05</th>
<th>DS06</th>
<th>DS07</th>
<th>DS08</th>
<th>DS09</th>
<th>DS10</th>
</tr>
</thead>
<tbody>
<tr>
<td>True positives</td>
<td>95%</td>
<td>100%</td>
<td>94%</td>
<td>99%</td>
<td>99%</td>
<td>96%</td>
<td>97%</td>
<td>96%</td>
<td>97%</td>
<td>98%</td>
</tr>
<tr>
<td>False positives</td>
<td>5%</td>
<td>0%</td>
<td>6%</td>
<td>1%</td>
<td>1%</td>
<td>4%</td>
<td>3%</td>
<td>4%</td>
<td>3%</td>
<td>2%</td>
</tr>
<tr>
<td>True negatives</td>
<td>59%</td>
<td>65%</td>
<td>65%</td>
<td>74%</td>
<td>67%</td>
<td>65%</td>
<td>69%</td>
<td>68%</td>
<td>62%</td>
<td>67%</td>
</tr>
<tr>
<td>False negatives</td>
<td>41%</td>
<td>35%</td>
<td>35%</td>
<td>26%</td>
<td>33%</td>
<td>35%</td>
<td>31%</td>
<td>32%</td>
<td>38%</td>
<td>33%</td>
</tr>
</tbody>
</table>

Table 10. Levenshtein results in first series of experiments

<table>
<thead>
<tr>
<th>PEDA</th>
<th>DS01</th>
<th>DS02</th>
<th>DS03</th>
<th>DS04</th>
<th>DS05</th>
<th>DS06</th>
<th>DS07</th>
<th>DS08</th>
<th>DS09</th>
<th>DS10</th>
</tr>
</thead>
<tbody>
<tr>
<td>True positives</td>
<td>86%</td>
<td>78%</td>
<td>90%</td>
<td>81%</td>
<td>73%</td>
<td>79%</td>
<td>79%</td>
<td>21%</td>
<td>76%</td>
<td>70%</td>
</tr>
<tr>
<td>False positives</td>
<td>14%</td>
<td>22%</td>
<td>10%</td>
<td>19%</td>
<td>27%</td>
<td>21%</td>
<td>21%</td>
<td>79%</td>
<td>24%</td>
<td>30%</td>
</tr>
<tr>
<td>True negatives</td>
<td>92%</td>
<td>91%</td>
<td>91%</td>
<td>96%</td>
<td>93%</td>
<td>92%</td>
<td>98%</td>
<td>94%</td>
<td>96%</td>
<td>93%</td>
</tr>
<tr>
<td>False negatives</td>
<td>8%</td>
<td>9%</td>
<td>9%</td>
<td>4%</td>
<td>7%</td>
<td>8%</td>
<td>2%</td>
<td>6%</td>
<td>4%</td>
<td>7%</td>
</tr>
</tbody>
</table>

Table 11. PEDA results in first series of experiments
As table 10 and table 11 show our expectation came true, and in Levenshtein the number of true matches which labeled as negatives is larger than the respected number in PEDA, in 10 iterations. Furthermore we expect that there will be an increase in true positives when false negatives get larger. This expectation came true as we can see in table 10 and table 11. The performance measures of execution Levenshtein are in table 12. Compare to table 9, we can see there is a strong decrease in recall, caused by increase in false negatives. Finally the average of f-measure in this table is 0.8 smaller than the correspondent average in PEDA with 0.86 value.

<table>
<thead>
<tr>
<th>Levenshtein</th>
<th>Precision</th>
<th>Recall</th>
<th>f-measure</th>
</tr>
</thead>
<tbody>
<tr>
<td>DS01</td>
<td>0.95</td>
<td>0.65</td>
<td>0.77</td>
</tr>
<tr>
<td>DS02</td>
<td>1</td>
<td>0.67</td>
<td>0.8</td>
</tr>
<tr>
<td>DS03</td>
<td>0.94</td>
<td>0.68</td>
<td>0.79</td>
</tr>
<tr>
<td>DS04</td>
<td>0.98</td>
<td>0.76</td>
<td>0.86</td>
</tr>
<tr>
<td>DS05</td>
<td>0.99</td>
<td>0.68</td>
<td>0.81</td>
</tr>
<tr>
<td>DS06</td>
<td>0.96</td>
<td>0.69</td>
<td>0.8</td>
</tr>
<tr>
<td>DS07</td>
<td>0.97</td>
<td>0.7</td>
<td>0.81</td>
</tr>
<tr>
<td>DS08</td>
<td>0.96</td>
<td>0.7</td>
<td>0.81</td>
</tr>
<tr>
<td>DS09</td>
<td>0.97</td>
<td>0.69</td>
<td>0.77</td>
</tr>
<tr>
<td>DS10</td>
<td>0.97</td>
<td>0.73</td>
<td>0.83</td>
</tr>
<tr>
<td>f-measure mean</td>
<td></td>
<td></td>
<td>0.8</td>
</tr>
</tbody>
</table>

**Table 12. The performance measures of Levenshtein in the first series of experiments**

**Second series of experiments**

In these series we used known true matches and false matches to see how well the algorithm performs matching. Here we also calculate all measures as in the first series of experiments. In table 10 we can see the measures calculated:

<table>
<thead>
<tr>
<th></th>
<th>Precision</th>
<th>Recall</th>
<th>f-measure</th>
</tr>
</thead>
<tbody>
<tr>
<td>DS01</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>DS02</td>
<td>0.64</td>
<td>0.78</td>
<td>0.7</td>
</tr>
<tr>
<td>DS03</td>
<td>0.83</td>
<td>0.55</td>
<td>0.66</td>
</tr>
<tr>
<td>DS04</td>
<td>0.75</td>
<td>0.43</td>
<td>0.55</td>
</tr>
<tr>
<td>DS05</td>
<td>0.71</td>
<td>0.83</td>
<td>0.76</td>
</tr>
<tr>
<td>DS06</td>
<td>0.86</td>
<td>1</td>
<td>0.92</td>
</tr>
<tr>
<td>DS07</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>DS08</td>
<td>1</td>
<td>0.75</td>
<td>0.86</td>
</tr>
<tr>
<td>DS09</td>
<td>1</td>
<td>0.71</td>
<td>0.83</td>
</tr>
<tr>
<td>DS10</td>
<td>0.71</td>
<td>0.83</td>
<td>0.76</td>
</tr>
<tr>
<td>f-measure mean</td>
<td></td>
<td></td>
<td>0.80</td>
</tr>
</tbody>
</table>

**Table 13. The performance measures for the second series of experiments**
As can be seen in table 10 the f-measure mean has a high and acceptable value that shows that accuracy and performance is acceptable.

In figure 16 of performance measures in these series, we can see the result values in a more comparable way:

![Performance measures](image)

**Figure 20. The bar chart of performance measures for the second series of experiments**

As the chart shows, in these series we do not have the measures values close to each other in different executions, just as in the previous series. In some executions results are perfect, while in some others they are rather low. Although we tried to cover all different name variations in the selection of known true matches, the randomly division of them into 10 data sets placed the different kinds of name variations unevenly among these ten data sets. And because PEDA works different on various types of name variations, this inequality has happened to the output matches in different executions. Also because the numbers of records in these ten data sets are less than the numbers of records in data sets of previous series, this heterogeneity in name variations influence more on getting the heterogeneous results. But the average is reasonable and well.

In order to see and compare the results between these two series, the following charts are used to give visualization on the performance measures in these two series:
A name matching algorithm for Persian language

**Precisions**

![Precision Graph]

**Recalls**

![Recall Graph]

**f-measures**

![f-measure Graph]
Third series of experiments

Here, we wanted to know how PEDA works for different number of edits. Thus we counted the true positives and false negatives in the results of matching which were different for one edit, two edits, three edits and four or more edits, separately. It should be mentioned that chose a threshold of 80% in all experiments to make them comparable with each other. If PEDA works well for all these categories, the results of them should have high true positives and low false positives. Table 11 shows the results:

<table>
<thead>
<tr>
<th></th>
<th>1 edit</th>
<th>2 edits</th>
<th>3 edits</th>
<th>4 + edits</th>
</tr>
</thead>
<tbody>
<tr>
<td>True positives</td>
<td>99%</td>
<td>81%</td>
<td>69%</td>
<td>42%</td>
</tr>
<tr>
<td>False positives</td>
<td>1%</td>
<td>19%</td>
<td>31%</td>
<td>58%</td>
</tr>
</tbody>
</table>

Table 14. The true positives and false positives for four executions

As can be seen in table 11, PEDA works well especially for 1 edit, then for 2 edits, and approximately for 3 edits, but for 4 edits and more it does not work as well as for a lower number of edits.

During the experiments, we tried to include all kinds of name variations. We observed that PEDA works well and efficiently for some type of name variations and does not work enough well for the others. We gathered some of the worth results in table 12:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Similarity</th>
</tr>
</thead>
<tbody>
<tr>
<td>ح ح</td>
<td>پر</td>
<td>45%</td>
</tr>
<tr>
<td>زر</td>
<td>زهرا</td>
<td>68%</td>
</tr>
<tr>
<td>س</td>
<td>احمد</td>
<td>70%</td>
</tr>
<tr>
<td>نسب</td>
<td>نسب</td>
<td>69%</td>
</tr>
<tr>
<td>اقا</td>
<td>شهزاد</td>
<td>64%</td>
</tr>
<tr>
<td>مهوجا</td>
<td>مهوجا</td>
<td>76%</td>
</tr>
<tr>
<td>زینب</td>
<td>نه خانم</td>
<td>68%</td>
</tr>
<tr>
<td>فاطمه</td>
<td>فاطمی</td>
<td>64%</td>
</tr>
</tbody>
</table>

Table 15. Some examples PEDA works not enough efficient on them

We noticed that PEDA does not work well for missing or extra elements, name equivalence, fielding variations, punctuations, short forms, and translation. It also has problems with the matching of names which their starts are different from each other. But it works well for spelling variation and segmentation.
Chapter 5: Conclusion and future work

Conclusion

As mentioned earlier there are different name variations, all algorithms developed up to now focused on some of them but not all. Because they are different from each other and in some cases need totally different solutions. Thus one algorithm cannot be a suitable solution to deal with all different name variations. This is also true for PEDA. PEDA is based on Levenshtein that is an edit distance type of algorithm and these types of algorithms are better for spelling variations and they do not work for extra elements, punctuations, field variations or, initials. As in the result data set this issue is obvious. So in a real environment PEDA should be used with other kinds of algorithms, the ones focus on the other types of name variations to make us able to get high and more accurate results. For example, we should apply a preprocessor to remove for example the punctuations from the names. Then apply an algorithm which deals with extra words in names and initials, followed by PEDA to work on spelling variations. Levenshtein is a time consuming algorithm and thus PEDA is slow on large data sets. So it may not be suitable for problems where time is crucial if they include a large number of names. On the other hand because PEDA is parametric and similarity in different levels feed to the algorithm, it is flexible and adjustable to the target environment.

Future Work

As it is the first time that a project of this kind has been done for the Persian language, obviously there are issues which can be alleviated in order to make the algorithm more efficient. They are included:

- In this project, at each time one letter is read from the source word and another letter is read from the target word, and then compared to each other. Thus the rules are defined among each two Persian letters. But it could have used a grammar or variation rules which connected a collection of letters, one, two or more to another collection of letters, one, two or more. This project does not have this capability to manage these kinds of rules for now. It can be considered in the future in order to cover more rules. More rules has an advantage which improve the accuracy in the output, although it has a negative influence on running time.
- In this project, the algorithm can have access to two letters of a word at each time, the current and the previous one, while there are rules in Persian language which needs to have access to the next letters of the current letter. This capability can be added in the future to the program in order to make it adhere to more rules.
• In our work, similarity rules are defined in the first step as a collection of fixed rules which feed to the algorithm at running time and all of them are static during the execution. To enhance this name matching algorithm, PEDA, and make more accurate rules, a method of data mining like neural networks can be to make the rules dynamic, update and adopt them to real data based on the occurrences or discovery of name variations in the data.

• The rules are defined according to the origin shape of letters. But there are rules which exist just among specific position-related letter forms. One can add a text render to the algorithm to detect the position of letters inside the words and give the position-related letter forms as input to PEDA to works on more accurate rules based on position forms of letters.

• Study on weighted average formula which is used for calculating replacement cost, substitution it with other possible formulas and examine their differences, their advantages and disadvantages.
Bibliography


Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extra-ordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

Copyright

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© Leila Momeninasab